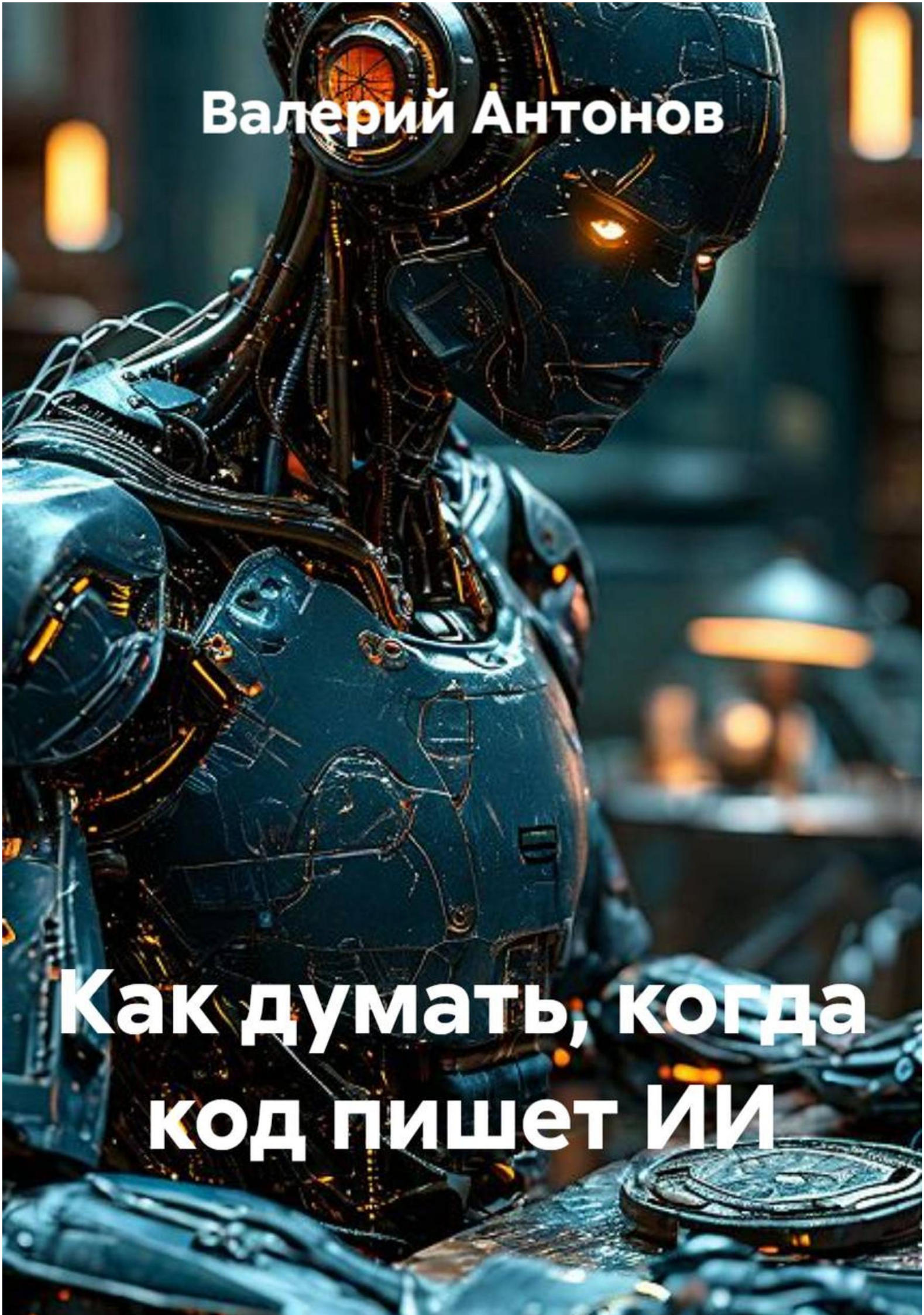
Валерий Антонов
Как думать, когда код пишет ИИ

Валерий Антонов

Как думать, когда код пишет ИИ

«Автор»
2026

Антонов В.

Как думать, когда код пишет ИИ / В. Антонов — «Автор», 2026

Строка кода перестала быть валютой. ИИ генерирует микросервисы быстрее, чем вы набираете запрос, и вместе с этим рухнула идентичность разработчика. Если ценность больше не в синтаксисе — то в чём?Эта книга — не учебник по промпт-инжинирингу, а руководство по переизобретению себя. Вы пройдёте путь от скепсиса к новой роли — Архитектор Намерений: человек, который не пишет код, а проектирует смыслы, проверяет контракты и дирижирует ансамблем ИИ-агентов. Внутри — боевые техники: от двухфазной генерации до мульти-агентного моделирования. В финале — Кодекс разработчика эпохи LLM: десять принципов для тех, кто хочет не просто сохранить работу, а стать незаменимым.

Содержание

Введение 5
1.2. От машины-исполнителя к машине-соавтору: краткая история отношений программиста и инструмента 9
1.3. Экономика кода завтрашнего дня: почему за синтаксис больше не платят 12
1.4. Деконструкция страха: что на самом деле скрывается за «ИИ меня уволит» 15
2.1. Новый центр тяжести: смещение от «Как написать» к «Что заставить сделать». 18
2.2. Определение новой должности: кто такой разработчик промптов на уровне Senior и Staff 21
2.3. Когнитивный разворот: мышление не циклами и условиями, а сущностями и контрактами 24
2.4. Почему «грязный код» прототипа теперь ценнее «чистого кода» бездумного 27
Часть II. Карта новых компетенций: чему учиться, когда учить синтаксис бесполезно 29
Глава 3. Искусство спецификации: промпт-инжиниринг как системная дисциплина 29
3.1. Анатомия идеального промпта: не просьба, а техзадание 29
Конец ознакомительного фрагмента. 34

Как думать, когда код пишет ИИ

Введение

Я пишу это введение в 2026 году, когда строка кода перестала быть валютой.

Ещё три года назад разработчик измерял свою ценность написанными строками, закрытыми тикетами, количеством коммитов. Мы называли себя инженерами, но в глубине души оставались ремесленниками, которые гордятся мозолями на пальцах и объёмом написанного кода. Мы спорили о стилях форматирования, о паттернах проектирования, о том, как правильно называть переменные. Мы читали код великих, чтобы научиться писать так же. Мы коллекционировали сниппеты в памяти и вызывали их в нужный момент, как заклинания. Код был нашим ремеслом, нашей идентичностью, нашей гордостью.

Эта эпоха закончилась.

Сегодня строка кода стоит дешевле, чем когда-либо в истории. Вы можете попросить модель написать микросервис, и она напишет его быстрее, чем вы наберёте запрос в поисковике. Вы можете попросить её переписать модуль на другом языке, и она сделает это за секунды. Вы можете попросить её найти баг, и она проанализирует тысячи строк кода и выдаст гипотезу. Код перестал быть дефицитным ресурсом. Он стал изобильным. А когда что-то становится изобильным, оно перестаёт быть источником ценности.

Это книга не о том, как писать промпты.

Книг о промпт-инжиниринге уже десятки. Они рассказывают, как формулировать запросы, как использовать few-shot learning, как управлять контекстным окном. Это важные знания, и часть этой книги тоже о них. Но это не главное. Главное происходит не в промпте. Главное происходит в голове разработчика, который смотрит на пустой экран и понимает: «Я больше не знаю, в чём моя работа».

Потому что кризис, вызванный появлением больших языковых моделей, — это не кризис безработицы. Рынок разработки не рухнул, вакансии не исчезли, зарплаты не обнулились. Кризис — внутри. Это кризис идентичности. Если я не пишу код, то кто я? Если моя ценность больше не в знании синтаксиса, то в чём она? Если машина решает задачи быстрее меня, то зачем я?

Эта книга — ответ на эти вопросы.

Я предлагаю вам переосмыслить себя не как «кодера», а как Архитектора Намерений. Человека, чья работа — не писать код, а проектировать смыслы. Не реализовывать алгоритмы, а описывать желаемое поведение системы и проверять, что оно достигнуто. Не знать синтаксис наизусть, а понимать, какую проблему мы решаем и почему именно так. Не командовать машиной, а дирижировать ансамблем из специализированных ИИ-агентов, каждый из которых гениален в своей узкой области.

Это звучит как фантазия. Но это реальность, в которой я живу и работаю последние несколько лет. Я прошёл через все пять стадий принятия: от отрицания («это игрушка, она не сможет написать мой код») до принятия («я — мыслитель, вооружённый машиной, а не оператор клавиатуры»). Я набил шишки, совершил ошибки, которые стоили мне и моим проектам времени и денег. И я написал эту книгу, чтобы вы прошли этот путь быстрее и с меньшими потерями.

Книга построена как путешествие через четыре территории.

Первая часть — прощание с эпохой человеческого кода. Мы посмотрим в лицо страху, поймём, что именно умирает в нашей профессии, а что — только рождается. Мы разберём экономику кода завтрашнего дня и поймём, за что платят деньги теперь.

Вторая часть — карта новых компетенций. Это практический инструментарий: как формулировать спецификации с математической точностью, как управлять вниманием модели через контекстное окно, как декомпозировать систему на кванты смысла, как тренировать вкус, чтобы отличать элегантное решение от синтаксически верного мусора, как находить ошибки, которые вы не могли бы придумать сами.

Третья часть — новая методология разработки. Как устроена отладка, когда баг может быть не в коде, а в вашей ментальной модели. Почему комментарий в коде — это надгробие над умершей мыслью, и как сделать так, чтобы знание жило в спецификации. Как проектировать агентов с характером и распределять между ними зоны автономности. Как управлять ансамблем из нескольких моделей, которые спорят друг с другом, а вы разрешаете их конфликты.

Четвёртая часть — самое важное. О том, что не может быть автоматизировано. Эмпатия к пользователю, которая не вычисляется. Способность усомниться в рамке задачи и спросить: «А ту ли проблему мы решаем?» Этический компас, который не может быть сведён к оптимизации целевой функции. Интуиция, которая чувствует неправильность архитектуры до того, как это можно доказать. Озарение, которое прыгает через логическую пропасть, недоступную градиентному спуску. И ответственность — то, что отличает автора от оператора.

В конце книги я поместил Кодекс разработчика эпохи LLM — десять принципов, которые можно распечатать и повесить над монитором. Это не догма, а компас. Когда вас накроет волной новых инструментов, новых моделей, новых метрик — сверяйтесь с ним. Он напомнит вам, кто вы есть на самом деле.

Эта книга — не плач по уходящей эпохе и не ода технологическому прогрессу. Это полевое руководство по выживанию и процветанию в мире, где код пишет ИИ. Она для тех, кто чувствует тревогу, глядя на пустой редактор, и не знает, что теперь делать. Для тех, кто уже использует LLM, но подозревает, что использует лишь малую часть возможностей. Для тех, кто хочет не просто сохранить работу, а стать лучшим в новой реальности.

Ремесло наборщика текста умерло. Родилась профессия Архитектора Намерений.

Добро пожаловать в неё.

Часть I. Конец эпохи «человеческого кода».

Глава 1. Прощай, ремесленник: почему строка кода перестала быть ценностью.

1.1. Синдром «чистого листа»: почему ИИ побеждает нас на стадии boilerplate-а

Аннотация (Полная версия):

Это вскрытие главной психологической травмы разработчика — страха пустого редактора, который мы привыкли героизировать. Десятилетиями нас учили, что «начать сложнее всего», а умение быстро набивать import, class и main — признак профессионализма. Мы исследуем фундаментальный сдвиг кривой продуктивности. Раньше она была логарифмической: первые 80% типового кода писались легко, создавая иллюзию прогресса, а оставшиеся 20% сложной бизнес-логики требовали 80% времени. LLM обнулила первый этап. Порог входа в «первые 80%» исчез, оставив разработчика один на один с самой сложной частью сразу. Это вызывает не облегчение, а экзистенциальный ужас: «Если я не пишу код, что я вообще делаю?». Глава отвечает на этот вопрос.

Мы начнем с яркого автобиографического sketches (зарисовки) типового утра разработчика «до» и «после». «До»: ритуал наливания кофе, ощущение важности первых строк, медитативная настройка. «После»: палец зависает над клавиатурой, нужно не писать, а формулировать. Возникает тревога «чистого листа промпта» — парадоксально, она оказывается сильнее страха чистого листа кода.

1. Экономика когнитивной нагрузки и сломанный цикл дофамина.

Здесь мы вводим понятие «Когнитивного дофаминового долга».

Раньше: Написание boilerplate-кода было легальным способом «разогнаться». Выполняя простые, монотонные действия (создать файл, написать каркас класса, подключить библиотеки), разработчик получал микро-дозы дофамина за «завершенные дела». Мозг входил в рабочий ритм. Это была смазка для перехода к сложным задачам.

Теперь: ИИ съедает эту «разминку». Разработчик сразу сталкивается со сложной задачей декомпозиции и верификации без предварительного разогрева. Мозг, не получив привычного дофаминового аванса, саботирует работу. Мы разберем, как обмануть мозг, создавая новые ритуалы «быстрых побед» через микропромптинг и визуальное моделирование вместо ручного набора текста.

2. Эксперимент: «Скорость создания CRUD-контроллера» (Практическая часть).

Мы проведем читателя через иммерсивный A/B-тест, который он может повторить сам.

Сценарий: Создать REST API эндпоинт для управления сущностью «Продукт» с валидацией, маппингом DTO, пагинацией и базовой аутентификацией.

Фаза А (Классический подход): Засекаем время. Пишем ручками: модель, репозиторий, сервис, контроллер. По пути гуглим забытые аннотации валидации, настраиваем маппер. Результат: ~40 минут работы, чувство «честной усталости», код на 150-200 строк.

Фаза В (ИИ-подход): Пишем промпт-спецификацию (5 минут). Получаем готовый код (15 секунд). Тратим следующие 35 минут на: чтение, понимание, правку нейминга под стандарты команды, дописывание кастомной логики в пагинации, написание тестов (которые тоже сгенерировал ИИ, но мы проверяем их честность).

Анализ результатов: Время сопоставимо. Но в Фазе А мы были «кодирующими машинами». В Фазе В — архитекторами и контролерами качества. Мы задаем читателю вопрос: «Какая усталость ценнее для вашего роста? Усталость в пальцах или усталость в принимаемых решениях?». Сравнительная таблица «Затраченная энергия» покажет, что ИИ не экономит время на сложных задачах (пока), а перераспределяет его с механики на мышление.

3. Таблица: «Типовые задачи, которые вы *никогда* больше не должны писать руками».

Это будет провокационный, но методичный список, разбитый по доменам. Смысл в том, чтобы прекратить расценивать написание этого кода как «работу».

Бэкенд: Контроллеры и роутинг, DTO и маппинги (ModelMapper/MapStruct), валидация входных данных, генерация схем баз данных и миграций, стандартные реализации JPA-репозиториев, документация OpenAPI/Swagger.

Фронтенд: Верстка типовых форм и списков (особенно с Tailwind/Bootstrap), стейт-менеджмент (Redux boilerplate), хуки для fetch-запросов с обработкой loading/error, юнит-тесты с моками для тривиальных функций.

DevOps: Написание Dockerfile'ов для стандартных рантаймов, GitHub Actions для CI/CD (простые пайплайны), Helm-чарты по умолчанию.

Принцип: Если ваш запрос в Google по задаче всегда заканчивался на статью с Baeldung, Medium или StackOverflow, то эту задачу уже сейчас должен делать ИИ.

4. Чек-лист: «Занимаюсь ли я механическим кодированием вместо мышления?»

Глава завершится инструментом для самоаудита прямо в процессе работы. Читателю предлагается повесить этот чек-лист на монитор. Если во время работы он ловит себя на 3 и более пунктах, он должен остановиться и передать задачу модели.

Вы копируете код с одного экрана на другой или из прошлого проекта? *Симптом: вы — компилятор.*

Вы пишете структуру, которая повторяет структуру из соседнего файла? *Симптом: вы — генератор бойлерплейта.*

Вы вручную преобразуете один формат данных в другой (Entity -> DTO -> JSON)? *Симптом: вы — транслятор.*

Ваша текущая мысль — не «как это должно работать», а «как пишется синтаксис этой аннотации»? *Симптом: вы — словарь, а не инженер.*

Вы откладываете основную задачу, потому что «нужно сначала всё красиво настроить»? *Симптом: прокрастинация через псевдо-работу.*

Итоговый вывод подраздела: Синдром чистого листа излечивается не смирением перед пустотой, а переводом взгляда с редактора кода на редактор требований. Ваше ремесло больше не в том, чтобы заполнить пустоту символами, а в том, чтобы править ею.

1.2. От машины-исполнителя к машине-соавтору: краткая история отношений программиста и инструмента

Аннотация (Полная версия):

Это не просто исторический экскурс, а терапия принятия неизбежного через понимание цикличности прогресса. Мы совершим путешествие от перфокарт и тумблеров до GPT-4, но представим его не как технологический прогресс, а как эволюцию *языков абстракции* и смену ролей человека. Ассемблер, C, Python — каждый шаг отдалял нас от «железа» и уничтожал целый класс специалистов, чтобы породить новый, более высокооплачиваемый. LLM — это не просто очередной язык сверхвысокого уровня. Это первый инструмент, который оперирует не синтаксисом, а *интенцией*. Это шаг от «машины-инструмента» (молоток) к «машине-коллеге» (подмастерье), что требует не просто освоения нового синтаксиса, а смены модели поведения с «командира» на «наставника». Глава доказывает: мы уже проходили этот ужас и восторг трижды за последние 70 лет.

Мы откроем главу сильной метафорой: «Программист — это не тот, кто говорит с машиной. Это тот, кто говорит с абстракцией. Просто раньше абстракцией был язык, а теперь — сама логика рассуждения».

1. Историческая цепь «убитых» профессий и рожденных гениев.

Мы проследим три тектонических сдвига, каждый из которых сопровождался криком «Программирование умерло!».

Эпоха 1: Конец «Властелинов Железа» (1950-е).

Кто исчез: Специалисты по коммутации штекеров и наборщики машинных кодов в восьмеричной системе. Работа требовала уникальной памяти и «чувства железа».

Катализатор: Появление Ассемблера и первых компиляторов (Fortran, COBOL). Сначала их высмеивали: «Настоящие программисты не используют мнемоники, они знают коды операций процессора наизусть!».

Кто родился: Инженер-программист. Тот, кто мыслит структурами данных и алгоритмами, а не электрическими сигналами. Ценность сместилась с «как накормить машину перфокартами» на «как решить дифференциальное уравнение с помощью кода».

Параллель с сегодня: «Настоящие программисты не используют подсказки ИИ, они помнят все методы стандартной библиотеки!». Звучит знакомо?

Эпоха 2: Конец «Оптимизаторов памяти» (1990-е – 2000-е).

Кто исчез: Специалисты по ручному управлению памятью (malloc/free), гуру побитовых сдвигов и оптимизаций на уровне ассемблерных вставок.

Катализатор: Массовое внедрение языков с автоматическим управлением памятью (Java, C#, Python). Старожилы были в ярости: «Этот мусорщик непредсказуем! Вы не контролируете железо! Эти ваши виртуальные машины жрут всю память!».

Кто родился: Архитектор программных систем. Разработчик, который перестал думать о выравнивании байтов и начал думать о паттернах проектирования, масштабировании и

чистых интерфейсах. Стоимость железа упала, стоимость сложности взлетела. Освободившуюся когнитивную мощность направили на борьбу со сложностью, а не с утечками памяти.

Параллель с сегодня: «ИИ генерирует нечитаемый, неоптимальный код! Он использует лишние аллокации!». Мы перестаем думать о том, *как именно* выделилась память, и начинаем думать о том, *почему* мы вообще выделяем эту сущность в домене.

Эпоха 3: Конец «Энциклопедистов API» (2020-е и далее).

Кто исчезает: Разработчик, чья основная ценность — знание наизусть сигнатур методов, флагов конфигурации, нюансов конкретного фреймворка или API библиотеки. «Живая документация».

Катализатор: LLM, обученные на всём корпусе публичного кода. Модель знает любой фреймворк лучше любого человека.

Кто рождается: Инженер по спецификациям и верификации (то, о чём вся книга). Человек, который не спрашивает «Какой параметр у этой функции?», а задает вопрос: «Какой контракт эта функция должна выполнять, и как мы проверим, что она его не нарушает?».

2. Смена модели поведения: от командира к наставнику.

Здесь мы вводим ключевую таксономию отношений, которую будем использовать на протяжении всей книги.

Модель «Командир-Солдат» (Эпоха детерминизма): Командир (человек) отдает приказы на языке, который солдат (машина) понимает буквально. Если солдат сделал не то — виноват командир, неправильно сформулировал приказ. Ошибка — это всегда ошибка в синтаксисе или алгоритме. Машина — функция.

Модель «Мастер-Подмастерье» (Эпоха LLM): Мастер (человек) описывает желаемый результат (артефакт), а Подмастерье (ИИ) изготавливает его, используя свои обширные, но бессистемные знания. Подмастерье может напутать, проявить «творчество» не там, где надо, или буквально понять метафору. Мастер должен: 1) Четко описать артефакт (Техзадание). 2) Проверить работу. 3) Объяснить, что не так, чтобы Подмастерье переделал. Машина — недетерминированный коллега.

Мы нарисуем таблицу сравнения этих двух моделей по критериям: «Источник ошибки», «Стиль коммуникации», «Единица работы», «Главный навык человека».

3. Анатомия умирающих «священных коров».

Провокационный и отрезвляющий список навыков, которые переходят из разряда «Капитал разработчика» в разряд «Устаревшее ремесло».

Корова №1: Знание API наизусть. Раньше разработчик, знающий 200 методов jQuery или Spring Boot, стоил дороже. Теперь это бесполезно. ИИ вспомнит любой метод за вас. Ценность переходит в понимание, *какой класс задач решает* это API, и в умение выбрать между библиотеками, а не запомнить их.

Корова №2: Синтаксический перфекционизм. Споры о «правильном» форматировании, скобках на новой строке, именовании переменных уходят в прошлое. Это работа для линтера и ИИ-форматтера. Попытка человека контролировать это вручную — аналог ручной полировки каждого болта в двигателе, когда есть заводской робот.

Корова №3: Умение писать «с нуля» без гугла. Этот навык был артефактом эпохи bad internet и собеседований у белой доски. Теперь он вреден. Умение писать, не опираясь на коллективный разум (будь то Google или LLM), означает писать медленно и изолированно. Новый навык — скорость навигации в сгенерированном и умение задать правильный вопрос.

Корова №4: Искусство «хорошего комментария». Комментарий, объясняющий *как* работает код, умрет (код должен читаться). Комментарий, объясняющий *почему*, переедет в промпт и будет привязан к сгенерированному блоку как его неотъемлемая мета-шапка.

Итоговый вывод подраздела: Каждый виток эволюции инструментов «убивал» программиста как оператора и возрождал его как мыслителя. Отказ признать в LLM соавтора, а не просто «умную автодополнялку» — это попытка остаться высокооплачиваемым наборщиком текста в мире, где печатный станок уже изобрели. История не прощает такой слепоты дважды.

1.3. Экономика кода завтрашнего дня: почему за синтаксис больше не платят

Аннотация (Полная версия):

Модель ценообразования в индустрии переворачивается с ног на голову. На наших глазах происходит то же, что случилось с фотографией при переходе от плёнки к цифре: «сделать снимок» перестало быть актом мастерства и затрат. Если код (как текст) стремится к нулевой предельной стоимости, то куда мигрирует ценность? Она уходит в две сферы: *дизайн системы* (почему мы делаем именно это, а не другое) и *верификация* (как мы узнаем, что сгенерированное работает правильно). Мы разберём понятие «токенизируемой» и «нетокенизируемой» ценности. Токенизируемое (то, что можно перевести в последовательность символов промпта) — дешевеет. Нетокенизируемое (неявное знание, вкус, понимание контекста бизнеса, личная ответственность) — резко дорожает. Эта глава — компас для навигации в новом рынке труда.

Мы начнём главу с убийственной аналогии из другой индустрии, чтобы снять накал и включить холодный анализ.

1. Урок Kodak: когда создание ценности становится бесплатным.

В 1990-х Kodak контролировал рынок. Ценность была в химическом процессе проявки, в умении сделать 36 хороших кадров. Цифровая фотография сделала сам акт съёмки бесплатным. Нажатие на кнопку обесценилось.

Кто умер: «Чистые фотографы», чья ценность была в умении правильно выставить экспозицию без участия автоматики.

Кто разбогател: Визуальные storyteller'ы, инфлюенсеры, дизайнеры смыслов. А также создатели инструментов обработки (Photoshop, Instagram) — те, кто помогает *отбирать, улучшать и распространять*.

Параллель: «Чистый кодер», чья ценность в знании синтаксиса и скорости набора, — это фотограф, гордящийся умением вручную настраивать диафрагму. Это всё ещё искусство, но за него больше не платят премию. Платят за историю (продукт) и за композицию (архитектуру).

2. Модель «3D-ценности» разработчика.

Мы вводим замену устаревшей плоской шкале «Junior/Middle/Senior» по количеству технологий. Новая система координат трёхмерна:

Ось X: Глубина (Domain Expertise).

Что это: Не просто «знаю Java», а «понимаю бухгалтерский учёт», «разбираюсь в логистике последней мили», «чувствую регуляторный контекст финтеха».

Почему дорожает: ИИ не может провести 50 интервью с бухгалтерами, чтобы понять их невысказанную боль. Он не чувствует, что «вот этот налог на добавленную стоимость рассчитывается не по тому документу, хотя в ТЗ написано иначе». Глубина — это способность заметить, что промпт, спущенный бизнесом, — чушь, которая сломает реальный процесс.

Метрика: Время от входа в домен до первого найденного логического противоречия в требованиях.

Ось Y: Ширина (System Integration & Lateral Vision).

Что это: Понимание, как система встраивается в ландшафт. Что аутентификация — это не просто «залогинился», а цепочка: Keycloak -> API Gateway -> Audit Log -> Billing.

Почему дорожает: LLM мыслит в рамках контекстного окна. Даже с RAG она склонна к локальной оптимизации. Только человек видит, что «ускорение этого микросервиса вызовет каскадный сбой в очереди сообщений».

Метрика: Количество системных инцидентов, предотвращённых на стадии дизайна, а не отловленных в проде.

Ось Z: Четкость (Precision in Ambiguity).

Что это: Умение формализовать «нечёткие хотелки». Способность взять расплывчатый запрос бизнеса («Сделайте нам красивый дашборд, чтобы мы видели эффективность») и превратить его в исчерпывающую спецификацию, которую ИИ не сможет интерпретировать двусмысленно.

Почему дорожает: Это навык перевода с «человеческого неструктурированного» на «формально-человеческий». Это главный интерфейсный навык будущего. Мусор на входе в промпт — мусор на выходе из модели. Чёткость — это фильтр, который стоит миллионы.

Метрика: Количество итераций на доработку сгенерированного кода до его приемки.

3. Анализ рынка: кто умрёт, кто родится, и где деньги.

Мы дадим отрезвляющую таблицу вакансий, избегая хайпа, но опираясь на ранние сигналы с рынка (Job Postings, аналитика венчурных стартапов).

Исчезающие или стагнирующие роли (3-5 лет):

«Верстальщик форм» / HTML/CSS Coder (без UX-мышления). Причина: Figma-to-Code плагины и V0 от Vercel.

«CRUD-разработчик» (пишущий стандартные REST API). Причина: генерация конечных точек по схеме данных.

«Копирайтер-технарь» для написания стандартной документации. Причина: авто-документирование кода.

Ручной тестировщик регрессии без навыков авто-тестинга. Причина: ИИ-агенты для обхода UI.

Растущие и высокооплачиваемые роли:

Инженер по спецификациям (Specification Engineer). Человек, который пишет не промпты, а формальные, верифицируемые ТЗ. Это элитный аналитик.

AI Code Auditor / Verifier. Специалист по надзору за ИИ-кодом. Его работа — не найти баг, а найти место, где ИИ прошел мимо контракта. Работа с property-based тестами и формальными методами.

AI Toolsmith. Разработчик DSL и «обёрток» для агентов внутри компании. Создатель внутренних инструментов для других разработчиков.

Responsible AI Deployer. Специалист по «последней миле» кода: проверка на compliance, безопасность, отсутствие предвзятости.

4. График: «Зависимость зарплаты от процента рукописного кода в проекте» (Обратная корреляция).

Мы представим провокационную иллюстрацию, которая заставит читателя замереть.

Ось X: Процент написанного вручную (от руки) production-кода в проекте.

Ось Y: Ценность разработчика на рынке труда (ЗП + востребованность).

Точка A (90% ручного кода): «Я всё пишу сам на Vanilla JS». Зарплата — нижний квартиль рынка. Продуктивность низкая, оверхеды огромные.

Точка B (50% ручного кода): Тревожная зона. Разработчик автоматизирует часть, но не доверяет ИИ, постоянно переписывая за ним. Стоимость растет, но медленно.

Точка C (5% ручного кода): Пик ценности. Это ключевые модули: ядро системы, уникальные алгоритмы, критичные компоненты безопасности и те самые интеграционные склейки. Всё остальное сгенерировано и верифицировано под его руководством.

Точка D (0% ручного кода): Опасное падение (менеджер, только пишущий промпты и не способный проревьюить результат). Ценность снова стремится вниз, так как отсутствует навык контроля качества.

Итоговый вывод подраздела: Если ваш доход зависит от того, сколько строк кода вы произвели, вы — фабрика, которую цифровая эпоха заменит роботом. Если ваш доход зависит от того, какие решения вы приняли, чтобы эти строки не пришлось переписывать трижды, и какой риск вы предотвратили, — вы неуязвимы. Экономика безжалостна: она платит не за процесс, а за невоспроизводимый результат.

1.4. Деконструкция страха: что на самом деле скрывается за «ИИ меня уволит»

Аннотация (Полная версия):

Это сеанс психоаналитики профессионального сообщества, приглашение заглянуть в собственное нутро. Массовая истерия вокруг «ИИ отнимет работу» — это, как правило, не рациональный анализ рынка труда, а проекция внутреннего конфликта. Страх увольнения редко связан с реальной угрозой голодной смерти. Чаще это ужас перед потерей идентичности «крутого кодера», на которую ушли годы жизни, или кошмар разоблачения «самозванца» (Impostor Syndrome), который годами выезжал на знании синтаксиса и StackOverflow, а не на инженерном мышлении. ИИ не увольняет человека — он срывает с него маску, показывая, чем он является на самом деле: механическим транслятором требований в синтаксис или творцом, решающим проблемы. Глава предлагает пройти через пять стадий принятия неизбежного по модели Кюблер-Росс, чтобы выйти из них не выжившим, а усилившимся.

Мы начнём с провокации: «Если вы боитесь, что вас заменит машина — возможно, вы и были машиной». Это задаст тон беспощадной, но освобождающей честности.

1. Пять стадий принятия ИИ по Кюблер-Росс (Разбор сообщества).

Мы пройдём по каждой стадии, показывая её типичные проявления в профессиональных спорах, и дадим «диагноз», какая часть личности страдает на этом этапе.

Стадия 1: Отрицание. «Он пишет чушь / Он не сможет написать МОЙ код».

Типичное высказывание: «Я попробовал ChatGPT — он сгенерировал ерунду с несуществующим API. Это игрушка для студентов, мой проект слишком сложен для него».

Что на самом деле происходит: Защитный механизм. Разработчик оценивает модель по её текущему срезу (сегодняшнему уровню), отказываясь экстраполировать кривую роста. Это как говорить в 1905 году: «Автомобиль — это игрушка для богатых, он сломался через 5 миль, лошадь надёжнее».

Теневая сторона: Страх утраты монополии на знание. «Если эту сложную штуку можно сгенерировать, то чем я лучше других?»

Стадия 2: Гнев. «Это воровство! Они просто скопипастили код с GitHub!».

Типичное высказывание: «Это не интеллект, а плагиат! Модель обучили на моём опенсорсе и теперь продают, а я должен остаться без работы?».

Что на самом деле происходит: Легитимная этическая и юридическая проблема используется как дымовая завеса для экзистенциальной ярости. Человек учился точно так же — читал чужой код, копировал, адаптировал. Гнев направлен на скорость и бесстыдство процесса. Мы разберём, что право на обучение на публичных данных — это спор юристов, а вот наша реакция на это — предмет самоанализа.

Теневая сторона: Чувство несправедливости не от того, что у человека что-то отняли, а от того, что его сверхспособность (помнить и комбинировать паттерны) обесценили так быстро и нагло.

Стадия 3: Торг. «Я буду использовать, но только как автодополнение / умный гугл».

Типичное высказывание: «Copilot — это удобно, он как продвинутый автокомплит. Но архитектуру и главное я всегда пишу сам. Я контролирую процесс».

Что на самом деле происходит: Самая коварная и залипательная стадия. Разработчик признаёт инструмент, но низводит его до подчинённой роли, чтобы сохранить свою главенствующую идентичность «творца». Это самообман, позволяющий оттянуть перестройку мышления. Это как использовать смартфон только для звонков, отказываясь от интернета.

Теневая сторона: Страх потери контроля и авторства. «Если я не написал эту функцию ручками, я не могу ей гордиться и не чувствую, что это моё».

Стадия 4: Депрессия. «Нас всех заменят. Учить что-либо бессмысленно».

Типичное высказывание: «Джуниоров больше не наймут. Сеньоры через 5 лет станут никому не нужны. Это конец профессии».

Что на самом деле происходит: Крах гедонистической адаптации. Человек искренне наслаждался процессом написания кода (ремеслом), а не только результатом. Осознание, что эта часть уйдёт, вызывает горевание. Это реальная потеря, и её нужно оплакать, а не обесценивать. Мы нормализуем эту печаль: да, мы потеряем радость «набивания» идеального класса ручками, как гончар потерял радость ручной лепки горшков, когда появился круг.

Теневая сторона: Выученная беспомощность, маскирующая нежелание переучиваться под благовидным предлогом «всё равно конец».

Стадия 5: Принятие. «Я — визионер, а не клавиатурный тигр».

Типичное высказывание: «Я трачу 2 часа на дизайн контракта и 5 минут на генерацию кода, который идеально ему следует. Я стал больше думать о бизнесе и меньше о синтаксисе».

Что на самом деле происходит: Трансформация идентичности. Разработчик перестаёт измерять свою ценность в килобайтах написанного кода. Он начинает измерять её в килобайтах *не написанного* кода (сложности, которой удалось избежать) и в качестве решённых бизнес-проблем. Метафора: он перестал быть каменщиком и стал архитектором, который ещё и проверяет качество кладки, не брезгуя мастерком при необходимости.

Приобретение: Спокойная уверенность. Это не самоуспокоение, а трезвый расчёт: кто-то должен проектировать то, что ИИ будет собирать.

2. Практический тест для самодиагностики: «Кто вы на самом деле?»

Мы дадим читателю не опросник с баллами, а серию провокационных дилемм, которые нужно проговорить с самим собой честно. Ответ на них — это вектор его личного развития.

Дилемма 1 (Гордость): «Вспомните свой лучший рабочий момент за последний месяц. Что вызвало у вас большую гордость: а) красивое применение хитрого алгоритма, который вы написали в 3 часа ночи, или б) архитектурное решение, которое позволило выкинуть 5000 строк старого легаси-кода и упростило жизнь трём командам?»

Интерпретация: Вариант «а» — риск «синдрома ремесленника». Вариант «б» — мышление Архитектора Намерений.

Дилемма 2 (Угроза): «Представьте, что вышел приказ: весь код в проекте с завтрашнего дня пишут ИИ-агенты. Ваша роль — только проверять, задавать вопросы и описывать желае-

мое поведение. Что вы чувствуете? а) "Меня лишили главного удовольствия и моей сути", б) "Ну наконец-то я смогу заняться тем, до чего вечно не доходили руки — системным анализом и улучшением архитектуры"».

Интерпретация: Это тест на готовность к Стадии 5.

Дилемма 3 (Обучение): «Вам нужно освоить новую предметную область (например, логистику). Вы: а) идёте читать документацию фреймворков и API, которые используются в логистических системах, или б) идёте говорить с логистами, читать учебник по управлению цепочками поставок и рисовать схему процессов?»

Интерпретация: Вариант «а» — фокус на «токенизируемую» ценность. Вариант «б» — движение в сторону Глубины (Ось X из Главы 1.3).

3. Рефрейминг страха: от паралича к действию.

Заключительная часть раздела превратит выявленный страх в топливо.

Страх как компас: Ваш страх указывает на то, какая часть вашей нынешней работы является механической и уязвимой. Боитесь, что ИИ напишет этот модуль? Значит, этот модуль — boilerplate. Не боитесь за архитектуру? Значит, вы чувствуете в себе «нетокенизируемую» ценность.

Упражнение: Возьмите список своих еженедельных задач. Напротив каждой поставьте маркер: «Страх, что это сделает ИИ» (шкала 1-5). Задачи с рейтингом 4-5 — это ваш план немедленного апгрейда навыков. Их нужно передать ИИ первыми, чтобы освободить себя. Задачи с рейтингом 1 — это ваша будущая специализация, которую нужно углублять.

Итоговый вывод подраздела: ИИ не увольняет человека. Он просто делает экономически невыгодным заниматься тем, что не требует человечности. Кризис, вызванный LLM, — это не кризис безработицы, это кризис идентичности. И в отличие от экономического, этот кризис решается не поиском новой работы, а поиском нового себя. Пройдя пять стадий, мы выходим не просто «принявшими неизбежное», а впервые за долгое время — настоящими.

Глава 2. Архитектор намерений вместо наборщика текста.

2.1. Новый центр тяжести: смещение от «Как написать» к «Что заставить сделать».

Это фундаментальный когнитивный сдвиг, без которого все остальные навыки книги не работают. Речь идёт о переходе от императивного мышления (пошаговые инструкции: создай переменную, пройдись циклом, вызови метод) к декларативному (описание желаемого результата и системы ограничений, в рамках которых этот результат должен быть достигнут). Это не просто «написание хороших промптов» — это смена операционной системы мозга. Императивное мышление предполагает, что мы контролируем *процесс*. Декларативное — что мы контролируем *результат*, делегируя процесс недетерминированному агенту. Главный вызов здесь — не технический, а психологический: как управлять ожиданиями от системы, которая может решить задачу не так, как вы себе представляли, но при этом формально правильно. Глава учит «дирижировать» оркестром, а не играть за каждую скрипку.

Мы откроем раздел яркой метафорой, которая станет сквозной для всей книги.

1. Два образа: Композитор против Дирижёра.

Мы детально разберём две роли, чтобы читатель мог физически ощутить разницу между старым и новым мышлением.

Начнём с **образа прошлого — Разработчик как Композитор**. Это знакомая всем модель. Композитор пишет партитуру. Каждая нота, каждая пауза, каждый инструмент прописаны им лично. Оркестр, то есть компьютер, — идеальный исполнитель, который играет строго то, что написано. Если нота фальшивая — виновата партитура, нужно найти ошибку в нотах. Мышление здесь императивное, линейное: «Я знаю, *как именно* система должна прийти к результату». Ценность композитора — в его способности создать идеальную последовательность нот, ничего не забыть и не перепутать. Проблемы этого образа в эпоху LLM очевидны: представьте композитора, которому дали оркестр, способный самостоятельно импровизировать партии по наброску. Композитор в панике пытается прописать каждому музыканту его импровизацию — это абсурдно и уничтожает весь смысл нового инструмента. Именно так выглядит разработчик, который использует LLM только как автодополнение и продолжает мыслить пошагово: «сначала создай переменную, потом пройдись циклом, потом вызови метод». Он пытается управлять процессом, который ему уже не принадлежит и не должен принадлежать.

Теперь **образ будущего — Разработчик как Дирижёр**. Это целевая модель. Дирижёр не играет ни на одном инструменте. У него есть партитура — Спецификация, — но он управляет *интерпретацией*. Он задаёт темп, то есть скорость разработки и деплоя. Он регулирует громкость — приоритеты фич. Он указывает вступление партий — очерёдность реализации модулей. Он слышит, когда скрипки фальшивят — это баг, — и останавливает оркестр, чтобы поправить *общее звучание*, а не перебирать струны за музыканта. Его мышление декларативное, управляющее ожиданиями: «Я знаю, *что* должно прозвучать в этом месте, и я опишу это словами. *Как именно* скрипач это сыграет — его мастерство, пока оно служит общему звучанию». Ключевой навык дирижёра — слышать целое, а не отдельные партии; управлять не кодом, а поведением системы.

Теперь проведём прямое сравнение этих двух моделей по ключевым критериям.

Критерий первый — **единица работы**. Для композитора это строка кода, функция, класс. Он мыслит строительными блоками реализации. Для дирижёра — контракт, спецификация, инвариант. Он мыслит ограничениями и гарантиями.

Критерий второй — **отношение к ошибке**. Для композитора ошибка — это дефект в логике, баг. Он ищет, где в цепочке рассуждений произошёл сбой. Для дирижёра ошибка —

это отклонение от спецификации. Он не спрашивает «почему код упал», он спрашивает «в какой момент поведение системы перестало соответствовать контракту».

Критерий третий — **точка контроля**. Композитор контролирует процесс — «как именно делается работа». Дирижёр контролирует результат — «что именно должно быть сделано».

Критерий четвёртый — **общение с ИИ**. Композитор говорит: «Напиши функцию, которая принимает A и возвращает B, проходя циклом по C». Он продолжает программировать, просто на естественном языке. Дирижёр говорит: «Мне нужен компонент, который гарантирует, что при любых входных данных X, состояние системы Y останется неизменным. Опиши, как ты это обеспечишь, а потом реализуй». Он ставит задачу и требует обоснования.

Критерий пятый, самый важный — **главный страх**. Композитор боится: «Я не знаю, как это написать». Его тревога живёт в пространстве синтаксиса и алгоритмов. Дирижёр боится другого: «Я не могу точно описать, что мне нужно». Его тревога сместилась в пространство спецификации и коммуникации. И это — продуктивный страх, потому что он ведёт к росту, а не к параличу.

2. Практикум: Переписываем сложную бизнес-логику декларативно.

Это центральное упражнение раздела. Мы возьмём реальный, запутанный кейс и проведём читателя через три шага трансформации его мышления. Кейс: «Система бронирования переговорных комнат в офисе».

Шаг 1. Императивный подход — как мы привыкли думать.

Сначала мы моделируем типичный мыслительный процесс разработчика старой школы. Он звучит примерно так: «Создам таблицу Rooms, таблицу Bookings. В Bookings будет start_time, end_time, room_id, user_id. Потом напишу сервис, который перед созданием бронирования проверяет: а нет ли уже записей, где room_id равен нужному, и интервал времени пересекается? Если есть — возвращаю ошибку. Если нет — создаю запись. О, и ещё надо учесть часовые пояса. И сделать cron-задачу, которая снимает просроченные брони. И админку, чтобы админ мог удалять чужие бронирования...»

Заметим, что происходит в этот момент. Разработчик сразу проваливается в реализацию. Он думает таблицами, запросами, cron-задачами. Бизнес-правила уже искажены техническими деталями: он уже решил, что это будет реляционная база, хотя задача этого не требует; он уже думает о часовых поясах, которые не упоминались в требованиях; он смешивает бизнес-логику и инфраструктурные заботы. Самое страшное — он *уже спроектировал систему*, не осознав этого. Архитектурные решения приняты на автомате.

Шаг 2. Декларативная трансформация — выделяем сущности и правила.

Теперь мы говорим: «Стоп. Забудем про базу данных. Забудем про код. Что мы знаем о реальном мире переговорок?» И начинаем выписывать декларативные утверждения.

Первое — **состояния и сущности**. Комната может быть либо Доступна, либо Забронирована в конкретный временной слот. Бронирование принадлежит конкретному пользователю. Временной слот имеет начало и конец.

Второе — **инварианты, то есть правила, которые не должны нарушаться никогда**. Две разные брони не могут пересекаться во времени для одной комнаты. Бронирование не может иметь конец раньше начала. Пользователь не может удалить чужое бронирование, если он не администратор.

Третье — **переходы состояний**. Бронирование создаётся пользователем и переходит в статус «Активно». Бронирование может быть отменено автором или администратором и переходит в статус «Отменено». Бронирование автоматически переходит в статус «Завершено» по истечении времени.

Четвёртое — **желаемые поведения**. При попытке создать пересекающееся бронирование система должна отклонить запрос с указанием конфликтующего слота. При запросе списка

комнат на заданный день система должна вернуть все комнаты с их статусами доступности по часам.

Обратите внимание: здесь нет ни слова про SQL, REST, JSON или Docker. Это чистая модель предметной области, очищенная от технологического шума. И именно это — новый центр тяжести работы разработчика.

Шаг 3. Формируем промпт и управляем результатом.

Только теперь мы идём к ИИ-модели. Но не с командой «напиши мне API», а с декларативной спецификацией.

Наш промпт будет выглядеть примерно так: «Я проектирую систему бронирования переговорных комнат. Вот мои ограничения и инварианты. Контекст: офис с множеством комнат, пользователи бронируют слоты. Предложи три разных архитектурных подхода к реализации: первый — классический монолит с реляционной базой, второй — event-driven архитектура, третий — минималистичное решение на ключ-значение хранилище. Для каждого опиши: как он обеспечивает инвариант непересечения броней, как он масштабируется, какие у него риски. Пока не пиши код, только дизайн».

Заметим разницу: мы не спрашиваем у модели *как писать*, мы просим её *обосновать дизайн*. Мы становимся дирижёром: модель играет партии трёх архитектур, а мы слушаем, задаём вопросы и выбираем.

После выбора подхода мы даём следующий промпт: «Реализуй решение по архитектуре №2. Но сначала напиши юнит-тесты, которые проверяют инвариант непересечения броней. Тесты должны покрывать: две брони на одно время — отклонено; бронь с началом позже конца — отклонено; бронь, примыкающая к существующей вплотную — разрешено. Когда тесты будут готовы, напиши код, который их проходит».

Это и есть работа дирижёра: задать рамку (тесты как формальная спецификация), потребовать обоснования (архитектурные варианты) и оценивать результат по звучанию целого, а не по нотам.

Итоговый вывод подраздела: Новый центр тяжести — это не технический навык, а точка приложения внимания. Императивный разработчик спрашивает: «Как мне описать процесс, чтобы машина его исполнила?» Декларативный разработчик спрашивает: «Как мне описать результат и ограничения, чтобы машина не смогла ошибиться, даже если найдёт неожиданный путь?» Первый вопрос ведёт к контролю над кодом. Второй — к контролю над системой. Разница между ними и есть разница между тем, кого ИИ заменит, и тем, кого ИИ сделает невероятно востребованным.

2.2. Определение новой должности: кто такой разработчик промптов на уровне Senior и Staff

Это развенчание опасного и унизительного мифа, который уже укоренился в индустрии. Миф звучит так: «Промпт-инженер — это junior, который просто пишет "ты — эксперт" и подбирает волшебные слова, и эту роль автоматизируют через полгода». Этот взгляд безнадёжно плоский. Он путает оператора чата с архитектором лингвистических интерфейсов. Настоящий Senior и Staff в этой парадигме — это не человек, который лучше всех формулирует запросы. Это человек, который проектирует *системы* общения с моделями: создаёт DSL для домена, выстраивает конвейеры верификации, управляет контекстной памятью агентов и оркестрирует ансамбли моделей. Он проектирует не промпт — он проектирует «промпт-программу» (Prompt Program) со своей логикой ветвления, циклами и утверждениями, где каждый отдельный промпт — лишь атомарная инструкция в большой архитектуре. Глава даёт профессиограмму трёх уровней и детальный разбор того, чем на самом деле занят Staff Prompt Architect.

Мы начнём с дерзкого заявления: «Если вы называете себя промпт-инженером и ваша работа состоит в переборе формулировок в ChatGPT — вы не инженер. Вы тестировщик чужой модели. Инженерная работа начинается там, где заканчивается одиночный промпт».

1. Профессиограмма: три уровня зрелости новой роли.

Мы детально опишем каждую ступень, чтобы читатель мог определить своё текущее положение и увидеть горизонт роста. Это не просто названия — это разные способы мышления.

Уровень 1. Junior Prompt Engineer: «Оператор фактов».

Это входная точка, которую многие ошибочно принимают за всю профессию. Такой специалист решает атомарные задачи: «напиши функцию валидации email», «переведи этот JSON в XML», «объясни, что делает этот код». Его главный навык — чёткая формулировка запроса и умение итеративно уточнять результат: «нет, сделай без регулярок», «добавь обработку ошибок», «теперь на TypeScript». Он работает в парадигме «один запрос — один ответ». Он хорошо знает определённую модель, её капризы и особенности. Его ценность — в скорости решения типовых задач. Но это нижний уровень, и да, он наиболее уязвим для автоматизации, потому что модели становятся лучше в понимании с полуслова, а интерфейсы — удобнее. Расти с этого уровня нужно не в сторону «ещё более хитрых формулировок», а в сторону системного мышления.

Уровень 2. Middle Prompt Engineer: «Управляющий контекстом».

На этом уровне разработчик перестаёт мыслить одиночными запросами. Он понимает, что главный ресурс — не слова, а *контекстное окно*. Он управляет памятью диалога.

Его компетенции: он проектирует «разговоры с продолжением», разбивая сложную задачу на цепочку промптов, где каждый следующий опирается на результат предыдущего; он вручную управляет контекстом, понимая, когда нужно «забыть» предыдущие сообщения, чтобы не зашумлять модель; он использует технику «скользящего окна» для длинных задач, пересобирая ключевую информацию из истории диалога; он интегрирует внешнюю память — подключает RAG, векторные базы, чтобы модель имела доступ к актуальной документации проекта, а не только к тому, что он ей вручную скопировал.

Здесь появляется первая инженерная сложность: Middle работает не с одним ответом модели, а с *потоком диалога* как с программой, где есть состояние, память и переходы. Он уже не просто оператор, он — создатель «сессий» с управляемым жизненным циклом. Пример его работы: он не просит «напиши микросервис», а проводит модель через пять последовательных

стадий: сначала обсуждает контракты API, затем генерирует тесты под эти контракты, затем — код, затем — документацию, и на каждом шаге подаёт на вход выжимку из предыдущего шага, контролируя целостность.

Уровень 3. Senior / Staff Prompt Architect: «Создатель DSL и систем верификации».

Вот где происходит тектонический сдвиг. Senior не пишет промпты для решения задач. Он создаёт *язык*, на котором другие разработчики или даже другие агенты будут общаться с моделью. Он проектирует не текст запроса, а «промпт-программу» — систему, в которой промпты являются модулями со своей логикой.

Что он создаёт на практике. Первое — **DSL общения для конкретного домена**. Это формальный, но человекочитаемый язык описания сущностей и операций, который становится прослойкой между бизнес-требованием и генерацией кода. Например, DSL для бэкенда компании может выглядеть как набор декларативных конструкций: ENTITY Order, API POST / orders WITH AUTH, CONSTRAINT: order.total > 0. Модель обучается понимать этот DSL через системный промпт-легенду. Другие разработчики компании пишут на этом DSL — и получают предсказуемый, стандартизированный код.

Второе — **системы автоматической верификации**. Senior не доверяет сгенерированному коду. Он проектирует «контур проверки»: модель генерирует код, затем другой промпт (или другая модель) проверяет этот код на соответствие исходному DSL, ищет уязвимости и логические дыры, и только после этого код попадает к человеку на финальное ревью. Он строит пайплайн «Спецификация → Генерация → Авто-ревью → Человеческое утверждение».

Третье — **оркестрация ансамблей моделей**. Staff-уровень означает, что разработчик управляет не одной LLM, а роем специализированных агентов. Одна модель — эксперт по безопасности, другая — по перформансу, третья — по пользовательскому опыту. Они «спорят» друг с другом в автоматическом режиме, а человек-дирижёр модерирует этот спор и принимает финальное решение в сложных случаях.

Четвёртое — **проектирование self-correcting циклов**. Это высший пилотаж: промпт-программа, которая сама обнаруживает свои ошибки и исправляет их. Например: модель генерирует SQL-запрос; второй промпт просит её «представь, что ты — злобный DBA, найди уязвимость в этом запросе»; третий промпт даёт исходную задачу плюс найденную уязвимость и просит переписать запрос безопасно. И так по кругу, пока уязвимости не закончатся. Человек не участвует в итерациях — он спроектировал этот самоисправляющийся механизм.

2. Детальный разбор: что такое «промпт-программа» и чем она отличается от просто промпта.

Здесь мы дадим читателю новый концептуальный инструмент. Одиночный промпт — это линейная инструкция: запрос → ответ. Промпт-программа — это граф или конечный автомат, где узлы — это вызовы модели, а рёбра — логические переходы, зависящие от результатов предыдущих вызовов.

Элементы промпт-программы. **Ветвление**: «Если модель ответила кодом, который не компилируется, — передай ошибку компилятора обратно в модель с инструкцией исправить». Это цикл «генерация — компиляция — исправление», который не требует участия человека. **Циклы**: «Генерируй тест-кейсы для функции, пока покрытие веток не достигнет 95%». Модель в цикле генерирует тесты, прогоняет их, видит непокрытые ветки и генерирует ещё. **Утверждения (assertions)**: встроенные в промпт-программу проверки. «Если сгенерированный ответ содержит слово "предполагая" или "допустим", останови выполнение и запроси у разработчика уточнение — модель угадывает, а не следует спецификации». **Память состояний**: промпт-программа ведёт журнал принятых решений. Если на пятом шаге выясняется, что решение на втором шаге было неверным, программа автоматически откатывается и перезапускает генерацию с уточнённым контекстом.

3. Кто нанимает Staff Prompt Architects и зачем.

Мы дадим читателю отрезвляющий взгляд на рынок. Сегодня такие позиции редко называются «Prompt Architect» — это временный термин. Они скрываются за названиями: AI Systems Engineer, LLM Ops Lead, Head of AI Tooling, Principal Architect (AI-augmented systems). Но суть одна: компаниям нужны люди, которые превращают дорогую и непредсказуемую модель в стабильный, конвейерный инструмент производства кода. Им не нужен человек, который пишет промпты. Им нужен человек, который создаёт *фабрику*, где другие люди и агенты пишут промпты по его стандартам, а на выходе получается качественный продукт.

Итоговый вывод подраздела: Разница между Junior и Staff в новой парадигме — это разница между тем, кто использует микроскоп, чтобы лучше видеть, и тем, кто проектирует сам микроскоп. Junior спрашивает модель. Middle управляет диалогом с моделью. Senior проектирует систему, в которой модели работают согласованно и проверяемо. Если вы хотите быть незаменимым — переставайте писать промпты. Начинайте проектировать процессы, в которых промпты пишутся, исполняются и верифицируются автоматически. Ваш продукт — не текст запроса. Ваш продукт — архитектура взаимодействия с искусственным интеллектом.

2.3. Когнитивный разворот: мышление не циклами и условиями, а сущностями и контрактами

Традиционное обучение программированию — будь то университетский курс или буткемп — ставит во главу угла поток управления. Студента учат мыслить последовательностями: «сначала проверь условие, потом пройдись циклом, потом вызови функцию, потом обработай исключение». Весь интеллект разработчика направлен на то, чтобы выстроить правильную цепочку команд. Алгоритм — это маршрут. Код — это карта этого маршрута. И это работало, пока код был нашей единственной материализованной мыслью.

Новая реальность требует когнитивного разворота — возможно, самого трудного во всей книге, потому что он ломает многолетние нейронные связи. Мышление должно сместиться с потока управления на трансформации данных и инварианты. Это подход, известный как Design by Contract — проектирование по контракту, — но в эпоху LLM он из нишевой методологии становится основой выживания. Суть проста: вы перестаёте думать о том, *как* данные проходят через систему, и начинаете думать о том, *какими* они должны быть на входе, *какими* — на выходе, и *что никогда не должно нарушаться* в процессе. Код перестаёт быть вашим главным интеллектуальным продуктом. Он становится лишь одним из возможных доказательств соблюдения контракта, сгенерированным ИИ. И как любое доказательство теоремы, оно может быть длинным или коротким, элегантным или корявым — это неважно, если оно корректно.

Этот сдвиг можно сравнить с переходом от рисования карты вручную к описанию географии. Раньше вы были картографом, который прорисовывал каждую тропинку. Теперь вы — географ, который говорит: «Между точкой А и точкой Б есть река. Через реку всегда есть мост. Ни одна дорога не поднимается выше 500 метров». А ИИ-картограф по этому описанию рисует десять разных карт — и вы оцениваете их не по красоте линий, а по тому, соблюдены ли ваши географические ограничения.

Перейдём к практике. Я возьму сложный модуль — систему бронирования авиабилетов, — и покажу, как выглядит мышление контрактами на каждом этапе.

Первое, с чего мы начинаем, — **предусловия**. Это утверждения, которые должны быть истинны до того, как операция выполнится. Мы не пишем код. Мы формулируем правила мира. Для операции «забронировать билет» предусловия выглядят так: пользователь аутентифицирован и идентифицирован; рейс существует в расписании; на рейсе есть как минимум одно свободное место; дата вылета находится в будущем относительно текущего момента; пассажир предоставил все обязательные данные — имя, документ, дату рождения; платёжный метод пользователя валиден и не просрочен. Обратите внимание: здесь нет ни слова о том, как это проверить. Нет запросов к базе данных. Нет API-вызовов. Есть только утверждения о реальности, которые должны быть истинны на входе в операцию.

Второе — **постусловия**. Это утверждения, которые гарантированно истинны после успешного выполнения операции. Для того же «забронировать билет»: создана запись о бронировании с уникальным идентификатором; количество свободных мест на рейсе уменьшилось ровно на количество забронированных мест; билет привязан к конкретному пассажиру и не может быть случайно передан другому; сумма, списанная с платёжного метода, в точности равна стоимости билета на момент бронирования; пользователь получил подтверждение — email или push — с деталями рейса. Снова никакого кода. Только факты о состоянии мира после операции.

Третье, самое мощное, — **инварианты класса**. Это утверждения, которые истинны всегда, в любой момент времени, вне зависимости от того, какие операции выполняются. Для системы бронирования: количество проданных билетов на рейс никогда не превышает вмести-

мость самолёта; один и тот же пассажир не может иметь два билета на один и тот же рейс — возможно, на разные, но не на один; стоимость билета не может быть отрицательной или нулевой; дата бронирования всегда предшествует дате вылета; каждое бронирование принадлежит ровно одному пользователю и ровно одному рейсу. Инварианты — это конституция системы. Они не привязаны ни к какой конкретной операции. Они — закон, который нельзя нарушить.

Теперь магия. Имея этот набор из предусловий, постусловий и инвариантов, мы идём к ИИ-модели и говорим: «Вот контракт системы бронирования авиабилетов. Реализуй его четырьмя разными способами».

Модель возвращает четыре архитектуры. Первая — классический монолит с реляционной базой: транзакции, пессимистические блокировки, проверка мест через SELECT FOR UPDATE. Вторая — event-driven архитектура с Kafka: события «БилетЗабронирован», «ПлатёжПодтверждён», асинхронные саги для отката при сбоях. Третья — минималистичное решение на ключ-значение хранилище с оптимистическими блокировками: каждое место — ключ, атомарная операция compare-and-swap. Четвёртая — акторная модель: каждый рейс — актор, который сериализует все операции над собой и гарантирует инварианты естественным образом.

Теперь мы, как Архитекторы Намерений, оцениваем эти четыре решения. Но оцениваем мы их не по стилю кода — не по тому, насколько «чистый» получился Java-код или насколько идиоматичен Elixir. Мы оцениваем их по единственному критерию: **полнота покрытия контракта**.

Первый вопрос, который мы задаём каждому решению: «Где в твоём коде проверяется инвариант "количество проданных билетов никогда не превышает вместимость самолёта"?» Монолит показывает транзакцию с SELECT FOR UPDATE — окей, это работает в пределах одного инстанса, но что если инстансов два? Event-driven решение показывает сагу — но что если между событием «ПлатёжПодтверждён» и «МестоЗабронировано» другой пользователь купил последнее место? Решение на compare-and-swap показывает атомарную операцию, которая упадёт при конфликте — это честно. Акторная модель показывает сериализацию — конфликт невозможен в принципе.

Второй вопрос: «Как ты гарантируешь, что один пассажир не купит два билета на один рейс?» Монолит показывает уникальный constraint в базе — надёжно, но только если нет распределённой транзакции с платёжным шлюзом. Event-driven показывает проверку в сервисе-саге — но между проверкой и записью есть окно. Актор показывает, что сама сущность Рейса проверяет список пассажиров — окей. Мы не говорим «этот код красивый» или «этот код уродливый». Мы говорим: «Вот этот код покрывает контракт полностью, а вот этот — создаёт окно уязвимости длиной в 50 миллисекунд, которое ты, разработчик, должен осознать и либо закрыть, либо принять как допустимый риск».

В этом и заключается когнитивный разворот. Раньше мы оценивали код по его внутренним качествам: читаемость, модульность, соответствие паттернам. Теперь мы оцениваем его по внешнему критерию: насколько он доказывает соблюдение контракта. Код может быть написан на языке, которого мы не знаем, с использованием фреймворка, о котором мы слышали впервые — это не имеет значения. Если контракт покрыт полностью, код корректен. Если есть дыра — код некорректен, как бы красиво он ни выглядел.

Это освобождает невообразимый объём когнитивной мощности. Вам больше не нужно держать в голове, как работает ORM или как настроить пул соединений. ИИ разберётся с реализацией. Ваша голова занята более важным: «А точно ли мы описали все инварианты? А что будет, если платёжный шлюз ответит через 30 секунд? А не упустили ли мы случай, когда пользователь меняет паспортные данные между бронью и вылетом?» Вы перестаёте быть контролёром потока команд и становитесь исследователем пространства возможных нарушений контракта.

Подытожим этот когнитивный разворот одной фразой, которую стоит запомнить как мантру: «Раньше я думал, как заставить машину сделать правильно. Теперь я думаю, что значит "правильно", — а машина пусть сама ищет способ». В этой мантре — вся суть новой профессии.

2.4. Почему «грязный код» прототипа теперь ценнее «чистого кода» бездумного

Это глава-ниспровержение. Глава-атака на одну из самых оберегаемых догм современной разработки — догму Чистого Кода. Я не буду утверждать, что Роберт Мартин был не прав. Я скажу хуже: он был прав для своей эпохи, но его советы стали опасным анахронизмом в эпоху, когда код может быть переписан за секунды.

Десятилетиями нас учили: «Всегда оставляйте код чище, чем вы его нашли». «Думайте о том, кто будет читать ваш код через полгода». «Не смешивайте ответственности». «Выделяйте абстракции». «Следуйте принципам SOLID». Эти правила рождались из суровой реальности: стоимость изменения кода экспоненциально росла со временем. Грязный код был токсичным долгом, который душил проект. И каждый ответственный разработчик должен был думать о будущем читателе, потому что этим читателем, скорее всего, будет он сам через полгода, и он ничего не вспомнит.

LLM переворачивает эту экономику с ног на голову. Если ИИ может за секунду взять любой «грязный» прототип и сделать его рефакторинг до «чистого» кода с выделенными абстракциями, правильными паттернами и идиоматичным синтаксисом, — тогда время, потраченное на мысленное причёсывание синтаксиса до первого запуска, является не добродетелью, а преступной медлительностью. Это всё равно что шлифовать и полировать каждую деталь прототипа моста, собранного из зубочисток, прежде чем проверить, выдержит ли он вес игрушечной машинки. Вы полировали то, что, возможно, нужно было просто выбросить после первого эксперимента.

Ценность в новой парадигме смещается в скорость проверки гипотезы. Главный вопрос больше не звучит как «Насколько легко этот код будет поддерживать через год?». Главный вопрос теперь: «Насколько быстро этот код докажет, что моя идея работает — или что она ошибочна?». Прототип из артефакта, который стыдно показывать, превращается в расходный материал для мышления. Он — не фундамент, а щуп, которым мы протыкаем неизвестность.

Я называю это концепцией **Одноразовых Архитектурных Спайков**. Термин «спайк» пришёл из экстремального программирования — это эксперимент для исследования неизвестного. Но раньше спайк был дорогим: вы тратили день на его написание, а потом ещё полдня на разбор последствий. Теперь спайк можно сгенерировать, протестировать гипотезу и выбросить за 15 минут. И это меняет всё.

Перейдём к практическому упражнению, которое должен выполнить каждый читатель, чтобы сломать свой внутренний запрет на грязный код.

Возьмём бизнес-гипотезу: «Наш интернет-магазин должен показывать пользователю персонализированную ленту товаров, отсортированную не по дате добавления, а по предсказанной вероятности покупки, которая вычисляется на основе его последних трёх просмотров». Это туманная, нечёткая задача. Мы не знаем, какой алгоритм сработает. Мы не знаем, какие данные реально коррелируют с покупкой. Мы не знаем, как это будет выглядеть в интерфейсе. У нас есть 10 минут.

Шаг первый. Мы пишем чудовищный промпт, намеренно нарушая все принципы чистоты: «Создай один единственный файл на Python. Смешай всё в кучу: прямые SQL-запросы к базе через сырые строки, логику вычисления рекомендаций, рендеринг HTML с инлайн-стилями, обработку HTTP-запроса через примитивный веб-сервер. Не используй никаких фреймворков, никакого ORM, никаких шаблонизаторов. Не разделяй на модули. Не пиши тесты. Не обрабатывай ошибки элегантно — просто упади, если что-то не так. Мне нужен монструозный, однофайловый прототип, который просто работает. Алгоритм рекомендации:

тупо посчитай косинусное сходство между вектором последних трёх просмотренных товаров и всеми остальными товарами. Всё. Сделай это за один ответ».

Шаг второй. Мы получаем 300 строк ужасного, нечитаемого, нарушающего все мыслимые принципы кода. Мы запускаем его. Он работает — медленно, коряво, но работает. Мы открываем в браузере, кликаем. Видим персонализированную ленту. И через 30 секунд взаимодействия понимаем главное: гипотеза о косинусном сходстве даёт бессмысленные результаты. Рекомендации выглядят случайными. Мы только что спасли компанию от месяцев разработки ненужной фичи, а себя — от рефакторинга никому не нужного кода. Время, потраченное на прототип — 10 минут. Время, которое мы *не* потратили на написание чистого кода с репозиториями, сервисами, фабриками и тестами — примерно два дня.

Шаг третий. Мы выбрасываем этот файл. Полностью. Без сожаления. Он сделал своё дело — проверил гипотезу. Теперь, вооружённые знанием «косинусное сходство не работает, нужно пробовать коллаборативную фильтрацию», мы создаём второй грязный прототип. И так до тех пор, пока не нащупаем работающий подход. И только тогда, когда гипотеза подтверждена, мы говорим ИИ: «Теперь возьми этот грязный прототип и сделай из него чистый production-код. Раздели на модули. Добавь обработку ошибок. Напиши тесты. Используй нормальный фреймворк».

Но чтобы этот подход работал, нам нужны чёткие критерии — когда прототип «достаточно хорош», чтобы остановиться и либо выбросить его, либо начать очистку. Я предлагаю три железных правила.

Критерий первый: **Прототип отвечает ровно на один вопрос**. Если вы ловите себя на мысли «заодно сделаем красивый интерфейс» или «и авторизацию прикрутим заодно» — вы уже не прототипируете, вы начали продакшен-разработку, просто под прикрытием. Хороший прототип отвечает на чёткий бинарный вопрос: «Работает ли подход А?» — да или нет. Всё остальное — шум.

Критерий второй: **Время жизни прототипа измеряется минутами, а не днями**. Если вы потратили на прототип больше часа — вы делаете что-то не то. Прототип, который живёт дольше одного рабочего дня, перестаёт быть прототипом. Он становится легаси-кодом, просто вы притворяетесь, что это не так. У прототипа должен быть таймер самоуничтожения — метафорический или даже реальный. Вы принимаете решение о его судьбе в течение одного дня: выбросить или начать production-версию с нуля.

Критерий третий: **Прототип не попадает в репозиторий проекта**. Никогда. Он жив

Часть II. Карта новых компетенций: чему учиться, когда учить синтаксис бесполезно

Глава 3. Искусство спецификации: промпт-инжиниринг как системная дисциплина

3.1. Анатомия идеального промпта: не просьба, а техзадание

Это первая глава практического блока, и её задача — перевести читателя из режима «общения с моделью» в режим «составления технической документации для исполнителя». Большинство разработчиков, впервые столкнувшись с LLM, совершают одну и ту же ошибку: они общаются с моделью как с равным коллегой, который «поймёт с полуслова». «Напиши авторизацию». «Добавь пагинацию». «Почини этот баг». Это не инженерия. Это гадание.

Идеальный промпт — это не просьба и не вежливое пожелание. Это техническое задание в миниатюре. Оно должно содержать все элементы, которые вы бы включили в спецификацию для аутсорс-команды, работающей в другом часовом поясе и не имеющей возможности переспросить. Потому что, по сути, LLM — это и есть такая команда. Она не переспрашивает. Она интерпретирует. И если вы оставили пространство для интерпретации там, где его быть не должно, — она его заполнит. Причём заполнит статистически наиболее вероятным, а не правильным для вашего контекста способом.

Я предлагаю адаптировать для кода известный в промпт-инжиниринге шаблон CO-STAR. В оригинале это аббревиатура для Context, Objective, Style, Tone, Audience, Response Format. Применительно к разработке мы трансформируем её в боевой инструмент спецификации.

Первый элемент — **Контекст**. Это самая объёмная и самая важная часть промпта. Вы должны описать не просто текущую задачу, а среду, в которой она существует. Какой это проект? Монолит или микросервисы? Какая версия языка? Какие фреймворки уже используются и почему? Какие соглашения о нейминге приняты в команде? Как устроена база данных? Есть ли существующий код, с которым новый должен взаимодействовать, и как именно? Если вы пишете «сделай авторизацию» без контекста, модель сделает вам стандартную JWT-авторизацию на Node.js, даже если ваш проект — на Go, с сессионной авторизацией через Redis, и в компании принято называть модули не «auth», а «identity». Контекст — это ваша защита от «статистически правильного, но для вас неправильного» решения.

Второй элемент — **Цель**. Это не расплывчатое «сделай фичу», а точное описание желаемого результата, желательно с критериями приёмки. «Реализовать API-эндпоинт для входа пользователя, который принимает email и пароль, возвращает access token в теле ответа и refresh token в http-only куке, проверяет пароль через bcrypt, блокирует учётную запись после 5 неверных попыток на 15 минут, логирует каждую попытку входа с указанием IP». Заметьте: это не инструкция «сделай цикл, потом проверь, потом заблокируй». Это описание того, что должно получиться на выходе. Модель сама построит поток управления.

Третий элемент — **Стиль и тон**. Применительно к коду это означает: архитектурный стиль и идиоматичность. Вы должны явно указать, как должен выглядеть результат. «Используй функциональный стиль без мутаций». «Придерживайся чистой архитектуры с разделением на сущности, use cases и инфраструктуру». «Пиши в объектно-ориентированном стиле с явным внедрением зависимостей». «Код должен быть идиоматичным для Go, используй стандартную

библиотеку где возможно». Без этого указания модель смешает паттерны из разных экосистем, и вы получите Java-подобный Python или Python-подобный Rust.

Четвёртый элемент — **Аудитория**. Для кого этот код? Кто будет его читать и поддерживать? «Этот код будут читать джуниор-разработчики, поэтому избегай сложных абстракций и магии метапрограммирования — любой метод должен быть понятен после одного прочтения». Или наоборот: «Это высоконагруженный микросервис, который будут поддерживать опытные инженеры. Допустимы продвинутые паттерны, приоритет — производительность и минимальное потребление памяти». Аудитория определяет сложность допустимых решений.

Пятый элемент — **Формат ответа**. Это самая недооценённая часть промпта, и именно она отличает любителя от профессионала. Вы должны указать, в каком виде вы хотите получить результат. «Выведи три файла: auth_service.go, auth_handler.go, auth_service_test.go. Каждый файл начни с комментария, описывающего его назначение. Код должен компилироваться без ошибок». Или: «Сначала опиши свой подход в трёх абзацах. Затем представь диаграмму классов в виде ASCII-графики. И только потом выведи код». Управление форматом ответа — это управление вашим же временем на распаковку и понимание сгенерированного.

Шестой элемент, который я добавляю к CO-STAR лично, — **Ограничения и негативные спецификации**. Это то, чего модель НЕ должна делать. «Не используй внешние библиотеки, кроме стандартной». «Не генерируй main.go — это часть другого сервиса». «Не добавляй логирование, у нас свой слой логирования». «Если тебе не хватает информации — не додумывай, а укажи в ответе, каких именно данных не хватает и почему». Это страховка от самой частой болезни LLM — галлюцинаторного заполнения пробелов.

Теперь разберём боевой пример, который покажет разницу между тремя уровнями промптов на одной задаче — «реализовать авторизацию». Это не выдуманный стенд, а собирательный образ сотен реальных диалогов.

Плохой промпт. Разработчик пишет: «Напиши модуль авторизации». Всё. Модель возвращает стандартную JWT-авторизацию на Express.js с middleware, который проверяет заголовок Authorization, извлекает токен, верифицирует через секретный ключ, зашитый прямо в коде, и возвращает пользователя. Результат непригоден. Почему? Модель угадала стек (Node.js — самый популярный в её обучающей выборке для таких запросов). Она выбрала самую примитивную схему. Она зашила секрет в код. Она не учла refresh-токены. Она не добавила блокировку после неудачных попыток. Она сделала статистически среднее, а не нужное вам.

Средний промпт. Разработчик, наученный горьким опытом, пишет: «Нужен модуль авторизации на Python с FastAPI. Используй JWT. Храни пользователей в PostgreSQL. Пароли должны хешироваться». Модель возвращает рабочий код. Но всё ещё сырой: схема базы создана, но миграций нет; access token живёт вечно, нет механизма обновления; нет rate-limiting; нет логирования попыток входа; секретный ключ для JWT читается из переменной окружения, но без пояснения, как его генерировать. Этот код можно запустить, и он даже будет работать. Но в production он создаст дыры в безопасности и неудобства в поддержке.

Идеальный промпт. Теперь мы применяем наш шаблон. Это займёт полторы-две страницы текста, но результат будет пригоден к деплою с минимальными правками. Выглядит это примерно так.

Контекст: «Мы разрабатываем backend для сервиса доставки еды. Стек: Python 3.11, FastAPI, SQLAlchemy 2.0 с async, PostgreSQL 15, Redis для сессий. В проекте уже принята модульная структура: каждый домен лежит в своей папке со своими моделями, сервисами и роутами. Аутентификация должна быть выделена в модуль identity. В проекте используется гексагональная архитектура: модели домена не зависят от базы данных, репозитории реализуют интерфейсы, описанные в доменном слое. Соглашение о нейминге: эндпоинты в kebab-case, функции в snake_case, классы в PascalCase. Все ошибки API возвращаются в формате JSON

с полями error_code и message. У нас уже есть модель User с полями id, email, password_hash, is_active, failed_login_attempts, locked_until. Она лежит в modules/identity/domain/user.py».

Цель: «Реализовать API-эндпоинт POST /auth/login, который принимает JSON с полями email и password. Требования: проверить, что пользователь с таким email существует, и что он активен (is_active == True); если пользователь заблокирован — вернуть ошибку 423 с соответствующим сообщением и кодом USER_LOCKED; сравнить пароль через bcrypt; при неверном пароле инкрементировать failed_login_attempts, и если попыток стало 5 — установить locked_until на текущее время плюс 15 минут и вернуть ошибку 423 с кодом USER_LOCKED; при успешном входе сбросить failed_login_attempts в 0, создать access token через JWT со сроком жизни 15 минут и refresh token со сроком жизни 7 дней; access token вернуть в теле ответа, refresh token установить в http-only secure куку; логировать каждую попытку входа — успешную и неуспешную — с полями user_id, ip_address, timestamp, success (bool) через вызов logger.info* с экранированием персональных данных; написать постусловие: после успешного входа пользователь должен иметь валидную сессию, проверяемую через эндпоинт GET /auth/me».

* Сам по себе logger.info — это **команда (вызов метода)**, которая используется в программировании для записи информационного сообщения в лог (журнал событий).

Чтобы понять, что именно это значит, нужно разобрать его по частям:

1. logger **(объект)**

Это специально созданный объект в коде, который отвечает за логирование. Он обычно настраивается один раз в начале программы (с помощью библиотек типа logging в Python, log4j в Java, winston в JavaScript и т.д.).

2. .info() **(метод)**

Это функция, которая говорит логгеру: *«Запиши это сообщение, но пометь его как* ***информационное*** *(Information)».*

Какой смысл у уровня INFO ?

В логировании есть уровни важности (по возрастанию серьезности):

DEBUG — для отладки (мельчайшие детали, что происходит внутри функций).

INFO — информационные сообщения (ход выполнения бизнес-процессов, важные этапы работы).

WARNING — предупреждения (что-то пошло не так, но программа работает дальше).

ERROR — ошибки (функция не выполнилась, но программа жива).

CRITICAL — критические ошибки (программа падает).

logger.info используется для того, чтобы отметить **нормальное, штатное выполнение программы**. Это сообщения, которые должны быть видны администратору или разработчику в продакшене, чтобы понимать, что система жива и работает корректно.

Пример на Python (для наглядности)

```
python
import logging

# Настраиваем логгер
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def transfer_money(user, amount):
logger.info(f"Начинаем перевод {amount} рублей для пользователя {user}") # <-- Это logger.info
# ... какой-то код ...
```

logger.info(f"Перевод для {user} успешно завершен") # <-- *И это logger.info*

Если вы запустите код, в консоли вы увидите что-то вроде:

INFO:__main__:Начинаем перевод 100 рублей для пользователя Alex

Что происходит под капотом?

Когда вызывается logger.info("Текст"), происходит следующее:

Создается объект "событие" (LogRecord) с текущим временем, именем файла, строкой кода и вашим текстом.

Логгер проверяет, разрешен ли уровень INFO в настройках (если стоит уровень WARNING, то сообщение INFO не будет показано).

Сообщение передается в **Handler** (обработчик), который решает, куда его отправить: в консоль, в файл на диске, в базу данных или в систему мониторинга (например, Sentry или ELK).

Что обычно пишут в logger.info ?

Хороший тон — писать:

Начало и конец важных операций (например, "Загрузка файла начата", "Загрузка завершена").

Ключевые бизнес-события (например, "Пользователь зарегистрирован", "Заказ оплачен").

Изменения состояния системы (например, "Сервис перезапущен", "Кэш очищен").

Важно: В INFO не пишут пароли, токены доступа или персональные данные (кроме ID пользователя), так как эти логи часто хранятся в открытом виде.

****

Стиль: «Следуй гексагональной архитектуре. Интерфейсы репозиториев уже определены в доменном слое, используй их — не создавай новые. Внедрение зависимостей через конструкторы сервисов. Код должен быть покрыт type hints. Используй Pydantic для схем запросов и ответов. JWT должен подписываться алгоритмом RS256, приватный ключ читается из settings.JWT_PRIVATE_KEY».

Аудитория: «Код будет поддерживаться мидл-разработчиками, которые знакомы с FastAPI, но не являются экспертами в криптографии. Поэтому в сложных местах добавь краткий комментарий, объясняющий, почему выбран именно этот подход. Но не комментируй очевидное».

Формат ответа: «Выведи три файла: modules/identity/api/auth_router.py с эндпоинтами, modules/identity/services/auth_service.py с бизнес-логикой, modules/identity/schemas/auth_schemas.py с Pydantic-моделями. Каждый файл начни с импортов. Код должен запускаться без ошибок при условии, что все зависимости, указанные в контексте, уже установлены. В конце ответа приведи список допущений, которые ты сделал при реализации, если тебе не хватило информации».

Ограничения: «Не используй библиотеку python-jose — используй PyJWT. Не создавай новых моделей базы данных. Не меняй существующую структуру папок. Не добавляй middleware — у нас он уже есть на уровне API Gateway. Если потребуется новая зависимость, явно укажи это в списке допущений».

Разница в выхлопе между плохим и идеальным промптом колоссальна. Плохой промпт даёт код, который нужно переписывать. Средний — код, который нужно допиливать. Идеальный — код, который можно ревьюить, а не переписывать. Экономия времени не на этапе написания промпта — идеальный промпт пишется дольше плохого. Экономия на этапе работы с результатом. Вы тратите десять минут на написание идеального промпта, чтобы не тратить два часа на правку сгенерированного кода.

Здесь важен ещё один мета-навык, который приходит с практикой: умение определять, насколько подробным должен быть промпт для конкретной задачи. Не каждая задача требует двух страниц спецификации. Если вы пишете утилиту для форматирования дат — контекст и ограничения минимальны. Если вы проектируете модуль аутентификации — промпт должен быть исчерпывающим. Правило простое: объём промпта должен быть пропорционален стоимости ошибки в сгенерированном коде. Ошибка в форматировании даты видна сразу и правится за секунду. Дыра в аутентификации может стоить компании репутации и штрафов. Поэтому идеальный промпт для аутентификации — это не перфекционизм, а соразмерная осторожность.

Конец ознакомительного фрагмента.