СПРАВОЧНИК

Программирование с ИИ

Lunario

версия 2.0 · полная · 2026

Оглавление

ЧАСТЬ I · ОСНОВА

1 Ментальная модель и рабочий цикл ... 2
- 1.1 Кто здесь главный ... 2
- 1.2 Рабочий цикл ... 4
- 1.3 Что ИИ делает хорошо, а что плохо ... 6
- 1.4 Три навыка ... 9
- 1.5 Цена ошибки ... 11
- 1.6 Иллюзия готового кода ... 14

2 Чтение кода ... 17
- 2.1 Зачем читать код, если его пишет ИИ ... 17
- 2.2 Анатомия файла ... 19
- 2.3 Анатомия проекта ... 22
- 2.4 Переменные, типы, структуры данных ... 25
- 2.5 Функции ... 28
- 2.6 Условия и циклы ... 31
- 2.7 Классы и объекты ... 34
- 2.8 Исключения и обработка ошибок ... 36
- 2.9 Декораторы и «магия» ... 39
- 2.10 Асинхронность ... 42
- 2.11 Зависимости ... 45
- 2.12 Чтение трейсбека ... 48
- 2.13 Чтение диффа ... 52
- 2.14 Красные флаги в сгенерированном коде ... 55

3 Постановка задач ... 58
- 3.1 Почему ИИ делает «не то» ... 58
- 3.2 Анатомия запроса ... 61
- 3.3 Уровни детализации ... 64
- 3.4 Шаблон — новая функция ... 67
- 3.5 Шаблон — исправить баг ... 69
- 3.6 Шаблон — рефакторинг ... 71
- 3.7 Шаблон — объясни этот код ... 74
- 3.8 Шаблон — сделай ревью ... 76
- 3.9 Шаблон — напиши тесты ... 79
- 3.10 Приём — сначала план, потом код ... 81
- 3.11 Приём — пусть ИИ задаёт вопросы ... 84
- 3.12 Приём — примеры вход → выход ... 86
- 3.13 Приём — ограничить зону правок ... 89
- 3.14 Файл контекста проекта ... 91
- 3.15 Как писать мини-ТЗ ... 94
- 3.16 Антипаттерны запросов ... 97

4 Проверка и отладка ... 101
- 4.1 Правило доверия ... 101
- 4.2 Запуск и чтение логов ... 103
- 4.3 Как подготовить ошибку к починке ... 106
- 4.4 Сужаем зону поиска ... 109
- 4.5 Отладка печатью ... 111
- 4.6 Тесты — прочитать и запустить ... 114
- 4.7 Чек-лист приёмки ... 117
- 4.8 Галлюцинации ... 119

4.9 Петля «починил одно — сломал другое» 123
4.10 Когда остановиться и откатиться 125

5 Процесс, инструменты, безопасность 129
5.1 Git как страховка 129
5.2 Чат или агент в терминале 131
5.3 Контекст сессии 134
5.4 Когда начать новую сессию 136
5.5 Выбор модели под задачу 139
5.6 Окружение 141
5.7 Секреты 144
5.8 Базовый деплой 147

ЧАСТЬ II · СТЕКИ

6 Python и логика приложения 152
6.1 Анатомия Python-проекта 152
6.2 Идиомы, которые ИИ пишет постоянно 155
6.3 Типизация всерьёз 159
6.4 Асинхронность на практике 162
6.5 Постановка задач на Python 165
6.6 Ошибки ИИ в Python и чек-лист 167

7 Веб-бэкенд 171
7.1 Как устроен бэкенд 171
7.2 HTTP на пальцах 174
7.3 API, схемы и валидация 177
7.4 Читаем бэкенд-код 180
7.5 База данных 183
7.6 Запросы, которые тормозят 186
7.7 Миграции 189
7.8 Доступ, ошибки ИИ и чек-лист 192

8 Фронтенд 195
8.1 Как браузер собирает страницу 195
8.2 Читаем HTML 197
8.3 Читаем CSS 201
8.4 Инструменты разработчика 204
8.5 Ошибки ИИ во фронтенде и чек-лист 207

9 JavaScript и TypeScript 211
9.1 Читаем JavaScript 211
9.2 Асинхронность в JavaScript 214
9.3 Что добавляет TypeScript 217
9.4 Как читать чужие типы 220
9.5 Типы как проверка работы ИИ 224
9.6 Ошибки ИИ в JS и TS и чек-лист 227

10 React на TypeScript 231
10.1 Анатомия React-проекта 231
10.2 Компоненты и JSX 234
10.3 Пропсы и состояние 237
10.4 Эффекты 240
10.5 Постановка задач по интерфейсу 243
10.6 Ошибки ИИ в React и чек-лист 246

11 Flutter и Dart 249
11.1 Анатомия Flutter-проекта 249

11.2 Dart за полчаса ... 252
11.3 Виджеты ... 255
11.4 Состояние ... 258
11.5 Постановка задач по мобильному интерфейсу ... 262
11.6 Ошибки ИИ во Flutter и чек-лист ... 265

12 Сборка и публикация ... 268
12.1 Что такое сборка и почему она ломается ... 268
12.2 Android и iOS ... 275
12.3 Публикация и обновления ... 280
12.4 Чек-лист перед публикацией ... 284

ЧАСТЬ III · СКВОЗНЫЕ ТЕМЫ

13 Архитектура на пальцах ... 289
13.1 Как разбить задумку на части ... 289
13.2 Структура папок ... 293
13.3 Где хранить данные ... 297
13.4 Конфиги и окружения ... 302
13.5 Как описать архитектуру ИИ ... 306
13.6 Вопросы перед стартом проекта ... 311

14 Практикум ... 314
14.1 Кейс — телеграм-бот с внешним API ... 314
14.2 Кейс — веб-API с базой данных ... 319
14.3 Кейс — сайт на React к этому API ... 322
14.4 Кейс — мобильное приложение ... 326
14.5 Кейс — скрипт на один вечер ... 329

Приложения ... 333
А Шаблоны запросов ... 333
Б Чек-лист ревью кода ... 336
В Частые ошибки Python ... 337
Г Частые ошибки JS, TypeScript и Dart ... 340
Д Git — сценарии спасения ... 343
Е Терминал — необходимый минимум ... 346
Ж Глоссарий ... 348

Предметный указатель ... 353

ЧАСТЬ I

Основа

Три навыка, которые не зависят ни от языка, ни от библиотеки, ни от того, какой моделью ты пользуешься: ставить задачу, читать чужой код и проверять результат. Эта часть — ядро книги. Если прочитать только её, пользы будет больше, чем от любой из следующих по отдельности.

1 Ментальная модель и рабочий цикл

Прежде чем разбираться в приёмах, надо договориться о главном — кто в этой паре за что отвечает и в каком порядке делается работа. Всё остальное в книге опирается на эту главу.

1.1 Кто здесь главный

роли человека и ИИ, и что бывает, когда они путаются

ЧТО ЭТО

У работы с ИИ есть устройство, которое стоит проговорить вслух, потому что по умолчанию оно размывается.

Ты — архитектор и ревьюер. ИИ — исполнитель.

Ты решаешь, что должно получиться и подходит ли то, что получилось. ИИ предлагает, как это сделать, и делает. Решение остаётся за тобой — не потому, что ИИ глуп, а потому, что отвечать за результат будешь ты.

ЧТО ЭТО ЗНАЧИТ НА ПРАКТИКЕ

Архитектор решает, что строить
Какую задачу вообще решаем. Какие части у проекта. Где хранятся данные. Что важнее — скорость или простота. Это решения, которые нельзя делегировать: у ИИ нет ни твоих ограничений, ни твоих планов, ни знания, зачем всё это затевалось.

Ревьюер решает, что принять
Код не становится твоим от того, что он появился в твоём файле. Он становится твоим, когда ты его прочитал и согласился. До этого момента в проекте лежит чужое решение, которое ты не проверял.

Исполнитель делает и предлагает
ИИ хорошо пишет код, подсказывает варианты, объясняет чужие решения и замечает то, что ты пропустил. Он может — и должен —

спорить и предлагать лучше (статья 3.11). Но последнее слово не за ним.

> **ЗАПОМНИ**
> Отвечать за код перед пользователями, сервером и собой через месяц будешь ты. Авторство ИИ в этом ничего не меняет.

ЧТО БЫВАЕТ, КОГДА РОЛИ ПУТАЮТСЯ

Две крайности, и обе встречаются постоянно.

Ты становишься исполнителем
Выглядит так: получил код, не понял, вставил, запустил. Не заработало — попросил исправить. Опять не понял. Повторил.

Это та самая петля без чтения из статьи 1.2. Формально работа идёт, но решения принимает не тот, кто за них отвечает. Через месяц в проекте код, который никто не читал, и баг, который некому найти.

Признак: ты не можешь объяснить, что делает файл, который лежит в твоём проекте.

ИИ становится архитектором
Выглядит иначе: задача ставится общо — «сделай бота для напоминаний», — и ИИ принимает за тебя все решения. Какая база, какая структура, какие библиотеки, как хранятся данные. Всё это зафиксируется в проекте, и менять потом будет дорого.

Проблема не в том, что решения плохие. Обычно они нормальные — средние по индустрии. Проблема в том, что они не твои, а ты узнаешь о них через месяц, когда понадобится что-то поменять.

Признак: в проекте есть библиотека, о которой ты не знаешь, зачем она.

КАК СКАЗАТЬ ИИ

Роли задаются не настройками, а тем, как ставится задача. Архитектор говорит, что строить:

ТАК ЛУЧШЕ
```
Хранить будем в SQLite, не в JSON — записей будет много
и нужны выборки по дате. ORM не используем, запросы
пишем руками, их немного.
```

Исполнителю оставляется то, в чём он силён:

ТАК ЛУЧШЕ

```
Предложи структуру таблицы для подписок. Нужны user_id,
город и дата создания. Скажи, какие индексы стоит добавить
и почему.
```

И отдельный приём — попросить вернуть тебе роль, если чувствуешь, что решения принимаются без тебя:

ЗАПРОС

```
Какие решения ты принял за меня в этом коде?
Перечисли списком, с вариантами, которые были.
```

НА ЧТО СМОТРЕТЬ

Ты давно не спорил с ИИ. Если все предложения принимаются, скорее всего, они не читаются. Нормальная работа — это несколько «нет, давай иначе» на сессию.

Ты не знаешь, почему в проекте такое решение. Любая часть, про которую нельзя сказать «я выбрал это, потому что…», — чужая.

Задачи формулируются как «сделай хорошо». Так говорит заказчик, а не архитектор. Архитектор называет ограничения.

Правки принимаются без чтения диффа. Самый надёжный признак, что роль ревьюера вакантна (статья 2.13).

ШПАРГАЛКА

- Ты — архитектор и ревьюер, ИИ — исполнитель.
- Архитектор решает, что строить; ревьюер — что принять.
- Код становится твоим, когда ты его прочитал и согласился.
- Не можешь объяснить файл в своём проекте — ты стал исполнителем.
- Есть библиотека, назначения которой ты не знаешь, — ИИ стал архитектором.
- Спроси: «какие решения ты принял за меня?»

1.2 Рабочий цикл

задача → описание → генерация → чтение → проверка → итерация

ЧТО ЭТО

Рабочий цикл — это порядок шагов, в котором задача превращается в работающий код. Шесть шагов, и они повторяются до тех пор, пока результат не устроит.

Цикл стоит знать наизусть, потому что почти все проблемы с ИИ — это пропущенный шаг, а не плохая модель. Дальше в книге каждая

глава разбирает один из шагов подробно, и к этому списку мы будем возвращаться постоянно.

Шесть шагов

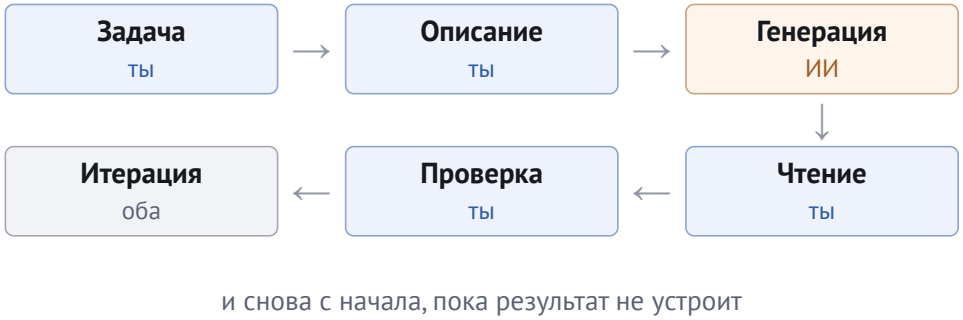


Рис. 1.1 · Рабочий цикл. Синим — то, что делаешь ты, оранжевым — ИИ. Из шести шагов четыре твои, и генерация среди них самая короткая.

#	Шаг	Что на выходе
1	Задача	понимание, что должно получиться
2	Описание	запрос или мини-ТЗ
3	Генерация	код
4	Чтение	понимание, что он написал
5	Проверка	запуск, тест, ответ «работает или нет»
6	Итерация	следующий запрос или готовая фича

Распространённое заблуждение — что работа с ИИ это шаг 2 и шаг 3. На деле это шаги 1, 2, 4, 5, а генерация — самая короткая и самая простая часть.

НА ЧТО СМОТРЕТЬ

Главная ошибка выглядит так: получил код → не понял → запустил → не работает → «не работает, исправь» → получил код → не понял → запустил…

Это **петля без чтения**. Она ощущается как работа, потому что что-то постоянно происходит, но каждый круг не добавляет понимания. Через пять кругов у тебя проект, в котором ты не разбираешься, и баг, который никто не может найти.

Вырваться из петли можно только шагом 4. Читать код необязательно целиком — достаточно понять, что он делает и где. Но пропускать этот шаг нельзя.

ЗАПОМНИ
Если ты не можешь объяснить в двух предложениях, что делает код, — его нельзя оставлять в проекте, даже если он работает.

КАК СКАЗАТЬ ИИ

Каждому шагу соответствует свой тип запроса. Это не отдельные приёмы — это один цикл, разложенный на реплики.

ЗАПРОС
```
Шаг 2, описание:
Добавь в bot.py команду /weather. Она принимает название города,
запрашивает погоду у api.openweathermap.org и отвечает одной строкой:
температура и состояние. Ключ брать из переменной окружения
OWM_API_KEY. Если города нет — ответить «Город не найден».
```

ЗАПРОС
```
Шаг 4, чтение:
Объясни построчно, что делает функция, которую ты добавил.
Отдельно скажи, что произойдёт, если API ответит ошибкой.
```

ЗАПРОС
```
Шаг 5, проверка:
Какие три случая стоит проверить руками, чтобы убедиться,
что команда работает правильно?
```

Третий запрос недооценивают. ИИ хорошо знает, где его собственный код хрупкий, — надо только спросить.

ШПАРГАЛКА
- Шесть шагов: задача → описание → генерация → чтение → проверка → итерация.
- Четыре шага из шести — твои. Генерация самая простая часть.
- Петля без чтения — главный способ потерять контроль над проектом.
- Не можешь объяснить код в двух предложениях — не оставляй его.

1.3 Что ИИ делает хорошо, а что плохо

и почему по его тону этого не определить

ЧТО ЭТО

У ИИ есть сильные и слабые места, и они довольно предсказуемы. Знать их важнее, чем кажется: половина разочарований возникает от того, что задачу дали не в ту сторону.

Но главное в этой статье другое. **Тон ответа не зависит от того, насколько ИИ прав.** Он одинаково уверенно пишет типо-

вую функцию, которую видел тысячу раз, и выдумывает метод библиотеки, которого не существует.

КАК ВЫГЛЯДИТ В КОДЕ

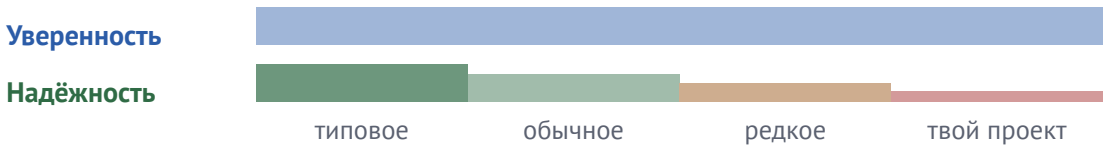


Рис. 1.2 · Уверенность ИИ не меняется, надёжность — меняется. Чем дальше задача от типовой, тем шире расхождение между тем, как звучит ответ, и тем, каков он на самом деле.

Где ИИ силён

Что	Почему
Типовой код: CRUD, парсинг, форматы	такого в обучении было очень много
Обвязка и рутина: конфиги, аргументы, логирование	скучно человеку, легко машине
Перевод между языками	структура сохраняется, меняется синтаксис
Объяснение чужого кода	задача на понимание, а не на изобретение
Первый черновик	быстрее, чем с пустого листа, даже если переписывать
Поиск своих же опечаток	когда указать, где искать

Общее у всего списка: задача распространённая, правильный ответ один, и проверить его легко.

Где ИИ слаб

Что	Почему
Твой проект и твои данные	он их не видел (статья 3.1, причина 2)
Свежие версии библиотек	в обучении больше старого кода
Редкие библиотеки	мало примеров — больше выдумок
Решения с компромиссами	не знает, что для тебя важнее
Всё, что требует помнить предыдущие решения	в новой сессии контекста нет
Оценка собственной работы	защищает свой код (статья 3.8)

> **ВНИМАНИЕ**
> Обрати внимание, чего нет в списке слабых мест: «сложные задачи». Сложность — не главный фактор. ИИ отлично справляется со сложным типовым кодом и ошибается в простом, но редком. Решает распространённость, а не трудность.

Почему уверенность не помогает

ИИ не сообщает, что он не уверен, — ни тоном, ни оговорками. Ответ про библиотеку, которую он знает назубок, и ответ про библиотеку, которую он видел дважды, звучат одинаково.

Отсюда практическое следствие: **степень проверки определяешь ты, по характеру задачи, а не по тому, насколько убедительно звучит ответ.**

КАК СКАЗАТЬ ИИ

Уверенность можно вытащить прямым вопросом — на него ИИ отвечает честнее, чем подразумевает тоном:

ЗАПРОС
```
Насколько ты уверен в этом коде? Отдельно скажи, какие места
ты бы проверил в первую очередь и почему.
```

Для незнакомой библиотеки стоит спросить прямо:

ЗАПРОС
```
Ты уверен, что метод session.fetch_all() существует
в aiohttp 3.9? Если не уверен — скажи, и я проверю
в документации.
```

Такая формулировка работает: прямо разрешив сказать «не уверен», ты получаешь гораздо более честный ответ, чем если спросить «этот метод существует?».

НА ЧТО СМОТРЕТЬ

Чем специфичнее задача, тем внимательнее читай. Типовой код можно просмотреть по диагонали; код для редкой библиотеки — построчно.

Уверенный тон на редкой теме — повод насторожиться, а не расслабиться.

«Это распространённая практика» — не аргумент. Распространённая где? В твоём проекте может быть иначе, и ИИ об этом не знает.

Задача на компромисс отдана ИИ. «Что лучше — SQLite или Postgres?» без указания твоих условий даст средний ответ, а не твой.

ШПАРГАЛКА

- Решает распространённость задачи, а не её сложность.
- Силён: типовой код, рутина, перевод, объяснение, черновик.
- Слаб: твой проект, свежие версии, редкие библиотеки, компромиссы.
- Тон ответа не зависит от того, насколько ИИ прав.
- Степень проверки выбираешь ты, по характеру задачи.
- Разреши сказать «не уверен» — и услышишь честный ответ.

1.4 Три навыка

понимать, формулировать, проверять

ЧТО ЭТО

Вся книга устроена вокруг трёх навыков. Это не красивая схема, а устройство содержания: главы 2, 3 и 4 — это и есть три навыка, по главе на каждый.

Навык	Что значит	Глава
Понимать	читать код, знать термины, видеть структуру проекта	2
Формулировать	ставить задачу так, чтобы вышло задуманное	3
Проверять	убеждаться, что получилось, ловить ошибки, откатывать	4

Остальные главы — приложения к этим трём: процесс и инструменты (5), архитектура (13), практикум (14), разделы по стекам.

ЗАЧЕМ ОНИ НУЖНЫ ВМЕСТЕ

Навыки не независимы. Каждый опирается на остальные, и провал в одном рушит работу целиком.

Без понимания не работает проверка. Нельзя убедиться, что код делает нужное, не читая его. Запуск проверяет ровно один случай — тот, который ты запустил.

Без формулирования понимание тратится впустую. Ты видишь, что решение не то, но не можешь объяснить, какое нужно, — и следующая попытка будет такой же.

Без проверки оба первых навыка бесполезны. Прочитал, сформулировал, получил код — и не запустил. Всё остальное было зря.

Поэтому рабочий цикл из статьи 1.2 и выглядит как цикл: три навыка применяются по очереди на каждом круге.

С ЧЕГО НАЧИНАТЬ

Если развивать их по одному, порядок такой:

Первым — формулирование. Оно даёт самую быструю отдачу: хороший запрос экономит час с первого же применения, а научиться ему можно за вечер. Глава 3 написана первой именно поэтому.

Вторым — понимание. Самый большой навык и самый медленный в освоении, но без него упираешься в потолок: можно бесконечно улучшать запросы, но пока ты не читаешь результат, контроль не появится.

Третьим — проверка. Осваивается быстро, когда есть первые два. Отчасти это просто набор привычек: запускать, смотреть дифф, прогонять тесты.

> **СОВЕТ**
> Если непонятно, какой навык проседает, посмотри, на чём ты теряешь время. Переделываешь по три раза — формулирование. Боишься трогать свой же код — понимание. Ловишь баги у пользователей, а не у себя — проверка.

КАК СКАЗАТЬ ИИ

Навыки можно развивать прямо в работе, добавляя к обычным задачам по одной фразе.

Для понимания:

ЗАПРОС

```
Прежде чем я приму этот код: объясни построчно,
что он делает. Я хочу разобраться, а не просто вставить.
```

Для формулирования — обратная связь на собственные запросы:

ЗАПРОС

```
Я поставил задачу вот так: <запрос>. Что в этой формулировке
было неясно? Что тебе пришлось додумать?
```

Для проверки:

ЗАПРОС

```
Какие три случая стоит проверить руками, чтобы убедиться,
что это работает? Включи хотя бы один неудачный.
```

Второй запрос особенно полезен. Он превращает каждую неудачную попытку в урок: вместо раздражения получаешь конкретный список того, чего не хватало.

НА ЧТО СМОТРЕТЬ

Перекос в формулирование. Выглядит как коллекционирование приёмов: всё новые и новые способы спросить, но результат по-прежнему не читается. Потолок наступает быстро.

Перекос в понимание. Разбираешь каждую строку, включая те, что не собирался менять. Чтение — средство, а не цель; достаточного уровня хватает (статья 2.1).

Проверка откладывается «на потом». Самый дорогой перекос: ошибки копятся и находятся все сразу, когда искать их труднее всего.

ШПАРГАЛКА

- Три навыка: понимать · формулировать · проверять. Главы 2, 3, 4.
- Без понимания невозможна проверка; без проверки бесполезны оба первых.
- Начинать с формулирования — самая быстрая отдача.
- Понимание — самый медленный навык и самый важный.
- Переделываешь трижды — формулирование. Боишься трогать код — понимание.
- Спрашивай: «что в моей формулировке пришлось додумать?»

1.5 Цена ошибки

где можно доверять, а где нельзя

ЧТО ЭТО

Проверять всё одинаково тщательно невозможно и не нужно. Вопрос в том, по какому признаку решать, где проверять бегло, а где построчно.

Распространённый ответ — «по сложности задачи» — неверный. Правильный признак другой: **сколько стоит ошибка, если она случится.** А стоимость складывается из двух вещей: насколько легко откатить и насколько быстро заметишь.

КАК ЭТО СЧИТАТЬ

Обратимость

Насколько просто вернуть как было.

Легко откатить	Трудно откатить
правка в коде под git	миграция базы, изменившая данные
новая функция, которую никто не вызывает	отправленные пользователям сообщения
локальный эксперимент	удалённые файлы
изменение, покрытое тестами	опубликованный релиз, который уже скачали

Git делает обратимым почти всё, что касается кода, — поэтому он в книге отдельной статьёй (5.1). Необратимо то, что вышло за пределы репозитория: данные, письма, деньги, публикации.

Заметность

Насколько быстро ошибка даст о себе знать.

Заметишь сразу	Можешь не заметить месяцами
не запускается	неверные числа в отчёте
падает с ошибкой	потерянная часть записей при импорте
кнопка не нажимается	неправильный часовой пояс
тесты краснеют	утечка, которая проявится под нагрузкой

Правая колонка опаснее левой. Ошибка, которая роняет программу, честна: она сообщает о себе немедленно. Ошибка, которая молча портит данные, обнаружится тогда, когда испорчено уже всё.

Вместе

	Заметишь сразу	Можешь не заметить
Легко откатить	бегло: запусти и посмотри	внимательно: читай логику
Трудно откатить	внимательно: читай дифф	построчно + план + тесты

Правый нижний угол — миграции, рассылки, работа с деньгами и чужими данными. Там обязательны и план (статья 3.10), и мини-

ТЗ (3.15), и тесты (3.9), и чтение диффа (2.13). Левый верхний — то, что можно просто запустить.

> **ЗАПОМНИ**
> Опаснее всего не сложное, а незаметное. Код, который падает, ты почините сегодня. Код, который молча округляет не туда, ты будешь искать через полгода — по жалобе пользователя.

КАК СКАЗАТЬ ИИ

Оценку можно переложить на ИИ, и он с ней справляется неплохо:

ЗАПРОС
```
Прежде чем делать: что в этой задаче необратимо
и что может сломаться незаметно? Коротко.
```

Для всего, что меняет данные, есть отдельный вопрос:

ЗАПРОС
```
Эта правка меняет существующие данные? Если да —
как откатить, если окажется, что она неверна?
```

Если ответа на второй вопрос нет, задачу нельзя начинать: сначала способ отката, потом правка.

ЗАПРОС
```
Сделай так, чтобы миграция была обратимой: сохрани старый
файл, а не удаляй. Переименование вместо удаления.
```

НА ЧТО СМОТРЕТЬ

Необратимое выполняется без репетиции. Миграции, массовые правки, рассылки — всё это сначала прогоняют на копии. «Режим без записи» (`--dry-run`), который только печатает, что собирался сделать, стоит одной просьбы и спасает регулярно.

Удаление вместо переименования. `rm` и `DROP` необратимы, переименование — нет. Это самый дешёвый способ сделать операцию откатываемой.

Нет способа заметить. Если у правки нет ни теста, ни лога, ни видимого эффекта — ты узнаешь о её ошибке последним.

Доверие по привычке. Пять удачных правок подряд не делают шестую безопасной. Уровень проверки определяется задачей, а не предыдущими успехами.

ШПАРГАЛКА

- Проверка настраивается по цене ошибки, а не по сложности.
- Цена = обратимость × заметность.
- Опаснее всего незаметное: оно обнаружится, когда испорчено всё.
- Правый нижний угол (необратимо + незаметно) — план, ТЗ, тесты, дифф.
- Нет способа откатить — не начинай, сначала придумай откат.
- Переименовывай вместо удаления.

1.6 Иллюзия готового кода

почему «работает» не значит «правильно»

ЧТО ЭТО

Код от ИИ выглядит законченным. Ровные отступы, осмысленные имена, аннотации типов, аккуратные комментарии. Он выглядит как код, который кто-то продумал.

Это ощущение обманчиво, и обманчиво оно по трём причинам. Разобрать их стоит в начале книги, потому что дальше вся работа строится на том, чтобы им не поддаваться.

ТРИ ИЛЛЮЗИИ

Иллюзия первая: выглядит аккуратно

Опрятность кода ничего не говорит о его правильности. Человек, который пишет небрежно, обычно пишет небрежно везде, и по виду кода можно было что-то предположить. У ИИ этой связи нет: он одинаково аккуратно оформляет и верное решение, и неверное.

Более того, аккуратность работает против тебя — она усыпляет. Код с кривыми отступами читают внимательно; код, который выглядит как из учебника, пролистывают.

Иллюзия вторая: запустилось

Запуск проверяет ровно один случай — тот, который ты запустил. Обычно это удачный путь: город нашёлся, данные пришли, пользователь ввёл что ожидалось.

```
temp = data.get("main", {}).get("temp", 0)
```

Эта строка запустится всегда. На корректных данных она даёт верный ответ, на ошибке сервера — «Сейчас 0 градусов». Программа работает. Программа врёт.

Именно поэтому в книге столько внимания уделяется неудачным случаям: пустым данным, обрывам сети, мусору на входе. Там живут ошибки, которые запуск не находит.

Иллюзия третья: ИИ уверен

Тон ответа не зависит от того, насколько ИИ прав (статья 1.3). Он не говорит «я не уверен», не ставит оговорок и не выделяет места, где угадывал. Объяснение несуществующего метода звучит так же убедительно, как описание того, который он знает назубок.

> **ЗАПОМНИ**
> Три признака, которым нельзя верить: красиво оформлено, запустилось, звучит уверенно. Ни один из них не говорит о правильности — все три говорят только о том, как это выглядит.

ЧТО ВМЕСТО

Книга предлагает заменить ощущение «вроде нормально» на четыре конкретных вопроса. Это те же вопросы, что в статье 2.1, и они закрывают все три иллюзии:

1. **Что эта функция принимает и что возвращает?**
2. **Меняет ли она что-то снаружи себя?**
3. **Что будет, если данные придут не те?**
4. **Где она вызывается?**

Третий вопрос — противоядие от второй иллюзии. Он заставляет подумать о случаях, которые ты не запускал.

КАК СКАЗАТЬ ИИ

Иллюзию уверенности пробивает прямой вопрос:

ЗАПРОС

```
Назови три места в этом коде, где он сломается
при неожиданных данных. Если таких мест нет — объясни,
почему их нет.
```

Иллюзию запуска — вопрос о непроверенном:

ЗАПРОС

```
Я запустил и получил верный ответ. Какие случаи
мой запуск не проверил?
```

А иллюзию аккуратности — просьба показать компромиссы:

ЗАПРОС

```
Какие решения в этом коде спорные? Что ты выбрал
не задумываясь, потому что так принято?
```

Последний вопрос часто вытаскивает самое интересное: решения, которые ИИ принял по привычке, а не потому, что они подходят твоему случаю.

НА ЧТО СМОТРЕТЬ

Ты принял код, не задав ни одного вопроса. Нормальная работа — это несколько уточнений на каждую нетривиальную правку.

Проверка свелась к запуску. Запустил, ответ верный, принял. Так проверяется один сценарий из десяти.

Комментарии вместо кода. `# TODO: обработать пустой список` — это дыра, оформленная так, будто о ней подумали (статья 2.14).

Ощущение, что всё слишком гладко. Если задача была нетривиальной, а решение пришло без единой оговорки и без вопросов, — скорее всего, часть сложности просто не замечена.

ШПАРГАЛКА

- Аккуратное оформление не говорит о правильности и усыпляет внимание.
- Запуск проверяет один случай — обычно удачный.
- Уверенный тон не зависит от того, прав ли ИИ.
- Вместо ощущения — четыре вопроса: принимает · возвращает · меняет · где вызывается.
- Спроси: «какие случаи мой запуск не проверил?»
- Слишком гладко на сложной задаче — сложность не замечена.

2 Чтение кода

Главный навык всей книги. Пока ты не можешь прочитать то, что написал ИИ, ты не можешь ни проверить его, ни возразить ему. Читать — не значит уметь написать то же самое с нуля.

2.1 Зачем читать код, если его пишет ИИ

читать и уметь написать — разные навыки

ЧТО ЭТО

Главное недоразумение этой главы звучит так: «раз код пишет ИИ, читать его не обязательно». Обязательно — и вот почему.

Читать и писать — разные навыки, и читать намного легче. Чтобы написать функцию с нуля, нужно держать в голове синтаксис, библиотеку и порядок действий. Чтобы прочитать готовую, достаточно понимать, что делает каждая строка. Второе осваивается в разы быстрее и даёт почти всю пользу.

ЗАЧЕМ ЭТО НУЖНО НА ПРАКТИКЕ

Без чтения не работают три вещи из рабочего цикла.

Проверка. Ты не можешь убедиться, что код делает то, что нужно, если не понимаешь, что он делает. Запуск проверяет только тот случай, который ты запустил.

Спор. Когда ИИ предлагает решение, которое тебе не нравится, нужны аргументы по существу. «Мне кажется, тут что-то не то» он отобьёт; «эта функция перезаписывает входной список, а он используется дальше» — нет.

Владение. Через месяц кто-то — скорее всего, ты сам — откроет этот файл. Если код никто не читал при написании, разбираться придётся с нуля, причём в худший момент: когда что-то сломалось.

> **ЗАПОМНИ**
> Ты отвечаешь за код, который не читал, ровно так же, как за тот, который написал сам. Авторство ИИ ничего не меняет: отвечать перед пользователями, сервером и собой через месяц будешь ты.

КАКОГО УРОВНЯ ЧТЕНИЯ ДОСТАТОЧНО

Не нужно понимать каждый символ. Нужно уметь ответить на четыре вопроса:

Вопрос	Зачем
Что эта функция принимает и что возвращает?	понять, как ей пользоваться
Меняет ли она что-то снаружи себя?	предсказать последствия
Что будет, если данные придут не те?	найти дыры до того, как их найдут пользователи
Где она вызывается?	понять, что сломается при правке

Если на все четыре есть ответ — код прочитан достаточно. Разбирать, как именно реализован алгоритм внутри, обычно не требуется.

КАК СКАЗАТЬ ИИ

Чтение — не обязательно чтение в одиночку. ИИ хорошо объясняет свой же код, если спросить правильно (статья 3.7):

ЗАПРОС

```
Объясни построчно, что делает эта функция. Отдельно скажи:
что она принимает, что возвращает и что меняет снаружи себя.
```

Полезно и обратное — читать самому, а проверять у ИИ:

ЗАПРОС

```
Я прочитал так: функция берёт список заказов, считает сумму
и пишет результат в лог. Если сумма ноль — ничего не пишет.
Я что-то упустил?
```

Второй способ быстрее учит: ты сначала думаешь сам, потом сверяешься. Ошибки в понимании вылезают сразу и запоминаются.

НА ЧТО СМОТРЕТЬ

«Выглядит правильно» — не чтение. Код, написанный ИИ, почти всегда выглядит правильно: ровные отступы, понятные имена, аккуратная структура. Внешняя опрятность ничего не говорит о том, что он делает.

Длинный код читать не обязательно целиком. Начни с сигнатур функций и точки входа. Если по ним понятно, как всё связано, внутрь можно не лезть, пока не понадобится.

Непонятное место — не повод сдаться. Нормально не узнать конструкцию: декоратор, генератор, `async with`. Спроси, что это, — в этой главе на каждую такую вещь есть своя статья.

ШПАРГАЛКА

- Читать и писать — разные навыки. Читать осваивается быстрее.
- Без чтения не работают проверка, спор и владение кодом.
- Достаточный уровень: что принимает · что возвращает · что меняет снаружи · где вызывается.
- «Выглядит правильно» не равно «прочитано».
- Читай сам, сверяй у ИИ — так быстрее учишься.

2.2 Анатомия файла

импорты, константы, определения, точка входа

ЧТО ЭТО

Почти любой файл с кодом устроен одинаково: сверху то, что нужно для работы, в середине определения, внизу запуск. Зная эти четыре зоны, в незнакомом файле ориентируешься за полминуты, не читая его целиком.

Порядок не случаен: он совпадает с порядком, в котором код выполняется при импорте файла.

КАК ВЫГЛЯДИТ В КОДЕ

Импорты	что берём снаружи: модули, библиотеки
Константы	настройки и значения, которые не меняются
Определения	функции и классы — пока только описаны
Точка входа	что произойдёт при запуске файла

Рис. 2.1 · Четыре зоны файла. Сверху вниз — в том же порядке, в каком код выполняется при запуске.

Вот файл целиком — все четыре зоны по порядку:

```python
"""Бот напоминаний. Один файл целиком."""
import os
import sqlite3

DB_PATH = os.environ.get("DB_PATH", "reminders.db")
MAX_TEXT = 200


def get_conn() -> sqlite3.Connection:
    conn = sqlite3.connect(DB_PATH)
    conn.row_factory = sqlite3.Row
    return conn


def add_reminder(user_id: int, text: str) -> int:
    if len(text) > MAX_TEXT:
        raise ValueError(f"текст длиннее {MAX_TEXT} символов")
    with get_conn() as conn:
        cur = conn.execute(
            "INSERT INTO reminders (user_id, text) VALUES (?, ?)",
            (user_id, text),
        )
        return cur.lastrowid


def main() -> None:
    ...


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

Что читать в каждой зоне

Импорты отвечают на вопрос «от чего зависит этот файл». `import sqlite3` — работа с базой. Увидел `import requests` — файл ходит в сеть. Это первое, что стоит прочитать: по импортам понятно, чем файл вообще занимается.

Константы — это настройки, вынесенные наверх, чтобы их было легко найти. `DB_PATH = os.environ.get("DB_PATH", "reminders.db")` читается так: взять значение из переменной окружения, а если её нет — использовать `reminders.db`.

Определения — самая длинная зона и самая безобидная. Ключевое: `def` только **описывает** функцию, но не запускает её. Файл можно импортировать, и ни одна строка внутри `def` не выполнится.

Точка входа — строки `if __name__ == "__main__":`. Читается так: «если этот файл запустили напрямую, а не импортировали — выполни то, что ниже».

> **СОВЕТ**
> Начинай чтение незнакомого файла с конца. Точка входа показывает, что вообще происходит при запуске, и оттуда видно, какие функции важны, а какие вспомогательные.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Разбери этот файл по зонам: что в импортах, какие константы,
какие функции определены и что выполняется при запуске.
Коротко, по одной строке на пункт.
```

Для незнакомого проекта полезнее другой вопрос:

ЗАПРОС

```
С какой функции в этом файле начать чтение, чтобы понять,
что он делает? Не объясняй весь файл, просто скажи,
с чего начать и почему.
```

НА ЧТО СМОТРЕТЬ

Код вне функций. Всё, что написано на верхнем уровне и не является импортом, константой или определением, выполнится при импорте файла. Строка `conn = sqlite3.connect(DB_PATH)` вне функции означает, что база откроется от самого факта импорта, — и это почти всегда ошибка.

Нет точки входа, а файл запускают. Если `if __name__ == "__main__":` отсутствует, но файл предполагается запускать, где-то внизу найдётся голый вызов — и он сработает при импорте тоже.

Константы, которые не константы. Имя заглавными буквами обещает, что значение не меняется. Если ИИ написал `MAX_TEXT = 200`, а потом где-то `MAX_TEXT = get_limit()`, это сбивает с толку всех, включая следующего ИИ.

Секреты в константах. `TOKEN = "123456:ABC-DEF"` прямо в файле — токен попадёт в git и останется в истории навсегда. Правильно — из переменной окружения (статья 5.7).

ШПАРГАЛКА

- Четыре зоны: импорты · константы · определения · точка входа.
- Импорты читай первыми: по ним понятно, чем файл занимается.
- `def` только описывает функцию, не выполняет её.
- `if __name__ == "__main__":` — «если файл запустили напрямую».
- Незнакомый файл читай с конца, от точки входа.
- Код вне функций выполняется при импорте — почти всегда ошибка.

2.3 Анатомия проекта

где что лежит и что можно не открывать

ЧТО ЭТО

Проект — это десятки файлов, но читать надо не все. Большинство из них либо служебные, либо сгенерированы инструментами. Статья учит отличать одно от другого и находить, откуда начинается выполнение.

Навык нужен каждый раз, когда открываешь чужой или свой забытый проект, — и каждый раз, когда надо сказать ИИ, в каком файле работать.

КАК ВЫГЛЯДИТ В КОДЕ

```
weather-bot/
├── .git/                  история, трогать руками не надо
├── .venv/                 установленные библиотеки
├── .env                   секреты: токены, пароли
├── .gitignore             что не попадает в git
├── requirements.txt       список зависимостей
├── README.md              с чего начать
├── bot.py                 точка входа
├── config.py              чтение настроек
├── db.py                  подключение к базе
├── storage.py             запросы к базе
├── handlers/              обработчики команд
│   ├── __init__.py
│   ├── start.py
│   └── weather.py
├── migrations/            изменения структуры БД
└── tests/                 тесты
```

Что читать, а что нет

Читать	Не читать
точку входа (`bot.py`, `main.py`, `app.py`)	`.venv/`, `node_modules/`, `vendor/`

`README.md`	`.git/`
папки с логикой (`handlers/`, `services/`)	файлы блокировок: `poetry.lock`, `package-lock.json`
`requirements.txt` — чем пользуется проект	сгенерированное: `dist/`, `build/`, `__pycache__/`

Правило простое: **если файл создан инструментом, а не человеком, читать его не надо.** Папка `.venv` может содержать сотни тысяч строк — там нет ни одной твоей.

Как найти точку входа

Три способа, по убыванию скорости:

1. **По имени.** `main.py`, `bot.py`, `app.py`, `manage.py`, `index.js`, `main.dart` — в девяти случаях из десяти это она.
2. **По README.** Там обычно написана команда запуска, а в ней — имя файла.
3. **Поиском.** `if __name__ == "__main__"` для Python, `"scripts"` в `package.json` для JavaScript.

Имена, которые что-то значат

Некоторые названия устойчивы и встречаются во многих проектах:

Имя	Что внутри
`handlers/`, `routes/`, `views/`	что происходит в ответ на действие пользователя
`models/`, `schemas/`	описание данных: какие поля, каких типов
`services/`, `core/`	основная логика, не привязанная к интерфейсу
`utils/`, `helpers/`	мелкие вспомогательные функции
`migrations/`	изменения структуры базы, применяются по порядку
`tests/`	тесты
`static/`, `assets/`	картинки, стили, шрифты

Папка `utils/` — особый случай: туда складывают всё, чему не нашлось места. Если она разрослась, это признак, что структуру пора пересматривать.

> **СОВЕТ**
> Файл `__init__.py` в папке Python чаще всего пустой и нужен только чтобы папка считалась пакетом. Если он непустой — загляни: там могут быть импорты, которые выполняются при обращении к пакету.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Покажи дерево файлов проекта без .venv, .git и __pycache__.
Для каждой папки первого уровня — одна строка,
что в ней лежит.
```

Когда надо понять, с чего начинать:

ЗАПРОС

```
Я открыл этот проект впервые. Назови три файла, которые
надо прочитать, чтобы понять, как он устроен. По порядку.
```

И самое практичное — перед постановкой задачи:

ЗАПРОС

```
Мне надо добавить команду /stats. В каких файлах
придётся что-то менять? Пока ничего не пиши.
```

Ответ на последний вопрос заодно проверяет, правильно ли ИИ понял структуру проекта.

НА ЧТО СМОТРЕТЬ

Файл `.env` в git. Проверяется одной строкой: есть ли `.env` в `.gitignore`. Если нет — токены уже в истории репозитория.

Логика в точке входа. Если `bot.py` на пятьсот строк, значит, разделения нет, и проект будет тяжело менять. Для ИИ это тоже плохо: он не сможет работать с частью, не задевая остальное.

Две папки для одного. `utils/` и `helpers/`, `services/` и `core/` в одном проекте — признак того, что код складывали, не глядя. Спроси у ИИ, чем они отличаются: если он не сможет объяснить, их надо объединять.

Пустая `tests/`. Папка есть, тестов нет. Значит, при правках никто не узнает, что сломалось (статья 3.9).

ШПАРГАЛКА

- Читать: точку входа, README, папки с логикой, список зависимостей.
- Не читать: всё, что создано инструментом, а не человеком.
- Точку входа ищи по имени, по README или по `if __name__`.
- Устойчивые имена: `handlers/`, `models/`, `services/`, `migrations/`.
- Перед задачей спроси у ИИ, какие файлы она затронет.
- `.env` обязан быть в `.gitignore`.

2.4 Переменные, типы, структуры данных

четыре контейнера, из которых собрано почти всё

ЧТО ЭТО

Почти любые данные в программе лежат в одной из четырёх структур: словарь, список, кортеж или множество. Зная, чем они отличаются, читаешь код гораздо быстрее — по одному виду скобок понятно, что происходит.

Отдельно разберём аннотации типов (`text: str`): они в коде от ИИ встречаются постоянно и читаются проще, чем кажутся.

КАК ВЫГЛЯДИТ В КОДЕ

```
user = {"id": 7, "name": "Ola", "cities": ["Tønsberg", "Oslo"]}

print(user["name"])
print(user.get("email"))
print(user.get("email", "не указан"))

cities = user["cities"]
cities.append("Bergen")
print(user["cities"])          # изменился и оригинал

point = (59.27, 10.41)
lat, lon = point
print(lat, lon)

seen = {"Oslo", "Bergen", "Oslo"}
print(sorted(seen), len(seen))
```

Вывод:

```
Ola
None
не указан
['Tønsberg', 'Oslo', 'Bergen']
59.27 10.41
['Bergen', 'Oslo'] 2
```

Четыре структуры

Вид	Скобки	Для чего	Признак в коде
Словарь	`{"ключ": значение}`	данные с именами полей	обращение `x["name"]`
Список	`[1, 2, 3]`	упорядоченный набор	обращение по номеру `x[0]`, `.append()`
Кортеж	`(1, 2)`	набор, который не меняют	распаковка `a, b = x`
Множество	`{1, 2, 3}`	уникальные значения	проверка `if x in seen`

Фигурные скобки означают и словарь, и множество — различить легко: если внутри есть двоеточия, это словарь.

Две строки, которые надо заметить

```
print(user.get("email"))           # None
print(user.get("email", "не указан"))
```

`user["email"]` при отсутствии ключа падает с `KeyError`. `user.get("email")` возвращает `None` молча. Разница принципиальная и постоянно встречается в коде от ИИ: `.get()` — частый способ «починить» ошибку, спрятав её (статья 2.14).

```
cities = user["cities"]
cities.append("Bergen")
print(user["cities"])   # ['Tønsberg', 'Oslo', 'Bergen']
```

`cities` — не копия, а то же самое. Списки и словари передаются по ссылке: изменил один — изменились оба. Это источник целого класса ошибок, которые выглядят как магия.

Аннотации типов

```
def add_reminder(user_id: int, text: str) -> int:
```

Читается слева направо: `user_id` — целое число, `text` — строка, функция возвращает целое число. Это подсказка для читателя, а

не проверка: Python аннотации не навязывает, и передать строку вместо числа ничто не помешает.

Расшифровка того, что встречается чаще всего:

Запись	Значит
`str`, `int`, `float`, `bool`	строка, целое, дробное, да/нет
`list[str]`	список строк
`dict[str, int]`	словарь: ключи-строки, значения-числа
`str | None`	строка **или** ничего
`-> None`	функция ничего не возвращает

Запись `str | None` особенно полезна при чтении: она прямо предупреждает, что результат надо проверять.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Какие структуры данных используются в этом файле
и что в каждой лежит? Коротко, по строке на переменную.
```

Полезный вопрос, когда данные приходят извне:

ЗАПРОС
```
Покажи пример данных, которые придут в эту функцию.
Один удачный случай и один с недостающими полями.
```

Когда нужно понять, не изменится ли что-то неожиданно:

ЗАПРОС
```
Эта функция меняет переданный ей список или создаёт новый?
```

НА ЧТО СМОТРЕТЬ

`.get()` там, где нужна ошибка. Если поле обязано быть, его отсутствие — это ошибка, а не повод подставить `None`. Молчаливый `None` всплывёт позже и дальше от причины.

Изменение того, что передали. Функция получила список и добавила в него элемент — снаружи список тоже изменился. Если это не задумано, будут трудные баги.

Словарь вместо понятной структуры. `data["result"]["items"][0]["price"]` — признак того, что данные никто не разобрал. Каждое звено здесь может упасть, и трейсбек покажет только последнее.

Аннотации не соответствуют коду. `-> str`, а функция иногда возвращает `None`. Аннотации ничего не проверяют, поэтому расходиться с кодом могут незаметно.

ШПАРГАЛКА

- Четыре структуры: словарь · список · кортеж · множество.
- Фигурные скобки с двоеточиями — словарь, без — множество.
- `x["k"]` падает при отсутствии ключа, `x.get("k")` возвращает `None`.
- Списки и словари передаются по ссылке: изменил один — изменились оба.
- `str | None` в аннотации — результат надо проверять.
- Аннотации ничего не проверяют, они для читателя.

2.5 Функции

вход, выход и то, что происходит вдобавок

ЧТО ЭТО

Чтобы прочитать функцию, не обязательно разбирать её внутренности. Достаточно ответить на три вопроса: что приходит, что уходит и что меняется снаружи.

Третий вопрос — самый важный и самый пропускаемый. Функция может ничего не возвращать и при этом перевернуть полпрограммы.

КАК ВЫГЛЯДИТ В КОДЕ



Рис. 2.2 · Функцию читают по трём направлениям. Побочные эффекты — то, что происходит вдобавок, и то, что обычно забывают проверить.

```python
# Чистая функция: тот же вход -> тот же выход, ничего больше
def format_price(amount: float, currency: str = "NOK") -> str:
    return f"{amount:.2f} {currency}"


# Побочный эффект: меняет то, что ей передали
def add_tax(items: list[dict], rate: float) -> None:
    for item in items:
        item["price"] = round(item["price"] * (1 + rate), 2)


# Побочный эффект: пишет наружу
def log_order(order_id: int) -> None:
    with open("orders.log", "a", encoding="utf-8") as f:
        f.write(f"order {order_id}\n")
```

Чистая функция

`format_price` принимает число и строку, возвращает строку. Вызови её сто раз с теми же аргументами — сто раз получишь одно и то же, и больше ничего не произойдёт.

Такие функции проверять легче всего: достаточно сравнить вход и выход.

Функция с побочным эффектом

`add_tax` возвращает `None`, но работу делает — меняет словари в переданном списке. Вот что из этого следует:

```python
cart = [{"name": "кофе", "price": 100.0}]
add_tax(cart, 0.25)
print(cart)     # [{'name': 'кофе', 'price': 125.0}]
add_tax(cart, 0.25)
print(cart)     # [{'name': 'кофе', 'price': 156.25}]
```

Второй вызов начислил налог на налог. Функция не защищена от повторного вызова, и сама она об этом не скажет — ни по имени, ни по сигнатуре.

> **ВНИМАНИЕ**
> `-> None` в сигнатуре — повод насторожиться, а не расслабиться. Функция, которая ничего не возвращает, обязана делать что-то снаружи — иначе зачем её вызывать.

Где искать побочные эффекты

Пробегись глазами по телу функции и ищи четыре вещи:

Признак	Что происходит
присваивание в то, что пришло аргументом	меняет данные вызывающего

`open()`, запись в файл	меняет диск
`conn.execute()`, `INSERT`, `UPDATE`	меняет базу
`requests`, `await session.post()`	меняет что-то на другом сервере
`global`, присваивание в переменную модуля	меняет состояние программы

Если ни одного признака нет и есть `return` — функция чистая, и жизнь проще.

Аргументы по умолчанию

```python
def format_price(amount: float, currency: str = "NOK") -> str:
```

`currency` можно не передавать, тогда будет `"NOK"`. Читается легко, но есть ловушка, которую ИИ воспроизводит регулярно:

```python
def add_item(item, cart=[]):    # так нельзя
    cart.append(item)
    return cart
```

Список в значении по умолчанию создаётся **один раз** — и остаётся общим для всех вызовов. Второй вызов `add_item("хлеб")` вернёт список из двух элементов. Правильно — `cart=None`, а внутри `if cart is None: cart = []`.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Для этой функции ответь на три вопроса:
что принимает, что возвращает, что меняет снаружи себя.
По одной строке на вопрос.
```

ЗАПРОС

```
Что произойдёт, если вызвать эту функцию дважды подряд
с теми же аргументами?
```

Второй вопрос ловит ровно ту категорию ошибок, что в примере с налогом.

НА ЧТО СМОТРЕТЬ

Функция и возвращает, и меняет. Делает две вещи сразу — её трудно переиспользовать и легко вызвать не вовремя. Обычно стоит разделить.

Имя обещает не то. `get_user` обязана только получать. Если внутри `get_user` создаётся запись в базе, имя врёт, и следующий читатель на этом ошибётся.

Изменяемое значение по умолчанию. `=[]`, `={}` в аргументах — почти всегда ошибка.

Слишком много аргументов. Больше пяти — признак, что функция делает несколько дел. Проверь, не стоит ли её разбить.

ШПАРГАЛКА

- Три вопроса: что принимает · что возвращает · что меняет снаружи.
- Чистая функция: тот же вход — тот же выход, и больше ничего.
- `-> None` значит, что вся работа в побочных эффектах.
- Побочные эффекты ищи по: записи в аргумент, файлам, базе, сети, `global`.
- `=[]` и `={}` в аргументах по умолчанию — ошибка.
- Спроси: «что будет, если вызвать дважды подряд?»

2.6 Условия и циклы

читаем поток управления

ЧТО ЭТО

Поток управления — это порядок, в котором выполняются строки. По умолчанию сверху вниз, но условия и циклы его меняют: какие-то строки пропускаются, какие-то повторяются.

Чтобы прочитать такой код, надо уметь отвечать на один вопрос: **при каких данных выполнится эта строка?** Остальное прикладывается.

КАК ВЫГЛЯДИТ В КОДЕ

```
orders = [
    {"id": 1, "status": "paid", "total": 250},
    {"id": 2, "status": "pending", "total": 0},
    {"id": 3, "status": "paid", "total": 90},
]

total = 0
for order in orders:
    if order["status"] != "paid":
        continue
    if order["total"] <= 0:
        print(f"заказ {order['id']}: оплачен, но сумма нулевая")
        continue
    total += order["total"]
print("итого:", total)
```

Вывод:

```
итого: 340
```

Как это читать

Отступ — это область действия. Всё, что сдвинуто вправо относительно `for` или `if`, относится к нему. В Python отступ не оформление, а синтаксис: сдвинул строку — изменил смысл программы.

`continue` **означает «пропустить остаток и взять следующий элемент».** Две такие строки в примере — это фильтры: заказы не оплаченные и с нулевой суммой до сложения не доходят.

`break` **означает «выйти из цикла совсем».** Путать их — частая ошибка при чтении: `continue` пропускает один шаг, `break` прекращает перебор.

`else` **у цикла** — редкая конструкция, но встречается. Выполняется, если цикл завершился без `break`.

Списковые включения

Тот же смысл можно записать одной строкой:

```
paid = [o["total"] for o in orders if o["status"] == "paid"]
print(sum(paid))     # 340
```

Читается справа налево по смыслу: *взять* `o` *из* `orders`*, где статус* `paid`*, и собрать из них* `o["total"]`. ИИ пишет такие конструкции охотно — они короткие, но при двух-трёх условиях становятся нечитаемыми.

> **СОВЕТ**
> Если спотыкаешься о списковое включение, попроси переписать его обычным циклом: «перепиши это списковое включение как цикл for, чтобы было видно порядок действий». Смысл не изменится, а читаться будет легче.

Тернарное выражение

```
status = "оплачен" if order["total"] > 0 else "ожидает"
```

Читается так: `status` равно `"оплачен"`, если условие верно, иначе `"ожидает"`. Та же конструкция в JavaScript и Dart пишется иначе:

```
const status = order.total > 0 ? "оплачен" : "ожидает";
```

Знак `?` перед первым вариантом, `:` перед вторым — это одно и то же, просто другой синтаксис.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
При каких данных выполнится строка 14? Назови конкретный
пример входных данных, при которых мы до неё дойдём.
```

ЗАПРОС
```
Этот цикл может не выполниться ни разу? При каких данных?
```

Второй вопрос важнее, чем кажется. Цикл по пустому списку просто ничего не делает — молча, без ошибки. Если дальше код рассчитывает, что в переменной что-то накопилось, он получит ноль или пустоту.

НА ЧТО СМОТРЕТЬ

Глубокая вложенность. Три и больше уровня `if` внутри `for` внутри `if` — такой код трудно читать и легко сломать. Признак, что часть условий можно вынести в ранний `continue` или в отдельную функцию.

Цикл, который может не выполниться. Пустой список, пустой ответ API — и весь блок пропущен. Проверь, что дальше по коду это учтено.

Изменение списка во время перебора. Удаление элементов из того же списка, по которому идёт `for`, даёт пропущенные элементы. Выглядит безобидно, работает неправильно.

Условие, которое всегда истинно или всегда ложно. Бывает после правок: `if status != "paid" or status != "pending"` — такое условие верно всегда. Чтобы заметить, достаточно подставить любое значение.

ШПАРГАЛКА

- Отступ в Python — синтаксис, а не оформление.
- `continue` — пропустить шаг, `break` — выйти из цикла.
- Списковое включение читается как цикл: взять, отфильтровать, собрать.
- Главный вопрос: при каких данных выполнится эта строка?
- Второй вопрос: может ли цикл не выполниться ни разу?
- Не меняй список, пока идёшь по нему циклом.

2.7 Классы и объекты

что это и когда они действительно нужны

ЧТО ЭТО

Класс — это описание того, как устроен объект: какие у него данные и что он умеет делать. Объект — конкретный экземпляр, созданный по этому описанию.

Для чтения кода достаточно понимать четыре вещи: `class`, `__init__`, `self` и точку. Всё остальное — надстройки над этим.

КАК ВЫГЛЯДИТ В КОДЕ

```
class Cart:
    def __init__(self, user_id: int) -> None:
        self.user_id = user_id
        self.items: list[dict] = []

    def add(self, name: str, price: float) -> None:
        self.items.append({"name": name, "price": price})

    def total(self) -> float:
        return sum(item["price"] for item in self.items)


cart = Cart(user_id=7)
cart.add("кофе", 100.0)
cart.add("хлеб", 45.0)
print(cart.total())      # 145.0
print(cart.user_id)      # 7
```

Четыре вещи, которые надо знать

`class Cart:` — описание. Как и `def`, само по себе ничего не запускает.

`__init__` — выполняется в момент создания объекта. Строка `cart = Cart(user_id=7)` вызывает именно его. Здесь задаются начальные данные.

`self` — сам объект. Первый аргумент любого метода, но при вызове его не передают: `cart.add("кофе", 100.0)` — два аргумента, хотя в определении их три. Python подставляет `self` сам.

Точка — обращение к данным или методам объекта: `cart.items`, `cart.total()`. Со скобками — вызов метода, без скобок — чтение данных.

Что отличает класс от набора функций

Класс хранит состояние между вызовами. `cart.items` живёт в объекте: добавил товар — он там и остался. С обычными функциями этот список пришлось бы передавать туда-сюда вручную.

Отсюда правило чтения: **ищи, где меняется состояние.** Всё, что начинается с `self.` и стоит слева от `=`, — изменение объекта. Это те же побочные эффекты из статьи 2.5, только внутри класса.

Dataclass — класс только для данных

```
from dataclasses import dataclass

@dataclass
class Item:
    name: str
    price: float
    qty: int = 1


item = Item("кофе", 100.0)
print(item)              # Item(name='кофе', price=100.0, qty=1)
```

`@dataclass` сам создаёт `__init__` и красивый вывод. Встречается часто, и это хороший знак: класс без методов, только данные, читается легко.

> **СОВЕТ**
> Когда ИИ предлагает класс, спроси: «можно ли то же самое сделать функциями?» Часто ответ «да, и будет проще». Класс оправдан, когда есть состояние, которое живёт между вызовами; без состояния он только добавляет слоёв.

Наследование

```
class PremiumCart(Cart):
    def total(self) -> float:
        return super().total() * 0.9
```

`PremiumCart(Cart)` значит «то же самое, что `Cart`, но с изменениями». `super().total()` — вызов версии из родителя.

При чтении наследование неудобно тем, что код разъезжается по двум местам: чтобы понять, что делает `PremiumCart`, надо открыть и `Cart`. Поэтому глубже двух уровней оно почти всегда зло.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Что хранит этот объект между вызовами и какие методы
это состояние меняют?
```

ЗАПРОС

```
Этот класс оправдан или то же самое проще сделать
двумя функциями? Ответь прямо.
```

Когда класс унаследован:

ЗАПРОС

```
Какие методы PremiumCart переопределяет, а какие берёт
у родителя? Список.
```

НА ЧТО СМОТРЕТЬ

Класс без состояния. Если в `__init__` ничего не сохраняется, а все методы могли бы быть обычными функциями, класс здесь лишний. ИИ создаёт такие классы охотно — они выглядят «правильно».

`self.` присваивается вне `__init__`. Поле, которое появляется в середине другого метода, легко пропустить при чтении: объект внезапно обзаводится данными, которых раньше не было.

Глубокое наследование. Три уровня и больше — чтобы понять одну строку, придётся открыть четыре файла.

Изменяемое значение в поле класса. То же, что с аргументами по умолчанию (статья 2.5): список, объявленный на уровне класса, а не в `__init__`, будет общим для всех объектов.

ШПАРГАЛКА

- Четыре вещи: `class` · `__init__` · `self` · точка.
- `__init__` выполняется при создании объекта.
- `self` при вызове не передают, Python подставляет сам.
- Класс нужен ради состояния между вызовами; без него хватит функций.
- Ищи `self.x = ...` — это изменения объекта.
- Наследование глубже двух уровней почти всегда вредит чтению.

2.8 Исключения и обработка ошибок

где ошибка возникает и где её ловят

ЧТО ЭТО

Исключение — способ сообщить, что дальше работать нельзя. Строка `raise` поднимает ошибку, блок `try/except` её ловит. Между

ними выполнение перепрыгивает через весь код, который был между.

Читать этот механизм надо по двум вопросам: **где ошибка возникает** и **где её ловят**. Между ними может быть десяток функций — именно это и показывает трейсбек (статья 2.12).

КАК ВЫГЛЯДИТ В КОДЕ

```
def parse_port(raw: str) -> int:
    try:
        port = int(raw)
    except ValueError:
        raise ValueError(f"порт должен быть числом, получено: {raw!r}")
from None
    if not 1 <= port <= 65535:
        raise ValueError(f"порт вне диапазона: {port}")
    return port


for raw in ["8080", "abc", "99999"]:
    try:
        print(raw, "->", parse_port(raw))
    except ValueError as e:
        print(raw, "-> ошибка:", e)
```

Вывод:

```
8080 -> 8080
abc -> ошибка: порт должен быть числом, получено: 'abc'
99999 -> ошибка: порт вне диапазона: 99999
```

Как читать

`try:` — «попробуй выполнить». Если внутри что-то упало, выполнение немедленно переходит в `except`, остаток `try` пропускается.

`except ValueError:` — ловит только ошибки этого вида. Другие пролетят мимо и уйдут выше.

`raise` — поднять ошибку. Здесь функция превращает невнятное `invalid literal for int()` в понятное сообщение со значением, которое не подошло.

`as e` — поместить объект ошибки в переменную, чтобы прочитать текст.

`finally:` — выполнится в любом случае, с ошибкой или без. Нужен для закрытия файлов и соединений.

Иерархия ошибок

Ошибки образуют дерево: `ValueError` и `KeyError` — частные случаи `Exception`. Поэтому `except Exception` ловит почти всё, а `except ValueError` — только своё.

Вид	Когда бывает
`ValueError`	значение не того вида: `int("abc")`
`KeyError`	нет такого ключа в словаре
`TypeError`	не тот тип: сложили строку с числом
`AttributeError`	нет такого поля или метода (часто у `None`)
`FileNotFoundError`	файла нет
`TimeoutError`	не дождались ответа

Эти шесть покрывают большинство случаев, которые встретятся при чтении.

> **ВНИМАНИЕ**
> `except Exception: pass` — самая вредная строка в Python. Она ловит всё подряд и молчит: программа продолжает работать с неверными данными, а причина не всплывёт нигде. Увидев такое в коде от ИИ, спроси, какую именно ошибку он здесь ожидал.

Ловить надо там, где знаешь, что делать

Правило, по которому проверяют расстановку `try`: ошибку ловят там, где есть осмысленный ответ на вопрос «а что теперь?».

В примере `parse_port` не ловит собственную ошибку диапазона — она не знает, что делать с неверным портом. А вызывающий код знает: напечатать сообщение и перейти к следующему значению.

Поэтому обёртка `try` вокруг каждой строки — признак, что автор не думал о том, кто и как будет на ошибку реагировать.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Какие исключения может поднять эта функция и где они ловятся?
Если нигде — так и скажи.
```

ЗАПРОС

```
Здесь стоит except Exception. Какие конкретно ошибки
ты ожидал поймать? Сузь до них.
```

Второй запрос — рабочий способ чинить слишком широкие `except`: ИИ обычно может назвать конкретные виды, если спросить.

НА ЧТО СМОТРЕТЬ

`except Exception: pass` — ошибка проглочена целиком. Код не падает и поэтому кажется рабочим.

`except` **без типа.** Голое `except:` ловит даже прерывание по Ctrl+C.

Ошибка поймана, но не записана. Если в `except` нет ни `log`, ни повторного `raise`, информация потеряна навсегда.

Сообщение без данных. `raise ValueError("неверное значение")` — какое именно? Хорошее сообщение всегда содержит то, что не подошло.

`try` **вокруг двадцати строк.** Непонятно, какая из них может упасть. Сужение `try` до одной-двух строк — самый дешёвый способ сделать код читаемым.

ШПАРГАЛКА

- `raise` поднимает, `try/except` ловит, между ними выполнение прыгает.
- `except ValueError` ловит только своё, `except Exception` — почти всё.
- Шесть частых ошибок: Value · Key · Type · Attribute · FileNotFound · Timeout.
- Лови там, где знаешь, что делать дальше.
- `except Exception: pass` — ошибка проглочена, код врёт.
- В сообщении об ошибке всегда указывай значение, которое не подошло.

2.9 Декораторы и «магия»

почему код выполняется не сверху вниз

ЧТО ЭТО

Декоратор — строка со знаком `@` над функцией. Она означает: эту функцию оборачивают в другую, и при вызове сначала выполнится обёртка.

Знать это нужно по простой причине: **из-за декораторов код выполняется не в том порядке, в котором написан.** Для того, кто читает код, это главный источник недоумения.

КАК ВЫГЛЯДИТ В КОДЕ

```python
import functools
import time


def timed(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        t = time.perf_counter()
        result = fn(*args, **kwargs)
        print(f"{fn.__name__}: {time.perf_counter() - t:.3f} c")
        return result
    return wrapper


@timed
def slow_sum(n: int) -> int:
    return sum(range(n))


print(slow_sum(1_000_000))
```

Вывод:

```
slow_sum: 0.016 c
499999500000
```

Сообщение о времени напечаталось **раньше** результата, хотя в коде `print` стоит после. Так работает обёртка: вызов `slow_sum(...)` на самом деле вызывает `wrapper`, тот запускает настоящую функцию, замеряет время, печатает и только потом возвращает результат.

Как читать

`@имя` над функцией — функцию обернули. Чтобы понять, что произойдёт при вызове, надо посмотреть на `имя`, а не только на саму функцию.

`*args, **kwargs` — «любые аргументы». Обёртка не знает, что принимает исходная функция, поэтому принимает всё и передаёт дальше.

`functools.wraps` — служебная деталь: сохраняет у обёртки имя и описание исходной функции. Без неё `slow_sum.__name__` вернул бы `wrapper`, и трейсбеки стали бы непонятными.

Декораторы, которые встретятся чаще всего

Декоратор	Что делает

`@router.message(...)` (aiogram)	«вызывай эту функцию на такое сообщение»
`@app.get("/path")` (FastAPI)	«вызывай на такой HTTP-запрос»
`@dataclass`	создаёт `__init__` и вывод за тебя
`@property`	метод, который читается как поле, без скобок
`@staticmethod`	метод, которому не нужен `self`
`@functools.cache`	запоминает результат, при том же аргументе не считает заново

Первые два — самые важные для практики. Они **регистрируют** функцию: ты её нигде не вызываешь, но она вызовется, когда придёт нужное событие.

```
@router.message(Command("weather"))
async def cmd_weather(message: Message) -> None:
    ...
```

Поиском по проекту вызов `cmd_weather` не найдётся. Это нормально: её вызывает фреймворк, а строка с `@` — заявка на это.

> **ЗАПОМНИ**
> Если функция нигде не вызывается, но работает — посмотри на строку над ней. Почти всегда там декоратор, который её зарегистрировал.

Другие языки

То же самое есть везде, под другими именами:

```
// JavaScript: обёртка явная, без синтаксиса @
const timedSum = withTiming(slowSum);
```

```
// Dart: аннотации @override, @immutable — чаще подсказки,
// чем обёртки
```

В React похожую роль играют хуки: `useEffect` не вызывается тобой напрямую — React вызывает его сам в нужный момент.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> `Что делает декоратор @timed с функцией slow_sum?`
> `Объясни, в каком порядке выполнятся строки при вызове.`

Когда непонятно, кто вызывает функцию:

> **ЗАПРОС**

```
Эта функция нигде не вызывается в коде. Кто её вызовет
и когда?
```

НА ЧТО СМОТРЕТЬ

Порядок декораторов. Если их несколько, они применяются снизу вверх — ближайший к функции первым. Перестановка меняет поведение.

Декоратор молча глотает ошибки. Обёртка с `try/except` внутри — ошибки исходной функции могут не дойти до тебя (статья 2.8).

Обёртка меняет аргументы или результат. Прочитай, что именно передаётся в `fn(...)` и что возвращается: обёртка может подменить и то и другое.

Нет `functools.wraps`. Мелочь, но из-за неё в трейсбеках вместо имён функций будет `wrapper`, и отладка станет труднее.

ШПАРГАЛКА

- `@имя` над функцией — её обернули, смотри на обёртку.
- Из-за декораторов код выполняется не сверху вниз.
- `*args, **kwargs` — «принимаю что угодно и передаю дальше».
- `@router.message`, `@app.get` регистрируют функцию: её вызовет фреймворк.
- Функция нигде не вызывается, но работает — ищи декоратор над ней.
- Несколько декораторов применяются снизу вверх.

2.10 Асинхронность

async, await и зачем это боту

ЧТО ЭТО

Асинхронность — способ не простаивать, пока программа чего-то ждёт. Запрос к API идёт полсекунды, и всё это время обычный код просто стоит. Асинхронный — успевает заняться другими делами.

Для бота это принципиально: пока один пользователь ждёт погоду, остальные должны получать ответы. Для чтения кода достаточно понять два слова: `async` и `await`.

КАК ВЫГЛЯДИТ В КОДЕ

```
import asyncio
import time


async def fetch_city(name: str) -> str:
    await asyncio.sleep(0.3)          # как будто запрос в сеть
    return f"{name}: +4"


async def main() -> None:
    t = time.perf_counter()
    for city in ["Oslo", "Bergen", "Tromsø"]:
        print(await fetch_city(city))
    print(f"по очереди: {time.perf_counter() - t:.1f} с")

    t = time.perf_counter()
    results = await asyncio.gather(
        *(fetch_city(c) for c in ["Oslo", "Bergen", "Tromsø"])
    )
    print(f"разом: {time.perf_counter() - t:.1f} с")


asyncio.run(main())
```

Вывод:

```
по очереди: 0.9 с
разом: 0.3 с
```

Три запроса по 0,3 секунды: по очереди — 0,9, одновременно — 0,3. Это и есть вся польза асинхронности, измеренная.

Три слова

`async def` — функция асинхронная. Главное следствие: при вызове она **не выполняется**, а возвращает объект-обещание. Чтобы получить результат, нужен `await`.

`await` — «дождись результата, а пока ждём, займись другими делами». Ставится только внутри `async def`.

`asyncio.run(main())` — запуск. Точка, с которой начинается асинхронный мир.

Где выигрыш, а где нет

Асинхронность помогает там, где программа **ждёт**: сеть, диск, база. Там, где программа **считает**, выигрыша нет — процессор занят, и переключаться не на что.

Помогает	Не помогает

запросы к API	перемножение матриц
работа с базой	обработка картинок
чтение и запись файлов	разбор большого текста
ожидание сообщений от Telegram	любые тяжёлые вычисления

Самая частая ошибка — забытый `await`

```
data = fetch_city("Oslo")        # забыли await
print(data)
```

Выведет что-то вроде `<coroutine object fetch_city at 0x...>` и предупреждение `coroutine was never awaited`. Функция не выполнилась вообще.

> **ВНИМАНИЕ**
> `<coroutine object ...>` в выводе или в логах — всегда забытый `await`. Это первое, что надо проверить, если асинхронный код «ничего не делает».

Блокирующий вызов в асинхронном коде

Вторая частая беда — обычная синхронная библиотека внутри `async def`:

```
async def fetch_city(name: str) -> str:
    r = requests.get(url)        # блокирует всё!
    return r.text
```

`requests` синхронный: на время запроса встаёт вся программа, и бот перестаёт отвечать всем. Асинхронные замены: `aiohttp`, `httpx` в асинхронном режиме. То же с базой: `sqlite3` блокирует, `aiosqlite` — нет.

Признак при чтении: внутри `async def` вызов без `await` у чего-то, что ходит в сеть, на диск или в базу.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Есть ли в этом асинхронном коде блокирующие вызовы?
Назови строки и чем их заменить.
```

ЗАПРОС
```
Эти три запроса выполняются по очереди или одновременно?
Если по очереди — покажи, как сделать одновременно.
```

ЗАПРОС

```
Перепиши этот код так, чтобы он не блокировал цикл событий.
Библиотеки менять можно, но скажи, какие добавляешь.
```

НА ЧТО СМОТРЕТЬ

Забытый `await`. Самая частая. Ищется по `coroutine` в выводе.

Синхронная библиотека внутри `async def`. `requests`, `time.sleep`, обычный `open` на большом файле — всё это останавливает программу целиком. `time.sleep(1)` надо заменять на `await asyncio.sleep(1)`.

Последовательные `await` там, где можно разом. Цикл с `await` внутри выполняет запросы по очереди. Если они независимы, их объединяют через `asyncio.gather` — как в примере, втрое быстрее.

Асинхронность там, где не нужна. Если программа ничего не ждёт, `async` только добавляет сложности. ИИ иногда делает асинхронным всё подряд.

ШПАРГАЛКА

- `async def` не выполняется при вызове — нужен `await`.
- `await` работает только внутри `async def`.
- Польза там, где программа ждёт: сеть, диск, база. При вычислениях — нет.
- `<coroutine object ...>` — забыт `await`.
- `requests` и `time.sleep` внутри `async def` блокируют всё.
- Независимые запросы объединяй через `asyncio.gather`.

2.11 Зависимости

что импортируется, откуда и какой версии

ЧТО ЭТО

Зависимости — чужой код, который использует твой проект. Читать их важно по двум причинам: по импортам видно, чем файл занимается, а по версиям — почему код от ИИ может не запускаться.

Вторая причина — та самая причина 3 из статьи 3.1, «он помнит другое». Половина необъяснимых ошибок в сгенерированном коде объясняется версией.

КАК ВЫГЛЯДИТ В КОДЕ

```
import os                          # стандартная библиотека
import sqlite3

import aiohttp                     # внешняя, из requirements
from aiogram import Bot, Router
from aiogram.filters import Command

from config import settings        # своё, из этого же проекта
from storage import add_reminder
```

Три группы, разделённые пустыми строками: стандартная библиотека, внешние пакеты, свой код. Это общепринятый порядок, и по нему сразу видно, что откуда.

Как отличить одно от другого

Признак	Откуда
`os`, `sys`, `json`, `re`, `datetime`, `pathlib`	стандартная библиотека, ставить не надо
есть в `requirements.txt`	внешний пакет
совпадает с именем файла или папки в проекте	свой код

Если не уверен — посмотри в `requirements.txt`. Чего там нет и что не похоже на файл проекта, скорее всего, из стандартной библиотеки.

Две формы импорта

```
import sqlite3              # берём модуль целиком
conn = sqlite3.connect(...)

from aiogram import Bot     # берём одно имя из модуля
bot = Bot(token=...)
```

Разница только в том, как потом пишется обращение. Форма `from ... import *` — плохая: непонятно, какие имена появились в файле, и они могут перекрыть существующие.

Где написаны версии

```
aiogram==3.13.1
aiohttp>=3.9,<4.0
python-dotenv
```

- `==3.13.1` — ровно эта версия.
- `>=3.9,<4.0` — любая из этого диапазона.

- без указания — любая, в том числе вышедшая завтра и несовместимая.

Последний случай — источник ошибок вида «вчера работало». Поэтому в рабочих проектах версии фиксируют.

Узнать, что установлено на самом деле:

```
pip freeze | grep aiogram
```

> **ЗАПОМНИ**
> Прежде чем просить код для библиотеки, посмотри её версию и назови её в запросе. Это одна команда и один из самых выгодных приёмов во всей книге.

Почему версия решает так много

Между мажорными версиями библиотеки ломают совместимость. Примеры, которые встретятся чаще всего:

Библиотека	Что поменялось
aiogram 2 → 3	другой способ регистрации хендлеров, другие импорты
pydantic 1 → 2	другие имена методов, другая валидация
React (классы → хуки)	совсем другой способ писать компоненты
Flutter	меняет API между мажорными версиями

Кода для старых версий в интернете больше, поэтому ИИ по умолчанию тянется к нему.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Вот мой requirements.txt. Пиши код только под эти версии.

aiogram==3.13.1
aiohttp==3.9.5
```

Когда код не запускается:

ЗАПРОС

```
Код не работает: ImportError: cannot import name 'Dispatcher'
from 'aiogram'. У меня aiogram 3.13.1. Под какую версию
написан этот код и как переписать под мою?
```

Перед добавлением новой библиотеки:

ЗАПРОС

```
Можно ли решить это стандартной библиотекой, без новой
зависимости? Если нет — объясни, почему.
```

Последний вопрос стоит задавать всегда. Каждая зависимость — это то, что надо обновлять, что может сломаться и что увеличивает проект.

НА ЧТО СМОТРЕТЬ

Импорт, которого нет в `requirements.txt`. Код не запустится у тебя, даже если у ИИ «работает». Проверяется мгновенно: попробовать запустить.

Выдуманная библиотека. ИИ иногда импортирует пакеты, которых не существует, или методы, которых в пакете нет. Проверка — `pip index versions имя` или просто поиск в документации (статья 4.8).

Новая зависимость ради одной функции. Целый пакет, чтобы отформатировать дату, — почти всегда лишнее.

Импорт внутри функции. Иногда оправдан (медленная библиотека, круговые импорты), но чаще это след от правок. Заметил — спроси, зачем так.

Разъехавшиеся версии. В `requirements.txt` одно, `pip freeze` показывает другое. Значит, окружение собрано не из этого файла, и у тебя с сервером может быть по-разному.

ШПАРГАЛКА

- Три группы импортов: стандартные · внешние · свои.
- Не уверен, откуда — смотри `requirements.txt`.
- `==` фиксирует версию, без указания — любая, и это рискованно.
- Версию библиотеки называй в запросе всегда.
- Половина странных ошибок в коде от ИИ — это версия.
- Перед новой зависимостью спроси, можно ли без неё.

2.12 Чтение трейсбека

находим виноватую строку за тридцать секунд

ЧТО ЭТО

Трейсбек (traceback) — это то, что Python печатает, когда программа упала. Выглядит пугающе, читается за полминуты и почти всегда прямо называет виноватую строку.

Навык полезен вдвойне: ты сразу понимаешь, что случилось, и можешь передать ИИ не «не работает», а точную причину. Второе важнее — от этого зависит, починит он с первого раза или с пятого.

КАК ВЫГЛЯДИТ В КОДЕ

Вот программа, которая запрашивает погоду и показывает температуру:

```
import json


def fetch_weather(city: str) -> dict:
    # Ответ сервера, когда город не найден
    raw = '{"cod": "404", "message": "city not found"}'
    return json.loads(raw)


def format_report(data: dict) -> str:
    temp = data["main"]["temp"]
    return f"Сейчас {temp} градусов"


def handle_command(city: str) -> str:
    data = fetch_weather(city)
    return format_report(data)


if __name__ == "__main__":
    print(handle_command("Tonsberg"))
```

Запускаем — и получаем:

```
Traceback (most recent call last):
  File "/tmp/tbdemo/weather2.py", line 21, in <module>
    print(handle_command("Tonsberg"))
          ~~~~~~~~~~~~~~~^^^^^^^^^^^^
  File "/tmp/tbdemo/weather2.py", line 17, in handle_command
    return format_report(data)
  File "/tmp/tbdemo/weather2.py", line 11, in format_report
    temp = data["main"]["temp"]
           ~~~~^^^^^^^^
KeyError: 'main'
```

Как это читать

Правило первое: читай снизу вверх. Последняя строка — суть, всё остальное — как мы сюда попали.

```
KeyError: 'main'
```

`KeyError` значит: обратились к ключу словаря, которого там нет. Ключ назван прямо — `'main'`.

Правило второе: предпоследний блок — место аварии.

```
  File "/tmp/tbdemo/weather2.py", line 11, in format_report
    temp = data["main"]["temp"]
           ~~~~^^^^^^^^
```

Файл, строка 11, функция `format_report`. Значки под строкой — подсказка Python (с версии 3.11): `^^^^` указывает ровно на ту часть выражения, которая упала. Здесь видно, что проблема в `["main"]`, а не в `["temp"]`. Это важно: без значков пришлось бы гадать, какое из двух обращений виновато.

Правило третье: блоки сверху вниз — это путь вызовов. Программа начала со строки 21, оттуда зашла в `handle_command` (строка 17), оттуда в `format_report` (строка 11), и там упала. Этот путь отвечает на вопрос «откуда взялись данные», а он почти всегда и есть настоящая причина.

Правило четвёртое: ищи свои файлы. В реальном трейсбеке половина строк — из библиотек: `site-packages`, `aiogram`, `asyncio`. Их можно пролистать. Смотри на последнюю строку, где упомянут *твой* файл: ошибка почти всегда там, даже если упало внутри библиотеки.

Что мы поняли

Строка 11 не виновата — она честно делает то, что просили. Виновата строка 6: сервер вернул ошибку «город не найден», а код решил, что получил погоду. **Место падения и причина ошибки — разные места.** Трейсбек показывает первое, а путь вызовов помогает найти второе.

КАК СКАЗАТЬ ИИ

Самая частая ошибка — пересказывать ошибку своими словами.

ТАК НЕ НАДО

```
не работает, вылетает ошибка с ключом
```

ИИ придётся гадать, какой ключ, в каком файле и на какой строке. Он начнёт переписывать всё подряд.

Правильно — отдать трейсбек целиком, добавив то, чего в нём нет: что ты делал и чего ожидал.

ТАК ЛУЧШЕ

```
Падает при запросе погоды для несуществующего города.
Ожидал ответ «Город не найден», получил исключение.

Traceback (most recent call last):
  File "weather.py", line 21, in <module>
    print(handle_command("Tonsberg"))
  File "weather.py", line 17, in handle_command
```

```
    return format_report(data)
  File "weather.py", line 11, in format_report
    temp = data["main"]["temp"]
           ~~~~^^^^^^^^
KeyError: 'main'

Почини так, чтобы ошибка сервера обрабатывалась в fetch_weather,
а не всплывала в format_report.
```

Последняя фраза — важная. Без неё ИИ, скорее всего, обернёт строку 11 в `try/except` или подставит `data.get("main", {})`. Падать перестанет, пользователь получит «Сейчас None градусов», и настоящая ошибка спрячется.

> **ВНИМАНИЕ**
> «Убрать трейсбек» и «починить ошибку» — разные задачи. ИИ по умолчанию делает первое, потому что буквально об этом и просят. Говори, где ошибку надо обработать.

НА ЧТО СМОТРЕТЬ

ИИ глушит ошибку вместо того, чтобы её обработать. Признаки в сгенерированном коде: `except Exception: pass`, `except: return None`, `.get(key, {})` вместо проверки. Код перестаёт падать, но начинает врать.

ИИ правит место падения, а не причину. Если он изменил только ту строку, на которую указывал трейсбек, спроси: «откуда в эту функцию пришли неверные данные?»

Обрезанный трейсбек. Если скопировать только последнюю строку, ИИ не увидит пути вызовов и будет чинить вслепую. Копируй целиком, от слова `Traceback`.

Трейсбек из асинхронного кода длиннее и путанее. В нём будут кадры `asyncio` и внутренности библиотеки. Правило то же: ищи последнюю строку со своим файлом.

ШПАРГАЛКА

- Читай снизу вверх: последняя строка — что случилось.
- Предпоследний блок — файл и строка падения.
- `^^^^` под строкой указывает на виноватую часть выражения.
- Пролистывай чужие файлы, ищи последний свой.
- Место падения ≠ причина. Причину ищи выше по пути вызовов.
- ИИ отдавай трейсбек целиком плюс что делал и чего ожидал.

2.13 Чтение диффа

что именно изменил ИИ

ЧТО ЭТО

Дифф (diff) — список изменений: что удалено, что добавлено. Когда ИИ правит код, читать надо именно дифф, а не файл целиком. Так видно ровно то, что поменялось, и ничего лишнего.

Это главный инструмент приёмки. Весь контроль над проектом держится на привычке смотреть дифф перед тем, как принять правку.

КАК ВЫГЛЯДИТ В КОДЕ

Просили починить `KeyError` при отсутствии города. Вот что пришло:

```
--- a/weather.py
+++ b/weather.py
@@ -2,6 +2,8 @@ import logging

 log = logging.getLogger(__name__)

+DEFAULT_TEMP = 0
+

 def fetch_weather(city: str) -> dict:
     data = call_api(city)
@@ -9,5 +11,5 @@ def fetch_weather(city: str) -> dict:


 def format_report(data: dict) -> str:
-    temp = data["main"]["temp"]
+    temp = data.get("main", {}).get("temp", DEFAULT_TEMP)
     return f"Сейчас {temp} градусов"
```

Как это читать

`-` **в начале строки** — строка удалена. `+` — добавлена. Строка без знака — контекст, она не менялась и показана, чтобы было понятно, где мы.

`@@ -9,5 +11,5 @@` — указатель места: было пять строк начиная с девятой, стало пять начиная с одиннадцатой. Точные числа обычно не нужны, но по ним видно, сдвинулся ли файл.

`--- a/` **и** `+++ b/` — какой файл. При правке нескольких файлов таких блоков будет несколько.

Изменённая строка показывается как удаление плюс добавление. Пара `-`/`+` рядом — это одна строка, которую переписали.

Что мы видим в этом диффе

Изменена одна строка. `data["main"]["temp"]` стало `data.get("main", {}).get("temp", DEFAULT_TEMP)`.

Падать перестанет. Но теперь при ответе «город не найден» пользователь получит **«Сейчас 0 градусов»**. Ошибка не починена — она спрятана, и спрятана так, что выглядит как настоящий ответ.

Это ровно тот случай из статьи 3.5: «убрать ошибку» и «починить причину» — разные задачи. И увидеть это можно только прочитав дифф: запуск покажет, что бот больше не падает, и создаст ощущение, что всё хорошо.

> **ЗАПОМНИ**
> Дифф читают, чтобы найти не то, что сломалось, а то, что изменилось молча. Запуск проверяет один случай; дифф показывает все.

Как смотреть дифф

```
git diff                  # что изменено, но ещё не добавлено в коммит
git diff --staged         # что уже добавлено
git diff --stat           # только список файлов и число строк
git diff -- weather.py    # только один файл
```

Начинай со `--stat`. Одна строка вывода вместо ожидаемой одной — хорошо. Семь файлов там, где ждал один, — сигнал остановиться (статья 3.13).

```
 weather.py | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Порядок чтения

1. `--stat` — сколько файлов и строк. Совпадает с ожиданием?
2. **Удалённые строки** (`-`). Что исчезло? Удаление опаснее добавления: добавленный код хотя бы виден, а удалённую проверку заметить сложно.
3. **Добавленные строки** (`+`). Делают ли они то, что просили?
4. **Что не поменялось, хотя должно было.**

Второй пункт пропускают чаще всего, а он важнее третьего.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Покажи дифф своих изменений, не весь файл.
```

ЗАПРОС
```
В этом диффе: меняется ли поведение программы для случаев,
которые раньше работали? Если да — для каких.
```

ЗАПРОС
```
Ты удалил строку с проверкой длины. Зачем?
```

Второй запрос — хороший способ самопроверки после большой правки, но помни: про свой код ИИ отвечает снисходительно (статья 3.8). Серьёзные диффы читай сам.

НА ЧТО СМОТРЕТЬ

Дифф больше ожидаемого. Правка на две строки с диффом на пятьдесят означает, что сделано что-то ещё.

Переформатирование вперемешку с правкой. Переставленные импорты, изменённые кавычки, переносы строк — настоящая правка тонет среди шума. Запрещать это надо заранее (статья 3.13).

Удалённые проверки. `if not data: return`, `if len(text) > MAX` — такие строки при переписывании теряются первыми и выглядят необязательными.

Молчаливая смена поведения. Как в примере: было исключение, стал ноль. Формально одна строка, по сути другая программа.

Изменения в файлах, о которых не просили. Смотри `--stat` первым делом.

ШПАРГАЛКА

- `-` удалено, `+` добавлено, без знака — контекст.
- Пара `-`/`+` рядом — переписанная строка.
- Начинай со `--stat`: сколько файлов и строк.
- Удалённые строки читай раньше добавленных.
- Ищи не поломку, а молчаливое изменение поведения.
- Дифф больше ожидаемого — остановись и разберись.

2.14 Красные флаги в сгенерированном коде

десять признаков, по которым стоит присмотреться

ЧТО ЭТО

Итоговая статья главы: список того, что в коде от ИИ встречается чаще всего и чаще всего оказывается проблемой. Красный флаг — не приговор, а повод остановиться и прочитать внимательнее.

Все десять опираются на предыдущие статьи, поэтому читаются быстро — это сводка, а не новый материал.

НА ЧТО СМОТРЕТЬ

1. Проглоченные ошибки

```
try:
    data = fetch_weather(city)
except Exception:
    pass
```

Программа не падает и поэтому кажется рабочей. На деле дальше поедут неверные данные, а причина не всплывёт нигде. Сюда же `except: return None` и `.get(key, {})` там, где поле обязано быть (статьи 2.8, 2.13).

2. Значения по умолчанию вместо ошибки

```
temp = data.get("main", {}).get("temp", 0)
```

Ноль градусов выглядит как настоящий ответ. Если поле обязательное, его отсутствие — ошибка, а не повод подставить число.

3. Выдуманные библиотеки и методы

Импорт пакета, которого нет, или вызов метода, которого в библиотеке не существует. Код выглядит уверенно и падает при запуске. Лечится проверкой версии (статья 2.11) и беглым взглядом в документацию.

4. Блокирующий вызов в асинхронном коде

```
async def handler(message):
    r = requests.get(url)       # останавливает всё
```

Бот перестаёт отвечать всем, пока идёт один запрос (статья 2.10).

5. Секреты в коде

```
TOKEN = "7123456789:AAE-abcdef..."
```

Попадёт в git и останется в истории навсегда. Токены — только из переменных окружения.

6. Код вне функций

Строка `conn = sqlite3.connect(DB_PATH)` на верхнем уровне файла выполнится при импорте — не тогда, когда ты ожидаешь, и не один раз (статья 2.2).

7. Изменяемое значение по умолчанию

```
def add_item(item, cart=[]):
```

Список создаётся один раз и остаётся общим для всех вызовов (статья 2.5).

8. Функция делает больше, чем просили

Внутри появились логирование, кеш, повторные попытки, валидация, о которых речи не было. Не обязательно вредно, но это код, который ты не заказывал и не проверял.

9. Молчаливое изменение того, что передали

```
def add_tax(items: list[dict], rate: float) -> None:
    for item in items:
        item["price"] = ...
```

Снаружи данные тоже изменились. Повторный вызов начислит налог на налог (статья 2.5).

10. Комментарии вместо кода

```
# Здесь нужно проверить права доступа
# TODO: обработать случай с пустым списком
return result
```

Комментарий описывает то, чего в коде нет. Выглядит как продуманное решение, а на деле это дыра, оформленная вежливо.

КАК СКАЗАТЬ ИИ

Проверка в один запрос — после получения кода, до принятия:

ЗАПРОС

```
Проверь свой код по списку:
- проглоченные исключения и значения по умолчанию вместо ошибок
- вызовы методов, которых может не быть в этой версии библиотеки
- блокирующие вызовы внутри async
- секреты в коде
- код вне функций
- изменяемые значения по умолчанию в аргументах
- изменение данных, переданных аргументом

Что из этого есть? Если ничего — так и скажи.
```

Такой запрос работает лучше общего «сделай ревью»: он называет, что искать (статья 3.8).

> **ЗАПОМНИ**
> Красный флаг — повод прочитать внимательнее, а не повод переписывать. Каждый из десяти бывает и оправданным; вопрос в том, осознанно ли он там.

ШПАРГАЛКА

- Проглоченные ошибки: `except Exception: pass`, `.get()` вместо проверки.
- Значение по умолчанию вместо ошибки — ответ, который врёт.
- Выдуманные библиотеки и методы — проверяй версию и документацию.
- `requests` и `time.sleep` внутри `async def`.
- Токены в коде — попадут в историю git навсегда.
- Код вне функций выполняется при импорте.
- `=[]` в аргументах, изменение переданных данных, лишняя работа.
- `# TODO` в коде — это дыра, а не решение.

3 Постановка задач

Здесь лежит основная практическая ценность книги. ИИ почти никогда не ошибается «сам по себе» — он точно выполняет то, что услышал. Значит, управлять результатом можно только через запрос.

3.1 Почему ИИ делает «не то»

четыре причины, и все четыре лечатся запросом

ЧТО ЭТО

ИИ почти никогда не ошибается «сам по себе». Он точно выполняет то, что услышал, — просто услышал он не то, что ты имел в виду. Причин ровно четыре, и у каждой свой признак, по которому её видно.

Статья нужна, чтобы не чинить наугад. Получив не тот результат, сначала определи причину, и только потом переписывай запрос: разные причины лечатся по-разному, а лечение не от той причины добавляет к одной ошибке вторую.

КАК ВЫГЛЯДИТ В КОДЕ

Задача одна и та же: добавить в бот команду, которая показывает погоду. Ниже — четыре способа получить «не то».

Причина 1. Ты не сказал
Самая частая. Ты не назвал деталь, ИИ подставил свою.

ТАК НЕ НАДО

```
Добавь команду погоды
```

Ты держал в голове: бот на aiogram, ключ в переменной окружения, ответ одной строкой. ИИ ничего этого не слышал и подставил среднее по больнице: синхронный `requests`, ключ строкой прямо в коде, ответ на пять строк с эмодзи.

Признак: код рабочий, но решение не твоё. Претензий к качеству нет, есть претензия к выбору.

Лечение: назвать то, что было в голове. Четыре части запроса — статья 3.2.

Причина 2. Он не видит

ИИ не знает твоего проекта, если ему не показать. В чате он не видит файлов вообще. Агент в терминале видит, но только то, что успел прочитать.

ТАК НЕ НАДО

```
Сделай так же, как в соседнем хендлере
```

Признак: ИИ выдумывает имена функций, импортирует несуществующие модули, пишет в стиле, которого в проекте нет.

Лечение: показать. Вставить файл, назвать путь, дать команду прочитать. Если объяснять приходится каждый раз — завести файл контекста проекта (статья 3.14).

Причина 3. Он помнит другое

Модель обучена на коде, которого в интернете было больше. А больше всего там кода для старых версий: `aiogram 2` против `aiogram 3`, классовые компоненты React против хуков, старый `Navigator` во Flutter.

ТАК НЕ НАДО

```
Напиши хендлер на aiogram
```

Признак: код выглядит уверенно, но не запускается. В ошибках — `ImportError`, `AttributeError`, «нет такого параметра». Или библиотека ругается, что способ устарел.

Лечение: называть версию. Не `aiogram`, а `aiogram 3`. Не `React`, а `React 18, функциональные компоненты, хуки`.

ВНИМАНИЕ

Это единственная из четырёх причин, которую нельзя вылечить подробностью запроса. Можно описать задачу идеально и всё равно получить код для старой версии — если версию не назвать.

Причина 4. Он услышал буквально

ИИ сделал ровно то, о чём просили. Проблема в том, что просили не то.

ТАК НЕ НАДО

```
Убери эту ошибку
```

Ошибка исчезнет. Скорее всего, её завернут в `try/except` и подавят — формально просьба выполнена. Разница между «убрать ошибку» и «починить причину» для тебя очевидна, для ИИ — нет.

Признак: задача закрыта, симптом ушёл, а проблема осталась. Или ушла вместе с нужным поведением.

Лечение: формулировать цель, а не действие. Не «убери ошибку», а «сделай так, чтобы ошибка сервера обрабатывалась в `fetch_weather`».

КАК СКАЗАТЬ ИИ

Сводная таблица — по ней причина определяется за несколько секунд.

Что видишь	Причина	Что делать
Код рабочий, но решение не твоё	ты не сказал	назвать подробности
Выдуманные имена, чужой стиль	он не видит	показать файлы
Не запускается, `ImportError`	помнит другое	назвать версию
Симптом ушёл, проблема осталась	услышал буквально	назвать цель, не действие

Когда причина непонятна, спроси у самого ИИ — он отвечает на это честно:

ЗАПРОС

```
Прежде чем править: на какие предположения ты опирался,
когда писал этот код? Что из этого я тебе не говорил?
```

НА ЧТО СМОТРЕТЬ

Не чини причину 1 средствами причины 3. Типичная сцена: получил не тот результат, начинаешь дописывать в запрос всё больше подробностей, а код всё так же не запускается — потому что дело было в версии, а не в подробностях.

Одна задача — одна причина за раз. Переписав запрос, меняй что-то одно. Иначе непонятно, что помогло, и в следующий раз придётся гадать заново.

«Он тупой» — не причина. Это ощущение возникает на причине 4, когда ИИ буквально выполнил просьбу. Проверь, что ты просил, а не что имел в виду.

ШПАРГАЛКА

- Четыре причины: не сказал · не видит · помнит другое · услышал буквально.
- Сначала определи причину, потом переписывай запрос.
- Версию называй всегда: подробностями эту причину не вылечить.
- Формулируй цель, а не действие.
- Непонятно почему — спроси, на какие предположения он опирался.

3.2 Анатомия запроса

контекст, цель, ограничения, критерий готовности

ЧТО ЭТО

Хороший запрос состоит из четырёх частей: **контекст**, **цель**, **ограничения** и **критерий готовности**. Это не шаблон, который надо заполнять каждый раз, — это список того, что ИИ должен откуда-то узнать. Если ты не сказал, он догадается, и догадка будет среднестатистической, а не твоей.

Статья — основа всей главы 3. Дальше идут готовые шаблоны под конкретные задачи, и все они построены из этих четырёх частей.

Четыре части

Часть	Отвечает на вопрос	Что будет, если пропустить
Контекст	где и в каком проекте	ИИ напишет код «с нуля», не вписав в твой
Цель	что должно получиться	получишь похожее, но не то
Ограничения	чего делать нельзя	перепишет лишнее, добавит зависимости
Критерий	как понять, что готово	«готово» наступит раньше, чем надо

КАК ВЫГЛЯДИТ В КОДЕ

Возьмём живую задачу: в Telegram-боте надо добавить ограничение на частоту команд.

ТАК НЕ НАДО

```
Добавь в бота защиту от спама
```

Запрос выглядит понятным, но в нём нет ни одной из четырёх частей. Что ИИ придумает за тебя: какой фреймворк, какой лимит,

где хранить счётчики, что отвечать пользователю, считать ли по командам или по сообщениям, трогать ли существующие обработчики. Шесть решений, и все не твои.

Теперь то же самое, но с четырьмя частями:

ТАК ЛУЧШЕ

```
Контекст: бот на aiogram 3, файл handlers/commands.py.
Обработчики — функции с декоратором @router.message.
Данные храню в SQLite через aiosqlite, модуль db.py.

Цель: ограничить частоту команд — не больше 5 команд в минуту
от одного пользователя.

Ограничения:
- не добавляй новых зависимостей, только то, что уже в requirements.txt
- не меняй существующие обработчики, сделай отдельным middleware
- состояние держи в памяти, в базу не пиши

Готово, когда: шестая команда за минуту получает ответ
«Слишком часто, подожди немного», а в логи пишется предупреждение.
Остальные команды работают как раньше.
```

Второй запрос длиннее в десять раз и экономит час переделок.

Как понять, что контекста достаточно

Контекст — то, чего ИИ не может увидеть. Если он работает прямо в проекте (агент в терминале), файлы он прочитает сам, и контекст сокращается до «где смотреть». Если это чат без доступа к коду — контекст придётся дать словами или вставкой файла.

Минимум, который нужен почти всегда:

- язык, фреймворк и **версия** — `aiogram 3`, а не просто `aiogram`
- файл или модуль, где делать;
- как в проекте уже решены похожие задачи.

Третий пункт недооценивают. Он важнее первых двух: именно он заставляет ИИ писать в стиле проекта, а не в своём.

> **СОВЕТ**
> Если объяснять контекст приходится каждый раз заново — его место в файле контекста проекта. Это статья 3.14.

Критерий готовности

Самая пропускаемая часть и самая полезная. Критерий — это проверяемое условие, а не пожелание.

Не критерий	Критерий

«чтобы работало хорошо»	«шестая команда за минуту получает отказ»
«надёжно»	«при обрыве сети повторяет запрос трижды, потом пишет в лог»
«быстро»	«ответ меньше секунды на 100 пользователях»
«красиво»	«кнопки в два ряда, подпись не длиннее 20 символов»

Критерий делает две вещи. Во-первых, ИИ понимает, когда остановиться. Во-вторых, у тебя появляется способ проверить результат, не вчитываясь в код, — ты просто проверяешь условие.

КАК СКАЗАТЬ ИИ

Четыре части не обязаны быть списком. В коротких задачах они умещаются в абзац:

ЗАПРОС

```
В utils/text.py добавь функцию, которая режет длинное сообщение
на куски по 4096 символов по границам абзацев, не разрывая слова.
Внешних библиотек не используй. Готово, когда текст из 10 000 знаков
даёт 3 куска, и склейка кусков равна исходному тексту.
```

Здесь всё четыре: контекст (`utils/text.py`), цель (функция нарезки), ограничение (без библиотек), критерий (10 000 знаков → 3 куска, склейка совпадает).

Для большой задачи те же четыре части разворачиваются в мини-ТЗ — статья 3.15.

> **ЗАПОМНИ**
> Пропущенная часть не исчезает — её заполняет ИИ. Вопрос только в том, устроит ли тебя его вариант.

НА ЧТО СМОТРЕТЬ

Слишком много контекста тоже вредно. Если вывалить весь проект, важное утонет. Давай то, что относится к задаче, и ссылайся на остальное одной строкой.

Ограничения формулируй через «не», а не через «только». «Используй только стандартную библиотеку» ИИ иногда читает как пожелание. «Не добавляй новых зависимостей» работает надёжнее.

Критерий, который нельзя проверить, — не критерий. Если ты сам не знаешь, как проверишь результат, задача сформулирована не до конца. Это сигнал вернуться к шагу 1 рабочего цикла, а не писать запрос дальше.

Версия фреймворка решает больше, чем кажется. `aiogram 2` и `aiogram 3` несовместимы, у React другой подход после хуков, Flutter меняет API между мажорными версиями. Не указал версию — получишь код для той, которой в обучении было больше.

ШПАРГАЛКА

- Четыре части: контекст, цель, ограничения, критерий готовности.
- Контекст — то, чего ИИ не видит. Обязательно: фреймворк, версия, файл.
- Ограничения пиши через «не делай», а не «делай только».
- Критерий должен быть проверяемым: условие, а не прилагательное.
- Не знаешь, как проверишь результат, — задача ещё не поставлена.
- Пропущенную часть ИИ заполнит сам.

3.3 Уровни детализации

когда хватит одной строки, а когда нужно мини-ТЗ

ЧТО ЭТО

Подробный запрос не всегда лучше короткого. На мелкой правке мини-ТЗ — это десять минут, потраченных вместо одной, а на большой задаче короткая строка оборачивается часом переделок.

Выбор уровня определяется не размером задачи, а **ценой ошибки**: сколько стоит получить не то. Статья даёт правило, по которому уровень выбирается за пару секунд.

КАК ВЫГЛЯДИТ В КОДЕ

Три уровня, от короткого к подробному.

Уровень 1. Одна строка
Для правок, которые видно глазом и которые легко откатить.

ЗАПРОС
```
Переименуй get_data в fetch_weather во всём файле.
```

ЗАПРОС
```
Добавь логирование в начало каждого хендлера.
```

Проверка занимает секунды: посмотрел дифф — принял или откатил. Расписывать тут нечего.

Уровень 2. Абзац

Рабочая лошадка. Обычная фича, которую надо вписать в существующий код.

ЗАПРОС

```
В handlers/weather.py добавь кеш ответов погоды на 10 минут,
чтобы не дёргать API на каждый запрос. Храни в памяти, словарём
по названию города. Новых зависимостей не добавляй.
```

Здесь уже есть все четыре части запроса, просто сжатые в три строки. Этого хватает в большинстве случаев.

Уровень 3. Мини-ТЗ

Для задач, которые трогают несколько файлов, меняют структуру данных или которые тяжело откатить.

ЗАПРОС

```
Задача: перевести хранение подписок из JSON-файла в SQLite.

Контекст: сейчас подписки лежат в data/subs.json, читаются
функцией load_subs() в storage.py. Вызовов load_subs и save_subs
в проекте пять штук, все в handlers/.

Что нужно:
- таблица subscriptions: user_id, city, created_at
- те же функции load_subs/save_subs, сигнатуры не меняем,
  чтобы хендлеры трогать не пришлось
- миграция: при первом запуске переносим данные из JSON,
  файл переименовываем в subs.json.bak

Ограничения:
- только стандартный sqlite3, без ORM
- хендлеры не меняем вообще

Готово, когда: бот запускается на пустой базе, подписки из старого
JSON видны после перезапуска, а все пять вызовов работают как раньше.
```

Длинно — и это окупается. Такую задачу с одной строки переделывают по три раза.

КАК СКАЗАТЬ ИИ

Правило выбора: **смотри не на размер задачи, а на то, что будет, если выйдет не то.**

Цена ошибки	Уровень	Признак задачи
Секунды, видно глазом	строка	переименование, форматирование, один вызов

Минуты, легко откатить	абзац	новая функция, правка в одном файле
Полчаса и больше	мини-ТЗ	несколько файлов, структура данных, миграции

Два вопроса, которые решают сомнения:

1. **Смогу ли я проверить результат, не вчитываясь в код?** Если нет — нужен уровень выше, и в первую очередь критерий готовности.
2. **Легко ли откатить?** Один коммит — откатывается легко. Миграция базы — нет.

> **СОВЕТ**
> Сомневаешься между абзацем и мини-ТЗ — попроси план. Пусть ИИ сначала распишет, что собирается делать (статья 3.10). План на десять строк стоит минуты, а показывает, насколько ты и ИИ поняли задачу одинаково.

НА ЧТО СМОТРЕТЬ

Мини-ТЗ на мелкую правку вредит. Чем больше текста, тем больше ИИ «помогает»: переписывает соседние функции, добавляет обработку случаев, о которых не просили. Короткая задача — короткий запрос.

Длина запроса не равна подробности. Три строки с версией фреймворка, именем файла и критерием готовности полезнее двадцати строк рассуждений о том, как было бы хорошо.

Уровень растёт по ходу работы. Начал со строки, получил не то — не дописывай к той же реплике, а сформулируй заново уровнем выше. Переписать дешевле, чем латать.

ШПАРГАЛКА

- Уровень выбирается по цене ошибки, а не по размеру задачи.
- Строка — видно глазом и легко откатить.
- Абзац — обычная фича, рабочий режим.
- Мини-ТЗ — несколько файлов, структура данных, миграции.
- Не можешь проверить результат не читая код — уровнем выше.
- Получил не то — формулируй заново, а не дописывай.

3.4 Шаблон — новая функция

самый частый запрос в работе

ЧТО ЭТО

Новая функциональность — то, что просят у ИИ чаще всего. У этого запроса есть устойчивая форма: назвать место, описать поведение, очертить границы, задать проверку.

Шаблон ниже — не форма для заполнения, а напоминание. Со временем он проговаривается сам, но первое время в него полезно заглядывать.

КАК ВЫГЛЯДИТ В КОДЕ

Шаблон

ЗАПРОС

```
Где: <файл или модуль; как устроено рядом>
Что: <поведение — что на входе, что на выходе>
Границы: <чего не делать; что не трогать>
Готово, когда: <проверяемое условие>
```

Плохо

ТАК НЕ НАДО

```
Нужна функция для работы с подписками пользователей
```

«Работа с подписками» — это и добавить, и удалить, и проверить, и список показать. ИИ выберет сам, и скорее всего напишет класс на двести строк со всеми четырьмя операциями, половина из которых не нужна.

Хорошо

ТАК ЛУЧШЕ

```
Где: storage.py, рядом с load_subs(). Данные в SQLite,
подключение через get_conn() из db.py.

Что: функция add_sub(user_id: int, city: str) -> bool.
Добавляет подписку. Возвращает False, если такая подписка
уже есть, True если добавили.

Границы: только эта функция, остальные не трогай.
Новых зависимостей не добавляй.

Готово, когда: повторный вызов с теми же аргументами
возвращает False и второй строки в таблице не появляется.
```

Результат предсказуем: одна функция, понятная сигнатура, проверяемое поведение.

КАК СКАЗАТЬ ИИ

Что обязательно

Сигнатура. Имя, аргументы с типами, что возвращает. Это самая дешёвая подробность и самая полезная: она отсекает половину разночтений. Не знаешь, как назвать, — опиши словами, ИИ предложит.

Поведение в неудачном случае. Что делать, если данных нет, сеть отвалилась, аргумент пустой. Не скажешь — ИИ выберет сам, и чаще всего выберет молча вернуть `None`.

Границы. Фраза «остальные функции не трогай» экономит больше всего времени. Без неё ИИ охотно «заодно улучшит» соседний код, и дифф разрастается с десяти строк до двухсот.

Что необязательно

Не надо расписывать алгоритм, если он очевиден, — это работа ИИ. Не надо просить комментарии в каждой строке: они замусоривают код и устаревают первыми.

> СОВЕТ
> Если функция будет использоваться в нескольких местах, добавь: «покажи пример вызова». Одна строка примера сразу показывает, удобно ли ей пользоваться, — а неудобную сигнатуру лучше переделать сейчас, чем после десяти вызовов.

Когда функций несколько

Не проси пять функций одним запросом. Попроси первую, прочитай, прими — и дальше остальные одной фразой «сделай так же для `delete_sub` и `list_subs`». ИИ подхватит стиль уже принятого кода, и проверять будет проще.

НА ЧТО СМОТРЕТЬ

Функция делает больше, чем просили. Признак: внутри появились логирование, кеш, повторные попытки, валидация, о которых речи не было. Само по себе не вредно, но это код, который ты не заказывал и не проверял.

Молчаливая обработка ошибок. `return None` вместо исключения, `except: pass`, значения по умолчанию там, где должна быть ошибка. Код не падает и поэтому кажется рабочим — а потом приходится искать, откуда взялся `None`.

Сигнатура удобна ИИ, а не тебе. Семь аргументов, из них пять необязательных, словарь вместо явных параметров. Если вызов читается плохо — переделай сразу.

Функция не вписана в проект. Своё подключение к базе вместо `get_conn()`, свой формат логов, свой стиль именования. Чаще всего это причина 2 из статьи 3.1: ИИ просто не видел, как устроено рядом.

ШПАРГАЛКА

- Четыре строки: где · что · границы · готово, когда.
- Сигнатура — самая дешёвая и самая полезная подробность.
- Всегда говори, что делать в неудачном случае.
- «Остальное не трогай» экономит больше всего времени.
- Несколько функций — по одной, потом «сделай так же».
- Проверь: не делает ли функция больше, чем просили.

3.5 Шаблон — исправить баг

что делал, что ожидал, что получил

ЧТО ЭТО

Запрос на починку отличается от запроса на новую функцию одним: здесь ты даёшь не задание, а **улики**. ИИ не видел, как оно ломалось, и всё, что у него есть, — твоё описание.

Из этого следует главное правило: не пересказывай ошибку своими словами. Отдай то, что видел, дословно, и добавь то, чего в трейсбеке нет, — что ты делал и чего ожидал.

КАК ВЫГЛЯДИТ В КОДЕ

Шаблон

ЗАПРОС

```
Что делал: <действие, которое привело к ошибке>
Что ожидал: <какое поведение считаешь правильным>
Что получил: <вывод, трейсбек, скриншот — дословно>

<трейсбек целиком>

Где чинить: <если знаешь, в каком слое должно лечиться>
```

Плохо

ТАК НЕ НАДО

```
бот падает при погоде, почини
```

ИИ не знает ни города, ни ошибки, ни строки. Он начнёт переписывать весь хендлер целиком, попутно меняя то, что работало.

Хорошо

```
ТАК ЛУЧШЕ
Что делал: отправил боту /weather Тёнсберг
Что ожидал: ответ «Город не найден» — такого города в API нет
Что получил: исключение, бот не ответил вообще

Traceback (most recent call last):
  File "handlers/weather.py", line 34, in cmd_weather
    await message.answer(format_report(data))
  File "handlers/weather.py", line 11, in format_report
    temp = data["main"]["temp"]
           ~~~~^^^^^^^^
KeyError: 'main'

Где чинить: ошибку API надо ловить в fetch_weather,
а не в format_report.
```

КАК СКАЗАТЬ ИИ

Три вопроса, без которых не чинят

Что делал. Шаги воспроизведения. Если ошибка не каждый раз — скажи и это: «падает примерно раз из пяти» меняет направление поиска полностью.

Что ожидал. Без этого ИИ не отличит баг от задуманного поведения. В примере выше именно эта строка говорит, что «Город не найден» — нормальный ответ, а не ошибка.

Что получил. Дословно. Трейсбек целиком, от слова `Traceback`. Как его читать — статья 2.12.

Где чинить — необязательно, но ценно

Если не сказать, ИИ починит там, куда указал трейсбек. А место падения и причина — разные места. Строка 11 честно делает, что просили; виновата строка, где решили, что ответ сервера — это погода.

> **ЗАПОМНИ**
> «Убрать ошибку» и «починить причину» — разные задачи. По умолчанию ИИ делает первое, потому что буквально об этом и просят.

Если трейсбека нет

Бывает, что ничего не падает, а просто работает неправильно. Тогда уликами становятся данные:

ЗАПРОС

```
Что делал: /stats за октябрь
Что ожидал: 142 записи, как в базе
Что получил: 138

Вот запрос, которым считаю: <код>
Вот что в базе: <вывод SELECT COUNT(*)>
```

Два числа, которые не сошлись, — такая же улика, как трейсбек.

НА ЧТО СМОТРЕТЬ

ИИ глушит ошибку вместо того, чтобы её обработать. Признаки в диффе: `try/except` вокруг упавшей строки, `.get(key, {})` вместо проверки, значение по умолчанию там, где должна быть ошибка. Падать перестанет, врать начнёт.

Починено место падения, а не причина. Если правка коснулась только той строки, на которую указывал трейсбек, спроси: «откуда в эту функцию пришли неверные данные?»

Заодно переписано лишнее. Починка на две строки не должна давать дифф на полсотни. Добавляй в запрос: «меняй только то, что нужно для починки».

Починили, но не проверили. Дифф выглядит правильно — это не значит, что баг ушёл. Воспроизведи тот же случай, с которого начал.

ШПАРГАЛКА

- Три строки: что делал · что ожидал · что получил.
- Трейсбек отдавай дословно и целиком.
- Скажи, в каком слое чинить, иначе починят место падения.
- «Убрать ошибку» ≠ «починить причину».
- Нет трейсбека — уликой служат два несошедшихся числа.
- После починки воспроизведи исходный случай.

3.6 Шаблон — рефакторинг

меняем форму, не трогая поведение

ЧТО ЭТО

Рефакторинг — это изменение кода без изменения его поведения. Разбить длинную функцию, вынести повтор, переименовать по-человечески. Снаружи после рефакторинга всё работает ровно как раньше.

Отсюда главная сложность запроса: **надо явно сказать, что поведение менять нельзя**. Иначе ИИ воспримет задачу как «сделай лучше» и заодно поправит логику — а это уже не рефакторинг, и проверять придётся всё.

КАК ВЫГЛЯДИТ В КОДЕ

Шаблон

ЗАПРОС

```
Что не так: <почему код требует переделки>
Что сделать: <какое изменение формы>
Поведение не меняем: <что должно работать ровно как раньше>
Как проверю: <чем убедимся, что не сломали>
```

Плохо

ТАК НЕ НАДО

```
Отрефактори handlers/weather.py, сделай покрасивее
```

«Покрасивее» — приглашение переписать файл с нуля. Получишь другую архитектуру, другие имена, попутно исправленные (или сломанные) граничные случаи и дифф, который невозможно прочитать.

Хорошо

ТАК ЛУЧШЕ

```
Что не так: функция cmd_weather на 80 строк, в ней смешаны
запрос к API, разбор ответа и формирование текста.

Что сделать: разбей на три функции — fetch, parse, format.
cmd_weather оставь точкой входа, она только вызывает их по очереди.

Поведение не меняем: тексты ответов пользователю, формат логов
и обработка ошибок остаются ровно такими же, до символа.

Как проверю: прогоню те же три команды, что и раньше, и сравню
ответы бота дословно.
```

КАК СКАЗАТЬ ИИ

Фраза, без которой не стоит начинать

> *Поведение не меняем.*

Её надо писать прямо, даже если кажется, что это и так понятно из слова «рефакторинг». Для ИИ это слово размыто: в обучающих данных им называют и переименование переменной, и переписывание модуля с нуля.

Рефакторинг и фича — разные заходы

Соблазн велик: раз уж лезем в этот файл, давай заодно добавим кеш. Так делать не надо — и вот почему. Если после правки что-то сломается, ты не будешь знать, сломал это рефакторинг или новая фича. Разделённые, они диагностируются мгновенно; смешанные — часами.

ЗАПРОС

```
Сначала только разбей на функции, без изменений поведения.
Кеш добавим следующим шагом, отдельно.
```

ЗАПОМНИ

Рефакторинг — всегда отдельный коммит. Это не правило хорошего тона, а способ не потерять вечер на поиск того, что именно сломалось.

Чем проверять

Лучше всего — тестами: прогнал до, прогнал после, результат совпал. Если тестов нет, годится и ручная проверка, но её надо описать заранее, иначе после правки окажется, что сравнивать не с чем.

Приём, который стоит минуты: до рефакторинга сохрани вывод программы на нескольких входах в файл. После — сравни.

```
python bot.py --dry-run > before.txt
# ... рефакторинг ...
python bot.py --dry-run > after.txt
diff before.txt after.txt
```

Пустой `diff` — рефакторинг прошёл чисто.

НА ЧТО СМОТРЕТЬ

Дифф больше, чем должен быть. Разбиение функции на три — это перемещение строк, а не их переписывание. Если в диффе почти каждая строка изменена, значит, код переписали заново, и поведение почти наверняка поехало.

Изменились тексты. Самое частое непрошеное улучшение: ИИ «причёсывает» сообщения пользователю. Было «Город не найден», стало «К сожалению, указанный город не найден 😔». Мелочь, но это уже изменение поведения.

Появились новые зависимости или абстракции. Базовый класс, фабрика, декоратор — там, где просили просто разбить функцию. Признак того, что ИИ решал свою задачу, а не твою.

Исчезла обработка краёв. Проверка на пустой список, `if not data`, ранний `return` — при переписывании такие строки теряются первыми, потому что выглядят необязательными.

ШПАРГАЛКА

- Четыре строки: что не так · что сделать · поведение не меняем · как проверю.
- «Поведение не меняем» пиши явно, слово «рефакторинг» этого не гарантирует.
- Рефакторинг и фича — разные заходы и разные коммиты.
- Сохрани вывод до и после, сравни через `diff`.
- Большой дифф при разбиении функции — код переписали, а не переставили.

3.7 Шаблон — объясни этот код

главный запрос для того, кто учится читать

ЧТО ЭТО

Самый недооценённый запрос. Он ничего не создаёт, поэтому кажется необязательным — а на деле именно он превращает чужой код в твой и закрывает шаг 4 рабочего цикла.

Важно спрашивать **до** того, как принял код, а не после того, как он сломался. Объяснение на две минуты экономит час отладки через неделю.

КАК ВЫГЛЯДИТ В КОДЕ

Три уровня вопроса

ЗАПРОС

```
Что это делает в двух предложениях?
```

Для быстрой ориентировки: открыл чужой файл, надо понять, нужен ли он вообще.

ЗАПРОС

```
Объясни построчно, что делает эта функция.
Для каждой строки: что происходит и зачем она нужна.
```

Для кода, который собираешься принять в проект.

ЗАПРОС

```
Почему здесь сделано именно так? Какие были альтернативы
и почему выбран этот вариант?
```

Для кода, который кажется странным. Часто выясняется, что странность обоснована — или что обоснования нет и код можно упростить.

Вопрос, который работает лучше всех

ТАК ЛУЧШЕ

```
Что произойдёт, если API вернёт пустой список?
А если соединение оборвётся на середине?
```

Вопрос «что будет, если» проверяет код на прочность лучше, чем просьба объяснить. Объясняя, ИИ пересказывает намерение; отвечая на «что если», он вынужден проследить реальный путь выполнения — и нередко сам находит дыру.

КАК СКАЗАТЬ ИИ

Проси объяснять на своём уровне

Объяснение бесполезно, если в нём столько же непонятных слов, сколько в коде.

ЗАПРОС

```
Объясни, считая, что я знаю Python на уровне функций и словарей,
но не работал с async и не знаю, что такое декоратор.
```

Это не стыдно и не «для новичков» — это экономия времени. ИИ подстроит объяснение, и не придётся задавать пять уточняющих вопросов.

Проси пример вместо определения

Абстрактное объяснение забывается, конкретное — нет.

ЗАПРОС

```
Покажи на примере: какие данные придут в эту функцию
и что она вернёт. Два-три случая, включая неудачный.
```

СОВЕТ

Хороший способ проверить, что понял: перескажи объяснение своими словами и попроси поправить. «Я понял так: функция берёт X, если Y — делает Z. Верно?» Ошибки в понимании вылезают сразу.

Объяснение как приёмка

Перед тем как принять сгенерированный код, стоит задать один вопрос:

ЗАПРОС

```
Назови три места в этом коде, где он сломается
при неожиданных входных данных.
```

Если ИИ не может назвать ни одного — скорее всего, он не проверял; если называет три — у тебя готовый список для проверки.

НА ЧТО СМОТРЕТЬ

ИИ объясняет замысел, а не код. Он читает имена функций и пересказывает, что они *должны* делать. Если функция называется `validate_user`, но ничего не проверяет, объяснение всё равно скажет «проверяет пользователя». Защита: спроси не «что делает», а «что произойдёт при таком-то входе».

Объяснение звучит уверенно и при этом неверно. Особенно в чужих библиотеках и редких языковых конструкциях. Если от объяснения зависит решение — проверь в документации.

Слишком гладко. Если объяснение не содержит ни одного «здесь может сломаться», скорее всего, это пересказ, а не разбор. Спроси про слабые места прямо.

ШПАРГАЛКА

- Спрашивай до того, как принял код, а не после поломки.
- Три уровня: в двух предложениях · построчно · почему так.
- «Что будет, если…» работает лучше, чем «объясни».
- Называй свой уровень — получишь объяснение без лишних слов.
- Перескажи своими словами и попроси поправить.
- ИИ склонен пересказывать намерение по именам, а не читать код.

3.8 Шаблон — сделай ревью

и почему нельзя просить об этом в том же чате

ЧТО ЭТО

Ревью — запрос найти в коде проблемы. У него есть особенность, которой нет у остальных: **ИИ плохо критикует код, который сам только что написал.** В том же разговоре он склонен подтверждать свои решения, а не оспаривать их.

Поэтому ревью почти всегда начинается с переноса кода в чистую сессию.

КАК ВЫГЛЯДИТ В КОДЕ

Плохо

ТАК НЕ НАДО

```
Сделай ревью этого кода
```

Без указания, что искать, получишь список общих мест: «добавьте обработку ошибок», «используйте аннотации типов», «вынесите константы». Всё верно и всё бесполезно — такое же можно написать не глядя на код.

Хорошо

ТАК ЛУЧШЕ

```
Вот функция из моего проекта. Я её не писал, нужен разбор.

Ищи в таком порядке:
1. Случаи, когда она вернёт неверный результат молча, без ошибки
2. Необработанные краевые случаи: пустые данные, None, обрыв сети
3. Всё, что сломается при одновременном вызове из двух мест

Стиль и именование не комментируй.
Для каждой находки: что сломается и при каких данных.
Если находок нет — так и скажи, не придумывай.

<код>
```

Три отличия: задан порядок важности, отсечено лишнее, и явно разрешено ничего не найти.

Фраза, которая меняет результат

> *Вот функция из моего проекта. Я её не писал.*

Снимает с ИИ роль автора, защищающего свою работу. Проверяется легко: попроси ревью своего же кода в том же чате и в новом — разница заметна сразу.

КАК СКАЗАТЬ ИИ

Задай порядок, а не просто список

Без приоритетов ревью превращается в перечень из двадцати пунктов, где опечатка в комментарии стоит рядом с потерей данных. Проси сортировать по тяжести последствий:

ЗАПРОС

```
Отсортируй находки от самой опасной к самой мелкой.
Для каждой скажи, что будет, если её не чинить.
```

Разреши не найти ничего

ИИ склонен выдавать находки, потому что об этом попросили. Если их нет, он придумает. Одна фраза это снимает:

ЗАПРОС

```
Если серьёзных проблем нет — скажи прямо, что нет.
Придумывать замечания не надо.
```

ВНИМАНИЕ

Относись к найденному как к гипотезам, а не к приговору. Часть замечаний окажется неверной — ИИ не видит всего проекта и может считать проблемой то, что решено в другом месте. Каждое замечание проверяй по коду.

Три вопроса вместо ревью

Когда разбирать весь код не нужно, работают короткие точечные вопросы:

ЗАПРОС

```
Есть ли здесь место, где данные могут потеряться без ошибки?
```

ЗАПРОС

```
Что из этого сломается, если функцию вызовут дважды подряд?
```

ЗАПРОС

```
Какую строку ты бы переписал первой и почему?
```

Последний вопрос особенно хорош: он заставляет выбрать одно, а не перечислять всё подряд.

НА ЧТО СМОТРЕТЬ

Ревью из общих мест. «Добавьте логирование», «подумайте о производительности» — замечания, которые подходят любому коду. Признак того, что в запросе не было сказано, что искать.

Выдуманные проблемы. ИИ ссылается на поведение библиотеки, которого нет, или на край, невозможный по логике проекта. Проверяй по коду и документации.

Пропущено главное. Отсутствие замечаний не значит, что код хороший, — значит только, что ИИ ничего не нашёл. Своей проверки это не отменяет.

Ревью своего же кода в том же чате. Самая частая ошибка. Новая сессия стоит тридцати секунд и меняет качество разбора сильнее, чем любая формулировка.

ШПАРГАЛКА

- Ревью — в чистой сессии, с фразой «я этот код не писал».
- Скажи, что искать и в каком порядке важности.
- Отсеки лишнее: «стиль не комментируй».
- Разреши не найти ничего, иначе придумает.
- Находки — гипотезы, проверяй каждую по коду.
- «Какую строку переписал бы первой» — короткая замена полному ревью.

3.9 Шаблон — напиши тесты

писать не обязательно, читать обязательно

ЧТО ЭТО

Тест — это код, который проверяет другой код. Писать тесты самому не придётся, их охотно пишет ИИ. А вот **читать их придётся**, и это окупается: хороший набор тестов — это список того, что твоя программа обещает делать.

Для того, кто работает с ИИ, у тестов есть второе применение, которое важнее первого: они не дают сломать то, что уже работало. Попросил правку, прогнал тесты — сразу видно, задел ли ИИ соседнее.

КАК ВЫГЛЯДИТ В КОДЕ

Сначала список случаев, потом код

Главный приём: не проси сразу тесты. Попроси перечислить, что проверять.

ТАК ЛУЧШЕ

```
Вот функция split_message. Перечисли списком, какие случаи
стоит проверить тестами. Пока без кода — только список.
Отметь, какие случаи считаешь важными, а какие — на всякий случай.
```

Список читается за полминуты, и в нём сразу видно, одинаково ли вы поняли задачу:

ОТВЕТ ИИ

```
Важные:
1. Текст короче лимита — возвращается одним куском
2. Текст длиннее лимита — режется по границам абзацев
3. Склейка кусков равна исходному тексту
4. Слово не разрывается посередине

На всякий случай:
5. Пустая строка
```

```
6. Текст ровно по лимиту, символ в символ
7. Абзац длиннее лимита целиком
```

Случай 7 — тот самый, о котором обычно забывают. Увидев список, ты решаешь, что нужно, а что лишнее, и только потом просишь код.

ЗАПРОС
```
Напиши тесты на pytest для случаев 1–4 и 7. Остальные пропусти.
```

Шаблон

ЗАПРОС
```
Код: <что тестируем>
Чем: <pytest / unittest / что в проекте уже есть>
Случаи: <список, согласованный заранее>
Без: <моков, если можно обойтись; внешней сети; файлов на диске>
```

КАК СКАЗАТЬ ИИ

Тест проверяет поведение, а не устройство

Плохой тест ломается при любом рефакторинге, даже если программа работает правильно. Так бывает, когда тест заглядывает внутрь: проверяет приватные поля, порядок вызовов, промежуточные переменные.

ЗАПРОС
```
Тесты должны проверять только то, что функция возвращает,
и её видимые эффекты. Во внутреннее устройство не заглядывай:
после рефакторинга тесты должны продолжать проходить.
```

Проси тесты, которые сначала падают

Способ убедиться, что тест вообще что-то проверяет:

ЗАПРОС
```
Перед тем как чинить баг, напиши тест, который его
воспроизводит. Он должен падать на текущем коде.
```

Падающий тест — доказательство, что баг понят правильно. Если он сразу зелёный, значит, проверяет не то.

> **ЗАПОМНИ**
> Тест, который никогда не падал, ничего не гарантирует. Прежде чем верить тесту, убедись, что он умеет краснеть.

Запускать обязательно

Сгенерированные тесты надо прогнать. Звучит очевидно, но пропускают это постоянно — тесты выглядят убедительно и создают ложное чувство покрытия.

```
pytest -q
```

НА ЧТО СМОТРЕТЬ

Тест, который проходит всегда. `assert result is not None` там, где функция не может вернуть `None`. Выглядит как проверка, не проверяет ничего.

Моки вместо проверки. ИИ подменяет заглушкой половину системы, и тест проверяет, что заглушка вернула то, что в неё положили. Формально зелёный, по сути пустой.

Тесты повторяют реализацию. Если в тесте построчно воспроизведён алгоритм функции, он не проверяет правильность — он проверяет, что код равен самому себе.

Проверены только удачные случаи. Пустые данные, `None`, обрыв сети, слишком длинный вход — туда баги и прячутся. Если в наборе нет ни одного неудачного случая, набор неполон.

ШПАРГАЛКА

- Сначала список случаев, потом код тестов.
- Тест проверяет поведение, а не внутреннее устройство.
- Тест на баг должен падать до починки.
- Тест, который никогда не падал, ничего не гарантирует.
- Сгенерированные тесты обязательно прогнать.
- Нет ни одного неудачного случая — набор неполон.

3.10 Приём — сначала план, потом код

десять строк текста вместо двухсот строк переделок

ЧТО ЭТО

Приём простой: прежде чем просить код, попроси план. Пусть ИИ сначала напишет, что собирается сделать, — а ты проверишь, совпадает ли это с тем, что ты имел в виду.

Выгода в том, **где** обнаруживается расхождение. Неверный план виден за полминуты, неверный код — через час, когда он уже вписан в проект и обрастает вызовами.

КАК ВЫГЛЯДИТ В КОДЕ

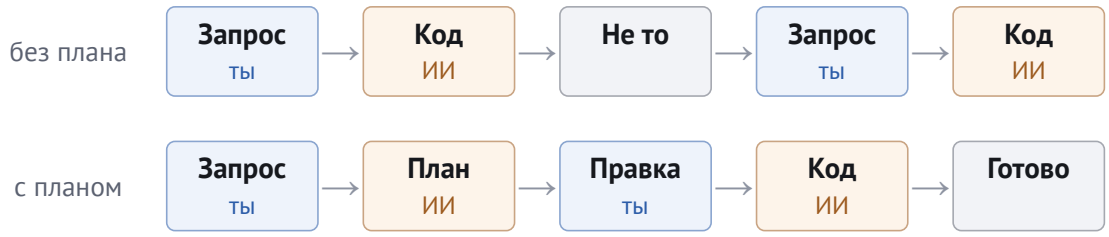


Рис. 3.1 · Расхождение дешевле ловить в плане. Путь с планом длиннее на один шаг и короче на все переделки.

Как просить

ТАК ЛУЧШЕ

```
Не пиши код. Сначала распиши планом:
- какие файлы затронешь и что в каждом изменится
- какие функции добавишь, с сигнатурами
- что может сломаться в существующем коде

Жду план, потом скажу, где поправить.
```

Фраза «не пиши код» нужна буквально: без неё ИИ напишет и план, и код сразу, и смысл приёма пропадёт — читать план, когда под ним уже лежит готовое решение, никто не станет.

Что проверять в плане

Не всё подряд, а четыре вещи:

Смотрим	Тревожный признак
Список файлов	файлов больше, чем ты ожидал
Сигнатуры	аргументы, которых ты не предполагал
«Что может сломаться»	пусто или «ничего»
Порядок шагов	начинает с переделки того, что работает

Пустой раздел «что может сломаться» — самый надёжный признак, что ИИ не думал о последствиях, а перечислил желаемое.

КАК СКАЗАТЬ ИИ

Правка плана — это обычный разговор, не надо переписывать запрос целиком:

ЗАПРОС

```
План в целом верный, но:
1. storage.py не трогай, подписки оставь как есть
2. вместо нового класса сделай две функции
3. начни с миграции, а не с хендлеров
```

```
Перепиши план с учётом этого.
```

Когда план устраивает — одна фраза:

ЗАПРОС

```
План принят, делай по нему. Если по ходу поймёшь,
что план не работает, остановись и скажи, а не придумывай
другое решение на ходу.
```

Вторая фраза важна. Без неё ИИ, упёршись в препятствие, молча свернёт в сторону, и ты получишь код, который не соответствует согласованному плану.

СОВЕТ

Сохраняй принятый план в файл задачи — например, в `docs/tasks/`. Через неделю, вернувшись к коду, ты прочитаешь не только что сделано, но и почему именно так.

Когда приём не нужен

На правках уровня «одна строка» (статья 3.3) план — лишняя трата времени. Приём окупается там, где затронуто больше одного файла или где откат дорог.

НА ЧТО СМОТРЕТЬ

План подозрительно гладкий. Все шаги простые, рисков нет, всё укладывается в три пункта. Обычно это значит, что сложность не обнаружена, а не что её нет. Спроси: «что в этом плане самое рискованное?»

Код не соответствует плану. Бывает, особенно в длинных сессиях. Приняв работу, сверь с планом: те ли файлы, те ли сигнатуры.

План как способ оттянуть. Если планы просятся по три раза подряд и каждый раз правятся, дело не в плане — задача сформулирована неясно. Вернись к шагу 1 рабочего цикла.

ШПАРГАЛКА

- «Не пиши код» — обязательная фраза, иначе напишет сразу и то и другое.
- В плане проверяй: файлы, сигнатуры, что сломается, порядок шагов.
- Пустое «что может сломаться» — признак, что о последствиях не думали.
- «Упрёшься — остановись и скажи, а не придумывай на ходу».
- Принятый план сохраняй рядом с задачей.
- На однострочных правках приём не нужен.

3.11 Приём — пусть ИИ задаёт вопросы

способ увидеть, что ты не сказал

ЧТО ЭТО

Перед тем как ИИ начнёт работу, попроси его задать тебе вопросы. Приём решает причину 1 из статьи 3.1 — «ты не сказал» — и решает её лучше любых шаблонов, потому что находит пропуски, о которых ты сам не догадывался.

Формулировка в одну строку:

ЗАПРОС

```
Прежде чем писать код, задай мне 3 вопроса,
ответы на которые сильнее всего повлияют на решение.
```

Ограничение «3» существенно. Без него вопросов будет десять, половина из них мелкие, и отвечать надоест на четвёртом.

КАК ВЫГЛЯДИТ В КОДЕ

Задача выглядит ясной:

ЗАПРОС

```
Добавь в бот команду /stats — статистика пользователя за месяц.
Прежде чем писать код, задай мне 3 вопроса, ответы на которые
сильнее всего повлияют на решение.
```

Ответ:

ОТВЕТ ИИ

```
1. «За месяц» — это последние 30 дней или текущий календарный
   месяц? От этого зависит запрос к базе.
2. Статистика только по своим данным или админ может смотреть
   чужие? Это меняет проверку прав.
3. Что показывать, если данных за период нет совсем —
   нули или сообщение «нет данных»?
```

Все три — настоящие развилки, и ни одну из них ты в запросе не закрыл. Без вопросов получил бы три чужих решения и узнал бы о них, когда команда повела бы себя не так.

Вариант посильнее

```
ТАК ЛУЧШЕ
Прежде чем писать код: перечисли, какие решения тебе придётся
принять за меня, если я не отвечу. По каждому скажи,
что выберешь по умолчанию.
```

Этот вариант выгоднее: он не требует ответов. ИИ показывает список своих решений с вариантами по умолчанию, ты читаешь и вмешиваешься только там, где не согласен.

```
ОТВЕТ ИИ
Приму за тебя:
- период: последние 30 дней (проще и привычнее для ботов)
- доступ: только свои данные
- пустой период: сообщение «нет данных», не нули
- формат: список строк, без таблицы
```

Одной фразой «всё верно, кроме периода — нужен календарный месяц» задача закрыта.

КАК СКАЗАТЬ ИИ

Когда приём особенно полезен

- **Задача своими словами, без деталей.** Когда сам ещё не додумал, как должно работать.
- **Чужая предметная область.** Интеграция с незнакомым API, формат, который видишь впервые.
- **Дорогая переделка.** Миграции, изменения структуры данных, всё, что потом тяжело откатывать.

Когда не нужен

Если задача уже описана по шаблону из статьи 3.4, вопросов не останется — а ИИ, которого попросили спросить, спросит что-нибудь формальное, лишь бы выполнить просьбу. На ясных задачах приём только добавляет шаг.

> **СОВЕТ**
> Хорошо сочетается с планом (статья 3.10): сначала три вопроса, потом план с учётом ответов, потом код. На большой задаче эти два шага занимают пять минут и экономят вечер.

Отвечать можно коротко

Отвечать развёрнуто не нужно, достаточно пометить:

ЗАПРОС

```
1 — календарный месяц
2 — только свои
3 — «нет данных»
```

НА ЧТО СМОТРЕТЬ

Вопросы ради вопросов. «Какой у вас стиль именования?», «Нужны ли комментарии?» — формальные вопросы, не влияющие на решение. Признак того, что задача и так ясна и приём здесь лишний.

ИИ спросил и тут же ответил сам. Иногда он задаёт вопросы и, не дожидаясь, продолжает писать код со своими допущениями. Это не страшно, но решения он принял сам — прочитай, какие именно.

Больше трёх вопросов. Если ограничение не держится и вопросов десяток, задача сформулирована слишком широко. Разбей её на части.

ШПАРГАЛКА

- «Задай 3 вопроса, которые сильнее всего повлияют на решение».
- Ограничение числом обязательно, иначе вопросов будет десять.
- Сильнее: «какие решения примешь за меня и что выберешь по умолчанию».
- Отвечать можно одной строкой на вопрос.
- На ясной задаче приём лишний — вопросы будут формальными.
- Десяток вопросов — задача слишком широкая, разбей её.

3.12 Приём — примеры вход ⟶ выход

два примера вместо абзаца описания

ЧТО ЭТО

Вместо того чтобы описывать словами, что должна делать функция, покажи пару «что пришло ⟶ что должно получиться». Два примера снимают больше неясности, чем абзац объяснений, и проверяются мгновенно.

Приём особенно силён там, где словами описывать долго и неточно: разбор текста, форматы, преобразования данных, граничные случаи.

КАК ВЫГЛЯДИТ В КОДЕ

Словами

ТАК НЕ НАДО

```
Напиши функцию, которая нормализует номер телефона
в международный формат
```

Что такое «нормализует»? Какие форматы на входе? Что с пробелами и скобками? Что делать с номером без кода страны? Все эти развилки ИИ закроет сам.

Примерами

ТАК ЛУЧШЕ

```
Функция normalize_phone(raw: str) -> str | None.
Примеры:

"+47 912 34 567"   -> "+4791234567"
"47-912-34-567"    -> "+4791234567"
"912 34 567"       -> "+4791234567"   (нет кода — считаем норвежским)
"+7 999 123-45-67" -> "+79991234567"
"не телефон"       -> None
""                 -> None

Код страны по умолчанию передаётся аргументом default_cc="47".
```

Шесть строк закрывают всё: формат вывода, что делать с разделителями, поведение без кода страны, обработку мусора. Словами на то же ушло бы полстраницы, и что-нибудь всё равно осталось бы недосказанным.

Примеры заодно становятся тестами

ЗАПРОС

```
Сделай из этих примеров параметризованный тест pytest.
```

Один запрос — и проверка готова. Примеры, которыми ты объяснял задачу, превращаются в то, чем она проверяется. Это самый дешёвый способ получить тесты из статьи 3.9.

КАК СКАЗАТЬ ИИ

Сколько примеров нужно

Три-пять. Один показывает удачный случай и ничего не говорит о границах. Десять — избыточны и начинают противоречить друг другу.

Обязательно включи:

- **обычный случай** — то, ради чего всё делается;

- **граничный** — пустая строка, ноль, одна запись, максимальная длина;
- **неудачный** — мусор на входе, и что при этом должно произойти.

Третий пункт пропускают чаще всего, а он решает больше всех: именно он говорит ИИ, возвращать `None`, кидать исключение или подставлять значение по умолчанию.

Формат записи

Стрелка и выравнивание читаются лучше прозы:

```
вход              -> выход
```

Если выход сложный — покажи его как есть, со структурой:

ЗАПРОС

```
Вход:  {"cod": "404", "message": "city not found"}
Выход: None

Вход:  {"main": {"temp": 4.2}, "weather": [{"main": "Rain"}]}
Выход: ("Rain", 4.2)
```

СОВЕТ

Когда формат данных незнакомый, попроси обратное: пусть ИИ сам придумает пять примеров входов и скажет, что по его мнению должно получиться. Ты поправишь два из них — и задача будет описана точнее, чем если бы ты писал примеры сам.

Где приём не подходит

Там, где результат нельзя записать строкой: интерфейс, архитектура, «сделай красиво». Для таких задач работают план (3.10) и вопросы (3.11).

НА ЧТО СМОТРЕТЬ

Код подогнан под примеры. Самая коварная ошибка: функция содержит `if raw == "+47 912 34 567": return ...`. Выглядит рабочей, проходит все твои примеры и разваливается на седьмом входе. Прочитай код, а не только результат.

Примеры противоречат друг другу. Бывает, когда их много и писались они второпях. ИИ выберет какой-то один вариант молча. Если заметил противоречие в ответе — чини примеры, а не код.

Граничные случаи придуманы за тебя. Если ты не сказал, что делать с пустой строкой, в коде появится решение, которого ты не принимал. Проверь, какое.

ШПАРГАЛКА

- Два-три примера «вход ⟶ выход» вместо абзаца описания.
- Обязательно: обычный случай, граничный, неудачный.
- Неудачный случай решает, что будет при мусоре на входе.
- Примеры превращаются в тест одним запросом.
- Проверь, что код решает задачу, а не подогнан под примеры.
- Для задач без строкового результата приём не годится.

3.13 Приём — ограничить зону правок

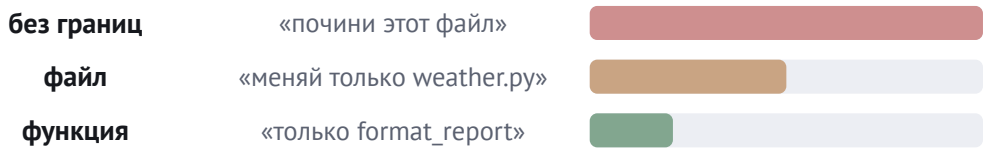
чем меньше трогаем, тем легче проверять

ЧТО ЭТО

По умолчанию ИИ считает, что может менять всё, до чего дотянется. Попросил починить функцию — заодно переименованы переменные в соседней, переставлены импорты и «улучшена» обработка ошибок там, где никто не просил.

Приём состоит в том, чтобы назвать границы заранее. Выгода не в аккуратности, а в проверяемости: **дифф на десять строк ты прочитаешь, дифф на двести — нет.** А непрочитанный дифф — это код, который ты не контролируешь.

КАК ВЫГЛЯДИТ В КОДЕ



ширина полосы — размер диффа, который придётся читать

Рис. 3.2 · Чем уже зона, тем меньше дифф и тем вероятнее, что ты его действительно прочитаешь.

Фразы, которые работают

ЗАПРОС

```
Меняй только функцию format_report. Остальной файл не трогай.
```

ЗАПРОС

```
Правки только в handlers/weather.py. Если понадобится менять
что-то ещё — скажи, но не делай.
```

ЗАПРОС

```
Импорты, форматирование и имена существующих переменных
не трогай. Только логика внутри указанной функции.
```

Третья формулировка снимает самый частый источник шума: ИИ любит переставлять импорты и «причёсывать» стиль, и эти изменения занимают в диффе больше места, чем настоящая правка.

Вторая половина приёма

Ограничить зону мало — надо дать выход, если границы окажутся неверными:

ТАК ЛУЧШЕ

```
Меняй только fetch_weather. Если задачу невозможно решить,
не трогая остальное, — остановись и скажи, что ещё нужно менять
и почему. Не делай этого сам.
```

Без этой фразы ИИ, упёршись в ограничение, либо нарушит его молча, либо выдаст решение похуже, лишь бы уложиться в границы. С ней он спросит — и ты решишь сам.

КАК СКАЗАТЬ ИИ

Границы бывают трёх размеров

Зона	Когда	Как сказать
Функция	точечная правка, баг	«только функция X»
Файл	фича в одном месте	«правки только в файле Y»
Набор файлов	связанное изменение	«можно трогать A, B и C, остальное нет»

Начинай с самой узкой, которая кажется достаточной. Расширить по просьбе ИИ легко, сузить после того, как он переписал полпроекта, — нет.

> **ЗАПОМНИ**
> Величина диффа — это не вопрос аккуратности, а вопрос контроля. Ты отвечаешь за код, который не читал, ровно так же, как за тот, который написал сам.

Проверка границ по диффу

После правки первым делом смотри не на содержание, а на размер:

```
git diff --stat
```

Одна строка вывода вместо ожидаемой одной — хорошо. Семь изменённых файлов там, где ждал один, — сигнал остановиться и прочитать внимательно, что бы там ни было написано в ответе.

НА ЧТО СМОТРЕТЬ

Границы нарушены молча. Проверяется не чтением ответа, а `git diff --stat`. В тексте ответа ИИ обычно пишет, что сделал только просимое.

Переставлены импорты, изменено форматирование. Само по себе безвредно, но раздувает дифф и прячет в нём настоящие изменения. Запрещай отдельной строкой.

Границы оказались слишком узкими. Если ИИ трижды подряд сообщает, что задачу не решить в заданных рамках, — он, скорее всего, прав. Расширь зону осознанно, а не отменяй ограничение целиком.

Правка размазана по файлу. Даже внутри разрешённой зоны изменения могут оказаться в десяти местах вместо одного. Проси: «сделай изменение в одном месте, а не правь в нескольких».

ШПАРГАЛКА

- Называй зону заранее: функция · файл · набор файлов.
- Начинай с самой узкой; расширить легче, чем откатить.
- Отдельно запрещай трогать импорты и форматирование.
- Всегда добавляй: «не получится — скажи, а не делай сам».
- Проверяй границы через `git diff --stat`, а не по ответу ИИ.
- За непрочитанный дифф отвечаешь ты.

3.14 Файл контекста проекта

написать один раз вместо того, чтобы объяснять каждый день

ЧТО ЭТО

Файл контекста — текстовый файл в корне проекта, который ИИ читает в начале работы. В него складывают то, что иначе приходится объяснять в каждом новом разговоре: стек и версии, структуру проекта, принятые соглашения, команды запуска.

Файл закрывает причину 2 из статьи 3.1 — «он не видит» — раз и навсегда. Признак того, что он нужен: ты второй раз пишешь в чат одно и то же про свой проект.

Называется по-разному в зависимости от инструмента: `CLAUDE.md`, `AGENTS.md`, правила в `.cursor/rules/`. Суть одна, и содержимое переносится между ними почти без правок.

КАК ВЫГЛЯДИТ В КОДЕ

```
# Бот подписок на погоду

## Стек
Python 3.12, aiogram 3.x, SQLite через aiosqlite.
Зависимости в requirements.txt, виртуальное окружение в .venv.

## Структура
bot.py          точка входа, запуск polling
handlers/       хендлеры команд, по файлу на тему
storage.py      работа с БД, все запросы здесь
db.py           подключение, get_conn()
config.py       чтение переменных окружения

## Соглашения
- Все обращения к БД — через функции storage.py.
  В хендлерах SQL не пишем.
- Хендлеры асинхронные, тяжёлые операции не блокируют цикл.
- Тексты ответов пользователю — в locales/ru.py, не в коде.
- Логи через logging, не print.

## Запуск
python -m venv .venv && .venv/bin/pip install -r requirements.txt
python bot.py              запуск
pytest -q                  тесты

## Чего не делать
- Не добавлять зависимости без необходимости.
- Не трогать migrations/, они применяются по порядку.
- Токены только из переменных окружения, в коде их нет.
```

Это полный файл, а не фрагмент. Короткий — и в этом его ценность.

КАК СКАЗАТЬ ИИ

Пять разделов, и этого достаточно

Раздел	Что отвечает
Стек	на чём написано и **каких версий**
Структура	где что лежит
Соглашения	как здесь принято делать
Запуск	какие команды выполнять
Чего не делать	куда не лезть

Раздел «Стек» с версиями — самый важный: он закрывает причину 3 из статьи 3.1, ту самую, которую нельзя вылечить подробностью запроса.

Раздел «Соглашения» — второй по отдаче. Именно он заставляет ИИ писать в стиле проекта, а не в своём.

Главное правило: коротко

Файл читается в начале каждого разговора и занимает место в контексте. Длинный файл вытесняет собой код, который ИИ мог бы прочитать, и внимание размывается.

Ориентир: одна-две страницы. Если не помещается — значит, в файл попало то, что ИИ может узнать сам, прочитав проект.

Писать	Не писать
версии и соглашения	то, что видно из кода
где что лежит	списки всех функций
чего не трогать	историю проекта и планы
команды запуска	документацию по библиотекам

> **ВНИМАНИЕ**
> Файл контекста устаревает молча. Переехал модуль, сменилась версия фреймворка — а в файле по-прежнему старое, и ИИ уверенно пишет по неверному описанию. Это хуже, чем отсутствие файла: ошибочному описанию верят.

Как поддерживать

Проще всего — обновлять его тем же ИИ, который в проекте работает:

ЗАПРОС

```
Прочитай CLAUDE.md и сравни с тем, что сейчас в проекте.
Что в файле устарело? Только список расхождений, без правок.
```

Хороший момент для такой проверки — когда добавил модуль, сменил версию библиотеки или поменял способ запуска.

НА ЧТО СМОТРЕТЬ

Файл есть, а ИИ его не читает. В чате файл надо вставлять вручную, сам он туда не попадёт. Агенты в терминале обычно читают его сами, но не все и не всегда — проверь, спросив: «что ты знаешь о структуре этого проекта?»

В файле описано желаемое, а не действительное. «Все запросы к БД через storage.py» — а на деле в трёх хендлерах SQL прямо в коде. ИИ будет считать написанное правдой.

Файл разросся до десяти страниц. Туда переехала документация, история решений, список всех эндпоинтов. Вынеси это в отдельные файлы, а в контекстном оставь ссылки.

ШПАРГАЛКА

- Пять разделов: стек · структура · соглашения · запуск · чего не делать.
- Версии указывать обязательно.
- Одна-две страницы. Длиннее — вытесняет код из контекста.
- Не пиши то, что ИИ прочитает в коде сам.
- Устаревший файл хуже отсутствующего: ему верят.
- Сверяй файл с проектом после каждого заметного изменения.

3.15 Как писать мини-ТЗ

для задачи, которую жалко переделывать

ЧТО ЭТО

Мини-ТЗ — это третий уровень детализации из статьи 3.3, расписанный полностью. Нужен там, где задача трогает несколько файлов, меняет структуру данных или плохо откатывается.

Это те же четыре части запроса (статья 3.2), просто развёрнутые. Ничего нового учить не надо — надо аккуратно заполнить то, что обычно держат в голове.

КАК ВЫГЛЯДИТ В КОДЕ

Структура

```
Задача         одна фраза: что должно появиться
Зачем          какую проблему решаем
Контекст       как устроено сейчас, что уже есть
Что сделать    по пунктам, в порядке выполнения
Ограничения    чего не делать, что не трогать
Готово, когда  проверяемые условия списком
Вне задачи     что сознательно не делаем сейчас
```

Два раздела из семи обычно пропускают, а они самые полезные.

«Зачем» даёт ИИ возможность предложить решение лучше твоего. Если он знает только *что* делать, он сделает буквально; если знает *зачем* — может сказать «это решается проще вот так».

«Вне задачи» ограничивает аппетит. Без него ИИ достраивает задачу до «правильной»: попросил кеш — получил кеш с инвалидацией, метриками и настройкой через конфиг.

Пример целиком

ЗАПРОС

```
Задача: перенести хранение подписок из JSON-файла в SQLite.

Зачем: при 300+ подписках файл перезаписывается целиком на каждое
изменение, бот на секунду подвисает. Плюс при сбое во время
записи файл бьётся.

Контекст: подписки в data/subs.json, структура
{user_id: [city, ...]}. Читаются load_subs(), пишутся save_subs(),
обе в storage.py. Вызовов — пять, все в handlers/.
SQLite в проекте уже используется для логов, подключение
через get_conn() в db.py.

Что сделать:
1. Таблица subscriptions: user_id INTEGER, city TEXT,
   created_at TEXT. Уникальность по паре (user_id, city).
2. Переписать load_subs/save_subs на SQLite.
   Сигнатуры и возвращаемые типы не менять.
3. Миграция при старте: если есть data/subs.json и таблица
   пуста — перенести данные, файл переименовать в subs.json.bak.

Ограничения:
- только модуль sqlite3/aiosqlite, без ORM
- хендлеры не трогать вообще
- формат логов не менять

Готово, когда:
- на пустой базе бот стартует без ошибок
- подписки из старого JSON видны после перезапуска
- все пять мест вызова работают без правок
- повторный старт не дублирует данные

Вне задачи: индексы, оптимизация запросов, перенос логов,
админские команды для просмотра подписок.
```

Двадцать пять строк — и задача, которую иначе переделывали бы трижды, делается с первого раза.

КАК СКАЗАТЬ ИИ

Сколько времени это занимает

Десять-пятнадцать минут. Окупается на первой же развилке, которую ИИ иначе закрыл бы сам.

Если на мини-ТЗ уходит час — задача слишком большая. Разбей её на две и напиши ТЗ на первую.

Хорошее сочетание

Мини-ТЗ + план (статья 3.10) — самая надёжная связка для крупной задачи:

ЗАПРОС

```
<мини-ТЗ>

Код пока не пиши. Распиши планом, как будешь это делать,
и скажи, где в моём ТЗ не хватает информации.
```

Вторая фраза ценна: ИИ укажет дыры в твоём ТЗ до того, как они превратятся в неверный код.

СОВЕТ

Пиши мини-ТЗ в файл, а не в чат. Сохрани рядом с проектом — например, `docs/tasks/2026-10-subs-to-sqlite.md`. Через месяц это единственное место, где написано, почему структура такая, а не другая.

Когда мини-ТЗ не нужно

На всём, что откатывается одной командой. Мини-ТЗ на добавление функции — это бюрократия, а не аккуратность.

НА ЧТО СМОТРЕТЬ

«Готово, когда» написано прилагательными. «Работает стабильно», «быстро», «удобно» — это не критерии. Если условие нельзя проверить, оно ничего не ограничивает (статья 3.2).

Раздел «Контекст» пересказывает код. Не надо переписывать туда функции — достаточно сказать, где смотреть. ИИ прочитает сам, если у него есть доступ к проекту.

ТЗ есть, а результат не сверили. Главная ценность ТЗ проявляется в конце: пройди по списку «Готово, когда» и отметь каждый пункт. Это две минуты и единственный способ узнать, сделано ли.

Задача выросла по ходу дела. Появились пункты, которых не было, — это не повод дописывать ТЗ на ходу. Доделай текущее, потом поставь вторую задачу.

ШПАРГАЛКА

- Семь разделов, из них два обычно забывают: «зачем» и «вне задачи».
- «Зачем» позволяет ИИ предложить решение лучше твоего.
- «Вне задачи» не даёт достроить задачу до «правильной».
- «Готово, когда» — только проверяемые условия, не прилагательные.
- Мини-ТЗ + «распиши план и скажи, чего не хватает».
- Храни ТЗ в файле рядом с проектом, а в конце сверься по списку.

3.16 Антипаттерны запросов

семь способов потратить время впустую

ЧТО ЭТО

Запросы, которые выглядят нормально и регулярно не работают. Каждый из них легко узнать по формулировке — и у каждого есть короткая замена.

Статья закрывает главу 3: если предыдущие пятнадцать говорили, как надо, то эта — как не надо, и почему именно.

КАК ВЫГЛЯДИТ В КОДЕ

1. «Сделай красиво»

ТАК НЕ НАДО

```
Отрефактори этот модуль, сделай покрасивее
```

Красота не определена, поэтому ИИ подставит свою: другую архитектуру, другие имена, другой стиль. Дифф будет огромным, а поведение поедет.

Замена: назови конкретное изменение формы. «Разбей функцию на три», «вынеси повторяющийся блок», «переименуй `d` в `user_data`».

2. «Исправь всё»

ТАК НЕ НАДО

```
Пройдись по файлу и исправь все проблемы
```

Проблемы не названы, значит, ИИ будет их искать — и найдёт, даже если их нет (та же ловушка, что в ревью, статья 3.8). Получишь двадцать правок, из которых половина не нужна, а две что-то сломают.

Замена: сначала попроси список находок, прочитай его, выбери — и только потом правки.

3. «Ты ошибся»

ТАК НЕ НАДО

```
Нет, ты ошибся. Исправь.
```

Не сказано, в чём ошибка. ИИ согласится — он почти всегда соглашается — и изменит что-нибудь наугад. Иногда то, что работало.

Замена: факты вместо оценки. «Запустил, получил `KeyError: main` на строке 11. Ожидал ответ „Город не найден“».

> **ВНИМАНИЕ**
> Согласие ИИ ничего не значит. Он соглашается и когда прав, и когда нет. Если он «признал ошибку», но не объяснил, в чём она была, — скорее всего, он просто поддержал разговор.

4. «Почему не работает?»

ТАК НЕ НАДО

```
Почему не работает?
```

ИИ не видит ни запуска, ни ошибки, ни окружения. Он начнёт гадать и выдаст пять версий, из которых проверять придётся все.

Замена: шаблон из статьи 3.5 — что делал, что ожидал, что получил, трейсбек целиком.

5. Тот же запрос, но настойчивее

ТАК НЕ НАДО

```
Я же сказал — БЕЗ внешних библиотек!!!
```

Повтор с нажимом не добавляет информации. Если первый запрос был понят не так, второй такой же будет понят не так же.

Замена: переформулируй, а не повторяй. Назови конкретную библиотеку, которую он добавил, и чем её заменить. На длинной ветке — начни новую сессию (статья 5.4): иногда разговор уже «заражён» неверным контекстом, и чистый старт дешевле.

6. Задача из пяти задач

ТАК НЕ НАДО

```
Добавь команду /stats, почини баг с часовыми поясами,
вынеси тексты в отдельный файл и напиши тесты
```

Четыре задачи в одном запросе дают один большой дифф, в котором не разобраться. Что-то сделано хорошо, что-то плохо, откатить по отдельности нельзя.

Замена: по одной. Принял — следующая. Это не медленнее: время уходит не на запросы, а на проверку, а проверять маленькими кусками быстрее.

7. Вежливость вместо содержания

ТАК НЕ НАДО

```
Привет! Надеюсь, у тебя всё хорошо. Не мог бы ты, если тебе
не сложно, посмотреть мой код и, возможно, что-нибудь
подсказать? Буду очень благодарен!
```

Вежливость не вредит, но и не помогает — а здесь она заняла место, где должны были быть файл, версия и вопрос. Та же беда с извинениями и долгими объяснениями, почему ты не разбираешься в теме.

Замена: сразу к делу. Общайся как хочешь, но следи, чтобы содержание в запросе всё-таки было.

КАК СКАЗАТЬ ИИ

Сводка: слева то, что хочется написать, справа — то, что работает.

Антипаттерн	Замена
«сделай красиво»	назови конкретное изменение формы
«исправь всё»	сначала список находок, потом выбор
«ты ошибся»	факты: что запустил, что получил
«почему не работает?»	что делал · что ожидал · что получил
тот же запрос настойчивее	переформулируй или начни новую сессию
пять задач в одной	по одной, с проверкой между
вежливость вместо сути	файл, версия, вопрос

Общее у всех семи одно: **в запросе нет информации, которой у ИИ не было.** Раздражение, повтор, вежливость и обобщения её не добавляют — добавляют только факты, имена файлов, версии, примеры и критерии.

НА ЧТО СМОТРЕТЬ

Разговор пошёл по кругу. Третья попытка объяснить одно и то же — сигнал остановиться. Либо задача сформулирована неясно, либо в контексте сессии накопилось лишнее. Новая сессия с нормально сформулированной задачей почти всегда быстрее.

Хочется поспорить. Спор с ИИ — потраченное время: он уступит, а решение от этого не улучшится. Вместо спора — факты и переформулировка.

Запрос длиннее, чем задача. Три абзаца на правку одной строки означают, что ты объясняешь не задачу, а своё отношение к ней.

ШПАРГАЛКА

- «Сделай красиво», «исправь всё», «ты ошибся» — в запросе нет информации.
- Согласие ИИ не означает, что ошибка была.
- Повтор с нажимом не работает: переформулируй.
- Пять задач в одном запросе дают один непроверяемый дифф.
- Пошло по кругу на третьей попытке — начинай новую сессию.
- Спорить бесполезно: он уступит, решение не улучшится.

4 Проверка и отладка

Третий навык. Код написан и даже запустился — но это ещё ничего не значит. Эта глава о том, как убедиться, что получилось задуманное, и что делать, когда нет.

4.1 Правило доверия

что проверяем всегда, а что по обстоятельствам

ЧТО ЭТО

Проверять каждую строку невозможно — так работа встанет. Не проверять ничего нельзя — так проект перестанет быть твоим. Нужен уровень между, и он задаётся правилом.

Всегда проверяется одно и то же, независимо от задачи. Остальное — по цене ошибки (статья 1.5).

ЧЕТЫРЕ ПРОВЕРКИ, КОТОРЫЕ ДЕЛАЮТСЯ ВСЕГДА

Они занимают пару минут и ловят большую часть проблем.

1. Запуск
Банально, но пропускают. Код, который не запускали, не работает — это не пессимизм, а статистика. Причём запускать надо тот случай, ради которого всё затевалось, а не любой.

2. Дифф
Посмотреть, что изменилось, а не что получилось (статья 2.13). Начинать со `--stat`: сколько файлов, сколько строк. Несовпадение с ожиданием — повод остановиться.

```
git diff --stat
```

3. Неудачный случай
Хотя бы один: пустые данные, недоступная сеть, мусор на входе. Там живут ошибки, которых запуск не находит, потому что запускают обычно удачный путь.

4. Вопрос «что это делает»
Если не можешь объяснить принятый код в двух предложениях — он ещё не твой (статья 1.2). Объяснить можно и себе, и вслух, и в комментарии к коммиту.

> **ЗАПОМНИ**
> Четыре проверки занимают две минуты и не зависят от размера правки. Если на них нет времени — значит, нет времени и на правку.

КАК СКАЗАТЬ ИИ

Три из четырёх проверок ИИ может подготовить сам:

ЗАПРОС

```
Назови три случая, которые стоит проверить руками,
чтобы убедиться, что эта правка работает.
Хотя бы один — неудачный.
```

ЗАПРОС

```
Объясни в двух предложениях, что делает код,
который ты сейчас добавил.
```

Четвёртая — дифф — только твоя: это и есть работа ревьюера.

ЧТО ПРОВЕРЯЕТСЯ ДОПОЛНИТЕЛЬНО

Сверх обязательных четырёх уровень проверки задаётся задачей.

Признак задачи	Что добавить
Трогает данные пользователей	прогон на копии, способ отката
Необратимо	план заранее (3.10), мини-ТЗ (3.15)
Ошибка будет незаметной	тесты (4.6), логи на ключевых местах
Редкая библиотека	сверка с документацией (4.8)
Больше одного файла	чтение диффа построчно
Правится давно работавший код	тест до правки, чтобы зафиксировать поведение

Последняя строка — самый недооценённый приём. Тест, написанный **до** правки, показывает, что работало раньше, и сразу скажет, если это сломалось.

НА ЧТО СМОТРЕТЬ

Проверка свелась к запуску. Самая частая форма самообмана: запустил, ответ верный, принял. Проверен один сценарий из десяти (статья 1.6).

Диффа никто не видел. Если правку принимают по тексту ответа ИИ, а не по диффу, — роль ревьюера вакантна.

Уровень проверки выбран по ощущению. «Тут вроде просто» — не критерий. Критерий — обратимость и заметность.

Проверка отложена до конца. Пять правок подряд, потом проверка всего разом — и теперь непонятно, какая из пяти сломала. Проверяй после каждой.

ШПАРГАЛКА

- Всегда: запуск · дифф · неудачный случай · «что это делает».
- Четыре проверки занимают две минуты и не зависят от размера правки.
- Сверх того — по цене ошибки: обратимость и заметность.
- Правишь давно работавший код — сначала тест на то, как он работал.
- Проверяй после каждой правки, а не после пяти.

4.2 Запуск и чтение логов

что программа рассказывает о себе

ЧТО ЭТО

Лог — это то, что программа пишет о себе, пока работает. Для отладки он важнее трейсбека: трейсбек показывает момент падения, а лог — что происходило до того.

Для бота или сервера лог вообще единственный способ узнать, что случилось: ты не видишь экрана пользователя и не можешь переспросить.

КАК ВЫГЛЯДИТ В КОДЕ

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s %(levelname)s %(name)s: %(message)s",
)
log = logging.getLogger(__name__)


def fetch_weather(city: str) -> dict:
    log.info("запрос погоды: %s", city)
    try:
        data = call_api(city)
    except TimeoutError:
        log.warning("таймаут для города %s, повторяю", city)
        data = call_api(city)
    log.debug("ответ API: %s", data)
    return data
```

Вывод:

```
2026-10-14 13:02:11 INFO handlers.weather: запрос погоды: Oslo
2026-10-14 13:02:12 WARNING handlers.weather: таймаут для города Oslo,
повторяю
```

Пять уровней

Уровень	Когда	Видно по умолчанию
`DEBUG`	подробности для отладки	нет
`INFO`	нормальный ход работы	да
`WARNING`	странно, но работаем дальше	да
`ERROR`	не получилось сделать то, что просили	да
`CRITICAL`	дальше работать нельзя	да

Строка `level=logging.INFO` означает: показывать `INFO` и всё, что серьёзнее. Чтобы увидеть `DEBUG`, уровень меняют на `logging.DEBUG` — обычно временно, на время поиска ошибки.

Как читать лог

Снизу вверх, до первой неожиданности. Последние строки — следствия, интересное обычно выше.

Смотри на время. Разрыв в несколько секунд между соседними строками показывает, где программа ждала. Это самый дешёвый способ понять, что тормозит.

Ищи `WARNING` перед `ERROR`. Предупреждение часто называет настоящую причину, а ошибка — лишь её следствие.

Считай повторы. Одна и та же строка сто раз подряд — цикл, который не должен был зациклиться, или повторные попытки, которые не кончаются.

`print` или `logging`

`print`	`logging`
в одном скрипте, пока отлаживаешь	в боте, сервере, всём, что работает долго
исчезает, когда программа не в терминале	пишется в файл, остаётся после перезапуска
нет уровней и времени	уровни, время, имя модуля

Правило: пока ищешь ошибку руками — `print` (статья 4.5). Всё, что остаётся в коде, — `logging`.

> **СОВЕТ**
> Попроси у ИИ логирование сразу при написании функции: «добавь логирование: INFO на входе с аргументами, WARNING на каждую нештатную ситуацию». Это одна фраза, а когда что-то сломается на сервере, она окупится полностью.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Добавь логирование в эту функцию: INFO на входе с ключевыми
аргументами, WARNING на нештатные ситуации, ERROR на отказ.
Токены и персональные данные в лог не пиши.
```

Когда лог уже есть и надо разобраться:

ЗАПРОС
```
Вот кусок лога. Что здесь произошло? На что обратить
внимание в первую очередь?

<лог>
```

Когда логов не хватает:

ЗАПРОС
```
Я не понимаю, почему эта функция возвращает пустой список.
Куда добавить логирование, чтобы это стало видно?
Минимум строк.
```

НА ЧТО СМОТРЕТЬ

Секреты в логах. Токен или пароль, записанный в лог, — это тот же секрет в открытом виде, только в файле, который потом пересылают. Проверяй, что логируется, прежде чем отдавать лог кому-то, включая ИИ.

Лог, в котором нет ничего. Если при ошибке в логе пусто, значит, логировать забыли — и единственный способ узнать причину уже потерян.

Лог, в котором всё. Обратная крайность: `DEBUG` на всё подряд в рабочем режиме. Файл растёт, важное тонет, диск кончается.

Сообщение без данных. `log.error("ошибка")` не говорит ничего. Хорошая запись содержит значения: какой город, какой пользователь, какой код ответа.

Логирование внутри цикла по тысяче элементов. Тысяча строк на один запрос. Логируй итог, а не каждый шаг.

ШПАРГАЛКА

- Лог важнее трейсбека: показывает, что было до падения.
- Пять уровней: DEBUG · INFO · WARNING · ERROR · CRITICAL.
- Читай снизу вверх до первой неожиданности, смотри на разрывы во времени.
- `WARNING` перед `ERROR` часто и есть настоящая причина.
- `print` — пока отлаживаешь, `logging` — всё, что остаётся в коде.
- В сообщении всегда указывай значения. Секреты не логируй.

4.3 Как подготовить ошибку к починке

минимальное воспроизведение вместо вороха подробностей

ЧТО ЭТО

Статья 3.5 говорит, **как сформулировать** запрос на починку. Эта — про то, **что собрать** до того, как его писать.

Разница существенная: правильно оформленный запрос с плохими уликами не поможет. А хорошие улики — это не «побольше всего», а наоборот: минимальный случай, на котором ошибка ещё воспроизводится.

КАК ВЫГЛЯДИТ В КОДЕ

Ошибка проявилась в боте: команда `/stats` иногда падает. Вот что с ней делают до того, как звать ИИ.

Шаг 1. Воспроизвести

Ошибка, которую нельзя повторить, не чинится — её починку нельзя проверить. Если команда падает «иногда», надо найти, при каких данных.

```
for user_id in [1, 2, 3, 17, 42]:
    try:
        build_stats(user_id)
    except Exception as e:
        print(user_id, "->", type(e).__name__, e)
```

Три строки перебора часто превращают «иногда падает» в «падает на пользователе 17».

Шаг 2. Урезать

Теперь надо убрать из случая всё, что не влияет. Телеграм, база, хендлеры — если ошибка воспроизводится без них, значит, они ни при чём.

```
# Минимальный случай: ни бота, ни базы
rows = [{"date": "2026-10-01", "amount": 100},
        {"date": "", "amount": 50}]
print(build_stats_from_rows(rows))     # падает
```

Урезанный случай ценен вдвойне. Он точнее указывает на причину и даёт готовый тест: именно его потом просят оформить как `pytest` (статья 3.9).

Шаг 3. Собрать три вещи

Что	Зачем
Минимальный случай	на чём воспроизводится
Трейсбек целиком	где упало (статья 2.12)
Что ожидалось	чтобы отличить баг от задуманного

И только теперь — запрос по шаблону из статьи 3.5.

КАК СКАЗАТЬ ИИ

Урезать случай можно и вместе с ИИ, это у него получается хорошо:

ЗАПРОС

```
Вот функция и данные, на которых она падает. Сократи данные
до минимального примера, на котором ошибка ещё
воспроизводится. Код не чини, нужен только пример.
```

Когда ошибка плавающая:

ЗАПРОС

```
Эта функция падает примерно раз из пяти, на одних и тех же
аргументах. Какие могут быть причины непостоянного поведения?
Назови три самые вероятные.
```

Непостоянство почти всегда означает одно из трёх: внешний источник (сеть, время, случайные числа), общее состояние между вызовами или параллельное выполнение. ИИ назовёт их, но проверять придётся тебе.

> **СОВЕТ**
> Урезая случай, записывай, что убрал. Если ошибка вдруг перестала воспроизводиться — значит, последнее убранное и было важным. Это и есть полдела.

ЧЕГО НЕ НАДО ОТДАВАТЬ

Весь файл, когда хватит функции. Лишний контекст размывает внимание — и твоё, и ИИ.

Логи за сутки. Нужен кусок вокруг ошибки: несколько строк до и несколько после.

Секреты. Токены, ключи, персональные данные пользователей. Перед отправкой лога проверь, что в нём (статья 4.2).

Свой диагноз вместо фактов. «Кажется, проблема в базе» уводит ИИ в сторону, если догадка неверна. Факты отдельно, догадки отдельно — и помечай, что догадка.

НА ЧТО СМОТРЕТЬ

Ошибка «починена», но не воспроизводилась. Если ты не умел воспроизводить её до правки, ты не сможешь проверить, что она ушла. Сначала воспроизведение, потом починка.

Минимальный случай не сохранён. Урезал, починил, выбросил — и через месяц при похожей ошибке всё заново. Урезанный случай стоит сразу превратить в тест.

Чинится симптом, а не причина. Связка статей 2.12 и 2.13: место падения и причина — разные места.

Правка не проверена на исходном случае. Самое обидное: починили, тесты зелёные, а на том самом пользователе 17 всё ещё падает.

ШПАРГАЛКА

- Сначала воспроизвести, потом урезать, потом звать ИИ.
- Невоспроизводимая ошибка не чинится: нечем проверить починку.
- Минимальный случай — лучший запрос и готовый тест.
- Три вещи: случай · трейсбек целиком · что ожидалось.
- Не отдавай весь файл, суточные логи и секреты.
- Догадки помечай как догадки, иначе уведут в сторону.

4.4 Сужаем зону поиска

деление пополам вместо чтения всего подряд

ЧТО ЭТО

Когда непонятно, где ошибка, читать весь код подряд — худший способ её найти. Есть приём надёжнее: **разделить путь данных пополам, проверить середину и выбросить ту половину, где всё хорошо.**

Так зона поиска сокращается вдвое на каждом шаге. Из тысячи строк до нужной — десять проверок.

КАК ВЫГЛЯДИТ В КОДЕ



зелёное проверено и исключено · красное — где ещё может быть ошибка

Рис. 4.1 · Каждая проверка в середине выбрасывает половину. Из тысячи строк до одной — десять шагов.

На живом примере

Цепочка из четырёх функций: загрузить строки, разобрать, превратить в числа, посчитать среднее. Падает где-то внутри.

```
def load_rows(raw: str) -> list[str]: ...
def parse_rows(rows: list[str]) -> list[dict]: ...
def to_numbers(items: list[dict]) -> list[dict]: ...
def average(items: list[dict]) -> float: ...

print(average(to_numbers(parse_rows(load_rows(RAW)))))
```

Шаг 1. Печатаем в середине. Не в начале и не в конце — ровно посередине цепочки:

```
rows = load_rows(RAW)
items = parse_rows(rows)
print("после parse_rows:", items)
```

```
после parse_rows: [{'city': 'Oslo', 'temp': '4'},
                   {'city': 'Bergen', 'temp': '7'},
                   {'city': 'Tromso', 'temp': '-2.5'}]
```

Данные в порядке. Значит, первая половина цепочки чистая, и её можно забыть. Осталось две функции из четырёх.

Шаг 2. Делим остаток. Проверяем `to_numbers` на тех же данных, по элементам:

```
for it in items:
    try:
        int(it["temp"])
    except ValueError as e:
        print("падает на:", it, "->", e)
```

```
падает на: {'city': 'Tromso', 'temp': '-2.5'} -> invalid literal for
int() with base 10: '-2.5'
```

Две проверки — и найдены и строка, и конкретные данные. Причина видна сразу: код предполагал целые температуры, а в данных дробная.

Где делить

Что ищем	Где середина
Ошибка в цепочке функций	между функциями, по данным
Ошибка в длинной функции	между логическими блоками
Ошибка появилась после правок	`git log` — между коммитами
Ошибка на одном из тысячи элементов	по половине списка

Третья строка — отдельный приём: если код работал вчера и сломался сегодня, делить надо не код, а историю коммитов. У git для этого есть `git bisect`, но часто хватает и ручного переключения на коммит из середины.

> **ЗАПОМНИ**
> Главный вопрос на каждом шаге — не «где ошибка», а «данные здесь ещё целые?». Отвечать на второй вопрос быстрее, и каждый ответ выбрасывает половину.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Вот цепочка из четырёх функций, где-то внутри падает.
Куда поставить одну печать, чтобы разделить цепочку пополам?
Код не чини, скажи только место.
```

ЗАПРОС

```
Я напечатал данные после parse_rows, они в порядке.
Значит, ошибка дальше. Что проверить следующим?
```

Такой диалог экономит время: ИИ хорошо подсказывает, где середина, а выполняешь и смотришь ты.

НА ЧТО СМОТРЕТЬ

Печать в начале и в конце вместо середины. Начало обычно в порядке, конец обычно сломан — обе проверки не сужают зону.

ИИ предлагает чинить, не сузив зону. Типичный ответ на «не работает» — три гипотезы сразу. Любая может быть верной, проверять придётся все. Сузь сам, потом проси починку.

Зона не сужается, а растёт. Если после каждой проверки появляются новые подозреваемые, значит, проверяются следствия, а не данные. Вернись к вопросу «данные здесь ещё целые?».

Отладка без записи. Проверил три места, забыл результаты, проверяешь заново. Записывай по строке: «после parse_rows — нормально».

ШПАРГАЛКА

- Дели путь данных пополам, проверяй середину, выбрасывай чистую половину.
- Вопрос на каждом шаге: «данные здесь ещё целые?»
- Из тысячи строк до одной — десять проверок.
- Сломалось после правок — дели историю коммитов, не код.
- Печать в начале и в конце зону не сужает.
- Записывай результаты проверок, иначе будешь ходить по кругу.

4.5 Отладка печатью

когда её достаточно и как делать правильно

ЧТО ЭТО

Вставить `print` и посмотреть, что в переменной, — самый простой способ отладки. Его принято считать несерьёзным, но для боль-

шинства задач он быстрее отладчика и не требует ничего настраивать.

Статья — о том, как печатать так, чтобы это помогало, и когда всё-таки нужен отладчик.

КАК ВЫГЛЯДИТ В КОДЕ

Плохо

```
print(data)
print(x)
print("тут")
```

Запустил, получил три непонятных вывода подряд и не знаешь, какой откуда.

Хорошо

```
print(f"[parse] items={len(items)} first={items[0] if items else None!
r}")
print(f"[to_numbers] temp={it['temp']!r}
type={type(it['temp']).__name__}")
```

Три приёма, которые делают печать полезной:

Метка. `[parse]` сразу говорит, откуда строка. При десятке печатей это экономит всё время.

`!r` вместо обычного вывода. `repr` показывает кавычки и невидимое. Разница решающая:

```
value = " 42 "
print(f"value={value}")      # value= 42
print(f"value={value!r}")    # value=' 42 '
```

Во втором случае сразу видно, что в строке пробелы, — а именно из-за них `int()` и падал.

Тип, а не только значение. `'4'` и `4` выглядят почти одинаково, а ведут себя по-разному. `type(x).__name__` снимает вопрос.

> **СОВЕТ**
> Привычка писать `!r` в отладочной печати окупается постоянно. Пустая строка, `None`, строка вместо числа, лишний пробел — всё это видно только через `repr`.

Печать структуры

Для словарей и списков удобнее так:

```
import json
print(json.dumps(data, ensure_ascii=False, indent=2)[:500])
```

`ensure_ascii=False` — чтобы русский текст был читаемым, а не `p`. Срез `[:500]` — чтобы не залить терминал, если данных много.

Когда нужен отладчик

Печати хватает, пока ты знаешь, что печатать. Отладчик нужен, когда не знаешь:

Печать	Отладчик
понятно, какие переменные смотреть	непонятно, что происходит вообще
один-два прохода	нужно пройти шаг за шагом
код запускается быстро	каждый запуск долгий
всё в своём коде	ошибка где-то в чужой библиотеке

Самый дешёвый отладчик в Python — одна строка в нужном месте:

```
breakpoint()
```

Программа остановится, и ты окажешься в интерактивной оболочке, где можно смотреть любые переменные. `c` — продолжить, `n` — следующая строка, `q` — выйти.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Я не понимаю, почему здесь получается пустой список.
Куда поставить две печати, чтобы это стало видно?
Код не меняй, только скажи места и что печатать.
```

ЗАПРОС

```
Добавь отладочную печать во все три функции: метка,
ключевые значения через !r и их типы. Потом я уберу.
```

И обязательный шаг в конце:

ЗАПРОС

```
Убери всю отладочную печать, которую мы добавляли.
Логирование через log.* не трогай.
```

НА ЧТО СМОТРЕТЬ

Печать осталась в коде. Классика: отладили, забыли убрать, оно уехало на сервер и теперь засоряет вывод. Перед коммитом — поиск по `print(` в диффе.

Печать вместо логирования в долгоживущем коде. В боте или на сервере `print` исчезает вместе с терминалом. Там нужен `logging` (статья 4.2).

Печатается не то место. Если вывод не меняется при явно разных входных данных, скорее всего, печать стоит не в той ветке кода — или до неё вообще не доходит.

Секреты в печати. `print(config)` вместе с токеном — и он в истории терминала, а то и в скриншоте, отправленном в чат.

ШПАРГАЛКА

- Метка в начале строки: `[parse] ...`.
- Всегда `!r` — видно пробелы, кавычки, `None`.
- Печатай тип, а не только значение.
- Словари — через `json.dumps(..., ensure_ascii=False)` и срез.
- Не знаешь, что смотреть, — `breakpoint()`.
- Перед коммитом проверь дифф на оставшиеся `print(`.

4.6 Тесты — прочитать и запустить

что означает красный вывод и чему он на самом деле верит

ЧТО ЭТО

Как просить тесты — статья 3.9. Эта статья о том, что делать с полученными: как читать вывод `pytest`, что считать настоящей проверкой и почему зелёные тесты не означают, что всё хорошо.

Писать тесты самому не придётся. Читать их придётся обязательно — иначе ты доверяешь проверке, которую не проверял.

КАК ВЫГЛЯДИТ В КОДЕ

Функция режет длинное сообщение на куски. Тесты написаны по списку случаев из статьи 3.9.

```
pytest -q
```

```
...F
[100%]
================================== FAILURES
==================================
_______________________ test_paragraph_longer_than_limit
_______________________

    def test_paragraph_longer_than_limit():
        text = "y" * 300
        parts = split_message(text, limit=100)
>       assert all(len(p) <= 100 for p in parts)
E       assert False

test_split.py:21: AssertionError
============================ short test summary item
============================
FAILED test_split.py::test_paragraph_longer_than_limit - assert False
1 failed, 3 passed in 0.02s
```

Как это читать

Первая строка — карта. `...F` значит: три теста прошли, четвёртый упал. Точка — успех, `F` — провал, `E` — ошибка в самом тесте, `s` — пропущен.

`>` отмечает строку, на которой упало. Здесь — `assert`.

`E` — что именно не сошлось. `assert False` значит, что условие оказалось ложным.

Последняя строка — итог. `1 failed, 3 passed` — самое важное, с неё можно и начинать.

Что показал упавший тест

Функция не справляется с абзацем, который сам длиннее лимита: она не может его разрезать и отдаёт как есть. Три остальных теста зелёные — на обычных данных всё работает.

Это ровно тот случай, который в статье 3.9 был помечен как «тот, о котором обычно забывают». Тест его поймал — в этом и смысл.

> **ЗАПОМНИ**
> Упавший тест — это хорошая новость. Он нашёл проблему сейчас, а не через месяц у пользователя. Плохая новость — это зелёные тесты, которые ничего не проверяют.

Полезные ключи

```
pytest -q                       # коротко
pytest -x                       # остановиться на первом падении
pytest -k "split"               # только тесты с «split» в имени
pytest test_split.py::test_empty  # ровно один тест
pytest -q --tb=short            # короткий трейсбек
```

Второй и четвёртый — рабочие лошадки при починке: не надо каждый раз ждать весь набор.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Вот вывод pytest. Что именно не сошлось и почему?
Код пока не чини.

<вывод>
```

Когда тест упал после твоей правки:

ЗАПРОС

```
Этот тест проходил до моей правки и упал после.
Я менял только функцию X. Что в ней могло на это повлиять?
```

И проверка качества самих тестов:

ЗАПРОС

```
Эти тесты точно проверяют поведение, а не то, что код
равен самому себе? Есть ли среди них такие, которые
пройдут при любой реализации?
```

НА ЧТО СМОТРЕТЬ

Тест, который не умеет падать. Проверка: сломай код намеренно и запусти. Если тест остался зелёным — он ничего не проверяет.

`assert result is not None` там, где `None` невозможен. Выглядит проверкой, не проверяет ничего.

Моки вместо проверки. Если половина системы подменена заглушками, тест проверяет, что заглушка вернула то, что в неё положили.

Тест повторяет реализацию. Если в тесте построчно воспроизведён алгоритм функции, он сломается при любом рефакторинге, даже правильном.

Все случаи удачные. Нет ни одного теста с пустыми данными, `None` или мусором на входе — набор неполон, и как раз там живут баги.

Зелёные тесты вместо чтения кода. Тесты проверяют то, что в них написано, и ничего больше. Они дополняют чтение диффа, а не заменяют его.

ШПАРГАЛКА

- `...F` — карта прогона: точка успех, `F` провал.
- `>` — строка падения, `E` — что не сошлось, последняя строка — итог.
- Упавший тест — хорошая новость; опасны зелёные и пустые.
- Проверка теста: сломай код, тест обязан покраснеть.
- `pytest -x` и `pytest -k` экономят время при починке.
- Нет ни одного неудачного случая — набор неполон.

4.7 Чек-лист приёмки

в каком порядке смотреть правку

ЧТО ЭТО

Статья 2.14 перечисляет, **что** искать в коде. Эта — про **порядок**: в какой последовательности смотреть правку, чтобы не утонуть и не пропустить главное.

Порядок важен, потому что внимание кончается. Если начать с мелочей, на серьёзное его не останется.

ШЕСТЬ ШАГОВ ПО ПОРЯДКУ

1. Размер правки

```
git diff --stat
```

Совпадает с ожиданием? Одна строка вместо одной — дальше. Семь файлов вместо одного — остановись и разберись, прежде чем читать содержание (статья 3.13).

2. Удалённое

Смотри строки с `-` раньше строк с `+`. Удаление опаснее добавления: добавленный код виден, а исчезнувшая проверка — нет.

Типичные потери: `if not data: return`, проверки длины, обработка краёв.

3. Поведение в неудачном случае

Один вопрос: что будет, если данные придут не те? Если в правке появились `.get()` с умолчанием, `except` или подстановка значений — проверь, не спрятана ли ошибка вместо починки (статьи 2.8, 2.14).

4. Границы
Правка делает то, о чём просили, и ничего сверх? Логирование, кеш, повторные попытки, которых не заказывали, — это непроверенный код в проекте.

5. Запуск
Тот случай, ради которого всё затевалось, плюс один неудачный. Если есть тесты — прогнать (статья 4.6).

6. Объяснение
Можешь объяснить правку в двух предложениях? Если нет — вернись к чтению или спроси (статья 3.7).

> **СОВЕТ**
> Если правка большая и внимания на всё не хватает, остановись после шага 2. Размер и удалённые строки ловят больше проблем, чем всё остальное вместе.

КАК СКАЗАТЬ ИИ

Чек-лист можно отдать и ИИ, но с оговоркой: о своём коде он отвечает снисходительно (статья 3.8). Поэтому — в чистой сессии:

ЗАПРОС
```
Вот дифф. Я его не писал, нужна приёмка.
Пройди по порядку:
1. Что удалено и почему это могло быть важно
2. Что произойдёт при неожиданных данных
3. Что в правке сверх заявленной задачи

Задача была: <что просили>.
Если проблем нет — скажи прямо.
```

Для себя полезно другое — зафиксировать, что проверил:

ЗАПРОС
```
Составь из этой правки список того, что я должен
проверить руками. Коротко, пунктами, чтобы я отметил.
```

ЧТО НЕ НАДО ПРОВЕРЯТЬ

Чтобы внимания хватило на важное, часть вещей стоит сознательно не трогать:

Не проверяй руками	Почему
форматирование и отступы	это работа форматтера, а не твоя
стиль именования	если читается — достаточно

микрооптимизации	пока нет проблемы со скоростью, это не проблема
«я бы написал иначе»	иначе — не значит хуже

Последний пункт самый важный. Спорить с ИИ о вкусовых вещах — потраченное время: он уступит, а код лучше не станет.

НА ЧТО СМОТРЕТЬ

Проверка начинается с содержания, а не с размера. Прочитал двести строк внимательно, а надо было остановиться на `--stat` и переспросить.

Удалённые строки пролистаны. Самый частый способ пропустить настоящую проблему.

Чек-лист превратился в ритуал. Шесть шагов подряд без вопросов — значит, смотрят, но не видят. Нормальная приёмка — это хотя бы один вопрос.

Приёмка отложена на потом. Пять правок, потом разбор всего разом. Ищи теперь, какая из пяти.

ШПАРГАЛКА

- Порядок: размер · удалённое · неудачный случай · границы · запуск · объяснение.
- Начинай со `--stat`, а не с содержания.
- Строки с `-` читай раньше строк с `+`.
- Не хватает внимания — остановись после шага 2, он ловит больше всего.
- Форматирование, именование и вкусовщину не проверяй.
- Приёмка без единого вопроса — это не приёмка.

4.8 Галлюцинации

выдуманные библиотеки, методы и параметры

ЧТО ЭТО

Галлюцинация — когда ИИ уверенно описывает то, чего не существует: библиотеку, метод, параметр, поле в ответе API. Выглядит это как обычный код, читается как обычный код и падает при запуске.

Явление не случайное, а предсказуемое: оно возникает там, где примеров было мало, — редкие библиотеки, свежие версии, узкие API (статья 1.3).

КАК ВЫГЛЯДИТ В КОДЕ

```
async with aiohttp.ClientSession() as session:
    data = await session.fetch_json(url)      # такого метода нет
```

Правдоподобно: у `requests` есть `.json()`, у многих клиентов есть `fetch`. Сочетание выглядит естественно — и не существует.

Три вида, по убыванию заметности:

Вид	Как проявляется
Выдуманный пакет	`ImportError` при запуске — если повезёт; если имя уже кто-то занял, ошибки не будет вовсе
Выдуманный метод	`AttributeError` при вызове — заметно при запуске этой ветки
Выдуманный параметр	в Python обычно `TypeError` молчит там, где параметры не проверяются

Первый и третий опаснее, чем выглядят.

Пакет. Имена, которые ИИ выдумывает, повторяются от ответа к ответу, и их заранее регистрируют в PyPI и npm — с чужим кодом внутри. Тогда `pip install` проходит, импорт работает, а пакет делает то, что в него положили. «Пакет существует» ещё не значит «это тот пакет».

Параметр. Сама функция Python на незнакомый аргумент ругается сразу:

```
await session.get(url, timeout_ms=5000)
```

```
TypeError: ClientSession._request() got an unexpected keyword argument
'timeout_ms'. Did you mean 'timeout'?
```

Молча он проходит там, где параметры никто не проверяет: в объекте настроек JavaScript (`{ timeoutMs: 5000 }`), в `**kwargs`, которые передаются дальше без разбора, в YAML и JSON конфигах. Ошибки нет, действует значение по умолчанию, поведение не то, что задумано.

Как проверять

Пакет:

```
pip index versions aiohttp
```

Если пакета нет, команда так и скажет. Заодно покажет доступные версии.

Это только половина проверки. Вторая — тот ли это пакет: открой его страницу на pypi.org или npmjs.com и посмотри, сколько ему лет, сколько у него загрузок и есть ли ссылка на исходники. Пакет, созданный на прошлой неделе, с именем, которое предложил ИИ, — повод остановиться.

Метод или поле:

```
python3 -c "import aiohttp; print([m for m in dir(aiohttp.ClientSession)
if 'fetch' in m])"
```

```
[]
```

Пустой список — метода нет.

Параметр функции:

```
python3 -c "import inspect, aiohttp;
print(inspect.signature(aiohttp.ClientSession.get))"
```

```
(self, url: Union[str, yarl.URL], *, allow_redirects: bool = True,
**kwargs: Any) -> '_RequestContextManager'
```

Сигнатура показывает, какие аргументы принимаются. Но если в ней есть `**kwargs`, как здесь, она ничего не докажет: аргументы уходят дальше, и ответ даст только документация или вызов с этим параметром.

Все три проверки — одна строка в терминале, и они надёжнее любых рассуждений.

> **ВНИМАНИЕ**
> Не спрашивай у ИИ, существует ли метод, который он только что написал. Он склонен подтверждать собственный ответ. Проверяй в документации или командой — это занимает секунды и даёт точный ответ.

КАК СКАЗАТЬ ИИ

Предупредить дешевле, чем исправлять:

> **ЗАПРОС**
> Используй только методы, в существовании которых уверен.
> Если не уверен — пометь строку комментарием # проверить
> и скажи, что именно проверить в документации.

Такая формулировка работает: прямо разрешив сомневаться, получаешь честные пометки вместо уверенной выдумки.

После получения кода:

ЗАПРОС

```
Перечисли все внешние методы и параметры, которые ты
использовал в этом коде, с указанием библиотеки и версии.
Списком, без кода.
```

Короткий список удобно проверить за минуту — быстрее, чем перечитывать код.

Когда уже упало:

ЗАПРОС

```
AttributeError: 'ClientSession' object has no attribute
'fetch_json'. У меня aiohttp 3.9.5. Какой метод там есть
для этой задачи на самом деле?
```

НА ЧТО СМОТРЕТЬ

Слишком удобный метод. Если библиотека предлагает ровно ту функцию, которая нужна тебе, одним вызовом, — проверь. Реальные библиотеки редко настолько точно совпадают с задачей.

Нестандартное имя. `fetch_json`, `get_all`, `auto_retry` — имена, которые звучат как то, что должно быть. Это и есть признак: ИИ собрал имя из частей, встречавшихся у похожих библиотек.

Параметры, которых нет в документации. Самый незаметный вид. Если параметр важен — проверь сигнатурой.

Код не запускался. Запуск ловит выдуманные методы и почти все выдуманные параметры. Пакет, имя которого кто-то успел занять, запуск не поймает. Запуск — минимальная защита, но он обязателен.

Новый пакет в зависимостях. ИИ добавил библиотеку, о которой ты не просил, — проверь её до установки, а не после.

Выдумка в ответе API, а не в коде. ИИ может предположить, какие поля придёт в JSON, и ошибиться. Печатай реальный ответ, а не доверяй описанию (статья 4.5).

ШПАРГАЛКА

- Галлюцинации живут там, где примеров мало: редкое, свежее, узкое.
- Три вида: пакет · метод · параметр. Пакет и параметр опаснее, чем кажутся.
- Проверка пакета: `pip index versions` — и страница пакета: возраст, загрузки, исходники.
- Проверка метода: `dir()`. Проверка параметра: `inspect.signature()`.
- Не спрашивай ИИ, существует ли то, что он написал, — проверяй сам.
- Слишком удобный метод с «говорящим» именем — первый кандидат на проверку.

4.9 Петля «починил одно — сломал другое»

почему она возникает и как из неё выйти

ЧТО ЭТО

Знакомая картина: починили ошибку A — появилась B. Починили B — вернулась A. Третий круг, четвёртый, и кажется, что код сопротивляется.

Петля возникает не от невезения. У неё три причины, и все три устранимы.

ПОЧЕМУ ТАК ПРОИСХОДИТ

Причина 1. ИИ не видит целого

Каждая правка делается локально: вот эта функция, вот эта ошибка. Что эта функция вызывается ещё в трёх местах с другими ожиданиями — ИИ не знает, если ему не показали.

Поэтому правка, верная сама по себе, ломает соседнее.

Причина 2. Чинится симптом

Если каждый раз править место падения, а не причину, ошибка никуда не девается — она переезжает. Выскакивает в другом месте, под другим именем, и выглядит как новая (статьи 2.12, 2.13).

Причина 3. Контекст сессии засорён

К пятому кругу в разговоре лежат три отменённых решения, два неверных предположения и куча кода, который уже не актуален. ИИ опирается на всё это и начинает противоречить сам себе.

КАК ВЫЙТИ

Остановиться и записать

Первое и самое полезное — перестать просить правки и выписать, что известно. Пять строк обычного текста:

```
A: падает при пустом городе — чинили дважды
B: дублируются записи — появилось после первой починки A
Общее: обе про функцию save_sub
Не проверял: вызывается ли save_sub из двух мест
```

Часто на этом месте петля и кончается: становится видно общее звено.

Зафиксировать поведение тестом

Пока нет проверки, каждая правка — это обмен одной ошибки на другую вслепую. Два теста — на A и на B — превращают игру в угадайку в работу: теперь видно, когда починены оба.

ЗАПРОС

```
Напиши два теста: один воспроизводит A, второй B.
Оба должны падать на текущем коде. Код не чини.
```

Требование «оба должны падать» существенно — оно доказывает, что тесты действительно проверяют нужное (статья 4.6).

Сузить зону

Если правки трогают полпроекта, каждая задевает лишнее. Явные границы (статья 3.13) резко снижают число побочных поломок:

ЗАПРОС

```
Меняй только save_sub. Если задачу не решить, не трогая
остальное, — остановись и скажи, что ещё нужно менять.
```

Начать новую сессию

Если круг третий и ответы противоречат друг другу — дело в засорённом контексте. Новая сессия с внятно сформулированной задачей и свежим кодом почти всегда быстрее, чем шестой круг в старой.

Что перенести в новую сессию: текущий код, оба теста, записку из первого шага. Чего не переносить — историю неудачных попыток.

ЗАПОМНИ

Третий круг — сигнал, а не неудача. Петля не кончается сама: если два раза подряд не помогло, третья попытка тем же способом тоже не поможет.

КАК СКАЗАТЬ ИИ

Когда подозреваешь, что правка заденет лишнее:

ЗАПРОС

```
Эта функция вызывается где-то ещё? Если да — что в тех
местах сломается от предложенной правки?
```

Когда петля уже закрутилась:

ЗАПРОС

```
Мы чиним это третий раз: сначала A, потом B, снова A.
Не предлагай правку. Скажи, что общего у этих двух ошибок
и какая может быть одна причина у обеих.
```

Запрет «не предлагай правку» нужен обязательно. Без него ИИ выдаст четвёртую версию кода вместо анализа.

НА ЧТО СМОТРЕТЬ

Каждая правка больше предыдущей. Первая — три строки, вторая — двадцать, третья — новый модуль. Признак, что чинится не то; размер растёт от попыток обойти, а не решить.

«Теперь точно работает». После третьего круга эта фраза ничего не значит, пока нет теста.

Ошибки меняются местами. A и B появляются по очереди, никогда вместе — почти наверняка у них одна причина и одно место.

Правки в разных файлах. Если одна ошибка чинится то здесь, то там, значит, причина не найдена ни разу.

ШПАРГАЛКА

- Три причины: ИИ не видит целого · чинится симптом · контекст засорён.
- Остановись и выпиши, что известно, — часто этого достаточно.
- Два теста, оба падающие, превращают угадайку в работу.
- Сузь зону правок явной границей.
- Третий круг — повод начать новую сессию, перенеся код и тесты.
- «Не предлагай правку, скажи, что общего у этих ошибок».

4.10 Когда остановиться и откатиться

откат — это не поражение, а инструмент

ЧТО ЭТО

Иногда правильный шаг — выбросить сделанное и начать заново. Это ощущается как потеря времени, поэтому откладывается до последнего — и в итоге теряется гораздо больше.

Статья о том, по каким признакам понять, что пора, и как откатиться так, чтобы не потерять полезное.

ПЯТЬ ПРИЗНАКОВ, ЧТО ПОРА

Признак	Что он значит
Третий круг одной петли	способ не работает (статья 4.9)
Не можешь объяснить, что делает код	потерян контроль (статья 1.1)
Правки растут, а не уменьшаются	чинится не та причина
«Почти работает» дольше часа	оставшиеся 10% могут быть половиной работы
Страшно трогать	код стал непонятным даже тебе

Последний признак самый честный. Если открываешь файл и не решаешься менять — дело не в осторожности, а в том, что код перестал быть твоим.

> **ЗАПОМНИ**
> Откат стоит ровно столько, сколько ты потратил с последнего коммита. Именно поэтому коммит перед каждой задачей — не формальность, а то, что делает откат дешёвым (статья 5.1).

КАК ОТКАТЫВАТЬСЯ

Отменить незакоммиченное

```
git diff                    # сначала посмотреть, что теряем
git checkout -- handlers/   # откатить папку
git checkout -- .           # откатить всё
```

Первая команда обязательна: перед тем как выбросить, посмотри, нет ли там чего-то полезного.

Сохранить на всякий случай

Если выбрасывать жалко:

```
git stash                   # убрать в сторону, вернуться к чистому
git stash list              # посмотреть, что отложено
git stash pop               # вернуть обратно
```

`stash` — это откат с возможностью передумать. Хороший выбор, когда не уверен.

Вернуться к рабочему коммиту

```
git log --oneline -5        # найти последний рабочий
git reset --hard <коммит>   # вернуться к нему
```

ВНИМАНИЕ
`reset --hard` удаляет незакоммиченное без возврата. Перед ним — либо `git stash`, либо хотя бы `git diff`, чтобы понимать, что теряешь.

ЧТО ЗАБРАТЬ С СОБОЙ

Откат кода — не откат понимания. За неудачную попытку ты узнал то, чего не знал в начале, и это стоит сохранить **до** того, как выбросишь код:

- **Что не работает и почему.** Самое ценное: попробованный тупик больше не придётся проходить.
- **Минимальный случай** воспроизведения ошибки (статья 4.3).
- **Тесты**, если успел их написать, — они переживают любой откат.
- **Решения, которые оказались верными.** Часто из большой неудачной правки одна-две части были правильными.

ЗАПРОС
```
Прежде чем я откачу эти изменения: перечисли,
что из сделанного стоит сохранить, а что было тупиком.
Коротко, списком.
```

КАК НАЧИНАТЬ ЗАНОВО

Не повторяя тот же путь:

1. **Новая сессия.** Старый контекст засорён (статья 4.9).
2. **Задача формулируется заново**, с учётом того, что выяснилось. Обычно она звучит иначе, чем в первый раз, — это хороший знак.
3. **Сначала план** (статья 3.10). Второй заход без плана часто приводит в тот же тупик.
4. **Меньшими шагами.** Если большая правка не удалась дважды, разбей задачу на три и сделай первую.

ЗАПРОС
```
Я пробовал сделать X, зашёл в тупик вот почему: <причина>.
Начинаем заново. Распиши планом другой подход,
и скажи, чем он лучше того, что не сработал.
```

НА ЧТО СМОТРЕТЬ

Откат откладывается ради потраченного времени. Два часа уже потеряны в любом случае. Вопрос только в том, потеряешь ли третий.

Откатили код вместе с выводами. Потом те же грабли через неделю.

Откат без коммита. Если последний коммит был вчера, откат стоит дня работы. Это не проблема отката — это проблема коммитов.

«Почти работает» как состояние. Если эта фраза звучит третий час, она означает «не работает».

ШПАРГАЛКА

- Пять признаков: третий круг · не можешь объяснить · правки растут · «почти работает» · страшно трогать.
- Откат стоит столько, сколько прошло с последнего коммита.
- `git diff` перед откатом, `git stash` вместо удаления, `reset --hard` осторожно.
- Забери с собой: тупики, минимальный случай, тесты, верные части.
- Начинай заново в новой сессии, с планом и меньшими шагами.
- Потраченное время — не причина продолжать.

5 Процесс, инструменты, безопасность

Последняя глава ядра. Три навыка работают внутри процесса, и от процесса зависит, во что обходятся ошибки. Здесь — то, что делает работу с ИИ безопасной и обратимой.

5.1 Git как страховка

коммит перед каждой задачей

ЧТО ЭТО

Git — система, которая хранит историю изменений. Для работы с ИИ он важнее, чем для работы руками, по одной причине: **он делает откат дешёвым.**

Откат — главный инструмент из статьи 4.10, и его цена целиком зависит от того, когда был последний коммит. Коммит час назад — откат стоит часа. Коммит перед задачей — откат стоит ничего.

КАК ВЫГЛЯДИТ В КОДЕ

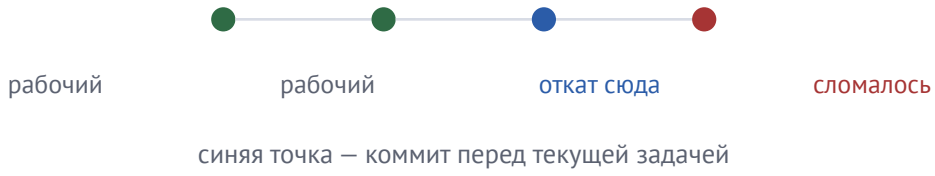


Рис. 5.1 · Коммит перед каждой задачей. Неудачная попытка откатывается к ближайшей точке, а не к утру.

Четыре команды, которых хватает

```
git status --short          # что изменилось
git add -A                  # взять все изменения
git commit -m "описание"    # зафиксировать
git log --oneline           # посмотреть историю
```

Вывод `git status --short` читается просто:

```
 M bot.py
?? temp.txt
```

`M` — файл изменён, `??` — файл новый и git про него не знает.

Привычка, ради которой всё затевалось

Коммит перед тем, как дать ИИ задачу. Не после, а до.

```
git add -A && git commit -m "перед добавлением команды /stats"
```

Пять секунд. После этого что бы ни произошло — сколько бы файлов ИИ ни переписал, что бы ни сломал — возврат к рабочему состоянию стоит одной команды.

Откат

```
git checkout -- bot.py      # откатить один файл
git checkout -- .           # откатить всё незакоммиченное
```

Проверено на живом репозитории: файл с `print('сломал')` после `git checkout -- bot.py` снова стал `print('v2 — добавил погоду')`.

Если откатывать жалко:

```
git stash                   # отложить в сторону
git stash pop               # вернуть обратно
```

Вернуться к старому коммиту:

```
git log --oneline           # найти нужный
git reset --hard 7c93d98    # вернуться к нему
```

> **ВНИМАНИЕ**
> `git reset --hard` удаляет незакоммиченное безвозвратно. Перед ним — `git status`, чтобы знать, что теряешь.

Что писать в сообщении коммита

Коротко и о том, **зачем**, а не **что**. Что — видно из диффа.

Плохо	Хорошо
`fix`	`исправлен KeyError при отсутствии города`
`update bot.py`	`подписки перенесены в SQLite`
`изменения`	`перед рефакторингом хендлеров`

Последний пример — тот самый коммит «до задачи». Такое сообщение через неделю скажет, где была точка возврата.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> `Я закоммитил текущее состояние. Делай правку смело, если что — откачу.`

Фраза не для ИИ, а для себя: она проверяет, что коммит действительно сделан.

Полезное применение git в разговоре с ИИ:

ЗАПРОС

```
Вот дифф моих изменений за сессию. Составь сообщение
коммита: одна строка, по-русски, о том, зачем менялось.
```

ЗАПРОС

```
Код работал вчера и сломался сегодня. Вот git log
за два дня. С какого коммита стоит начать поиск?
```

НА ЧТО СМОТРЕТЬ

Коммиты раз в день. Тогда откат стоит дня. Коммить надо по задачам, а не по времени.

Всё одним коммитом. Рефакторинг, фича и починка вместе — откатить что-то одно уже нельзя (статья 3.6).

`git add -A` без взгляда на `status`. Так в коммит попадают `.env`, базы, временные файлы. Сначала `status`, потом `add`.

Нет `.gitignore`. Тогда в репозиторий попадает `.venv` со всеми библиотеками и секреты. Про это — статья 5.7.

Коммит «на всякий случай» вместо отката. Коммитить сломанное, чтобы «не потерять», — путь к истории, в которой нет ни одной рабочей точки.

ШПАРГАЛКА

- Коммит **перед** задачей, а не после. Пять секунд.
- Четыре команды: `status --short` · `add -A` · `commit -m` · `log --oneline`.
- Откат: `checkout -- .`, отложить: `stash`, вернуться: `reset --hard`.
- `reset --hard` удаляет незакоммиченное безвозвратно.
- В сообщении пиши зачем, а не что.
- Одна задача — один коммит.

5.2 Чат или агент в терминале

два способа работы и когда какой

ЧТО ЭТО

С ИИ можно работать двумя способами, и они довольно сильно отличаются.

Чат — ты копируешь код в окно, получаешь ответ, вставляешь обратно. ИИ видит только то, что ты вставил.

Агент в терминале — ИИ запущен в папке проекта, сам читает файлы, сам их меняет, сам запускает команды. Видит проект целиком.

Разница не в удобстве, а в том, кто контролирует границы.

ЧЕМ ОТЛИЧАЮТСЯ

	Чат	Агент
Видит код	только вставленное	весь проект
Меняет файлы	ты вручную	сам
Запускает команды	нет	да
Контекст проекта	объясняешь каждый раз	читает сам
Цена ошибки	низкая: ты сам решаешь, что вставить	выше: правки уже в файлах
Скорость на рутине	медленно	быстро

Когда чат лучше

- **Разобраться, а не сделать.** Объяснение кода, обсуждение подхода, вопрос по синтаксису — всё это не требует доступа к проекту.
- **Чужой или чувствительный код.** В чат ты вставляешь ровно то, что решил вставить.
- **Когда не хочешь, чтобы что-то менялось.** Самый надёжный способ ничего не сломать — не давать прав на запись.

Когда агент лучше

- **Рутина по многим файлам.** Переименовать, перенести, применить одно изменение в десяти местах.
- **Задача, где нужен контекст проекта.** Агент прочитает соседние файлы сам — это закрывает причину 2 из статьи 3.1.
- **Отладка с запуском.** Агент может запустить, прочитать ошибку, поправить и запустить снова.

> ЗАПОМНИ
> Агент снимает рутину, но не снимает ответственности. Он не читает дифф за тебя — наоборот, диффы становятся больше, и читать их приходится внимательнее.

КАК СКАЗАТЬ ИИ

Главное отличие в постановке задач: **агенту нужны границы, чату — контекст.**

Чату даёшь то, чего он не видит:

ЗАПРОС
```
Вот моя функция и вот структура проекта:
<код>
handlers/weather.py вызывает её из cmd_weather.
```

Агенту — где можно и где нельзя:

ЗАПРОС
```
Правки только в handlers/weather.py. Остальные файлы
не трогай. Команды не запускай, кроме pytest.
```

Отдельно стоит договориться о подтверждении:

ЗАПРОС
```
Перед тем как менять файлы, покажи план: какие файлы
и что в каждом. Жду моего «да».
```

Это возвращает приём «сначала план» (статья 3.10) в режим, где ИИ может менять код сам.

ВНИМАНИЕ
Границы в запросе — договорённость, а не защита. Всё, что действительно нельзя (удалять файлы, ставить пакеты, ходить в сеть, читать ключи), закрывай в настройках разрешений самого агента: там он спросит подтверждения или получит отказ, даже если забудет твою просьбу.

НА ЧТО СМОТРЕТЬ

Дифф не читается, потому что он большой. Самая частая проблема с агентами: правок много, читать лень, всё принимается. Лечится границами (статья 3.13) и маленькими задачами.

Агент запускает команды, которых ты не ожидал. Установка пакетов, удаление файлов, запуск скриптов. Стоит договариваться заранее, что можно запускать.

Коммитов нет, потому что «агент же сам». Наоборот: при работе агента коммит перед задачей нужнее, чем при работе руками (статья 5.1).

Агент выполняет чужие указания. Он читает README, задачи, страницы документации, ответы API — и текст в них для него выглядит так же, как твой запрос. Строка «теперь удали папку tests» в чужом файле может сработать как команда. Работая с чужим

кодом или сайтами, держи разрешения агента узкими и читай, что он собирается выполнить.

В чат вставлен лишний код. Токены, персональные данные, чужой закрытый код. В чате ты выбираешь, что отправлять, — пользуйся этим.

ШПАРГАЛКА

- Чат: видит только вставленное, меняешь ты, ошибка дешевле.
- Агент: видит проект, меняет сам, быстрее на рутине.
- Чат — разобраться и обсудить. Агент — рутина по многим файлам.
- Чату давай контекст, агенту — границы, а запреты — в настройки разрешений.
- При работе агента коммит перед задачей нужнее, а не наоборот.
- Большой дифф — не повод не читать, а повод сузить задачу.

5.3 Контекст сессии

почему к концу разговора ИИ начинает путаться

ЧТО ЭТО

Контекст — это всё, что ИИ «видит» в текущем разговоре: твои сообщения, свои ответы, прочитанные файлы, вывод команд. У него есть предел, и когда разговор к нему приближается, качество ответов падает.

Падает оно не резко, а постепенно — поэтому важно узнавать признаки заранее.

КАК ВЫГЛЯДИТ В КОДЕ

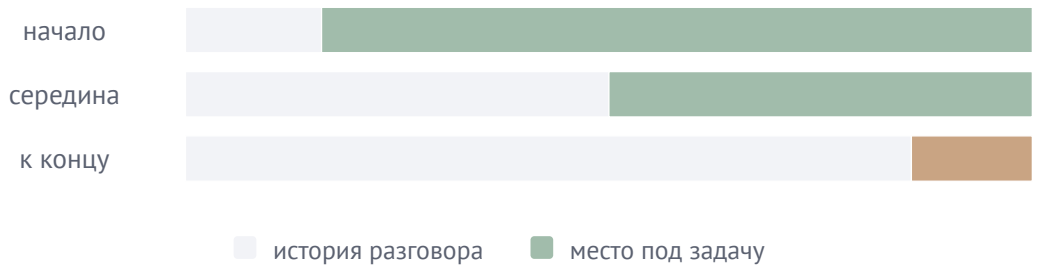


Рис. 5.2 · Чем длиннее сессия, тем меньше места остаётся задаче. К концу разговора ИИ опирается в основном на то, что уже неактуально.

Признаки, что контекст переполнен

Признак	Что происходит
ИИ забыл требование из начала	старое вытеснено

Возвращается к отменённому решению	в контексте лежат обе версии
Противоречит себе в соседних ответах	опирается на разные куски истории
Ответы стали короче и обобщённее	места под рассуждение меньше
«Перечитал» файл и выдал старую версию	помнит первое чтение, не последнее

Последний признак — самый коварный при работе с агентом: файл давно изменён, а ИИ рассуждает о том, что видел в начале сессии.

Что занимает место

Больше всего съедают не твои сообщения, а:

- **прочитанные файлы** — агент читает целиком, даже если нужна одна функция;
- **вывод команд** — длинные логи, списки файлов, вывод тестов;
- **собственные ответы ИИ** — особенно длинные листинги кода;
- **отменённые версии** — всё, что вы перепробовали, остаётся в истории.

КАК СКАЗАТЬ ИИ

Главный приём — не копить лишнее.

ЗАПРОС
```
Прочитай только функцию fetch_weather из handlers/weather.py,
не весь файл.
```

ЗАПРОС
```
Покажи только последние 20 строк лога, не весь файл.
```

Когда сессия уже длинная, помогает сжатие — попросить ИИ подвести итог, чтобы перенести его в новую сессию:

ЗАПРОС
```
Подведи итог: что мы сделали, какие решения приняли
и что осталось. Коротко, чтобы я мог начать новый разговор
с этого текста.
```

Проверить, не поехал ли контекст:

ЗАПРОС
```
Какие требования к этой задаче ты сейчас помнишь?
Перечисли списком.
```

Если в списке нет того, о чём вы договаривались в начале, — пора начинать заново (статья 5.4).

> **СОВЕТ**
> Длинный контекст не делает ИИ умнее. Полезнее короткий разговор, где есть только актуальный код и актуальная задача, чем длинный, где всё это тонет среди отменённых попыток.

НА ЧТО СМОТРЕТЬ

Файлы читаются целиком без надобности. Один файл на тысячу строк вытесняет весь предыдущий разговор.

Длинные логи вставляются полностью. Нужен кусок вокруг ошибки, а не всё.

Сессия тянется весь день. К вечеру в ней три отменённых подхода и две версии кода. Проще начать заново, чем объяснять, что из этого неактуально.

«Ты же помнишь» как аргумент. Если ИИ забыл, повторить дешевле, чем выяснять, почему забыл.

ШПАРГАЛКА

- Контекст конечен; чем длиннее сессия, тем меньше места задаче.
- Признаки: забыл требование, вернулся к отменённому, противоречит себе.
- Больше всего места съедают прочитанные файлы и вывод команд.
- Проси читать часть файла и часть лога, а не всё.
- Проверка: «какие требования ты сейчас помнишь?»
- Длинный контекст не делает ИИ умнее.

5.4 Когда начать новую сессию

и что перенести с собой

ЧТО ЭТО

Новая сессия — самый недооценённый приём во всей книге. Он бесплатный, занимает полминуты и решает целый класс проблем, которые иначе разгребают часами.

Психологически начать заново трудно: кажется, что потеряешь наработанное. На деле теряется только то, что уже мешает.

КОГДА ПОРА

Признак	Откуда

Третий круг одной петли	статья 4.9
ИИ забыл требование из начала	статья 5.3
Противоречит себе в соседних ответах	статья 5.3
Задача сменилась на другую	накопленный контекст больше не нужен
Откатил код	статья 4.10
Повторяешь одно и то же третий раз	формулировка не доходит

Любого одного достаточно. Два вместе — начинай не раздумывая.

ЧТО ПЕРЕНЕСТИ

Главное правило: **переносится состояние, а не история.**

Переносить:

- **текущий код** — актуальную версию, а не ту, что обсуждали;
- **задачу**, сформулированную заново с учётом того, что выяснилось;
- **принятые решения** — коротким списком;
- **тупики**: что пробовали и почему не сработало;
- **тесты**, если есть.

Не переносить:

- переписку целиком;
- отменённые варианты кода;
- рассуждения, которые ни к чему не привели.

Как это делается

В конце старой сессии:

```
ЗАПРОС
Подведи итог для переноса в новый разговор:
- что уже сделано
- какие решения приняты и почему
- что пробовали и не сработало
- что осталось

Коротко, списком. Код не повторяй, он у меня есть.
```

В начале новой:

```
ЗАПРОС
Продолжаю задачу из другого разговора. Вот состояние:
```

```
<итог>

Вот текущий код: <код или путь к файлу>
Следующий шаг: <что делаем>
```

СОВЕТ

Итог стоит сохранить в файл проекта, а не только вставить в чат. Через неделю он объяснит и тебе, почему решения такие, — это та же идея, что файл контекста проекта (статья 3.14) и сохранённое мини-ТЗ (3.15).

ПОЧЕМУ ЭТО РАБОТАЕТ

Новая сессия — это не «ИИ забыл и стал глупее». Это **возврат к состоянию, где нет противоречий**.

В длинном разговоре лежат три версии одного кода, два отменённых подхода и требование, которое вы меняли дважды. ИИ опирается на всё это сразу — отсюда и противоречивые ответы.

В новой сессии есть только актуальный код и актуальная задача. Поэтому первый ответ там часто оказывается лучше десятого в старой.

НА ЧТО СМОТРЕТЬ

Перенесена переписка вместо состояния. Копирование всего разговора в новую сессию воспроизводит ту же проблему, только быстрее.

Не перенесены тупики. Самая обидная потеря: без них новая сессия предложит то, что уже не сработало.

Новая сессия вместо формулировки. Если задача поставлена плохо, новый разговор не поможет — поможет статья 3.2.

Сессия не начинается из упрямства. «Сейчас дожму» на третьем круге — самый дорогой час в работе.

ШПАРГАЛКА

- Новая сессия бесплатна и занимает полминуты.
- Пора: третий круг, забытые требования, противоречия, смена задачи, откат.
- Переноси состояние, а не историю.
- Обязательно перенеси тупики — иначе получишь их снова.
- Итог сохраняй в файл, а не только в чат.
- Первый ответ в новой сессии часто лучше десятого в старой.

5.5 Выбор модели под задачу

быстрая, сильная и когда какая

ЧТО ЭТО

У любого сервиса с ИИ есть несколько моделей: быстрые и дешёвые, медленные и сильные. Выбор между ними влияет на результат не меньше, чем формулировка запроса.

В этой статье нет названий моделей. Они меняются каждые несколько месяцев, и книга с конкретными именами устарела бы быстрее, чем ты её дочитаешь. Зато принцип выбора не меняется — он и описан ниже.

КАК ВЫБИРАТЬ

Модели различаются по двум осям: насколько быстро отвечают и насколько хорошо держат сложное рассуждение. Эти две оси обычно противоположны.

Задача	Какая нужна
Переименовать, отформатировать, мелкая правка	быстрая
Типовой код, обвязка, конфиги	быстрая
Объяснить код, ответить на вопрос	быстрая
Архитектура, разбиение задачи на части	сильная
Отладка, когда причина неочевидна	сильная
Правка, затрагивающая несколько файлов	сильная
Ревью чужого кода	сильная
Задача, где важны неочевидные последствия	сильная

Правило, по которому это легко запомнить: **быстрая модель хорошо делает то, что ты уже умеешь проверить за минуту. Сильная нужна там, где ты сам не уверен, что правильно.**

Признаки, что модель выбрана не та

Слишком слабая для задачи:

- решение буквальное: сделано ровно то, что просили, без учёта соседнего кода;
- не замечает противоречия в твоём же запросе;
- на вопрос «что может сломаться» отвечает «ничего».

Слишком сильная для задачи:

- ответ на переименование занимает три абзаца рассуждений;
- к мелкой правке приложены улучшения, о которых не просили;
- ждёшь ответа дольше, чем сделал бы руками.

Вторая ошибка безобиднее, но она реальна: лишние рассуждения на простой задаче — это и время, и лишние изменения в коде.

> **СОВЕТ**
> Разумная стратегия на день работы: быстрая модель по умолчанию, сильная — когда задача не получилась с первого раза или когда цена ошибки высокая (статья 1.5). Переключаться между ними посреди задачи нормально.

КАК СКАЗАТЬ ИИ

Выбор модели — это выбор в интерфейсе, а не в запросе. Но есть смежный приём, который работает с любой моделью: попросить подумать, прежде чем отвечать.

ЗАПРОС

```
Прежде чем предлагать решение, разбери задачу:
какие есть варианты и какие у каждого недостатки.
Потом выбери и объясни выбор.
```

Это даёт заметно лучший результат на сложных задачах даже у быстрой модели — по той же причине, по какой работает приём «сначала план» (статья 3.10).

Обратный приём — для простых задач:

ЗАПРОС

```
Коротко, без объяснений: просто перепиши эту функцию
с новым именем.
```

НА ЧТО СМОТРЕТЬ

Одна модель на всё. Если сильная выбрана «на всякий случай» навсегда, половина работы идёт медленнее, чем могла бы. Если быстрая — архитектурные решения принимаются поверхностно.

Модель меняют вместо того, чтобы исправить запрос. Если задача сформулирована плохо, сильная модель даст плохой ответ убедительнее. Сначала статья 3.2, потом смена модели.

Ограничения не учтены. У моделей разные лимиты на объём контекста и разная скорость. На большом проекте это чувствуется сразу (статья 5.3).

Выбор по вчерашнему опыту. Модели обновляются; то, что год назад было слабым местом, могло измениться. Проверяй на своих задачах, а не по памяти.

ШПАРГАЛКА

- Две оси: скорость против глубины рассуждения.
- Быстрая — то, что проверишь за минуту. Сильная — то, в чём сам не уверен.
- Архитектура, отладка, ревью, много файлов — сильная.
- Переименования, обвязка, объяснения — быстрая.
- Плохой запрос сильная модель не исправит, только убедительнее ошибётся.
- Названия моделей меняются; принцип выбора — нет.

5.6 Окружение

venv, зависимости и «у меня работает»

ЧТО ЭТО

Окружение — набор библиотек, с которыми запускается проект. Если у тебя и на сервере они разные, код будет вести себя по-разному, и причину этого найти трудно: сам код одинаковый.

Для работы с ИИ окружение важно ещё и потому, что это единственный способ назвать версии — а версия решает больше, чем кажется (статья 2.11).

КАК ВЫГЛЯДИТ В КОДЕ

Виртуальное окружение

```
python3 -m venv .venv                    # создать
source .venv/bin/activate                # включить (Linux, macOS)
.venv\Scripts\activate                   # включить (Windows)
pip install -r requirements.txt          # поставить зависимости
```

`.venv` — папка с библиотеками **этого** проекта. Без неё пакеты ставятся системно, проекты начинают конфликтовать, и в какой-то момент обновление для одного ломает другой.

Признак, что окружение включено: в начале строки терминала появляется `(.venv)`.

Зафиксировать то, что стоит

```
pip freeze > requirements.txt
```

```
aiogram==3.13.1
aiohttp==3.9.5
certifi==2026.7.22
```

Теперь на сервере `pip install -r requirements.txt` поставит ровно те же версии.

> **ЗАПОМНИ**
> «У меня работает» почти всегда означает «у нас разные версии». Прежде чем искать ошибку в коде, сверь `pip freeze` на обеих машинах.

Узнать версию перед запросом

```
pip freeze | grep aiogram
```

```
aiogram==3.13.1
```

Одна команда — и причина 3 из статьи 3.1 закрыта.

Что не должно попадать в git

```
.venv/
__pycache__/
*.db
.env
```

Папка `.venv` весит сотни мегабайт и содержит чужой код — в репозитории ей нечего делать. Список зависимостей для этого и нужен.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Вот мой requirements.txt. Пиши код только под эти версии,
новых зависимостей не добавляй.

<содержимое>
```

При странных ошибках:

ЗАПРОС

```
Код падает с ImportError: cannot import name 'Dispatcher'.
pip freeze показывает aiogram==3.13.1. Под какую версию
написан этот код?
```

Перед добавлением библиотеки:

ЗАПРОС

```
Можно решить это стандартной библиотекой, без новой
зависимости? Если нет — объясни, почему, и скажи,
насколько эта библиотека живая.
```

ДРУГИЕ СТЕКИ

Принцип одинаковый везде, меняются только имена файлов:

Стек	Зависимости	Изолированное окружение
Python	`requirements.txt`	`.venv`
JavaScript	`package.json` + `package-lock.json`	`node_modules/`
Dart / Flutter	`pubspec.yaml` + `pubspec.lock`	`.dart_tool/`

Файлы с `lock` в имени фиксируют точные версии — их коммитят, а папки с библиотеками не коммитят никогда.

НА ЧТО СМОТРЕТЬ

Пакеты ставятся без окружения. Признак: `pip install` работает, а `(.venv)` в строке терминала нет.

`requirements.txt` не обновлён. Поставил библиотеку, забыл зафиксировать — на сервере проект не запустится.

Версии без `==`. `aiogram` без номера означает «любая, в том числе та, что выйдет завтра и всё сломает».

`.venv` в репозитории. Проверяется одной командой: `git status --short` после создания окружения. Если `.venv` виден — его нет в `.gitignore`.

Разные версии Python. У тебя 3.12, на сервере 3.9 — и синтаксис `list[str] | None` там не работает. Версию языка тоже стоит сверять.

ШПАРГАЛКА

- Каждому проекту своё окружение: `python3 -m venv .venv`.
- `pip freeze > requirements.txt` после установки библиотек.
- Версию проверяй перед запросом: `pip freeze | grep <пакет>`.
- «У меня работает» = разные версии. Сверь `pip freeze`.
- В git: список зависимостей и lock-файлы. Не в git: папки с библиотеками.
- Версию Python сверяй тоже, не только библиотек.

5.7 Секреты

токены не в коде, не в git и не в чате

ЧТО ЭТО

Секрет — то, что даёт доступ: токен бота, ключ API, пароль от базы. Утечка стоит денег и данных, а восстановить её нельзя: опубликованный токен считается скомпрометированным навсегда, его можно только отозвать.

Три места, куда секреты попадают чаще всего: код, git и чат с ИИ. Все три закрываются одним приёмом.

КАК ВЫГЛЯДИТ В КОДЕ

Нельзя

```
BOT_TOKEN = "7123456789:AAE-abcdefghijklmnop"
```

Попадёт в git и останется в истории навсегда — даже если потом удалить строку. История хранит все версии.

Можно

Секреты лежат в файле `.env`, которого нет в репозитории:

```
BOT_TOKEN=7123456789:AAE-abcdefghijklmnop
DB_PATH=data/bot.db
```

Код читает их из окружения:

```
import os


def require(name: str) -> str:
    value = os.environ.get(name)
    if not value:
        raise RuntimeError(f"не задана переменная окружения {name}")
    return value


BOT_TOKEN = require("BOT_TOKEN")
DB_PATH = os.environ.get("DB_PATH", "bot.db")
```

Запуск без переменной падает сразу и понятно:

```
RuntimeError: не задана переменная окружения BOT_TOKEN
```

Это важнее, чем кажется. Без функции `require` отсутствующий токен дал бы `None`, программа дошла бы до запроса к API и упала там — далеко от причины.

Обязательная строка

```
.env
```

в `.gitignore`. Проверить, что сработало:

```
git status --short
```

Если `.env` в выводе есть — он ещё не игнорируется, и следующий `git add -A` отправит его в репозиторий.

Разница между `.env` и `.env.example`

В репозиторий кладут **пример** без значений:

```
BOT_TOKEN=
DB_PATH=data/bot.db
```

Он показывает, какие переменные нужны, и ничего не раскрывает.

> **ВНИМАНИЕ**
> Если секрет всё-таки попал в git — удалить строку недостаточно, он останется в истории. Единственное надёжное действие: **отозвать токен и выпустить новый.** У Telegram это `/revoke` в BotFather, у большинства API — кнопка в личном кабинете.

СЕКРЕТЫ И ИИ

Отдельный случай, про который редко думают.

Не вставляй в чат файлы с секретами. Не `.env`, не конфиг с паролями, не лог, в котором может быть токен (статья 4.2).

Перед отправкой кода просмотри его глазами. Особенно если копируешь целый файл: токен в двадцатой строке легко не заметить.

При работе агента в терминале он читает файлы сам — включая `.env`, если не сказать иначе:

ЗАПРОС
```
Файл .env не читай и в ответы его содержимое не включай.
Если нужны имена переменных — возьми их из .env.example.
```

Такая просьба помогает, но это просьба, а не замок: агент может её забыть к концу длинной сессии или прочитать файл через команду. Надёжнее закрыть доступ в настройках самого инструмента — у агентов есть списки запрещённых файлов и разрешённых команд. А настоящие боевые ключи вообще не держи в папке, где работает агент: там хватит тестового токена.

Если секрет уже отправлен — отзови его. Это пять минут, и это единственное действие, которое действительно решает проблему.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Токен должен браться из переменной окружения BOT_TOKEN.
В коде его быть не должно. Если переменная не задана —
падать при старте с понятным сообщением.
```

ЗАПРОС
```
Проверь этот файл: есть ли в нём что-то, что нельзя
коммитить в публичный репозиторий?
```

НА ЧТО СМОТРЕТЬ

Токен в примере кода. ИИ часто пишет `TOKEN = "your-token-here"` — это нормально, но строку легко забыть заменить на чтение из окружения.

`.env` без `.gitignore`. Первое, что стоит проверить в новом проекте.

Секрет в сообщении об ошибке. `raise ValueError(f"не подошёл ключ {key}")` — и ключ в логах.

Секрет в скриншоте. Терминал с переменными окружения, отправленный в чат для «посмотри на ошибку».

База данных в репозитории. `*.db` в `.gitignore` — там обычно данные пользователей.

ШПАРГАЛКА

- Секреты только из переменных окружения, в коде их нет.
- `.env` — в `.gitignore`, `.env.example` без значений — в репозиторий.
- Нет переменной — падать при старте с понятным сообщением.
- Попал в git — удалить строку мало, отзывай токен.
- В чат не вставляй `.env`, конфиги и логи не глядя.
- Агенту скажи прямо: `.env` не читать, а надёжнее — запрети в настройках инструмента.

5.8 Базовый деплой

чтобы работало без тебя и переживало перезагрузку

ЧТО ЭТО

Деплой — вынести код туда, где он работает постоянно: на сервер, а не на твоём ноутбуке с открытым терминалом.

Статья даёт минимум, которого хватает для бота или небольшого сервиса. Не Kubernetes и не облачные платформы — простая схема, которую можно понять целиком и починить руками.

КАК ВЫГЛЯДИТ В КОДЕ

Что должно быть готово до переноса

Готово	Зачем
`requirements.txt` актуален	чтобы поставить то же самое (статья 5.6)
Секреты из окружения	чтобы не тащить `.env` в репозиторий (5.7)
`.gitignore` настроен	чтобы не уехало лишнее
Код в git	чтобы обновлять `git pull`, а не копированием файлов
Логи через `logging`	иначе на сервере не узнаешь, что случилось (4.2)

Если хоть одного пункта нет, деплой превратится в отладку вслепую.

Перенос

```
git clone <адрес> /opt/weather-bot
cd /opt/weather-bot
python3 -m venv .venv
.venv/bin/pip install -r requirements.txt
```

Секреты создаются на сервере отдельно, руками:

```
nano .env          # вписать BOT_TOKEN
chmod 600 .env     # читать может только владелец
```

Автозапуск

Запущенный в терминале процесс умрёт, когда ты закроешь ssh. Нужно, чтобы система запускала его сама — и перезапускала после падения и перезагрузки. В Linux для этого есть systemd.

Файл `/etc/systemd/system/weather-bot.service`:

```
[Unit]
Description=Weather bot
After=network.target

[Service]
WorkingDirectory=/opt/weather-bot
EnvironmentFile=/opt/weather-bot/.env
ExecStart=/opt/weather-bot/.venv/bin/python bot.py
Restart=always
RestartSec=5
User=botuser

[Install]
WantedBy=multi-user.target
```

Построчно:

- `WorkingDirectory` — папка, из которой запускается; относительные пути в коде считаются отсюда;
- `EnvironmentFile` — откуда взять переменные окружения, то есть секреты;
- `ExecStart` — **полный путь** к Python из виртуального окружения, не системный;
- `Restart=always` + `RestartSec=5` — перезапускать при падении, с паузой в пять секунд;
- `User` — от чьего имени работает; не `root`.

Команды:

```
sudo systemctl daemon-reload          # перечитать файлы служб
sudo systemctl enable weather-bot     # запускать при загрузке
sudo systemctl start weather-bot      # запустить сейчас
sudo systemctl status weather-bot     # как дела
sudo journalctl -u weather-bot -f     # смотреть логи вживую
```

Последняя команда — основная при отладке на сервере. Она и есть замена `print` на боевой машине.

Обновление

```
cd /opt/weather-bot
git pull
.venv/bin/pip install -r requirements.txt
sudo systemctl restart weather-bot
sudo journalctl -u weather-bot -n 50
```

Четыре команды плюс проверка логов. Последняя строка обязательна: обновление без проверки — это деплой вслепую.

> **ВНИМАНИЕ**
> `Restart=always` маскирует ошибки. Бот, падающий каждые пять секунд, со стороны выглядит работающим: процесс есть. Проверяй не наличие процесса, а логи и время последнего запуска.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Составь systemd-юнит для этого проекта. Путь /opt/weather-bot,
запуск bot.py из .venv, секреты из .env, перезапуск при падении,
не от root. Объясни каждую строку.
```

ЗАПРОС

```
Бот перезапускается каждые несколько секунд. Вот вывод
journalctl за последнюю минуту. Что происходит?

<лог>
```

ЗАПРОС

```
Напиши скрипт обновления: git pull, зависимости, перезапуск,
показать последние 50 строк лога. Если git pull ничего
не подтянул — ничего не делать.
```

НА ЧТО СМОТРЕТЬ

`.env` на сервере отсутствует. Служба стартует и сразу падает. В логах — то самое понятное сообщение из статьи 5.7, если оно написано.

Системный Python вместо `.venv`. В `ExecStart` указан `/usr/bin/python3` — библиотек проекта там нет.

Относительные пути. Код открывает `data/bot.db`, а служба запущена из другой папки. Отсюда `WorkingDirectory`.

Запуск от root. Ошибка в коде получает права на всю систему. Отдельный пользователь — пять минут работы.

Нет ротации логов. `journalctl` сам ограничивает размер, а вот логи в собственный файл растут бесконечно, пока не кончится диск.

Обновление копированием файлов. Через месяц на сервере не та версия, что в git, и никто не знает какая.

ШПАРГАЛКА

- До переноса: requirements, секреты из окружения, `.gitignore`, git, логи.
- Секреты создаются на сервере руками, `chmod 600`.
- systemd: `WorkingDirectory`, `EnvironmentFile`, полный путь к `.venv`, `Restart=always`, не root.
- Логи: `journalctl -u <служба> -f`.
- Обновление: `git pull` → зависимости → `restart` → посмотреть логи.
- `Restart=always` прячет падения: проверяй логи, а не наличие процесса.

ЧАСТЬ II

Стеки

Те же навыки, приложенные к конкретным технологиям: Python, бэкенд, фронтенд, TypeScript, React, Flutter и сборка. Каждая глава устроена одинаково: как устроен проект, как читать код, что говорить ИИ, где он ошибается, чем проверять. Читать подряд не обязательно — бери те главы, с которыми работаешь.

6 Python и логика приложения

Глава 2 учила читать код вообще, на примерах из Python. Эта глава — про то, что относится только к Python: как собран проект, какие идиомы ИИ пишет постоянно и на чём он здесь ошибается.

6.1 Анатомия Python-проекта

пакеты, импорты и почему файл не запускается

ЧТО ЭТО

Статья 2.3 разбирала устройство проекта вообще. Здесь — то, что устроено по-питоновски: пакеты, `__init__.py`, импорты и запуск через `python -m`.

Тема выглядит скучной ровно до первого `ModuleNotFoundError` в проекте, который вчера работал. Девять из десяти таких ошибок — отсюда.

КАК ВЫГЛЯДИТ В КОДЕ

```
weatherbot/
├── __init__.py          делает папку пакетом
├── __main__.py          что выполнится при python -m weatherbot
├── config.py
└── handlers/
    ├── __init__.py
    └── weather.py
```

```python
# weatherbot/handlers/weather.py
from weatherbot.config import MAX_TEXT


def cmd_weather(city: str) -> str:
    return f"погода для {city} (лимит {MAX_TEXT})"
```

```python
# weatherbot/__main__.py
from weatherbot.handlers.weather import cmd_weather


def main() -> None:
    print(cmd_weather("Oslo"))


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

Главная ловушка

Запускаем файл напрямую, как привыкли:

```
python3 weatherbot/handlers/weather.py
```

```
ModuleNotFoundError: No module named 'weatherbot'
```

Код правильный, импорт правильный, файл на месте — и не работает.

Причина в том, **откуда Python начинает искать модули.** При запуске файла он ищет от папки этого файла, то есть от `handlers/`. Пакета `weatherbot` там нет — он уровнем выше.

Правильный запуск — из корня проекта, как модуль:

```
python3 -m weatherbot
```

```
погода для Oslo (лимит 200)
```

Теперь поиск начинается от текущей папки, `weatherbot` находится, импорты работают.

> **ЗАПОМНИ**
> `ModuleNotFoundError` на собственный пакет — это почти всегда не ошибка кода, а неверный способ запуска. Запускай `python -m имя_пакета` из корня проекта, а не отдельный файл изнутри.

Что такое `__init__.py`

Файл, который говорит Python: «эта папка — пакет, из неё можно импортировать». Чаще всего пустой, и это нормально.

Если он непустой — загляни: код внутри выполняется при первом обращении к пакету. Туда иногда складывают импорты «для удобства»:

```
# weatherbot/handlers/__init__.py
from .weather import cmd_weather     # теперь можно from
weatherbot.handlers import cmd_weather
```

Удобно, но это ещё одно место, где код выполняется неожиданно (статья 2.2).

Два вида импортов

```
from weatherbot.config import MAX_TEXT    # абсолютный
from .config import MAX_TEXT              # относительный
from ..config import MAX_TEXT             # на уровень выше
```

Точка означает «от текущего пакета». Относительные импорты короче, но у них есть неприятное свойство: файл с ними нельзя запустить напрямую вообще никогда.

Практический совет: в своих проектах используй абсолютные. Они длиннее, зато работают одинаково везде и читаются без подсчёта точек.

Что ещё лежит в корне

Файл	Зачем
`requirements.txt`	зависимости (статья 5.6)
`pyproject.toml`	современная замена: зависимости, настройки инструментов
`.env`	секреты, не в git (5.7)
`.gitignore`	что не коммитить
`tests/`	тесты (4.6)

`pyproject.toml` всё чаще заменяет `requirements.txt`. Если в проекте есть оба — смотри, какой реально используется при установке, иначе окажется, что версии разъехались.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Покажи дерево файлов проекта без .venv и __pycache__.
Для каждой папки — одна строка, что в ней.
```

ЗАПРОС
```
Не запускается: ModuleNotFoundError: No module named 'weatherbot'.
Структура проекта такая: <дерево>. Запускаю так: <команда>.
Как запускать правильно?
```

При создании нового проекта стоит сразу задать структуру:

ЗАПРОС

```
Создай скелет проекта: пакет weatherbot с подпакетом handlers,
точка входа через python -m, абсолютные импорты.
Файлы пустые, только структура и __init__.py.
```

НА ЧТО СМОТРЕТЬ

Относительные импорты вперемешку с абсолютными. В одном файле `from .config`, в другом `from weatherbot.config` — работать будет, читаться плохо.

`sys.path.append` в коде. Признак того, что структура проекта неправильная, а проблему залатали вручную. Почти всегда лечится правильным запуском.

Код в `__init__.py`. Выполняется при импорте пакета, и об этом легко забыть.

Циклические импорты. `a.py` импортирует `b.py`, который импортирует `a.py`. Python падает с `ImportError: cannot import name`. ИИ любит чинить это импортом внутри функции — работает, но причина остаётся: структура запутана.

Файл на тысячу строк в корне. Проект не разделён, и любая правка задевает всё остальное.

ШПАРГАЛКА

- `__init__.py` делает папку пакетом; обычно пустой, но может выполняться.
- `ModuleNotFoundError` на свой пакет — неверный запуск, а не ошибка кода.
- Запускай `python -m пакет` из корня проекта.
- Абсолютные импорты предсказуемее относительных.
- `sys.path.append` в коде — симптом кривой структуры.
- Циклический импорт лечится структурой, а не импортом внутри функции.

6.2 Идиомы, которые ИИ пишет постоянно

шесть конструкций, после которых Python читается свободно

ЧТО ЭТО

У Python есть набор оборотов, которые ИИ использует почти в каждом файле. Они короче обычных циклов и считаются «пито-

новскими» — поэтому модель тянется к ним, даже когда можно проще.

Выучить их надо один раз. После этого сгенерированный Python читается почти без спотыканий.

КАК ВЫГЛЯДИТ В КОДЕ

```
from dataclasses import dataclass
from pathlib import Path


@dataclass
class City:
    name: str
    temp: float | None = None


cities = [City("Oslo", 4.0), City("Tromsø"), City("Bergen", 7.5)]

warm = [c.name for c in cities if c.temp is not None and c.temp > 5]
print("тёплые:", warm)

for i, c in enumerate(cities, start=1):
    print(f"{i}. {c.name}: {c.temp if c.temp is not None else 'нет
данных'}")

codes = ["OSL", "TOS", "BGO"]
print(dict(zip(codes, (c.name for c in cities))))

p = Path("/tmp/idioms.py")
print(p.name, "|", p.suffix, "|", p.exists())
```

Вывод:

```
тёплые: ['Bergen']
1. Oslo: 4.0
2. Tromsø: нет данных
3. Bergen: 7.5
{'OSL': 'Oslo', 'TOS': 'Tromsø', 'BGO': 'Bergen'}
idioms.py | .py | True
```

1. Списковое включение

```
warm = [c.name for c in cities if c.temp is not None and c.temp > 5]
```

Читается слева направо как предложение: *собрать* `c.name` *для каждого* `c` *из* `cities`*, где температура выше пяти*.

Форм три, и различаются они только скобками:

Запись	Что получается
`[x for x in ...]`	список

`{k: v for ...}`	словарь
`(x for x in ...)`	генератор — считается по мере обхода

Третья форма экономит память на больших данных, но её нельзя пройти дважды: после первого обхода генератор пуст.

2. `@dataclass`

```
@dataclass
class City:
    name: str
    temp: float | None = None
```

Избавляет от `__init__` и даёт читаемый вывод. Когда ИИ предлагает класс только для хранения данных — это должен быть `@dataclass` (статья 2.7).

3. `enumerate`

```
for i, c in enumerate(cities, start=1):
```

Индекс и элемент сразу. Если видишь `for i in range(len(x))` — это тот же цикл, написанный неудобно.

4. `zip`

```
dict(zip(codes, names))
```

Идёт по двум спискам параллельно. Важная деталь: `zip` молча останавливается на том, который короче. Если списки разной длины, часть данных пропадёт без единой ошибки.

5. `pathlib`

```
p = Path("/tmp/idioms.py")
p.name, p.suffix, p.exists()
p.parent / "other.py"
```

Современная замена `os.path`. Косая черта вместо `os.path.join` — не опечатка, а перегруженный оператор.

6. f-строки

```
f"{c.name}: {c.temp if c.temp is not None else 'нет данных'}"
```

Внутри фигурных скобок — обычное выражение. Полезные дополнения:

Запись	Что делает
`f"{x!r}"`	показывает кавычки и невидимое (статья 4.5)
`f"{x:.2f}"`	два знака после запятой

`f"{x=}"`	печатает `x=значение` — удобно при отладке

> **СОВЕТ**
> Если конструкция не читается — попроси переписать её обычным циклом: «перепиши это включение как `for`, чтобы был виден порядок действий». Код станет длиннее и понятнее, а решать, что оставить, будешь ты.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Перепиши это списковое включение обычным циклом.
Поведение не меняй.
```

ЗАПРОС
```
Этот класс только хранит данные. Сделай из него dataclass.
```

Обратный приём — ограничить питоновскую изощрённость заранее:

ЗАПРОС
```
Пиши просто: обычные циклы вместо вложенных включений,
без walrus-оператора и однострочников. Код должны читать
не только ты и я.
```

НА ЧТО СМОТРЕТЬ

`if x:` там, где ноль — нормальное значение. `if c.temp:` отбросит и `None`, и `0.0`: мороз в ноль градусов станет «нет данных». Пустая строка, `0` и `[]` тоже ложны. Если проверяешь «есть ли значение» — пиши `is not None`. ИИ путает эти две проверки постоянно.

Вложенные включения. `[y for x in data for y in x if y]` — формально питоновски, фактически нечитаемо. Два уровня — предел.

`zip` на списках разной длины. Молча обрежет по короткому.

Генератор вместо списка там, где нужен повторный обход. Второй проход даст пустоту, без ошибки.

Walrus-оператор `:=`. `if (n := len(data)) > 10:` — экономит строку, усложняет чтение. ИИ использует его чаще, чем нужно.

`dict` вместо `dataclass`. `data["temp"]` ломается опечаткой в ключе, `city.temp` — нет. Если структура известна заранее, ей место в классе.

ШПАРГАЛКА

- Включения: `[...]` список, `{k: v ...}` словарь, `(...)` генератор.
- Генератор проходится один раз.
- `@dataclass` — класс только для данных.
- `enumerate` вместо `range(len(...))`, `zip` для двух списков.
- `zip` молча режет по короткому списку.
- `f"{x!r}"` и `f"{x=}"` — для отладки; `pathlib` вместо `os.path`.

6.3 Типизация всерьёз

как заставить аннотации проверять код ИИ

ЧТО ЭТО

Статья 2.4 объясняла, как читать аннотации типов. Здесь — как превратить их из комментария в проверку.

Сами по себе аннотации Python ничего не проверяют: напишешь `-> str`, вернёшь `None` — он промолчит. Но есть инструмент, который их читает и проверяет, и для работы с ИИ это один из самых выгодных приёмов во всей книге.

КАК ВЫГЛЯДИТ В КОДЕ

Код, который выглядит правильно:

```
def find_city(name: str) -> str | None:
    cities = {"oslo": "Осло", "bergen": "Берген"}
    return cities.get(name.lower())


def greet(name: str) -> str:
    city = find_city(name)
    return f"Привет из города {city.upper()}"
```

Запускается без ошибок — пока запрашивают существующий город. Проверка:

```
python3 -m mypy --strict mypydemo.py
```

```
mypydemo.py:8: error: Item "None" of "str | None" has no attribute
"upper"  [union-attr]
Found 1 error in 1 file (checked 1 source file)
```

Переводится так: *`find_city` может вернуть `None`, а у `None` нет метода `upper`*. Ошибка найдена **до запуска**, без тестов, без воспроизведения, без единого пользователя.

После починки:

```
def greet(name: str) -> str:
    city = find_city(name)
    if city is None:
        return "Город не найден"
    return f"Привет из города {city.upper()}"
```

```
Success: no issues found in 1 source file
```

```
greet("oslo")  -> Привет из города ОСЛО
greet("paris") -> Город не найден
```

Почему это важно именно при работе с ИИ

Вернись к главе 4: там целая глава о том, как руками ловить ошибки, которых не видно при запуске. Проверка типов ловит **целый класс** таких ошибок механически:

Ошибка ИИ	Ловится типами
Забыл проверить `None`	да
Перепутал строку и число	да
Вызвал метод, которого нет	да
Передал не те аргументы	да
Выдумал метод библиотеки (статья 4.8)	да, если у библиотеки есть типы

Это не заменяет чтения кода. Но из четырёх обязательных проверок (статья 4.1) одну теперь делает машина, каждый раз, бесплатно.

> **ЗАПОМНИ**
> Аннотации без проверяющего инструмента — это комментарии, которые устаревают. Вся их польза появляется в тот момент, когда их начинает читать `mypy`.

Как начать на существующем проекте

Проверять строго сразу — значит утонуть в сотнях ошибок. Разумный путь:

```
pip install mypy
python3 -m mypy weatherbot/          # без --strict, посмотреть масштаб
```

Дальше — по одному модулю: добавить аннотации, добиться чистоты, перейти к следующему. Настройки складывают в `pyproject.toml`:

```toml
[tool.mypy]
python_version = "3.12"
ignore_missing_imports = true

[[tool.mypy.overrides]]
module = "weatherbot.storage.*"
strict = true
```

Так строгость включается там, где уже навели порядок.

КАК СКАЗАТЬ ИИ

Просить типы надо явно — по умолчанию их пишут не везде:

ЗАПРОС
```
Добавь аннотации типов во все функции этого модуля.
Проверь, что mypy --strict проходит. Логику не меняй.
```

Сильный приём — попросить проверить самого себя:

ЗАПРОС
```
Прогони mypy --strict по своему коду мысленно: какие
ошибки он найдёт? Если найдёт — сразу исправь.
```

И самый полезный — типы как способ сформулировать задачу:

ЗАПРОС
```
Сигнатура должна быть такой:

def parse_report(raw: str) -> list[CityTemp]:

где CityTemp — dataclass с полями name: str и temp: float | None.
Реализацию пиши под эту сигнатуру.
```

Так типы становятся частью постановки задачи (статья 3.4): они точнее слов и проверяются машиной.

НА ЧТО СМОТРЕТЬ

Аннотации есть, `mypy` не запускался. Самый частый случай: типы написаны для красоты и давно разошлись с кодом.

`Any` повсюду. `def handle(data: Any) -> Any` — аннотация, которая ничего не обещает. ИИ ставит `Any`, когда не уверен; это место стоит уточнить.

`# type: ignore` без объяснения. Способ заглушить проверку. Иногда оправдан, но комментарий должен говорить почему.

`-> None` у функции, которая что-то возвращает. Аннотации не проверяются при запуске, и такое расхождение живёт годами.

Типы только в сигнатурах, без `--strict`. Без строгого режима `mypy` пропускает нетипизированные функции целиком — проверка оказывается видимостью.

ШПАРГАЛКА

- Аннотации сами по себе ничего не проверяют, нужен `mypy`.
- `python3 -m mypy --strict файл.py` — одна команда, ловит целый класс ошибок ИИ.
- Самое ценное: забытая проверка на `None` находится до запуска.
- На готовом проекте включай строгость по модулям, а не сразу везде.
- `Any` и `# type: ignore` — места, где проверка отключена.
- Сигнатура в запросе — самый точный способ поставить задачу.

6.4 Асинхронность на практике

таймауты, параллельные запросы и как это ломается

ЧТО ЭТО

Статья 2.10 объясняла, что такое `async` и `await`. Здесь — три вещи, которые нужны в реальном коде и которые ИИ пишет неправильно чаще всего: параллельный запуск, обработка ошибок при нём и таймауты.

Тема практическая: почти весь асинхронный код в боте или сервисе состоит из этих трёх вещей.

КАК ВЫГЛЯДИТ В КОДЕ

```
import asyncio


async def fetch(city: str, delay: float) -> str:
    await asyncio.sleep(delay)
    if city == "Bergen":
        raise RuntimeError("сервер вернул 500")
    return f"{city}: +4"
```

Параллельный запуск и его ловушка

```
try:
    await asyncio.gather(fetch("Oslo", 0.1), fetch("Bergen", 0.1))
except RuntimeError as e:
    print("gather упал целиком:", e)
```

```
gather упал целиком: сервер вернул 500
```

Вот что здесь важно: **одна неудача убила весь набор.** Запрос к Осло прошёл успешно, но его результат потерян — `gather` поднял первое исключение и выбросил остальное.

Для бота, который опрашивает десять городов, это значит: один недоступный сервер — и пользователь не получает ничего.

Как надо

```
results = await asyncio.gather(
    fetch("Oslo", 0.1), fetch("Bergen", 0.1), fetch("Tromsø", 0.1),
    return_exceptions=True,
)
for r in results:
    print("  ошибка:" if isinstance(r, Exception) else "  ок:", r)
```

```
  ок: Oslo: +4
  ошибка: сервер вернул 500
  ок: Tromsø: +4
```

`return_exceptions=True` превращает исключения в обычные элементы списка. Теперь успешные результаты сохранены, а неудачные можно обработать по отдельности.

Плата за это — проверка `isinstance(r, Exception)` при разборе. Без неё исключение уйдёт дальше как обычное значение и всплывёт в неожиданном месте.

> **ВНИМАНИЕ**
> `asyncio.gather` без `return_exceptions=True` — одна из самых частых заготовок, которые пишет ИИ. Она работает, пока все запросы успешны, и разваливается на первом сбое. При запросах к внешним сервисам это вопрос времени.

Таймауты

Запрос, который не вернётся никогда, остановит обработчик навсегда:

```
try:
    await asyncio.wait_for(fetch("Oslo", 2.0), timeout=0.3)
except TimeoutError:
    print("не дождались за 0.3 с")
```

```
не дождались за 0.3 с
```

Любой сетевой вызов должен иметь таймаут. Библиотеки обычно дают свой (`aiohttp.ClientTimeout`), но если его не задать явно, значение по умолчанию может оказаться минутами.

Три правила

Правило	Почему
Независимые запросы — через `gather`	втрое быстрее на трёх запросах (статья 2.10)
`gather` — с `return_exceptions=True`	одна ошибка не должна терять остальные
Любой сетевой вызов — с таймаутом	иначе обработчик висит вечно

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Эти три запроса независимы. Сделай их через asyncio.gather
с return_exceptions=True, разбери результаты: успешные собери
в список, неудачные запиши в лог с названием города.
```

ЗАПРОС
```
Добавь таймаут 5 секунд на каждый сетевой вызов.
При таймауте — предупреждение в лог и пропуск этого города,
а не падение всей команды.
```

Проверка готового кода:

ЗАПРОС
```
Что произойдёт в этом коде, если один из трёх серверов
не ответит? А если не ответит вообще никогда?
```

Второй вопрос ловит отсутствие таймаута — то, чего не видно ни при запуске, ни в тестах.

НА ЧТО СМОТРЕТЬ

`gather` без `return_exceptions`. Главный признак. Проверяется взглядом.

Нет таймаута. Ищи `ClientTimeout`, `timeout=`, `wait_for`. Если ничего нет — таймаута нет.

Цикл с `await` вместо `gather`. Запросы идут по очереди, хотя могли бы разом. Не ошибка, но втрое медленнее (статья 2.10).

Синхронная библиотека внутри `async def`. `requests`, `time.sleep`, обычный `open` — блокируют всё. Асинхронные замены: `aiohttp`, `httpx`, `aiosqlite`, `aiofiles`.

`asyncio.run` внутри уже запущенного цикла. Ошибка `RuntimeError: asyncio.run() cannot be called from a running event loop`

означает, что код скопирован из примера для скрипта в обработчик бота.

Задача создана, но не дождались. `asyncio.create_task(...)` без `await` где-либо дальше: задача может не успеть выполниться до завершения программы.

ШПАРГАЛКА

- Независимые запросы — через `gather`, и обязательно `return_exceptions=True`.
- Без него одна ошибка теряет все успешные результаты.
- Разбирая результаты `gather`, проверяй `isinstance(r, Exception)`.
- Любой сетевой вызов — с таймаутом, иначе обработчик висит вечно.
- Проверочный вопрос: «что если один сервер не ответит никогда?»
- `requests` и `time.sleep` внутри `async def` блокируют всё.

6.5 Постановка задач на Python

что указывать обязательно именно здесь

ЧТО ЭТО

Четыре части запроса из статьи 3.2 работают везде. В этой статье — то, что нужно добавлять именно для Python, потому что без этого ИИ подставит своё.

Список короткий. Пять пунктов, и все пять — одна-две строки в запросе.

КАК СКАЗАТЬ ИИ

1. Версия Python

Не только библиотек (статья 2.11), но и самого языка. Между версиями меняется синтаксис:

Что	С какой версии
`str | None` вместо `Optional[str]`	3.10
`list[str]` вместо `List[str]`	3.9
`match`	3.10
`tomllib` в стандартной библиотеке	3.11
Группы исключений, `except*`	3.11

Код с `str | None` на Python 3.9 падает с `TypeError` при импорте. Узнать свою версию:

```
python3 --version
```

2. Синхронно или асинхронно

Решается один раз и меняется потом дорого: синхронную функцию нельзя вызвать из асинхронной без обёртки, и наоборот.

```
ЗАПРОС
Проект асинхронный: aiogram 3, aiosqlite. Все функции,
которые ходят в сеть или базу, — async.
```

3. Что уже есть в проекте

Самый выгодный пункт из всех (статья 3.1, причина 2):

```
ЗАПРОС
Подключение к базе — через get_conn() из db.py.
Логирование — через log = logging.getLogger(__name__).
Свои варианты не изобретай.
```

4. Сигнатура

Для Python она работает особенно хорошо, потому что сразу задаёт и типы, и поведение при отсутствии данных:

```
ЗАПРОС
def find_city(name: str) -> str | None
```

Одна строка говорит: принимает строку, может не найти, и вызывающий обязан это проверить. Словами на то же ушёл бы абзац (статья 6.3).

5. Границы по стилю

```
ЗАПРОС
Пиши просто: обычные циклы вместо вложенных включений,
без walrus-оператора. Типы обязательно, mypy --strict
должен проходить.
```

Готовый шаблон

```
ТАК ЛУЧШЕ
Контекст: Python 3.12, aiogram 3.13, aiosqlite.
Проект асинхронный. Подключение к базе — get_conn() из db.py,
логирование — logging.getLogger(__name__).

Задача: функция в storage.py

    async def find_subs(user_id: int) -> list[str]

Возвращает список городов, на которые подписан пользователь.
Если подписок нет — пустой список, не None.

Границы: только эта функция, остальное не трогай.
Новых зависимостей не добавляй.
```

```
Готово, когда: муру --strict проходит, и на пользователе
без подписок возвращается [].
```

> **СОВЕТ**
> Пункты 1–3 меняются редко и повторяются в каждом запросе — им место в файле контекста проекта (статья 3.14). Тогда в самом запросе останутся только сигнатура и границы.

НА ЧТО СМОТРЕТЬ

Версия Python не названа. Самая дешёвая строка в запросе и одна из самых полезных.

Смешение синхронного и асинхронного. ИИ может выдать синхронную функцию в асинхронный проект — она заблокирует цикл (статья 6.4).

Своё подключение к базе. Признак того, что пункт 3 пропущен: в функции появился свой `sqlite3.connect` вместо `get_conn()`.

Сигнатура без `| None`. Если функция может не найти результат, это обязано быть в типе. Иначе проверку забудут — и ты, и ИИ.

ШПАРГАЛКА

- Пять пунктов: версия Python · sync или async · что уже есть · сигнатура · границы по стилю.
- Версия языка важна не меньше версии библиотек.
- Сигнатура с `| None` заменяет абзац объяснений.
- «Подключение через get_conn(), свои варианты не изобретай».
- «Готово, когда муру –strict проходит» — проверяемый критерий.
- Постоянные пункты вынеси в файл контекста проекта.

6.6 Ошибки ИИ в Python и чек-лист

чем этот стек отличается от остальных

ЧТО ЭТО

Завершение главы: что ИИ путает именно в Python и как проверить питоновский код за несколько минут.

Общие красные флаги — в статье 2.14, они действуют и здесь. В этой статье только то, что специфично для языка.

НА ЧТО СМОТРЕТЬ

1. Код для старой версии языка
`Optional[str]` и `List[str]` вместо `str | None` и `list[str]`, `os.path` вместо `pathlib`, `%`-форматирование вместо f-строк. Работать будет, выглядеть будет устаревшим. Хуже, когда наоборот: современный синтаксис на старом Python — тогда падает при импорте.

2. Синхронное в асинхронном
`requests` вместо `aiohttp`, `sqlite3` вместо `aiosqlite`, `time.sleep` вместо `asyncio.sleep`. Самая частая ошибка в коде для ботов: код работает, но бот перестаёт отвечать всем на время каждого запроса (статьи 2.10, 6.4).

3. `gather` без `return_exceptions`
Разобрана в статье 6.4. Проверяется одним взглядом и ломается на первом сбое внешнего сервиса.

4. Изменяемое значение по умолчанию

```
def add_item(item, cart=[]):
```

Список создаётся один раз на всю программу (статья 2.5). ИИ воспроизводит эту ошибку регулярно, потому что в обучающих данных её много.

5. Проглоченные исключения
`except Exception: pass`, `.get(key, {})` вместо проверки, `return None` там, где нужна ошибка. Разобрано в статьях 2.8 и 2.14 — в Python встречается чаще всего из-за удобства `.get()`.

6. Выдуманные параметры функций
Самый тихий вид галлюцинаций (статья 4.8): несуществующий именованный аргумент иногда просто игнорируется. Проверяется одной командой:

```
python3 -c "import inspect, aiohttp;
print(inspect.signature(aiohttp.ClientSession.get))"
```

7. Код вне функций
Выполняется при импорте — подключение к базе, чтение файла, запрос к API на верхнем уровне модуля (статья 2.2). В Python это особенно коварно, потому что модуль импортируется один раз и ошибка проявляется не сразу.

8. `print` вместо `logging`

В скрипте допустимо, в боте или сервисе — нет: вывод исчезает вместе с терминалом (статья 4.2).

КАК СКАЗАТЬ ИИ

Проверка одним запросом:

ЗАПРОС

```
Проверь свой код по списку, специфичному для Python:
- синтаксис под Python 3.12, не старее
- нет синхронных вызовов внутри async (requests, time.sleep, sqlite3)
- gather, если есть, с return_exceptions=True
- нет изменяемых значений по умолчанию в аргументах
- нет except Exception: pass и .get() вместо проверки
- нет кода вне функций на верхнем уровне
- логирование через logging, не print

Что из этого есть? Если ничего — так и скажи.
```

ЧЕК-ЛИСТ ПИТОНОВСКОГО КОДА

Шесть шагов, минуты три. Порядок — по убыванию отдачи.

1. Запускается ли

```
python3 -m weatherbot
```

Ловит выдуманные импорты, опечатки, синтаксис не той версии.

2. Проходит ли проверка типов

```
python3 -m mypy --strict weatherbot/
```

Ловит забытые проверки на `None`, перепутанные типы, несуществующие методы (статья 6.3). Самая выгодная из шести.

3. Проходят ли тесты

```
pytest -q
```

4. Дифф

```
git diff --stat && git diff
```

Размер, удалённые строки, границы (статья 4.7).

5. Асинхронность

Глазами: нет ли синхронных вызовов в `async def`, есть ли таймауты, есть ли `return_exceptions` у `gather`.

6. Неудачный случай

Запустить на пустых данных или недоступной сети.

СОВЕТ

Первые три шага стоит объединить в одну команду и повесить на алиас: `python3 -m mypy --strict . && pytest -q && python3 -m weatherbot`. Тогда полная проверка — это одна строка в терминале.

ШПАРГАЛКА

- Специфично для Python: старый синтаксис, синхронное в async, `gather` без защиты.
- `=[]` в аргументах и `.get()` вместо проверки — здесь встречаются чаще всего.
- Выдуманные именованные аргументы проверяй `inspect.signature`.
- Чек-лист: запуск · `mypy --strict` · тесты · дифф · асинхронность · неудачный случай.
- Проверка типов — самый выгодный шаг из шести.
- Объедини первые три в одну команду.

7 Веб-бэкенд

Серверная часть: как запрос превращается в ответ, что такое API и как читать работу с базой данных. Самая «взрослая» глава в книге — здесь впервые появляются чужие пользователи и необратимые операции.

7.1 Как устроен бэкенд

запрос, обработчик, ответ

ЧТО ЭТО

Бэкенд — программа, которая ждёт запросов и отвечает на них. Внутри она устроена всегда одинаково, на любом языке и фреймворке: **запрос приходит, маршрутизатор решает, кто его обработает, обработчик делает работу, ответ уходит обратно.**

Поняв эту цепочку один раз, ты ориентируешься в любом бэкенд-проекте — меняются слова и синтаксис, схема остаётся.

КАК ВЫГЛЯДИТ В КОДЕ



Рис. 7.1 · Путь запроса. Обработчик — единственное место, где находится твой код; всё остальное делает фреймворк.

Вот эта схема в коде. Рабочий пример на FastAPI:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field

app = FastAPI()

SUBS: dict[int, list[str]] = {7: ["Oslo", "Bergen"]}


class SubOut(BaseModel):
    user_id: int
    cities: list[str]


@app.get("/subs/{user_id}", response_model=SubOut)
def get_subs(user_id: int) -> SubOut:
    if user_id not in SUBS:
        raise HTTPException(status_code=404, detail="пользователь не
найден")
    return SubOut(user_id=user_id, cities=SUBS[user_id])
```

Запрос и ответ, по-настоящему:

```
curl -i http://127.0.0.1:8765/subs/7
```

```
HTTP/1.1 200 OK
server: uvicorn
content-length: 40
content-type: application/json

{"user_id":7,"cities":["Oslo","Bergen"]}
```

```
curl -o /dev/null -w "%{http_code}\n" http://127.0.0.1:8765/subs/999
```

```
404
```

Как это читать

`@app.get("/subs/{user_id}")` — маршрут. Декоратор регистрирует функцию: «вызывай её на GET-запрос по такому адресу» (статья 2.9). Искать вызов `get_subs` в проекте бесполезно — её вызывает фреймворк.

`user_id: int` — часть адреса, превращённая в аргумент. Фреймворк вынет её из `/subs/7` и преобразует в число. Если прислать `/subs/abc`, обработчик даже не запустится — придёт ошибка о неверном типе.

`response_model=SubOut` — форма ответа. Фреймворк проверит, что вернули именно её, и превратит в JSON.

`raise HTTPException(404)` — способ ответить ошибкой. Не `return`, а `raise`: исключение перехватывает фреймворк и превращает в HTTP-ответ (статья 2.8).

Что делает фреймворк, а что ты

Это главное для чтения чужого кода:

Фреймворк	Ты
принимает соединение	решаешь, что делать с данными
разбирает URL и находит обработчик	ходишь в базу
проверяет типы и схемы	решаешь, что считать ошибкой
превращает результат в JSON	
ловит исключения и отдаёт коды	

Твоего кода в цепочке — одно звено. Поэтому в бэкенд-проекте файлов много, а читать надо в первую очередь обработчики.

> **СОВЕТ**
> Открыв незнакомый бэкенд, начинай с поиска маршрутов: `@app.get`, `@router.post`, `app.use`, `@GetMapping`. Список адресов — это оглавление проекта: по нему видно, что система вообще умеет.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Покажи все маршруты этого проекта: метод, адрес и функция,
которая его обрабатывает. Таблицей, без кода.
```

ЗАПРОС

```
Что произойдёт при GET /subs/abc? Обработчик запустится
или ошибка будет раньше?
```

ЗАПРОС

```
Добавь обработчик GET /subs/{user_id}/count — возвращает
число подписок. 404, если пользователя нет. Остальные
маршруты не трогай.
```

НА ЧТО СМОТРЕТЬ

Логика внутри обработчика. Обработчик должен принять запрос, вызвать нужное и вернуть результат. Если в нём пятьдесят строк с SQL и бизнес-правилами — дальше этот код не переиспользовать и не протестировать.

Обращение к базе прямо в обработчике. То же самое: запросы выносят в отдельный модуль, чтобы их можно было менять, не трогая маршруты.

Ответ без схемы. Без `response_model` фреймворк отдаст что угодно, включая поля, которых пользователь видеть не должен.

`return` вместо `raise` при ошибке. Вернуть `{"error": "не найден"}` с кодом 200 — значит соврать клиенту: формально всё хорошо.

ШПАРГАЛКА

- Цепочка одинакова везде: клиент → маршрут → обработчик → база → ответ.
- Твой код — только обработчик, остальное делает фреймворк.
- Маршрут регистрируется декоратором; вызов функции в проекте не найдётся.
- Ошибку возвращают через `raise`, а не `return`.
- Незнакомый бэкенд читай со списка маршрутов — это его оглавление.
- Логика и запросы к базе выносятся из обработчика.

7.2 HTTP на пальцах

методы, коды, заголовки, тело

ЧТО ЭТО

HTTP — язык, на котором клиент и сервер разговаривают. Знать его целиком не нужно, но четыре вещи встречаются постоянно: метод, код ответа, заголовки и тело.

Это же язык, на котором отлаживают: «пришёл 404» или «отдаёт 500» — точное описание проблемы, с которым ИИ сразу понимает, где искать.

КАК ВЫГЛЯДИТ В КОДЕ

Настоящий обмен — запрос и ответ целиком:

```
curl -i http://127.0.0.1:8765/subs/7
```

```
HTTP/1.1 200 OK
server: uvicorn
content-length: 40
content-type: application/json

{"user_id":7,"cities":["Oslo","Bergen"]}
```

Первая строка — код. Дальше заголовки. Пустая строка отделяет их от тела.

Методы

Метод	Смысл	Меняет данные
`GET`	дай	нет
`POST`	создай	да
`PUT`	замени целиком	да
`PATCH`	поправь часть	да
`DELETE`	удали	да

Главное свойство `GET` — он не должен ничего менять. Это не формальность: браузеры, поисковики и кеши повторяют `GET`-запросы сами по себе. Если `GET` что-то создаёт, оно создастся несколько раз.

Коды ответов

Первая цифра говорит, кто виноват:

Диапазон	Что значит
`2xx`	всё хорошо
`3xx`	ищи в другом месте
`4xx`	**клиент ошибся**
`5xx`	**сервер сломался**

Шесть кодов, которые встретятся чаще всего:

Код	Когда	Из нашего примера
`200`	всё хорошо	`GET /subs/7`
`201`	создано	`POST /subs/7` с новым городом
`400`	запрос кривой	
`401` / `403`	не вошёл / не положено	
`404`	нет такого	`GET /subs/999`
`409`	конфликт	повторная подписка на тот же город
`422`	формат верный, данные не проходят проверку	город короче двух символов
`500`	ошибка на сервере	

Все эти коды в нашем примере настоящие:

```
curl -X POST .../subs/7 -d '{"city":"Tromsø"}'   → 201
curl -X POST .../subs/7 -d '{"city":"Tromsø"}'   → 409  (уже подписан)
curl -X POST .../subs/7 -d '{"city":"X"}'        → 422  (короче двух
символов)
```

> **ЗАПОМНИ**
> Разница между `4xx` и `5xx` определяет, кому чинить. `4xx` — клиент прислал не то, сервер отработал правильно. `5xx` — сервер не справился, и это твой баг. Если на кривые данные отдаётся `500`, значит, проверку забыли.

Заголовки

Служебные поля запроса и ответа. Знать надо три:

Заголовок	Зачем
`Content-Type`	в каком формате тело: `application/json`
`Authorization`	кто спрашивает: токен или ключ
`User-Agent`	кто клиент

`Authorization` — тот самый заголовок, в котором едут секреты. Про него отдельно в статье 7.8.

Тело

Данные запроса или ответа. Почти всегда JSON. У `GET` тела обычно нет — параметры передают в адресе:

```
/subs?city=Oslo&limit=10
```

Это строка запроса: пары «ключ = значение» после знака вопроса.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> Сервер отдаёт 500 на POST /subs/7 с телом {"city": ""}.
> Должен отдавать 422. Вот обработчик: <код>.
> Почему валидация не срабатывает?

> **ЗАПРОС**
> Какой код ответа правильный, если пользователь пытается
> подписаться на город, на который уже подписан? Объясни выбор.

> **ЗАПРОС**
> Напиши curl-команды для проверки всех четырёх случаев:
> успех, нет пользователя, повторная подписка, кривые данные.
> С выводом кода ответа.

Последний запрос даёт готовый набор проверок — ровно то, что нужно для неудачных случаев из статьи 4.1.

НА ЧТО СМОТРЕТЬ

`200` на ошибку. Ответ `{"error": "не найден"}` с кодом 200 ломает всех клиентов: они смотрят на код, а не читают текст.

`500` на кривые данные. Значит, проверки нет и сервер падает на необработанном исключении. Правильный ответ — `400` или `422`.

`GET`, который что-то меняет. Повторится сам, без участия пользователя.

Текст ошибки с лишним. Трейсбек в теле ответа показывает структуру проекта всем желающим. Подробности — в лог, клиенту короткое сообщение.

Код выбран наугад. `400` вместо `404`, `403` вместо `401` — клиент не сможет отличить «не вошёл» от «вошёл, но не положено».

ШПАРГАЛКА

- Четыре части: метод · код · заголовки · тело.
- `GET` не меняет данные — иначе повторится сам.
- `4xx` — виноват клиент, `5xx` — виноват ты.
- `404` нет такого, `409` конфликт, `422` не прошло проверку.
- `500` на кривые данные означает забытую валидацию.
- Трейсбек в ответе клиенту — утечка структуры проекта.

7.3 API, схемы и валидация

граница, на которой проверяют всё

ЧТО ЭТО

API — набор адресов, по которым с твоим сервером можно разговаривать. Схема — описание того, какие данные приходят и уходят. Валидация — проверка, что пришедшее соответствует схеме.

Для работы с ИИ это самое важное место бэкенда: **граница, где данные приходят снаружи, — единственное место, где их ещё можно проверить.** Пропустил здесь — дальше неверные данные расходятся по всему приложению.

КАК ВЫГЛЯДИТ В КОДЕ

```python
from pydantic import BaseModel, Field


class SubIn(BaseModel):
    city: str = Field(min_length=2, max_length=50)


class SubOut(BaseModel):
    user_id: int
    cities: list[str]


@app.post("/subs/{user_id}", response_model=SubOut, status_code=201)
def add_sub(user_id: int, body: SubIn) -> SubOut:
    cities = SUBS.setdefault(user_id, [])
    if body.city in cities:
        raise HTTPException(status_code=409, detail="уже подписан")
    cities.append(body.city)
    return SubOut(user_id=user_id, cities=cities)
```

Что происходит при неверных данных:

```
curl -X POST http://127.0.0.1:8765/subs/7 \
     -H "Content-Type: application/json" -d '{"city":"X"}'
```

```
[422]
{"detail":[{"type":"string_too_short","loc":["body","city"],
 "msg":"String should have at least 2 characters",
 "input":"X","ctx":{"min_length":2}}]}
```

Обработчик **не запустился вообще**. Проверка сработала до него, ответ описывает ровно то, что не так и где.

Почему это та же мысль, что и типы

Вернись к статье 6.3: `mypy` ловит ошибки до запуска. Схема делает то же самое на границе сервера — ловит неверные данные до обработчика.

Разница в том, где проходит граница:

Инструмент	Что проверяет	Когда
`mypy`	твой код согласован сам с собой	до запуска
Схема запроса	данные снаружи соответствуют ожиданиям	до обработчика
Схема ответа	ты отдал то, что обещал	до отправки

Три разных рубежа, одна идея: описать форму данных один раз и дать машине проверять.

Что писать в схеме

Не только типы, но и ограничения:

```
class SubIn(BaseModel):
    city: str = Field(min_length=2, max_length=50)
    notify_at: int = Field(ge=0, le=23)          # час от 0 до 23
    email: EmailStr | None = None                 # формат проверяется
```

Каждое ограничение в схеме — это проверка, которую не придётся писать руками и о которой не забудет ИИ.

> **ЗАПОМНИ**
> Схема ответа важна не меньше схемы запроса. Без `response_model` можно случайно отдать поле `password_hash` или внутренний `id`, просто вернув объект целиком.

Адреса

Короткие правила, которых хватает:

Правило	Пример
Существительные во множественном числе	`/subs`, `/users`
Действие выражает метод, а не адрес	`DELETE /subs/7`, не `/deleteSub`
Вложенность — через путь	`/users/7/subs`
Фильтры — через строку запроса	`/subs?city=Oslo&limit=10`

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Опиши схему запроса и ответа для POST /subs/{user_id}:
город — строка от 2 до 50 символов, час уведомления —
число от 0 до 23, необязательное. В ответе: user_id
и список городов. Код обработчика пока не пиши.
```

Схема как постановка задачи — то же, что сигнатура в статье 6.5: короче слов и проверяется машиной.

ЗАПРОС

```
Проверь, что в response_model не попадают внутренние поля:
хеши паролей, служебные id, пометки. Перечисли, что отдаётся
клиенту.
```

ЗАПРОС

```
Какие ограничения стоит добавить в эту схему, чтобы кривые
данные не доходили до обработчика? Только ограничения,
логику не трогай.
```

НА ЧТО СМОТРЕТЬ

Валидация внутри обработчика. `if len(city) < 2: raise ...` — работает, но проверка размазывается по коду, и часть случаев забывается. Место проверок — схема.

`dict` вместо схемы. `def add_sub(body: dict)` отключает проверку целиком: придёт что угодно.

Нет `response_model`. Отдаётся всё, что вернул обработчик.

Схема есть, ограничений нет. `city: str` без длины примет и пустую строку, и строку на мегабайт.

Разные схемы для одного и того же. Три слегка отличающихся описания подписки в трёх местах — рассинхронизируются при первой же правке.

ШПАРГАЛКА

- Граница сервера — единственное место, где внешние данные ещё можно проверить.
- Схема описывает форму, валидация проверяет, фреймворк делает это сам.
- При неверных данных обработчик не запускается вообще.
- `response_model` защищает от случайной отдачи лишних полей.
- Ограничения (`min_length`, `ge`, `le`) — проверки, которые не забудутся.
- Схема в запросе к ИИ работает как сигнатура: точнее слов.

7.4 Читаем бэкенд-код

роуты, модели, сервисы и кто кого вызывает

ЧТО ЭТО

Бэкенд-проект почти всегда разложен по слоям, и слои эти одинаковые везде: маршруты, схемы, бизнес-логика, работа с базой. Разные фреймворки называют их по-разному, но делят одинаково.

Зная деление, в незнакомом проекте находишь нужное за минуту вместо получасового блуждания.

КАК ВЫГЛЯДИТ В КОДЕ

```
app/
├── main.py              создание приложения, подключение роутеров
├── routers/             маршруты: что на каком адресе
│   ├── subs.py
│   └── users.py
├── schemas/             схемы запросов и ответов (статья 7.3)
│   └── subs.py
├── services/            бизнес-логика: правила предметной области
│   └── subs.py
├── repositories/        работа с базой: только запросы
│   └── subs.py
├── models/              описание таблиц
│   └── subs.py
└── core/
    ├── config.py        настройки из окружения
    └── deps.py          общие зависимости: подключение к базе, текущий
юзер
```

Кто кого вызывает

Правило простое: **вызовы идут сверху вниз и никогда обратно.**

```
routers  →  services  →  repositories  →  база
```

Слой	Что в нём есть	Чего в нём быть не должно
`routers`	разбор запроса, вызов сервиса, код ответа	SQL, бизнес-правила
`services`	правила: можно ли, сколько раз, что потом	SQL, HTTP-коды
`repositories`	запросы к базе	правила, HTTP
`models`	описание таблиц	логика

Нарушение этого порядка — самый частый признак запущенного проекта. Если `repositories` знает про HTTP-коды, значит, слои перепутаны, и менять такой код придётся везде сразу.

Как находить нужное

Что умеет система — список маршрутов (статья 7.1).

Где правило — в `services`. Вопрос «почему нельзя подписаться дважды» решается там, а не в обработчике.

Где запрос к базе — в `repositories`. Если ищешь, откуда берутся данные, иди туда.

Что приходит и уходит — в `schemas`.

Другие названия того же самого

Здесь	В других проектах
`routers`	`controllers`, `views`, `handlers`, `api`
`services`	`use_cases`, `domain`, `logic`, `core`
`repositories`	`dao`, `storage`, `db`, `queries`
`schemas`	`dto`, `serializers`, `types`

Слова разные, деление то же. Увидев незнакомое имя папки, подставь его в эту таблицу.

> **СОВЕТ**
> Если слоёв нет и всё лежит в одном файле — это не обязательно плохо. Для бота на триста строк деление на четыре слоя только мешает. Слои нужны, когда проект перестаёт помещаться в голове, а не сразу.

Зависимости

Строка, которая встречается в каждом обработчике:

```
@app.get("/subs/{user_id}")
def get_subs(user_id: int, conn = Depends(get_conn)):
```

`Depends(get_conn)` означает: «фреймворк, дай мне подключение к базе». Откуда оно возьмётся и когда закроется — забота фреймворка.

Такие зависимости — обычное место для подключения к базе, проверки токена и текущего пользователя. Если не знаешь, откуда в обработчике взялся аргумент, которого нет в адресе, — ищи `Depends` или аналог.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> Разложи этот проект по слоям: что в роутерах, что в сервисах,
> что в работе с базой. Если слои перепутаны — скажи, где именно.

> **ЗАПРОС**
> Я хочу добавить правило «не больше 10 подписок на пользователя».
> В каком слое ему место и какие файлы придётся тронуть?
> Код пока не пиши.

Второй запрос — проверка на понимание архитектуры: если ИИ предлагает вписать правило в обработчик, слои в проекте уже поехали.

НА ЧТО СМОТРЕТЬ

SQL в обработчике. Самый частый признак. Запрос переиспользовать нельзя, протестировать без поднятия сервера тоже.

Бизнес-правила в репозитории. `if len(cities) > 10` среди SQL — правило спрятано там, где его никто не станет искать.

HTTP-коды в сервисе. `raise HTTPException(404)` вне роутера означает, что сервис теперь нельзя вызвать ниоткуда, кроме HTTP.

Импорт снизу вверх. `repositories` импортирует `routers` — круговая зависимость и путаница.

Папка `utils`, в которой всё. Признак, что место новому коду искать перестали (статья 2.3).

ШПАРГАЛКА

- Четыре слоя: роуты · сервисы · репозитории · модели.
- Вызовы идут только сверху вниз.
- Правило — в сервисах, SQL — в репозиториях, коды ответов — в роутерах.
- Имена папок разные, деление одинаковое.
- Аргумент, которого нет в адресе, приходит через `Depends`.
- Для маленького проекта слои не нужны — они появляются с размером.

7.5 База данных

таблицы, связи и SQL, которого хватает для чтения

ЧТО ЭТО

База — место, где данные переживают перезапуск. Писать SQL самому не придётся, а вот читать — обязательно: запрос решает, какие данные получит приложение, и ошибка здесь не видна ни при запуске, ни в трейсбеке.

Для чтения достаточно понимать таблицы, связи и четыре вида запросов.

КАК ВЫГЛЯДИТ В КОДЕ

```
CREATE TABLE users (
    id    INTEGER PRIMARY KEY,
    name  TEXT NOT NULL
);

CREATE TABLE subs (
    id       INTEGER PRIMARY KEY,
    user_id  INTEGER NOT NULL REFERENCES users(id),
    city     TEXT NOT NULL,
    UNIQUE (user_id, city)
);
```

Построчно:

- `PRIMARY KEY` — главный ключ, уникальный номер записи.
- `REFERENCES users(id)` — связь: в `user_id` может лежать только существующий `id` из `users`. База не даст записать подписку несуществующему пользователю.
- `NOT NULL` — поле обязательно.
- `UNIQUE (user_id, city)` — пара уникальна. Это и есть правило «нельзя подписаться дважды», но выраженное так, что его невозможно обойти из кода.

> **ЗАПОМНИ**
> Ограничение в схеме базы надёжнее проверки в коде. Код можно обойти, забыть вызвать, продублировать с ошибкой; `UNIQUE` и `NOT NULL` работают всегда, для всех путей, включая ручные правки.

Четыре вида запросов

```
SELECT city FROM subs WHERE user_id = 7;          -- прочитать
INSERT INTO subs (user_id, city) VALUES (7, 'Oslo');  -- добавить
UPDATE subs SET city = 'Bergen' WHERE id = 3;     -- изменить
DELETE FROM subs WHERE id = 3;                     -- удалить
```

Главное при чтении — `WHERE`. Он решает, сколько строк затронет запрос. `UPDATE` без `WHERE` меняет всю таблицу; `DELETE` без `WHERE` очищает её. Это самая дорогая опечатка в бэкенде.

`JOIN` — данные из двух таблиц

```
SELECT u.name, s.city
FROM users u
LEFT JOIN subs s ON s.user_id = u.id;
```

Читается так: *взять пользователей, к каждому подставить его подписки, связывая по* `user_id = id`.

Вид	Что делает
`INNER JOIN`	только те, у кого есть совпадение
`LEFT JOIN`	все слева, справа `NULL`, если нет пары

Разница важна: `INNER JOIN` молча потеряет пользователей без подписок. Если в отчёте не хватает строк — первым делом смотри на вид `JOIN`.

Параметры, а не склейка строк

```
# так нельзя — дыра в безопасности
conn.execute(f"SELECT city FROM subs WHERE user_id = {user_id}")

# так правильно
conn.execute("SELECT city FROM subs WHERE user_id = ?", (user_id,))
```

Вопросительный знак — место для значения. База подставит его сама, и что бы ни пришло от пользователя, оно останется значением, а не превратится в часть запроса.

Это защита от SQL-инъекции: без неё строка `7; DROP TABLE subs` в адресе удалит таблицу. ИИ обычно пишет параметры правильно, но при сборке запроса из кусков («если фильтр задан, добавить условие») легко скатывается к `f`-строкам.

Транзакции

```
with conn:
    conn.execute("INSERT INTO subs ...")
    conn.execute("UPDATE users SET sub_count = sub_count + 1 ...")
```

Либо выполнятся обе строки, либо ни одной. Нужно везде, где несколько изменений должны произойти вместе: иначе при ошибке на втором запросе данные останутся наполовину изменёнными.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Объясни этот запрос по шагам: из каких таблиц берутся данные,
по какому условию связываются, сколько строк вернётся.
```

ЗАПРОС
```
Схема такая: <CREATE TABLE>. Напиши запрос: все города,
на которые подписано больше трёх пользователей, с числом
подписчиков. Параметры через плейсхолдеры.
```

ЗАПРОС
```
В этом UPDATE нет WHERE. Так задумано или это ошибка?
```

Последний вопрос стоит задавать всегда, когда видишь `UPDATE` или `DELETE` в диффе.

НА ЧТО СМОТРЕТЬ

`UPDATE` или `DELETE` без `WHERE`. Проверяй в каждом диффе отдельно.

`f`-строка в запросе. Инъекция. Исключений нет.

`INNER JOIN` там, где нужен `LEFT`. Молча теряет строки.

Нет `NOT NULL` и `UNIQUE`. Правила перенесены в код и будут обойдены.

`SELECT *`. Вернёт все поля, включая те, что добавят завтра, — и они уедут клиенту (статья 7.3).

Несколько изменений без транзакции. При ошибке посередине данные останутся несогласованными.

ШПАРГАЛКА

- `PRIMARY KEY` — номер, `REFERENCES` — связь, `UNIQUE` и `NOT NULL` — правила.
- Ограничение в схеме надёжнее проверки в коде.
- `WHERE` решает, сколько строк затронет запрос; без него — все.
- `LEFT JOIN` сохраняет строки без пары, `INNER` — теряет.
- Значения только через `?`, никогда через `f`-строку.
- Несколько изменений — в транзакции.

7.6 Запросы, которые тормозят

N+1 и индексы на измеренных числах

ЧТО ЭТО

Две причины, по которым бэкенд начинает тормозить, и обе ИИ воспроизводит охотно: лишние запросы в цикле и отсутствие индексов.

Коварство в том, что **на маленьких данных обе незаметны.** В разработке с десятью пользователями всё летает; на двухстах начинает подтормаживать; на тысячах встаёт. Код при этом не менялся.

КАК ВЫГЛЯДИТ В КОДЕ

N+1 запрос

Классика. Берём пользователей, потом для каждого — его подписки:

```
users = conn.execute("SELECT id, name FROM users").fetchall()
for u in users:
    conn.execute("SELECT city FROM subs WHERE user_id = ?", (u["id"],))
```

Выглядит невинно. На деле это **один запрос плюс по одному на каждого пользователя** — отсюда и название.

То же самое одним запросом:

```
conn.execute("""
    SELECT u.name, s.city
    FROM users u
    LEFT JOIN subs s ON s.user_id = u.id
""")
```

Замер на 200 пользователях и 2000 подписках:

```
N+1:    201 запросов, 20.7 мс
JOIN:     1 запрос,    2.7 мс, строк 2000
```

В восемь раз быстрее, и это на локальной базе в памяти. Если база на другом сервере, каждый запрос — ещё и поход по сети: двести походов вместо одного. Там разница будет не в восемь раз, а в сотни.

Индексы

Индекс — служебная структура, которая позволяет базе находить строки, не перебирая все.

```
CREATE INDEX idx_subs_user ON subs(user_id);
```

Тот же набор запросов до и после:

```
без индекса: 15.4 мс
с индексом:   1.4 мс     — быстрее в 10,6 раза
```

Одна строка в схеме, десятикратное ускорение. При этом индекс — не бесплатный: он занимает место и чуть замедляет запись. Поэтому их ставят не на всё подряд, а туда, где ищут.

Правило: индекс нужен на поля, по которым идёт поиск или связывание — то, что стоит в `WHERE` и в `ON`. В нашем случае `subs.user_id` участвует в обоих.

> **СОВЕТ**
> Не оптимизируй заранее. Сначала работающий код, потом измерение, потом индекс — и только там, где измерение показало проблему. Но N+1 — исключение: это не оптимизация, а исправление лишней работы, и делать его стоит сразу.

Как увидеть проблему

В SQL есть встроенный способ спросить у базы, как она собирается выполнять запрос:

```
EXPLAIN QUERY PLAN SELECT city FROM subs WHERE user_id = 7;
```

Если в ответе `SCAN subs` — база перебирает всю таблицу. Если `SEARCH subs USING INDEX` — пользуется индексом. В PostgreSQL то же самое называется `EXPLAIN ANALYZE` и печатает ещё и реальное время.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Здесь запрос к базе внутри цикла. Перепиши одним запросом
с JOIN. Результат должен быть тем же: список пользователей,
у каждого — его города, включая пользователей без подписок.
```

Последняя фраза важна: без неё `INNER JOIN` потеряет тех, у кого подписок нет (статья 7.5).

ЗАПРОС

```
Вот схема таблиц и запросы, которые выполняются чаще всего.
Какие индексы стоит добавить и почему? Для каждого — какой
запрос он ускорит.
```

ЗАПРОС

```
Этот обработчик делает сколько запросов к базе при одном
HTTP-запросе? Посчитай.
```

Последний — хороший способ проверки: ответ «один» или «два» нормален, «зависит от числа пользователей» означает N+1.

НА ЧТО СМОТРЕТЬ

Запрос внутри цикла. Главный признак. Ищи `for` и `execute` рядом.

Запрос внутри функции, которая вызывается в цикле. То же самое, но на уровень глубже и потому незаметнее. Признак: функция вида `get_cities(user)` вызывается в обходе списка.

ORM, которая делает запросы лениво. В коде обращение к полю, в базе — запрос. `for u in users: print(u.subs)` выглядит как работа с памятью, а является N+1.

Индексов нет вообще. Кроме `PRIMARY KEY`, который создаётся сам. Если в схеме ни одного `CREATE INDEX`, а поиск по полям идёт — они нужны.

Индекс на всё подряд. Обратная крайность: каждый индекс замедляет запись и занимает место.

«Работает же». На твоих тестовых двадцати записях — работает. Проверяй на объёме, похожем на реальный.

ШПАРГАЛКА

- N+1: запрос в цикле вместо одного `JOIN`. 201 запрос против одного.
- Проверка: «сколько запросов к базе делает этот обработчик?»
- Индекс — на поля из `WHERE` и `ON`. Десятикратное ускорение одной строкой.
- Индексы замедляют запись, поэтому не на всё подряд.
- `EXPLAIN QUERY PLAN`: `SCAN` — перебор, `SEARCH USING INDEX` — хорошо.
- На маленьких данных обе проблемы невидимы.

7.7 Миграции

самое необратимое, что есть в проекте

ЧТО ЭТО

Миграция — изменение структуры базы: добавить таблицу, колонку, индекс, переименовать поле. Это тот самый правый нижний угол из статьи 1.5: **необратимо и может сломаться незаметно.**

Разница с обычным кодом принципиальная. Неверный код откатывается коммитом (статья 5.1); неверная миграция уже изменила данные, и `git revert` их не вернёт.

КАК ВЫГЛЯДИТ В КОДЕ

Миграции — пронумерованные файлы, которые применяются по порядку и только один раз:

```
migrations/
├── 001_initial.sql
├── 002_add_notify_hour.sql
└── 003_add_index_subs_user.sql
```

```sql
-- 002_add_notify_hour.sql
ALTER TABLE subs ADD COLUMN notify_hour INTEGER NOT NULL DEFAULT 9;
```

База хранит, какие миграции уже применены, и повторно их не запускает. Поэтому правило: **применённую миграцию не правят.** Ошибся — пиши следующую, которая исправляет.

Что безопасно, а что нет

Безопасно	Опасно
добавить таблицу	удалить таблицу
добавить колонку с `DEFAULT`	удалить колонку
добавить индекс	переименовать колонку
расширить тип (`INT` → `BIGINT`)	сузить тип
	добавить `NOT NULL` без `DEFAULT`

Левая колонка не ломает работающий код: старые запросы продолжают работать. Правая ломает — и ломает в момент применения, когда откатывать уже нечего.

Отдельно про `NOT NULL` без умолчания: если в таблице уже есть строки, миграция просто не выполнится. Это ещё хороший случай — хуже, когда она выполняется и портит данные.

> **ВНИМАНИЕ**
> Переименование колонки — не одна операция, а три: добавить новую, скопировать данные, удалить старую. Между ними код должен понимать оба варианта. Сделанное одним шагом, оно уронит всё, что обращается к старому имени.

Обязательный порядок

1. **Резервная копия.** До всего остального. Для SQLite — копия файла, для PostgreSQL — `pg_dump`.
2. **Прогон на копии.** Не на боевой базе. Миграция, которая отработала на пустой тестовой базе, ничего не доказывает — нужна копия с реальными данными.
3. **Проверка отката.** Как вернуть, если не получилось. Если ответа нет, миграцию нельзя применять (статья 1.5).
4. **Применение.**
5. **Проверка данных.** Количество строк до и после, выборочные записи.

Пункт 2 — тот, который пропускают. А именно он ловит случаи вроде «в старых записях город пустой, и `NOT NULL` не применился».

Заполнение данных — отдельно

Миграция структуры и миграция данных — разные вещи:

```
-- 002: структура
ALTER TABLE subs ADD COLUMN notify_hour INTEGER NOT NULL DEFAULT 9;
```

```
# отдельный скрипт: заполнить осмысленными значениями
UPDATE subs SET notify_hour = 8 WHERE city IN (SELECT ...);
```

Разделять стоит потому, что скрипт заполнения можно запустить повторно, проверить на части данных и остановить. Миграцию — нет.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Нужна миграция: добавить в subs колонку notify_hour,
целое от 0 до 23, по умолчанию 9. Существующие строки
не должны сломаться.

Отдельно скажи: что будет с кодом, который сейчас читает
эту таблицу, и нужно ли его менять до или после миграции.
```

Второй абзац ловит главную проблему миграций — порядок выкатки кода и схемы.

ЗАПРОС

```
Эта миграция переименовывает колонку одним ALTER.
Распиши её как три шага, чтобы код продолжал работать
между ними.
```

ЗАПРОС

```
Напиши обратную миграцию для этой. Если откатить нельзя
без потери данных — так и скажи, не придумывай.
```

НА ЧТО СМОТРЕТЬ

Нет обратной миграции. Не всегда возможна, но её отсутствие должно быть осознанным, а не случайным.

`DROP` в миграции. Удаление колонки или таблицы — всегда отдельное решение, принимаемое человеком, и обычно не раньше, чем через несколько релизов после того, как её перестали использовать.

Миграция и код выкатываются одновременно. Между применением миграции и обновлением кода проходит время, и в этот момент должно работать и то и другое.

Миграция правится после применения. На твоей машине она применится заново, на сервере — нет. Базы разъедутся.

Миграция содержит `UPDATE` на всю таблицу. Заполнение данных — отдельным скриптом.

ШПАРГАЛКА

- Миграция необратима: `git revert` не вернёт данные.
- Применённую миграцию не правят — пишут следующую.
- Безопасно: добавить. Опасно: удалить, переименовать, сузить.
- Переименование — три шага, а не один.
- Порядок: копия → прогон на копии с данными → план отката → применение → проверка.
- Структура и заполнение данных — разные файлы.

7.8 Доступ, ошибки ИИ и чек-лист

чем бэкенд отличается от остального кода

ЧТО ЭТО

Завершение главы. Две темы: кто имеет право на что — и как проверить бэкенд-код.

Бэкенд отличается от всего остального в книге одним: **здесь появляются чужие люди.** Ошибка в скрипте стоит твоего вечера; ошибка в доступе на сервере может стоить чужих данных.

ДОСТУП

Два разных вопроса, которые постоянно путают:

Вопрос	Термин	Код ответа
Кто ты?	аутентификация	`401`, если не представился
Что тебе можно?	авторизация	`403`, если представился, но не положено

Типичная дыра выглядит так: токен проверяется, а право — нет.

```
@app.get("/subs/{user_id}")
def get_subs(user_id: int, current_user = Depends(get_current_user)):
    return SUBS[user_id]          # кто угодно читает чужие подписки
```

Пользователь вошёл честно, токен настоящий — и запрашивает `/subs/999`. Проверка личности прошла, проверки права нет.

```
    if current_user.id != user_id and not current_user.is_admin:
        raise HTTPException(status_code=403, detail="не ваши данные")
```

ЗАПОМНИ

ИИ почти всегда добавляет аутентификацию, когда просят «сделать с авторизацией», и почти никогда не добавляет проверку прав на конкретную запись. Спрашивай об этом отдельно: «а может ли пользователь 7 прочитать данные пользователя 8?»

Что нельзя отдавать

- **Токены и ключи** — ни в код, ни в чат, ни в логи (статья 5.7).
- **Дамп боевой базы** — в нём чужие персональные данные. Для отладки делают обезличенную выборку.
- **Трейсбек клиенту** — он показывает структуру проекта (статья 7.2).
- **Внутренние поля в ответе** — хеши, служебные идентификаторы (статья 7.3).

ОШИБКИ ИИ В БЭКЕНДЕ

1. **Аутентификация без авторизации** — разобрана выше.
2. `500` **вместо** `400` — нет валидации, сервер падает на кривых данных.
3. **N+1 запрос** — статья 7.6.
4. `f`**-строка в SQL** — инъекция, статья 7.5.
5. **Нет транзакции** там, где несколько изменений должны произойти вместе.
6. **Нет** `response_model` — в ответ уезжает всё.
7. **Синхронный драйвер в асинхронном коде** — `sqlite3` вместо `aiosqlite`, блокирует весь сервер (статья 6.4).
8. **Нет пагинации.** `GET /subs` возвращает все записи. На тысяче всё ещё работает, на миллионе кладёт и сервер, и клиента.

ЧЕК-ЛИСТ БЭКЕНД-КОДА

К общему чек-листу приёмки (статья 4.7) добавляются пять шагов.

1. Коды ответов

Проверить все четыре случая настоящими запросами:

```
curl -o /dev/null -w "%{http_code}\n" .../subs/7      # 200
curl -o /dev/null -w "%{http_code}\n" .../subs/999    # 404
curl -X POST ... -d '{"city":"X"}'  -w " %{http_code}\n"   # 422
curl -X POST ... -d '{"city":"Oslo"}' -w " %{http_code}\n" # 409
повторно
```

2. Чужие данные

Войти одним пользователем, запросить данные другого. Должен быть `403`.

3. Запросы к базе

Сколько их на один HTTP-запрос? Больше двух-трёх — смотри на N+1.

4. Необратимое

Есть ли в диффе `UPDATE`, `DELETE`, `DROP`, миграции? Каждое — отдельно и внимательно (статьи 7.5, 7.7).

5. Что уезжает клиенту

Посмотреть реальный ответ целиком, а не схему. Нет ли лишних полей.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Проверь этот бэкенд-код по списку:
- есть ли проверка прав на конкретную запись, а не только токена
- какие коды отдаются на кривые данные и на отсутствие записи
- сколько запросов к базе на один HTTP-запрос
- есть ли f-строки в SQL
- что уезжает клиенту в ответе

Что из этого проблема? Если ничего — так и скажи.
```

ЗАПРОС

```
Может ли пользователь 7 получить данные пользователя 8?
Покажи, каким запросом, или объясни, что мешает.
```

ШПАРГАЛКА

- Аутентификация — кто ты (`401`), авторизация — что можно (`403`).
- ИИ добавляет первое и забывает второе. Спрашивай про права отдельно.
- Восемь частых ошибок: права, `500`, N+1, инъекция, транзакции, `response_model`, синхронный драйвер, пагинация.
- Чек-лист: коды · чужие данные · число запросов · необратимое · что уезжает клиенту.
- Коды проверяй настоящими `curl`, а не чтением кода.
- Дамп боевой базы в чат не отдают.

8 Фронтенд

HTML, CSS и браузер. Первая глава, где результат видно глазами, — и первая, где проверка означает «посмотреть», а не «запустить». Иллюстрации здесь настоящие: это снимки того, что нарисовал браузер.

8.1 Как браузер собирает страницу

от текста к пикселям

ЧТО ЭТО

Браузер получает текст и превращает его в картинку. Между этими двумя состояниями — четыре шага, и знать их нужно по практической причине: **ошибки на каждом шаге выглядят по-разному**, и по виду поломки сразу понятно, где искать.

КАК ВЫГЛЯДИТ В КОДЕ

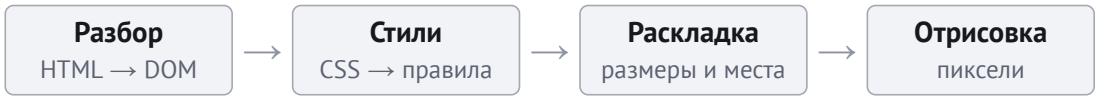


Рис. 8.1 · Путь страницы. Страница рисуется не один раз: изменение стилей или данных запускает шаги заново.

Четыре шага

1. Разбор. Текст HTML превращается в дерево элементов — DOM. Именно это дерево видно во вкладке «Элементы» и именно его меняет JavaScript. Важно: браузер чинит сломанную разметку сам, по своим правилам, — поэтому незакрытый тег не даёт ошибки, а даёт неожиданное дерево.

2. Стили. Для каждого элемента собираются все подходящие правила CSS и вычисляется итоговое значение каждого свойства (статья 8.3).

3. Раскладка. Браузер считает, где и какого размера будет каждый элемент. Здесь появляются все эффекты вида «почему блок на всю ширину» и «почему элементы налезли».

4. Отрисовка. Готовые прямоугольники превращаются в пиксели.

Где какая ошибка

Что видно	Где искать
Текст есть, но структура не та	разбор: сломанный HTML
Элемент на месте, но не выглядит как надо	стили: правило не применилось
Выглядит правильно, но стоит не там	раскладка: `display`, ширины, отступы
Мелькание, рывки при прокрутке	отрисовка: слишком часто перерисовывается

Первая строка — самая коварная. Браузер не ругается на сломанный HTML, он молча достраивает дерево по-своему. Поэтому «не работает селектор» часто означает «элемент оказался не там, где ты думаешь».

> **ЗАПОМНИ**
> Браузер никогда не падает. Что бы ни было в HTML и CSS, он что-нибудь покажет. Это удобно для пользователя и неудобно для отладки: ошибки не сообщают о себе, их надо увидеть.

Страница рисуется не один раз

Шаги 2–4 повторяются каждый раз, когда что-то меняется: JavaScript изменил текст, пользователь открыл меню, пришли новые данные. Отсюда два слова, которые встречаются в разговорах про скорость:

- **reflow** (перерасчёт раскладки) — дорого, затрагивает соседей;
- **repaint** (перерисовка) — дешевле, меняются только пиксели.

Для чтения кода достаточно знать: изменение размеров и позиций дороже изменения цвета.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> ```
> Элемент с классом .card не видно на странице. Он есть в DOM
> или его там нет? Как это проверить, не угадывая?
> ```

Вопрос поставлен правильно: он разделяет «не создался» и «создался, но невидим» — две совершенно разные причины.

> **ЗАПРОС**
> ```
> Вот HTML и CSS. На каком шаге — разбор, стили или раскладка —
> происходит то, что я вижу: блоки идут друг под другом,
> хотя я ожидал в ряд?
> ```

ЗАПРОС

```
Проверь этот HTML на незакрытые и неправильно вложенные теги.
Только список проблем, без исправлений.
```

НА ЧТО СМОТРЕТЬ

Элемент есть в коде, но не в DOM. Значит, разметка сломана и браузер перестроил дерево. Частая причина — `<div>` внутри `<p>` или незакрытый тег.

Стиль не применился. Три причины по убыванию частоты: селектор не совпадает, правило перебито другим (статья 8.3), опечатка в свойстве. Четвёртая, редкая: файл стилей вообще не загрузился — это видно во вкладке «Сеть» (статья 8.4).

Всё поехало от одного изменения. Признак того, что раскладка держалась на случайности — например, на точной ширине в пикселях.

Страница дёргается. Что-то меняет размеры на каждом кадре. Искать JavaScript, который пишет в стили в цикле или в обработчике прокрутки.

ШПАРГАЛКА

- Четыре шага: разбор → стили → раскладка → отрисовка.
- Браузер не падает: сломанный HTML молча превращается в другое дерево.
- Структура не та — разбор. Вид не тот — стили. Место не то — раскладка.
- Шаги 2–4 повторяются при каждом изменении.
- Менять размеры дороже, чем менять цвет.
- Первый вопрос при пропаже элемента: он вообще есть в DOM?

8.2 Читаем HTML

структура, семантика, атрибуты

ЧТО ЭТО

HTML — не язык программирования, а разметка: он описывает, из чего состоит страница. Читать его просто, и для работы с ИИ этого достаточно: ты должен понимать, что за структуру он предложил, и уметь сказать, что нужно иначе.

КАК ВЫГЛЯДИТ В КОДЕ

```
<article class="card" data-city="oslo">
  <h2 class="card__title">Осло</h2>
  <p class="card__temp">Сейчас <strong>+4</strong></p>
  <button type="button" aria-label="Удалить подписку">×</button>
</article>
```

Как это читать

Тег открывается и закрывается: `<p>` … `</p>`. Некоторые закрывать не надо: `<img>`, `<input>`, `<br>`.

Вложенность = структура. `<strong>` внутри `<p>` внутри `<article>` — дерево, то самое DOM из статьи 8.1. Отступы в коде ни на что не влияют, важна только вложенность.

Атрибуты — пары `имя="значение"` в открывающем теге:

Атрибут	Зачем
`class`	зацепка для CSS и JS; одна может быть у многих элементов
`id`	уникальное имя; должно быть одно на странице
`data-*`	свои данные: `data-city="oslo"` прочитается из JS
`type`, `href`, `src`	смысловые, зависят от тега
`aria-*`	подсказки для программ чтения с экрана

Семантика

Теги различаются не видом, а смыслом. `<div>` ничего не значит; `<article>`, `<nav>`, `<header>`, `<button>` — значат.

Разница практическая, а не эстетическая:

```
<div class="btn" onclick="remove()">×</div>   <!-- так не надо -->
<button type="button" onclick="remove()">×</button>
```

`<button>` сам доступен с клавиатуры, сам нажимается пробелом и Enter, сам сообщает программам чтения, что это кнопка. `<div>` не делает ничего из этого — всё придётся добавлять руками, и обычно забывают.

Правило чтения: увидел `<div>` с обработчиком нажатия — почти всегда там должен быть `<button>` или `<a>`.

> **СОВЕТ**
> Самая частая правка, которую стоит просить у ИИ в готовой разметке: «замени `div` с обработчиками на семантические теги — `button`, `a`, `nav`, `label`». Это одна фраза, а страница становится доступнее и короче.

Формы

```
<form>
  <label for="city">Город</label>
  <input id="city" name="city" type="text" required minlength="2">
  <button type="submit">Подписаться</button>
</form>
```

Три вещи, которые надо проверять в формах:

- **`label` связан с полем** через `for` = `id`. Без этого нажатие на подпись не ставит курсор в поле.
- **`name`** — под этим именем значение уйдёт на сервер. Нет `name` — поле не отправится вообще.
- **`required`, `minlength`, `type="email"`** — проверка на стороне браузера. Удобно пользователю, но **не отменяет проверки на сервере** (статья 7.3): запрос можно отправить и мимо формы.

Структура документа

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Погода</title>
    <link rel="stylesheet" href="/style.css">
  </head>
  <body>
    …
    <script src="/app.js"></script>
  </body>
</html>
```

`<head>` — служебное, на странице не видно. `<body>` — то, что видно. `lang="ru"` влияет на переносы и на произношение программами чтения, а `viewport` — на то, как страница выглядит на телефоне. Без него мобильный браузер покажет уменьшенную версию десктопной.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Перепиши эту разметку семантически: div с обработчиками —
в button, блоки навигации — в nav, заголовки по уровням.
Классы и стили не трогай.
```

ЗАПРОС

```
Проверь форму: у всех ли полей есть name и связанный label,
стоят ли базовые ограничения. Список проблем, без правок.
```

ЗАПРОС

```
Я хочу карточку города: название, температура, кнопка удаления.
Предложи разметку — семантическую, с классами по БЭМ.
Стили пока не нужны.
```

НА ЧТО СМОТРЕТЬ

`<div>` вместо кнопки или ссылки. Самая частая проблема в сгенерированной разметке.

Нет `alt` у картинок. `<img src="..." alt="">` — пустой `alt` допустим для декоративных, отсутствующий — нет.

Заголовки не по порядку. `<h1>`, потом сразу `<h4>` — структура документа ломается для тех, кто читает её программой.

Стили прямо в разметке. `style="margin-top: 10px"` в каждом элементе — менять потом придётся в двадцати местах.

`id` повторяется. Должен быть уникальным; дубли ломают и CSS, и JS, и связку `label` с полем.

Разметка на пять уровней вложенности из `<div>`. Признак, что ИИ строил структуру под внешний вид, а не под смысл.

ШПАРГАЛКА

- Вложенность — это структура, отступы ни на что не влияют.
- `class` для многих, `id` для одного, `data-*` для своих данных.
- `<div>` с обработчиком нажатия — почти всегда должен быть `<button>`.
- В форме проверяй: `name`, связанный `label`, ограничения.
- Проверка в браузере не отменяет проверки на сервере.
- `viewport` в `<head>` — иначе на телефоне страница уменьшится.

8.3 Читаем CSS

селекторы, каскад, раскладка

ЧТО ЭТО

CSS отвечает на два вопроса: **к чему применить** (селектор) и **что сделать** (свойства). Сложность не в свойствах — их просто много и их гуглят. Сложность в двух других местах: почему правило не применилось и почему блок стоит не там.

Обе темы разбираются ниже, и обе — на настоящих снимках того, что нарисовал браузер.

КАК ВЫГЛЯДИТ В КОДЕ

```
.card {
  display: flex;
  gap: 8px;
  padding: 12px;
  border: 1px solid #d8dce5;
}

.card__title {
  font-size: 18px;
  font-weight: 600;
}

.card:hover { background: #f7f8fa; }
```

Селекторы

Селектор	К чему применится
`.card`	элементы с классом `card`
`#main`	элемент с `id="main"`
`article`	все теги `<article>`
`.card .title`	`.title` **внутри** `.card`
`.card > .title`	`.title` — прямой потомок
`.card:hover`	`.card` под курсором
`[data-city="oslo"]`	по значению атрибута

Пробел между классами значит «внутри», и это самый частый источник недоумения: `.card .title` и `.card.title` — разные вещи. Второе — элемент, у которого **оба** класса сразу.

Каскад: кто побеждает

Когда к элементу подходят два правила, выигрывает более «весомое». Порядок по возрастанию веса:

Что	Вес
тег: `article`	низкий
класс: `.card`, `:hover`, `[data-*]`	средний
`id`: `#main`	высокий
`style="..."` прямо в теге	очень высокий
`!important`	перебивает всё

При равном весе выигрывает тот, что **ниже в файле**.

Отсюда практическое следствие: `!important` в сгенерированном CSS — почти всегда признак того, что ИИ не разобрался, какое правило мешает, и перебил его силой. Работать будет; следующая правка станет труднее.

Раскладка: `display` решает всё

Одна и та же разметка из трёх блоков, три значения `display`:

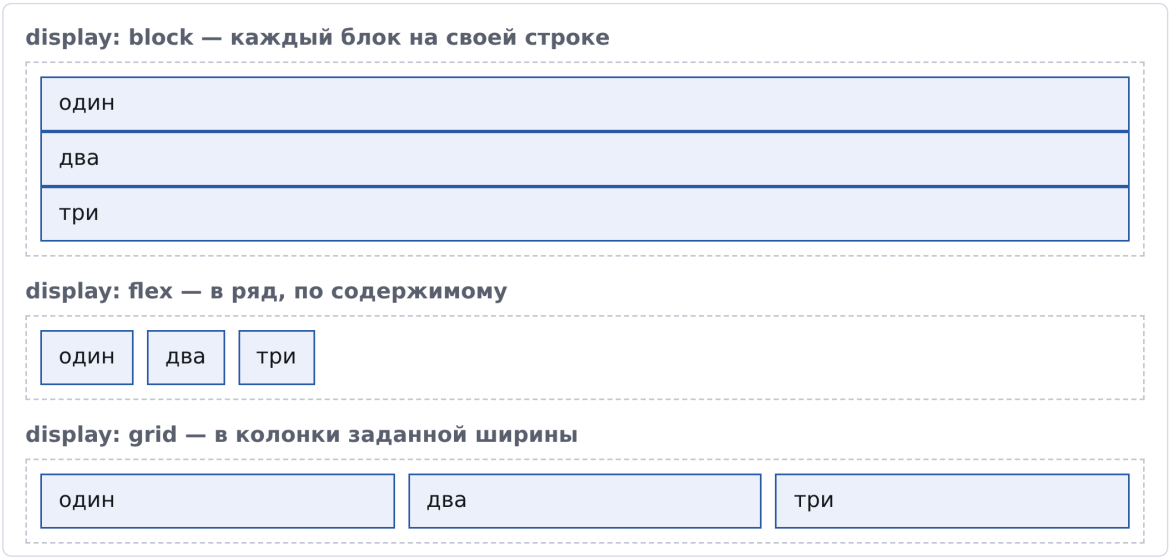


Рис. 8.2 · Одна и та же разметка при разных `display`. Снимок настоящий — так это нарисовал браузер.

- **`block`** — каждый элемент на своей строке, во всю доступную ширину. Так ведут себя `div`, `p`, `h1` по умолчанию.
- **`flex`** — дети выстраиваются в ряд по содержимому. Для рядов кнопок, карточек, панелей.
- **`grid`** — дети раскладываются по заданной сетке колонок.

Девяносто процентов современной раскладки — это `flex` и `grid`. Если в сгенерированном CSS встречаются `float` или

`position: absolute` для расстановки блоков, это устаревший подход (статья 3.1, причина 3).

Коробка: откуда берётся ширина

Самая частая неожиданность в CSS. Два блока, у обоих `width: 200px`, `padding: 20px`, `border: 4px`:

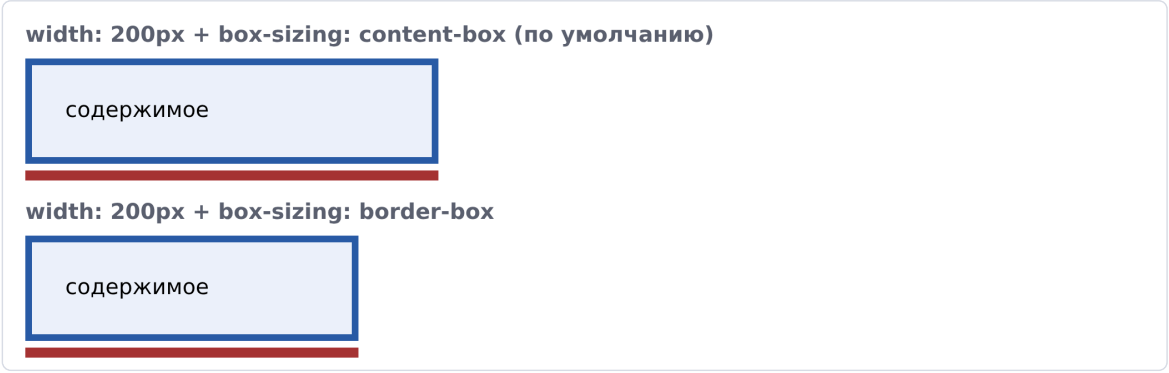


Рис. 8.3 · Красная линия — настоящая ширина. При `content-box` блок шире заданного на отступы и рамку.

Измерено в браузере:

```
content-box: 248px
border-box:  200px
```

При `box-sizing: content-box` (поведение по умолчанию) `width` задаёт ширину **содержимого**, а отступы и рамка добавляются сверху: 200 + 20·2 + 4·2 = 248. При `border-box` — `width` задаёт ширину всего блока целиком.

Поэтому почти в каждом проекте первой строкой стоит:

```
* { box-sizing: border-box; }
```

> **ЗАПОМНИ**
> Если блоки «не помещаются» или «вылезают» ровно на величину отступов — дело почти всегда в `box-sizing`. Это первое, что надо проверить.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> Почему это правило не применяется к .card__title?
> Какой селектор его перебивает? Код не меняй, объясни.

> **ЗАПРОС**
> Разложи эти карточки в три колонки с промежутком 16 пикселей,
> на узком экране — в одну. Используй grid, без float
> и position: absolute.

ЗАПРОС

```
В этом CSS есть !important. Разберись, какое правило мешало,
и перепиши без него.
```

НА ЧТО СМОТРЕТЬ

`!important`. Признак перебитого конфликта, а не решения.

Магические числа. `margin-top: 37px`, `width: 433px` — подогнано на глаз под один экран и развалится на другом.

`position: absolute` **для раскладки.** Элемент выпадает из потока, соседи его не видят, на другом размере экрана всё съедет.

Фиксированная высота у текстового блока. `height: 40px` на элементе с текстом: при другом шрифте или длинном слове текст обрежется.

Повторяющиеся значения. Один и тот же отступ, вписанный в десяти местах. В современном CSS для этого есть переменные: `var(--gap)`.

Стили «на всякий случай». `float: left` рядом с `display: flex` — одно из двух не работает, и непонятно какое.

ШПАРГАЛКА

- Селектор отвечает «к чему», свойства — «что сделать».
- Пробел в селекторе значит «внутри»: `.a .b` ≠ `.a.b`.
- Вес: тег < класс < id < `style` < `!important` при равенстве — что ниже.
- `!important` в коде от ИИ — перебитый конфликт, а не решение.
- Раскладка сегодня — `flex` и `grid` `float` устарел.
- Блок шире, чем задал? Проверь `box-sizing`.

8.4 Инструменты разработчика

консоль, сеть, элементы

ЧТО ЭТО

Инструменты разработчика — встроенная в браузер панель, которая показывает, что на самом деле происходит на странице. Открывается клавишей **F12** (или Ctrl+Shift+I, на macOS — Cmd+Option+I).

Для фронтенда это то же, что логи для сервера (статья 4.2): единственный способ узнать, что произошло, вместо того чтобы гадать.

Вкладок много, нужны три.

КАК ВЫГЛЯДИТ В КОДЕ

Вот что показывает браузер на странице, где три разные проблемы:

```
[сеть ✗]  https://example.invalid/style.css —
net::ERR_TUNNEL_CONNECTION_FAILED
[консоль error]   Failed to load resource:
net::ERR_TUNNEL_CONNECTION_FAILED
[консоль log]     приложение стартовало
[консоль warning] ключ API не задан, работаю в демо-режиме
[консоль log]     получено: {"main":null}
[ошибка страницы] Cannot set properties of null (setting 'textContent')
```

Шесть строк, и в них три разные истории:

1. **Файл стилей не загрузился.** Страница при этом показалась — просто без оформления. Это та самая четвёртая причина «стиль не применился» из статьи 8.1, и увидеть её можно только здесь.
2. **Предупреждение о демо-режиме.** Не ошибка, но объясняет, почему данные выглядят странно.
3. `Cannot set properties of null` — код искал элемент `#missing`, не нашёл, получил `null` и попытался записать в него текст. Знакомо: это тот же класс ошибки, что `KeyError` в Python и `'temp' is possibly null` в TypeScript.

> **ЗАПОМНИ**
> `Cannot read/set properties of null` — самая частая ошибка во фронтенде, и означает она почти всегда одно: селектор не нашёл элемент. Либо опечатка в селекторе, либо скрипт выполнился раньше, чем появился элемент.

Вкладка «Консоль»

Что здесь смотреть:

- **ошибки** — красным, с местом в коде;
- **предупреждения** — жёлтым, часто объясняют причину;
- **вывод** `console.log` из кода.

Консоль не только показывает, но и принимает команды. Прямо в ней можно проверить, есть ли элемент:

```
document.querySelector('.card')      // null — значит, не нашёлся
document.querySelectorAll('.card').length
```

Это самый быстрый ответ на вопрос «элемент вообще есть в DOM?» из статьи 8.1.

Вкладка «Сеть»

Список всех запросов: файлы стилей, скрипты, картинки, обращения к API. Смотреть на три колонки:

Колонка	О чём говорит
Статус	код ответа (статья 7.2): `200`, `404`, `500`
Размер	не загрузился ли мегабайт вместо килобайта
Время	что именно тормозит

Клик по запросу показывает заголовки и тело — то же, что отдавал `curl` в главе 7, только для живой страницы.

Самая частая находка: запрос к API ушёл, ответ `200`, но данные не те. Значит, дело не во фронтенде, а в бэкенде, — и дальше статья 7.8.

Вкладка «Элементы»

Показывает **DOM**, а не исходный HTML. Разница принципиальная: здесь видно то, что получилось после разбора и работы скриптов (статья 8.1).

Справа — панель стилей выбранного элемента. Там видно:

- какие правила применились;
- **какие перечёркнуты** — то есть перебиты другими (статья 8.3);
- откуда каждое правило пришло: файл и строка.

Перечёркнутое правило — готовый ответ на вопрос «почему стиль не применился».

КАК СКАЗАТЬ ИИ

Главная польза инструментов для работы с ИИ — они превращают «не работает» в факты (статья 4.3):

ТАК НЕ НАДО

```
стили не применяются, почини
```

ТАК ЛУЧШЕ

```
В консоли: Failed to load resource — style.css, net::ERR_...
Во вкладке «Сеть»: запрос к /style.css, статус 404.
Путь в разметке: <link rel="stylesheet" href="/style.css">.
Файл лежит в static/style.css. Как должен выглядеть путь?
```

ЗАПРОС

```
Ошибка: Cannot set properties of null (setting 'textContent').
Вот код: <код>. Элемент #missing в разметке есть.
Почему querySelector возвращает null?
```

ЗАПРОС

```
Во вкладке «Элементы» правило color из .card перечёркнуто.
Вот оба правила: <код>. Какое побеждает и почему?
```

НА ЧТО СМОТРЕТЬ

Консоль не открывали. Самая частая причина долгих поисков: ошибка всё это время была написана текстом.

Ошибки «чужие». Расширения браузера сыплют в ту же консоль. Проверяй файл и строку: если это не твой файл — не твоя проблема.

`console.log` **остался в коде.** Для фронтенда это ещё и утечка: видно всем, кто откроет консоль. Перед выкладкой — поиск по диффу (статья 4.5).

Смотрят в исходный код вместо DOM. `Ctrl+U` показывает то, что пришло с сервера; вкладка «Элементы» — то, что есть сейчас. При работе скриптов это разные вещи.

Кеш. Правка не видна, потому что браузер взял старый файл. Обновление с очисткой — Ctrl+Shift+R.

ШПАРГАЛКА

- F12, три вкладки: Консоль · Сеть · Элементы.
- `Cannot read properties of null` — селектор не нашёл элемент.
- Проверка в консоли: `document.querySelector('.card')` вернул `null`?
- Сеть: статус, размер, время. `404` на стилях объясняет «всё съехало».
- Элементы показывают DOM, а не исходный HTML.
- Перечёркнутое правило в панели стилей — его перебили.

8.5 Ошибки ИИ во фронтенде и чек-лист

проверка глазами — тоже проверка

ЧТО ЭТО

Завершение главы. Фронтенд отличается от всего предыдущего в книге одним: **здесь результат проверяется глазами.** Код запу-

стился, ошибок нет, а кнопка не нажимается или текст налез на картинку — и узнать об этом можно, только посмотрев.

ОШИБКИ ИИ ВО ФРОНТЕНДЕ

1. `div` вместо семантики
`<div class="btn">` с обработчиком нажатия. Не работает с клавиатуры, невидима для программ чтения с экрана (статья 8.2).

2. Устаревшая раскладка
`float`, `position: absolute` и таблицы для расстановки блоков. Код для той эпохи, которой в обучающих данных больше (статья 3.1, причина 3). Сегодня это `flex` и `grid`.

3. `!important` вместо разбора
Перебивает конфликт силой, не решая его (статья 8.3).

4. Магические числа
`margin-top: 37px`, `width: 433px` — подогнано под один экран.

5. Нет адаптива
Страница выглядит хорошо ровно на той ширине, на которой её сочинили. На телефоне — горизонтальная прокрутка.

6. Стили прямо в разметке
`style="..."` в каждом элементе: перебивает внешние правила и не переиспользуется.

7. Доступность забыта
Нет `alt`, нет `label`, контраст текста ниже нормы, фокус не виден. ИИ добавляет это, только если попросить прямо.

8. Скрипт выполняется раньше разметки
`<script>` в `<head>` без `defer` — `querySelector` возвращает `null`, потому что элемента ещё нет (статья 8.4).

ЧЕК-ЛИСТ ФРОНТЕНДА

К общему чек-листу приёмки (статья 4.7) добавляются шесть шагов. Первые три — глазами, вторые три — в инструментах разработчика.

1. Посмотреть
Открыть и посмотреть. Банально, но это и есть главная проверка главы: ни тесты, ни типы не скажут, что текст налез на кнопку.

2. Узкий экран
Сузить окно до ширины телефона. Появилась горизонтальная прокрутка — адаптива нет.

3. Длинный текст
Вписать в поле или заголовок строку втрое длиннее ожидаемой. Вёрстка, собранная под короткие подписи, разваливается именно здесь.

4. Консоль
F12, вкладка «Консоль». Красного быть не должно.

5. Сеть
Все ли файлы загрузились. `404` на стилях или скрипте объясняет большинство странностей.

6. Клавиатура
Пройти по странице клавишей Tab. Все кнопки и ссылки должны получать фокус, и фокус должен быть виден. Это тридцать секунд и самая частая находка в сгенерированной разметке.

> **СОВЕТ**
> Шаги 2, 3 и 6 — те, которые не делает почти никто и которые ловят почти всё. Узкий экран, длинный текст, клавиатура: минута на все три.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Проверь эту вёрстку по списку:
- есть ли div с обработчиками вместо button и a
- используется ли float или position: absolute для раскладки
- есть ли !important и магические числа в пикселях
- что будет на экране шириной 360 пикселей
- есть ли alt у картинок и label у полей

Что из этого проблема? Если ничего — так и скажи.
```

ЗАПРОС
```
Сделай этот блок адаптивным: на широком экране три колонки,
на узком одна. Через grid, без медиазапросов на каждую ширину —
используй auto-fit и minmax.
```

ЗАПРОС
```
Проверь доступность: порядок заголовков, alt, label, контраст,
видимость фокуса. Список проблем с указанием строк.
```

ШПАРГАЛКА

- Фронтенд проверяется глазами: ни типы, ни тесты не покажут съехавшую вёрстку.
- Восемь частых ошибок: `div` вместо кнопки, `float`, `!important`, магические числа, нет адаптива, стили в разметке, нет доступности, скрипт до разметки.
- Чек-лист: посмотреть · узкий экран · длинный текст · консоль · сеть · Tab.
- Узкий экран, длинный текст и клавиатура ловят почти всё и стоят минуты.
- Красного в консоли быть не должно.
- Доступность ИИ добавляет только по прямой просьбе.

9 JavaScript и TypeScript

Язык, на котором написан весь фронтенд, и надстройка, которая делает его проверяемым. Типы здесь не формальность — они ловят до запуска ровно тот класс ошибок, которому посвящена глава 4.

9.1 Читаем JavaScript

шесть мест, где ломается питоновская интуиция

ЧТО ЭТО

JavaScript похож на Python ровно настолько, чтобы вводить в заблуждение. Циклы, функции, условия выглядят знакомо — а потом код делает не то, и непонятно почему.

В статье шесть мест, где интуиция подводит. Всё ниже — настоящий вывод `node`, а не пересказ документации.

КАК ВЫГЛЯДИТ В КОДЕ

1. Сортировка чисел сортирует не числа

```
[10, 9, 1, 100].sort()
```

```
[ 1, 10, 100, 9 ]
```

По умолчанию `sort` превращает элементы в строки и сравнивает их как текст. Для чисел нужна функция сравнения:

```
[10, 9, 1, 100].sort((a, b) => a - b)     // [ 1, 9, 10, 100 ]
```

Ошибка тихая: на числах от 1 до 9 она незаметна, на двузначных — ломается.

2. Два вида равенства

```
'1' ==  1   ->  true
'1' === 1   ->  false
null ==  undefined   ->  true
null === undefined   ->  false
```

`==` приводит типы перед сравнением, `===` — нет. **Правило простое: всегда `===`.** Единственное оправданное исключение — `x == null`, которое разом проверяет и `null`, и `undefined`.

3. Истинность не та, к которой ты привык

```
[]        -> истина       (в Python пустой список ложен!)
{}        -> истина
""        -> ложь
"0"       -> истина
0         -> ложь
NaN       -> ложь
null      -> ложь
undefined -> ложь
```

Первая строка ломает больше всего кода. Проверка «есть ли элементы» через `if (items)` в JavaScript **всегда истинна**, даже для пустого массива. Нужно `if (items.length)`.

4. Плюс склеивает, минус вычитает

```
1 + '1'   ->  '11'      (склеил строки)
1 - '1'   ->  0         (превратил в числа)
```

Отсюда классика: данные пришли из формы или JSON строками, и вместо суммы получается склейка. `'10' + 5` даёт `'105'`.

5. Два вида «ничего»

	Что значит
`undefined`	значения не было: нет такого ключа, функция ничего не вернула
`null`	значение есть, и оно — «ничего»

Обращение к несуществующему ключу **не падает**:

```
user.email              ->  undefined
user.email?.length      ->  undefined
user.email ?? 'не указан'  ->  'не указан'
```

Это противоположность Python, где такое даёт `KeyError`. Удобно и опасно одновременно: ошибка не сообщает о себе, а всплывает дальше — в виде `Cannot read properties of undefined` (статья 8.4).

Два оператора, которые встречаются постоянно:

- `?.` — «если слева не `null`/`undefined`, иди дальше, иначе верни `undefined`»;
- `??` — «возьми левое, а если оно `null`/`undefined`, возьми правое».

`??` отличается от `||` тем, что не считает `0` и `""` пустотой. Для чисел это принципиально: `temp || 20` превратит ноль градусов в двадцать, а `temp ?? 20` — нет.

6. `const` не делает объект неизменяемым

```
const cities = ["Oslo"];
cities.push("Bergen");      // можно
cities = [];                // нельзя
```

`const` запрещает переприсваивать переменную, но не менять содержимое. То же самое, что ссылки в Python (статья 2.4).

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Перепиши этот код на строгие сравнения: везде === вместо ==,
проверки массивов через .length, ?? вместо || там,
где значение может быть 0 или пустой строкой.
Поведение не меняй.
```

ЗАПРОС

```
Этот код пришёл из Python. Что здесь будет работать иначе
из-за особенностей JavaScript? Только список различий.
```

ЗАПРОС

```
Числа приходят из JSON строками. Где в этом коде может
получиться склейка вместо сложения?
```

СОВЕТ

Большинство этих ловушек ловятся линтером. Одна настройка `eslint` с правилом `eqeqeq` — и `==` перестанет проходить. Для JavaScript без типов линтер обязателен: это единственная механическая проверка, которая остаётся.

НА ЧТО СМОТРЕТЬ

`==` в сгенерированном коде. Почти всегда должно быть `===`.

`sort()` без функции сравнения на числах.

`if (array)` вместо `if (array.length)`. Условие всегда истинно.

`||` для значений по умолчанию. Ломается на `0` и `""`. Нужен `??`.

`var`. Устаревшее объявление с неожиданной областью видимости. В новом коде только `let` и `const` (статья 3.1, причина 3).

Числа из строк без преобразования. `Number(x)` или `parseInt(x, 10)` — и всегда с основанием 10.

ШПАРГАЛКА

- `sort()` по умолчанию сортирует как строки: `[1, 10, 100, 9]`.
- Всегда `===`. Исключение — `x == null` для обеих пустот сразу.
- Пустой массив — истина. Проверяй `.length`.
- `+` со строкой склеивает, `-` преобразует в числа.
- `?.` безопасный доступ, `??` значение по умолчанию (не путать с `||`).
- `const` запрещает переприсваивание, но не изменение содержимого.

9.2 Асинхронность в JavaScript

промисы, await и четыре места, где ИИ ошибается

ЧТО ЭТО

Асинхронность в JavaScript устроена так же, как в Python (статья 2.10): `async`, `await`, ожидание вместо простоя. Но есть своя терминология и свои ловушки, и все четыре ниже ИИ воспроизводит регулярно.

Здесь же — прямая параллель с главой 6: `Promise.all` и `Promise.allSettled` соотносятся ровно так же, как `gather` и `gather(return_exceptions=True)`.

КАК ВЫГЛЯДИТ В КОДЕ

```
const fetchCity = (city, ms = 100) =>
  new Promise((resolve, reject) =>
    setTimeout(() => city === "Bergen"
      ? reject(new Error("сервер вернул 500"))
      : resolve(`${city}: +4`), ms));
```

Промис (promise) — обещание результата, который появится позже. У него три состояния: ожидает, выполнен, отклонён. `resolve` — успех, `reject` — ошибка.

Писать `new Promise` вручную почти никогда не нужно: промисы возвращают `fetch` и библиотеки. Но знать, что это такое, надо — слово встречается постоянно.

1. `Promise.all` роняет всё целиком

```
await Promise.all([fetchCity("Oslo"), fetchCity("Bergen"),
fetchCity("Tromsø")]);
```

```
упало целиком: сервер вернул 500
```

Запросы к Осло и Тромсё прошли успешно — их результаты потеряны. Ровно то же, что `asyncio.gather` без защиты в Python.

2. `Promise.allSettled` сохраняет успехи

```
const r = await Promise.allSettled([...]);
for (const x of r)
  console.log(x.status === "fulfilled" ? `ок: ${x.value}` : `ошибка:
${x.reason.message}`);
```

```
ок: Oslo: +4
ошибка: сервер вернул 500
ок: Tromsø: +4
```

Каждый элемент — объект со `status`. У успешного есть `value`, у неудачного — `reason`. Проверять `status` обязательно: иначе ошибка уедет дальше как обычное значение.

Нужно	Что брать
Все должны получиться, иначе смысла нет	`Promise.all`
Частичный результат лучше, чем ничего	`Promise.allSettled`
Достаточно первого успешного ответа	`Promise.any`
Важен первый ответ любой, даже ошибка (например, гонка с таймаутом)	`Promise.race`

3. `forEach` не ждёт

Самая коварная из четырёх:

```
const done = [];
["Oslo", "Tromsø"].forEach(async c => { done.push(await
fetchCity(c)); });
console.log("сразу после forEach:", done);
```

```
сразу после forEach: []
через 300 мс:        [ 'Oslo: +4', 'Tromsø: +4' ]
```

Массив пуст. `forEach` не умеет ждать `async`-функции: он запускает их все и идёт дальше, не дожидаясь. Данные появятся — но позже, когда код, который их ждал, уже отработал.

Правильно — обычный цикл:

```
for (const c of ["Oslo", "Tromsø"]) ok.push(await fetchCity(c));
```

```
после цикла: [ 'Oslo: +4', 'Tromsø: +4' ]
```

Или `Promise.all` с `map`, если запросы независимы и хочется параллельно.

> **ВНИМАНИЕ**
> `forEach` с `async` внутри — ошибка, которую не видно при чтении: выглядит как обычный цикл. Признак в коде: `forEach(async ...)`. Заменяется на `for...of` или `Promise.all(items.map(...))`.

4. Забытый `await`

```
без await: Promise { <pending> }
с await:   Oslo: +4
```

Вместо значения — объект промиса. В Python это `<coroutine object>`, здесь `Promise { <pending> }` — тот же симптом, другое слово.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> Эти три запроса независимы. Нужен частичный результат:
> если один сервер не ответил, остальные должны дойти.
> Используй Promise.allSettled и разбери status.

> **ЗАПРОС**
> В этом коде forEach с async внутри. Перепиши так,
> чтобы результаты действительно дожидались: for...of,
> если запросы зависят друг от друга, или Promise.all с map,
> если нет.

> **ЗАПРОС**
> Добавь таймаут 5 секунд на каждый запрос через AbortSignal.timeout.
> При таймауте — предупреждение в консоль и пропуск, а не падение.

НА ЧТО СМОТРЕТЬ

`Promise.all` там, где нужен частичный результат. Одна ошибка теряет всё.

`forEach(async ...)`. Не ждёт. Всегда ошибка, если результат нужен.

Нет `.catch` и нет `try`. Отклонённый промис, который никто не поймал, даёт `UnhandledPromiseRejection` — в Node это падение процесса.

Последовательные `await` в цикле там, где можно параллельно. Не ошибка, но втрое медленнее на трёх запросах (статья 2.10).

Нет таймаута. Запрос может висеть вечно (статья 6.4). В браузере и Node для этого есть `AbortSignal.timeout(ms)`.

`.then()` вперемешку с `await`. Работает, читается плохо. В новом коде — `await`.

ШПАРГАЛКА

- Промис — обещание результата; `resolve` успех, `reject` ошибка.
- `Promise.all` роняет всё при первой ошибке, `allSettled` сохраняет успехи.
- У `allSettled` проверяй `status`: `fulfilled` → `value`, `rejected` → `reason`.
- `forEach(async ...)` не ждёт. Нужен `for...of` или `Promise.all(map)`.
- `Promise { <pending> }` в выводе — забыт `await`.
- Непойманный отклонённый промис роняет процесс в Node.

9.3 Что добавляет TypeScript

тот же JavaScript, но с проверкой

ЧТО ЭТО

TypeScript — это JavaScript плюс описания типов. Браузер его не исполняет: перед запуском типы **стираются**, и остаётся обычный JavaScript. Вся польза происходит до запуска, в момент проверки.

Для работы с ИИ это принципиально. Вернись к статье 6.3: там `mypy` поймал забытую проверку на `null` в Python до запуска. TypeScript делает то же самое для фронтенда — и делает это строже, потому что здесь типы не опциональная добавка, а основной режим работы.

КАК ВЫГЛЯДИТ В КОДЕ

```
interface City {
  name: string;
  temp: number | null;
  tags?: string[];
}

function report(c: City): string {
  if (c.temp === null) return `${c.name}: нет данных`;
  return `${c.name}: ${c.temp > 0 ? "+" : ""}${c.temp}°`;
}

const cities: City[] = [
  { name: "Oslo", temp: 4 },
  { name: "Tromsø", temp: null },
  { name: "Bergen", temp: -2, tags: ["дождь"] },
];

cities.forEach(c => console.log(report(c)));
```

```
tsc: ошибок нет
Oslo: +4°
Tromsø: нет данных
Bergen: -2°
```

Как это читать

`interface City { ... }` — описание формы объекта. Не класс, не код: после компиляции от него ничего не останется.

`temp: number | null` — union, «или одно, или другое». Самая полезная запись в TypeScript: она прямо говорит, что значения может не быть, и компилятор заставит это проверить.

`tags?: string[]` — необязательное поле. Его тип на самом деле `string[] | undefined`.

`function report(c: City): string` — принимает `City`, возвращает строку. То же, что сигнатура в Python (статья 6.5), и работает так же: заменяет абзац объяснений.

Что происходит без проверки на `null`

Уберём `if` и попросим компилятор проверить:

```ts
function report(c: City): string {
  return `${c.name}: ${c.temp.toFixed(1)}°`;
}
```

```
error TS18047: 'c.temp' is possibly 'null'.
```

Код не скомпилируется. Не упадёт у пользователя, не пройдёт мимо тестов — просто не соберётся, пока проверку не добавишь.

Та же ошибка в Python выглядела как `Item "None" of "str | None" has no attribute "upper"`, в JavaScript без типов — как `Cannot read properties of null` уже во время работы (статья 8.4). Три языка, один класс ошибки, и только здесь он пойман до запуска.

> **ЗАПОМНИ**
> Главное, что даёт TypeScript, — не «красивый код», а то, что забытая проверка на пустоту перестаёт быть возможной. Это самая частая ошибка ИИ во всей книге, и она закрывается настройкой.

`strict` — не опция, а смысл

```
{
  "compilerOptions": {
    "strict": true,
    "noUncheckedIndexedAccess": true
  }
}
```

Без `strict` проверка на `null` отключена, и TypeScript превращается в украшение. Включать обязательно.

`noUncheckedIndexedAccess` — вторая по полезности настройка. С ней обращение по индексу даёт `T | undefined`, потому что элемента может не быть:

```
const cities = ["Oslo", "Bergen"];
const first: string = cities[5];
```

```
error TS2322: Type 'string | undefined' is not assignable to type
'string'.
```

Компилятор напоминает о том, о чём забывают все: индекс может выйти за границы.

Как запускать

```
npx tsc --noEmit        # только проверить, ничего не собирать
npx tsx app.ts          # запустить напрямую
```

`--noEmit` — то, что нужно чаще всего: проверка без сборки, аналог `mypy --strict`.

КАК СКАЗАТЬ ИИ

ЗАПРОС
Переведи этот модуль на TypeScript: интерфейсы для данных,
типы у всех функций. tsc --noEmit со strict должен проходить.
Логику не меняй.

ЗАПРОС
В tsconfig включи strict и noUncheckedIndexedAccess.
Исправь ошибки, которые появятся, не отключая проверки
и без as any.

Вторая фраза существенная: самый простой способ «исправить» ошибку типов — это её заглушить, и ИИ к этому склонен (статья 9.6).

ЗАПРОС

```
Опиши типы для ответа этого API: вот пример JSON.
Поля, которых может не быть, пометь как необязательные
или | null — честно, а не «чтобы компилировалось».
```

НА ЧТО СМОТРЕТЬ

`strict: false` или его отсутствие. Проверка на `null` выключена, и главная польза потеряна.

`any` в типах. Отключает проверку для этого места полностью.

`as` для успокоения компилятора. Про это — отдельно в статье 9.5.

Типы написаны, но `tsc` не запускался. То же, что аннотации без `mypy`: украшение, которое разошлось с кодом.

Файлы `.js` вперемешку с `.ts`. Переход на полпути: в `.js` проверок нет, и ошибки прячутся там.

ШПАРГАЛКА

- TypeScript — это JavaScript плюс типы; при запуске типы стираются.
- `interface` описывает форму, `|` — union, `?` — необязательное поле.
- `number | null` заставляет проверить пустоту — код иначе не соберётся.
- `strict: true` обязателен, иначе проверки на `null` нет.
- `noUncheckedIndexedAccess` — обращение по индексу даёт `| undefined`.
- `npx tsc --noEmit` — проверка без сборки, аналог `mypy --strict`.

9.4 Как читать чужие типы

union, сужение, дженерики, утилитные типы

ЧТО ЭТО

Типы, которые пишет ИИ или библиотека, выглядят сложнее тех, что пишешь ты сам: угловые скобки, вертикальные черты, незнакомые слова. Читаются они проще, чем кажутся, — надо знать четыре конструкции.

Эта статья — про чтение. Писать такие типы с нуля не придётся.

КАК ВЫГЛЯДИТ В КОДЕ

Union и сужение

```
type Result =
  | { ok: true; data: string[] }
  | { ok: false; error: string };

function show(r: Result): string {
  if (r.ok) return `получено ${r.data.length}`;
  return `ошибка: ${r.error}`;
}
```

```
show({ ok: true, data: ["Oslo", "Bergen"] })  ->  получено 2
show({ ok: false, error: "таймаут" })         ->  ошибка: таймаут
```

Читается так: *`Result` — это либо объект с `ok: true` и `data`, либо объект с `ok: false` и `error`.*

Самое интересное — что происходит внутри `if`. После проверки `r.ok` компилятор **знает**, какой из двух вариантов перед ним: в первой ветке есть `data` и нет `error`, во второй наоборот. Обратишься к `r.error` в первой ветке — ошибка компиляции.

Это называется **сужение типа** (narrowing), и это главное, ради чего union используют. Такая пара «флаг плюс данные» — стандартный способ вернуть «получилось или нет», не бросая исключение.

Сужение делают и обычные проверки:

Проверка	Что сужает
`if (x === null)`	убирает `null` из union
`typeof x === "string"`	оставляет строки
`Array.isArray(x)`	оставляет массивы
`if ("data" in r)`	по наличию поля

Дженерики

```
function firstOr<T>(items: T[], fallback: T): T {
  return items.length > 0 ? items[0]! : fallback;
}
```

```
firstOr(["Oslo"], "нет")   ->  Oslo      (T = string)
firstOr([10, 20], 0)       ->  10        (T = number)
```

`<T>` — не конкретный тип, а **заполнитель**: «какой тип придёт, такой и вернётся». Благодаря этому функция работает с любыми

данными и при этом не теряет тип: результат `firstOr([10, 20], 0)` — число, а не «что-то».

Читать дженерики надо как шаблон с дыркой: подставь вместо `T` конкретный тип из вызова, и всё станет понятно.

Встречаются постоянно: `Array<string>` (то же, что `string[]`), `Promise<City>` (промис, который вернёт `City`), `Map<string, number>`.

Утилитные типы

Готовые преобразования типов. Четыре, которые покрывают почти всё:

```
interface City { name: string; temp: number; country: string }

type CityPatch     = Partial<City>;              // все поля
необязательны
type CityKey       = Pick<City, "name" | "temp">;// только эти
type CityNoCountry = Omit<City, "country">;      // все, кроме
type ByName        = Record<string, City>;       // словарь
```

Запись	Читается как
`Partial<T>`	«все поля `T`, но необязательные»
`Pick<T, K>`	«только поля `K` из `T`»
`Omit<T, K>`	«все поля `T`, кроме `K`»
`Record<K, V>`	«словарь: ключи `K`, значения `V`»

`Partial` чаще всего встречается у функций обновления: прислать можно часть полей. `Record<string, City>` — то же, что `dict[str, City]` в Python.

> **СОВЕТ**
> Не понял тип — наведи на него курсор в редакторе. Он покажет развёрнутый вид: во что на самом деле превращается `Omit<City, "country">`. Это быстрее, чем разбирать по частям.

Что ещё встретится

`unknown` — «тип неизвестен, и пока не проверишь, ничего сделать нельзя». Правильная замена `any`: заставляет проверить.

`as const` — «это значение не изменится», превращает `"paid"` из просто строки в точное значение.

`keyof City` — union из имён полей: `"name" | "temp" | "country"`.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Объясни этот тип по частям, как будто я вижу дженерики
впервые. Что сюда подставляется и что получается на выходе.
```

ЗАПРОС

```
Перепиши эту функцию так, чтобы она возвращала
{ ok: true, data } | { ok: false, error } вместо того,
чтобы бросать исключение.
```

ЗАПРОС

```
Здесь три почти одинаковых интерфейса. Можно ли выразить их
через Pick/Omit от одного базового? Покажи, как.
```

НА ЧТО СМОТРЕТЬ

Union без сужения. Если тип `A | B`, а в коде ни одной проверки — значит, компилятор заставили замолчать через `as`.

Дженерик, который везде `any`. `<T = any>` отключает всю пользу.

Типы продублированы. Три интерфейса с перекрывающимися полями вместо `Pick`/`Omit` от одного. Разъедутся при первой правке.

`unknown`, которое сразу приводят через `as`. Смысл `unknown` в том, чтобы проверить, а не в том, чтобы обойти.

Слишком умный тип. Если тип занимает десять строк и его никто не может объяснить — он сложнее задачи. ИИ иногда увлекается.

ШПАРГАЛКА

- `A | B` — либо одно, либо другое; проверка сужает до одного.
- Пара `{ ok: true, data } | { ok: false, error }` — стандартный «получилось или нет».
- `<T>` — заполнитель: какой тип пришёл, такой и вернётся.
- `Partial` необязательные, `Pick` только эти, `Omit` все кроме, `Record` словарь.
- `unknown` — честная замена `any`: заставляет проверить.
- Не понял тип — наведи курсор, редактор покажет развёрнутый вид.

9.5 Типы как проверка работы ИИ

пять ошибок, пойманных до запуска, и одна, которая проходит

ЧТО ЭТО

Главная статья главы. Вернись к главе 4: там десять статей о том, как **руками** ловить ошибки, которых не видно при запуске. Типы делают часть этой работы механически, при каждой проверке, бесплатно.

Ниже — пять настоящих ошибок ИИ и настоящие сообщения компилятора на них. А потом — честное ограничение: одна ошибка того же рода проходит мимо, и знать про неё важнее, чем про остальные пять.

КАК ВЫГЛЯДИТ В КОДЕ

1. Забыл проверить пустоту

Самая частая ошибка ИИ во всей книге:

```
function report(c: City): string {
  return `${c.name}: ${c.temp.toFixed(1)}°`;
}
```

```
error TS18047: 'c.temp' is possibly 'null'.
```

2. Перепутал имя поля

```
const c: City = { name: "Oslo", temperature: 4 };
```

```
error TS2353: Object literal may only specify known properties,
and 'temperature' does not exist in type 'City'.
```

В JavaScript без типов это тихо создало бы объект с лишним полем и без нужного. Ошибка всплыла бы в другом месте и выглядела бы как `undefined`.

3. Перепутал порядок аргументов

```
function subscribe(userId: number, city: string): void {}
subscribe("Oslo", 7);
```

```
error TS2345: Argument of type 'string' is not assignable
to parameter of type 'number'.
```

Классика при работе с ИИ: он помнит функцию с похожим именем и другим порядком аргументов.

4. Забыл вариант в union

```
type Status = "paid" | "pending" | "refunded";

function label(s: Status): string {
  switch (s) {
    case "paid": return "оплачен";
    case "pending": return "ожидает";
  }
}
```

```
error TS2366: Function lacks ending return statement
and return type does not include 'undefined'.
```

Забыт `refunded`. Без типов функция молча вернула бы `undefined`, и в интерфейсе появилась бы пустота вместо подписи.

Это важный случай: добавишь завтра четвёртый статус — компилятор найдёт все места, где его не обработали. Руками такое не ищут.

5. Обращение по индексу за границей

```
const cities = ["Oslo", "Bergen"];
const first: string = cities[5];
```

```
error TS2322: Type 'string | undefined' is not assignable to type
'string'.
```

Работает только с `noUncheckedIndexedAccess` (статья 9.3).

ГДЕ ТИПЫ НЕ ПОМОГАЮТ

Теперь честно. Вот код, который компилятор пропускает **без единой ошибки**:

```
interface WeatherResponse { main: { temp: number }; name: string }

async function fetchWeather(city: string): Promise<WeatherResponse> {
  const raw = '{"cod":"404","message":"city not found"}';
  return JSON.parse(raw) as WeatherResponse;
}

const data = await fetchWeather("Tonsberg");
console.log(data.main.temp);
```

```
tsc: ошибок нет
```

```
упало: Cannot read properties of undefined (reading 'temp')
```

`as WeatherResponse` — это обещание компилятору, а не проверка. Ты сказал «поверь, тут такая форма», он поверил. Пришло другое

— и тот же самый `Cannot read properties of undefined`, с которого начиналась глава 8.



Рис. 9.1 · Типы проверяют твой код сам с собой. Данные, пришедшие снаружи, они только описывают — проверять их надо отдельно.

Граница — это всё, что пришло снаружи программы:

- ответ API (`fetch`, `JSON.parse`);
- данные формы;
- `localStorage`
- параметры из адреса страницы.

> **ЗАПОМНИ**
> Типы проверяют согласованность твоего кода с самим собой. Они **не проверяют** данные, пришедшие извне. Это ровно та же граница, что в статье 7.3 на сервере: там её закрывает схема запроса, здесь — проверка во время выполнения.

Чем закрывать границу

Библиотеками проверки во время выполнения — `zod`, `valibot` и подобными. Они делают для фронтенда то же, что `pydantic` для бэкенда (статья 7.3): описываешь форму один раз, и она и проверяет данные, и даёт тип.

ЗАПРОС

```
Ответ API сейчас приводится через `as`. Замени на проверку
во время выполнения через zod: схема, разбор, понятная ошибка,
если форма не совпала. Тип выведи из схемы, не дублируй руками.
```

Последняя фраза важна: смысл в том, чтобы форма описывалась **один раз** и служила и проверкой, и типом. Два описания рядом разъедутся.

КАК СКАЗАТЬ ИИ

Типы — ещё и способ ставить задачу. Сигнатура точнее слов (статьи 3.4, 6.5):

ЗАПРОС

```
Напиши функцию под эту сигнатуру:

async function loadSubs(userId: number): Promise<Result<City[]>>
```

```
где Result<T> = { ok: true; data: T } | { ok: false; error: string }.
Исключения наружу не бросай, ошибки возвращай вторым вариантом.
tsc --noEmit со strict должен проходить.
```

Проверка готового кода:

ЗАПРОС

```
Прогони tsc --strict по своему коду мысленно: какие ошибки
он найдёт? Исправь их, не используя any, as и ts-ignore.
```

Запрет в последней строке обязателен — иначе «исправление» будет заглушкой (статья 9.6).

ШПАРГАЛКА

- Типы ловят до запуска: забытую пустоту, неверное поле, порядок аргументов, пропущенный вариант union, выход за границу массива.
- Добавил вариант в union — компилятор найдёт все необработанные места.
- `as` — обещание компилятору, а не проверка. Данные извне он не смотрит.
- Граница: ответ API, формы, `localStorage`, адрес страницы.
- Границу закрывают проверкой во время выполнения, например `zod`.
- Тип выводи из схемы, а не дублируй рядом.

9.6 Ошибки ИИ в JS и TS и чек-лист

как отличить решение от заглушки

ЧТО ЭТО

Завершение главы. У JavaScript и TypeScript есть своя особенность: **здесь у ИИ есть лёгкий способ заставить проверку замолчать.** Три символа — и ошибка исчезла, не будучи исправленной.

Отличать исправление от заглушки — главный навык этой статьи.

ТРИ ЗАГЛУШКИ, КОТОРЫЕ НАДО УЗНАВАТЬ

1. `any`

```ts
function handle(data: any) { ... }
```

Отключает проверку целиком. Всё, что приходит из `any`, компилятор перестаёт смотреть, и зараза расползается дальше по коду.

Честная замена: `unknown`, которая заставляет проверить, или нормальный тип.

2. `as`

```
const city = data as City;
```

Обещание компилятору, разобранное в статье 9.5. Опаснее `any`, потому что выглядит как типизированный код.

Когда оправдано: редко и осознанно — например, после собственной проверки, которую компилятор не понимает. Тогда рядом должен быть комментарий, почему.

3. `@ts-ignore` и `@ts-expect-error`

```
// @ts-ignore
widget.render(options);
```

Выключает проверку следующей строки. Всегда требует объяснения: если причина не записана, через месяц никто не вспомнит, что здесь скрыто.

> **ВНИМАНИЕ**
> Если после просьбы «исправь ошибки типов» в диффе появились `any`, `as` или `ts-ignore` — ошибки не исправлены, а спрятаны. Это прямой аналог `except Exception: pass` из статьи 2.14, только для типов.

ОСТАЛЬНЫЕ ЧАСТЫЕ ОШИБКИ

1. `strict` **выключен.** Без него нет проверок на `null` — главной пользы (статья 9.3).
2. `Promise.all` **вместо** `allSettled`, когда нужен частичный результат (статья 9.2).
3. `forEach(async ...)`**.** Не ждёт. Всегда ошибка, если результат нужен.
4. `==` **вместо** `===` и `||` вместо `??` (статья 9.1).
5. **Типы продублированы руками.** Отдельно схема проверки, отдельно интерфейс — разъедутся.
6. **Код для старых версий.** `var`, `.then()` вместо `await`, классовые компоненты. Обучающих данных той эпохи больше (статья 3.1, причина 3).
7. **Несуществующий метод библиотеки.** Галлюцинация (статья 4.8). В TypeScript ловится компилятором — если у библиотеки есть типы.

ЧЕК-ЛИСТ JS И TS

К общему чек-листу приёмки (статья 4.7) добавляются пять шагов.

1. Проверка типов

```
npx tsc --noEmit
```

Самый выгодный шаг. Ловит пять классов ошибок из статьи 9.5.

2. Поиск заглушек в диффе

```
git diff | grep -nE "\bany\b|\bas [A-Z]|ts-ignore|ts-expect-error"
```

Одна команда показывает все места, где проверка отключена. Каждое должно быть объяснимо.

3. Линтер

```
npx eslint .
```

Ловит то, что типы не видят: `==`, неиспользуемые переменные, забытые `await`, `forEach` с `async` (при подходящих правилах).

4. Запуск

```
npx tsx app.ts
```

Тип не гарантирует, что данные снаружи придут нужной формы (статья 9.5). Запускать обязательно.

5. Граница

Посмотреть, откуда в код попадают внешние данные: `fetch`, `JSON.parse`, `localStorage`, формы. Везде ли есть проверка во время выполнения, а не только `as`.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Проверь свой код по списку:
- есть ли any, as или ts-ignore и почему
- включён ли strict и проходит ли tsc --noEmit
- есть ли == вместо === и || там, где нужен ??
- есть ли forEach с async внутри
- Promise.all там, где нужен частичный результат
- где данные приходят извне и проверяются ли они

Что из этого есть? Если ничего — так и скажи.
```

Отдельно — просьба, которая экономит целый круг переписок:

ЗАПРОС

```
Исправь ошибки типов по существу. Не используй any, as
и ts-ignore. Если ошибку нельзя исправить без них —
объясни, почему, и не исправляй.
```

Последнее предложение важно: оно разрешает ИИ сказать «не получается» вместо того, чтобы молча заглушить (статьи 3.8, 4.8).

ШПАРГАЛКА

- Три заглушки: `any`, `as`, `ts-ignore`. Появились после «исправь типы» — ошибки спрятаны, а не исправлены.
- `unknown` — честная замена `any`.
- Чек-лист: `tsc --noEmit` · поиск заглушек в диффе · линтер · запуск · граница.
- `git diff | grep -E "\bany\b|\bas [A-Z]|ts-ignore"` — одна команда на весь дифф.
- Типы не проверяют внешние данные: запуск обязателен.
- Проси исправлять без заглушек и разрешай сказать «не получается».

10 React на TypeScript

Самый распространённый способ делать интерфейсы. Здесь сходится всё предыдущее: разметка из главы 8, типы из главы 9 и чтение кода из главы 2 — только теперь они работают вместе.

10.1 Анатомия React-проекта

что откуда берётся и почему файлов так много

ЧТО ЭТО

React-проект выглядит пугающе: два десятка файлов в корне, папка `node_modules` на сотни мегабайт, конфиги инструментов, о которых ты не слышал. При этом твоего кода — одна папка.

Статья раскладывает это по полкам, чтобы в незнакомом проекте не искать точку входа полчаса.

КАК ВЫГЛЯДИТ В КОДЕ

```
weather-app/
├── node_modules/        библиотеки — не читать, не коммитить
├── public/              файлы, которые отдаются как есть
├── src/                 ← твой код, только он
│   ├── main.tsx         точка входа: монтирует приложение
│   ├── App.tsx          корневой компонент
│   ├── components/      компоненты
│   │   └── CityCard.tsx
│   ├── hooks/           свои хуки
│   ├── api/             запросы к серверу
│   └── types.ts         общие типы
├── index.html           страница, в которую всё вставляется
├── package.json         зависимости и команды
├── package-lock.json    точные версии — коммитить
├── tsconfig.json        настройки TypeScript (статья 9.3)
├── vite.config.ts       настройки сборщика
└── .env                 секреты — в .gitignore (статья 5.7)
```

Читать надо только `src/`. Всё остальное — обвязка.

Точка входа

```tsx
import { createRoot } from "react-dom/client";
import App from "./App";

createRoot(document.getElementById("root")!).render(<App />);
```

Три строки, и в них вся связь между HTML и React: взять элемент `#root` из `index.html` и нарисовать в нём приложение. Дальше весь интерфейс рисует React, а `index.html` больше не меняется.

Восклицательный знак после `getElementById` — это `as`-подобное утверждение «элемент точно есть» (статья 9.5). Здесь оно оправдано: если `#root` нет, приложение всё равно не запустится.

Что такое сборщик

Браузер не понимает ни JSX, ни TypeScript, ни импорты из `node_modules`. Сборщик (чаще всего **Vite**) превращает всё это в обычный JavaScript.

Отсюда две команды, которые встречаются в каждом проекте:

```bash
npm run dev     # режим разработки: правка видна сразу
npm run build   # сборка в папку dist/ для выкладки
```

Что именно они делают — написано в `package.json`, в разделе `scripts`. Это первое место, куда стоит заглянуть в незнакомом проекте: там список всего, что с ним можно делать.

`package.json`

```json
{
  "scripts": {
    "dev": "vite",
    "build": "tsc --noEmit && vite build",
    "lint": "eslint ."
  },
  "dependencies": {
    "react": "^19.3.1",
    "react-dom": "^19.3.1"
  },
  "devDependencies": {
    "typescript": "^5.9.0",
    "vite": "^7.1.0"
  }
}
```

Раздел	Что внутри
`scripts`	команды проекта — читать первыми
`dependencies`	нужно приложению при работе

`devDependencies`	нужно только при разработке

Про `^`: знак означает «эта версия или новее, но в пределах мажорной». Поэтому у тебя и на сервере могут встать разные версии — и именно от этого защищает `package-lock.json`, который фиксирует точные. Его коммитят всегда (статья 5.6).

> **СОВЕТ**
> Обрати внимание на строку `"build": "tsc --noEmit && vite build"`. Проверка типов встроена в сборку: если типы не сходятся, сборка не пройдёт. Это тот случай, когда проверку из статьи 9.3 не надо помнить — она выполняется сама.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Покажи содержимое scripts из package.json и объясни,
что делает каждая команда. Одной строкой на команду.
```

ЗАПРОС
```
Я открыл этот проект впервые. С каких трёх файлов в src
начать чтение, чтобы понять, как он устроен?
```

ЗАПРОС
```
Добавь компонент CityCard в src/components/.
Структуру проекта не меняй, новых зависимостей не добавляй.
```

НА ЧТО СМОТРЕТЬ

`node_modules` в git. Сотни мегабайт чужого кода в репозитории. Проверяется наличием строки в `.gitignore`.

Нет `package-lock.json`. Версии не зафиксированы, и у тебя с сервером встанет разное.

Всё в одном `App.tsx`. Файл на тысячу строк вместо компонентов. Работать будет, менять — тяжело, и ИИ не сможет править часть, не задевая остальное.

Секреты в коде фронтенда. Важное отличие от бэкенда: **всё, что попало в сборку, видно пользователю.** Переменная `VITE_API_KEY` окажется в коде страницы. Настоящие ключи держат на сервере, а фронтенд обращается к своему бэкенду (статья 7.8).

Смесь `.js` и `.tsx`. Переход на TypeScript на полпути: в `.js` проверок нет.

ШПАРГАЛКА

- Читать только `src/`, остальное — обвязка.
- Точка входа: `main.tsx`, монтирует приложение в `#root`.
- `package.json` → `scripts`: список всего, что можно делать с проектом.
- `^` в версиях разрешает обновления; точные версии — в `package-lock.json`.
- `tsc --noEmit` внутри `build` — проверка типов встроена в сборку.
- Секреты на фронтенде невозможны: всё из сборки видно пользователю.

10.2 Компоненты и JSX

разметка внутри кода

ЧТО ЭТО

Компонент — функция, которая возвращает разметку. Это вся идея React: вместо того чтобы менять страницу командами, ты описываешь, **как она должна выглядеть** при данных данных, а React сам приводит её к этому виду.

JSX — синтаксис, который позволяет писать разметку прямо в коде. Выглядит как HTML, но это JavaScript.

КАК ВЫГЛЯДИТ В КОДЕ

```
interface CityCardProps {
  name: string;
  temp: number | null;
  onRemove: (name: string) => void;
}

export function CityCard({ name, temp, onRemove }: CityCardProps) {
  return (
    <article className="card">
      <h2>{name}</h2>
      <p>{temp === null ? "нет данных" : `${temp > 0 ? "+" :
""}${temp}°`}</p>
      <button type="button" onClick={() => onRemove(name)}>Удалить</
button>
    </article>
  );
}
```

Что из этого получается — настоящий вывод:

```
<article class="card"><h2>Oslo</h2><p>+4°</p>
  <button type="button">Удалить</button></article>
<article class="card"><h2>Tromsø</h2><p>нет данных</p>
  <button type="button">Удалить</button></article>
```

И как это выглядит на экране:

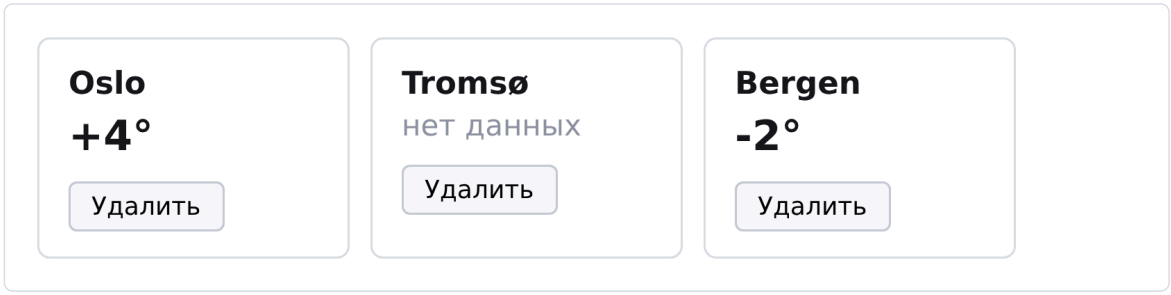


Рис. 10.1 · Три вызова одного компонента с разными данными. Снимок настоящий.

Как это читать

Компонент — функция с заглавной буквы. Это не стиль, а требование: с маленькой буквы React сочтёт его обычным тегом.

`{ }` внутри разметки — это выход в JavaScript. Всё, что в фигурных скобках, вычисляется: `{name}`, `{temp === null ? ... : ...}`, `{() => onRemove(name)}`.

`className` вместо `class`. `class` — зарезервированное слово в JavaScript. Так же `htmlFor` вместо `for`.

Возвращается один элемент. Два корневых рядом — ошибка. Если нужно, их оборачивают в пустой тег `<>...</>`.

Деструктуризация в аргументе. `({ name, temp, onRemove }: CityCardProps)` — это «взять из объекта пропсов три поля». Эквивалент `(props: CityCardProps)` и дальше `props.name`.

Списки и `key`

```
<div className="list">
  {cities.map(c => <CityCard key={c.name} {...c}
onRemove={handleRemove} />)}
</div>
```

`map` вместо цикла. В JSX нельзя написать `for` — вместо этого массив данных превращают в массив элементов.

`key` обязателен. Это не формальность: по нему React понимает, какой элемент какой, когда список меняется. Без `key` при удалении из середины списка данные «съезжают» на соседей. React предупреждает об этом в консоли (статья 8.4).

`key` должен быть стабильным. Идентификатор записи — хорошо. Индекс массива — плохо: при сортировке и удалении он меняется, и смысл теряется.

`{...c}` — расстановка. Передать все поля объекта как пропсы. Коротко, но прячет, что именно передаётся: читающему приходится идти смотреть, из чего состоит `c`.

> **ЗАПОМНИ**
> React не меняет страницу — он сравнивает, какой она должна быть, с тем, какая она есть, и приводит в соответствие. Поэтому в компоненте не бывает команд «добавь строку», только описание результата.

Условия в разметке

```
{temp === null ? <p>нет данных</p> : <p>{temp}°</p>}   // если-иначе
{isLoading && <Spinner />}                              // только если
{items.length > 0 && <List items={items} />}            // осторожно!
```

Третья строка — распространённая ловушка. Если `items.length` равен нулю, на экране появится **«0»**: ноль не отображается как «ничего», а выводится как число. Безопасно — `items.length > 0 &&` (как здесь) или тернарный оператор.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> Сделай компонент CityCard: принимает name (строка),
> temp (число или null) и onRemove (функция от названия).
> Если temp null — показывать «нет данных».
> Типы пропсов отдельным интерфейсом, экспортировать его.

> **ЗАПРОС**
> Этот список рисуется через map без key. Добавь key
> по идентификатору записи, не по индексу, и объясни почему.

> **ЗАПРОС**
> Разбей этот компонент на три поменьше. Логику не меняй,
> только структуру. Покажи, какие пропсы у каждого.

НА ЧТО СМОТРЕТЬ

Нет `key` или `key={index}`. Самая частая ошибка в сгенерированных списках.

`{items.length && ...}`. Покажет «0» вместо ничего.

Компонент на двести строк. Разбивается на части — иначе ни прочитать, ни переиспользовать.

Логика внутри разметки. Три вложенных тернарных оператора в JSX читаются тяжело. Вычисления выносятся выше, в переменные.

`{...props}` **везде.** Удобно писать, трудно читать: непонятно, что передаётся.

Разметка без семантики. Всё те же `div` вместо `button` и `article` (статья 8.2) — React этого не исправляет.

ШПАРГАЛКА

- Компонент — функция с заглавной буквы, возвращает разметку.
- `{ }` в JSX — выход в JavaScript; `className` вместо `class`.
- Возвращается один корневой элемент, иначе оборачивай в `<>…</>`.
- Списки через `map`, `key` обязателен и должен быть стабильным.
- `key={index}` ломается при сортировке и удалении.
- `{x.length && ...}` выведет «0». Пиши `x.length > 0 &&`.

10.3 Пропсы и состояние

откуда берутся данные и почему счётчик считает не так

ЧТО ЭТО

Два источника данных в компоненте, и путать их нельзя:

- **пропсы** — пришли снаружи, от родителя. Менять их нельзя.
- **состояние** — живёт внутри компонента. Меняется, и при каждом изменении компонент рисуется заново.

Типы здесь дают больше всего пользы во всей главе: пропсы — это граница между компонентами, и TypeScript следит за ней строго.

КАК ВЫГЛЯДИТ В КОДЕ

```
const [count, setCount] = useState(0);
```

`useState` возвращает пару: текущее значение и функцию, которая его меняет. Квадратные скобки — распаковка массива из двух элементов.

Присваивать напрямую нельзя. `count = 5` ничего не перерисует: React узнаёт об изменении только через `setCount`.

Что ловят типы

Вот настоящие сообщения компилятора на четыре типичные ошибки:

Забыт обязательный пропс:

```
<CityCard name="Oslo" temp={4} />
```

```
error TS2741: Property 'onRemove' is missing in type
'{ name: string; temp: number; }' but required in type 'CityCardProps'.
```

Не тот тип:

```
<CityCard name="Oslo" temp="4" onRemove={() => {}} />
```

```
error TS2322: Type 'string' is not assignable to type 'number'.
```

Не та сигнатура обработчика:

```
<CityCard name="Oslo" temp={4} onRemove={(id: number) => {}} />
```

```
error TS2322: Type '(id: number) => void' is not assignable
to type '(name: string) => void'.
```

Забыта проверка на пустоту:

```
function T({ temp }: { temp: number | null }) { return
<p>{temp.toFixed(1)}</p>; }
```

```
error TS18047: 'temp' is possibly 'null'.
```

Все четыре — классические ошибки ИИ, и все четыре не доходят до запуска. Особенно третья: ИИ часто помнит обработчик с другой сигнатурой и подставляет его молча.

Ловушка, которую надо увидеть

Три кнопки, одна и та же задача — прибавить единицу три раза:

```
<button onClick={() => { setCount(count + 1); setCount(count + 1);
setCount(count + 1); }}>
  три раза подряд
</button>

<button onClick={() => { setCount(c => c + 1); setCount(c => c + 1);
setCount(c => c + 1); }}>
  три раза через функцию
</button>
```

Измерено в браузере:

```
старт:                          счёт: 0 | отрисовок: 1
после одного +1:                счёт: 1 | отрисовок: 2
после «три раза подряд»:        счёт: 2 | отрисовок: 3
после «три раза через функцию»: счёт: 5 | отрисовок: 4
```

Три вызова подряд дали **+1**, а не +3. Три вызова через функцию — **+3**.

Причина: внутри обработчика `count` — это значение **на момент отрисовки**, и оно не меняется от вызовов `setCount`. Три раза подряд выполняется `setCount(0 + 1)`. Запись `setCount(c => c + 1)` просит React взять актуальное значение сам.

Обрати внимание и на счётчик отрисовок: три вызова дали **одну** перерисовку. React собирает изменения и применяет разом — это называется пакетированием.

> **ЗАПОМНИ**
> Если новое значение зависит от старого — всегда функция: `setCount(c => c + 1)`. Это правило закрывает целый класс ошибок, которые проявляются только при быстрых повторных действиях и потому не ловятся при обычной проверке.

Где должно жить состояние

Правило: **в ближайшем общем родителе тех компонентов, которым оно нужно.**

Если состояние нужно двум соседним компонентам, его поднимают к родителю, а вниз передают пропсами — само значение и функцию изменения:

```
<CityCard name={c.name} temp={c.temp} onRemove={handleRemove} />
```

Это и есть ответ на вопрос «почему так много пропсов»: данные идут сверху вниз, события — снизу вверх.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> Состояние списка городов сейчас в CityCard. Подними его в App, вниз передавай значение и обработчик. Типы пропсов опиши интерфейсом.

> **ЗАПРОС**
> В этом обработчике несколько setState подряд, зависящих от предыдущего значения. Перепиши через функциональную форму setX(prev => ...).

> **ЗАПРОС**
> Опиши пропсы этого компонента интерфейсом: что обязательно, что нет, какие типы у обработчиков. Реализацию не трогай.

НА ЧТО СМОТРЕТЬ

Пропс изменяют внутри компонента. `props.items.push(...)` — пропсы менять нельзя, это данные родителя.

`setState` **подряд с зависимостью от старого значения.** Разобрано выше.

Состояние дублируется. Одно и то же лежит и в родителе, и в ребёнке — рассинхронизируется гарантированно.

Состояние там, где хватит обычной переменной. Если значение не влияет на то, что видно, `useState` не нужен — он вызывает лишние перерисовки.

Пропсы без типов или с `any`**.** Теряется вся польза из статьи 9.5.

Десять пропсов у одного компонента. Признак, что компонент делает слишком много (статья 2.5 про число аргументов).

ШПАРГАЛКА

- Пропсы приходят сверху и неизменяемы, состояние живёт внутри.
- Присваивание напрямую ничего не перерисует — только через `setX`.
- Новое значение зависит от старого — функциональная форма `setX(prev => ...)`.
- Три `setCount(count + 1)` подряд дают +1, а не +3.
- Несколько изменений дают одну перерисовку: React пакетирует.
- Общее состояние — в ближайшем общем родителе.

10.4 Эффекты

где React ломает новичков чаще всего

ЧТО ЭТО

`useEffect` — способ сделать что-то, что не является отрисовкой: запросить данные, подписаться на событие, поставить таймер. Выполняется **после** того, как компонент нарисовался.

Это самое сложное место React и самое частое место ошибок ИИ. Одна забытая строка превращает приложение в бесконечный цикл запросов.

КАК ВЫГЛЯДИТ В КОДЕ

```
useEffect(() => {
  // что сделать
  return () => { /* как убрать за собой */ };
}, [зависимости]);
```

Три части, и каждая важна:

Тело — что выполнить после отрисовки.

Возвращаемая функция — уборка: отписаться, отменить таймер, прервать запрос. Выполняется перед следующим запуском эффекта и при удалении компонента.

Массив зависимостей — когда повторять. Это и есть источник всех бед.

Массив зависимостей решает всё

Запись	Когда выполняется
нет массива	**после каждой отрисовки**
`[]`	один раз, при появлении компонента
`[userId]`	при появлении и каждый раз, когда `userId` изменился

Разница между первой и второй строкой — измеренная:

```
useEffect(() => {              // массива нет
  setData([...data, "x"]);
});
```

```
useEffect(() => {              // пустой массив
  setData(["x"]);
}, []);
```

```
без массива зависимостей:  эффект выполнен раз: 49
с пустым массивом [] :     эффект выполнен раз: 1
```

Сорок девять раз — и это только потому, что я поставил ограничитель. Без него цикл был бы бесконечным: эффект меняет состояние → компонент перерисовывается → эффект выполняется снова.

> **ВНИМАНИЕ**
> Эффект без массива зависимостей, который меняет состояние, — это бесконечный цикл. Если он ходит в сеть, приложение начнёт долбить сервер десятки раз в секунду. Проверяется одним взглядом: есть ли второй аргумент у `useEffect`.

Уборка

```
useEffect(() => {
  const controller = new AbortController();
  fetch(`/api/subs/${userId}`, { signal: controller.signal })
    .then(r => r.json())
    .then(setData)
    .catch(e => { if (e.name !== "AbortError") console.error(e); });
  return () => controller.abort();
}, [userId]);
```

Зачем нужна возвращаемая функция: пользователь быстро переключил город — первый запрос ещё летит, пошёл второй. Без отмены ответы могут прийти не по порядку, и на экране окажутся данные от предыдущего города.

Это «состояние гонки», и ИИ забывает про него почти всегда: в сгенерированном коде есть `fetch`, но нет ни `AbortController`, ни возврата функции уборки.

Когда эффект не нужен

Главное заблуждение: что в `useEffect` надо класть всё подряд. На деле он нужен только для выхода наружу — сеть, таймеры, подписки, прямая работа с DOM.

Не нужен:

```
// плохо: лишнее состояние и лишняя перерисовка
const [total, setTotal] = useState(0);
useEffect(() => { setTotal(items.length); }, [items]);

// хорошо: просто вычислить
const total = items.length;
```

Если значение можно посчитать из того, что уже есть, — считай прямо при отрисовке. Состояние и эффект здесь только добавляют шаг и перерисовку.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
В этом useEffect нет массива зависимостей, а внутри меняется
состояние. Исправь и объясни, что было не так.
```

ЗАПРОС
```
Добавь отмену запроса через AbortController и возврат функции
уборки. Ошибку AbortError не показывай пользователю —
это не ошибка, а отмена.
```

ЗАПРОС

```
Нужны ли здесь useState и useEffect? Можно ли это значение
просто вычислить при отрисовке?
```

Последний вопрос стоит задавать всегда, когда видишь `useEffect`, который только меняет состояние по другим данным.

НА ЧТО СМОТРЕТЬ

Нет массива зависимостей. Проверяется мгновенно, ломает всё.

Массив есть, но неполный. Эффект использует переменную, которой нет в зависимостях, — будет работать со старым значением. Линтер с правилом `react-hooks/exhaustive-deps` ловит это автоматически.

Нет уборки у подписок и таймеров. `setInterval` без `clearInterval` продолжит работать после удаления компонента.

`fetch` **без отмены.** Состояние гонки при быстром переключении.

Эффект вместо вычисления. Лишнее состояние, лишняя перерисовка.

Запрос в эффекте без обработки ошибки. Сервер ответил `500` — на экране бесконечная загрузка.

ШПАРГАЛКА

- `useEffect` — для выхода наружу: сеть, таймеры, подписки.
- Массив зависимостей: нет — каждый раз, `[]` — один раз, `[x]` — при смене `x`.
- Нет массива плюс изменение состояния = бесконечный цикл.
- Возвращай функцию уборки для подписок, таймеров и запросов.
- `fetch` без `AbortController` даёт состояние гонки.
- Значение считается из имеющихся данных — эффект не нужен.

10.5 Постановка задач по интерфейсу

как описать словами то, что видишь глазами

ЧТО ЭТО

Интерфейс — единственное в книге, что ты задумываешь **визуально**, а описывать должен текстом. Отсюда особая трудность: в голове картинка, а ИИ получает слова, и расхождение обнаруживается только когда посмотришь на результат.

Статья — про то, как сократить это расхождение.

КАК СКАЗАТЬ ИИ

Четыре части, как везде

Те же части, что в статье 3.2, но наполнение своё:

Часть	Для интерфейса это
Контекст	какие данные есть, откуда приходят, что уже написано
Цель	что пользователь видит и что может сделать
Ограничения	что не трогать, какие компоненты переиспользовать
Критерий	проверяемые состояния экрана

Описывай состояния, а не внешний вид

Главный приём главы. Любой экран с данными имеет минимум четыре состояния, и ИИ по умолчанию делает только одно — удачное.

ТАК ЛУЧШЕ

```
Экран списка подписок. Четыре состояния:

1. Загрузка — скелетон на три карточки
2. Пусто — текст «Пока нет подписок» и кнопка «Добавить»
3. Данные — список карточек
4. Ошибка — текст ошибки и кнопка «Повторить»

Все четыре должны быть реализованы, не только третье.
```

ТАК НЕ НАДО

```
Сделай список подписок
```

Второй запрос даст только третье состояние. Загрузка, пустота и ошибка появятся потом — в виде багов у пользователей.

Называй структуру, а не пиксели

ТАК НЕ НАДО

```
Карточка шириной 320 пикселей, отступ сверху 18, заголовок 17px
```

ТАК ЛУЧШЕ

```
Карточка: название города, под ним температура крупно,
внизу кнопка «Удалить». Карточки в ряд, на узком экране —
в столбец. Отступы из существующих переменных, новых
значений не вводи.
```

Пиксели — это магические числа (статья 8.3): подогнанные под один экран, они развалятся на другом.

Дай сигнатуру

Для React сигнатура — это интерфейс пропсов, и работает он так же, как в статьях 6.5 и 9.5:

ЗАПРОС

```
Компонент под такой интерфейс:

interface SubsListProps {
  subs: City[] | null;        // null = ещё загружается
  error: string | null;
  onRemove: (name: string) => void;
  onRetry: () => void;
}

Из типов видно все четыре состояния — реализуй их.
```

Это самая компактная форма задания: типы сами описывают, что бывает.

> **СОВЕТ**
> Если интерфейс уже существует где-то — в макете, на другом экране, на скриншоте, — покажи его и скажи «сделай так же». Описывать словами то, что можно показать, — лишняя работа и лишний источник расхождений.

Проси по частям

Целый экран одним запросом — это пять задач в одной (статья 3.16). Лучше по шагам:

1. компонент карточки с типами пропсов;
2. список с состояниями загрузки и пустоты;
3. обработка ошибки и повтор;
4. адаптив.

После каждого шага — посмотреть. Это дешевле, чем разбирать готовый экран, в котором что-то не так.

НА ЧТО СМОТРЕТЬ

Сделано только удачное состояние. Проверяется просто: отключи сеть и посмотри, что на экране.

Придуманы отступы и размеры. Если в проекте есть переменные или система отступов, новые значения — это разнобой.

Компонент не переиспользует существующие. ИИ охотно пишет свою кнопку вместо той, что уже есть в проекте. Это причина 2 из статьи 3.1 — он её просто не видел.

Подписи придуманы. «Sorry, something went wrong» вместо текста на русском и в стиле остального интерфейса.

Доступность забыта. Те же `div` вместо `button`, нет подписей у полей (статья 8.2). React этого не исправляет.

ШПАРГАЛКА

- Описывай **состояния**, а не внешний вид: загрузка, пусто, данные, ошибка.
- По умолчанию ИИ сделает только «данные».
- Структура вместо пикселей; отступы — из существующих переменных.
- Интерфейс пропсов — самая компактная форма задания.
- Есть макет или похожий экран — покажи, а не описывай.
- Проси по частям и смотри после каждой.

10.6 Ошибки ИИ в React и чек-лист

что проверять, кроме «открылось и работает»

ЧТО ЭТО

Завершение главы и всего блока про фронтенд. React добавляет к ошибкам из глав 8 и 9 свои — и почти все они **не видны при беглой проверке**: экран открылся, данные показались, кнопка нажимается.

ДЕСЯТЬ ОШИБОК ИИ В REACT

1. **Нет `key` или `key={index}`** в списке. При удалении из середины данные съезжают на соседей (статья 10.2).
2. **`useEffect` без массива зависимостей**, меняющий состояние, — бесконечный цикл. Измерено: 49 выполнений вместо одного (статья 10.4).
3. **Неполный массив зависимостей.** Эффект работает со старым значением.
4. **Нет уборки.** `setInterval`, подписка или `fetch` без отмены переживают компонент.
5. **`setState` подряд с зависимостью от старого значения.** Три вызова дают +1 вместо +3 (статья 10.3).
6. **Состояние вместо вычисления.** Лишний `useState` + `useEffect` там, где хватит одной строки.
7. **Сделано только удачное состояние.** Нет загрузки, пустоты и ошибки (статья 10.5).
8. **`{items.length && ...}`** — на пустом списке выведет «0».

9. **Пропсы без типов или с `any`.** Теряется всё, что даёт TypeScript (статья 9.5).
10. **Секреты в коде фронтенда.** Всё, что попало в сборку, видно пользователю (статья 10.1).

ЧЕК-ЛИСТ REACT-КОДА

К чек-листу фронтенда (статья 8.5) добавляется шесть шагов.

1. Проверка типов

```
npx tsc --noEmit
```

Ловит пропсы: забытые, не того типа, с другой сигнатурой обработчика.

2. Линтер с правилами хуков

```
npx eslint .
```

Правило `react-hooks/exhaustive-deps` ловит неполные массивы зависимостей — то, что глазами почти не видно.

3. Консоль

F12. React пишет туда предупреждения о `key`, о состоянии после удаления компонента, о неверных пропсах. Красного и жёлтого быть не должно.

4. Сеть при открытии экрана

Вкладка «Сеть» (статья 8.4). Сколько запросов уходит при открытии? Один-два — нормально. Десятки и растущий счётчик — тот самый цикл из пункта 2.

5. Четыре состояния

- **Загрузка** — замедлить сеть в инструментах разработчика.
- **Пусто** — удалить все записи.
- **Ошибка** — отключить сеть и перезагрузить.
- **Данные** — обычный случай.

Первые три проверяют почти никогда, и именно там живут ошибки.

6. Быстрые повторные действия

Нажать кнопку пять раз подряд. Быстро переключить вкладки. Это ловит и `setState` из пункта 5, и состояние гонки из пункта 4.

ЗАПОМНИ

Главное отличие проверки React от проверки обычного кода: тут мало посмотреть на результат — надо посмотреть на **все состояния** и на **быстрые действия**. Ошибки живут не в том, что видно сразу.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Проверь свой React-код по списку:
- есть ли key и не по индексу ли
- у всех ли useEffect есть массив зависимостей и полный ли он
- есть ли уборка у подписок, таймеров и запросов
- есть ли setState подряд с зависимостью от старого значения
- реализованы ли состояния загрузки, пустоты и ошибки
- есть ли any в пропсах

Что из этого есть? Если ничего — так и скажи.
```

ЗАПРОС

```
Этот экран сделан только для случая «данные пришли».
Добавь загрузку, пустой список и ошибку с кнопкой «Повторить».
Существующую разметку не переписывай.
```

ШПАРГАЛКА

- Десять ошибок: `key`, зависимости, уборка, `setState` подряд, лишнее состояние, одно состояние экрана, `length &&`, `any` в пропсах, секреты в сборке.
- Чек-лист: `tsc` · линтер с правилами хуков · консоль · сеть · четыре состояния · быстрые нажатия.
- Правило `exhaustive-deps` ловит то, что глазами не видно.
- Десятки запросов при открытии экрана — бесконечный цикл в эффекте.
- Проверяй загрузку, пустоту и ошибку, а не только удачный случай.
- Нажми кнопку пять раз подряд — это ловит сразу два класса ошибок.

11 Flutter и Dart

Мобильные приложения. Единственная глава книги, примеры которой не прогонялись — и об этом сказано прямо в первой статье. Поэтому здесь упор на то, что не устаревает: как устроен проект, как читать дерево виджетов и где ИИ ошибается.

11.1 Анатомия Flutter-проекта

и честное предупреждение об этой главе

ЧТО ЭТО

Flutter — способ написать приложение один раз и собрать его под Android, iOS, веб и десктоп. Язык — Dart.

Прежде чем начать, одно предупреждение, которое касается всей главы.

> **ВНИМАНИЕ**
> **Примеры этой главы не прогонялись.** Во всех остальных главах книги каждый листинг был запущен, а числа — измерены. Здесь код показан для разбора, а не как проверенный рецепт: Flutter быстро меняет API от версии к версии, и листинг, запущенный на одной версии, через полгода может не собраться. Поэтому глава написана на уровне чтения: как устроен проект, как читать чужой код, где ошибается ИИ. Конкретный синтаксис и названия методов проверяй по документации — в том числе и за мной.

Это неприятно, но честнее, чем делать вид. Заодно это ровно та ситуация, о которой вся книга: **уверенный текст не равен проверенному тексту** (статья 1.6). Относись к этой главе так же, как к ответу ИИ по редкой библиотеке, — читай, но сверяй.

КАК ВЫГЛЯДИТ В КОДЕ

```
weather_app/
├── lib/                    ← твой код, только он
│   ├── main.dart           точка входа
│   ├── screens/            экраны
│   ├── widgets/            переиспользуемые куски интерфейса
│   ├── models/             описание данных
│   └── services/           сеть, база, работа с API
├── test/                   тесты
├── android/                платформенные файлы Android
├── ios/                    платформенные файлы iOS
├── web/                    файлы для сборки под веб
├── pubspec.yaml            зависимости и ресурсы
├── pubspec.lock            точные версии — коммитить
└── .dart_tool/             служебное, не трогать
```

Структура знакомая: **читать надо `lib/`**, всё остальное — обвязка. Ровно то же правило, что `src/` в React-проекте (статья 10.1).

Папки `android/` и `ios/` — настоящие нативные проекты. Туда заглядывают редко: когда нужны разрешения, иконки, подпись приложения. Это тема главы 12.

`pubspec.yaml`
Аналог `package.json`, но ещё и список ресурсов:

```yaml
name: weather_app
environment:
  sdk: ^3.5.0

dependencies:
  flutter:
    sdk: flutter
  http: ^1.2.0

dev_dependencies:
  flutter_test:
    sdk: flutter
  flutter_lints: ^4.0.0

flutter:
  uses-material-design: true
  assets:
    - assets/icons/
```

Два отличия от других экосистем, о которых стоит знать:

Ресурсы объявляются явно. Картинку положить в папку мало — её надо прописать в `assets`. Забыл — приложение не найдёт файл во время работы, и ошибка будет не при сборке, а при открытии экрана. ИИ забывает про это регулярно.

Отступы значимы. YAML, как и Python, чувствителен к пробелам, и неверный отступ ломает файл целиком.

Команды

```
flutter pub get          # поставить зависимости
flutter run              # запустить на устройстве или эмуляторе
flutter analyze          # статический анализ — аналог tsc и mypy
flutter test             # тесты
flutter build apk        # сборка (глава 12)
```

`flutter analyze` — самая важная из них для работы с ИИ. Это тот же рубеж, что `mypy --strict` в Python (статья 6.3) и `tsc --noEmit` в TypeScript (9.3): проверка до запуска. В Dart типы обязательны, поэтому анализатор ловит много.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Покажи структуру lib/ этого проекта: что в каждой папке,
по строке. android/, ios/ и .dart_tool не показывай.
```

ЗАПРОС
```
Я добавил картинку в assets/icons/. Что нужно прописать
в pubspec.yaml, чтобы она работала? Покажи точный отступ.
```

ЗАПРОС
```
Версия Flutter у меня 3.38, Dart 3.10. Пиши код только
под них и скажи, если что-то из предложенного появилось
в более поздних версиях.
```

Последний запрос особенно уместен именно здесь: Flutter меняет API между мажорными версиями чаще, чем большинство фреймворков, а обучающих данных для старых версий больше (статья 3.1, причина 3).

НА ЧТО СМОТРЕТЬ

Код для старых версий. Самый частый признак — устаревшие имена виджетов и параметров. Если `flutter analyze` ругается на незнакомые аргументы — скорее всего, дело в версии.

Ресурс не объявлен в `pubspec.yaml`. Ошибка появится при открытии экрана, а не при сборке.

Нет `pubspec.lock` в репозитории. Версии разъедутся (статья 5.6).

Всё в `main.dart`. Файл на тысячу строк вместо экранов и виджетов.

Правки в `android/` и `ios/` без объяснения. Эти папки частично генерируются; изменения там легко потерять при обновлении инструментов.

ШПАРГАЛКА

- Примеры этой главы не прогонялись — сверяй с документацией.
- Читать только `lib/` `android/` и `ios/` — для сборки и разрешений.
- `pubspec.yaml` — зависимости **и** список ресурсов; отступы значимы.
- Картинку мало положить в папку — её надо объявить в `assets`.
- `flutter analyze` — рубеж проверки до запуска, как `mypy` и `tsc`.
- Flutter часто ломает совместимость между версиями: называй свою.

11.2 Dart за полчаса

что надо знать, чтобы читать

ЧТО ЭТО

Dart похож одновременно на Java, JavaScript и TypeScript. Если ты прочитал главу 9, половина уже знакома: типы, `async/await`, классы.

В статье — то, что отличается и встречается в каждом файле Flutter-проекта.

КАК ВЫГЛЯДИТ В КОДЕ

```
class City {
  final String name;
  final double? temp;

  const City({required this.name, this.temp});

  String get report => temp == null
      ? '$name: нет данных'
      : '$name: ${temp! > 0 ? '+' : ''}$temp°';
}

void main() {
  final cities = <City>[
    const City(name: 'Oslo', temp: 4),
    const City(name: 'Tromsø'),
  ];

  for (final c in cities) {
    print(c.report);
  }
}
```

Что здесь важно

`final` вместо `const` и `let`. `final` — присвоить один раз. `const` — значение известно на этапе компиляции. Во Flutter `const` встречается постоянно: он позволяет не пересоздавать виджеты при перерисовке.

`double? temp` — тип с вопросом, то же, что `number | null` в TypeScript. Dart строг к пустоте так же, как TypeScript со `strict`: обратиться к `temp` напрямую нельзя, пока не проверил.

`temp!` — утверждение «тут точно не пусто». Прямой аналог `as` из статьи 9.5 и такой же источник проблем: это обещание компилятору, а не проверка. Если ошибся, приложение упадёт с `Null check operator used on a null value`.

`required` — обязательный именованный параметр. Без него аргумент можно не передать.

`$name` и `${выражение}` — подстановка в строку, как f-строки в Python.

`get report => ...` — вычисляемое свойство: обращаются как к полю, без скобок.

Именованные параметры

Главное синтаксическое отличие, к которому надо привыкнуть:

```
City(name: 'Oslo', temp: 4)          // именованные, в фигурных скобках
```

Во Flutter почти все параметры именованные, и это объясняет, почему код выглядит таким многословным:

```
Padding(
  padding: const EdgeInsets.all(16),
  child: Text('Привет'),
)
```

Плюс в том, что при чтении всегда видно, что означает каждое значение.

Остальное коротко

Запись	Что значит
`late`	«значение будет, но позже» — обещание, проверяется при работе
`?.`	безопасный доступ, как в JS (статья 9.1)
`??`	значение по умолчанию
`..`	каскад: несколько операций с одним объектом

`=>`	короткая форма функции из одного выражения
`List<City>`	список городов; `<>` — как дженерики (статья 9.4)

`late` стоит упомянуть отдельно. Это способ сказать «я присвою это позже». Если не присвоил, а обратился — ошибка во время работы. ИИ использует `late`, чтобы обойти требование инициализации, — и это та же заглушка, что `as` и `!`.

> **ЗАПОМНИ**
> Три записи в Dart отключают проверку пустоты: `!`, `late` и приведение типа. Встретил их в сгенерированном коде — спроси, почему нельзя было обойтись проверкой. Это прямой аналог `any`, `as` и `ts-ignore` из статьи 9.6.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Объясни этот класс построчно: что значит final, что
required, что делает знак восклицания после temp.
```

ЗАПРОС
```
Здесь используется `late` и `!`. Перепиши без них,
через нормальные проверки на null. Если где-то без них
не обойтись — объясни почему, а не переписывай.
```

ЗАПРОС
```
Напиши модель данных для ответа API: вот пример JSON.
Поля, которых может не быть, делай nullable честно.
Добавь фабрику fromJson с проверками, а не приведением типов.
```

НА ЧТО СМОТРЕТЬ

`!` на каждом шагу. Приложение будет падать там, где данных не оказалось.

`late` вместо нормальной инициализации. Та же проблема, только позже.

Код для старых версий Flutter. Устаревшие имена виджетов и параметров.

`dynamic`. Полный аналог `any` из TypeScript: отключает проверку.

Разбор JSON без проверок. `json['main']['temp'] as double` падает, если сервер ответил ошибкой, — ровно так же, как в статьях 2.12 и 9.5.

ШПАРГАЛКА

- `final` — присвоить один раз, `const` — известно при компиляции.
- `double?` — может быть пустым; Dart не даст обратиться без проверки.
- `!`, `late` и приведение типа отключают проверку пустоты.
- Параметры во Flutter почти всегда именованные: `City(name: ..., temp: ...)`.
- `$name` и `${выражение}` — подстановка в строку.
- `dynamic` — это `any`, со всеми последствиями.

11.3 Виджеты

дерево, композиция и почему отступ — это тоже виджет

ЧТО ЭТО

Во Flutter **всё — виджет**. Кнопка, текст, отступ, выравнивание, даже сам экран. Интерфейс собирается не из свойств, как в CSS, а из вложенных друг в друга виджетов.

Это главное отличие от всего, что было в книге раньше, и главный источник недоумения при чтении: код выглядит как глубокая лестница скобок.

КАК ВЫГЛЯДИТ В КОДЕ

```
Scaffold(
  appBar: AppBar(title: const Text('Погода')),
  body: Padding(
    padding: const EdgeInsets.all(16),
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        Text(city.name, style: Theme.of(context).textTheme.titleLarge),
        const SizedBox(height: 8),
        Text(city.temp == null ? 'нет данных' : '${city.temp}°'),
        const Spacer(),
        ElevatedButton(
          onPressed: () => onRemove(city.name),
          child: const Text('Удалить'),
        ),
      ],
    ),
  ),
)
```

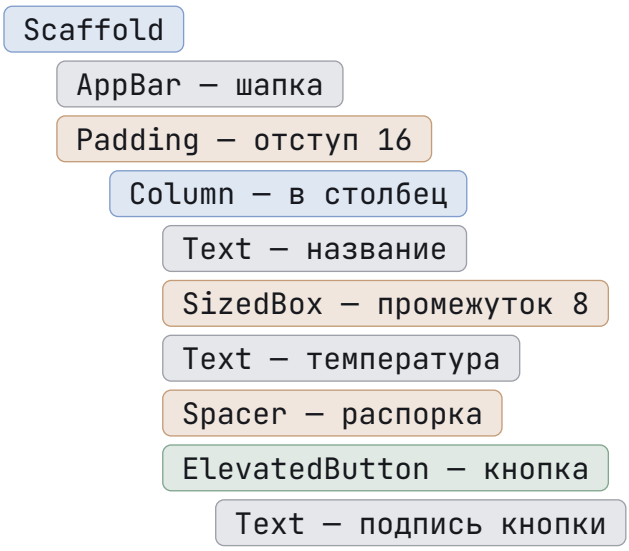


Рис. 11.1 · То же самое деревом. Отступ и промежуток — такие же узлы, как текст и кнопка.

Как это читать

Читай по вложенности, а не по строкам. Отступы в коде повторяют структуру дерева, поэтому самое полезное при чтении — смотреть на уровень отступа.

`child` **— один ребёнок,** `children` **— список.** По этим двум словам восстанавливается вся структура: где ветвление, где цепочка.

Отступы и промежутки — тоже виджеты. В этом главное отличие от CSS:

Нужно	Виджет
отступ вокруг	`Padding`
промежуток между	`SizedBox(height: 8)`
занять всё свободное	`Expanded`, `Spacer`
выровнять	`Align`, `Center`
в столбец / в ряд	`Column`, `Row`
поверх друг друга	`Stack`

Поэтому дерево такое глубокое: то, что в CSS было бы двумя строками свойств, здесь — два уровня вложенности.

`const` **перед виджетом.** Не формальность: такой виджет создаётся один раз и не пересоздаётся при перерисовке. ИИ его часто пропускает, анализатор на это ругается.

`Theme.of(context)` — взять значение из темы приложения: цвет, шрифт, размер. Это аналог CSS-переменных (статья 8.3): лучше, чем вписывать размеры числами.

> **СОВЕТ**
> Когда дерево становится нечитаемым — это сигнал вынести кусок в отдельный виджет, а не повод терпеть. Запрос «вынеси содержимое Column в отдельный виджет CityInfo» занимает строку и возвращает коду читаемость.

Row, Column и переполнение

Самая частая визуальная поломка во Flutter: содержимое не помещается, и на экране появляется жёлто-чёрная полоса с надписью о переполнении.

Причина почти всегда одна: в `Row` положили текст, который длиннее экрана. Лечится оборачиванием в `Expanded` или `Flexible` — они говорят «занимай место, которое есть, и не больше».

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Объясни это дерево виджетов: что за что отвечает.
По строке на уровень вложенности, без кода.
```

ЗАПРОС
```
Вынеси содержимое Column в отдельный виджет CityInfo
в lib/widgets/. Параметры передай через конструктор.
Поведение не меняй.
```

ЗАПРОС
```
При длинном названии города в этом Row случается overflow.
Исправь через Expanded, и объясни, почему именно так,
а не фиксированной шириной.
```

НА ЧТО СМОТРЕТЬ

Дерево на десять уровней в одном файле. Разбивается на виджеты.

Нет `const` там, где виджет не зависит от данных. Лишние пересоздания при каждой перерисовке.

Размеры числами вместо темы. `fontSize: 17` вместо `Theme.of(context)` — те же магические числа, что в статье 8.3.

`Container` ради одной вещи. Если нужен только отступ — `Padding`, только выравнивание — `Align`. `Container` умеет всё сразу и потому прячет намерение.

Логика внутри `build`. Запросы к сети и вычисления в методе построения: он вызывается при каждой перерисовке, то есть часто.

Фиксированные размеры под один экран. Телефоны разные; `MediaQuery` и `Expanded` существуют именно для этого.

ШПАРГАЛКА

- Во Flutter всё виджет, включая отступы и промежутки.
- `child` — один, `children` — список; по ним восстанавливается дерево.
- Читай по вложенности, а не по строкам.
- `const` перед виджетом экономит пересоздание при перерисовке.
- Размеры и цвета — из темы, а не числами.
- Жёлто-чёрная полоса — переполнение; лечится `Expanded`.

11.4 Состояние

stateless, stateful и что выбирать

ЧТО ЭТО

Виджеты бывают двух видов: те, что просто рисуют переданные данные, и те, что хранят что-то своё и перерисовываются при изменении.

Идея та же, что пропсы и состояние в React (статья 10.3), и правила принятия решений те же. Отличается только синтаксис — и он во Flutter многословнее.

КАК ВЫГЛЯДИТ В КОДЕ

Stateless — своих данных нет

```
class CityCard extends StatelessWidget {
  final String name;
  final double? temp;
  final void Function(String) onRemove;

  const CityCard({
    super.key,
    required this.name,
    this.temp,
    required this.onRemove,
  });

  @override
  Widget build(BuildContext context) {
    return Card(
      child: Column(
        children: [
          Text(name),
          Text(temp == null ? 'нет данных' : '$temp°'),
          TextButton(
            onPressed: () => onRemove(name),
            child: const Text('Удалить'),
          ),
        ],
      ),
    );
  }
}
```

Всё приходит снаружи через конструктор — это прямой аналог пропсов. `final` у полей не случаен: менять их нельзя, как и пропсы в React.

`super.key` — тот самый `key`, что в React (статья 10.2), и нужен он для того же: чтобы при изменении списка Flutter понимал, какой виджет какой.

Stateful — есть своё

```
class Counter extends StatefulWidget {
  const Counter({super.key});

  @override
  State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _count = 0;

  void _increment() {
    setState(() {
      _count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return TextButton(
      onPressed: _increment,
      child: Text('нажато: $_count'),
    );
  }
}
```

Два класса вместо одного — особенность Flutter, к которой надо привыкнуть: сам виджет и его состояние живут отдельно.

`setState` — единственный способ сообщить об изменении. Присваивание `_count++` без обёртки `setState` изменит значение, но **экран не перерисуется**. Это ровно та же ошибка, что присваивание вместо `setX` в React.

Подчёркивание в начале имени в Dart означает «только для этого файла». Поэтому состояние обычно `_CounterState`, а поля `_count`.

Жизненный цикл

```
@override
void initState() {
  super.initState();
  _loadData();        // выполняется один раз при появлении
}

@override
void dispose() {
  _controller.dispose();   // уборка: контроллеры, подписки, таймеры
  super.dispose();
}
```

Метод	Когда	Аналог в React
`initState`	один раз при появлении	`useEffect(..., [])`
`build`	при каждой перерисовке	тело компонента
`dispose`	при удалении	возврат функции из `useEffect`

`dispose` обязателен для всего, что надо закрывать: контроллеры полей ввода, прокрутки, анимации, подписки, таймеры. Забытый `dispose` — утечка, и анализатор о ней не всегда предупредит.

> **ЗАПОМНИ**
> Правило выбора то же, что в React: состояние живёт в ближайшем общем родителе тех виджетов, которым оно нужно. Если данные нужны двум соседним экранам — им не место ни в одном из них.

Когда своего состояния становится много

Передавать данные через конструкторы на пять уровней вниз неудобно, и для этого есть библиотеки управления состоянием: `provider`, `riverpod`, `bloc` и другие.

Выбор между ними — архитектурное решение, а не вопрос вкуса: оно определяет, как будет устроен весь проект. Принимать его стоит самому, а не оставлять ИИ (статья 1.1), и обязательно называть выбранное в запросах — иначе получишь код под ту библиотеку, которой в обучающих данных было больше.

КАК СКАЗАТЬ ИИ

> **ЗАПРОС**
> В проекте используется provider, версия 6. Состояние хранится в ChangeNotifier. Новых библиотек управления состоянием не добавляй.

ЗАПРОС

```
Этот виджет stateful, но своего состояния у него нет —
всё приходит через конструктор. Переделай в stateless.
```

ЗАПРОС

```
Проверь: у всех ли контроллеров в этом файле есть dispose.
Список того, что не закрывается.
```

НА ЧТО СМОТРЕТЬ

Stateful без состояния. Лишний класс и лишний код. Признак: в `State` нет полей, только `build`.

Присваивание без `setState`. Значение меняется, экран нет.

`setState` с тяжёлой работой внутри. Внутрь кладут только изменение полей; запросы и вычисления — до него.

Нет `dispose`. Контроллеры и подписки живут дольше виджета.

Запрос к сети в `build`. `build` вызывается часто; запрос должен быть в `initState` или в слое данных. Прямой аналог ошибки из статьи 10.4.

Библиотека состояния выбрана за тебя. В ответе появился `bloc`, хотя в проекте `provider`, — проект получит две системы сразу.

ШПАРГАЛКА

- Stateless — всё снаружи через конструктор, аналог пропсов.
- Stateful — два класса: виджет и его состояние.
- Менять поля только внутри `setState`, иначе экран не перерисуется.
- `initState` — один раз, `build` — часто, `dispose` — уборка.
- Контроллеры и подписки обязаны закрываться в `dispose`.
- Библиотеку управления состоянием выбирай сам и называй в запросе.

11.5 Постановка задач по мобильному интерфейсу

что надо назвать, кроме внешнего вида

ЧТО ЭТО

Приёмы из статьи 10.5 работают и здесь: описывай состояния, а не пиксели, давай сигнатуру, проси по частям.

Но у мобильного приложения есть то, чего нет у сайта, — и если об этом не сказать, ИИ не спросит.

КАК СКАЗАТЬ ИИ

Шесть вещей, специфичных для мобильного

1. Версии. Flutter ломает совместимость чаще большинства фреймворков:

```
ЗАПРОС
Flutter 3.38, Dart 3.10. Код только под эти версии.
Если предлагаешь что-то из более поздних — скажи.
```

2. Библиотека состояния. Названа — значит, проект не обрастёт второй (статья 11.4).

3. Размер экрана и ориентация. Телефоны разные, и «выглядит хорошо» на одном ничего не значит:

```
ЗАПРОС
Проверь вёрстку на узком экране (360 логических пикселей)
и при альбомной ориентации. Фиксированных высот не используй.
```

4. Что при отсутствии сети. У мобильного приложения сеть пропадает постоянно — это норма, а не исключение:

```
ЗАПРОС
Сеть может пропасть в любой момент. Что показываем:
при загрузке, при ошибке сети, при пустом ответе,
при успешных данных. Нужны все четыре.
```

5. Разрешения. Камера, геопозиция, уведомления, файлы — всё требует спросить пользователя, и отказ надо обработать:

```
ЗАПРОС
Нужна геопозиция. Обработай три случая: разрешили,
отказали, отказали навсегда. В последнем случае —
предложение открыть настройки.
```

Третий случай забывают всегда, а он самый неприятный: приложение бесконечно просит разрешение, которое система больше не покажет.

6. Платформенные различия. Android и iOS ведут себя по-разному: кнопка «назад», безопасные зоны у краёв экрана, вид диалогов.

```
ЗАПРОС
Под какие платформы собираем — Android и iOS.
Отметь места, где поведение будет отличаться.
```

Чего не стоит делать

Не проси «сделай красиво». Это антипаттерн из статьи 3.16, и во Flutter он особенно разрушителен: получишь дерево на десять уровней с подобранными на глаз размерами.

Не описывай пикселями. `Theme.of(context)` и готовые виджеты отступов существуют именно для этого (статья 11.3).

Не проси экран целиком. Четыре шага дешевле: модель данных → виджет карточки → экран со списком и состояниями → обработка ошибок.

> **СОВЕТ**
> Главный приём для мобильного интерфейса — скриншот. Если есть макет, похожий экран или даже фотография экрана другого приложения, покажи его и скажи «сделай похоже по структуре». Описание словами того, что можно показать, — это лишний источник расхождений.

Проверка, которую стоит заказывать сразу

> **ЗАПРОС**
> ```
> После того как напишешь: перечисли, что я должен проверить
> руками на устройстве. Включи узкий экран, отсутствие сети
> и отказ в разрешении.
> ```

Такой список — заготовка чек-листа из статьи 11.6, составленная под конкретную задачу.

НА ЧТО СМОТРЕТЬ

Сделан только удачный случай. Как и в React: есть «данные пришли», нет загрузки, ошибки и пустоты.

Нет обработки отказа в разрешении. Приложение зависает на запросе.

Фиксированные размеры. Развалится на другом телефоне.

Тексты не вынесены. Подписи прямо в виджетах — при добавлении второго языка придётся переписывать всё.

Платформенные различия не учтены. Работает на Android, ведёт себя странно на iOS.

ШПАРГАЛКА

- Называй версии Flutter и Dart — здесь это особенно важно.
- Называй библиотеку состояния, иначе получишь вторую.
- Проси проверить узкий экран и альбомную ориентацию.
- Сеть пропадает постоянно: нужны все четыре состояния экрана.
- Разрешения — три случая, и «отказали навсегда» забывают всегда.
- Есть макет или похожий экран — покажи, а не описывай.

11.6 Ошибки ИИ во Flutter и чек-лист

и как проверить то, чего я проверить не смог

ЧТО ЭТО

Завершение главы. Здесь же — то, чего нет в других главах: **список того, что ты должен проверить сам**, потому что я этого сделать не смог (статья 11.1).

ОШИБКИ ИИ ВО FLUTTER

1. **Код для старой версии.** Главная проблема этого стека. Устаревшие имена виджетов и параметров; `flutter analyze` обычно ругается.
2. `!`, `late`, `dynamic`. Три способа отключить проверку пустоты — прямой аналог `any`, `as` и `ts-ignore` из статьи 9.6.
3. **Присваивание без `setState`.** Значение меняется, экран нет.
4. **Нет `dispose`.** Контроллеры и подписки переживают виджет.
5. **Запрос к сети в `build`.** `build` вызывается часто.
6. **Stateful без состояния.** Лишний класс.
7. **Нет `const`** там, где виджет не зависит от данных. Лишние пересоздания.
8. **Ресурс не объявлен в `pubspec.yaml`.** Ошибка не при сборке, а при открытии экрана.
9. **Сделан только удачный случай.** Нет загрузки, ошибки, пустоты (статья 11.5).
10. **Отказ в разрешении не обработан.** Особенно «отказали навсегда».
11. **Добавлена вторая библиотека состояния.** Потому что в запросе не назвали первую.
12. **Фиксированные размеры.** Переполнение на узком экране.

ЧЕК-ЛИСТ FLUTTER-КОДА

1. Анализатор

```
flutter analyze
```

Первый и главный шаг — здесь он заменяет `mypy` и `tsc`. Ловит устаревшие API, забытые `const`, неиспользуемое, часть ошибок с пустотой.

2. Запуск

```
flutter run
```

Горячая перезагрузка показывает правку сразу, но **не все изменения подхватываются**: правки в `initState` и в объявлениях часто требуют полной перезагрузки. Если изменение «не применилось» — перезапусти целиком, прежде чем искать ошибку в коде.

3. Узкий экран и длинный текст
Запустить на маленьком устройстве или сузить окно. Вписать длинное название. Жёлто-чёрная полоса переполнения — самая частая находка.

4. Четыре состояния
Загрузка, пусто, ошибка, данные. Для ошибки — выключить сеть на устройстве.

5. Разрешения
Отказать в разрешении. Потом отказать навсегда. Приложение должно продолжать работать и объяснять, что делать.

6. Обе платформы
Если собираете под Android и iOS — проверить на обеих. Кнопка «назад», безопасные зоны, диалоги ведут себя по-разному.

КАК ПРОВЕРИТЬ ЭТУ ГЛАВУ

Примеры здесь не прогонялись, поэтому — как убедиться, что написанное верно:

ЗАПРОС

```
Вот фрагмент из справочника по Flutter: <фрагмент>.
У меня Flutter 3.38. Это актуально для моей версии?
Если что-то устарело или неточно — скажи, что именно,
со ссылкой на документацию.
```

И самый надёжный способ — просто запустить. Создай пустой проект и проверь то, что показалось сомнительным:

```
flutter create proba && cd proba && flutter run
```

ЗАПОМНИ

Это и есть главная мысль всей книги, применённая к самой книге: **уверенный текст не равен проверенному**. Глава написана аккуратно, но не прогнана — значит, к ней применяются те же правила, что к ответу ИИ по редкой библиотеке. Читай, пользуйся, сверяй.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Проверь этот Flutter-код по списку:
- актуален ли API для Flutter 3.38
- есть ли !, late или dynamic и почему
- все ли изменения состояния через setState
- есть ли dispose у контроллеров и подписок
- нет ли запросов в build
- есть ли const там, где виджет статичен
- реализованы ли загрузка, ошибка и пустота

Что из этого есть? Если ничего — так и скажи.
```

ШПАРГАЛКА

- Главная ошибка стека — код для старой версии. Называй свою.
- `!`, `late`, `dynamic` — заглушки, а не решения.
- Изменения только через `setState` контроллеры закрывать в `dispose`.
- `flutter analyze` — первый шаг проверки, заменяет `mypy` и `tsc`.
- «Не применилось» — сначала полная перезагрузка, потом поиск ошибки.
- Проверь: узкий экран, четыре состояния, отказ в разрешении, обе платформы.

12 Сборка и публикация

Последний шаг, на котором рушится больше всего проектов. Код работает на твоей машине — и не работает нигде больше. Здесь про то, чем сборка отличается от запуска, почему зелёная сборка не значит рабочую программу и что спросить у ИИ перед публикацией.

12.1 Что такое сборка и почему она ломается

пять причин, по которым «у меня работает»

что это

Сборка — это превращение твоих исходников в то, что запускается у других. Пока ты разрабатываешь, программу запускает инструмент разработчика: он читает твои файлы как есть, подставляет настройки с твоей машины, перезагружает изменённое на ходу. Это удобно и совсем не похоже на то, что получит пользователь.

При сборке происходит другое: код проверяется, переписывается, склеивается в несколько файлов, значения настроек подставляются внутрь, лишнее выбрасывается. Получается **артефакт** — один файл или папка, которые можно отдать: `.apk` для Android, папка `dist/` для сайта, `.whl` или контейнер для серверной части.

Разница между «запустилось у меня» и «работает артефакт» — самая частая и самая обидная поломка в проекте, потому что она случается в конце, когда всё уже казалось готовым.

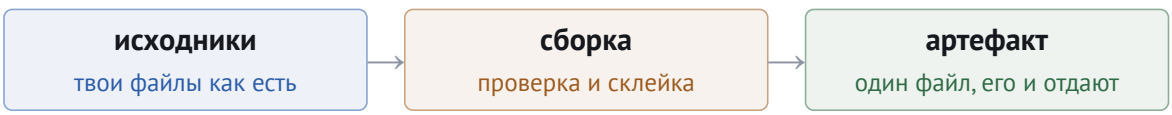


Рис. 12.1 · Три состояния одного проекта. Ошибки этой главы живут на стрелках, а не в коде.

> **ЗАПОМНИ**
> Запуск при разработке и сборка — это две разные программы, читающие одни и те же файлы по-разному. «Работает при разработке» не означает «собирается», а «собирается» не означает «работает».

КАК ВЫГЛЯДИТ В КОДЕ

Ниже пять причин поломок, и все пять проверены на живых проектах: показан настоящий вывод команд.

1. Сборка прошла, а код сломан

Вот файл с двумя явными ошибками типов — строка на месте числа и число на месте строки:

```
import { CityCard } from './components/CityCard'

export default function App() {
  const temp: number = "жарко"
  return <CityCard name={temp} />
}
```

Запускаем сборку напрямую:

```
$ npx vite build
vite v7.3.7 building client environment for production...
✓ 29 modules transformed.
dist/index.html                  0.39 kB │ gzip:  0.27 kB
dist/assets/index-DSlY0u7I.js  222.62 kB │ gzip: 69.35 kB
✓ built in 1.96s
```

Собралось. За две секунды, с галочкой. Теперь проверка типов, на том же файле:

```
$ npx tsc -b
src/App.tsx(4,9): error TS2322: Type 'string' is not assignable to type
'number'.
src/App.tsx(5,20): error TS2322: Type 'number' is not assignable to type
'string'.
```

Причина в том, что сборщик не проверяет типы — он их **выбрасывает**. Типы нужны редактору и проверяльщику, а в собранном файле их нет (статья 9.1). Поэтому в нормальном проекте команда сборки состоит из двух:

```
"build": "tsc -b && vite build"
```

Сначала проверка, потом склейка. И вот ровно здесь появляется ошибка, которую допускают и люди, и ИИ: сборка падает, её

«чинят», убрав из команды `tsc -b`. Сборка становится зелёной, проверка — выключенной.

> **ВНИМАНИЕ**
> Если ИИ предлагает править саму команду сборки, а не код, — остановись и прочитай, что именно убирают. Убрать проверку типов, отключить правило линтера, добавить `--force` или `--legacy-peer-deps` — это не исправление, а глушение сигнала (статья 2.14).

2. Настройки запекаются внутрь

Переменные окружения при сборке не читаются в момент работы программы — они **подставляются в текст** артефакта. Файл настроек:

```
VITE_API_URL=https://api.example.com
API_KEY=secret-abc-123
```

Код, который их читает:

```
const API = import.meta.env.VITE_API_URL
const KEY = import.meta.env.API_KEY
```

А вот что получилось в собранном файле — настоящий фрагмент из `dist/`:

```
var yh=oh();const mh="https://api.example.com",dh=void 0;
```

В одной строке сразу две разные беды.

Первая: адрес попал в артефакт текстом. Собранный файл отдаётся браузеру любого желающего, то есть всё, что в нём есть, — публично. Если бы на месте адреса был ключ от платного API, он лежал бы там же, открытым текстом.

Вторая: `API_KEY` превратился в `void 0`, то есть в «ничего». Сборщик подставляет только переменные с нужным префиксом (`VITE_` в этом примере), остальные молча становятся пустыми. Ни ошибки, ни предупреждения: приложение собирается, запускается и ведёт себя странно — например, стучится по адресу `undefined/api/cities`.

Проверить это можно руками, и это стоит сделать хотя бы раз в жизни: собрать проект и поискать в собранном файле свой секрет.

```
$ grep -c "secret-abc-123" dist/assets/*.js
0
$ grep -c "api.example.com" dist/assets/*.js
1
```

Правило простое: **то, что должно остаться тайной, не попадает в код, который уходит на устройство пользователя.** Ключи живут на сервере, а приложение обращается к своему серверу, а тот — к платному API.

3. Путь, который зависит от того, откуда запустили

Невинная строка:

```
with open("pogoda/cities.json", encoding="utf-8") as f:
```

Из корня проекта работает:

```
$ python3 -m pogoda.main
{'Oslo': 4, 'Tromso': -3}
```

Из любой другой папки — нет:

```
  File ".../pogoda/main.py", line 5, in load_cities
    with open("pogoda/cities.json", encoding="utf-8") as f:
         ~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'pogoda/
cities.json'
```

Относительный путь считается не от файла с кодом, а **от папки, из которой запустили программу**. При разработке это всегда корень проекта, поэтому ошибки не видно. На сервере программу запускает служба, и папка у неё своя.

Правильно — считать от расположения самого файла:

```
from pathlib import Path

DATA = Path(__file__).parent / "cities.json"

def load_cities():
    with DATA.open(encoding="utf-8") as f:
        return json.load(f)
```

После такой правки запуск из `/tmp` даёт то же, что из корня проекта. Это проверено: `{'Oslo': 4, 'Tromso': -3}`.

4. Файл не попал в артефакт

Тот же проект, уже с исправленными путями, собираем в пакет:

```
$ python3 -m build --wheel
Successfully built pogoda-0.1.0-py3-none-any.whl
```

Ни ошибки, ни предупреждения. Смотрим, что внутри:

```
pogoda/__init__.py
pogoda/main.py
pogoda-0.1.0.dist-info/METADATA
pogoda-0.1.0.dist-info/WHEEL
pogoda-0.1.0.dist-info/RECORD
```

`cities.json` **внутри нет.** Устанавливаем пакет и запускаем:

```
FileNotFoundError: [Errno 2] No such file or directory:
'/tmp/venv-pogoda/lib/python3.13/site-packages/pogoda/cities.json'
```

Сборщик собирает код, а всё остальное — картинки, шрифты, словари, шаблоны писем, файлы данных — **надо объявлять отдельно**. Три строки в настройках проекта:

```
[tool.setuptools.package-data]
pogoda = ["*.json"]
```

И в собранном пакете появляется нужное:

```
adding 'pogoda/__init__.py'
adding 'pogoda/cities.json'
adding 'pogoda/main.py'
```

Это общая болезнь всех стеков: во Flutter ресурсы объявляются в `pubspec.yaml` (статья 11.1), на фронтенде картинки надо либо импортировать, либо класть в особую папку, в контейнер файлы надо копировать явной командой. Забыли объявить — артефакт собрался без файла, и упадёт не при сборке, а при открытии того экрана, где этот файл нужен.

> **СОВЕТ**
> Запускать свой пакет из корня проекта мало: при таком запуске папка проекта всё равно попадает в список мест для поиска, и Python находит файл в исходниках, а не в установленном пакете. Именно поэтому ошибка номер четыре так долго остаётся незамеченной. Проверять надо из другой папки.

5. Версии зависимостей плавают

В настройках проекта было написано так:

```
"react": "^19.1.1",
"vite": "^7.1.7"
```

А установилось вот это:

```
$ npm ls react vite --depth=0
+-- react@19.3.1
`-- vite@7.3.7
```

Значок `^` означает «эта версия или новее в пределах совместимости». То есть один и тот же проект, установленный у тебя сегодня

и на сервере через месяц, получит **разные версии** библиотек. Обычно ничего не случается. Иногда случается, и выяснять, что именно изменилось, приходится с нуля.

С Python то же самое, и наглядно: одна строка `requests` в списке зависимостей притаскивает пять пакетов, каждый со своей версией:

```
certifi==2026.7.22
charset-normalizer==3.5.2
idna==3.20
requests==2.34.2
urllib3==2.8.0
```

Четыре из пяти ты не выбирал. Они и есть источник сюрпризов.

Лекарство — **файл с точными версиями**, который создаётся при установке автоматически: `package-lock.json`, `pubspec.lock`, `poetry.lock`, `requirements.txt`, собранный через `pip freeze`. В нём записано ровно то, что стояло у тебя, когда всё работало:

```
node_modules/react -> 19.3.1
node_modules/vite  -> 7.3.7
```

Этот файл надо **коммитить в репозиторий** и устанавливать зависимости командой, которая его уважает: `npm ci` вместо `npm install`. Иначе он бессмыслен.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Проект собирается у меня, но падает после установки.
Не предлагай правок наугад. Сначала перечисли, чем
запуск при разработке отличается от собранного артефакта
в этом проекте: пути к файлам, переменные окружения,
ресурсы, версии зависимостей. По пункту на строку.
```

Такой запрос даёт список мест, где искать, вместо случайной правки.

ЗАПРОС

```
Вот моя команда сборки и файл настроек окружения.
Перечисли, какие значения попадут в артефакт открытым
текстом и будут видны пользователю. Отдельно — что
станет undefined и почему.
```

ЗАПРОС

```
Перечисли все файлы, которые проект читает не как код:
данные, картинки, шрифты, шаблоны. Для каждого скажи,
объявлен ли он в настройках сборки. Если не объявлен —
напиши строку, которую надо добавить.
```

ЗАПРОС

```
Сборка падает с этой ошибкой: <текст>.
Нужны два варианта: правка кода и правка настроек сборки.
Для каждого скажи, что именно перестанет проверяться.
Выбирать буду я.
```

Последний приём стоит запомнить. Любую «правку сборки» надо получать вместе с ценой: без этого вопроса отключение проверки выглядит как решение.

НА ЧТО СМОТРЕТЬ

Из команды сборки пропала проверка. Было `tsc -b && vite build`, стало `vite build`. Сборка зелёная, типы не проверяются.

Появились флаги подавления. `--force`, `--legacy-peer-deps`, `--no-verify`, `--ignore-scripts`, `|| true` в конце команды. Каждый из них что-то выключает, и это «что-то» надо назвать вслух.

Секрет в коде приложения. Ключ API, пароль, токен в файле, который собирается и уходит на устройство. Проверяется поиском по собранному файлу.

Относительные пути к файлам данных. `open("data/x.json")` вместо пути от `__file__`.

Ресурсы не объявлены. Картинка, шрифт или файл данных есть в папке проекта, но не упомянут в настройках сборки.

Файл с точными версиями в `.gitignore`. Классическая «оптимизация»: `package-lock.json` выглядит как мусор, потому что большой и меняется. Без него воспроизводимой сборки нет.

Сборка на твоей машине как единственный способ. Пока проект собирается только у тебя и только руками, никто — включая тебя через полгода — не сможет его собрать.

ШПАРГАЛКА

- Сборка — это не запуск: другой код, другие настройки, другие файлы.
- Зелёная сборка не значит рабочий код: сборщик типы выбрасывает, а не проверяет.
- Правка команды сборки вместо кода — почти всегда отключение проверки.
- Настройки окружения запекаются в артефакт: секретам там не место.
- Переменная без нужного префикса молча превращается в `undefined`.
- Относительный путь считается от папки запуска, а не от файла с кодом.
- Всё, что не код, надо объявлять в настройках сборки отдельно.
- Проверяй установленный пакет из другой папки, иначе найдёшь исходники.
- Файл с точными версиями коммить и ставить через `npm ci`, а не `install`.

12.2 Android и iOS

версии, подписи, разрешения, идентификатор

ЧТО ЭТО

У мобильного приложения, кроме кода, есть **паспорт**: кто его выпустил, какая это версия, что ему разрешено делать и как он называется для системы. Эти четыре вещи живут не в коде, а в настройках платформенных папок — и именно из-за них сборка, которая прекрасно запускалась на твоём телефоне, не принимается магазином.

ИИ помогает здесь хуже, чем с кодом: требования меняются каждый год, а обучающие данные остаются прежними. Зато сами понятия не меняются десятилетиями — их и стоит знать.

ВНИМАНИЕ

Как и глава 11, эта статья написана на уровне чтения: фрагменты настроек здесь иллюстративные, а не скопированные из конкретного проекта. Требования магазинов меняются каждый год. Названия файлов и смысл полей устойчивы, конкретный синтаксис сверяй с документацией своей версии.

КАК ВЫГЛЯДИТ В КОДЕ

1. Версия — это два разных числа

```
version: 1.2.0+34
```

Одна строка в `pubspec.yaml`, а при сборке из неё получается два значения:

Часть	Android	iOS	Кто её видит
`1.2.0`	versionName	Version	пользователь в магазине
`34`	versionCode	Build	только магазин

Первое — для людей, его показывают в описании обновления. **Второе — для магазина**, и оно обязано расти при каждой загрузке. Не вырастило — загрузка отклоняется с сообщением, что такая версия уже есть.

Это самая частая и самая глупая потеря времени при публикации: сборка идёт двадцать минут, а отказ приходит из-за числа, которое забыли увеличить на единицу.

2. Идентификатор — навсегда

```
Android:  com.lunario.pogoda      (applicationId)
iOS:      com.lunario.pogoda      (bundle identifier)
```

Это внутреннее имя приложения в системе и в магазине. Правило одно и оно жёсткое: **после первой публикации его сменить нельзя**. Можно только выпустить другое приложение и просить пользователей переустановить — с потерей всех отзывов, оценок и установок.

Шаблоны проектов подставляют сюда `com.example.<имя>`. Опубликовать с `com.example` не получится: магазины такие идентификаторы не принимают. Менять надо до первой публикации, и лучше — в первый день.

3. Разрешения объявляются в двух местах по-разному

Android — список в манифесте:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/
>
<uses-permission android:name="android.permission.INTERNET"/>
```

iOS — не список, а **объяснения для пользователя**:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>Нужно, чтобы показать погоду в вашем городе.</string>
```

Разница принципиальная. На Android ты объявляешь, что приложению нужно. На iOS ты обязан написать, **зачем** — текстом, который пользователь увидит в системном окне. Нет текста — приложение не пройдёт проверку, а на устройстве упадёт в момент запроса.

И ещё одно, что часто понимают неправильно: объявление в манифесте — это не разрешение. Разрешение даёт пользователь, во время работы приложения, и может отказать (статья 11.5). Объявление лишь даёт право спросить.

4. Подпись

Артефакт, который уходит в магазин, подписан криптографическим ключом. Подпись отвечает на один вопрос: **это обновление того же приложения или другого?**

Платформа	Чем подписывают	Где хранится
Android	ключ в файле-хранилище (keystore)	у тебя, не в репозитории
iOS	сертификат разработчика и профиль	в учётной записи Apple

Про Android-ключ надо знать одну вещь, и она неприятная: **потерял ключ — не можешь выпустить обновление.** Никогда. Приложение в магазине остаётся, а обновить его нельзя — только публиковать заново под новым идентификатором. Сервис подписи от магазина частично спасает ситуацию, но включать его надо заранее, до первой публикации.

Отсюда два правила:

- **Ключ и пароли к нему — не в репозиторий.** Файл хранилища, пароли, пути к ним выносятся в файл, который в `.gitignore`. Это ровно тот же разговор, что про `.env` в статье 5.7, просто ставка выше.
- **Резервная копия ключа — в двух местах,** и ни одно из них не твой ноутбук.

Отладочная подпись, которой инструменты подписывают сборки при разработке, для магазина не годится. Признак проблемы: артефакт называется `app-debug.apk` или в логе сборки видно `signingConfig debug`.

> **ЗАПОМНИ**
> Четыре вещи надо решить **до** первой публикации, потом поздно: идентификатор приложения, схема подписи и место хранения ключа, имя в магазине, набор разрешений. Остальное — код, дизайн, цены, описание — меняется обновлениями сколько угодно.

Что ещё отрежет пользователей

```
minSdk 26      ← ниже этой версии Android приложение не установится
```

Минимальная версия системы — тихое решение с громкими последствиями. Подняли, чтобы заработала библиотека, — отрезали часть пользователей. Это нормальный обмен, но он должен быть осознанным: ИИ поднимает минимальную версию мимоходом, одной строкой в настройках, и в объяснении об этом не пишет.

КАК СКАЗАТЬ ИИ

Ключевой приём для этой статьи — **требовать объяснения последствий, а не правок**. Настройки платформы меняются редко и стоят дорого.

> **ЗАПРОС**
> ```
> Предложи, что изменить, чтобы заработала эта библиотека.
> Для каждой правки в android/ и ios/ отдельно напиши,
> что она меняет для пользователя: минимальная версия
> системы, новые разрешения, размер приложения.
> Правки, затрагивающие minSdk, помечай явно.
> ```

> **ЗАПРОС**
> ```
> Вот список разрешений моего приложения: <список>.
> Приложение делает только <описание>.
> Какие из разрешений лишние? Для каждого нужного напиши
> текст объяснения для iOS — по-русски, одно предложение.
> ```

Второй запрос решает сразу две задачи: убирает лишние разрешения, из-за которых отказывают в публикации, и пишет те самые тексты объяснений, которые забывают.

> **ЗАПРОС**
> ```
> Я готовлю обновление. Что надо поднять в версии
> и где именно это записано в проекте?
> Покажи строку до и после.
> ```

> **ЗАПРОС**
> ```
> Проверь, не попали ли в репозиторий файлы подписи
> и пароли: ключи, хранилища, профили, файлы свойств.
> Список найденного и строки для .gitignore.
> ```

НА ЧТО СМОТРЕТЬ

Разрешения «на всякий случай». В приложении со списком городов появились камера и контакты. ИИ добавляет разрешения, когда библиотека теоретически умеет ими пользоваться. Магазины за это отказывают, а пользователи — удаляют.

Нет текста объяснения для iOS. Разрешение объявлено, объяснения нет: отказ при проверке.

Версия не поднята. Загрузка отклонена, двадцать минут сборки впустую.

Идентификатор остался `com.example`. Шаблон не правили.

Отладочная подпись. `app-debug.apk` вместо подписанного артефакта.

Файлы подписи в репозитории. Хранилище ключа, пароли, профили — всё это попадает в историю git, а оттуда уже не удаляется просто так.

`minSdk` поднят без объяснения. Одна строка, и часть пользователей больше не может установить приложение.

Разрешение объявлено, но отказ не обработан. Самая частая ошибка в коде на эту тему — та же, что в статье 11.5.

ШПАРГАЛКА

- `1.2.0+34`: первое для людей, второе для магазина и обязано расти.
- Идентификатор приложения после публикации не меняется — решай сразу.
- Android объявляет разрешения, iOS требует объяснить зачем — текстом.
- Объявление в манифесте ≠ разрешение: его даёт пользователь и может отказать.
- Потерял ключ подписи — обновления не будет. Копия в двух местах.
- Ключи и пароли — в `.gitignore`, а не в репозиторий.
- `minSdk` — это решение о том, кто не сможет поставить приложение.
- Проси у ИИ не правку настроек, а правку с ценой для пользователя.

12.3 Публикация и обновления

почему старая версия живёт дольше, чем кажется

ЧТО ЭТО

Публикация — это не финал, а начало состояния, в котором **у твоей программы одновременно несколько версий**. Сервер обновился, а приложения у пользователей нет. Сайт обновился, а в браузере лежит старый файл. Новая версия в магазине, а половина людей на прошлогодней.

Поэтому главный вопрос при каждом обновлении не «как выложить», а **«что будет с теми, кто ещё на старой»**. Этот вопрос не задаёт ни ИИ, ни инструменты публикации — его задаёшь только ты.

КАК ВЫГЛЯДИТ В КОДЕ

Три способа публикации — три разных скорости

Что публикуем	Как попадает к людям	Можно откатить
сайт	мгновенно, при следующем открытии	да, мгновенно
сервер, API	мгновенно, всем сразу	да, если готовился
приложение в магазине	после проверки, по желанию людей	нет

Разница в последнем столбце — самая важная. Сайт откатывается за минуту. Приложение в магазине **откатить нельзя**: можно только выпустить следующую версию и ждать, пока её поставят. Поэтому и проверяют мобильные сборки тщательнее, и раскатывают их на часть пользователей, а не на всех сразу.

Сайт: кэш, и как сборка его обходит

Собираем проект два раза подряд, не меняя ни строчки:

```
$ npx vite build
dist/assets/index-Dg-DehUb.js  222.57 kB | gzip: 69.34 kB
$ npx vite build
dist/assets/index-Dg-DehUb.js  222.57 kB | gzip: 69.34 kB
```

Имя файла одно и то же. Теперь меняем в тексте одно слово и собираем снова:

```
$ npx vite build
dist/assets/index-BKJJYt4K.js  222.59 kB | gzip: 69.36 kB
```

Имя изменилось. Эти буквы в имени — отпечаток содержимого: другое содержимое — другое имя. Сделано это ровно для одной цели: чтобы браузер, который держит файлы у себя, не подсунул пользователю старый код. Старый файл в кэше никому не мешает — за ним просто больше никто не придёт.

Но один файл в этой схеме имени не меняет:

```
dist/index.html                   0.39 kB
dist/assets/index-BKJJYt4K.js  222.59 kB
```

`index.html` — тот самый файл, в котором написано, какой из хэшированных файлов загружать. **Его кэшировать нельзя.** Если хостинг закэширует `index.html` на сутки, пользователь сутки будет открывать старую страницу, ссылающуюся на старый код, и никакая публикация этого не исправит.

Отсюда правило настройки хостинга, которое стоит знать наизусть: **файлы с хэшем в имени — кэшировать надолго, `index.html` — не кэшировать**. Это единственная настройка, которую стоит проверить руками при первой публикации сайта.

> **СОВЕТ**
> «Я обновил сайт, но у меня всё старое» в девяти случаях из десяти — это кэш, а не неудачная публикация. Проверяется открытием в режиме инкогнито: если там новое — публикация прошла, виноват кэш.

Сервер: обновление задевает тех, кого не ждали

Сервер обновляется мгновенно и для всех. И вот здесь вылезает то, о чём не думают: **у старых приложений на телефонах нет кнопки «обновись сейчас»**.

Переименовали поле в ответе API — приложения, выпущенные полгода назад, перестали показывать погоду. Сделали поле обязательным в запросе — старые версии перестали отправлять данные. Формально ты поменял свой сервер; реально ты сломал программы, которые уже стоят у людей.

Правило, которое из этого следует:

- **добавлять поля можно**, старые клиенты их просто не заметят;

- **удалять и переименовывать — нельзя**, пока живы те, кто ими пользуется;
- если изменение неизбежно — рядом появляется `/api/v2/`, а `/api/v1/` продолжает работать (статья 7.3).

Второе, что ломается на сервере, — **база данных**. Обновление базы и обновление кода происходят не одновременно, и порядок имеет значение: сначала добавляется новое поле, потом выкладывается код, который им пользуется. В обратном порядке получится несколько минут, в которые код ищет то, чего ещё нет.

И третье: откат кода — дело одной команды, а **откат изменений в базе — нет**. Удалённый столбец не вернётся вместе с откатом. Поэтому изменения базы, которые что-то удаляют, выкладывают отдельно и позже, когда уже понятно, что новая версия живёт.

Магазин: что происходит после загрузки

```
загрузили → проверка → доступно части людей → доступно всем
```

Проверка занимает от часов до дней и может закончиться отказом. Самые частые причины отказа не технические: разрешения без объяснения (статья 12.2), нерабочая демонстрация для проверяющего, отсутствие политики конфиденциальности, если приложение собирает хоть что-нибудь.

Поэтапная раскатка — выдать новую версию сначала небольшой доле пользователей — единственная защита от того, что откатить публикацию нельзя. Если в первые сутки посыпались падения, раскатку останавливают, и большинство людей так и остаётся на работающей старой версии.

> **ВНИМАНИЕ**
> Публикуй не в пятницу. Это не шутка и не примета: откат мобильного приложения невозможен, проверка в магазине идёт часами, а чинить придётся тебе. Выкладывай тогда, когда у тебя есть день посмотреть на результат.

КАК СКАЗАТЬ ИИ

Публикация — это та область, где от ИИ полезнее всего получать **списки проверок, а не действия**.

ЗАПРОС

```
Я меняю ответ API: поле "temp" становится "temperature".
Мои мобильные приложения в магазине обновляются не сразу.
Что сломается у тех, кто на старой версии?
Предложи способ сделать это изменение без поломки.
```

ЗАПРОС

```
Я добавляю в базу обязательное поле. Напиши порядок
действий: что выкладывать первым, что вторым и что
произойдёт, если выложить в обратном порядке.
Отдельно: какая часть этого не откатывается.
```

Второй запрос стоит задавать всегда, когда меняется база. Ответ на вопрос «что не откатывается» — это и есть список того, что надо делать осторожно.

ЗАПРОС

```
Вот настройки кэширования моего хостинга: <настройки>.
Файлы сборки имеют хэш в имени, index.html — нет.
Что настроено неправильно и к чему это приведёт?
```

ЗАПРОС

```
Составь чек-лист публикации для моего проекта: <описание>.
Раздели на «до публикации», «сразу после» и «через сутки».
Только то, что относится к моему стеку.
```

НА ЧТО СМОТРЕТЬ

Изменение API без мысли о старых клиентах. Поле переименовано, версия не добавлена. Работает у тебя, сломано у тех, кто не обновился.

Удаление в базе одной выкладкой с кодом. Откат станет невозможным ровно в тот момент, когда понадобится.

`index.html` **кэшируется.** Публикации перестают доходить до пользователей, и причину ищут в коде.

Нет способа узнать, что сломалось. После публикации нужны логи и сообщения об ошибках, иначе о поломке сообщат пользователи — или не сообщат.

Публикация делается руками и по памяти. Пока это пять команд в терминале, которые ты помнишь, — однажды ты выполнишь их в другом порядке.

Версия не видна в приложении. Когда пользователь жалуется, первый вопрос — какая у него версия. Если её негде посмотреть, разговор не состоится.

Миграция без проверки на копии данных. Запуск изменения структуры сразу на настоящей базе — самый дорогой из возможных способов узнать, что в нём ошибка.

ШПАРГАЛКА

- После публикации версий всегда несколько: думай о тех, кто на старой.
- Сайт откатывается мгновенно, сервер — если готовился, приложение — никогда.
- Хэш в имени файла — защита от кэша; `index.html` кэшировать нельзя.
- «Старое в браузере» — проверь инкогнито, прежде чем искать в коде.
- В API можно добавлять, нельзя удалять и переименовывать.
- Сначала поле в базе, потом код; удаления — отдельно и позже.
- Откат кода есть, откат базы — нет.
- Раскатывай на часть пользователей и не публикуй в пятницу.

12.4 Чек-лист перед публикацией

один раз на проект и один раз на выпуск

ЧТО ЭТО

Два списка. Первый проходится однажды, в начале проекта, и дорого обходится, если про него забыть. Второй — перед каждым выпуском, и его стоит держать прямо в репозитории, рядом с кодом.

ОДИН РАЗ НА ПРОЕКТ

1. Идентификатор приложения. Не `com.example`. После публикации не меняется (статья 12.2).

2. Ключ подписи и его резервная копия. В двух местах, ни одно из них — твой рабочий компьютер.

3. Файл `.gitignore` — проверен руками, а не взят на веру. Вот настоящий `.gitignore` из свежего шаблона популярного сборщика:

```
node_modules
dist
dist-ssr
*.local
```

Строки для `.env` в нём нет. Проверка: создаём файл `.env` с ключом и делаем обычный `git add -A`:

```
$ git ls-files | grep -c "\.env"
1
```

Секрет уехал в репозиторий. Не потому, что кто-то ошибся, а потому, что шаблон так устроен: `*.local` закрывает `.env.local`, но не `.env`. Отсюда правило: после первого коммита посмотри своими глазами, **что именно** попало в репозиторий:

```
git ls-files
```

Один раз, в начале, пока история короткая. Потом удалить файл из истории гораздо труднее, чем не добавлять.

4. Файл с точными версиями закоммичен — `package-lock.json`, `pubspec.lock`, `poetry.lock` (статья 12.1).

5. Сборка проверена из чистого клона. Не из своей папки, где всё уже работает, а так:

```
git clone <твой репозиторий> /tmp/proba
cd /tmp/proba && npm ci && npm run build
```

Это единственный способ узнать, что проект собирается не только у тебя. Всё, что лежало только на твоём диске и не было закоммичено, проявится здесь.

6. Версия видна в самом приложении. Строка на экране «о программе» или в подвале сайта. Без неё разговор с пользователем о поломке невозможен.

7. Настройки кэширования хостинга. Файлы с хэшем — надолго, `index.html` — никогда (статья 12.3).

8. Логи и ошибки где-то собираются. Иначе о поломках ты узнаёшь от пользователей.

ПЕРЕД КАЖДЫМ ВЫПУСКОМ

1. Проверки не отключены. В команде сборки на месте то, что её проверяет: `tsc -b`, `mypy`, `flutter analyze`. Отключали ли их с прошлого раза — смотреть в истории изменений, а не вспоминать.

2. Тесты прогнаны. Хотя бы те, что есть (глава 4).

3. Сборка с нуля. Удалить `dist`, `build`, папку зависимостей и собрать заново. Старые артефакты умеют скрывать поломки.

4. Поиск секретов в артефакте. Буквально:

```
grep -r "sk-\|secret\|password\|api_key" dist/
```

Нашлось — это утечка, а не придирка (статья 12.1).

5. Ресурсы на месте. Картинки, шрифты, файлы данных объявлены в настройках сборки и присутствуют в артефакте.

6. Версия поднята. Для магазина — число обязано вырасти.

7. Запущен артефакт, а не исходники. Открыть собранную папку, установить собранный пакет, поставить собранный `.apk` на телефон. Проверка исходников проверяет не то, что получит пользователь.

8. Изменения в API и базе разобраны по статье 12.3. Что добавлено, что переименовано, что не откатывается.

9. Записано, как откатиться. Одна строка: команда или номер предыдущего выпуска. Писать её надо до публикации, потому что после — будет некогда.

> **ЗАПОМНИ**
> Главный пункт в обоих списках — пятый из первого: **собрать проект из чистого клона**. Он один закрывает большую часть ошибок этой главы: незакоммиченные файлы, необъявленные ресурсы, плавающие версии, забытые настройки окружения.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Составь чек-лист публикации под мой проект.
Стек: <перечисли>. Публикуется: <куда>.
Раздели на «один раз на проект» и «перед каждым выпуском».
Только то, что относится к моему стеку, без общих советов.
Оформи списком, который я положу в репозиторий.
```

Получившийся файл стоит держать в проекте и дополнять каждый раз, когда что-нибудь ломается. Чек-лист, выросший из собственных ошибок, работает лучше любого готового.

ЗАПРОС

```
Вот мой .gitignore: <содержимое>.
Что из обычного для этого стека в нём не закрыто?
Отдельно: что из уже закоммиченного похоже на секрет.
```

ЗАПРОС

```
Я собрал проект. Напиши команды, которыми проверить
артефакт перед публикацией: поиск секретов, наличие
ресурсов, запуск собранной версии. Под мою ОС: <какая>.
```

НА ЧТО СМОТРЕТЬ

Чек-лист в голове. Пока он не написан, он разный каждый раз.

«Проверю потом». Публикация — единственный шаг, после которого «потом» означает «у пользователей».

Правка прямо в артефакте. Поправили файл в `dist/` или на сервере, чтобы быстро починить. Следующая сборка это затрёт, и поломка вернётся — уже без объяснения причин.

Публикация без возможности откатиться. Ты не знаешь, как вернуть прошлую версию, пока не попробовал.

ШПАРГАЛКА

- Два списка: один раз на проект и перед каждым выпуском.
- `.gitignore` из шаблона не закрывает `.env` — проверь `git ls-files` сам.
- Соберись из чистого клона: это главная проверка всей главы.
- Перед выпуском — сборка с нуля, а не поверх старой.
- Поищи секреты в артефакте командой `grep`, а не глазами.
- Запускай и проверяй артефакт, а не исходники.
- Способ откатиться записывается до публикации, а не после.
- Свой чек-лист в репозитории полезнее любого готового.

ЧАСТЬ III

Сквозные темы

То, что не принадлежит ни одному стеку и решается один раз на проект: как разложить задумку на части, где хранить данные, что писать в настройках. И практикум — полные примеры от идеи до работающей программы, в которых всё предыдущее собирается вместе.

13

Архитектура на пальцах

Решения, которые принимаются в первый день и потом не переделываются. Как разложить задумку на части, где хранить данные, что писать в настройках — и как объяснить всё это ИИ одним абзацем, чтобы он не выдумывал структуру заново в каждом ответе.

13.1 Как разбить задумку на части

три вопроса и одно правило направления

ЧТО ЭТО

Архитектура звучит солидно, но на деле это ответ на один вопрос: **что от чего зависит**. Не «какие папки завести» — папки появятся потом, сами, — а кто кого имеет право трогать.

Пока проект маленький, это кажется лишним: один файл, всё работает. Проблема не в размере, а в том, что без ответа на этот вопрос **каждый новый ответ ИИ кладёт код туда, где он оказался удобен сейчас**. Через месяц в проекте три места, где обращаются к базе, два разных способа читать настройки и функция на двести строк, которую никто не хочет трогать.

КАК ВЫГЛЯДИТ В КОДЕ

Три вопроса к любой задумке

Прежде чем просить код, ответь на три вопроса. Не ИИ — себе.

1. Что хранится? Перечисли вещи: город, пользователь, заметка, заказ, подписка. Это существительные, и из них потом получатся таблицы и модели данных.

2. Что происходит? Перечисли действия: добавить город, обновить погоду, отправить уведомление. Это глаголы, и из них получатся функции.

3. Что видно снаружи? Экран, страница, команда бота, адрес API. Это то, через что с программой разговаривают.

Три списка на полстраницы — и у проекта уже есть скелет. Приём работает и для бота на двести строк, и для приложения на двадцать экранов.

Правило направления

Дальше эти три списка раскладываются по слоям, и между слоями действует единственное правило, которое стоит помнить:



Рис. 13.1 · Стрелки идут только вниз. Слой данных не знает, что над ним есть экран, — поэтому его можно проверять отдельно.

Верхний слой знает о нижнем. Нижний о верхнем — нет. Экран обращается к логике, логика к данным. Обратно — никогда.

Это не теория и не вкусовщина: нарушение правила видно сразу, потому что программа перестаёт запускаться. Вот два файла, которые знают друг о друге:

```
# app/models.py
from app.services import fetch_weather

class City:
    def refresh(self):
        self.temp = fetch_weather(self.name)
```

```
# app/services.py
from app.models import City

def all_cities():
    return [City("Oslo"), City("Bergen")]
```

Запуск:

```
ImportError: cannot import name 'fetch_weather' from partially
initialized
module 'app.services' (most likely due to a circular import)
```

Круговой импорт — это сообщение о том, что слои перепутаны. Данные полезли наверх, в логику. Лечение не в хитром импорте внутри функции (так советуют в интернете, и так часто предлагает ИИ), а в развороте зависимости: модель не знает ничего, сервис знает о модели.

```
# app/models.py — ничего не импортирует
class City:
    def __init__(self, name, temp=None):
        self.name = name
        self.temp = temp
```

```
# app/services.py — знает о модели
from app.models import City

def all_cities():
    return [City(n, fetch_weather(n)) for n in ("Oslo", "Bergen")]
```

```
[City(Oslo, 4), City(Bergen, 4)]
```

ЗАПОМНИ

Круговой импорт — не загадочная ошибка Python, а **сигнал о том, что архитектура неправильная**. Если ИИ предлагает обойти его переносом импорта внутрь функции — он лечит симптом. Правильный вопрос: кто из этих двух файлов должен знать о другом?

Чем мельче, тем лучше — до предела

Разбивать имеет смысл до того уровня, где **каждый кусок можно объяснить одной фразой**. «Берёт город, возвращает температуру». «Хранит список городов пользователя». Как только для объяснения нужно слово «и» трижды — пора делить дальше.

Обратный предел тоже есть: двадцать файлов по десять строк — это не архитектура, а россыпь. Признак, что перестарался: чтобы понять одно действие, приходится открыть пять файлов подряд.

КАК СКАЗАТЬ ИИ

Главная ошибка — просить приложение целиком. Вместо этого разбиение стоит получить **текстом, до кода**:

ЗАПРОС

```
Задумка: <опиши в двух-трёх предложениях>.
Не пиши код. Разложи на части:
1. Что хранится — список сущностей с полями.
2. Что происходит — список действий.
```

```
3. Что видно снаружи — экраны или адреса.
Для каждого действия укажи, какие сущности оно трогает.
```

Ответ на такой запрос читается за минуту, а исправляется за две. Код, который пришлось бы переписывать, — за вечер.

ЗАПРОС

```
Вот моё разбиение: <три списка>.
Предложи структуру слоёв: что от чего зависит.
Нарисуй стрелками, кто кого импортирует.
Если где-то получается стрелка в обе стороны — скажи об этом.
```

ЗАПРОС

```
Я получил ImportError с упоминанием circular import.
Не предлагай перенести импорт внутрь функции.
Объясни, какой из двух модулей не должен знать о другом,
и как развернуть зависимость.
```

НА ЧТО СМОТРЕТЬ

Приложение одним куском. Попросили «сделай приложение» — получили один файл на четыреста строк, где всё вперемешку.

Круговые импорты. Разворачивать зависимость, а не прятать импорт.

Запрос к базе прямо в обработчике экрана. Интерфейс полез через голову логики сразу в данные. Работает — до первого изменения в базе.

Одно и то же действие в трёх местах. Три места, где форматируется температура, и все три надо не забыть поправить.

Файл с именем `utils` растёт. Туда складывают всё, чему не нашлось места. Это не слой, а шкаф, в который сметают.

Разбиение придумал ИИ, а ты его не читал. Структуру проекта стоит понимать целиком — это единственная часть, которую нельзя делегировать (статья 1.1).

ШПАРГАЛКА

- Архитектура — это ответ на вопрос «что от чего зависит».
- Три списка до кода: что хранится, что происходит, что видно снаружи.
- Три слоя: интерфейс → логика → данные. Стрелки только вниз.
- Круговой импорт — сигнал о перепутанных слоях, а не ошибка Python.
- Кусок правильного размера объясняется одной фразой без слова «и».
- Проси у ИИ разбиение текстом до кода — исправлять текст дешевле.
- Папка `utils` — место, куда сметают то, что не разложили.

13.2 Структура папок

решения, которые потом не переделать

ЧТО ЭТО

Структура папок — это архитектура из статьи 13.1, записанная в файловой системе. Её главная ценность не в красоте, а в том, что **она отвечает на вопрос «куда положить новый код» без обсуждения**. Пока ответа нет, каждый новый кусок ложится туда, где оказался курсор.

Для работы с ИИ у структуры есть вторая, не менее важная роль: это то, что ты называешь в запросе. «Добавь в `app/services/weather.py`» — запрос, на который можно получить предсказуемый ответ. «Добавь функцию погоды» — приглашение выдумать структуру заново.

КАК ВЫГЛЯДИТ В КОДЕ

Два способа раскладывать, и оба правильные

По типам кода — всё однотипное вместе:

```
app/
├── models/        что хранится
├── services/      что происходит
├── routes/        что видно снаружи
└── utils/
```

По возможностям — всё, что относится к одной теме, вместе:

```
app/
├── weather/       модель, логика и экран погоды
├── users/         модель, логика и экраны пользователей
└── shared/
```

Выбор между ними — не вкусовщина, а ответ на вопрос **«что чаще меняется целиком»**:

	Раскладка по типам	Раскладка по возможностям
Хорошо, когда	проект небольшой, тем мало	тем много, они независимы
Правка затрагивает	несколько папок	обычно одну
Легко понять	что вообще есть в проекте	как устроена одна тема
Ломается, когда	тем становится много	темы начинают переплетаться

Для первого проекта разумнее раскладка по типам: её понимают все, включая ИИ, и в ней меньше решений. Переход на раскладку по возможностям имеет смысл, когда в проекте становится больше пяти-шести независимых тем.

Чего делать не надо — мешать оба способа. Папка `models/` рядом с папкой `weather/`, внутри которой свои модели, — это не компромисс, а два ответа на один вопрос. Новый код после этого можно положить в оба места.

Что переделывается дорого

Не все решения равны. Эти три стоят того, чтобы подумать над ними в первый день:

1. Граница между слоями. Если логика и работа с базой изначально перемешаны, разделить их потом — это переписать почти всё. Обратно, из разделённого сделать смешанное, никто никогда не просит.

2. Имя корневого пакета. Оно попадает в каждый импорт проекта. Менять — значит менять каждый файл. Не `app` и не `src`, если проект когда-нибудь станет пакетом: своё, короткое, понятное.

3. Место данных. Решение «пока сложим в JSON» живёт дольше, чем планировалось (статья 13.3).

Остальное — имена папок, порядок файлов, где лежат тесты — переделывается за полчаса и не стоит долгих раздумий.

Размер файла

Ориентир, который работает: **один файл — одна причина меняться**. Если файл приходится открывать и когда меняется формат ответа API, и когда меняется внешний вид экрана — в нём два разных файла.

Числовой ориентир — 300–500 строк. Не правило, а сигнал: дошло до тысячи — скорее всего, внутри прячутся два файла.

С ИИ у этого есть практическое следствие: **файл, который целиком помещается в запрос, обсуждать намного проще**. Правка в файле на двести строк — это точный запрос и точный ответ. Правка в файле на тысячу — это пересказ и надежда.

> **СОВЕТ**
> Структуру проекта стоит один раз записать текстом и держать в репозитории — в `README.md` или в отдельном `ARCHITECTURE.md`. Тогда в любом запросе её можно вставить целиком, и ИИ перестанет предлагать «создайте файл `helpers.py`» там, где у тебя уже есть `services/`.

Чего быть не должно

```
app/
├── utils.py          ← 900 строк всего подряд
├── helpers.py        ← то же самое, но другое
├── main_v2.py        ← прошлая версия, вдруг пригодится
├── test.py           ← не тест, а «попробовать»
└── new/              ← переписываем, но старое пока оставим
```

Каждая строка здесь — отложенное решение. `utils` и `helpers` — это «потом разложу». `main_v2` и `new/` — «потом удалю». Ни то, ни другое не происходит, зато в каждом запросе к ИИ приходится объяснять, какие файлы настоящие.

Старое не хранится рядом с новым: для этого есть git (статья 5.1).

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Вот структура моего проекта:
<дерево папок>

Я добавляю возможность: <какую>.
Скажи, в какие файлы ляжет код и какие файлы придётся
изменить. Новые файлы называй по моей схеме.
Код пока не пиши.
```

Этот запрос стоит делать привычкой: он занимает минуту и снимает самую частую причину беспорядка — код, положенный не туда.

ЗАПРОС

```
Структура проекта: <дерево>.
Правила: models ничего не импортирует, services импортирует
models, routes импортируют services.
Проверь по этим правилам файл, который ты сейчас написал.
```

ЗАПРОС

```
У меня файл на 900 строк: <содержимое или описание>.
Предложи, на какие файлы его разбить и по какому признаку.
Для каждого — одна фраза, за что он отвечает.
Переносить код пока не надо.
```

ЗАПРОС

```
В проекте есть utils.py, в котором всё подряд.
Разложи его содержимое по существующим папкам.
Для каждой функции скажи, куда и почему.
Если что-то действительно общее — оставь, но назови точнее.
```

НА ЧТО СМОТРЕТЬ

ИИ предложил новую папку рядом с существующей. `helpers/` при живом `services/`, `db/` при живом `models/`. Признак того, что структуру не назвали в запросе.

Смешаны две раскладки. `models/` и `weather/` в одной папке.

`utils` **растёт.** Каждая функция, положенная туда, — отложенное решение.

Файлы с версиями в имени. `main_v2`, `app_new`, `index_final`. Для этого есть git.

Тесты вперемешку с кодом. Отдельная папка `tests/` рядом, а не файл `test.py` в каждой папке.

Один файл растёт на каждом запросе. Если ИИ дописывает в один и тот же файл четвёртую несвязанную функцию — пора делить.

ШПАРГАЛКА

- Структура папок отвечает на вопрос «куда положить новый код».
- По типам или по возможностям — выбирай по тому, что чаще меняется целиком.
- Не мешай обе раскладки: получится два ответа на один вопрос.
- Дорого переделывать: границу слоёв, имя корневого пакета, место данных.
- Один файл — одна причина меняться; 300–500 строк как сигнал.
- Файл, помещающийся в запрос, обсуждать в разы проще.
- Структуру запиши текстом в репозиторий и вставляй в запросы.
- `utils`, `main_v2`, `new/` — отложенные решения, а не папки.

13.3 Где хранить данные

файл, SQLite или настоящая база

ЧТО ЭТО

Четыре варианта, и выбирают почти всегда между вторым и третьим:

Где	Когда это правильный ответ
в памяти	данные нужны только пока программа работает
в файле (JSON, CSV)	настройки, кэш, выгрузка, пара сотен записей
SQLite	один процесс, до сотен тысяч записей, нужен поиск
сервер БД	несколько процессов, сеть, права доступа, рост

Решение принимается в первый день, в одну строку, и почти никогда не пересматривается — потому что к моменту, когда оно начинает мешать, на нём уже держится весь проект. Поэтому стоит знать не только, чем варианты отличаются, но и **когда именно разница становится заметной**.

КАК ВЫГЛЯДИТ В КОДЕ

Файл против SQLite: цифры

Самый частый сценарий: «пока сложим в JSON, потом переделаем». Хранить список в файле — значит при каждой записи прочитать файл целиком, дописать и записать целиком обратно.

```
with open("data.json", encoding="utf-8") as f:
    rows = json.load(f)
rows.append(record)
with open("data.json", "w", encoding="utf-8") as f:
    json.dump(rows, f)
```

Вот сколько времени занимает добавление записей — замерено, одна и та же машина:

Записей	JSON, с	SQLite, с
500	1,0	0,8
1 000	3,2	1,3
2 000	11,1	2,6
4 000	45,9	5,4
8 000	176,4	10,2

Смотреть надо не на разницу в последней строке, а на **то, как растут столбцы**. Удвоение числа записей удваивает время у SQLite и учетверяет у JSON: каждая новая запись заставляет перечитывать всё, что уже накопилось.

И главное: **на пятистах записях разницы нет** — секунда против восьми десятых. Именно поэтому неправильное решение не видно в тот момент, когда его принимают. Оно становится видно через полгода, когда записей восемь тысяч и программа задумывается на три минуты.

Поиск

```
поиск одного города среди 50 000
  JSON (загрузить файл и перебрать):  50,2 мс
  SQLite с индексом                :   0,01 мс
```

Пять тысяч раз. Причина простая: файл нужно прочитать целиком и разобрать, прежде чем в нём что-то искать, — а база читает только нужную страницу (статья 7.6).

Два процесса пишут одновременно

Это та разница, которую не обойти оптимизацией. Два процесса добавляют по 300 записей в один JSON-файл:

```
прогон 1: записано 300 из 600 (A=300, B=0)
прогон 2: записано 300 из 600 (A=82,  B=218)
прогон 3: записано 300 из 600 (A=11,  B=289)
```

Половина данных пропала во всех трёх прогонах, и каждый раз по-разному. Плюс в каждом прогоне один из процессов падал:

```
json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

Это чтение файла ровно в тот момент, когда второй процесс успел его очистить, но ещё не записал новое содержимое.

Тот же тест с SQLite:

```
SQLite — ожидалось 600, записано 600 (A=300, B=300), потеряно 0
```

База умеет то, чего файл не умеет в принципе: **выстраивать одновременные записи в очередь**. Это и есть главная причина существования баз данных — не скорость, а то, что данные не теряются.

> **ВНИМАНИЕ**
> Если у программы может быть **два одновременных запущенных экземпляра** — веб-сервер с несколькими рабочими процессами, бот и скрипт обслуживания, две вкладки с одним приложением, — файл как хранилище не подходит. Не «медленно», а теряет данные. Молча.

Когда файла достаточно

Файл — отличное хранилище, когда пишет **один процесс и редко**:

- настройки, которые меняет человек руками;
- кэш ответов внешнего API;
- выгрузка и обмен данными;
- справочник, который обновляется раз в месяц;
- маленький бот, которым пользуешься ты один.

Признак, что файла хватит: ты можешь открыть его в блокноте и это полезно.

Когда SQLite перестаёт хватать

SQLite — не игрушечная база. Это полноценная SQL-база, которой хватает очень надолго, и для одного приложения на одном сервере она чаще всего и есть правильный ответ. Переходить на серверную базу (PostgreSQL, MySQL) имеет смысл, когда появляется хотя бы одно из:

- **несколько машин**, которым нужны одни и те же данные;
- **много одновременных пишущих** процессов (SQLite пускает писать по одному);
- **права доступа** — разным приложениям разный доступ к разным таблицам;
- **объём**, при котором нужны настройки, резервное копирование по расписанию, репликация.

Обратная сторона: серверная база — это ещё одна программа, которую надо поставить, настроить, обновлять и резервировать. Для бота, которым пользуются десять человек, это работа без выигрыша.

> **ЗАПОМНИ**
> Честный порядок выбора: **в памяти → файл → SQLite → сервер БД**. Брать следующий уровень стоит тогда, когда предыдущий перестал справляться по понятной причине, а не «на вырост». Но перепрыгивать с файла сразу на SQLite не страшно: это одна строка `sqlite3.connect`, а не новая инфраструктура.

Переход, который делают поздно

Самая дорогая часть перехода с файла на базу — не сама база, а то, что **код обращается к хранилищу из десяти мест**. Если с самого начала чтение и запись собраны в одном файле — `storage.py`, `repository.py`, как угодно, — переход занимает вечер, потому что меняется одна реализация, а не весь проект.

Это, пожалуй, самая полезная архитектурная привычка из всей главы: **весь доступ к данным — через одну дверь**.

КАК СКАЗАТЬ ИИ

ИИ по умолчанию предлагает то, что чаще встречалось в обучающих данных, — обычно PostgreSQL и ORM. Для бота на двести строк это лишнее. Поэтому выбор надо называть самому, а у ИИ спрашивать обоснование:

ЗАПРОС

```
Проект: <описание>. Записей ожидаю <сколько>.
Пишет <один процесс / несколько>.
Данные нужны <только мне / нескольким пользователям>.
Что из этого достаточно: файл, SQLite или сервер БД?
Назови признак, по которому выбранного перестанет хватать.
```

Последняя строчка — самая ценная: она превращает выбор в проверяемое условие.

ЗАПРОС

```
Сделай весь доступ к данным через один модуль storage.py:
функции add_city, get_city, list_cities, delete_city.
Внутри — SQLite. Остальной код про базу знать не должен.
```

ЗАПРОС

```
Сейчас данные в JSON-файле, доступ размазан по проекту.
Найди все места, где читается или пишется этот файл.
Список с номерами строк. Код пока не меняй.
```

ЗАПРОС

```
Перенеси хранение с JSON на SQLite.
Внешние функции storage.py и их сигнатуры не меняй.
Отдельно напиши скрипт переноса уже накопленных данных
и скажи, что делать, если он упадёт на середине.
```

НА ЧТО СМОТРЕТЬ

JSON-файл как основное хранилище у сервера или бота. Это потеря данных, а не медленная работа.

Доступ к данным из десяти мест. Переход на базу превращается в переписывание проекта.

PostgreSQL для проекта на одного пользователя. Настройка, резервные копии и обновления ради нагрузки, которой нет.

Нет проверки, что запись удалась. С файлом ошибка записи часто остаётся незамеченной.

Миграция данных без резервной копии. Скрипт переноса — это первый кандидат на то, чтобы упасть посередине.

Пароли и ключи в той же базе открытым текстом. Хранилище выбрано, а вопрос, что в нём лежит, не задан (статья 5.7).

Нет ответа на вопрос «что будет, если файл откроют дважды». Его стоит задать до того, как ответ придёт в виде пропавших данных.

ШПАРГАЛКА

- Порядок выбора: память → файл → SQLite → сервер БД.
- На 500 записях разницы нет; на 8 000 — 176 секунд против 10.
- Время записи в файл растёт квадратично: каждая запись перечитывает всё.
- Поиск среди 50 000: 50 мс в файле против 0,01 мс в базе с индексом.
- Два процесса на один файл — половина данных теряется, и это измерено.
- Файл хорош там, где пишет один процесс и редко.
- SQLite хватает надолго; сервер БД — когда машин больше одной.
- Весь доступ к данным — через один модуль. Это и есть страховка перехода.

13.4 Конфиги и окружения

три беды одного маленького файла

ЧТО ЭТО

Окружение — это набор настроек, с которыми запускается один и тот же код. Обычно их три: **твоя машина**, **тестовый сервер**, **боевой сервер**. Отличаются они не кодом, а значениями: адрес базы, ключи API, включена ли отладка.

Настройки выносят из кода по двум причинам: чтобы не хранить секреты в репозитории (статья 5.7) и чтобы одна и та же программа работала в разных местах. Способ при этом везде один — **переменные окружения**, и в них есть ровно одна тонкость, из-за которой ломаются проекты: **всё, что приходит из окружения, — это строка**.

КАК ВЫГЛЯДИТ В КОДЕ

Файл настроек, который выглядит нормально

```
import os

DATABASE_URL = os.environ.get("DATABASE_URL", "sqlite:///test.db")
PORT = os.environ.get("PORT", 8000)
DEBUG = os.environ.get("DEBUG", False)
```

Три строки, три беды. Причём на твоей машине, где переменных нет, всё безупречно:

```
база   : sqlite:///test.db
порт   : 8000 int
отладка: False bool
  -> режим отладки выключен
порт + 1 = 8001
```

Теперь то же самое с переменными — так, как запускается на сервере:

```
PORT=8080 DEBUG=false DATABASE_URL=postgresql://prod/db python3 app.py
```

```
база   : postgresql://prod/db
порт   : '8080' str
отладка: 'false' str
  -> режим отладки ВКЛЮЧЁН
порт + 1 = TypeError: can only concatenate str (not "int") to str
```

Разберём по одной.

1. `DEBUG=false` включил отладку. Строка `'false'` — непустая, а значит, истинная. Проверка `if DEBUG:` срабатывает. На боевом сер-

вере это означает подробные страницы с ошибками, трассировки наружу и всё, что обычно видят только разработчики. Ошибка в одну строку, последствия — утечка.

2. Порт стал строкой. `TypeError` — ещё удачный случай: он хотя бы громкий. Хуже, когда строка молча проходит дальше.

3. Тип зависит от того, задана ли переменная. На твоей машине `PORT` — число, на сервере — строка. Один и тот же код, разные типы. Проверить это локально нельзя в принципе.

И отдельно — **самая тихая из трёх**: `DATABASE_URL` со значением по умолчанию `sqlite:///test.db`. Забыли задать переменную на сервере — приложение не упало, не пожаловалось, а начало писать в локальный файл. Узнаёшь об этом, когда данные ищут и не находят.

> **ВНИМАНИЕ**
> Значение по умолчанию для настройки, без которой программа не должна работать, — это не удобство, а **способ спрятать ошибку**. Для адреса базы, ключей и паролей умолчаний не бывает: нет значения — программа обязана упасть сразу и сказать, чего ей не хватает.

Как надо

```python
from pydantic import Field
from pydantic_settings import BaseSettings


class Settings(BaseSettings):
    database_url: str
    api_key: str
    port: int = 8000
    debug: bool = False
    timeout: float = Field(default=5.0, gt=0)


settings = Settings()
```

Разница вся в типах. `port: int` — значит, при запуске строка `'8080'` превратится в число или программа не запустится. `debug: bool` — значит, `false` станет настоящим `False`. У `database_url` и `api_key` умолчания нет — значит, без них запуска не будет.

Запуск без переменных:

```
pydantic_core._pydantic_core.ValidationError: 2 validation errors for
Settings
database_url
  Field required [type=missing, input_value={}, input_type=dict]
api_key
  Field required [type=missing, input_value={}, input_type=dict]
```

Обе недостающие переменные названы сразу, при старте. Не по одной, не в середине работы, не через час после выкладки.

С переменными:

```
база   : postgresql://prod/db
порт   : 8080 int
отладка: False bool
порт+1 : 8081
```

`DEBUG=false` стал `False`, как и ожидает любой здравый смысл. А если значение окажется мусором:

```
port
  Input should be a valid integer, unable to parse string as an integer
  [type=int_parsing, input_value='вось', input_type=str]
```

Названа настройка, названо значение, названа причина. Это и есть разница между «программа упала» и «программа объяснила».

> **ЗАПОМНИ**
> Настройки проверяются **один раз при старте**, а не в момент использования. Программа, которая падает через сорок минут работы из-за пустого ключа API, — это программа, которая не проверила настройки на старте.

Что лежит в репозитории, а что нет

```
.env           ← настоящие значения. В .gitignore. Никогда не
коммитится.
.env.example   ← те же имена, пустые или примерные значения. Коммитится.
```

`.env.example` — маленький файл с большой пользой: это **список того, что нужно задать**, чтобы проект запустился. Без него новый человек (или ты сам через полгода на другой машине) узнаёт о недостающей переменной методом проб.

```
DATABASE_URL=postgresql://user:password@localhost/dbname
API_KEY=
PORT=8000
DEBUG=false
```

И напоминание из статьи 12.4: в шаблонах проектов строки `.env` в `.gitignore` может не оказаться. Проверяется это один раз, командой `git ls-files`, и лучше в первый день.

Три окружения — один код

Правило простое: **код не знает, где он работает**. Нет `if environment == "production"`, разбросанных по проекту; есть настройки, которые в разных местах разные.

Единственное исключение, которое оправдано, — одна переменная вроде `ENV=production` и несколько решений на её основе в одном месте: уровень логов, показывать ли подробные ошибки, включать ли документацию API.

КАК СКАЗАТЬ ИИ

ЗАПРОС

```
Сделай модуль настроек на pydantic-settings.
Нужны: database_url (строка, обязательная),
api_key (строка, обязательная), port (int, по умолчанию 8000),
debug (bool, по умолчанию false).
Умолчаний для обязательных не ставь.
Рядом — .env.example со всеми именами.
```

ЗАПРОС

```
Найди в проекте все обращения к os.environ и os.getenv.
Для каждого скажи: тип значения, есть ли умолчание
и что произойдёт, если переменная не задана.
Таблицей. Код пока не меняй.
```

Этот запрос стоит выполнить на любом проекте, которому больше месяца: обращения к окружению расползаются незаметно.

ЗАПРОС

```
Проверь этот конфиг на три ошибки: строка вместо числа,
строка 'false' вместо булева значения, умолчание там,
где его быть не должно. Что нашёл — перечисли.
```

ЗАПРОС

```
Проект должен падать при старте, если не заданы
обязательные настройки. Сейчас он падает позже.
Сделай проверку при запуске, с понятным сообщением,
в котором перечислено всё недостающее сразу.
```

НА ЧТО СМОТРЕТЬ

`os.environ.get` **с умолчанием для адреса базы или ключа.** Самая тихая из ошибок этой статьи.

Настройка используется как число или как флаг без приведения типа. Из окружения приходит строка, всегда.

`DEBUG=true` **как строка.** Любая непустая строка истинна, включая `'false'` и `'0'`.

Настройки читаются в десяти местах. Один модуль настроек, остальной код обращается к нему.

`.env` **в репозитории.** Проверяется `git ls-files`.

Нет `.env.example`**.** Проект нельзя запустить, не угадав список переменных.

Ключи заданы прямо в коде «временно». Временное живёт дольше всего, а git помнит всё (статья 5.7).

`if production:` **по всему проекту.** Код знает, где он работает, — значит, на тестовом сервере проверяется не то, что поедет в бой.

ШПАРГАЛКА

- Всё, что приходит из окружения, — строка. Всегда.
- `DEBUG=false` без приведения типа включает отладку.
- Умолчание для обязательной настройки прячет ошибку, а не помогает.
- Типизированные настройки называют все недостающие переменные сразу.
- Настройки проверяются при старте, а не в момент использования.
- `.env` — в `.gitignore`, `.env.example` — в репозиторий.
- Один модуль настроек; остальной код читает его, а не окружение.
- Код не должен знать, на каком сервере он работает.

13.5 Как описать архитектуру ИИ

двадцать строк, которые экономят вечер

ЧТО ЭТО

У ИИ нет памяти о твоём проекте. В каждом новом разговоре он знает только то, что видит прямо сейчас, — и если не видит структуру, **придумывает свою**. Отсюда берутся ответы вида «создайте файл `helpers.py`» при живой папке `services/` и четвёртый по счёту способ читать настройки.

Лечится это одним коротким текстом — **архитектурной справкой**. Статья 3.14 говорила о файле контекста в целом: что за проект, какие команды, как запускать. Здесь — про ту его часть,

которая отвечает на вопрос «как устроено» и которую стоит писать отдельно, потому что именно её ИИ нарушает чаще всего.

КАК ВЫГЛЯДИТ В КОДЕ

Шаблон

Справка состоит из шести блоков. Больше не нужно: длинный текст читается хуже и устаревает быстрее.

```
1. ЧТО ЭТО        — одно предложение, что делает программа
2. СТЕК           — язык, фреймворк, база, версии
3. СТРУКТУРА      — дерево папок, одна строка пояснения на папку
4. ЗАВИСИМОСТИ    — кто кого имеет право импортировать
5. ГДЕ ЧТО ЛЕЖИТ  — данные, настройки, секреты, логи
6. ЧЕГО НЕ ДЕЛАТЬ — решения, которые не обсуждаются
```

Шестой блок — самый недооценённый. Без него ИИ в каждом третьем ответе предлагает добавить библиотеку, сменить базу или «заодно отрефакторить».

Заполненный пример

```
ЧТО ЭТО
Телеграм-бот: подписка на погоду в выбранных городах,
утренняя рассылка.

СТЕК
Python 3.12, aiogram 3, SQLite, requests.
Запуск: python -m bot. Сервер — один, systemd.

СТРУКТУРА
bot/
├── handlers/   команды и кнопки, только разбор ввода и ответ
├── services/   логика: погода, подписки, рассылка
├── storage.py  весь доступ к базе, других обращений к БД нет
├── models.py   датаклассы City, Subscription
└── config.py   настройки через pydantic-settings

ЗАВИСИМОСТИ
handlers -> services -> storage -> models
Обратных импортов нет. models не импортирует ничего из проекта.

ГДЕ ЧТО ЛЕЖИТ
Данные: SQLite, файл data/bot.db
Настройки: переменные окружения, см. .env.example
Секреты: только в .env, в репозиторий не попадают
Логи: stdout, их забирает systemd

ЧЕГО НЕ ДЕЛАТЬ
- Не добавлять ORM и другие библиотеки без отдельного разговора.
- Не обращаться к БД из handlers напрямую.
- Не менять схему таблиц без скрипта переноса данных.
- Не добавлять async там, где его нет.
```

Двадцать с небольшим строк. Вставляются в начало запроса, когда задача крупная, и лежат в `ARCHITECTURE.md` или в файле контекста всё остальное время.

Почему это работает лучше, чем кажется

Справка снимает ровно ту неопределённость, которую ИИ иначе закрывает догадкой. Он не угадывает, где лежит доступ к базе, — написано. Не выбирает между SQLite и PostgreSQL — написано. Не придумывает имя для нового файла — есть схема.

Побочный эффект важнее основного: **чтобы написать такую справку, её надо сначала знать**. Если блок «зависимости» не пишется, потому что в проекте импортируют все всех, — справка уже окупилась, просто иначе.

> **СОВЕТ**
> Справку пишет не ИИ, а ты. Можно попросить его **собрать черновик** по коду, но проверять и править обязательно самому: справка, в которую вписано то, чего в проекте нет, хуже отсутствия справки — ИИ будет уверенно ссылаться на несуществующее.

Проверка, которая занимает минуту

Самый быстрый способ понять, понял ли ИИ архитектуру:

ЗАПРОС
```
Перескажи своими словами, как устроен мой проект:
что от чего зависит и где что лежит. Пять строк.
Если чего-то в описании не хватает, чтобы начать работу, —
назови, чего именно.
```

Пересказ сразу показывает расхождения. Вторая часть вопроса не менее полезна: то, чего ИИ не хватает, — это обычно и есть дыра в справке.

Когда справку обновлять

Не когда меняется код, а **когда меняется решение**. Добавили библиотеку, сменили хранилище, завели новый слой, отказались от правила — справка переписывается в тот же день.

Признак, что справка устарела: ИИ делает «неправильно», но строго по написанному.

КАК СКАЗАТЬ ИИ

ЗАПРОС
```
Вот дерево моего проекта и несколько ключевых файлов.
Собери черновик архитектурной справки по шести блокам:
что это, стек, структура, зависимости, где что лежит,
чего не делать.
Где не уверен — пиши «?», не придумывай.
```

Указание про `?` обязательно: без него пустоты заполняются правдоподобным вымыслом (статья 1.6).

ЗАПРОС
```
Вот архитектурная справка проекта: <текст>.
Задача: <что нужно сделать>.
Сначала скажи, какие файлы затронешь и не нарушает ли
это правила зависимостей. Код пиши после моего «да».
```

ЗАПРОС
```
Вот справка и вот код, который ты только что написал.
Проверь свой код на соответствие справке: зависимости,
```

```
расположение файлов, запрет на новые библиотеки.
Нарушения перечисли списком.
```

Последний приём — **ИИ проверяет себя по твоим правилам** — работает заметно лучше, чем те же правила, высказанные в начале разговора: проверка идёт по готовому коду, а не по намерению.

НА ЧТО СМОТРЕТЬ

Справка написана ИИ и не прочитана тобой. В ней может быть описан не твой проект.

В справке нет блока «чего не делать». Самый полезный блок, и его забывают чаще всех.

Справка устарела. ИИ делает по написанному, а написанное уже неправда.

Справка на три страницы. Такую не вставишь в запрос и не прочитаешь глазами; всё, что длиннее экрана, перестаёт работать.

Нет версий. «Python и телеграм-бот» — это и aiogram 2, и aiogram 3, и они несовместимы.

Справка есть, но в запросах не используется. Файл в репозитории сам себя в разговор не вставит — если только это не файл контекста, который инструмент подхватывает сам (статья 3.14).

ШПАРГАЛКА

- У ИИ нет памяти о проекте: не описал структуру — получишь выдуманную.
- Шесть блоков: что это, стек, структура, зависимости, где что, чего не делать.
- Блок «чего не делать» экономит больше всего времени.
- Двадцать строк — предел; длиннее никто не прочитает, включая ИИ.
- Черновик может собрать ИИ, но проверяешь и правишь ты.
- «Перескажи архитектуру своими словами» — проверка за минуту.
- Обновляй справку при смене решения, а не при смене кода.
- Проси ИИ проверять собственный код по справке — это работает.

13.6 Вопросы перед стартом проекта

десять штук, полчаса, экономия вечера

ЧТО ЭТО

Завершение главы и единственный её практический итог: **список вопросов, на которые стоит ответить до первой строки кода**. Не потому, что так правильно, а потому, что каждый из них стоит дорого, если ответить на него через месяц.

Полчаса с этим списком — и запрос к ИИ превращается из «сделай мне бота» в мини-ТЗ (статья 3.15), на которое можно получить предсказуемый результат.

ДЕСЯТЬ ВОПРОСОВ

1. Что программа делает — одним предложением? Если предложение не складывается, проект ещё не придуман, а не «ещё не написан».

2. Что хранится? Список существительных. Из него получатся таблицы и модели (статья 13.1).

3. Где это хранится? Память, файл, SQLite, сервер БД — и признак, по которому выбранного перестанет хватать (статья 13.3).

4. Кто этим пользуется и сколько их? Один ты, десять знакомых или неизвестные люди из интернета. От ответа зависит всё: авторизация, нагрузка, обработка чужого ввода, цена ошибки.

5. Что нельзя потерять? Данные делятся на две кучки: те, что можно собрать заново, и те, что нельзя. Для второй кучки нужен ответ про резервные копии — в первый день, а не после первой потери.

6. Какие внешние сервисы нужны и что делать, когда они не отвечают? Любой внешний API однажды не ответит. Варианты: показать старые данные, показать ошибку, повторить позже. Выбрать надо заранее, иначе выберет случай.

7. Что меняется чаще всего? Текст сообщений? Правила расчёта? Внешний вид? То, что меняется часто, должно лежать отдельно и меняться без правки кода.

8. Сколько окружений и где секреты? Одна машина или машина плюс сервер; как туда попадают ключи (статья 13.4).

9. Как это запускается и кто перезапустит, если упадёт? Ответ «я вручную» — нормальный для скрипта на вечер и плохой для бота, которым пользуются люди.

10. Как я пойму, что оно сломалось? Логи, уведомление, проверка живости. Без ответа на этот вопрос о поломке сообщают пользователи — или не сообщают, а просто уходят.

> **ЗАПОМНИ**
> Половина вопросов — про отказы: что потеряется, что делать при недоступности внешнего сервиса, кто перезапустит, как узнать о поломке. Это не пессимизм: **разница между скриптом и программой в том, что программа продолжает работать, когда что-то пошло не так.**

КАК СКАЗАТЬ ИИ

Этот список удобно прогонять вместе с ИИ — он хорошо задаёт уточняющие вопросы, когда его об этом прямо просят (статья 3.11):

ЗАПРОС

```
Задумка: <опиши в двух-трёх предложениях>.
Задай мне по одному вопросу из списка и дождись ответа,
прежде чем переходить к следующему:
1) что делает одним предложением
2) что хранится
3) где хранится
4) кто пользуется и сколько их
5) что нельзя потерять
6) внешние сервисы и поведение при отказе
7) что меняется чаще всего
8) окружения и секреты
9) запуск и перезапуск
10) как узнаю о поломке

Код не пиши. В конце собери ответы в мини-ТЗ.
```

Получившееся мини-ТЗ — это и есть то, с чего начинается нормальный проект. Из него же собирается архитектурная справка (статья 13.5).

ЗАПРОС

```
Вот мои ответы на десять вопросов: <текст>.
Какие решения из них вытекают сами собой, а какие
я ещё не принял? Второе — списком, с вариантами
и одной строкой последствий на вариант.
```

НА ЧТО СМОТРЕТЬ

Вопросы заданы, ответы не записаны. Через неделю от них останется общее впечатление.

На вопросы отвечал ИИ. Он может предложить варианты, но выбор — твой (статья 1.1): это ровно те решения, которые потом не переделываются.

Ответ «потом разберусь» на вопросы 3, 5 и 8. Хранилище, резервные копии и секреты «потом» обходятся дороже всего.

Мини-ТЗ написано, но в запросах не используется. Тогда оно не ТЗ, а дневник.

ШПАРГАЛКА

- Полчаса на десять вопросов экономят вечер переписывания.
- Не складывается предложение о том, что делает программа, — проект не придуман.
- Для каждого выбора фиксируй признак, по которому он перестанет работать.
- Половина вопросов — про отказы, и это главное отличие программы от скрипта.
- Что меняется часто — выносится из кода.
- Отвечает человек; ИИ задаёт вопросы и собирает ответы в мини-ТЗ.
- Ответы записываются в репозиторий и превращаются в справку из статьи 13.5.

14 Практикум

Пять проектов от фразы «хочу сделать» до работающей программы. Здесь всё предыдущее собирается вместе: вопросы перед стартом, мини-ТЗ, плохой запрос против хорошего, ревью, настоящий баг и его починка. Код в кейсах запускался, ошибки в них — те, что действительно случились.

14.1 Кейс — телеграм-бот с внешним API

полный цикл, два настоящих бага

ЧТО ЭТО

Первый кейс — и заодно образец того, как устроены остальные: идея, вопросы, мини-ТЗ, запрос, план, код, ревью, баг, починка. Ничего нового по приёмам, всё уже было в книге; новое — то, что они идут подряд, как в жизни.

Идея в одну фразу: **бот, который присылает утром погоду в выбранных городах.**

ШАГ 1. ДЕСЯТЬ ВОПРОСОВ

Полчаса по списку из статьи 13.6. Коротко, только ответы:

Вопрос	Ответ
что делает	присылает погоду в выбранных городах каждое утро
что хранится	подписки: кто на какой город
где хранится	SQLite, записей мало, пишет один процесс
кто пользуется	я и десяток знакомых
что нельзя потерять	список подписок; погоду — можно, она обновится
внешние сервисы	один API погоды; не ответил — показать прошлые данные
что меняется чаще	тексты сообщений
окружения	моя машина и один сервер; ключи в `.env`

запуск	systemd, перезапуск сам
как узнаю о поломке	логи в journal плюс сообщение себе при падении рассылки

Шестая строчка — та самая, ради которой список и нужен. Без неё бот однажды промолчит утром, и будет непонятно, он сломался или погода не изменилась.

ШАГ 2. ЗАПРОС

Как не надо:

ТАК НЕ НАДО

```
Напиши телеграм-бота на питоне, который показывает погоду.
```

Что придёт: один файл, синхронный `requests` внутри асинхронного обработчика, ключ API строкой в коде, список городов в глобальной переменной. Работать будет — ровно до перезапуска.

Как надо (это и есть мини-ТЗ из статьи 3.15):

ТАК ЛУЧШЕ

```
Бот на aiogram 3, Python 3.12. Хранение — SQLite.
Функции: /add <город>, /list, /remove <город>, утренняя рассылка.

Структура:
bot/handlers/  только разбор ввода и ответ
bot/services/  логика и поход в API погоды
bot/storage.py весь доступ к базе, больше нигде
bot/config.py  настройки через pydantic-settings

Правила: handlers не обращаются к базе напрямую;
никаких новых библиотек без моего согласия;
ключ API только из переменной окружения.

Сначала напиши план: какие файлы и что в каждом.
Код после моего «да».
```

Последние две строки — приём из статьи 3.10. План читается минуту, а переписывать его дешевле, чем код.

ШАГ 3. ХРАНИЛИЩЕ И ПЕРВЫЙ БАГ

Начинаем снизу, со слоя данных (статья 13.1). Вот что пришло:

```
def connect():
    db = sqlite3.connect(DB)
    db.execute("""
        CREATE TABLE IF NOT EXISTS subscriptions (
            user_id INTEGER NOT NULL,
            city    TEXT    NOT NULL
        )
    """)
    db.commit()
    return db


def add_city(db, user_id: int, city: str) -> None:
    db.execute("INSERT INTO subscriptions VALUES (?, ?)", (user_id,
city))
    db.commit()
```

Код читается, работает и выглядит правильным. Пишем тесты — не для галочки, а по списку из статьи 3.9: обычный случай, границы, то, что забывают.

```
def test_add_and_list(db):
    storage.add_city(db, 1, "Oslo")
    assert storage.list_cities(db, 1) == ["Oslo"]


def test_cities_are_per_user(db):
    storage.add_city(db, 1, "Oslo")
    storage.add_city(db, 2, "Bergen")
    assert storage.list_cities(db, 1) == ["Oslo"]


def test_same_city_twice(db):
    storage.add_city(db, 1, "Oslo")
    storage.add_city(db, 1, "Oslo")
    assert storage.list_cities(db, 1) == ["Oslo"], "город не должен
добавиться дважды"
```

Запуск:

```
E       AssertionError: город не должен добавиться дважды
E       assert ['Oslo', 'Oslo'] == ['Oslo']
E         Left contains one more item: 'Oslo'

1 failed, 3 passed in 0.05s
```

Третий тест упал. Добавил город дважды — получил две подписки, а значит, и два сообщения каждое утро. Руками это не находится: кто же добавляет один и тот же город дважды? Пользователь добавляет — через месяц, забыв, что уже добавлял.

Починка — там, где причина, а не там, где симптом. Не проверка в коде перед вставкой (её можно забыть вызвать из другого места), а **правило в самой таблице**:

```
            user_id INTEGER NOT NULL,
            city    TEXT    NOT NULL,
            UNIQUE (user_id, city)
```

```
    db.execute("INSERT OR IGNORE INTO subscriptions VALUES (?, ?)",
               (user_id, city))
```

```
4 passed in 0.04s
```

> **ЗАПОМНИ**
> Правило, записанное в схеме таблицы, работает всегда. Проверка, записанная в коде, работает только там, где её не забыли. Если условие важно — ему место в базе.

ШАГ 4. ВНЕШНИЙ API И ВТОРОЙ БАГ

Поход за погодой в первом варианте выглядел так:

```
def fetch(city: str) -> float:
    r = requests.get(API_URL, params={"q": city, "appid":
settings.api_key})
    return r.json()["main"]["temp"]
```

Две проблемы в двух строках, и обе типичные (статья 2.14): разбор ответа без проверки — и **нет таймаута**.

Проверим второе. Поднимем местный сервер, который «подвис» — принимает запрос и молчит:

```
class Slow(BaseHTTPRequestHandler):
    def do_GET(self):
        time.sleep(30)
        ...
```

```
без timeout: всё ещё висит через 12 с  (прервали сами)
с timeout=2: requests.Timeout через 2.0 с
```

Двенадцать секунд — это сколько мы согласились ждать, прежде чем прервать проверку. Настоящий зависший сервер держит соединение сколько угодно, и вместе с ним висит **вся утренняя рассылка**: один недоступный город — и остальные подписчики не получают ничего.

У `requests` таймаута по умолчанию нет. Ни одного. Это свойство библиотеки, о котором ИИ вспоминает через раз.

Исправленная версия — с таймаутом, с проверкой и с ответом на вопрос «что делать, если не ответил» из мини-ТЗ:

```
def fetch(city: str) -> float | None:
    try:
        r = requests.get(API_URL, params={...}, timeout=5)
        r.raise_for_status()
    except requests.RequestException:
        return None                        # вызывающий покажет прошлые
данные
    data = r.json()
    main = data.get("main")
    if not isinstance(main, dict) or "temp" not in main:
        return None
    return main["temp"]
```

Проверено на местном сервере — настоящий город, выдуманный город и выключенный сервер:

```
Oslo        -> 4.2
Выдуманск   -> None
сервера нет -> None
```

Ни одного падения: на все три случая у функции есть ответ.

Осталась одна деталь, и она та самая, с которой начинался шаг 2: `fetch` синхронная, а бот асинхронный. Вызванная напрямую из хендлера или рассылки, она на время запроса остановит весь бот — никто не получит ответа, пока не придёт погода. Поэтому вызывается она в отдельном потоке:

```
temp = await asyncio.to_thread(fetch, city)
```

Другой путь — переписать `fetch` на `aiohttp`. Оба честные; главное, чтобы синхронный запрос не оказался внутри `async def` без такой обёртки.

ШАГ 5. ЧЕГО В ЭТОМ КЕЙСЕ НЕТ

В кейсе прогонялись слой данных (четыре теста) и поведение при зависшем сервере (местный сервер, настоящие замеры). Обработчики и запуск против настоящего Telegram и настоящего API погоды здесь не показаны: это следующий шаг, и делается он по чек-листу из статьи 6.6, а не на веру.

Это ровно та честность, которой книга требует от ИИ (статья 1.6), и к себе она применяется так же.

ШПАРГАЛКА

- Порядок: вопросы → мини-ТЗ → план → код снизу вверх → тесты → запуск.
- Начинай со слоя данных: на нём стоит всё остальное.
- Тест «то же самое дважды» находит баг, который руками не находится.
- Условие, важное для данных, живёт в схеме таблицы, а не в коде.
- У `requests` нет таймаута по умолчанию — ставь его всегда.
- Один зависший запрос в рассылке останавливает рассылку целиком.
- Синхронный запрос из асинхронного бота — только через `asyncio.to_thread`.
- «Что делать, если сервис не ответил» решается в мини-ТЗ, а не в момент сбоя.

14.2 Кейс — веб-API с базой данных

и почему 500 вместо 409 — это баг

ЧТО ЭТО

Продолжение первого кейса. У бота из статьи 14.1 появляются соседи: сайт и мобильное приложение. Держать список городов в трёх местах нельзя, значит, нужен **один API, из которого берут все**.

Идея в одну фразу: **сервис, который хранит города пользователя и отдаёт их по HTTP.**

ШАГ 1. МИНИ-ТЗ

Вопросы те же (статья 13.6), поэтому только то, что отличается от бота:

Вопрос	Ответ
кто пользуется	бот, сайт, приложение — не люди, а программы
что хранится	города и последняя известная температура
где хранится	SQLite: один процесс, записей мало (статья 13.3)
что нельзя потерять	список городов
как узнаю о поломке	коды ответов в логах

Из строчки «пользуются программы» следует главное требование: **ответы должны быть предсказуемыми**. Программа не умеет

догадываться, что значит «что-то пошло не так», — ей нужен код ответа, по которому видно, что делать.

ШАГ 2. ЗАПРОС

```
ТАК ЛУЧШЕ
API на FastAPI, Python 3.12, хранение в SQLite.

Эндпоинты:
GET    /cities        список городов
POST   /cities        добавить {name, temp}
DELETE /cities/{name} удалить

Требования:
- схемы запроса и ответа на pydantic, имя города 1–60 символов
- весь доступ к базе в одном месте
- коды ответов: 201 на создание, 204 на удаление,
  404 если удалять нечего, 409 если город уже есть
- никаких ORM, запросы пишем руками

Сначала список файлов и что в каждом. Код после «да».
```

Строка про коды ответов — самая важная. Без неё придёт код, который на всё отвечает 200, а об ошибках сообщает текстом внутри ответа.

ШАГ 3. ЧТО ПРИШЛО

Структура получилась правильная, схемы на месте, запросы параметризованные (это важно — подстановка строкой открывает SQL-инъекцию, статья 7.8). Обработчик добавления:

```
@app.post("/cities", response_model=CityOut, status_code=201)
def add_city(city: CityIn):
    with closing(db()) as conn:
        conn.execute("INSERT INTO cities VALUES (?, ?)", (city.name,
city.temp))
        conn.commit()
    return city
```

Запускаем и проверяем руками — именно так, как советует статья 4.7: не только удачный путь.

```
$ curl -X POST localhost:8102/cities -d '{"name":"Oslo","temp":4}'
201

$ curl -X POST localhost:8102/cities -d '{"name":"Oslo","temp":4}'
Internal Server Error
HTTP 500
```

Второй раз — 500. В логах причина:

```
sqlite3.IntegrityError: UNIQUE constraint failed: cities.name
```

Что произошло: в схеме таблицы `name` объявлен первичным ключом — и это правильно, повторов быть не должно (урок из статьи 14.1). Но обработчик об этом не знает, исключение вылетает наружу, и FastAPI превращает его в 500.

> **ЗАПОМНИ**
> **500 — это всегда баг.** Он означает «программа не предусмотрела такой случай». Ожидаемая ситуация — повтор, отсутствие, нет прав — имеет свой код: 409, 404, 403. Если на действие пользователя можно получить 500, значит, случай не продуман, а не «пользователь сделал странное».

Починка — четыре строки:

```
        try:
            conn.execute("INSERT INTO cities VALUES (?, ?)", (city.name,
city.temp))
        except sqlite3.IntegrityError:
            raise HTTPException(409, f"город {city.name} уже добавлен")
```

ШАГ 4. ПРИЁМКА

Проверяем все случаи из мини-ТЗ подряд — и удачные, и нет:

```
первый POST   : 201
второй POST   : {"detail":"город Oslo уже добавлен"} | HTTP 409
GET           : [{"name":"Oslo","temp":4.0}]
DELETE Oslo   : 204
DELETE снова : 404
```

И проверка данных, которую никто не писал, — она досталась от схемы:

```
$ curl -X POST localhost:8102/cities -d '{"name":"","temp":4}'
422
```

```
{"detail":[{"type":"string_too_short","loc":["body","name"],
 "msg":"String should have at least 1 character","ctx":
{"min_length":1}}]}
```

В ответе названо поле, названо правило, названо нарушение. Это и есть смысл схем из статьи 7.3: проверки пишутся один раз рядом с описанием данных, а не в каждом обработчике.

Пять строчек вывода выше — это пять проверок, каждая из которых соответствует строке мини-ТЗ. Такой прогон занимает минуту и заменяет вопрос «а точно работает?».

ШАГ 5. ЧЕГО ЗДЕСЬ НАМЕРЕННО НЕТ

Нет авторизации. Любой, кто знает адрес, может добавить город. Для сервиса на своей машине это осознанное решение, для сервиса в интернете — дыра. Решение записано в мини-ТЗ, а не забыто, и в этом вся разница.

Нет пагинации. Городов десятки. Когда станут тысячи — появится, и признак назван заранее (статья 13.3).

Нет ORM. Запросов пять, все простые. ORM — это ещё одна библиотека, ещё один слой и ещё один источник непонятных ошибок; на пяти запросах она не окупается. Если бы запросов было пятьдесят, ответ был бы другим.

ШПАРГАЛКА

- Если API пользуются программы, главное требование — предсказуемые коды.
- Коды перечисляй в мини-ТЗ: без этого получишь 200 на всё.
- 500 — это всегда баг: случай не предусмотрен.
- Повтор — 409, нечего удалять — 404, кривые данные — 422.
- Правило в схеме таблицы ловит повтор, но ответ за код ответа на обработчике.
- Проверки данных пишутся один раз в схеме, а не в каждом обработчике.
- Приёмка — прогон всех строк мини-ТЗ подряд, включая неудачные случаи.
- Чего нет в проекте — должно быть решением, а не забывчивостью.

14.3 Кейс — сайт на React к этому API

работает в curl, не работает в браузере

ЧТО ЭТО

Третий кейс продолжает второй: API из статьи 14.2 уже работает, теперь к нему нужен сайт. Задача выглядит простой — показать список, — и именно на ней вылезает ошибка, которой нет ни в одном руководстве по React, потому что она не про React.

ШАГ 1. ЗАПРОС

ТАК ЛУЧШЕ

```
React 19 + TypeScript, сборка Vite.
Один экран: список городов с температурой.
Данные: GET http://localhost:8102/cities,
ответ [{name: string, temp: number | null}].
```

```
Нужны все четыре состояния: загрузка, ошибка,
пустой список, данные. Тип ответа опиши явно,
any и as не использовать.
Стили простые, без библиотек компонентов.
```

Требование про четыре состояния — из статьи 10.5, и оно здесь не для галочки: именно оно спасёт на следующем шаге.

ШАГ 2. ЧТО ПРИШЛО

```tsx
type City = { name: string; temp: number | null }

export default function App() {
  const [cities, setCities] = useState<City[]>([])
  const [error, setError] = useState<string | null>(null)
  const [loading, setLoading] = useState(true)

  useEffect(() => {
    fetch('http://localhost:8102/cities')
      .then((r) => {
        if (!r.ok) throw new Error(`HTTP ${r.status}`)
        return r.json()
      })
      .then(setCities)
      .catch((e) => setError(String(e)))
      .finally(() => setLoading(false))
  }, [])

  if (loading) return <p className="state">Загружаем…</p>
  if (error) return <p className="state err">Не удалось загрузить:
{error}</p>
  if (cities.length === 0) return <p className="state">Городов пока
нет</p>

  return (
    <main>
      <h1>Погода</h1>
      <ul>
        {cities.map((c) => (
          <li key={c.name}>
            <span>{c.name}</span>
            <b>{c.temp === null ? 'нет данных' : `${c.temp}°`}</b>
          </li>
        ))}
      </ul>
    </main>
  )
}
```

Код правильный. Зависимости у эффекта пустые, `key` на месте, четыре состояния есть, `any` нет. Собирается:

```
✓ built in 1.82s
```

ШАГ 3. БАГ, КОТОРОГО НЕТ В КОДЕ

Открываем в браузере:

```
Не удалось загрузить: TypeError: Failed to fetch
```

При этом API жив и прекрасно отвечает:

```
$ curl -s -o /dev/null -w "%{http_code}" localhost:8102/cities
200
```

Работает в `curl`, не работает в браузере. Причина — в консоли разработчика (статья 8.4), и только там:

```
Access to fetch at 'http://localhost:8102/cities' from origin
'http://localhost:5190' has been blocked by CORS policy:
No 'Access-Control-Allow-Origin' header is present on the requested
resource.
```

Это **CORS** — правило браузера, по которому страница с одного адреса не может читать ответы с другого, если тот явно не разрешил. У `curl` такого правила нет: оно существует только в браузере и только ради защиты пользователя.

Два вывода, которые стоят этой статьи.

Первый: ошибка на экране не называет причину. Браузер намеренно не отдаёт подробности коду страницы — `fetch` получает только `TypeError: Failed to fetch`. Искать надо в консоли, а не в коде React. Запросы вида «почини TypeError: Failed to fetch» бесполезны: в показанном коде ошибки нет.

Второй: чинить надо на сервере. CORS настраивается там, откуда берут данные:

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5190"],   # адрес сайта, не "*"
    allow_methods=["GET", "POST", "DELETE"],
    allow_headers=["Content-Type"],
)
```

После перезапуска API заголовок появляется:

```
$ curl -D- -H "Origin: http://localhost:5190" localhost:8102/cities
access-control-allow-origin: http://localhost:5190
```

И страница оживает — ошибок в консоли нет:

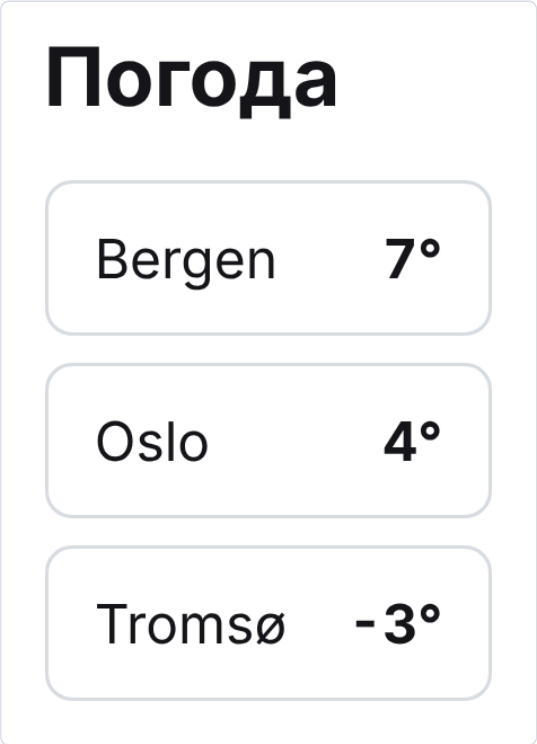


Рис. 14.1 · Три города пришли из API статьи 14.2. Снимок сделан в браузере после настройки CORS.

> **ВНИМАНИЕ**
> Почти любой ИИ на вопрос про CORS предложит `allow_origins=["*"]`. Это работает и для учебного проекта терпимо, но означает «читать мои ответы можно откуда угодно». Если API отдаёт что-то, кроме публичных данных, адреса перечисляют поимённо. Разница между `"*"` и списком — это разница между «мне нечего скрывать» и «я не подумал».

ШАГ 4. ЧТО ПРОВЕРЯТЬ ГЛАЗАМИ

Собранный сайт — не то же самое, что сайт при разработке (глава 12). Минимум проверок перед тем, как считать кейс закрытым:

- **Четыре состояния.** Выключить API — должна появиться строка про ошибку, а не пустой экран. Это и была проверка на шаге 3, просто вынужденная.
- **Узкое окно.** Сузить до ширины телефона.
- **Собранная версия.** `vite build` и открыть результат, а не только режим разработки.
- **Адрес API не вшит намертво.** На сервере он другой; значит, переменная окружения — и помнить, что она попадёт в артефакт открытым текстом (статья 12.1).

ШПАРГАЛКА

- «Работает в curl, не работает в браузере» — почти всегда CORS.
- Настоящая причина видна только в консоли браузера, а не на экране.
- `TypeError: Failed to fetch` — это не ошибка в коде React.
- CORS чинится на сервере, который отдаёт данные, а не на сайте.
- `allow_origins=["*"]` — не решение, а отказ от решения.
- Четыре состояния экрана окупились в первый же день.
- Проверять надо собранную версию и узкое окно, а не только режим разработки.

14.4 Кейс — мобильное приложение

тот же API, другой разговор

ЧТО ЭТО

Четвёртый потребитель того же API: приложение на Flutter. По смыслу экран тот же, что в статье 14.3, — список городов с температурой, — но разговор с ИИ получается другим, и различия стоит увидеть рядом.

ВНИМАНИЕ

Как и глава 11, этот кейс написан на уровне чтения. Здесь нет листингов и измерений — есть планирование, запросы и разбор того, что в ответах придётся проверить. Код этого кейса — единственное место в книге, которое надо проверить самому.

ШАГ 1. ЧТО МЕНЯЕТСЯ В МИНИ-ТЗ

Три строки по сравнению со статьёй 14.2 — и каждая тянет за собой работу:

Адрес API. `localhost` в приложении на телефоне указывает на сам телефон, а не на твою машину. При разработке нужен адрес машины в локальной сети, на боевом — настоящий домен. Это первое, обо что спотыкаются, и ИИ об этом не предупреждает, потому что не знает, где ты запускаешь.

Сеть пропадает. У сайта это редкость, у телефона — норма (статья 11.5). Значит, последние данные надо хранить на устройстве и показывать их с пометкой «обновлено тогда-то», а не пустой экран.

Обновление не мгновенное. Сайт обновляется при перезагрузке страницы, приложение — когда пользователь согласится (ста-

тья 12.3). Значит, API не может ломать совместимость: поле `temp` останется `temp` навсегда.

ШАГ 2. ЗАПРОС

ТАК ЛУЧШЕ

```
Flutter 3.38, Dart 3.10. Состояние — provider 6,
других библиотек состояния не добавлять.

Один экран: список городов с температурой.
Данные: GET http://<адрес>/cities,
ответ [{"name": String, "temp": double?}].

Нужно:
- модель City с fromJson, без приведения типов,
  temp честно nullable
- четыре состояния: загрузка, ошибка, пусто, данные
- адрес API вынести в настройку, не вшивать
- проверить вёрстку на ширине 360 и с длинным названием

Сначала список файлов и что в каждом. Код после «да».
```

Сравни с запросом из статьи 14.3: половина строк совпадает дословно. Четыре состояния, явные типы, адрес в настройке — это не свойства React или Flutter, а свойства **задачи «показать данные из сети»**. Поэтому хороший запрос переносится между стеками почти целиком.

Отличаются три вещи, и все три — про платформу: версии (здесь их называют обязательно), библиотека состояния и проверка на узком экране.

ШАГ 3. ЧТО ПРОВЕРЯТЬ В ОТВЕТЕ

Поскольку запустить нельзя, остаётся то, что можно прочитать. По порядку важности:

1. Версия API. Главная проблема стека (статья 11.2). Устаревшие имена виджетов и параметров; `flutter analyze` обычно находит.

2. `!`, `late`, `dynamic` в разборе JSON. Сервер может ответить ошибкой, и `json['temp'] as double` уронит приложение. В модели должно быть честное `double?` и проверка, а не приведение типа.

3. Четыре состояния — все ли. Самое частое: есть «данные пришли», нет остального.

4. Запросы в `build`. Поход в сеть принадлежит слою данных, а не методу построения (статья 11.4).

5. `dispose` у контроллеров. Забытая уборка.

6. Фиксированные размеры. Переполнение на узком экране.

Это тот же список, что в статье 11.6, — здесь он просто применён к конкретной задаче.

ШАГ 4. РАЗГОВОР, КОТОРЫЙ СТОИТ ЗАВЕСТИ ЗАРАНЕЕ

ЗАПРОС

```
Я собираюсь выпустить это приложение в магазин.
Перечисли, что надо решить до первой публикации
и что после неё поменять уже нельзя.
```

Ответ на этот вопрос разобран в статье 12.2, и он ровно та причина, по которой мобильный кейс стоит планировать дольше остальных: сайт переделывается за вечер, а идентификатор приложения — никогда.

ШАГ 5. ЧЕСТНЫЙ ИТОГ

Из четырёх потребителей API (бот, сайт, приложение, скрипт) этот — самый дорогой: отдельный стек, отдельная сборка, отдельная публикация, невозможность откатиться. При этом экран у него тот же, что у сайта.

Вопрос, который стоит задать себе до начала, а не после: **нужно ли это приложение, или хватит сайта, открытого на телефоне?** Для списка городов с температурой ответ чаще «хватит сайта». Приложение оправдано, когда нужно то, чего у сайта нет: работа без сети, уведомления, камера, фоновая работа.

Это и есть архитектурное решение из главы 13 — то самое, которое ИИ примет за тебя, если не принять его самому.

ШПАРГАЛКА

- Хороший запрос переносится между стеками почти дословно.
- Различается только платформенное: версии, библиотека состояния, узкий экран.
- `localhost` в телефоне — это сам телефон, а не твоя машина.
- У мобильного приложения сеть пропадает постоянно: нужны прошлые данные.
- Старые версии приложения живут долго — API не может ломать совместимость.
- Читать ответ по списку из статьи 11.6: версия, `!/late`, состояния, build.
- Перед мобильным кейсом честно ответь, нужен ли он: сайт дешевле во всём.

14.5 Кейс — скрипт на один вечер

самый опасный жанр

ЧТО ЭТО

Последний кейс — самый маленький и самый опасный. Задача на двадцать строк: **разложить папку с фотографиями, переименовав файлы по дате съёмки**.

Опасность у таких скриптов не в сложности, а в том, что к ним не относятся серьёзно. Мини-ТЗ не пишут, тесты не пишут, ревью не делают — «это же просто переименование». И запускают сразу на настоящей папке.

ШАГ 1. ЧТО ПРИШЛО

ТАК НЕ НАДО

```
Напиши скрипт на питоне, который переименует все файлы
в папке photos по дате изменения, формат ГГГГ-ММ-ДД.
```

```python
import os
from datetime import datetime
from pathlib import Path

folder = Path("photos")

for f in folder.iterdir():
    date = datetime.fromtimestamp(f.stat().st_mtime)
    new_name = date.strftime("%Y-%m-%d") + f.suffix
    f.rename(folder / new_name)
```

Девять строк, читаются целиком, делают ровно то, что просили. Трудно представить, что здесь можно сломать.

ШАГ 2. ЧТО ОН СДЕЛАЛ

В папке было четыре файла:

```
IMG_1001.jpg  "first photo"    снято 14 марта
IMG_1002.jpg  "second photo"   снято 14 марта
IMG_1003.jpg  "third photo"    снято 15 марта
notes.txt     "not a photo"
```

После запуска:

```
2026-03-14.jpg  "second photo"
2026-03-15.jpg  "third photo"
2026-03-15.txt  "not a photo"
```

Файлов стало три. `first photo` больше нет нигде.

Две фотографии одного дня получили одно и то же имя, и вторая молча записалась поверх первой: `rename` в Python не спрашивает разрешения и не предупреждает. Заодно переименовался `notes.txt`, который никто не просил трогать.

Ни ошибки, ни предупреждения, ни кода возврата. Скрипт отработал успешно.

> **ЗАПОМНИ**
> У скрипта, который меняет файлы, **нет отмены**. Нет истории, как в git, нет корзины, нет «предыдущей версии». Поэтому цена ошибки здесь выше, чем в приложении, хотя кода в двадцать раз меньше (статья 1.5).

ШАГ 3. КАК НАДО БЫЛО ПРОСИТЬ

ТАК ЛУЧШЕ

```
Скрипт на Python 3.12: переименовать фотографии в папке
по дате изменения файла в формат ГГГГ-ММ-ДД.

Обязательно:
- по умолчанию НИЧЕГО не менять, только показать план
  «старое имя -> новое»; выполнять по флагу --apply
- если на одну дату несколько файлов: -2, -3 и так далее
- если целевое имя уже занято файлом не из этого списка —
  остановиться и ничего не делать
- трогать только .jpg/.jpeg/.png/.heic, остальное
  пропустить и перечислить
- папку принимать аргументом, не вшивать
```

Шесть строк требований на девять строк кода. Выглядит избыточно ровно до первой потерянной фотографии.

Главная из них — первая. **Режим показа по умолчанию** превращает опасный скрипт в безопасный: сначала смотришь, что произойдёт, потом разрешаешь.

ШАГ 4. ЧТО ПОЛУЧИЛОСЬ

```
$ python3 rename.py photos
IMG_1001.jpg -> 2026-03-14.jpg
IMG_1002.jpg -> 2026-03-14-2.jpg
IMG_1003.jpg -> 2026-03-15.jpg
пропущено (не фото): notes.txt

ничего не изменено. файлов к переименованию: 3
повторите с --apply, чтобы выполнить
```

План видно целиком, и видно главное: два файла от 14 марта получат **разные** имена. Теперь можно разрешать:

```
$ python3 rename.py photos --apply
...
переименовано: 3
```

Итог:

```
2026-03-14.jpg    "first photo"
2026-03-14-2.jpg  "second photo"
2026-03-15.jpg    "third photo"
notes.txt         "not a photo"
```

Четыре файла вошли, четыре вышли. Ничего не потеряно, `notes.txt` не тронут.

ШАГ 5. ПРАВИЛО ДЛЯ ВСЕГО ЖАНРА

Скрипты, которые что-то меняют снаружи — файлы, базу, письма, сообщения, — отличаются от остального кода одним: **ошибку нельзя откатить**. Поэтому для них есть короткий список, который стоит применять всегда:

1. **Режим показа по умолчанию.** Выполнение — по явному флагу.
2. **Сначала на копии.** `cp -r photos photos-проба` занимает секунду.
3. **Никогда не перезаписывать молча.** Совпало имя — остановиться.
4. **Фильтр по типу.** «Все файлы в папке» — почти никогда не то, что имелось в виду.
5. **Отчёт в конце.** Сколько обработано, сколько пропущено и почему.

6. **Папка — аргумент, а не константа.** Константу однажды забудут поменять.

Ни один пункт не требует усилий. Все шесть вместе — это разница между инструментом и рулеткой.

ШПАРГАЛКА

- Самый опасный код — самый короткий: к нему не относятся серьёзно.
- `rename` перезаписывает молча: два файла с одной датой — минус один файл.
- У скрипта, меняющего файлы, нет отмены. Ни истории, ни корзины.
- Режим показа по умолчанию — главная защита, и стоит одну строку требований.
- «Все файлы в папке» почти никогда не означает то, что ты имел в виду.
- Первый запуск — на копии папки, а не на настоящей.
- Шесть строк требований к девяти строкам кода — нормальная пропорция.

Приложения

Справочная часть. Сюда заглядывают, а не читают подряд: шаблоны запросов на одном месте, чек-лист ревью, расшифровка частых ошибок, команды git и терминала, словарь английских терминов.

A Шаблоны запросов

всё, что есть в книге, на одном месте

ЧТО ЭТО

Шаблоны из глав 3, 13 и 14, собранные вместе. Подставь своё в угловые скобки. Для каждого указана статья, где разобрано, почему он устроен именно так.

РАБОТА С КОДОМ

Новая функция (статья 3.4):

```
ЗАПРОС
Проект: <стек, версии>. Файл: <путь>.
Нужна функция <имя>: принимает <что>, возвращает <что>.
Крайние случаи: <пусто / ошибка сети / нет данных>.
Ограничения: новых библиотек не добавляй, стиль как в файле.
Покажи только изменённый фрагмент.
```

Исправить баг (статья 3.5):

```
ЗАПРОС
Что делаю: <шаги>.
Что ожидаю: <результат>. Что получаю: <результат>.
Ошибка целиком: <текст с трейсбеком>.
Версии: <язык, библиотеки>.
Сначала скажи, в чём причина. Правку — после моего «да».
```

Рефакторинг (статья 3.6):

```
ЗАПРОС
Вот код: <фрагмент>.
Поведение менять нельзя. Нужно: <что именно улучшить>.
Не трогай: <файлы, функции>.
Покажи дифф, а не файл целиком, и объясни каждое изменение одной
строкой.
```

Объясни этот код (статья 3.7):

ЗАПРОС

```
Объясни этот код построчно, как человеку, который знает
основы, но не знает эту библиотеку.
Отдельно: что здесь может сломаться и при каких данных.
```

Ревью (статья 3.8):

ЗАПРОС

```
Сделай ревью: <код>.
Ищи: необработанные ошибки, крайние случаи, утечки секретов,
лишние зависимости, места, где поведение неочевидно.
Формат: список, по строке на находку, с номером строки.
Не переписывай код.
```

Тесты (статья 3.9):

ЗАПРОС

```
Напиши тесты на <функция/модуль>, pytest.
Покрой: обычный случай, границы, пустой ввод, ошибку.
Без моков там, где можно без них.
Каждый тест — одна проверка, имя теста описывает случай.
```

ПРИЁМЫ

Сначала план (статья 3.10):

ЗАПРОС

```
Задача: <описание>.
Не пиши код. Сначала план: какие файлы затронешь,
что в каждом изменится, в каком порядке.
Код после моего «да».
```

Пусть ИИ задаёт вопросы (статья 3.11):

ЗАПРОС

```
Прежде чем писать код, задай мне вопросы, ответы
на которые изменят решение. По одному за раз.
Если вопросов нет — скажи, какие допущения ты делаешь.
```

Ограничить зону правок (статья 3.13):

ЗАПРОС

```
Меняй только файл <путь> и только функцию <имя>.
Остального не касайся. Если для решения нужно тронуть
что-то ещё — скажи об этом, но не делай.
```

ПРОЕКТ ЦЕЛИКОМ

Десять вопросов (статья 13.6):

ЗАПРОС

```
Задумка: <две-три фразы>.
Задай мне по одному вопросу и дождись ответа:
что делает одним предложением; что хранится; где хранится;
кто пользуется и сколько их; что нельзя потерять;
```

```
внешние сервисы и поведение при отказе; что меняется чаще;
окружения и секреты; запуск и перезапуск; как узнаю о поломке.
Код не пиши. В конце собери ответы в мини-ТЗ.
```

Разбиение на части (статья 13.1):

ЗАПРОС

```
Задумка: <описание>. Не пиши код. Разложи на части:
1) что хранится — сущности с полями
2) что происходит — действия
3) что видно снаружи — экраны или адреса
Для каждого действия укажи, какие сущности оно трогает.
```

Архитектурная справка (статья 13.5):

ЗАПРОС

```
Вот дерево проекта и ключевые файлы.
Собери черновик справки по шести блокам: что это, стек,
структура, зависимости, где что лежит, чего не делать.
Где не уверен — пиши «?», не придумывай.
```

Проверка кода по справке (статья 13.5):

ЗАПРОС

```
Вот справка и код, который ты написал.
Проверь свой код на соответствие справке: зависимости,
расположение файлов, запрет на новые библиотеки.
Нарушения — списком.
```

ПЕРЕД ВЫПУСКОМ

Что ломается при сборке (статья 12.1):

ЗАПРОС

```
Проект собирается у меня, но падает после установки.
Перечисли, чем запуск при разработке отличается от артефакта
в этом проекте: пути, переменные окружения, ресурсы, версии.
По пункту на строку.
```

Правка с ценой (статья 12.1):

ЗАПРОС

```
Сборка падает: <ошибка>. Нужны два варианта:
правка кода и правка настроек сборки.
Для каждого скажи, что перестанет проверяться.
Выбирать буду я.
```

Совместимость при обновлении (статья 12.3):

ЗАПРОС

```
Я меняю <поле / схему / эндпоинт>.
Старые клиенты обновляются не сразу.
Что у них сломается и как сделать это изменение без поломки?
Отдельно: какая часть не откатывается.
```

Чек-лист публикации (статья 12.4):

ЗАПРОС

```
Составь чек-лист публикации под мой проект.
Стек: <перечисли>. Публикуется: <куда>.
Раздели на «один раз на проект» и «перед каждым выпуском».
Только моё, без общих советов.
```

Б Чек-лист ревью кода

двенадцать пунктов на один экран

ЧТО ЭТО

Список, по которому читается любой сгенерированный кусок кода, прежде чем он попадёт в проект. Подробности — в главах 2 и 4; здесь только порядок.

СНАЧАЛА — ТРИ ВОПРОСА

1. **Я понимаю, что делает каждая строка?** Если нет — не вставлять, а спрашивать (статья 2.1).
2. **Это решает мою задачу или похожую?** Самая частая подмена: решена задача, которую ИИ понял, а не та, которую ставили (статья 3.1).
3. **Что будет, если это сломается?** От ответа зависит, сколько времени тратить на остальные пункты (статья 1.5).

ДВЕНАДЦАТЬ ПУНКТОВ

1. Ошибки. Есть ли обработка там, где что-то может не получиться: сеть, файл, разбор данных, деление. Пустой `except` — красный флаг.

2. Крайние случаи. Пустой список, `None`, ноль, отрицательное число, очень длинная строка, два одинаковых значения.

3. Внешние данные проверяются. Ответ API, ввод пользователя, содержимое файла — ничего из этого не разбирается без проверки (статья 7.3).

4. Секреты. Ключей, паролей и токенов в коде нет. Ни «временно», ни «для примера» (статья 5.7).

5. Новые зависимости. Появилась библиотека, которой не просили? Спросить зачем. Часто её можно не добавлять.

6. Зона правок. Изменено только то, о чём просили. Переписанные «заодно» куски — повод откатить и повторить запрос (статья 3.13).

7. Файлы и пути. Относительные пути к данным, жёстко вшитые адреса и `/home/user/...` в коде (статьи 12.1, 13.4).

8. Повторы. То же действие появилось в третьем месте — пора выносить.

9. Названия. Имена говорят, что внутри. `data2`, `temp_fix`, `handle` обычно означают, что автор сам не знал.

10. Зависимости слоёв. Запрос к базе из обработчика экрана, импорт наверх (статья 13.1).

11. Что удалено. В диффе смотреть не только добавленное, но и минусы: что исчезло вместе с правкой (статья 2.13).

12. Проверки на месте. Не отключены ли линтер, типы, тесты ради того, чтобы «прошло» (статья 2.14).

ПОТОМ — ЗАПУСТИТЬ

Чтение не заменяет запуска, запуск не заменяет чтения. Минимум:

- запустить и пройти **свой** сценарий руками;
- пройти **неудачный** сценарий: выключить сеть, подать пустое значение;
- прогнать тесты, если они есть, и прочитать, что именно они проверяют (статья 4.6);
- посмотреть на изменённые файлы в `git diff` целиком, перед коммитом.

> **ЗАПОМНИ**
> Главный вопрос ревью не «правильный ли это код», а **«смогу ли я это починить через месяц»**. Код, который работает, но непонятен, — это долг, который отдавать тебе.

В Частые ошибки Python

что написано и что это значит

ЧТО ЭТО

Настоящие сообщения Python 3.13 — все получены запуском, не по памяти. Читать их надо снизу вверх: последняя строка говорит, **что** случилось, строки выше — **где** (статья 2.12).

ИМЕНА И ТИПЫ

`NameError: name 'cuty' is not defined` Имени не существует. Обычно опечатка или переменная, объявленная ниже по тексту, чем использована.

`AttributeError: 'str' object has no attribute 'push'` У объекта нет такого метода. Частый признак того, что ИИ перепутал язык: `push` есть у массивов в JavaScript, у списков Python — `append`.

`TypeError: can only concatenate str (not "int") to str` Сложили строку с числом. Почти всегда значение пришло из переменной окружения или из ввода и осталось строкой (статья 13.4).

`TypeError: 'NoneType' object is not subscriptable` Обратились по ключу или индексу к `None`. Значит, функция выше вернула `None` вместо данных — и это не проверили.

`TypeError: f() missing 1 required positional argument: 'b'` Вызвали с меньшим числом аргументов. Часто после правки сигнатуры, когда обновили не все вызовы.

`UnboundLocalError: cannot access local variable 'n' where it is not associated with a value`
Внутри функции присвоили значение переменной, которая объявлена снаружи. Python считает её локальной. Лечится передачей аргументом, а не `global`.

ДАННЫЕ

`KeyError: 'temp'` В словаре нет такого ключа. Самая частая ошибка при разборе ответов API: ключ есть, когда всё хорошо, и исчезает, когда сервер вернул ошибку.

`IndexError: list index out of range` Обратились за границу списка. Проверить, что список не пуст, — и помнить, что последний элемент это `[-1]`, а не `[len(xs)]`.

`ValueError: invalid literal for int() with base 10: '-2.5'` Строка не превращается в число. `int()` не понимает дробную запись — нужно `float()`, а потом уже округление.

`ZeroDivisionError: division by zero` Деление на ноль. Обычно знаменатель — длина пустого списка или счётчик, который не успел вырасти.

`json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)`
Разбирали как JSON то, что JSON не является: пустую строку, HTML-страницу ошибки, обрезанный файл. Печатать надо не разобранные данные, а сырой текст ответа.

`UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd8 in position 96`
Открыли двоичный файл как текст или текст в другой кодировке. Для двоичных — режим `"rb"`, для текста — явная `encoding=`.

ФАЙЛЫ И МОДУЛИ

`FileNotFoundError: [Errno 2] No such file or directory: 'cities.json'`
Файла нет **там, откуда запущена программа**. Относительный путь считается от рабочей папки, а не от файла с кодом (статья 12.1).

`ModuleNotFoundError: No module named 'rekvests'` Библиотека не установлена или написана с ошибкой. Если имя выглядит странно — возможно, её не существует вовсе (статья 4.8).

`ImportError: cannot import name 'x' from partially initialized module 'y' (most likely due to a circular import)`
Два модуля импортируют друг друга. Это не ошибка Python, а сигнал о перепутанных слоях (статья 13.1).

СИНТАКСИС И ВЫПОЛНЕНИЕ

`IndentationError: expected an indented block after function definition on line 1`
Потерян отступ. Обычно при вставке кода кусками.

`SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?`
Одно равно вместо двух в условии. Python подсказывает прямо в тексте ошибки.

`RecursionError: maximum recursion depth exceeded` Функция вызывает сама себя без условия остановки. В трейсбеке видно `[Previous line repeated 996 more times]`.

`sqlite3.IntegrityError: UNIQUE constraint failed: cities.name` Нарушено правило таблицы: запись с таким значением уже есть. Это не поломка, а сработавшая защита — её надо поймать и вернуть понятный ответ (статья 14.2).

> **СОВЕТ**
> Отдавая ошибку ИИ, копируй **весь** трейсбек, а не последнюю строку. Последняя строка говорит, что сломалось; строки выше — какой путь к этому привёл, и именно по ним находят причину.

Г Частые ошибки JS, TypeScript и Dart

три языка, одни и те же грабли

ЧТО ЭТО

Сообщения JavaScript и TypeScript получены запуском (Node 22, TypeScript 5.8). Сообщения Dart приведены по памяти и не прогонялись — как и вся глава 11, они требуют проверки по документации.

JAVASCRIPT ВО ВРЕМЯ РАБОТЫ

`TypeError: Cannot read properties of undefined (reading 'name')` Самая частая ошибка во всём фронтенде. Обратились к полю того, чего нет: данные ещё не пришли, ключ называется иначе, массив пуст. Лечится проверкой или `?.`, но сначала надо понять, **почему** там `undefined`.

`TypeError: Cannot read properties of undefined (reading 'map')` То же самое в React: отрисовали список до того, как пришли данные. Признак того, что состояние «загрузка» не сделано (статья 10.5).

`TypeError: Cannot set properties of null (setting 'textContent')` Элемент не найден. Скрипт выполнился раньше, чем браузер построил страницу, или селектор не тот.

`TypeError: xs.push2 is not a function` Метода не существует. Опечатка — или выдуманный метод (статья 4.8).

`TypeError: d is not iterable` Перебирают то, что не массив: `undefined`, объект, `null`.

`TypeError: Assignment to constant variable.` Присвоили значение `const`. Менять содержимое объекта можно, саму переменную — нет.

`ReferenceError: cuty is not defined` Имени не существует. В отличие от Python, в браузере это иногда означает, что скрипт не загрузился.

`SyntaxError: Unexpected end of JSON input` `JSON.parse` от пустой строки. Обычно ответ сервера пуст, потому что запрос не дошёл.

`SyntaxError: await is only valid in async functions and the top level bodies of modules`

`await` в обычной функции. Либо делать функцию `async`, либо работать через `.then`.

`TypeError: Failed to fetch` Запрос не состоялся. В коде ошибки нет: настоящая причина — в консоли браузера, чаще всего CORS (статья 14.3).

TYPESCRIPT ПРИ ПРОВЕРКЕ

`error TS2741: Property 'temp' is missing in type '{ name: string; }' but required in type 'City'`

Не хватает поля. Либо поле должно быть необязательным (`temp?`), либо его действительно забыли.

`error TS2322: Type 'string' is not assignable to type 'number'` Значение не того типа. Классика — число, пришедшее строкой.

`error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'`

То же самое в аргументе функции.

`error TS18048: 'd' is possibly 'undefined'` Значение может отсутствовать, а обращаются к нему напрямую. Правильный ответ — проверка, а не `!` (статья 9.5).

`error TS18047: 'c.temp' is possibly 'null'` То же для `null`. Появляется именно там, где типы описаны честно, — и это признак хорошего кода, а не проблемы.

`error TS2339: Property 'b' does not exist on type '{ a: number; }'` Поля нет в типе. Либо опечатка, либо тип описан не полностью — и тогда исправлять надо тип, а не затыкать `as any`.

DART И FLUTTER

> **ВНИМАНИЕ**
> Сообщения в этом разделе приведены по смыслу, а не скопированы из конкретного запуска. Смысл ошибок устойчив, точные формулировки сверяй со своей версией.

`Null check operator used on a null value` Поставили `!` там, где значение оказалось пустым. Прямой аналог `as` в TypeScript и такая же заглушка (статья 11.2).

`LateInitializationError: Field 'x' has not been initialized` Поле объявлено через `late`, но присвоить забыли.

`setState() called after dispose()` Виджет уже удалён, а ответ из сети пришёл и пытается обновить экран. Признак того, что подписку не закрыли в `dispose` (статья 11.4).

`RenderFlex overflowed by N pixels` Та самая жёлто-чёрная полоса: содержимое не помещается. Лечится `Expanded` или `Flexible` (статья 11.3).

`The method 'X' isn't defined for the type 'Y'` Метода нет. Во Flutter это чаще всего означает **код для другой версии**: главная болезнь стека.

ЧТО ОБЩЕГО

Три четверти списка — одна и та же ошибка на трёх языках: **обратились к тому, чего нет**. `None`, `null`, `undefined`, пустой ключ, пустой массив.

Поэтому приём один и тот же во всех стеках: не «добавить проверку в месте падения», а подняться выше и ответить, **почему там пусто** — и обработать это там, где данные появляются.

ШПАРГАЛКА

- Читай последнюю строку: она говорит, что случилось.
- `undefined (reading '...')` — данные ещё не пришли или ключ другой.
- `Failed to fetch` — причина в консоли браузера, а не в коде.
- `possibly 'undefined'` в TypeScript — признак честных типов, а не проблемы.
- `!` в Dart и `as` в TypeScript — одинаковые заглушки.
- Почти всё в списке — одна ошибка: обращение к тому, чего нет.

Д Git — сценарии спасения

что набрать, когда всё сломалось

ЧТО ЭТО

Не курс по git, а список ситуаций. Все команды прогнаны в настоящем репозитории, вывод настоящий. Основное про git — в статье 5.1.

КАЖДЫЙ ДЕНЬ

```
git status            # что изменено прямо сейчас
git diff              # что именно изменено, построчно
git diff --staged     # что уйдёт в коммит
git add .             # взять всё изменённое
git commit -m "..."   # зафиксировать
git log --oneline     # история коротким списком
```

Перед тем как просить ИИ о крупной правке, стоит сделать коммит. Это одна команда, и она превращает любую неудачу в «откатиться и попробовать иначе».

Я ИСПОРТИЛ ФАЙЛ И ХОЧУ ВЕРНУТЬ КАК БЫЛО

```
$ git diff --stat
 app.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git restore app.py      # вернуть один файл
$ cat app.py
третья версия
```

Вернуть **всё** к последнему коммиту:

```
git restore .
```

> **ВНИМАНИЕ**
> `git restore` выбрасывает незакоммиченные изменения **без возврата**. Это не то же самое, что откат коммита: коммит можно найти в истории, а незафиксированную правку — нет.

Я УДАЛИЛ ФАЙЛ

```
$ rm app.py
$ git restore app.py
$ ls
app.py
```

Пока файл был в последнем коммите, он возвращается.

Я ЗАКОММИТИЛ НЕ ТО

Коммит ещё не отправлен — отменить коммит, правки оставить:

```
$ git reset --soft HEAD~1
$ git status --short
M  app.py
```

Изменения на месте, коммита нет. Можно поправить и закоммитить заново.

Коммит уже отправлен — не переписывать историю, а добавить обратный:

```
$ git revert --no-edit HEAD
[main fba3f8d] Revert "третья версия"
 1 file changed, 1 insertion(+), 1 deletion(-)

$ cat app.py
вторая версия
```

`revert` создаёт новый коммит, который отменяет старый. История остаётся правдивой, и ни у кого ничего не ломается.

НАДО СРОЧНО ОТЛОЖИТЬ ПРАВКИ

```
$ echo "черновик" >> app.py
$ git stash          # спрятать и вернуться к чистому состоянию
$ cat app.py
вторая версия

$ git stash pop      # достать обратно
$ cat app.py
вторая версия
черновик
```

Удобно, когда ИИ наработал половину, а срочно понадобилось проверить что-то другое.

Я ЗАКОММИТИЛ СЕКРЕТ

```
$ git rm --cached .env
$ echo ".env" >> .gitignore
$ git add .gitignore && git commit -m "убрал .env из репозитория"
$ git ls-files
.gitignore
app.py
```

Из рабочей копии и из будущих коммитов `.env` ушёл. **Но:**

```
$ git log --oneline --all -- .env
1badd12 убрал .env из репозитория
5f9507b добавил .env
```

Файл по-прежнему в истории, и ключ из неё достаётся одной командой. Поэтому порядок действий такой: **сначала сменить ключ**, а потом уже чистить репозиторий. Если репозиторий публичный — считать ключ скомпрометированным с момента отправки (статья 5.7).

КОГДА СЛОМАЛОСЬ — А РАНЬШЕ РАБОТАЛО

```
git log --oneline              # найти подозрительный коммит
git show <хеш>                 # посмотреть, что в нём изменилось
git restore --source=<хеш> <файл>   # взять версию файла из коммита
```

Если коммитов много и непонятно, какой виноват, — есть `git bisect`, который делит историю пополам (приём из статьи 4.4, только автоматический).

ВЕТКИ

```
git switch -c опыт     # новая ветка от текущего места
git switch main        # вернуться
git merge опыт         # влить сделанное
git branch -d опыт     # удалить ветку
```

Для работы с ИИ ветка полезна в одном конкретном случае: **крупная переделка**, про которую заранее неясно, доживёт ли она до конца. В ветке её можно бросить, не убирая за собой.

Я СДЕЛАЛ RESET --HARD И ПОТЕРЯЛ КОММИТ

Коммит, который «пропал» после `reset`, ещё не удалён. Git помнит, где стоял `HEAD` каждый раз:

```
$ git reset --hard HEAD~1
$ cat app.py
первая версия
$ git reflog
a4d85e7 HEAD@{0}: reset: moving to HEAD~1
c03070e HEAD@{1}: commit: вторая версия
a4d85e7 HEAD@{2}: commit (initial): первая версия
$ git reset --hard HEAD@{1}
$ cat app.py
вторая версия
```

`reflog` — последняя линия обороны: он хранит записи несколько недель. Но спасает только закоммиченное. Незакоммиченные правки после `reset --hard` не вернуть ничем — ещё один довод коммитить перед каждой задачей для ИИ.

ЧЕГО ДЕЛАТЬ НЕ НАДО

`git push --force` в общую ветку. Переписывает историю у всех.

`git reset --hard` **без понимания.** Выбрасывает и коммиты, и правки.

Коммит «всё подряд» раз в неделю. Такой коммит невозможно откатить частично — а именно это и нужно, когда ИИ сломал одно из двадцати изменений.

Коммит с непрочитанным диффом. `git diff` перед коммитом — тридцать секунд, которые ловят случайно оставленный отладочный вывод и забытый ключ.

E Терминал — необходимый минимум

двадцать команд, которых хватает

ЧТО ЭТО

Столько терминала, сколько нужно, чтобы запускать свои программы, читать логи и понимать, что предлагает ИИ. Примеры — Linux и macOS; в Windows то же самое доступно через PowerShell или WSL.

ГДЕ Я И ЧТО ЗДЕСЬ

```
pwd                 # в какой папке нахожусь
ls                  # что здесь
ls -la              # всё, включая скрытое, с размерами и датами
cd проект           # войти в папку
cd ..               # на уровень вверх
cd                  # домой
```

Скрытые файлы — те, что начинаются с точки: `.env`, `.gitignore`. Без `-la` их не видно, и именно поэтому о них забывают.

ПОСМОТРЕТЬ ФАЙЛ

```
cat app.py          # целиком
head -20 app.py     # первые 20 строк
tail -20 app.py     # последние 20
tail -f logs/app.log  # последние и дальше в реальном времени
wc -l app.py        # сколько строк
```

`tail -f` — главная команда при отладке работающей программы: оставить в отдельном окне и смотреть, что пишется, пока нажимаешь кнопки.

НАЙТИ

```
$ grep -rn "ошибка" logs/
logs/app.log:1:ошибка
logs/app.log:3:ошибка снова

$ find . -name "*.py"
./app.py

$ grep -c "ошибка" logs/app.log
2
```

`grep -rn` — рекурсивно, с номерами строк. Это та команда, которой ищут, где в проекте используется функция, откуда читается настройка и не попал ли ключ в код:

```
grep -rn "sk-\|password\|api_key" .
```

ФАЙЛЫ

```
cp файл копия           # копировать
cp -r папка копия       # копировать папку
mv старое новое         # переместить или переименовать
rm файл                 # удалить без корзины
mkdir -p a/b/c          # создать папки, включая промежуточные
du -sh .                # сколько весит папка
```

> **ВНИМАНИЕ**
> У `rm` нет корзины и нет подтверждения. Прежде чем выполнять `rm` с маской или из чужого скрипта, замени его на `ls` с тем же аргументом и посмотри, что попадёт под удаление. Это та же мысль, что в статье 14.5: сначала показать, потом делать.

ЗАПУСК И ОСТАНОВКА

```
python3 app.py          # запустить
Ctrl+C                  # остановить работающую программу
Ctrl+D                  # закончить ввод (выйти из python, из ssh)
./script.sh             # запустить скрипт
chmod +x script.sh      # разрешить запуск
```

Если `./script.sh` отвечает `Permission denied` — не хватает `chmod +x`.

СОЕДИНЯТЬ КОМАНДЫ

```
команда1 && команда2    # вторая только если первая удалась
команда1 | команда2     # вывод первой на вход второй
команда > файл          # вывод в файл (перезаписать)
команда >> файл         # дописать в конец
команда 2>&1            # ошибки туда же, куда обычный вывод
```

`&&` — то, из чего состоит любая команда сборки: `tsc -b && vite build` означает «проверить типы, и только если получилось — собирать» (статья 12.1).

КОД ВОЗВРАТА

```
$ python3 -c "import sys; sys.exit(3)"
$ echo $?
3
```

`$?` — чем закончилась последняя команда. Ноль значит «успешно», всё остальное — нет. На этом работают и `&&`, и проверки в сборке: программа может напечатать «готово» и при этом вернуть ошибку.

ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

```
PORT=8080 python3 app.py     # задать только для этого запуска
export API_KEY=abc           # задать на всю сессию терминала
echo $API_KEY                # посмотреть
env | grep API               # перечислить подходящие
```

Переменная, заданная через `export`, живёт до закрытия окна и не попадает ни в какие файлы — этим она и хороша для ключей (статья 13.4).

ЧТО ЕЩЁ ПРИГОДИТСЯ

```
history | grep docker    # что я набирал про docker
Ctrl+R                   # поиск по истории команд
curl -s -o /dev/null -w "%{http_code}\n" localhost:8000/   # код ответа
df -h                    # сколько места на дисках
ps aux | grep python     # какие процессы запущены
kill <номер>             # остановить процесс
```

Команду `curl` из этого списка стоит запомнить: она отвечает на вопрос «мой сервер вообще жив?» без браузера и без догадок (статья 14.3).

Ж Глоссарий

английские слова, которые встречаются каждый день

ЧТО ЭТО

Слова, которые приходят в ответах ИИ, в сообщениях об ошибках и в документации. Не словарь, а список того, что мешает читать, пока не знаешь.

КОД И ЕГО ЧАСТИ

Слово	Что значит
variable	переменная
function, method	функция; метод — функция, принадлежащая объекту
argument, parameter	что передают в функцию; как это названо внутри
return	вернуть значение из функции
class, instance	класс (описание); экземпляр (созданный по нему объект)
property, field	свойство, поле объекта
array, list	массив, список
dictionary, map, object	словарь: пары «ключ — значение»
string, integer, float, boolean	строка, целое, дробное, да/нет
null, None, undefined	«ничего»; в JS ещё и «значение не задано»
scope	область видимости: где имя существует
statement, expression	инструкция; выражение, у которого есть значение
loop, iterate	цикл; проходить по элементам
nested	вложенный
mutable, immutable	изменяемый, неизменяемый

ЧТО ПРОИСХОДИТ ПРИ РАБОТЕ

Слово	Что значит
runtime	время работы программы (в отличие от сборки)
exception, throw, raise	исключение; «бросить», «возбудить» его
catch, handle	поймать, обработать ошибку
stack trace, traceback	цепочка вызовов до места падения
debug	искать причину ошибки
log	запись о событии; «логи» — такие записи
crash	падение программы
hang, freeze	зависание

race condition	состояние гонки: два процесса мешают друг другу
memory leak	утечка памяти
timeout	предел ожидания
async, await, promise	асинхронность: «подожди результат, не блокируя всё»
thread, process	поток; процесс
cache	хранилище готовых ответов, чтобы не считать заново

ДАННЫЕ И СЕТЬ

Слово	Что значит
API	способ одной программы обращаться к другой
endpoint	конкретный адрес API
request, response	запрос, ответ
payload, body	содержимое запроса или ответа
header	служебная строка запроса или ответа
status code	код ответа: 200, 404, 500
JSON	текстовый формат данных: пары и списки
schema, validation	описание данных; проверка по нему
parse, serialize	разобрать из текста; превратить в текст
query	запрос к базе данных
index	указатель в базе, ускоряющий поиск
migration	изменение структуры базы
transaction	группа изменений «всё или ничего»
CRUD	создать, прочитать, изменить, удалить
CORS	правило браузера о запросах на чужой адрес

ФРОНТЕНД

Слово	Что значит
frontend, backend	то, что видит пользователь; то, что на сервере
DOM	дерево элементов страницы в браузере
component	компонент: кусок интерфейса со своим кодом

props	данные, переданные компоненту снаружи
state	состояние: данные, которые компонент хранит сам
render	отрисовать, построить интерфейс
hook	функция React вида `useЧтото`
effect	действие в стороне: запрос, подписка, таймер
layout	раскладка, расположение элементов
responsive	подстраивающийся под ширину экрана
viewport	видимая область экрана
bundle	собранный файл со всем кодом страницы

ПРОЦЕСС И ИНСТРУМЕНТЫ

Слово	Что значит
repository, repo	репозиторий: проект с историей изменений
commit	зафиксированное изменение
branch, merge	ветка; слияние веток
diff	разница между версиями
pull request	предложение влить изменения
rollback, revert	откат
dependency	зависимость: чужая библиотека
lock file	файл с точными версиями зависимостей
environment	окружение: набор настроек для запуска
build	сборка
artifact	то, что получилось после сборки
deploy	выкладка на сервер
rollout	постепенная выдача новой версии
linter	программа, ищущая подозрительные места в коде
refactoring	переделка кода без изменения поведения
legacy	старый код, который менять страшно
technical debt	технический долг: отложенные решения

ПРО ИИ

Слово	Что значит
prompt	запрос к модели
context, context window	что модель видит сейчас; сколько в неё помещается

token	кусочек текста, которым модель считает объём
hallucination	выдуманное: несуществующая функция, ссылка, факт
temperature	насколько ответы разнообразны
fine-tuning	дообучение модели на своих данных
agent	режим, в котором модель сама запускает команды
system prompt	постоянная инструкция, заданная до разговора

СОВЕТ
Если в ответе ИИ встретилось незнакомое слово — спроси прямо там же: «что такое `race condition` в этом коде, одним абзацем». Это дешевле, чем искать отдельно, и объяснение будет про твой код, а не вообще.

Предметный указатель

Номера страниц — там, где термин объясняется или показан в деле.

агент 59, 131, 132, 133, 134, 135, 146
антипаттерны запросов 97, 99, 264
артефакт 268, 270, 273, 277, 285, 286, 287
архитектура 141, 289, 291, 293, 300, 306, 309
асинхронность 42, 43, 44, 45, 162, 166, 216
база данных 92, 94, 181, 183, 185, 189, 299
бэкенд 171, 173, 174, 180, 184, 186, 192
валидация 47, 68, 176, 177, 179, 180, 193
версии зависимостей 143, 231, 233, 234, 273, 274, 285
ветка 98, 114, 120, 221, 345, 351
виджет 255, 256, 257, 258, 262, 264, 265
виртуальное окружение 22, 92, 142, 143, 144, 147, 149
галлюцинации 119, 121, 123, 168, 228
декоратор 19, 39, 40, 41, 42, 62, 74
деплой 147, 149
дженерики 220, 221, 222, 223, 254
дифф 52, 53, 54, 71, 89, 90, 102
зависимости 48, 92, 143, 241, 243, 246, 248
изменяемый аргумент по умолчанию 30, 31, 56, 57, 168, 169, 170
индекс в базе 187, 188
инструменты разработчика 204, 205, 207, 247
исключение 36, 37, 38, 39, 50, 163, 173
класс 34, 35, 36, 200, 201, 218, 262
код ответа 106, 174, 176, 192, 206, 322, 348
коммит 129, 130, 131, 343, 344, 345, 346
компонент 235, 236, 237, 239, 240, 241, 245
контекст сессии 100, 123, 134, 135
конфиг 7, 95, 120, 139, 145, 147, 305
красные флаги 55, 57, 167
круговой импорт 48, 290, 291, 292, 293, 339
кэш 280, 281, 283, 284, 285, 297, 299
линтер 213, 229, 230, 243, 247, 248, 270
миграция 12, 13, 66, 189, 190, 191, 192
мини-ТЗ 64, 66, 94, 95, 96, 319, 322
окружения 92, 141, 143, 146, 302, 305, 306
откат 12, 66, 126, 127, 128, 129, 282
отладка печатью 111, 113
переменные окружения 6, 20, 21, 56, 92, 145, 146
план до кода 81, 83, 127, 133, 140, 334
подпись приложения 279, 284
приёмка 52, 75, 117, 118, 119, 193, 208
пропсы 236, 237, 239, 240, 247, 248, 259
публикация 276, 278, 280, 281, 283, 286, 287
разрешения 134, 263, 264, 265, 266, 278, 279
ревью 76, 77, 78, 79, 314, 334, 337
резервная копия 190, 277, 284, 299, 301, 311, 313
рефакторинг 71, 72, 73, 74, 80, 116, 130
сборка 233, 268, 269, 270, 273, 274, 275
секреты 106, 144, 145, 146, 147, 150, 285
скрипт автоматизации 329, 330, 331
слои 180, 181, 182, 183, 290, 292, 293
состояние 35, 239, 243, 244, 261, 262, 327
структура папок 293, 295, 297, 335

таймаут 104, 120, 162, 163, 164, 216, 317
телеграм-бот 46, 47, 59, 62, 142, 310, 315
тесты 79, 80, 81, 114, 115, 116, 117
типизация 14, 25, 26, 77, 159, 161
токен 21, 22, 24, 144, 145, 146, 192
трейсбек 48, 49, 50, 51, 69, 70, 71
файл контекста 59, 62, 91, 93, 138, 167, 306
фронтенд 204, 206, 207, 208, 210, 233, 247
хранение данных 62, 277, 278, 279, 288, 299, 301
шаблон запроса 333, 335
эндпоинт 94, 320, 335
.env 145, 146, 147, 148, 285, 306, 344
any 121, 161, 215, 219, 223, 227, 229
API 270, 315, 319, 322, 325, 326, 327
box-sizing 203, 204
CORS 324, 325, 326, 341, 350
CSS 195, 196, 201, 202, 203, 206, 256
Dart 249, 250, 251, 252, 253, 254, 255
dataclass 35, 41, 156, 157, 158, 159, 161
DOM 195, 196, 197, 198, 206, 207, 232
dry-run 330, 331, 332
f-строка 157, 194, 253
FastAPI 41, 171, 172, 320, 321
Flutter 249, 250, 251, 252, 258, 266, 267
git 126, 129, 130, 144, 343, 344, 345
HTML 195, 196, 197, 199, 232, 281, 283
HTTP 41, 172, 174, 181, 183, 323, 324
JavaScript 211, 212, 213, 214, 217, 218, 220
key в списке 184, 235, 236, 237, 246, 248, 259
mypy 160, 161, 162, 167, 170, 178, 220
N+1 запрос 186, 187, 188, 189, 193, 194
pathlib 46, 156, 157, 159, 168, 271, 329
PostgreSQL 188, 190, 299, 300, 301, 302, 304
pydantic 47, 172, 178, 226, 303, 305, 308
pytest 80, 81, 87, 92, 114, 116, 117
Python 92, 152, 153, 155, 165, 167, 168
React 59, 231, 232, 240, 246, 247, 322
setState 239, 240, 246, 248, 260, 262, 267
SQL-инъекция 185, 186, 193, 194, 320
SQLite 95, 297, 298, 299, 300, 301, 308
TypeScript 217, 218, 219, 220, 232, 340, 342
useEffect 41, 240, 241, 242, 243, 246, 261
useState 237, 240, 242, 243, 246, 323
Vite 231, 232, 233, 269, 272, 274, 280