

Павел Булкин

Вайб-Кодинг

Практический курс



Павел Булкин

Вайб-кодинг. Практический курс

«Автор»

2026

Булкин П.

Вайб-кодинг. Практический курс / П. Булкин — «Автор», 2026

ИИ уже умеет писать код быстрее, чем команда успевает его понимать. Вайб-кодинг дал разработчикам невероятную скорость — и вместе с ней новые риски: раздутые pull requests, слепое доверие агентам, дыры в безопасности, дрейф требований и технический долг, который появляется за один сеанс генерации. Эта книга — практический курс по инженерному управлению ИИ-разработкой. На сквозном проекте TaskFlow AI вы пройдёте полный цикл работы с AI-агентами: от постановки задачи и Plan Mode до спецификаций, контекстной инженерии, MCP, безопасного терминального агента, threat modeling, AI code review и рефакторинга с удалением лишнего кода. Книга подойдёт разработчикам, тимлидам и архитекторам, которые уже используют ИИ в работе и хотят перейти от хаотичного «навайбил и смержил» к контролируемой, проверяемой и безопасной разработке.

© Булкин П., 2026

© Автор, 2026

Содержание

Мы научились генерировать, но разучились понимать	6
Договор с читателем	7
Как устроен этот курс	8
Сквозной проект: TaskFlow AI	9
Что вы соберёте	10
Первый контраст: ленивый вайб против инженерного запроса	12
Ленивый промпт	13
Инженерный запрос	14
Практическое задание	15
Артефакт введения: BASELINE_COMPARISON.md	16
Пора протрезветь	17
Что Cursor делает хорошо	18
Почему «сделай фичу» — плохая команда	19
Минимальный контекст перед генерацией	20
PROJECT_RULES.md	21
ARCHITECTURE.md	22
TESTING.md	23
Практика: добавляем сущность Task	24
Как проверять план	25
Ошибки плана: красные флаги	28
Выполнение после утверждения	29
Мини-ревью результата	30
Что с вами происходит: смена ролей	31
Практическое задание	32
Артефакты главы	33
Что дальше	34
Разворачиваем ИИ в обратную сторону	35
Когда нужен глубокий контекст	36
Плохой подход: «объясни мне весь проект»	37
Хороший подход: ограниченный анализ	38
Карта модуля: MODULE_MAP_TASKS.md	39
Ловушка: убедительные галлюцинации	40
Правила работы с глубоким контекстом	41
Финальный запрос: карта с разметкой достоверности	42
Практическое задание	43
Конец ознакомительного фрагмента.	44

Павел Булкин

Вайб-кодинг. Практический курс

Содержание

Введение. Похмелье после вечеринки

ЧАСТЬ 1. ИНСТРУМЕНТЫ НОВОЙ ЭРЫ

Глава 1. Cursor: первая фишка через управляемую генерацию

Глава 2. Devin Desktop (бывший Windsurf): глубокий контекст и ИИ-археология

Глава 3. Claude Code: безопасный терминальный агент

Глава 4. Спецификация как код: от желания к контракту

ЧАСТЬ 2. МЕТОДОЛОГИЯ ВАЙБ-КОДИНГА 2.0

Глава 5. Контекстная инженерия: строим среду, а не промпт

Глава 6. Plan Mode: как читать и править планы ИИ

Глава 7. MCP: подключаем ИИ к реальности

ЧАСТЬ 3. КРИЗИС ДОВЕРИЯ

Глава 8. Security Lab: почему ИИ пишет дырявый код

Глава 9. AI Code Review: аудит чёрного ящика

Глава 10. Техническая инфляция: учим ИИ удалять код

ЧАСТЬ 4. БУДУЩЕЕ ПРОФЕССИИ

Глава 11. От джуниора к архитектору: дирижируем флотом агентов

Глава 12. Командный стандарт: дисциплина, которая работает без вас

Заключение. Похмелье прошло. Начинается ремесло.

Мы научились генерировать, но разучились понимать

Февраль 2025-го был временем почти детской эйфории. Мы все чувствовали себя волшебниками. Термин «вайб-кодинг», который Andrej Karpathy (Андрей Карпати) бросил в одном твите, за пару месяцев превратился из шутки в религию индустрии. Мы описывали желание словами, жали «Generate» в Cursor или тогдашнем Windsurf (с июня 2026-го — Devin Desktop) — и получали рабочее приложение. Прототип за обеденный перерыв. Стартап за выходные. Казалось, что синтаксис скоро станет таким же архаизмом, как перфокарты.

Это была грандиозная вечеринка. Но к началу 2026-го эйфория сменилась инженерным похмельем. Публичные опросы разработчиков показывают: ИИ-инструменты стали массовой частью рабочего процесса, но доверие к их точности остаётся ограниченным. Например, Stack Overflow Developer Survey 2025 фиксирует, что 46% разработчиков скорее не доверяют точности AI tools, 33% доверяют, а полностью доверяют только около 3%. Узкое место всё чаще переезжает из написания кода в его чтение, ревью, проверку безопасности и сопровождение. Мы научились очень быстро бежать — и всё чаще вынуждены останавливаться, чтобы понять, куда именно бежим и какой груз несём за спиной.

Похмелье у всех одинаковое. Симптомы такие:

- код генерируется быстрее, чем его кто-либо успевает прочитать;
- пул-реквесты пухнут до тысяч строк, которые невозможно осмысленно проверить;
- люди принимают код, который не понимают, нажимая не «Merge», а «верю на слово»;
- тесты проверяют не бизнес-правила, а то, что ИИ сам решил проверить;
- архитектура тихо расползается — пока в три часа ночи что-то не падает в продакшене из-за пограничного случая, который никто не продумал.

Проблема не в том, что ИИ пишет плохой код. Проблема в том, что он может писать слишком много кода быстрее, чем команда успевает его понять. Раньше технический долг копился месяцами из-за лени и сроков. Теперь он может появиться за один сеанс генерации — и стоимость чтения, отладки и поддержки никуда не делась. Она просто стала заметнее.

Договор с читателем

Эта книга не будет сборником магических промптов и не будет обзором модных IDE, который устареет через полгода. Обзоры инструментов гниют быстрее, чем выходят из печати. Эта книга — практический курс по управлению ИИ-разработкой. Курс, после которого вы не «знаете про ИИ», а умеете им управлять.

Чтобы между нами не было недопонимания, договоримся на берегу. Прочитав книгу и проделав упражнения, вы получите ответы на пять вопросов:

Какую проблему она решает. Как генерировать код с помощью ИИ и при этом не терять контроль над системой, безопасностью и архитектурой.

Кому она подходит. Разработчикам, тимлидам и архитекторам, которые уже пользуются ИИ-агентами и устали от хаоса. Не нужно быть сеньором; нужно быть готовым работать руками.

Что вы будете делать руками. Писать спецификации, читать и править планы ИИ, ревьюить чужой сгенерированный код, атаковать собственные фичи, заставлять агента удалять лишнее.

Какой проект соберёте. Один сквозной учебный сервис, который будет расти в сложности от главы к главе.

Что унесёте с собой. Набор файлов и навыков, которые можно в понедельник принести в свою реальную команду.

КАК ПОЛЬЗОВАТЬСЯ КУРСОМ

Каждая глава заканчивается практическим заданием и главным артефактом — иногда одним файлом, иногда небольшим набором файлов, которые вы создаёте сами. Не пропускайте их. Книгу можно прочитать за вечер, но курс нельзя «прочитать» — его можно только пройти. Артефакты складываются в один растущий репозиторий, и к финалу у вас на руках — рабочий каркас инженерной ИИ-разработки.

Как устроен этот курс

В основе книги — один принцип. Запомните его, он определяет всё остальное:

ГЛАВНЫЙ ПРИНЦИП

Один сквозной проект. Одна растущая сложность. Практический артефакт или небольшой набор артефактов после каждой главы.

Мы не будем перескакивать с примера на пример. Через всю книгу проходит один проект, и каждая глава добавляет к нему ровно столько сложности, сколько нужно, чтобы показать новый навык: роли и права, API, базу данных, авторизацию, безопасность, логи, баги, технический долг, ревью и рефакторинг. Сложность растёт вместе с вами, а после каждой главы в репозитории остаётся один главный артефакт или небольшой набор файлов.

И ещё одно важное обещание. Мы переписали этот курс так, чтобы практика не была пришта к концу теоретических глав. Каждая глава построена вокруг действия. Скелет главы один и тот же:

- плохой запрос — как сделать не надо и что при этом ломается;
- инженерный подход — как переформулировать то же самое в управляемую задачу;
- шаблоны файлов — что именно вы создаёте;
- практическое задание и усложнение;
- артефакт главы — файл или небольшой набор файлов, который остаётся у вас в проекте.

Сквозной проект: TaskFlow AI

Наш учебный проект — TaskFlow AI, сервис управления задачами для команды. Он выбран не случайно: на нём естественно вырастают все темы курса — от прав доступа до безопасной загрузки файлов. В нём будут:

- пользователи и роли `admin`, `manager`, `member`;
- команды: у `manager` и `member` есть `teamId`; для курса считаем, что пользователь состоит в одной команде;
- задачи со статусами `todo`, `in_progress`, `done` и отдельной архивацией через `archivedAt`, комментарии и вложения;
- REST API и база данных;
- права доступа и авторизация;
- тесты, аудит безопасности и рефакторинг.

Стек — Node.js + TypeScript на NestJS с базой PostgreSQL. Архитектура классическая, слоёная: контроллер принимает HTTP-запрос, сервис содержит бизнес-логику, репозиторий работает с базой. Если вы пишете на другом стеке — не страшно. Навыки этого курса (спецификация до кода, ревью плана, threat model, аудит PR) переносятся на любой язык и фреймворк с небольшой адаптацией. NestJS здесь — просто конкретные декорации, чтобы примеры были настоящими, а не «псевдокодом из вакуума».

Доменная оговорка для следующих глав: «своя команда» означает совпадение `teamId` у `manager`, задачи и `assignee`; `archived task = archivedAt != null`, а не `status=done`; `Guard` отвечает за аутентификацию, `TaskAccessPolicy/TaskAccessService` — за бизнес-правила доступа.

ЧТО ВАЖНЕЕ СТЕКА

Мы учим не «как писать на NestJS». Мы учим **думать системами** и управлять ИИ-агентом так, чтобы он не разрушил вашу систему. Код в этой книге — дешёв. Ценны решения о том, что и почему строить.

Что вы соберёте

К концу курса в репозитории TaskFlow AI появится отдельный пласт — инженерный контекст и инструменты управления ИИ. Вот полная карта основных артефактов; пока она выглядит как список незнакомых файлов, но каждый из них появится в соответствующей главе:

docs/ai-context/

PROJECT_RULES.md

ARCHITECTURE.md

SECURITY.md

TESTING.md

DOMAIN_GLOSSARY.md

API_CONTRACT.md

features/

SPEC_TASK_ASSIGNMENT.md

SPEC_REVIEW.md

SPEC_FILE_UPLOAD.md

PLAN_CREATE_TASK.md

TASK_FEATURE_REVIEW.md

PLAN_COMMENTS.md

PLAN_REVIEW_NOTES.md

PLAN_FILE_UPLOAD.md

THREAT_MODEL_FILE_UPLOAD.md

SECURITY_REVIEW_FILE_UPLOAD.md

NEGATIVE_TEST_PLAN.md

AI_PR_REVIEW.md

REFACTORING_PLAN.md

analysis/

MODULE_MAP_TASKS.md

CONTEXT_HALLUCINATION_CHECK.md

TEST_FAILURE_INVESTIGATION.md

CONTEXT_GAP_REPORT.md

CONTEXT_BUNDLE.md

BUG_DIAGNOSIS_PLAN.md

DUPLICATION_REPORT.md

context/

временная папка с логами, схемами, тикетами и другими источниками для диагностики

process/

AGENT_SANDBOX_RULES.md

MCP_ACCESS_POLICY.md

ARCHITECT_PLAYBOOK.md

TEAM_STANDARD.md

.github/

pull_request_template.md

BASELINE_COMPARISON.md

Это не «домашки ради домашних». Это рабочие документы, которые в настоящей команде превращают ИИ из источника хаоса в управляемый инструмент.

Первый контраст: ленивый вайб против инженерного запроса

Чтобы почувствовать разницу между «вайбом» и инженерией прямо сейчас, посмотрим на два запроса к одному и тому же агенту. Оба просят одно и то же — сервис задач. Но получают они разное.

Ленивый промпт

Сделай мне сервис задач.

Что почти наверняка пойдёт не так: агент выберет архитектуру за вас, создаст лишние файлы, забудет про права доступа, напишет тесты на то, что ему удобно проверять, и притащит пару зависимостей «на всякий случай». Результат запустится — и будет невозможно проверить.

Инженерный запрос

Сначала составь план реализации сервиса задач.

Код не писать до утверждения плана.

Контекст:

- есть роли admin, manager, member;
- manager может назначать задачи;
- member может менять статус только своих задач;
- admin может удалять любую задачу;
- создатель задачи может удалить свою задачу;
- member не может удалять чужие задачи;
- нужны тесты на права доступа;
- нельзя хранить секреты в коде.

Выведи:

1. модель данных;
2. API;
3. список файлов;
4. риски безопасности;
5. edge cases;
6. план реализации.

Разница не в «секретном слове» и не в вежливости. Разница в том, что во втором случае вы задали ограничения, потребовали план до кода и описали, что именно хотите получить на выходе. Вы перестали быть просителем и стали постановщиком задачи. Весь курс — про то, как делать это системно.

Практическое задание

Выполните один и тот же запрос двумя способами в вашем любимом ИИ-инструменте: ленивым промптом («Сделай мне сервис задач»); инженерным запросом с ограничениями (из примера выше).

Затем сравните оба результата по пяти критериям и заполните таблицу. Отвечайте честно — «да», «нет» или «частично»:

Критерий

Ленивый промпт

Инженерный запрос

Права доступа учтены?

Есть тесты?

Есть edge cases?

Есть план?

Агент изменил лишнее?

УСЛОЖНЕНИЕ

Дайте инженерный запрос дважды с интервалом — например, в начале и в конце рабочего дня — и сравните оба плана между собой. Где агент «доехал» до разных решений? Это ваш первый личный замер дрейфа намерений, к которому мы вернёмся в главе 4.

Артефакт введения: BASELINE_COMPARISON.md

Зафиксируйте результат сравнения в первом артефакте курса — файле BASELINE_COMPARISON.md в корне проекта. Это ваша точка отсчёта: к ней вы вернётесь в конце книги, чтобы увидеть, насколько изменился ваш способ работать с ИИ. Используйте такой скелет:

```
# BASELINE_COMPARISON.md
## Дата
## Инструмент и модель
## Запрос 1: ленивый промпт
Текст запроса:
Наблюдения:
- что сгенерировал агент;
- что пошло не так.
## Запрос 2: инженерный запрос
Текст запроса:
Наблюдения:
- что изменилось;
- какие риски агент назвал сам.
## Таблица сравнения
| Критерий | Ленивый | Инженерный |
| --- | --- | --- |
| Права доступа учтены? | | |
| Есть тесты? | | |
| Есть edge cases? | | |
| Есть план? | | |
| Агент изменил лишнее? | | |
## Вывод
Один абзац: что я понял про разницу между вайбом и инженерией.
```

Пора протрезветь

Похмелье — это неприятно, но это признак того, что вечеринка закончилась и пора браться за серьёзную работу. Мы больше не играем с ИИ. Мы строим с его помощью системы, за которые отвечаем.

Ваша ценность как разработчика больше не измеряется тем, как быстро вы печатаете функцию `sort()`. Она измеряется тем, насколько точно вы умеете ставить задачу, проектировать контекст и сохранять критический взгляд в мире бесконечной генерации. ИИ — гениальный, но забывчивый стажёр со склонностью срезать углы. Ваша работа — быть взрослым в этой паре.

В следующей главе мы добавим в TaskFlow AI первую фичу — создание задач — но не «навайбим» её, а проведём через управляемую генерацию: контекст, план, ревью, минимальный diff. Переверните страницу. Начинаем строить по-настоящему.

ЧАСТЬ 1

Инструменты новой эры

Соблазн первой части любой книги про ИИ-разработку — устроить парад инструментов: вот Cursor, вот Devin Desktop (бывший Windsurf), вот Claude Code, вот spec-first инструменты вроде Trauser, у каждого свои кнопки. Проблема в том, что такой обзор устаревает быстрее, чем читатель дойдёт до середины. Кнопки переименуют, меню переедет, появится новый модный редактор.

Поэтому мы смотрим на инструменты не как на продукты, а как на рабочие режимы. У каждого режима — своя сильная сторона, и важно не «какой инструмент лучше», а «какой режим нужен под текущую задачу»:

- Cursor — контролируемая генерация: понятная задача, ограниченный контекст, предсказуемый diff.
- Devin Desktop (бывший Windsurf) — глубокий контекст: понять чужой, плохо документированный код (глава 2).
- Claude Code — автономный терминальный цикл: запустить тесты, найти причину, починить (глава 3).
- Trauser и подход «спецификация до кода» — заморозить намерение раньше, чем оно поплывёт (глава 4).

Названия меняются. Режимы — нет. Начнём с самого частого: управляемой генерации новой фици.

ЧАСТЬ 1 · ГЛАВА 1

Cursor: первая фица через управляемую генерацию

Добавляем создание задач в TaskFlow AI — без хаоса

Вайб-кодинг: Практический курс

Что Cursor делает хорошо

Не будем пересказывать историю инструмента. Достаточно знать главное: Cursor удобен, когда агенту нужно работать не с изолированным фрагментом, а с контекстом проекта — открытыми файлами, правилами, поиском по кодовой базе и текущим diff. В актуальных версиях у Cursor есть Plan Mode: режим, в котором агент сначала формирует проверяемый план реализации. Не считайте это магической защитой от ошибок; считайте это удобной точкой контроля, где человек должен остановиться и прочитать план до изменения кода.

Практический вывод одной фразой:

КОГДА CURSOR В СВОЕЙ СТИХИИ

Cursor хорош, когда у вас есть **понятная задача, ограниченный контекст** и вы хотите получить **контролируемый diff** — конкретный набор изменений, который можно прочитать и принять или отклонить целиком.

И обратное тоже важно. Cursor — не лучший выбор, когда нужно разобраться в огромной чужой системе, где правка в одном сервисе каскадом роняет три других (это работа для главы 2), или когда задача — не добавить код, а удалить лишний (глава 10). Каждому режиму своё.

Почему «сделай фичу» — плохая команда

Самый частый способ испортить себе день — открыть Composer и написать что-то вроде:

Добавь задачи в приложение.

Команда короткая и звучит безобидно. Но вы не задали ни одного ограничения, поэтому все решения агент примет за вас — и почти наверняка не так, как нужно. Что пойдёт не так:

- создаст лишние файлы, которых вы не просили;
- выберет странную архитектуру — например, размажет логику по контроллеру;
- не добавит тесты или добавит тесты на то, что ему удобно проверять;
- смешает слои: контроллер полезет в базу мимо сервиса и репозитория;
- забудет про права доступа — кто вообще может создавать задачи?
- притащит новую зависимость «чтобы было проще», без всякой причины.

Заметьте: каждая из этих проблем — не баг модели, а пробел в вашем запросе. Вы не ска-
зали, какая архитектура, нужны ли тесты, кто имеет права. Агент заполнил пустоты по своему
усмотрению. Управляемая генерация начинается с того, что вы перестаёте оставлять пустоты.

Минимальный контекст перед генерацией

Прежде чем просить агента что-либо генерировать, дайте ему правила игры. На этом этапе достаточно трёх файлов в проекте: `PROJECT_RULES.md`, `ARCHITECTURE.md` и `TESTING.md`. Полноценный контекстный слой мы соберём в главе 5, а сейчас — необходимый минимум, чтобы Cursor не выдумывал архитектуру на ходу.

PROJECT_RULES.md

Это конституция проекта для агента. Не «советы», а «законы»:

PROJECT_RULES.md

- Код не писать до утверждения плана.
- Не менять больше 5 файлов без отдельного согласования.
- Бизнес-логика должна жить в service-слое.
- Controller не должен обращаться к базе напрямую.
- Каждый endpoint должен иметь тесты.
- Новые зависимости запрещены без объяснения.

ARCHITECTURE.md

Короткая карта слоёв и явных запретов. Для NestJS она почти очевидна, но именно поэтому её и нужно записать — чтобы агент не «улучшал» её на свой вкус:

ARCHITECTURE.md

Слои

- Controller принимает HTTP-запрос и передаёт вход через DTO.

- Runtime-валидация DTO работает только при включённом ValidationPipe и правилах валидации.

- Service содержит бизнес-логику.

- Repository работает с базой данных.

Запрещено

- бизнес-логика в контроллере;

- прямые запросы к базе из контроллера;

- дублирование проверки ролей в каждом endpoint.

TESTING.md

TESTING.md

- Каждый новый endpoint покрыт тестами.
- Обязательны негативные тесты (запреты, чужие данные).
- Тест проверяет поведение, а не реализацию.
- Happy path без негативных тестов не принимается.

ПОДСКАЗКА ПО CURSOR

Эти файлы нужно подключить как правила Cursor (rules) или явно прикладывать к запросу, иначе агент может их просто не увидеть. Где именно лежит настройка — смотрите в актуальной документации инструмента; механика меняется, а смысл нет: правила проекта должны попадать в контекст без ручного копипаста.

Практика: добавляем сущность Task

Теперь — тот же самый запрос «добавь задачи», но переписанный как инженерная задача. Сравните с «Добавь задачи в приложение» из начала главы:

Нужно добавить сущность Task.

Контекст:

- пользователь может создавать задачу;
- задача имеет title, description, status, createdById;
- assigneeId пока может существовать в модели как nullable-поле, но CreateTaskDto его не принимает;
- status: todo, in_progress, done;
- пока без комментариев, вложений и назначения задачи;
- использовать текущую архитектуру проекта;
- не добавлять новые зависимости.

Сначала составь PLAN_CREATE_TASK.md.

Код не писать.

Разберём, что мы сделали:

- Сузили задачу. Только сущность Task и её создание — без комментариев и вложений. Это страховка от того, что агент «заодно» сделает половину приложения.
- Зафиксировали модель данных. Перечислили поля и значения статуса, чтобы не получить случайный набор колонок.
- Запретили вольности. «Использовать текущую архитектуру», «не добавлять зависимости» — два самых частых источника мусора.
- Потребовали план до кода. Главная фраза: Сначала составь PLAN_CREATE_TASK.md. Код не писать. Пока нет плана — нет генерации.

Как проверять план

Агент вернёт план. Не принимайте его рефлексивно. Хороший план обязан отвечать на шесть вопросов:

- какие файлы будут изменены;
- какие файлы будут созданы;
- какие решения агент сознательно не принимает (что вне задачи);
- где будет жить бизнес-логика;
- какие нужны тесты;
- какие есть риски.

Вот как выглядит план, который проходит проверку. Это и есть ваш будущий артефакт `PLAN_CREATE_TASK.md`:

```
# PLAN_CREATE_TASK.md
```

```
## Задача
```

Добавить создание задачи (entity Task) в модуль tasks.

```
## Что НЕ входит в задачу
```

- комментарии;
- вложения;
- назначение задачи (assign);
- уведомления.

```
## Будут созданы
```

- src/tasks/dto/create-task.dto.ts
- src/tasks/task.entity.ts
- src/tasks/tasks.controller.ts
- src/tasks/tasks.service.ts
- src/tasks/tasks.repository.ts
- src/migrations/*-create-task.ts
- test/tasks/create-task.e2e-spec.ts

Будут изменены

- src/tasks/tasks.module.ts (регистрация провайдеров)

НЕ будут затронуты

- src/users/*
- src/auth/* (используем существующий guard только для аутентификации)

Где бизнес-логика

- TasksService.createTask(): валидация, выбор teamId создателя и запись;
- бизнес-решения доступа — через TaskAccessPolicy/TaskAccessService, если такой слой уже есть; если нет — не создавать его в этой главе, а только не дублировать будущие правила.

Модель данных

Task { id, title, description, status, createdById, teamId, assigneeId: uuid | null, archivedAt: Date | null, createdAt }

status: 'todo' | 'in_progress' | 'done' (default 'todo')

archivedAt != null означает архивную задачу; это не то же самое, что status='done'.

CreateTaskDto: title, description. assigneeId и archivedAt не принимать на этом этапе.

Лишние поля в DTO запрещаются через ValidationPipe whitelist + forbidNonWhitelisted или явно игнорируются и покрываются тестом.

Тесты

- member может создать задачу;
- созданная задача принадлежит создателю (createdById);
- задача получает teamId создателя;
- статус по умолчанию 'todo';
- archivedAt по умолчанию null;
- неавторизованный запрос отклоняется;

- попытка передать `assigneeId` или `archivedAt` при создании не приводит к назначению или архивации.

Риски

- дублирование проверки прав, если не использовать общий `policy layer`;
- расширение DTO лишними полями, особенно `assigneeId` до отдельной фичи `assign`.

Rollback

- удалить новые файлы, откатить миграцию `task`.

Порядок реализации

1. `entity` + DTO + миграция;
2. `repository` + `service`;
3. `controller` + маршрут `POST /tasks`;
4. тесты.

Когда перед глазами такой документ, ваш мозг переключается из режима «поиск опечаток» в режим «стратегический обзор». Кривой план правится одной фразой; кривой код после генерации — это часы отладки. Исправлять текст плана в сто раз дешевле, чем переписывать пятьсот сгенерированных строк.

Ошибки плана: красные флаги

Чаще проверять план придётся не на полноту, а на самодеятельность агента. Верните план на доработку, если видите хотя бы один из признаков:

КРАСНЫЕ ФЛАГИ

- агент хочет переписать архитектуру ради одной фичи;
- в плане нет тестов;
- нет списка файлов или нет раздела «что не делаем»;
- нет rollback;
- безопасность вынесена «на потом»;
- добавляется библиотека без объяснения причины;
- в плане фигурируют unrelated-файлы, не связанные с задачей.

Реакция на красный флаг — не «перепиши всё», а точечная правка. Например: «План слишком широкий. Убери изменения в `src/users/`. Не трогай `auth` — используй существующий `guard`. Сократи список файлов до минимально необходимого». Вы правите план, а не код.

Выполнение после утверждения

Только когда план вас устраивает, вы разрешаете писать код. Введите для себя жёсткое правило, которое мы пронесём через всю книгу:

ПРАВИЛО УТВЕРЖДЕНИЯ

Код можно писать только после явной команды вроде: «План утверждён. Выполняй шаг 1». Никакой генерации до этой фразы.

Выполнение по шагам, а не «сделай всё разом», даёт вам точки контроля. После каждого шага можно остановиться, посмотреть diff и решить, продолжать ли. Это и есть контролируемая генерация: вы держите руку на рубильнике, а не наблюдаете, как агент за минуту переписывает полпроекта.

Мини-ревью результата

Код сгенерирован. Соблазн — запустить, увидеть зелёные тесты и нажать «принять». Не торопитесь. Сначала сверьте то, что получилось, с тем, что планировалось. Попросите агента сделать это за вас, но проверьте его ответ глазами:

Сравни реализованный diff с PLAN_CREATE_TASK.md.

Покажи:

1. где реализация соответствует плану;
2. где отклонилась;
3. какие файлы были изменены вне плана;
4. какие тесты покрывают новую логику.

Особенно внимательно — к пункту 3. «Файлы, изменённые вне плана» — это самый частый канал, по которому в проект просачивается технический долг. Если агент «заодно» поправил `src/users/user.entity.ts`, хотя в плане этого не было, — это повод не принимать diff, а разобраться.

Что с вами происходит: смена ролей

Обратите внимание, что в этой главе вы ни разу не писали код руками — и при этом проделали инженерную работу. За один цикл вы примерили четыре роли:

- Product owner — сформулировали намерение и границы задачи;
- Архитектор — проверили план на соответствие слоям и запретам;
- Security officer — спросили про права доступа раньше, чем агент про них забыл;
- QA — потребовали тесты и сверили diff с планом.

Это и есть новая работа разработчика. Ценность никогда не была в наборе символов на клавиатуре — она в решении проблем. Cursor просто убрал трение между мыслью и реализацией, и теперь ваша мысль должна быть чище, точнее и системнее, чем раньше.

Практическое задание

Добавьте фичу «создание задачи» в TaskFlow AI через текущий агентный интерфейс Cursor / Plan Mode, строго по циклу этой главы:

создайте PROJECT_RULES.md, ARCHITECTURE.md, TESTING.md;

отправьте инженерный запрос с требованием PLAN_CREATE_TASK.md до кода;

проверьте план по шести вопросам, верните на доработку при красных флагах;

утвердите план и выполните его по шагам;

проведите мини-ревью: сравните diff с планом.

УСЛОЖНЕНИЕ

Добавьте в задачу ограничение и проследите, чтобы оно дошло до плана, кода и тестов:

Пользователь не может создать задачу с пустым title или с title длиннее 120 символов.

Проверьте: появился ли в плане соответствующий пункт валидации? Появились ли **негативные** тесты на пустой и слишком длинный заголовок? Если агент добавил валидацию, но забыл тесты на неё — план не принят.

Артефакты главы

Главные артефакты фичи после этой главы — два файла. Первый — `PLAN_CREATE_TASK.md` (шаблон выше). Второй — отчёт о мини-ревью `TASK_FEATURE_REVIEW.md`. Кроме них, в проекте уже остаётся минимальный контекстный набор: `PROJECT_RULES.md`, `ARCHITECTURE.md`, `TESTING.md`:

```
# TASK_FEATURE_REVIEW.md
## Фича
Создание задачи (Task).
## Ссылка на план
PLAN_CREATE_TASK.md
## Соответствие плану
- что совпало с планом;
## Отклонения
- где реализация ушла от плана и почему;
## Изменения вне плана
- файлы, затронутые без согласования (в идеале — пусто);
## Покрытие тестами
- какие тесты добавлены, включая негативные;
## Решение
- approve / request changes / regenerate from plan.
```

Эти два файла — не отчётность ради отчётности. `PLAN_CREATE_TASK.md` фиксирует намерение, `TASK_FEATURE_REVIEW.md` фиксирует, что намерение было исполнено. Вместе они дают то, чего не хватало в эпоху «навайбил и забыл»: прослеживаемую связь между тем, что вы хотели, и тем, что оказалось в репозитории.

Что дальше

Вы научились управляемой генерации нового кода: контекст, план, ревью плана, выполнение, ревью diff. Но в реальной работе чаще приходится иметь дело с кодом, который вы не писали — чужим, legacy, плохо документированным. Просить агента «добавь фичу» в такой код опасно вдвойне: он не понимает систему, и вы тоже.

В следующей главе мы развернём ИИ в обратную сторону — будем использовать его не для генерации, а для понимания существующей кодовой базы. Возьмём модуль задач, который уже есть, но никто толком не помнит, как он устроен, и проведём над ним «ИИ-археологию» с помощью Devin Desktop (бывший Windsurf).

ЧАСТЬ 1 · ГЛАВА 2

Devin Desktop (бывший Windsurf): глубокий контекст и ИИ-археология

Используем ИИ не для генерации, а для понимания чужого кода

Вайб-кодинг: Практический курс

Разворачиваем ИИ в обратную сторону

В первой главе ИИ писал за нас новый код. Но честно посмотрим на реальную работу: чаще мы не пишем с нуля, а разбираемся в том, что уже написано — кем-то другим, давно, без документации. Модуль работает, его боятся трогать, а человек, который его задумывал, уволился два года назад. Просить агента «добавь сюда фичу» в такой ситуации опасно вдвойне: он не понимает систему — и вы тоже.

Здесь в своей стихии другой режим — глубокий контекст. Devin Desktop (бывший Windsurf) с его агентным режимом полезен не потому, что «знает всё», а потому что помогает навигировать по проекту, связывать файлы, зависимости и места использования в одну проверяемую картину. Но эту мощь мы направим не на генерацию, а на археологию: реконструировать, как устроен незнакомый модуль, и — что важнее — отличить то, что агент действительно вычитал из кода, от того, что он убедительно придумал.

ЦЕЛЬ ГЛАВЫ

Научиться использовать ИИ для **понимания** существующей кодовой базы. Не «Windsurf видит проект», а «как с помощью глубокого контекста разобрать чужой модуль и не поверить агенту на слово».

Рабочий материал — модуль задач TaskFlow AI. Представим, что он уже существует (его «написал кто-то до вас»), работает, но документация по нему мертва. Наша задача — восстановить карту.

Когда нужен глубокий контекст

Режим археологии включают не всегда, а в конкретных ситуациях. Узнаёте хотя бы одну — значит, пора:

- вы пришли в legacy-проект, который видите впервые;
- документация устарела и врёт;
- фича работает, но никто не знает, почему именно так;
- изменение в одном файле необъяснимо ломает другой модуль;
- нужно провести ревью перед доработкой, а не доработать вслепую.

Во всех этих случаях цель — не изменить код, а снизить неопределённость перед тем, как его менять. Это отдельная фаза работы, и смешивать её с генерацией нельзя.

Плохой подход: «объясни мне весь проект»

Первое, что хочется сделать в незнакомом проекте, — попросить:

Объясни мне весь проект.

Звучит разумно, а на деле это худший возможный запрос. Почему:

- слишком широко — агент не сможет быть конкретным;
- он будет обобщать — выдаст правдоподобное описание «типичного проекта», а не вашего;
- появятся галлюцинации — на широком вопросе модель додумывает связи, которых нет;
- результат невозможно проверить — вы не отличите правду от выдумки, потому что сами проект ещё не знаете.

Широкий вопрос даёт широкий, гладкий и непроверяемый ответ. А непроверяемый ответ в археологии хуже, чем отсутствие ответа: он создаёт ложную уверенность.

Хороший подход: ограниченный анализ

Сузьте область до одного модуля и потребуйте конкретики со ссылками на файлы:

Проанализируй только модуль `tasks`.

Код не менять.

Выведи:

1. какие файлы участвуют в создании задачи;
2. где проверяются права доступа;
3. где находится бизнес-логика;
4. какие зависимости есть у модуля;
5. какие места выглядят рискованными;
6. какие утверждения ты можешь подтвердить ссылкой на файл.

Два пункта здесь — несущие. Код не менять отделяет фазу понимания от фазы изменений: в археологии агент только читает. А шестой пункт — какие утверждения ты можешь подтвердить ссылкой на файл — превращает рассказ в проверяемый отчёт. Если агент пишет «права проверяются в `TasksService`», он обязан указать файл и строки. Нет ссылки — нет факта, есть предположение.

ГЛАВНЫЙ ПРИЁМ АРХЕОЛОГИИ

Не «расскажи мне», а «покажи, где». Любое утверждение агента о коде должно опираться на конкретный файл и место в нём. Ссылка — это то, что вы можете открыть и проверить за десять секунд.

Карта модуля: MODULE_MAP_TASKS.md

Результат анализа читатель оформляет в артефакт — карту модуля. Это не пересказ кода, а структурированная схема: что где живёт, как течут данные, где слабые места. Шаблон:

```
# MODULE_MAP_TASKS.md
## Назначение модуля
## Основные файлы
## Поток данных
## Где создаётся задача
## Где проверяются права
## Где пишутся данные в БД
## Зависимости
## Потенциальные риски
## Что неясно
## Что нужно уточнить у человека
```

Обратите внимание на два последних раздела — Что неясно и Что нужно уточнить у человека. В обычном пересказе их нет, и зря: именно они отличают честную карту от красивой галлюцинации. Карта без зон неизвестности почти всегда означает, что агент чего-то недопонял и закрасил пробел догадкой.

Ловушка: убедительные галлюцинации

Вот парадокс глубокого контекста, который нужно прочувствовать на собственной шкуре:

ЗАПОМНИТЕ

Чем больше контекста видит агент, тем **убедительнее** его ошибки. Глубокий анализ делает выдумку правдоподобной: модель ссылается на реальные имена классов и складывает их в связную, но неверную картину.

Разберём на конкретном примере. Агент уверенно заявляет:

Права доступа проверяются в `TasksService`.

Звучит логично — бизнес-логика и должна быть в сервисе. Но прежде чем записать это в карту, проверьте четыре вещи:

Файл существует? Откройте `tasks.service.ts`. Он вообще есть, и так ли он называется?

Проверка реально там? Найдите в файле собственно проверку роли. Может оказаться, что её там нет — она в `guard'e` или в контроллере.

Нет ли дублирования? Частая беда: проверка прав размазана и по контроллеру, и по сервису. Тогда «проверяется в сервисе» — полуправда.

Реальная архитектура или желаемая? Это главный вопрос. Агент мог описать, как права должны проверяться по канону, а не как они проверяются на самом деле.

Четвёртый пункт — суть всей главы. Модель обучена на «правильных» проектах, поэтому она охотно описывает идеальную архитектуру и выдаёт её за вашу. Ваша работа — ловить подмену «как есть» на «как принято».

Правила работы с глубоким контекстом

ЧЕТЫРЕ ПРАВИЛА АРХЕОЛОГА

- просите файлы и строки;
- отделяйте факты от предположений;
- требуйте список «не уверен»;
- анализ не должен менять код.

Финальный запрос: карта с разметкой достоверности

Собрав анализ, попросите агента оформить карту так, чтобы каждый вывод был помечен по уровню достоверности. Это превращает карту из «мнения ИИ» в рабочий документ, с которым можно идти к команде:

На основе анализа модуля `tasks` создай `MODULE_MAP_TASKS.md`.

Раздели выводы на четыре категории:

- подтверждено кодом (со ссылкой на файл);
- предположение;
- риск;
- вопрос к человеку.

Теперь карту можно читать с правильным уровнем доверия. «Подтверждено кодом» — опора. «Предположение» — гипотеза, которую надо проверить перед доработкой. «Риск» — место, куда нельзя лезть без тестов. «Вопрос к человеку» — то, что ИИ в принципе не может узнать из кода: бизнес-причины, исторические решения, договорённости.

Практическое задание

Проведите «ИИ-археологию» модуля задач TaskFlow AI:
запустите ограниченный анализ только модуля tasks с запретом менять код;
потребуйте ссылки на файлы для каждого утверждения;
проверьте минимум одно утверждение агента руками (по схеме из раздела про галлюцинации);

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.