18+

Михаил Тарасов

ВАЙБ-КОДЕР

ОТ НУЛЯ ДО МАСТЕРА В ЭПОХУ AI

Михаил Тарасов

Вайб-кодер от нуля до мастера в эпоху AI

«Издательские решения»

Тарасов М.

Вайб-кодер от нуля до мастера в эпоху AI / М. Тарасов — «Издательские решения»,

Эта книга — твой мануал по выживанию в мире, где ИИ пишет код быстрее тебя. Забей на слепой копипаст! Мы разберем, как приручить полуплярные AI-сервисы, чтобы они пахали на тебя, а не роняли прод кривыми скриптами. От первого «Hello, AI» до деплоя и разборок с GPL-лицензиями. Учись дебажить ИИ-галлюцинации, строить архитектуру и оставаться инженером, а не промпт-придатком к нейронке. Залетай в наш гараж, наливай кофе — будем кодить с AI-турбонаддувом!

Содержание

Вайб-кодер от нуля до мастера в эпоху AI

Михаил Тарасов

ISBN 978-5-0071-0545-3
Создано в интеллектуальной издательской системе Ridero

Часть 1. Новый старт: твой гараж теперь с AI-турбонаддувом

Глава 1. Добро пожаловать в гараж кодеров 2.0

— Кто мы теперь: кодеры или менеджеры AI? Разбираем новую философию

Помнишь то чувство, когда ты впервые написал скрипт, и он заработал? Чистый кайф. Ты был творцом, демиургом, который из хаоса символов создавал порядок. Ты часами дебажил одну строчку, находил баг и чувствовал себя Шерлоком Холмсом в мире кода. Это была наша игра, наш вайб

А теперь представь: ты сидишь, смотришь в пустой файл, и тут GitHub Copilot, как слишком усердный джун-стажёр, предлагает тебе готовый код на 50 строк. Ты нажимаешь Tab, и магия происходит. Функция готова. Работает. Но… где кайф? Где борьба? И главный, самый стрёмный вопрос: если он может так, то зачем вообще нужен я?

Этот холодок по спине почувствовал каждый из нас. Паника, отрицание, гнев. «Да это просто glorified автокомплит!», «Он пишет говнокод!», «Это всё пузырь!». Мы пытались отмахнуться, как от назойливой мухи. Но муха оказалась размером с истребитель. И теперь она паркуется в нашей IDE.

Так что, всё? Сворачиваем гараж, продаём ноуты и идём учиться на бариста?

Стоп. Выдыхай.

Давай разберём эту ситуацию как кодеры — без паники и с холодным рассудком.

Проблема не в том, что появился инструмент, который умеет писать код. Проблема в том, что мы привыкли измерять свою крутость количеством написанных строк. Наша самооценка была привязана к процессу набора символов на клавиатуре. И когда этот процесс автоматизировали, мы почувствовали себя так, будто у нас отобрали любимый гаечный ключ.

Но правда в том, что хороший механик ценен не тем, как быстро он крутит гайки. Он ценен умением провести диагностику, понять, где именно стучит движок, и выбрать правильный инструмент для починки.

Мы не становимся **менеджерами AI**. Это слово отдаёт корпоративным булшитом. Менеджер — это тот, кто раздаёт задачи и следит за KPI. Это не наш вайб.

Мы становимся **архитекторами, дирижёрами и пилотами систем, усиленных AI**.

Давай по полочкам, в чём разница.

— Оператор конвейера (или менеджер AI): Ему говорят: «Сделай кнопку». Он идёт к AI, пишет промпт: «сделай кнопку». Копирует результат, вставляет. Если что-то не работает — разводит руками. Он не понимает, *как* это работает. Он просто жмёт на кнопки. Это путь в никуда, потому что завтра AI научится нажимать на кнопки сам.

— Вайб-кодер 2.0 (пилот/архитектор): Ему говорят: «Нам нужна система авторизации». Он не бежит к AI с криком «напиши мне авторизацию». Он сначала думает:

— Архитектура: «Так, здесь лучше подойдёт JWT, а не сессии. Пароли будем хешировать с помощью bcrypt. Нужна защита от брутфорса — добавим rate limiting. Роли пользователей будут храниться в отдельной таблице».

— **Декомпозиция:** Он разбивает большую задачу на мелкие куски, как мы разбираем сложный баг. «Сначала модель пользователя, потом роуты для регистрации, потом логин, потом middleware для проверки токена».

— **Пилотирование AI:** И только теперь он обращается к AI, как к своему второму пилоту: «Эй, Copilot, накидай-ка мне boilerplate для Express-роута /register с валидацией email и пароля».

— **Критический ревью:** Он получает код от AI и смотрит на него взглядом сеньора, который ревьюит пулл-реквест от джуна. «Ага, валидация слабая, надо добавить проверку на сложность пароля. О, а тут он забыл обработать ошибку, если пользователь уже существует. И хеширование лучше вынести в отдельный сервис».

— **Ответственность:** Он допиливает, рефакторит и пушит код в прод, потому что в git blame будет стоять его имя. Он, а не Скайнет, отвечает за то, что система не упадёт и данные пользователей не утекут.

Чувствуешь разницу? В первом случае ты — придаток к машине. Во втором — ты её командир.

AI — это не замена твоего мозга. Это самый мощный **турбонаддув** для него. Он берёт на себя рутину: написание бойлерплейта, запоминание синтаксиса редких функций, генерацию тестов. Он освобождает твои когнитивные ресурсы для самого главного — для того, что и делает нас кодерами, а не машинами. Для **решения проблем**.

Наша ценность смещается от «я могу быстро написать цикл for» к «я могу спроектировать систему, которая выдержит 100 000 пользователей, и объяснить, почему я выбрал именно такие технологии».

Так что выкинь из головы мысль о том, что ты станешь не нужен. Наоборот. В мире, где каждый может сгенерировать кусок кода, на вес золота будут те, кто умеет отличать хороший код от плохого, кто видит картину целиком и кто не боится брать на себя ответственность за конечный продукт. Наш вайб — креативность, инженерная смекалка, чутьё на «код с душком» — становится нашим главным конкурентным преимуществом.

Итак, кто мы теперь? Мы всё те же кодеры из гаража. Просто теперь в нашем распоряжении не только набор отвёрток и паяльник, но и грёбаный промышленный робот-манипулятор. И наша задача — научиться им управлять, чтобы строить не скворечники, а космические корабли.

Совет от гаражного кодера:

— **Промпт — это новое ТЗ.** Начни думать о своих запросах к AI не как о вопросах в Google, а как о техническом задании для стажёра. Чем чётче и детальнее ты опишешь задачу, тем меньше говнокода получишь в ответ.

— **Прокачивай скилл ревьюера.** Каждый сгенерированный кусок кода рассматривай под микроскопом. Задавай себе вопросы: «Это безопасно? Это оптимально? Это читаемо? Какие здесь могут быть пограничные случаи?». Твой главный скилл теперь — критическое мышление.

— **Не теряй хватку.** Хотя бы раз в неделю пиши что-то с нуля, без помощи AI. Небольшую утилиту, скрипт, компонент. Это как тренировка в спортзале. Ты должен держать свои «кодерские мышцы» в тонусе, чтобы не разучиться ходить без костылей.

— **Думай архитектурно.** Начни больше читать не про конкретные фреймворки, а про паттерны проектирования, принципы SOLID, системный дизайн. AI может построить стену из кирпичей, но план всего здания должен быть в твоей голове.

— Мой первый фейл с Copilot: как я чуть не запушил в прод код, который сам не понимал

Это было в те золотые времена, когда GitHub Copilot только-только выкатился из беты. Я установил его, и мир перевернулся. Это было похоже на то, как если бы ты всю жизнь ковырялся в движке старенького «Москвича», а потом тебе дали ключ от гаража с инструментами из Формулы-1.

Первые пару дней я был в эйфории. Я писал комментарии, а Copilot, как мой личный джинн, материализовывал код. Бойлерплейт? Tab. Юнит-тесты? Tab. Сложный алгоритм, который я гуглил бы полчаса? Два раза Tab и немного магии. Я чувствовал себя не просто кодером, а грёбаным Нео, который загрузил в мозг «Кунг-фу для программистов». Моя производительность взлетела до небес. Я закрывал таски в Jira со скоростью света, и тимлид уже начал коситься на меня с подозрением: не сижу ли я на каких-то запрещённых ноотропах?

И вот на этой волне самоуверенности и технологического экстаза мне прилетела задача. Обычная, рутинная хрень: нужно было написать скрипт для генерации отчёта. Взять массив объектов-пользователей из API, отфильтровать активных за последний год, сгруппировать по отделам и посчитать среднее количество их коммитов.

Дедлайн — «вчера». Классика.

Я сел за ноут, налил кофе. «Так, — думаю, — сейчас я эту задачку щёлкну за 15 минут, а остаток дня буду смотреть мемы про JS».

Я написал сигнатуру функции и комментарий-описание:

```
// Функция принимает массив пользователей, фильтрует активных за последний год,
// группирует по департаментам и считает среднее количество коммитов.
function generateDepartmentReport (users) {
// … тут должен быть код
}
```

Я даже не успел дотянуться до мышки, как Copilot серой подсветкой предложил мне решение. И это был не какой-то там for loop. Это был… шедевр. Элегантная, как балерина, цепочка из. filter (),.map () и. reduce (). Компактная, функциональная, красивая. Такой код не стыдно показать на конференции по функциональному программированию. Он выглядел так, будто его написал бородатый хипстер на «маке», попивая смузи из авокадо.

Я пробежался по нему глазами. Вроде всё логично. Вот фильтр по дате last_login. Вот группировка с помощью reduce, где аккумулятором служит объект. Всё выглядело настолько умно, что мой собственный мозг сказал мне: «Братан, расслабься, этот парень из кремниевой долины шарит лучше тебя».

Я создал моковый массив с двумя пользователями, подставил его в функцию. Запустил. Результат — правильный. «Ну, гениально же!» — подумал я. Скопировал код, который даже не я написал, и без тени сомнения запушил его в ветку фичи, создал пулл-реквест и написал тимлиду: «Готово, ревьюй».

И вот тут, братан, должен был прозвучать тревожный звоночек. Но я его не услышал. Я был опьянён скоростью. Я получил результат, не потратив ни калории умственной энергии. Я переложил самую сложную часть работы — *думать* — на машину.

Тимлид, к счастью, был старым, прожжённым волком, который пережил и флеш, и jQuery, и ангуляр первой версии. Он посмотрел на мой PR и оставил один-единственный комментарий: «Выглядит элегантно. А что будет, если массив users будет пустым? А если у юзера поле last_login будет null? А если у него не будет поля commits?».

У меня ёкнуло сердце.

Я, как прилежный ученик, пошёл проверять.

— Пустой массив. Кидаю [] в функцию. Скрипт падает с ошибкой TypeError: Cannot read properties of undefined (reading «…') где-то в недрах моего красивого. reduce (). Оказывается, Copilot не задал начальное значение для аккумулятора. Классический косяк.

— Пользователь без даты логина. Создаю юзера, у которого last_login: null. Снова TypeError. Код без проверки лезет в свойства null.

— Пользователь без коммитов. Создаю юзера без поля commits. На выходе получаю NaN в среднем значении. Отчёт для бизнеса с таким показателем — это просто бомба.

Я сидел и смотрел на этот «гениальный» код, и мне стало стыдно. Я чуть не запушил в прод мину замедленного действия. И самое паршивое было не то, что в коде были баги. Баги есть у всех. Самое паршивое было то, что **я не понимал этот код до конца**. Я не мог с уверенностью сказать, как он поведёт себя в каждом пограничном случае, потому что я не прошёл через процесс его создания, через отладку, через мысленные эксперименты. Я просто взял чужое, красивое на вид решение, не заглянув под капот.

Я удалил этот элегантный однострочник к чертям собачьим. И написал своё решение. Да, оно было в три раза длиннее. Да, там был старый добрый for… of. Да, там были уродливые if-проверки на null и undefined. Но, чёрт возьми, я знал каждую его строчку. Я мог поклясться своей коллекцией стикеров с GitHub, что он обработает любую дичь, которую ему подсунет API.

Я запушил новую версию, и тимлид молча её апрувнул. Он всё понял без слов.

Этот день стал для меня холодным душем. Я осознал главную истину новой эпохи:

AI-ассистент — это не твой мозг. Это твой самый быстрый, самый продуктивный, но при этом самый наивный и безответственный джуниор-стажёр.

Он напишет код быстро. Он может даже написать его красиво. Но он не будет думать об edge-кейсах. Он не будет думать о бизнес-логике. Он не будет думать о безопасности. Он просто соберёт пазл из тех миллиардов строк кода, на которых его обучали. И если в этих данных был баг, он с радостью воспроизведёт его в твоём проекте.

С тех пор я выработал для себя правило, которое вытатуировано на моём подсознании: **«Никогда не коммить код, который не можешь объяснить построчно в 3 часа ночи под давлением».**

AI — это наш новый инструмент. Мощный, как перфоратор. Но если ты не умеешь им пользоваться, ты рискуешь пробить не стену, а водопроводную трубу у соседа. Или, что ещё хуже, свою ногу.

Совет от гаражного кодера:

— Принцип «сначала пойми, потом используй». Прежде чем нажать Tab, прочитай предложенный код. Прогони его в голове. Подумай: «А что, если..?». Найди в нём хотя бы один потенциальный косяк. Это тренирует критическое мышление.

— Используй AI для генерации альтернатив, а не конечного решения. Когда застрял, спроси у Copilot: «Как ещё можно решить эту задачу?». Он может подкинуть идею или напомнить о методе, который ты забыл. Но финальную версию собирай сам, как конструктор, из лучших частей.

— Делегируй рутину, а не мышление. Проси AI написать тебе юнит-тесты, сгенерить документацию, накидать boilerplate для компонента. Но логику, ядро твоей фичи, пиши сам. Это та часть работы, где ты всё ещё незаменим.

— Будь параноиком. Всегда исходи из того, что AI-код содержит баги. Твоя задача — найти их до того, как их найдёт QA или, не дай бог, пользователь.

— Вайб> Автогенерация: почему твой мозг — всё ещё главный инструмент

Представь, что у тебя есть самый навороченный в мире GPS. Он прокладывает идеальные маршруты, знает все пробки, камеры и посты ДПС. Он говорит тебе: «Через 200 метров поверните направо». Ты поворачиваешь. «Через 500 метров держитесь левее». Ты держишься. Ты доезжаешь до точки Б быстрее, чем кто-либо другой. Но есть один вопрос: ты вообще знаешь, *куда* ты едешь и, главное, *зачем*?

Автогенерация кода с помощью AI — это и есть этот идеальный GPS. Она гениально решает задачу «как добраться из точки А в точку Б». Но она понятия не имеет, что это за точка Б, нужна ли она нам вообще, и не лучше ли было поехать в точку В, где шашлык вкуснее и девчонки симпатичнее.

Наш мозг, наш кодерский вайб — это то, что позволяет нам быть не просто водителем, а путешественником. Это то, что позволяет нам выбирать цель поездки.

Давай разложим этот «вайб» на конкретные инженерные скиллы, которые ни один AI в ближайшие годы у тебя не отберёт.

1. Чуйка на «код с душком» (Инженерная интуиция)

У каждого опытного кодера есть это чувство. Ты смотришь на код — даже не на свой, на чужой — и у тебя начинает слегка подергиваться глаз. Вроде всё работает, тесты зелёные, синтаксис чистый. Но ты чувствуешь, что здесь что-то не так. Этот странный вложенный цикл. Эта функция на 100 строк с десятью аргументами. Этот комментарий // TODO: fix this later.

Это и есть «чуйка». Она строится на тысячах часов чтения и написания кода, на десятках заваленных проектов и сотнях ночей дебага. Это твой внутренний линтер, который работает на уровне подсознания.

AI такой чуйки лишён. Он обучался на миллиардах строк кода с GitHub, включая тонны говнокода. Для него код — это просто статистика, последовательность токенов. Если в его обучающей выборке было много функций на 100 строк, он без зазрения совести сгенерит тебе ещё одну. Он не почувствует, что этот код будет невозможно поддерживать через полгода. Он не поймёт, что эта архитектура — прямой путь в ад рефакторинга.

Твой мозг видит не просто код. Он видит его будущее. Ты — тот самый механик, который по едва слышному стуку в движке может предсказать, что через 500 километров поршень вылетит через капот. AI — это стажёр, который слышит только то, что написано в инструкции.

2. Понимание «Зачем?» (Бизнес-контекст)

AI решает задачу, которую ты ему ставишь. Но он никогда не спросит: «А может, эту задачу вообще не нужно решать?».

Представь, тебе приходит таск: «Сделать на главной странице карусель с отзывами клиентов». Ты идёшь к Copilot, пишешь промпт, и через пять минут у тебя готова красивая, плавная карусель. Задача выполнена.

А вайб-кодер сначала спросит:

— **Зачем** нам эта карусель? Какую проблему она решает?

— Повысит ли она конверсию? Или она просто будет бесить пользователей и замедлять загрузку страницы?

— Может, вместо карусели лучше показать один, но самый мощный отзыв? Или вообще разместить кейс-стади?

— Кто наша целевая аудитория? Может, им вообще плевать на отзывы, и они хотят видеть цены?

Твой мозг способен выйти за рамки ТЗ. Он может соединить требования бизнеса, потребности пользователя и технические возможности. Он может прийти к продакт-менеджеру и ска-

зать: «Братан, я понимаю, что ты хочешь карусель, но это плохая идея. Вот данные аналитики, вот три причины, почему это не сработает. Давай лучше сделаем вот так — это и проще в разработке, и даст больше профита».

AI на такое не способен. Он — идеальный исполнитель, но никудышный стратег. Твоя ценность — в умении быть не просто руками, а думающим партнёром для бизнеса.

3. Архитектурное видение (Взгляд с высоты птичьего полёта)

AI отлично справляется с локальными задачами. Написать компонент, реализовать функцию, пофиксить баг в пределах одного файла. Он как рабочий, который идеально кладёт кирпичи.

Но кто проектирует всё здание? Кто решает, где будут несущие стены, где пройдёт проводка, а где — канализация? Кто думает о том, как здание будет расширяться через 5 лет?

Это работа архитектора. Твоя работа.

Когда ты добавляешь новую фичу, ты думаешь:

— Как это впишется в общую архитектуру проекта?

— Не создаст ли это «бутылочное горлышко» в производительности?

— Не нарушит ли это принципы SOLID?

— Как это будет взаимодействовать с другими микросервисами?

— Насколько легко это будет тестировать и поддерживать?

AI лишён этого глобального видения. Он может сгенерировать тебе идеальный кусок кода, который при этом будет абсолютно несовместим с остальным проектом. Он может предложить решение, которое прекрасно работает для 10 пользователей, но рухнет под нагрузкой в 1000 юзеров.

Твой мозг — это центральный процессор всего проекта. Он держит в голове общую схему и следит за тем, чтобы каждый новый «кирпич» ложился в нужное место.

4. Креативность и нестандартное мышление

AI — это машина по переработке и комбинированию того, что уже было создано. Он гениальный компилятор и ремиксер. Но он не создаёт по-настоящему нового.

Иногда перед тобой встаёт задача, у которой нет стандартного решения. Проблема, которую не нагуглишь и не найдёшь на Stack Overflow. Именно в такие моменты и проявляется магия кодера. Ты начинаешь комбинировать идеи из разных областей, проводить аналогии, экспериментировать. Ты можешь придумать элегантный хак или совершенно новый алгоритм.

Это и есть творчество. Твой мозг способен на «эвристический скачок», на озарение. AI работает по шаблонам. Да, эти шаблоны очень сложные, но это всё ещё шаблоны.

Твой вайб — это способность посмотреть на проблему под совершенно другим углом и сказать: «А что, если мы сделаем всё наоборот?».

Так что же нам делать?

Расслабиться и получать удовольствие. AI забирает у нас самую скучную, самую рутинную часть работы. Он освобождает наш мозг от необходимости помнить синтаксис и писать бесконечный бойлерплейт.

И это даёт нам невероятную возможность. Возможность тратить 100% нашего времени на то, что действительно важно: на архитектуру, на решение сложных проблем, на общение с командой, на творчество. На наш вайб.

Автогенерация — это круто. Но она всего лишь инструмент. Как молоток. Можно забить гвоздь, а можно ударить себе по пальцу. Настоящий мастер ценится не за молоток, а за то, что в его голове есть чертёж прекрасного дома, который он собирается построить.

Твой мозг, братан, — это и есть этот чертёж. И он всё ещё твой главный инструмент.

Совет от гаражного кодера:

— Практикуй «пустой лист». Хотя бы раз в неделю решай какую-нибудь задачу на LeetCode или Codewars вообще без AI. Начни с чистого листа и дойди до решения только своей головой. Это как отжимания для мозга.

— Задавай себе вопрос «Почему?» пять раз. Прежде чем бросаться писать код (или генерить его), попробуй докопаться до первопричины задачи. Почему мы это делаем? Какую боль пользователя это лечит? Это прокачает твой продуктовый вайб.

— Читай код, а не только пиши. Найди на GitHub крутой опенсорс-проект (например, из той же экосистемы, в которой ты работаешь) и просто читай его. Пытайся понять, почему архитектура сделана именно так. Это развивает ту самую «чуйку».

— Стань главным критиком AI. Преврати это в игру. Каждый раз, когда Copilot предлагает тебе код, твоя первая реакция должна быть: «Окей, а теперь давай найдём, где ты накосячил». Ищи баги, пограничные случаи, неоптимальности. Это лучший тренажёр для твоего внутреннего сеньора.

Глава 2. Сетап для вайб-кодера: VS Code + Copilot vs. Cursor

— Путь 1: Классика с апгрейдом. Ставим VS Code, накатываем GitHub Copilot

Выбрать этот путь — это как сказать: «Я не хочу готовую тачку из салона. Я возьму проверенный временем, надёжный кузов, а потом сам вкорячу в него турбину, настрою подвеску и поставлю спойлер». Наш кузов — это Visual Studio Code. Наша турбина — GitHub Copilot.

Почему это вайбовый выбор?

— **Полный контроль.** Ты решаешь всё. Каждый плагин, каждая настройка, каждый хоткей. Твоя IDE — это продолжение твоего мозга, а не чёрный ящик.

— **Гибкость.** Сегодня тебе нужен Copilot, завтра ты его выключил и работаешь с другим инструментом. VS Code — это платформа.

— **Огромное комьюнити.** Под VS Code есть миллион плагинов для всего на свете, от форматирования кода до управления Spotify. Ты можешь собрать себе не просто IDE, а настоящий командный центр.

Итак, погнали. Собираем наш сетап по шагам.

Шаг 1. Фундамент: ставим VS Code

Если у тебя его ещё нет (серьёзно, братан?), то это первое, что нужно сделать. Идёшь на code.visualstudio.com, качаешь, ставишь. Это наша чистая, свежеокрашенная рама. Сама по себе она уже хороша, но мы же хотим скорости.

Шаг 2. Турбонаддув: накатываем GitHub Copilot

Теперь прикручиваем к нашему движку AI-мощь.

— Открывай VS Code.

— Слева на панели есть иконка с квадратиками (Extensions). Жми на неё. Или просто нажми Ctrl+Shift+X.

— В строке поиска вбивай GitHub Copilot.

— Ты увидишь несколько плагинов. Нам нужны два:

— **GitHub Copilot:** Это сам движок, который даёт умные подсказки прямо в коде (inline suggestions).

— **GitHub Copilot Chat:** Это твой личный AI-кореш в чате. С ним можно болтать, просить объяснить код, генерить тесты и рефакторить куски. Ставь оба, не ошибёшься.

— После установки VS Code попросит тебя залогиниться через GitHub. Логинься. Тебе понадобится активная подписка на Copilot (студентам часто дают бесплатно, проверь GitHub Student Developer Pack).

Всё. Турбина установлена. Открой какой-нибудь JS-файл, напиши комментарий // function that sums two numbers и подожди секунду. Ты увидишь серый «призрачный» текст с готовой функцией. Нажми Tab. БУМ! Магия. Добро пожаловать в будущее.

Но постой… Эта магия может быть назойливой. Copilot начнёт лезть со своими советами везде, даже когда ты просто хочешь написать console. log. Он будет предлагать код в. md файлах, бесить тебя в конфигах. Он как слишком энергичный стажёр, который постоянно заглядывает через плечо.

Пора его приручить.

Шаг 3. Тонкая настройка: пилим settings. json, чтобы Copilot не бесил

Мы не хотим, чтобы турбина работала на полную мощность, когда мы паркуемся. Нам нужен контроль. Открываем настройки в формате JSON. Это наш «мозг» IDE.

— Нажми Ctrl+,, чтобы открыть настройки.

— В правом верхнем углу есть иконка файла с загнутым уголком. Наведи на неё, появится подсказка «Open Settings (JSON)». Жми.

Перед тобой файл settings. json. Всё, что мы сюда напишем, переопределит дефолтные настройки. Вот готовый конфиг, который я использую сам. Копируй и вставляй. Ниже я разложу по полочкам, что каждая строчка делает.

```
{
// — - Общие настройки для комфортной работы — —
"editor.fontSize»: 14,
"editor.fontFamily»: ««Fira Code', «JetBrains Mono', «Consolas', 'monospace'»,
"editor.fontLigatures»: true,
"workbench.colorTheme»: «Default Dark+», // или любая твоя любимая тема
"files.autoSave»: «onFocusChange»,


// — - Начинается магия для COPILOT — —


// 1. Главный рубильник. Иногда хочется подумать в тишине.
"github.copilot. enable»: {
«*»: true, // Включаем для всех языков по умолчанию
«plaintext»: false, // Выключаем для обычных текстовых файлов
«markdown»: false, // Выключаем для Markdown, чтобы не мешал писать доки
«json»: true // Для JSON пусть работает, помогает со скобками
},


// 2. Убираем мусор из поля зрения AI.
// Copilot учится на открытых файлах. Не давай ему учиться на мусоре.
«files. exclude»: {
"**/.git»: true,
"**/.svn»: true,
«**/.hg»: true,
«**/CVS»: true,
«**/.DS_Store»: true,
«**/Thumbs. db»: true,
«**/node_modules»: true, // Самое важное!
«**/bower_components»: true,
«**/dist»: true,
«**/build»: true
},
«search. exclude»: {
«**/node_modules»: true,
«**/bower_components»: true,
"**/*.code-search»: true
```

```
},

// 3. Отключаем встроенные подсказки VS Code, чтобы не конфликтовали с Copilot.
// Пусть будет один главный по подсказкам.
"editor.inlineSuggest. enabled»: true,

// 4. Тюним автодополнение, чтобы было удобно.
// По умолчанию он может предлагать код, когда ты этого не ждешь.
// Эта настройка (увы, пока нет прямого конфига) решается привычкой жать Esc.
// Но мы можем настроить, когда автокомплит появляется сам:
«editor. quickSuggestions»: {
«other»: «on»,
«comments»: «on»,
«strings»: «on»
},

// 5. Горячая клавиша для чата Copilot. `Ctrl+I` по умолчанию — топ.
// Можно переназначить, если мешает.
// «keybindings. json»
}
```

Разбор полётов по конфигу:

— **github.copilot. enable:** Это твой главный тумблер. Ты можешь точечно отключать Copilot для определённых типов файлов. Я всегда вырубаю его для markdown, потому что когда я пишу документацию или эту книгу, последнее, что мне нужно, — это предложения сгенерить for loop.

— **files. exclude и search. exclude:** Критически важные настройки! Папка node_modules — это чёрная дыра, в которой лежат миллионы файлов. Если AI начнёт их анализировать, твоя IDE превратится в черепаху. Этим конфигом мы говорим: «Братан, вот сюда не смотри, там ничего интересного».

— **editor.inlineSuggest. enabled:** Убедись, что эта опция включена, иначе магия «призрачного текста» работать не будет.

— **editor. quickSuggestions:** Эта настройка контролирует появление стандартного окошка с подсказками. Важно, чтобы оно не конфликтовало с Copilot. Настройки «on» для строк и комментов позволяют Copilot лучше «понимать» контекст, когда ты начинаешь писать.

Шаг 4. Хоткеи: приручаем зверя окончательно

Инструмент бесполезен, если ты не умеешь им пользоваться быстро. Заучи эти хоткеи, как git push. Это твоя новая мышечная память.

— Tab — Принять предложение. Твой лучший друг.

— Esc — Отклонить предложение. Твой второй лучший друг.

— Alt +] (или Option +] на Mac) — Показать следующее предложение.

— Alt + [(или Option + [на Mac) — Показать предыдущее предложение. (Многие не знают об этой фиче, а она — золото!)

— Ctrl + Enter — Открыть панель с 10 вариантами кода от Copilot. Полезно, когда первая предложенная идея — мусор.

— Ctrl + I — Открыть встроенный чат Copilot. Выдели кусок кода, нажми Ctrl+I и напиши «Refactor this» или «Explain this code». Это киллер-фича.

Вот и всё, братан. Твой классический, кастомный, но заряженный AI сетап готов. Ты получил мощь нейронки, но оставил за собой полный контроль. Ты сам решаешь, когда нажать на газ, а когда — притормозить и подумать своей головой. Это путь воина, который доверяет своему мечу, но полагается на собственное мастерство.

Совет от гаражного кодера:

— Поставь Prettier и ESLint. AI часто генерит код с кривым форматированием или не по стайлгайду проекта. Эти два плагина — твои автоматические «чистильщики». Они будут форматировать код (включая сгенерированный) при сохранении. Так в репозитории будет порядок.

— Создай файл. copilotignore. Да, такой существует! По аналогии с. gitignore. Создай его в корне проекта и пропиши пути к файлам или папкам, которые Copilot должен игнорировать полностью. Это более тонкий контроль, чем settings. json, и он работает на уровне проекта. Например, туда можно добавить файлы с секретами или ключами API (.env, secrets. js).

— Пиши «говорящие» комментарии. Copilot очень хорошо реагирует на комментарии. Вместо того чтобы ждать, пока он угадает твои мысли, напиши подробный коммент, что должна делать функция. // function that fetches users from API, filters by age> 18 and returns only their names. Ты удивишься, насколько точным будет сгенерированный код.

— Не бойся «спорить» с чатом. Используй Copilot Chat как спарринг-партнёра. Спроси его: «Почему это решение лучше, чем [твой вариант]?». Иногда он подкинет дельный аргумент, а иногда ты увидишь, что его логика хромает, и утвердишься в своей правоте. Это круто прокачивает скилл аргументации.

— Путь 2: Всё включено. Разбираем Cursor — IDE, где AI уже под капотом

Если VS Code с Copilot — это кастомная тачка, которую ты собираешь в гараже, то Cursor — это Tesla, которую ты забираешь прямо с завода. Она выглядит знакомо, руль на том же месте, но под капотом (которого нет) — совершенно другая философия. AI здесь не гость с плагином, а хозяин дома. Он не прикручен сбоку, он вплетён в саму ДНК этой IDE.

Что такое Cursor и почему это не «просто ещё один редактор»?

Начнём с главного: **Cursor — это форк VS Code.** Это значит, что ты не попадёшь в чужой, незнакомый мир. Все твои любимые темы, расширения (да, Prettier и ESLint встанут как родные) и хоткеи будут работать. Интерфейс тебе до боли знаком. Это как пересесть из Volkswagen Golf в Volkswagen e-Golf. Снаружи — почти то же самое, но внутри — тишина, мощь и технологии.

Ключевое отличие в том, что разработчики Cursor взяли опенсорсный код VS Code и пересобрали его вокруг искусственного интеллекта. AI здесь — это не фича. Это фундамент.

Шаг 1. Получи ключи и поехали: установка и первый запуск

Здесь всё до безобразия просто, и в этом первая прелесть этого пути.

— Идёшь на cursor.sh.

— Качаешь версию под свою ОС.

— Ставишь.

— Открываешь.

Всё. Никаких поисков расширений. Никаких отдельных логинов в GitHub для Copilot. Cursor сразу тебя встретит и предложит залогиниться. После этого AI-функции просто… работают. Это тот самый «вау-эффект», как когда ты первый раз видишь, как Tesla сама паркуется. Ты ещё ничего не сделал, а магия уже происходит.

Шаг 2. Что под капотом у этой тачки? Разбираем киллер-фичи

А теперь — мясо. То, ради чего мы здесь собрались. Что именно делает Cursor таким особенным?

1. Ctrl+K (или Cmd+K на Mac) — Твоя волшебная палочка

Это главная фишка, которая меняет правила игры. В связке VS Code + Copilot ты пишешь коммент и ждёшь, пока AI предложит код. Это пассивный процесс.

В Cursor ты действуешь активно. Ты выделяешь код (или просто ставишь курсор на пустое место) и нажимаешь Ctrl+K. Появляется небольшое поле для ввода. И вот тут начинается дискотека.

— Генерация с нуля:

— Ставишь курсор в пустой файл, жмёшь Ctrl+K и пишешь: *«Создай React-компонент UserProfile, который принимает пропсы name, avatarUrl и bio. Используй стили Tailwind CSS.»*

— Нажимаешь Enter. И Cursor прямо в редакторе, на твоих глазах, пишет готовый компонент. Не серым призрачным текстом, а по-настоящему.

— Рефакторинг и редактирование:

— Это ещё мощнее. Допустим, у тебя есть уродливая функция с кучей if-else.

```
function getUserStatus (user) {
if (user.isBanned) {
return «Banned»;
} else {
if (user.lastLogin <new Date («2024-01-01»)) {
return «Inactive»;
} else {
return «Active»;
}
}
}
```

Ты выделяешь всю эту функцию, жмёшь Ctrl+K и пишешь: *«Перепиши это с помощью тернарного оператора, сделай код чище»*.

Хлоп! Cursor заменяет твой код на элегантный вариант:

```
function getUserStatus (user) {
return user.isBanned
?«Banned»
: user.lastLogin <new Date («2024-01-01»)
?«Inactive»
: «Active»;
}
```

— Это как будто у тебя есть личный сеньор-помощник, который по команде чинит твой код. Ты больше не ждёшь предложений, ты отдаёшь приказы.

2. Чат, который знает твой проект (Ctrl+L)

В VS Code у тебя есть Copilot Chat. Он хорош. Но чат в Cursor — это его версия на стероидах. Почему? Потому что он умеет работать с контекстом твоего проекта через @-символы.

Ты открываешь чат (Ctrl+L) и можешь написать что-то вроде:

— «Эй, сравни файлы @components/LoginForm. js и @components/RegisterForm. js и скажи, какой код можно вынести в общий компонент.»

— «Найди все использования функции getUserData во всём проекте (@workspace) и предложи, как её лучше оптимизировать.»

— «Объясни мне, как работает файл @services/api. js. Какие основные функции он экспортирует?»

Он не просто отвечает на общие вопросы. Он читает твой код, анализирует его и даёт ответы, основанные на *твоём* проекте. Это как будто твой тимлид, который помнит каждую строчку кода, сидит с тобой в чате 24/7.

3. Встроенный AI-дебаггер и фиксер

Когда в твоём коде появляется ошибка (красная волнистая линия), рядом с ней в Cursor часто появляется маленькая кнопка… «Fix with AI». Ты нажимаешь на неё, и AI пытается проанализировать ошибку и предложить исправление.

Это не всегда работает идеально, но когда работает — это чистый кайф. Особенно для новичков, которые могут часами тупить над ошибкой Cannot read properties of undefined. Cursor может просто подсказать: «Братан, ты забыл проверить, что user не null, прежде чем лезть в user.name». Это экономит тонны времени и нервов.

Сравнение лоб в лоб: VS Code + Copilot vs. Cursor

Давай сведём всё в таблицу, как будто сравниваем две тачки для тест-драйва.

Критерий	VS Code + Copilot (Кастом)	Cursor (Tesla)
Установка и настройка	Нужно ставить всё отдельно, тюнить `settings.json` . Больше контроля, но и больше возни.	"Скачал и поехал". Идеально для тех, кто не хочет париться.
Генерация кода	Пассивная (ждёшь предложений). Хорошо для автодополнения.	Активная (`Ctrl+K`). Отдаёшь приказы. Лучше для создания и рефакторинга целых блоков.
Рефакторинг	Можно через чат, но требует копипасты.	Встроен в `Ctrl+K` . Выделил, написал, что сделать, — готово. Гораздо быстрее.
Чат и вопросы	Copilot Chat — мощный, но без глубокой индексации проекта.	Чат с `@` -символами. Понимает структуру твоего проекта, может работать с файлами и папками. На голову выше.
Контроль и кастомизация	Максимальный. Ты — хозяин своей IDE.	Меньше контроля над самим AI (он глубоко встроен). Но вся кастомизация VS Code доступна.
Цена	Платишь за подписку GitHub Copilot.	Есть бесплатный тариф с ограниченным количеством запросов. Pro-тариф стоит денег, но часто использует модели GPT-4,

Так что выбрать новичку? Кому давать ключи от Tesla?

А вот и главный вопрос. И ответ на него не такой однозначный.

Аргументы «ЗА» Cursor для новичка:

— **Низкий порог входа:** Не нужно ничего настраивать. Это снимает огромный пласт головной боли и позволяет сразу сфокусироваться на коде.

— **«Вау-эффект» и мотивация:** Когда ты видишь, как по твоей простой команде рождается целый компонент, это дико мотивирует. Ты чувствуешь себя всемогущим.

— **Обучение на практике:** Функция «Explain this code» и встроенные фиксы ошибок — это как персональный репетитор, который помогает разобраться в сложных моментах.

Аргументы «ПРОТИВ» (и это важно!):

— **Риск стать «обезьянкой с гранатой»:** Слишком большая сила в руках того, кто не понимает основ. Новичок может начать генерить тонны кода, не вникая в его суть. Это путь к фейлу, который я описывал в прошлой главе.

— **Магия вместо понимания:** Когда IDE сама чинит твои ошибки, есть риск, что ты так и не научишься их чинить сам. Ты не пройдёшь через боль дебага, а значит, не нарастишь ту самую «инженерную чуйку».

— **Зависимость:** Привыкнув к Ctrl+K, будет сложно пересесть на другую IDE, где такой магии нет (например, на работе у тебя может быть стандартный VS Code).

Вердикт:

Cursor — **феноменальный инструмент для новичка, но с одним большим «НО»**. Его нужно использовать не как костыль, а как тренажёр.

— Сгенерил код? Не беги дальше. Сядь и разберись в каждой строчке. Спроси у чата: «А почему ты использовал здесь. reduce (), а не for loop?».

— IDE починила твой баг? Не радуйся, что избавился от проблемы. Посмотри diff, сравни «было» и «стало». Пойми, в чём была твоя ошибка, чтобы не совершить её снова.

Cursor для новичка — это как езда на машине с инструктором, у которого есть свои педали. Он поможет тебе в сложной ситуации, но твоя задача — научиться водить так, чтобы его помощь была не нужна.

Совет от гаражного кодера:

— **Не удаляй VS Code.** Держи обе IDE. Используй Cursor для быстрого прототипирования и сложных рефакторингов. А потом открывай тот же проект в обычном VS Code, чтобы «почувствовать» код без AI-магии и убедиться, что ты всё ещё контролируешь ситуацию.

— **Используй @web в чате Cursor.** Это малоизвестная, но убойная фича. Ты можешь попросить его поискать информацию в вебе. Например: *"@web найди лучшую библиотеку для создания чартов в React и напиши пример её использования»*. Он сходит в гугл за тебя.

— **Настрой свой AI.** В настройках Cursor можно выбрать, какие модели использовать (например, быстрый Claude Sonnet или мощный GPT-4o). Поиграйся с ними. Для простых задач хватит быстрых моделей, для сложной логики переключайся на более умные.

— **Помни: Ctrl+K — это диалог.** Если первый результат тебе не понравился, не удаляй его. Просто нажми Ctrl+K ещё раз и напиши уточняющий промпт: «Сделай это проще», «Добавь обработку ошибок», «Используй TypeScript». Он будет итерировать поверх существующего кода.

— Путь 3: Облачный вайб. Replit — кодим в браузере с AI-ассистентом

Вступление: когда твой гараж — это ссылка, а не физический адрес

Давай честно — ты хоть раз мечтал, чтобы можно было сидеть на паре, в поезде или даже в туалете и фигачить код без лишней возни с установками? Чтобы не было проблем типа «у меня винда, а у вас мак», чтобы не нужно было настраивать ноду/питон/бабель/webpack и вот это всё, а было только: «Открыл сайт — и в бой!»

Вот именно для этого и существует Replit. Это не просто онлайн-редактор: это твоя полноценная облачная среда, которая сейчас ещё и с АИ-ассистентом под капотом (их фирменный Gun и Ghostwriter). Всё, что тебе нужно — браузер и интернет. Остальное делает Replit.

Почему Replit — реальный вайб для гаражного хакера и стартапщика

1. Кодишь хоть с холодильника

Ты реально можешь начать проект просто с телефона или старого хромбука. Всё хранится в облаке, твоя IDE всегда под рукой, не надо париться с обновлениями и зависимостями. Это прямо как Dropbox для кода: проект всегда свежачок, нигде ничего не слетает. Просто залогинился — и вперед.

2. Быстрый старт — меньше минуты

Запустить новый проект — как сделать чашечку рамена. Заходишь на replit.com, жамкаешь «New Repl», выбираешь язык (Node. js, Python, Rust, что угодно), называешь проект — и сразу в браузере получаешь полный редактор, встроенный терминал, file tree, package manager и кнопку RUN. Всё автоматически билдится и крутится. Даже сервер Node. js с Express поднимается одной кнопкой.

3. AI под капотом — Ghostwriter

AI у Replit называется **Ghostwriter**. Это твой новый кореш-подсказчик: помогает писать функции, объясняет непонятное, рефакторит, пишет тесты, дебажит и даже… генерит целые функции по простому описанию! Работает прямо в редакторе, советует код на лету или по prompt-запросу.

Типовые фишки:

— Автокомплит в стиле Copilot — написал коммент, получил функцию.

— AI chat. Спрашиваешь: «Почему у меня не работает fetch?» или «Перепиши функцию на async/await», и он реально отвечает.

— Генерация описания к коду, объяснение ошибок, поиск багов.

— Помогает с документацией и юнит-тестами по описанию.

4. Встроенный деплой

Кнопка «Deploy» — и вот уже твой пет-продакшен в интернете. Не надо возиться ни с Vercel, ни с Netlify, ни с VPS. Прототип заработал — кидай ссылку друзьям, инвестору, выкатывай демку для портфолио.

5. Простота коллаборации

Жмёшь Share — и даёшь ссылку корешу; вы вдвоём хачите код в реальном времени. Нет больше гемора с Git, merge conflict'ами, чужими package-версими и «оно не билдится у меня на машине». Всё синхронизировано.

6. Marketplace и Bounties

Находишь интересный проект или задание — можно сразу хакать чужие реплы, делать Pull Requests, даже заработать бабла на простых фичах или багах.

Плюсы Replit для пет-проектов

— **Лёгкий старт**: не надо ставить ничего локально, живи просто — сразу кодишь.

— **AI-ассистент уже встроен**, не надо покупать Copilot отдельно.

— **Быстрый деплой**: твой пет-проект сразу в интернете, можно показать на собеседовании ещё до заливки на GitHub Pages.

— **Идеальное решение для учебы и экспериментов**: хочешь потестить питоновский скрипт или JS-SDK — запускай тут и сейчас.

— **Коллаборации без боли**: прямо как Google Docs для кода.

— **Поддержка десятка языков из коробки**: питон, нода, руст, даже Bash.

Где появляются минусы?

1. Производительность и ограничения

Проекты запускаются на виртуалке в облаке, а не у тебя на ноуте. Это значит, что:

— Если проект тяжёлый — сборка, деплой и тесты идут медленнее, чем на ноуте с норм железом.

— Сервисы «засыпают» через пару минут неактивности (без платной подписки), то есть для демки — топ, для продакшена — уже так себе.

2. Платные фичи

AI-ассистент Ghostwriter и приватные реплы — только для подписчиков. Бесплатно дают поюзать, но если реально хочешь качать — готовь пару баксов в месяц.

3. Не полноценная IDE для энтерпрайза

Пишите пет-проекты, тестируйте, учитесь — кайф. Но если у тебя нужно собрать огромную монолитную систему на 20 микросервисов и 1000 депенденси — лучше брать локальную разработку с привычным Docker, Nx, etc.

4. Немного магии, мало контроля

Здесь за тебя решают, какой линтер, какой билд. Да, можно настраивать файлы, но если нужен хардкорный CI/CD, кастомная линейка dev-стейджингов — Replit покажется детским бассейном.

5. Файловые лимиты

На бесплатном тарифе реплы ограничены по размеру, как и ресурсы процесса. Иногда что-то не билдится из-за лимита, или база данных маленькая по размеру.

Честный гаражный разбор: кому и зачем подходит Replit?

— Новичок:

— Лучшее место, чтобы почувствовать себя настоящим кодером. Ты не тратишь два дня на установку npm и поиск правильной версии Node. js. Всё уже готово, можно писать «Hello, World!» через 1 минуту после регистрации.

— Проект под собеседование:

— Залей сюда pet-проект, посади минимальное API, покажи в браузере без установки, инвестор/компания охренеет.

— Коллаборация на лету:

— Когда срочно нужно скинуть другу фрагмент кода или быстро допилить проект перед дедлайном в универе.

— Быстрые прототипы:

— Если хочется накидать идею, проверить фреймворк или библиотеку, сделать MVP — Replit реально экономит время и нервы.

— Менторство и обучение:

— Ты можешь в один клик пересылать ссылку студенту/ментору, давать задание, получать отзыв прямо в облаке.

Как быстро ворваться в Replit с AI-ассистентом (мини-гайд)

— Регистрируешься на replit.com (быстрая авторизация через GitHub).

— Жмёшь «Create Repl», выбираешь нужный язык (например, Node. js для JS-проекта).

— Вводишь название проекта и описание (чтобы самому потом не забыть, что здесь).

— Внутри сверху сразу есть кнопка «Ask AI» — это ты и есть Ghostwriter. Пиши комменты, получай готовые фрагменты, проси объяснить ошибку или переписать на async/await.

— Кнопка «Run» — твой короткий путь к магии: сразу видишь результат в браузере, логи сервера — всё в одной вкладке.

— «Share» — накинь ссылку другу. Можно делать публичные и приватные проекты (платно).

— «Deploy» — твой пет-прод прям летит в интернет за пару кликов.

Реально рабочий низкоуровневый пример:

// Просто сгенери функцию с помощью Ghostwriter:
// Напиши коммент:
// TODO: Функция, генерящая JWT токен для юзера
// — и тут же прилетает код с правильным импортом, обработкой ошибок и expiry.

// Запусти, протестируй — вся магия онлайн.

TL; DR — Сравнительная табличка VS Code / Cursor / Replit

Критерий	VS Code + Copilot	Cursor (форк VS Code + AI)	Replit (облако + AI)
Где живёт	Локально	Локально	Браузер/облако
AI-ассистент	Copilot	Встроенный "ближе к ядру"	Ghostwriter
Быстрый старт	Нужно ставить всё ручками	Всё из коробки, привычно	1 минута, всё онлайн
Мощь/Контроль	Максимум	Высоко	Меньше всего
Легко для новичка	Порог выше	Очень легко	Элементарно
Деплой	Нужно внешнее решение	Как обычно в VS Code	Кнопка "Deploy"
Коллаборация	Через Git	Как в VS Code	Одним кликом, как Google Docs
Платные фичи	Copilot	Pro-тариф на AI	Pro на AI/ресурсы
Честное место живого вайба	Для больших и сложных проектов	Для прокачки в AI-вайбе, рефакторинга	Для быстрой идеи, учебы, MVP, коллабораций

Совет от гаражного кодера:

— **Не жалей старых привычек — пробуй новое!** Даже если ты кодишь 10 лет в VS Code, попробуй Replit для пет-проектов. Это реально освобождает от головняка с зависимостями и билдом.

— **Держи проекты небольшими.** Для pet-проектов и учебных задач — идеально. Но если проект начинает пухнуть, разбивай на микросервисы или пушь на GitHub и забирай локально.

— **Юзай AI по делу.** Ghostwriter крут, но не давай ему писать всю бизнес-логику. За генератор боилерплейта и проверку синтаксиса — топ. Критические фичи — только своими руками.

— **Экспериментируй с разными языками.** Здесь реально можно попробовать Python, Go, Rust, Bash — не надо ничего ставить! Это огромный плюс для роста как инженера.

— **Деплой сразу на демку.** Хочешь показать работу — кнопка «Deploy» выручит. Но для настоящего продакшена потом лучше собирать свой пайплайн.

Помни: облачный вайб — это не второй сорт, а новый способ быстро запускать идеи и учиться фаново. Где бы ты ни был: хоть в кафешке, хоть в поезде. Это и есть настоящий гараж эпохи AI — границ больше нет.

Мы в коде!

Глава 3. «Hello, AI!»: Пишем первый код, не касаясь клавиатуры (почти)

— Просим Copilot/Cursor сгенерить «Hello, World!» на JS

Помнишь, как в детстве казалось, что компьютер — это магия? Что там внутри живут какие-то умные гномики, которые всё делают. Что ж, братан, гномики выросли, накачались на миллиардах строк кода с GitHub и теперь готовы работать на тебя.

Подготовка к магии: создаём полигон

Нам нужен чистый лист, холст для нашего первого шедевра, созданного в соавторстве с кремниевым разумом.

— Открой свою IDE (VS Code или Cursor, неважно, магия работает в обоих).

— Создай новый файл. Назови его index. js..js — это как номерной знак, который говорит всем: «Эй, я тачка из страны JavaScript!».

Перед тобой пустой белый экран. Тишина. Лёгкое волнение. Это тот самый момент, когда старая школа достала бы шпаргалку и начала печатать c-o-n-s-o-l-e…, боясь опечататься. Но мы не из старой школы. Мы — пилоты.

Отдаём первый приказ: промпт

Теперь самое интересное. Мы не будем писать код. Мы напишем *инструкцию* для кода. Это называется промпт (prompt). Это язык, на котором мы общаемся с AI.

В файле index. js напиши обычный комментарий на русском или английском. Чем чётче приказ, тем лучше результат.

```
// Напиши простой скрипт, который выводит в консоль сообщение «Hello, World!»
```

И замри на секунду.

Что сейчас происходит? Твоя IDE отправляет этот комментарий и весь контекст (название файла, язык) на серверы GitHub. Там огромная нейронная сеть, обученная на всём коде мира, анализирует твой запрос. Она видит «JavaScript», «выводит в консоль», «Hello, World!». Она вспоминает миллионы раз, когда другие кодеры делали то же самое. И она генерирует наиболее вероятное, наиболее стандартное решение.

И вот оно… прямо под твоим комментарием появляется «призрачный» серый текст. Как будто дух машины предлагает тебе свою помощь.

```
// Напиши простой скрипт, который выводит в консоль сообщение «Hello, World!»
console. log («Hello, World!»);
```

Это оно. То самое чувство. Ты ещё не нажал ни одной клавиши для написания кода, а он уже здесь. Твоя единственная задача — нажать Tab.

Жми.

Бум. Серый текст становится частью твоего файла. Он настоящий. Ты только что сгенерил свою первую строчку кода с помощью AI. Поздравляю, ты сделал первый шаг в новый дивный мир.

Разбор полётов: что за магию мы только что создали?

Окей, эйфория прошла, давай наденем очки сеньора и разберём этот код по косточкам. Просто сгенерить — удел оператора. Вайб-кодер должен понимать, *что* он сгенерил.

```
console. log («Hello, World!»);
```

— **console**: Это объект. Думай о нём как о пульте управления от твоего кода. Это твой главный друг в мире JavaScript, твоё окно в душу работающей программы. Когда что-то идёт не так, ты первым делом кричишь: «Эй, console, покажи, что там у тебя!». Он живёт в браузере и в Node. js — везде, где исполняется JS.

— .**log ()**: Это метод. Функция, которая прикручена к объекту console. Название говорит само за себя — «логировать», «записывать в журнал». У console есть и другие методы (.warn,.error,.table), но. log — это рабочая лошадка для вывода любой информации.

— **«Hello, World!»**: Это строка (string). Аргумент, который мы передаём в метод. log (). Кавычки (одинарные или двойные, это вечный холивар) говорят JavaScript: «Братан, это не переменная, не команда, это просто набор символов. Выведи его как есть».

— ; (**точка с запятой**): О, это причина тысяч сломанных копий и бессонных ночей. В JavaScript она технически необязательна в большинстве случаев (спасибо механизму ASI — Automatic Semicolon Insertion). Но AI сгенерил её. Почему? Потому что это считается хорошей практикой (good practice). Это явный сигнал интерпретатору: «Здесь команда закончилась». AI обучался на коде хороших разработчиков, и он подсовывает тебе их привычки. Считай, что он бесплатно даёт тебе первый урок по стилю кода.

Запуск: оживляем нашего Франкенштейна

Код написан (сгенерирован). Теперь его надо запустить.

— Открой терминал прямо в VS Code (`Ctrl + ``).

— Убедись, что ты в той же папке, где лежит твой файл index. js.

— Напиши команду: node index. js

— Нажми Enter.

И ты увидишь в консоли заветные слова:

Hello, World!

Всё. Круг замкнулся. Ты отдал приказ на естественном языке, машина перевела его в код, а потом другая машина (Node. js) исполнила этот код. Ты — дирижёр этого цифрового оркестра.

А почему именно console. log? Глубокое погружение

А теперь давай копнём глубже. Почему AI выбрал именно этот способ? Новичок мог бы попробовать вывести сообщение через alert («Hello, World!»);. Почему AI не предложил этот вариант?

Потому что AI, как мы уже выяснили, учился на хорошем коде.

— **alert ()** — это старая школа. Это модальное окно, которое **блокирует** весь интерфейс браузера. Пока ты не нажмёшь «ОК», пользователь не сможет ничего сделать на странице. В современном вебе это считается ужасным UX (user experience). Это как если бы официант в ресторане хватал тебя за руку и не отпускал, пока ты не подтвердишь, что услышал специальное предложение дня.

— **console. log ()** — это инструмент разработчика. Он выводит информацию незаметно для пользователя, в специальной консоли. Это профессиональный, «чистый» способ для отладки и вывода информации.

AI знает разницу. Он не предложил тебе alert, потому что в 99% случаев на профессиональных проектах это будет ошибкой. Он сразу учит тебя делать правильно.

А что насчёт document. write («Hello, World!»)? Это ещё один древний артефакт. И AI его тоже избегает. Почему? Потому что document. write () после полной загрузки страницы может стереть всё её содержимое и заменить его твоей строкой. Это как выстрелить из базуки, чтобы убить муху. AI это тоже знает.

Вывод: AI — твой первый ментор по лучшим практикам

Когда ты просишь AI сделать что-то простое, он не просто даёт тебе рабочий код. Он даёт тебе *идиоматичный* код. То есть код, который написан так, как принято в сообществе.

Он показывает тебе самый стандартный, самый безопасный и самый распространённый способ решения задачи.

Это невероятно мощный инструмент для обучения. Вместо того чтобы набивать шишки и учиться на своих ошибках, ты сразу видишь, «как надо».

Но помни про «почти» в названии главы. Ты всё ещё главный.

— **Ты** решил, что нужен файл index. js.

— **Ты** сформулировал задачу в промпте.

— **Ты** нажал Tab, чтобы принять код (а мог бы и отклонить!).

— **Ты** запустил скрипт и проверил результат.

Ты не кодил руками, но ты *управлял* процессом. И в этом — суть новой философии вайб-кодера.

Совет от гаражного кодера:

— **Поиграй с промптами.** Попроси AI сделать то же самое, но по-другому. «Напиши функцию sayHello, которая возвращает „Hello, World!“» или «Создай переменную greeting и выведи её в консоль». Смотри, как он адаптируется. Это научит тебя лучше формулировать мысли.

— **Попробуй другой язык.** Создай файл main.py и напиши тот же комментарий. AI сгенерит тебе print («Hello, World!»). Создай main. go — получишь целый пакет с функцией main. Это самый быстрый способ увидеть синтаксис нового языка.

— **Заставь его ошибиться.** Напиши нарочито кривой промпт. «Выведи привет мир без консоли». Посмотри, что он сделает. Может, он предложит alert, а может, вообще ничего. Понимание ограничений инструмента не менее важно, чем понимание его возможностей.

— **Спроси у чата «Почему?».** Выдели сгенерированную строчку console. log («Hello, World!»);, открой Copilot Chat (Ctrl+I) и спроси: «Почему ты использовал console. log, а не alert?». Ты получишь подробное объяснение, которое закрепит твои знания. Используй AI не только как писателя, но и как репетитора.

— Анализируем, что он выдал: почему именно console. log, а не alert

Давай устроим очную ставку. В нашем полицейском участке три подозреваемых в выводе «Hello, World!» на экран. Мы должны допросить каждого и понять, почему наш AI-детектив выбрал именно первого.

Подозреваемый №1: alert («Hello, World!») — Крикливый паникёр

Представь, ты заходишь в крутой, тихий бар. Играет джаз, люди спокойно общаются. И тут кто-то вскакивает на стол и начинает орать во всё горло: «СПЕЦИАЛЬНОЕ ПРЕДЛОЖЕНИЕ ДНЯ!». Весь бар замирает. Музыка останавливается. Ты не можешь ни заказать напиток, ни поговорить с другом, пока этот тип не закончит свою тираду и ты не кивнёшь ему «Да, я услышал».

Вот это, братан, и есть alert ().

— Что он делает технически?

— Он создаёт модальное окно. «Модальное» — это ключевое слово. Это значит, что оно **блокирует** всё остальное. Пока это окошко висит на экране, пользователь не может взаимодействовать с твоей страницей. Ни проскроллить, ни нажать на другую кнопку, ни выделить текст. Ничего. Весь твой красивый интерфейс превращается в бесполезную картинку на фоне.

— Почему это катастрофа для UX (User Experience)?

— Это грубо. Это навязчиво. Это пережиток веба 90-х, когда сайты мигали, как новогодняя ёлка, и считали это крутым дизайном. В современном мире, где каждая миллисекунда

внимания пользователя на счету, блокировать его — это верный способ заставить его закрыть твою вкладку и больше никогда не возвращаться.

— Когда его (теоретически) можно использовать?

— В очень, ОЧЕНЬ редких случаях. Например, если тебе нужно гарантированно уведомить пользователя о чём-то критически важном, что требует его немедленного подтверждения перед тем, как он сделает что-то необратимое. Например, «Вы уверены, что хотите удалить свой аккаунт навсегда?». Но даже для этого сегодня есть более элегантные решения в виде красивых модальных окон, встроенных в интерфейс.

AI знает всё это. Он не предложил тебе alert, потому что он обучался на современном, профессиональном коде, где alert для отладки или простого вывода информации — это признак дурного тона. Он как бы говорит тебе: «Братан, давай не будем кричать на наших пользователей. Мы же не на рынке».

Подозреваемый №2: document. write («Hello, World!») — Бульдозерист-неадекват

А этот парень ещё опаснее. Представь, ты построил красивый дом. Поклеил обои, расставил мебель. И тут тебе нужно повесить на стену картину «Hello, World!». Ты зовёшь мастера, а он приезжает на бульдозере, сносит твой дом до основания и на руинах ставит табличку с твоей картиной.

Это — document. write ().

— Что он делает технически?

— Он пишет контент прямо в HTML-документ. Звучит безобидно, правда? Но дьявол, как всегда, в деталях, а точнее — во времени исполнения.

— Если ты вызываешь document. write () *во время* загрузки страницы, он просто добавляет твой текст в то место, где был вызван.

— Но если ты вызываешь его *после* того, как страница полностью загрузилась (например, по клику на кнопку), происходит катастрофа. JavaScript сначала выполняет неявную команду document. open (), которая **полностью стирает весь текущий HTML-документ**. А потом уже пишет твою строчку в пустой белый лист.

— Почему это кошмар для разработки?

— Это абсолютно непредсказуемое и разрушительное поведение. Один случайный вызов document. write () в асинхронном коде может уничтожить всю твою страницу. Это как держать в коде гранату с выдернутой чекой. Профессиональные разработчики избегают его, как огня.

AI видел этот «бульдозер» в старых скриптах из начала 2000-х. Но он также видел тысячи тредов на Stack Overflow с заголовками «ПОМОГИТЕ, МОЯ СТРАНИЦА ПРОПАЛА!». И он сделал выводы. Он не даст тебе в руки инструмент, которым ты с вероятностью 99% себе всё сломаешь.

Подозреваемый №3: console. log («Hello, World!») — Профессиональный инструмент хирурга

И вот мы подходим к нашему главному герою. Если alert — это крикун, а document. write — бульдозер, то console — это приборная панель в твоём болиде Формулы-1.

— Что он делает технически?

— Он выводит информацию в специальную панель — консоль разработчика. Эту консоль видит только тот, кто её специально открыл (обычно по F12). Обычный пользователь даже не знает о её существовании.

— Почему это идеальный выбор?

— **Ненавязчивость:** Он никак не влияет на пользовательский интерфейс. Твоя страница работает, анимации крутятся, кнопки нажимаются. А где-то там, «под капотом», в консоли, ты видишь свои отладочные сообщения.

— **Профессионализм:** Это стандартный инструмент для дебага во всём мире JavaScript. Умение пользоваться консолью — это первое, что отличает новичка от человека, который понимает, что делает.

— **Мощь и гибкость:** console — это не только. log (). Это целый набор инструментов!

— console. warn («Что-то пошло не так…') — выведет жёлтое предупреждение.

— console.error («ВСЁ СЛОМАЛОСЬ!») — выведет красную ошибку со стектрейсом.

— console. table ([{user: «Vanya', age: 30}, {user: «Petya', age: 25}]) — нарисует красивую интерактивную таблицу из массива объектов.

— console.dir(document.body) — выведет все свойства DOM-элемента в виде удобного дерева.

AI выбрал console. log, потому что это язык профессионалов. Он не просто дал тебе решение. Он с первой же строчки начал погружать тебя в правильную культуру разработки. Он показал тебе инструмент, которым ты будешь пользоваться каждый день на протяжении всей своей карьеры.

Вывод: AI — твой первый фильтр от говнокода

Понимаешь, в чём главная фишка? AI не просто знает синтаксис. Он знает **контекст**. Он знает, что хорошо, а что плохо. Он обучен не на «всех подряд» примерах, а на лучших практиках, выстраданных миллионами разработчиков до тебя.

Он — твой персональный ментор, который оберегает тебя от дурных привычек, которые ты мог бы подцепить на старых форумах или в устаревших туториалах. Он не даст тебе выстрелить себе в ногу, предложив document. write. Он не научит тебя грубить пользователям через alert. Он сразу даёт тебе в руки чистый, профессиональный инструмент.

И наша задача, как вайб-кодеров, — не просто принять это как данность. Наша задача — остановиться и спросить «Почему?». И когда мы находим ответ, мы не просто получаем строчку кода. Мы получаем знание. Мы прокачиваем свой собственный мозг, а не просто используем чужой.

Совет от гаражного кодера:

— **Открой консоль и не закрывай.** Сделай привычкой при любой веб-разработке сразу открывать DevTools (F12) и переходить на вкладку Console. Пусть она всегда будет перед глазами. Наблюдай, что туда пишут скрипты сайтов, на которые ты заходишь.

— **Изучи все методы console.** Не ограничивайся. log (). Загугли «console API MDN». Попробуй вывести в консоль ошибку, предупреждение, таблицу. Это расширит твой арсенал для дебага.

— **Создай свой логгер.** Для практики напиши простую функцию-обёртку: function log (message) {console. log ([MY_APP_LOG]: $ {message});}. Это первый шаг к созданию более сложных систем логирования.

— **Никогда, слышишь, никогда не используй alert для дебага.** Если тебе нужно остановить выполнение кода в какой-то точке, используй слово debugger;. Это поставит выполнение на паузу и откроет вкладку отладки в DevTools, где ты сможешь посмотреть все переменные. Это в миллион раз мощнее и профессиональнее.

— Дебажим первую ошибку, сгенерированную AI. Учимся задавать правильные промпты

До сих пор AI был для нас идеальным ментором. Он показал, как надо. Но что будет, если мы дадим ему задачу чуть сложнее, чуть более приближенную к реальной жизни? Задачу, где есть неизвестные, где что-то может пойти не так.

Новый челлендж: получаем данные из сети

Давай попросим нашего AI-помощника написать нам функцию, которая ходит по сети за данными. Это хлеб с маслом для любого веб-разработчика.

Садимся за наш index. js, стираем старый код и пишем новый, слегка наивный промпт.

```
// Напиши асинхронную функцию getUsersNames, которая делает
// GET-запрос на 'https://jsonplaceholder.typicode.com/users',
// получает массив пользователей и возвращает новый массив только с их именами.
```

Этот промпт выглядит вполне адекватным. Он чёткий, он описывает, что нужно сделать. Мы ждём секунду… и Copilot/Cursor выдаёт нам настоящее произведение искусства.

```
// Напиши асинхронную функцию getUsersNames, которая делает
// GET-запрос на 'https://jsonplaceholder.typicode.com/users',
// получает массив пользователей и возвращает новый массив только с их именами.
async function getUsersNames () {
const response = await fetch (»https://jsonplaceholder.typicode.com/users');
const users = await response. json ();
const names = users.map (user => user.name);
return names;
}


// Добавим вызов, чтобы проверить
getUsersNames().then (names => console. log (names));
```

Смотри, какой красавец! async/await — стильно, модно, молодёжно. fetch для запроса. Метод. map () для элегантной трансформации массива. Никаких тебе уродливых циклов for. Код чистый, как слеза младенца.

Давай запустим его. Открываем терминал, пишем node index. js.

И… о, чудо! В консоли появляется массив имён:

[«Leanne Graham’, «Ervin Howell’, «Clementine Bauch’, …]

Работает! AI снова на высоте! Можно идти пить второй кофе?

Нет.

А теперь, братан, давай сделаем то, что отличает прогера от юзера — давай попробуем это сломать. В реальном мире API не всегда доступен. Сервер может упасть, в URL может быть опечатка, интернет может моргнуть.

Давай сымитируем это. Изменим URL на заведомо нерабочий.

```
//…
const response = await fetch (»https://jsonplaceholder.typicode.com/users-produsers'); // Опечатка в URL
//…
```

Запускаем снова. И…

КРАХ. ПАНИКА. КРАСНЫЕ БУКВЫ В КОНСОЛИ

```
(node:12345) UnhandledPromiseRejectionWarning: TypeError:
Cannot read properties of undefined (reading 'map')
at getUsersNames (/path/to/your/project/index. js:6:23)
...
```

Вот он. Твой первый баг, сгенерированный искусственным интеллектом. Добро пожаловать в реальный мир, Нео.

Разбор полётов: дебажим, как сеньоры

Первая реакция новичка — паника. «Что это за абракадабра?! Всё сломалось!».

Реакция вайб-кодера — спокойствие. Берём лупу и начинаем читать. Ошибки в консоли — это не проклятья, это карта сокровищ, которая ведёт к багу.

— **UnhandledPromiseRejectionWarning**: Первое, что мы видим. «Необработанное отклонение Промиса». Это говорит нам, что внутри нашей async функции что-то пошло не так, и мы это падение не поймали. Наш Промис (а async функции всегда возвращают Промис) был «отклонён» (rejected), а у нас нет блока. catch () или try...catch, чтобы с этим разобраться.

— **TypeError: Cannot read properties of undefined (reading 'map')**: А вот и суть проблемы. «Ошибка типа: не могу прочитать свойство 'map' у undefined». Это значит, что мы пытались сделать так: undefined.map (…).

— **/path/to/your/project/index. js:6:23**: Это точные координаты бага. Файл index. js, строка 6, символ 23.

Идём на 6-ю строку. А там у нас: const names = users.map (user => user.name);

Ага! Значит, в этот момент переменная users была равна undefined.

Теперь отматываем плёнку назад. Откуда берётся users? Со строки выше: const users = await response. json ();. Значит, именно здесь что-то пошло не так.

Почему response. json () мог вернуть undefined или что-то, что привело к ошибке? Давай добавим немного старого доброго дебага — console. log.

```
async function getUsersNames () {
const response = await fetch (»https://jsonplaceholder.typicode.com/
users-produsers');
console. log («Response Status:», response.status); // Что нам
ответил сервер?
console. log («Response OK?», response. ok); // Всё ли хорошо?


const users = await response. json (); // Попробуем распарсить ответ
console. log («Users:», users); // Что в итоге в переменной?


const names = users.map (user => user.name);
return names;
}
```

Запускаем снова. Теперь картина становится яснее.

В консоли мы увидим что-то вроде:

Response Status: 404

Response OK? false

А потом снова крэш. Потому что когда мы пытаемся вызвать. json () на ответе с ошибкой 404 (Not Found), который содержит не JSON, а HTML-страницу с ошибкой, происходит сбой парсинга.

Вот он, корень зла! Наш «гениальный» AI написал код для идеального мира. Для «happy path», где сервер всегда отвечает 200 OK и возвращает правильный JSON. Он не подумал о том, что реальный мир полон боли и страданий (и 404-х ошибок).

Фиксим баг: добавляем броню

Теперь, когда мы поняли проблему, исправить её — дело техники. Нам нужно проверять ответ сервера, прежде чем пытаться его использовать.

```
async function getUsersNames () {
try {// Оборачиваем всё в try...catch — это сеть для ловли ошибок
const response = await fetch (»https://jsonplaceholder.typicode.com/
users-produsers');
// ГЛАВНАЯ ПРОВЕРКА!
if (!response. ok) {
// Если статус не 2xx, выбрасываем ошибку сами
throw new Error (`HTTP error! status: ${response.status} `);
}


const users = await response. json ();


if (!users || users. length === 0) {
// Проверяем, а есть ли вообще пользователи
return []; // Возвращаем пустой массив, а не падаем
}


const names = users.map (user => user.name);
return names;
} catch (error) {
console.error («Ой, всё сломалось при получении пользователей:»,
error);
return []; // В случае любой ошибки возвращаем пустой массив,
чтобы остальная часть программы не упала
}
}
```

Вот теперь наш код одет в броню. Он готов к суровой реальности. Он проверяет ответ сервера и ловит любые ошибки в блоке try...catch.

Учимся задавать правильные промпты: от джуна к сеньору

Мы починили код. Но вайб-кодер не просто чинит, он учится, чтобы не повторять ошибок. Проблема была не только в AI. Проблема была в *нашем промпте*. Он был слишком наивным.

Промпт уровня «Джуниор»:

// Напиши функцию, которая получает имена пользователей с API.

Промпт уровня «Миддл»:

// Напиши асинхронную функцию getUsersNames, которая делает GET-запрос на «…', получает массив пользователей и возвращает новый массив только с их именами. (Это то, что мы написали).

А вот как выглядит промпт уровня «Сеньор»:

// Напиши асинхронную функцию getUsersNames, которая делает GET-запрос на «…'. Функция должна быть надёжной. Добавь обработку сетевых ошибок с помощью try...catch. Если ответ сервера не успешный (статус не 200—299), функция должна выбрасывать ошибку с текстом статуса. Если в ответе пришёл пустой массив или вообще не массив, функция должна возвращать пустой массив, не падая. В случае любой другой ошибки, она также должна возвращать пустой массив и логировать ошибку в консоль.

Чувствуешь разницу? Сеньорский промпт не просто говорит, *что* делать. Он предвидит проблемы и говорит, *как на них реагировать*. Он описывает не только «happy path», но и все «sad paths».

Если ты дашь AI такой промпт, он сгенерит тебе тот самый, надёжный код, который мы написали руками.

Вывод: ты — навигатор, а не пассажир

Эта первая ошибка — твой обряд инициации. Ты понял, что AI — не волшебник. Он — невероятно мощный, но очень буквальный исполнитель. Он не будет думать за тебя об edge-кейсах.

Твоя новая работа — быть параноиком. Быть тем самым душным тимлидом для своего AI-стажёра. Прежде чем писать промпт, задай себе вопросы:

— А что, если данные придут пустые?

— А что, если придёт не тот тип данных (не массив, а объект)?

— А что, если сеть упадёт?

— А что, если сервер вернёт ошибку?

Думать о пограничных случаях, об ошибках, об архитектуре — вот твоя настоящая работа. А рутину по написанию кода можешь смело делегировать своему кремниевому помощнику.

Совет от гаражного кодера:

— Сделай «чек-лист параноика». Прежде чем принять AI-код, который работает с внешними данными (API, файлы), прогони его по списку: «Обработан ли null/undefined? Есть ли проверка на пустой массив/объект? Обернуто ли в try...catch? Проверяется ли статус ответа?».

— Используй чат для дебага. Когда получаешь ошибку, не спеши гуглить. Выдели код вместе с ошибкой, скопируй в AI-чат (Copilot Chat/Cursor) и спроси: «Я получаю вот такую ошибку. Объясни, что она значит и как её исправить в этом коде?». Он часто даёт очень точные и полезные объяснения.

— Пиши промпты итеративно. Не пытайся сразу написать идеальный сеньорский промпт. Начни с простого, получи базовый код. А потом используй Ctrl+K (в Cursor) или чат, чтобы доработать его: «А теперь добавь сюда обработку ошибок», «А теперь сделай так, чтобы он возвращал пустой массив при сбое».

— Создай коллекцию «хороших промптов». Если ты часто выполняешь однотипные задачи (например, создаёшь React-компонент с подключением к Redux), сохрани себе где-нибудь в заметках идеальный, подробный промпт для этой задачи. Это твой личный арсенал заготовок.

Часть 2. Основы под присмотром: учимся, а не просто копипастим

Глава 4. Логика и данные: когда AI твой штурман, а не пилот

— Переменные, типы, условия. Просим AI написать функцию и проверяем её на косяки (например, == вместо ===)

Представь, что мы пилим фичу для какого-нибудь сайта с платным контентом. Нам нужна функция, которая будет решать, можно ли показать пользователю секретную статью или нет. Классическая задача, которую ты будешь решать сотни раз в своей карьере.

Давай сформулируем для нашего AI-помощника промпт. Сделаем его достаточно чётким, но оставим немного пространства для «творчества».

```
// Напиши функцию canAccessContent, которая принимает два аргумента:
// 1. age (возраст пользователя, число)
// 2. subscription (статус подписки, строка: 'free', 'pro', 'admin')
// Функция должна возвращать true, если:
// — у пользователя подписка 'admin' (им можно всё, независимо от возраста)
// — ИЛИ у пользователя подписка 'pro' И ему 18 или больше лет.
// Во всех остальных случаях функция должна возвращать false.
```

Отправляем этот приказ в нашу IDE и ждём. Copilot или Cursor, почесав свои кремниевые мозги, скорее всего, выдаст нам что-то очень похожее на это:

```
function canAccessContent (age, subscription) {
if (subscription == 'admin' || (subscription == 'pro' && age> = 18)) {
return true;
} else {
return false;
}
}
```

Первый взгляд: вроде всё чики-пуки

Давай посмотрим на этот код глазами джуна. Выглядит логично. Условие в if полностью повторяет наше ТЗ. || — это «ИЛИ», && — это «И». Кажется, всё работает.

Давай проведём быстрый тест-драйв, проверим основные сценарии («happy paths»):

```
console. log («Админ, 16 лет:», canAccessContent (16, 'admin')); //
Ожидаем: true
console. log («Pro, 25 лет:», canAccessContent (25, 'pro')); //
Ожидаем: true
```

```
console. log («Pro, 17 лет:», canAccessContent (17, 'pro')); // Ожидаем: false
console. log («Free, 30 лет:», canAccessContent (30, 'free')); // Ожидаем: false
```

Запускаем… и получаем:
Админ, 16 лет: true
Pro, 25 лет: true
Pro, 17 лет: false
Free, 30 лет: false

Идеально! Задача решена, таск можно закрывать. AI — гений, я — молодец.

А теперь, братан, стоп. Выдыхай. Это была ловушка. Ловушка, в которую попадаются 9 из 10 новичков. Мы проверили только идеальные случаи. Но реальный мир, как мы знаем, далёк от идеала. Данные могут приходить с фронтенда в виде строк, API может вернуть null, пользователь может вообще ничего не ввести.

Второй взгляд: достаём сеньорскую лупу

Давай посмотрим на одну маленькую, но дьявольски коварную деталь в сгенерированном коде.

if (subscription == 'admin'…

Видишь? ==. Два знака равенства.

В мире JavaScript это не просто оператор сравнения. Это «оператор нестрогого, или абстрактного, сравнения». И он — источник самых безумных и трудноуловимых багов.

Что делает ==?

Он пытается быть твоим другом. Слишком хорошим другом. Прежде чем сравнить две переменные, он пытается привести их к одному типу. Этот процесс называется **приведение типов (type coercion)**. Иногда это удобно, но чаще всего это приводит к хаосу.

Например:

5 == «5» вернёт true. JS видит число и строку и думает: «Хм, наверное, они имели в виду числа», и сравнивает 5 с 5.

false == 0 вернёт true.

null == undefined вернёт true.

«« == false вернёт true.

Это безумие. Это как если бы твой GPS на команду «поехали в Питер» иногда вёз тебя в Павловск, потому что «ну, названия же похожи».

Атакуем нашу функцию: ищем уязвимости

Давай представим, что с фронтенда нам пришёл возраст не как число, а как строка. Или что-то пошло не так, и в subscription прилетело что-то странное.

Давай создадим другую, более простую функцию, чтобы наглядно увидеть этот ад. Попросим AI:

// Напиши функцию, которая проверяет, равно ли значение false

AI может сгенерить:

```
function isItFalse (value) {
if (value == false) {
return true;
}
return false;
```

```
}
```

А теперь смотри фокус:

```
console. log (isItFalse (0)); // Выведет: true. Ноль — это не false!
console. log (isItFalse (»»)); // Выведет: true. Пустая строка — это
не false!
console. log (isItFalse ([])); // Выведет: true (в некоторых движках).
Пустой массив — это не false!
```

Вот она, мина, которую нам подложил наш услужливый AI. Он написал синтаксически верный, но логически хрупкий и опасный код.

Фикс: === — наш бронежилет

Вайб-кодеры знают правило, выжженное на сердце годами дебага: **всегда используй === (оператор строгого сравнения)**.

Что делает ===?

Он — суровый, но справедливый судья. Он не пытается никого мирить. Он сначала проверяет, совпадают ли типы переменных. Если типы разные (например, число и строка) — он сразу возвращает false, даже не смотря на значения. И только если типы одинаковые, он сравнивает значения.

5 === «5» вернёт false. Типы разные. Точка.

false === 0 вернёт false.

null === undefined вернёт false.

Это предсказуемость. Это надёжность. Это твой главный инструмент против безумия JavaScript.

Давай починим наш код руками.

```
function canAccessContent (age, subscription) {
// Заменяем все '==' на '===»
if (subscription === 'admin' || (subscription === 'pro' && age> =
18)) {
return true;
} else {
return false;
}
}
```

А теперь давай сделаем его ещё чище, как настоящий сеньор. Весь этот if-else для возврата true/false — избыточен. Ведь само условие уже возвращает boolean.

Вот. Коротко, чисто, надёжно. И это то, до чего мы дошли *своим* мозгом, проанализировав и улучшив то, что предложил AI. Мы не просто приняли код, мы его отрефакторили.

Учимся правильно командовать: сеньорский промпт 2.0

Как мы могли бы избежать этой ошибки с самого начала? Сформулировав более точный приказ.

Наш джуниорский промпт был хорош, но в нём не хватало одного ключевого требования.

Промпт уровня «Сеньор»:

// … (всё то же самое) … Используй операторы строгого сравнения (===) для всех проверок.

Если ты добавишь это уточнение, AI с гораздо большей вероятностью сгенерит тебе сразу правильный, надёжный код. Ты, как пилот, должен давать своему штурману не только пункт назначения, но и важные указания по маршруту («избегай гравийных дорог», «не превышай скорость»).

Вывод: AI — это ускоритель, а не мыслитель

Эта простая функция вскрыла самую суть нашей новой работы.

AI взял на себя рутину: написал синтаксически правильную конструкцию function, if-else, return. Он сэкономил нам 30 секунд печатания.

Но **логическую ответственность** он оставил на нас. Он не подумал о хрупкости ==. Он не подумал о чистоте и лаконичности кода. Он просто выдал самый статистически вероятный ответ на наш запрос.

Наша задача как пилотов — не просто нажимать Tab. Наша задача — быть главным ревьюером, главным QA-инженером для нашего AI-помощника. Мы должны смотреть на сгенерированный код и задавать вопросы:

— А он надёжный?

— А он безопасный?

— А он выдержит плохие данные?

— А можно ли его написать чище?

Переменные, типы и логика — это не то, что можно делегировать. Это фундамент. И за качество этого фундамента отвечаем только мы. AI может подвезти нам кирпичи, но кладём мы их сами.

Совет от гаражного кодера:

— **Сделай === своей религией.** Используй его всегда и везде. Единственный случай, когда == может быть оправдан — это проверка variable == null, которая отлавливает и null, и undefined разом. Но даже это многие считают дурным тоном. Проще быть последовательным.

— **Настрой ESLint.** Это твой автоматический страж порядка. Установи плагин ESLint для своей IDE и настрой в нём правило «eqeqeq»: [«error», «always»]. После этого IDE будет подсвечивать красным любую попытку использовать ==, заставляя тебя писать правильно.

— **Валидируй данные на входе.** Прежде чем работать с данными, пришедшими извне (от пользователя, из API), убедись, что они того типа, который ты ожидаешь. Если ждёшь число age, преобразуй его явно: const numericAge = parseInt (age, 10);. Никогда не доверяй внешним данным.

— **Думай об «обратном» условии.** Когда пишешь логику, всегда думай не только о том, когда она вернёт true, но и о том, когда она вернёт false. Это помогает находить слабые места. «А что, если subscription будет null? А если age будет отрицательным?». Это развивает защитное мышление.

— Рофлим над тем, как Copilot предлагает три разных способа написать цикл for, и все — неоптимальные

Давай поставим нашему AI простую, как три копейки, задачу. У нас есть массив чисел, и нам нужно посчитать их сумму. Это «Hello, World!» в мире циклов.

Формулируем промпт. Ничего заумного, просто и по делу.

// Напиши функцию sumArray, которая принимает массив чисел

// и возвращает их сумму.

Мы ожидаем увидеть что-то простое и элегантное. Но наш AI-штурман, видимо, решил блеснуть эрудицией. Он, как студент-отличник, хочет показать, что знает не один, а сразу несколько способов решения. И он начинает нам их предлагать.

Нажимаем Ctrl+Enter (или используем другую команду для просмотра альтернатив), и начинается парад… не самых лучших идей.

Вариант №1: «Классика жанра» от деда

Первое, что часто предлагает Copilot, — это самый древний, самый кондовый способ, который был с нами со времен динозавров и jQuery.

```
function sumArray (arr) {
let sum = 0;
for (let i = 0; i <arr. length; i++) {
sum += arr [i];
}
return sum;
}
```

Что мы видим?

Это классический цикл for с итератором i. Он работает? Да. Он понятен? Вполне. Он хороший? Ну… как сказать.

Анализ от гаражного кодера:

— **Многословность:** Мы создаём переменную i, задаём ей начальное значение, пишем условие i <arr. length и инкрементируем i++ на каждой итерации. Это куча лишнего кода, который описывает *как* итерировать, а не *что* мы хотим сделать.

— **Риск ошибки «off-by-one»:** Новичок легко может ошибиться и написать i <= arr. length, что приведёт к выходу за пределы массива и получению undefined, который при сложении с числом даст NaN. Привет, часы дебага.

— **Производительность (микро-занудство):** На каждой итерации происходит обращение к arr. length. В современных движках JS это оптимизировано, но в старых браузерах или на очень больших массивах это могло быть микро-проблемой. Раньше «тру-кодеры» выносили const len = arr. length; за пределы цикла. AI об этом, видимо, тоже читал.

Вердикт: Это как ехать на старой дедовской «Волге». Она довезёт тебя из пункта А в пункт Б. Но она жрёт много бензина (твоего внимания), громоздкая и не очень безопасная. AI предложил это, потому что таких циклов — миллиарды в коде, на котором он учился. Это самый статистически вероятный ответ.

Вариант №2: «Модный хипстер» for...in

Окей, мы просим AI показать что-то ещё. И он, решив быть более современным, предлагает нам for...in.

```
function sumArray (arr) {
let sum = 0;
for (const index in arr) {
sum += arr [index];
}
return sum;
}
```

Что мы видим?

Выглядит чище! Нет этого уродливого i=0; i <…; i++. Кажется, это шаг вперёд.

Анализ от гаражного кодера:

А вот и нет, братан. Это не шаг вперёд, это прыжок в болото с крокодилами. Это одна из самых частых и опасных ошибок, которую совершают новички в JavaScript.

Цикл for...in **НЕ ПРЕДНАЗНАЧЕН для итерации по массивам!**

Он предназначен для итерации по **перечисляемым свойствам объекта**. И вот какие ужасы нас тут поджидают:

— Он итерирует по ключам, а не по значениям. Переменная index в нашем случае будет не числом, а **строкой**! «0», «1», «2» и так далее. Когда мы делаем sum += arr [index], JS может вести себя непредсказуемо, пытаясь сложить число и строку (хотя в данном случае он, скорее всего, справится и приведёт тип).

— Он может перебирать не только элементы массива. Если кто-то до тебя решил расширить прототип Array (что само по себе — ужасная практика), for...in вытащит и эти свойства.

— Смотри фокус:

```
Array.prototype.myCoolMethod = function () {/* … */};
const myArray = [10, 20, 30];


// Попробуем просуммировать с помощью for...in
let sum = 0;
for (const i in myArray) {
console. log (`Индекс: $ {i}, Тип: $ {typeof i} `);
sum += myArray [i];
}
console. log («Сумма:», sum);
```

— Что мы увидим в консоли?
— Индекс: 0, Тип: string
— Индекс: 1, Тип: string
— Индекс: 2, Тип: string
— Индекс: myCoolMethod, Тип: string
— Сумма: 60function () {/* … */}
— Мы получили конкатенацию строки и функции! Наш результат — полный мусор.

— Порядок не гарантирован. Спецификация не гарантирует, что for...in будет обходить свойства в каком-то определённом порядке. Для массивов это обычно работает, но полагаться на это нельзя.

Вердикт: AI предложил нам инструмент, который выглядит как отвёртка, но на самом деле это молоток. Использовать for...in для массивов — это как забивать шурупы. Можно, но результат будет плачевным. Это явный признак того, что AI не всегда понимает *семантику* и *назначение* конструкций языка. Он просто видел, что «так тоже делают», и предложил нам.

Вариант №3: «Переусложнённый forEach»

Хорошо, мы снова недовольны. Просим ещё вариант. AI, вздохнув, решает показать нам, что он знает про функциональное программирование, и выдаёт forEach.

```
function sumArray (arr) {
let sum = 0;
arr.forEach (function (num) {
sum += num;
});
return sum;
}
```

Что мы видим?

О, это уже гораздо лучше! Это декларативный подход. Мы не говорим *как* итерировать, мы говорим «для каждого элемента сделай вот это». Код стал чище, нет ручного управления индексом.

Анализ от гаражного кодера:

Это хороший, рабочий вариант. Гораздо лучше первых двух. Но… он всё ещё не *оптимальный* для нашей конкретной задачи.

— **Наличие сайд-эффекта:** Мы создаём внешнюю переменную sum и мутируем (изменяем) её изнутри коллбэк-функции. Это называется сайд-эффект. В маленькой функции это не страшно. Но в большом приложении это может затруднять чтение и отладку кода.

— **Есть инструмент получше:** Для задачи «свернуть массив в одно значение» (а суммирование — это именно такая задача) в JavaScript есть специальный, более мощный и семантически верный инструмент.

Так как же надо было?! Вайб-кодерский подход

А теперь, братан, давай покажем нашему AI, как это делают настоящие пилоты. Для задачи «взять массив и получить из него одно-единственное значение» (будь то сумма, произведение, самый большой элемент или объект сложной структуры) существует метод. reduce ().

```
function sumArray (arr) {
// Аккумулятор (acc) — это наша 'sum'
// Текущий элемент (current) — это наш 'arr [i]»
// 0 — это начальное значение аккумулятора
return arr.reduce ((acc, current) => acc + current, 0);
}
```

Почему это — топ?

— **Декларативность и чистота:** Код превратился в одну строчку. Он читается как предложение: «Возьми массив arr, сверни его, начиная с нуля, складывая аккумулятор с текущим элементом».

— **Отсутствие сайд-эффектов:** Мы не создаём и не мутируем никаких внешних переменных. Функция получается более «чистой» и предсказуемой.

— **Семантическая верность:** Мы используем инструмент, который был создан именно для этой задачи. Это как использовать гаечный ключ нужного размера, а не пытаться крутить гайку пассатижами.

— **Гибкость:.re**duce () — это швейцарский нож для работы с массивами. Немного изменив коллбэк, ты можешь с его помощью делать невероятные вещи, которые циклами for писать было бы долго и муторно.

Почему AI не предложил. reduce () сразу?

Возможно, в его обучающей выборке циклов for и forEach для суммирования было больше, чем. reduce (). Новички часто боятся. reduce (), считая его сложным. Поэтому статистически он менее вероятен.

А может, он, как хитрый препод, вёл нас по пути от простого к сложному, от плохого к хорошему, чтобы мы сами поняли разницу. Но мы-то знаем, что он просто машина, и наша задача — быть умнее.

Вывод: не доверяй первому предложению

Эта история с циклами — идеальная иллюстрация нашего нового рабочего процесса.

— **AI — генератор идей.** Он может быстро накидать тебе несколько вариантов решения. Он как твой личный Stack Overflow, который не будет тебя хейтить за глупые вопросы.

— **Ты — фильтр и архитектор.** Твоя работа — проанализировать эти идеи. Отбросить откровенно плохие (for...in). Оценить компромиссы в «нормальных» (for, forEach). И выбрать самый оптимальный, самый чистый и самый правильный для данной задачи (reduce).

AI знает *что* можно использовать. Но только ты, пилот, знаешь *что* нужно использовать. Твой опыт, твоё знание контекста и лучших практик — вот что превращает сгенерированный код в качественный продукт.

Совет от гаражного кодера:

— **Полюби декларативные методы.** Выучи и начни использовать map, filter, reduce, find, some, every. Они делают код чище, короче и менее подверженным ошибкам, чем классические циклы for.

— **Используй for… of для простых итераций.** Если тебе не нужен. reduce, а просто нужно пройтись по значениям массива (без индексов), то самый современный и безопасный способ — это for… of: for (const num of arr) {…}. Он лишён всех недостатков for...in и чище, чем классический for.

— **Устрой AI экзамен.** Дай ему задачу и попроси решить её тремя разными способами. А потом сядь и для каждого способа напиши комментарий, почему он хорош или плох. Это отличное упражнение для закрепления материала.

— **Спроси AI «почему?».** После того, как он сгенерил код, спроси у него в чате: «Почему ты использовал здесь forEach, а не reduce?». Иногда его ответы могут быть на удивление толковыми и подсветить нюансы, о которых ты не думал.

— Практика: Рефакторим код, сгенерированный AI, до состояния «можно показывать сеньору»

Задача: фильтруем и форматируем пользователей

Представь, что мы пилим админку для интернет-магазина. Нам нужна функция, которая будет готовить данные для отображения списка «VIP-клиентов».

ТЗ (Техническое задание):

Написать функцию getVipClientList, которая принимает массив объектов-пользователей.

Каждый пользователь — это объект вида:

{id: 1, name: «John Doe’, email: 'john@doe.com’, balance: 1500, orders: 12, last_login: «2025-07-10T12:00:00Z», status: ’active’}

Функция должна:

— Отфильтровать только «VIP-клиентов». VIP — это те, у кого статус active И баланс больше 1000 И количество заказов больше 10.

— Отформатировать отфильтрованных пользователей в новый, более простой вид для отображения. Новый объект должен содержать только id, name и новое поле report_line, которое будет строкой вида: «John Doe (john@doe.com)».

— Вернуть массив этих отформатированных объектов.

Шаг 1. Пишем промпт и получаем «сырой» код

Давай сформулируем промпт для нашего AI. Достаточно детальный, но без сеньорских замашек. Так, как его написал бы уверенный миддл.

// Напиши функцию getVipClientList, которая принимает массив объектов-пользователей.

// 1. Отфильтруй пользователей: статус ’active’, баланс> 1000, заказов> 10.

```
// 2. Отформатируй результат: верни массив объектов с полями id, name
// и новым полем report_line в формате «Имя (email)».
```

Отправляем этот приказ в бездну нейронных сетей. Секунда ожидания… и Copilot/ Cursor выдаёт нам первый драфт. Скорее всего, он будет выглядеть примерно так:

```
// — — — ВЕРСИЯ 1: СГЕНЕРИРОВАНО AI — — —
function getVipClientList (users) {
const vipClients = [];
for (let i = 0; i <users. length; i++) {
const user = users [i];
if (user.status == 'active' && user.balance> 1000 && user. orders>
10) {
vipClients. push (
```

Конец ознакомительного фрагмента.