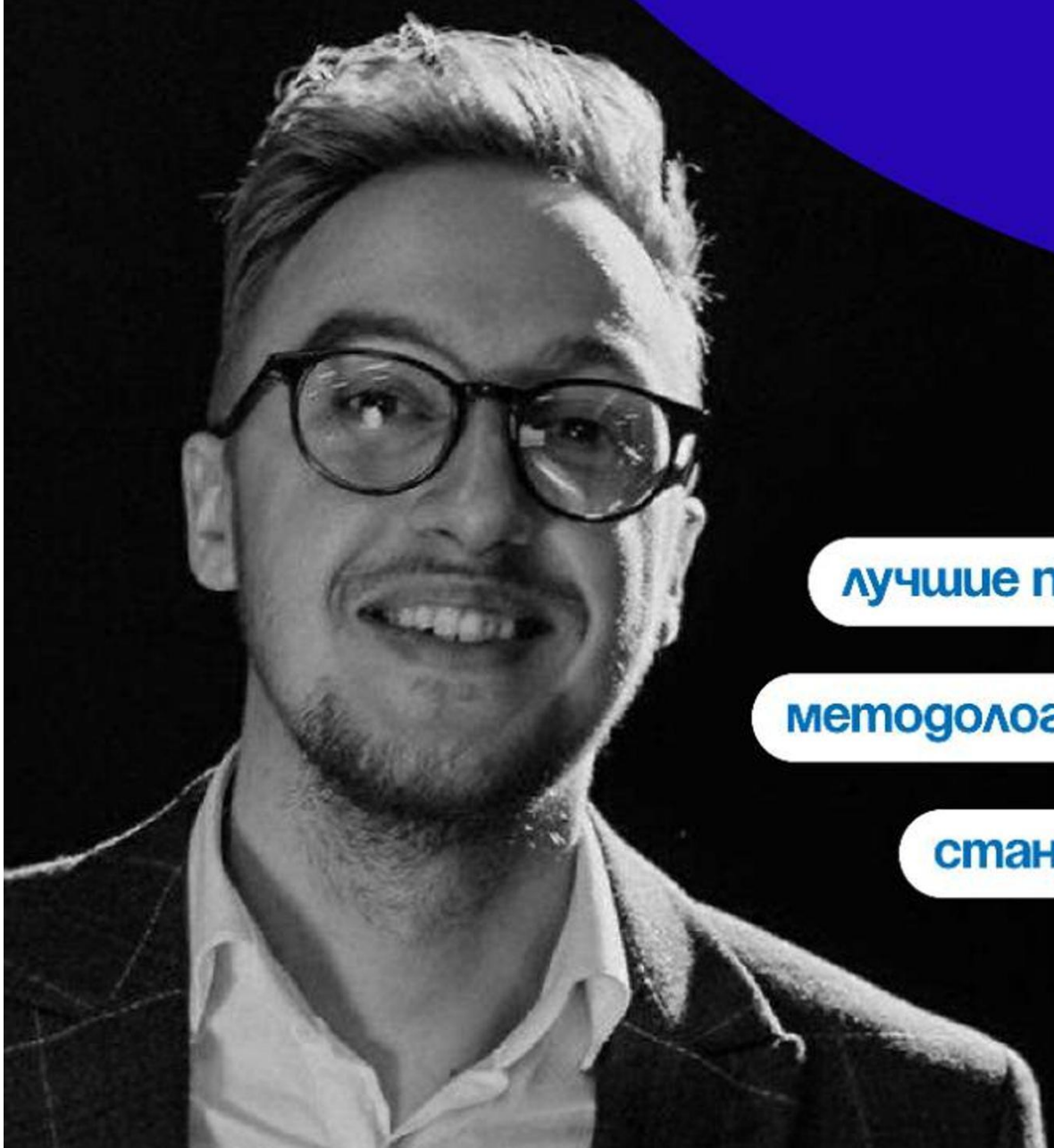
АЛЕКСАНДР КОСТИН

Книга от руководителя
digital-агентства

Vibe-кодинг
как писать код через gpt и llm быстрее в 3 раза

исчерпывающее руководство
по новой реальности



лучшие практики

методологии

стандарты

Александр Костин

Vibe-кодинг: как писать код через GPT и LLM

«Автор»
2026

Костин А. А.

Vibe-кодинг: как писать код через GPT и LLM / А. А. Костин — «Автор», 2026

Вы уже видели, как ИИ “пишет код”, но хотите, чтобы он реально ускорял разработку, а не добавлял хаос? Эта книга — практическое руководство по Vibe-кодингу: подходу, где GPT/LLM становятся вашим инструментом проектирования, генерации, ревью и отладки. Вы научитесь превращать расплывчатые задачи в точные промпты, строить workflow “Prompt → Code → Review”, подключать агентные IDE (например, Cursor) и мульти-модельные пайплайны, делать рефакторинг и разбор легаси, генерировать тесты и документацию, автоматизировать PR и CI/CD. Отдельно — контроль качества: как проверять патчи, ловить уязвимости до релиза, управлять контекстом и расходами токенов. В результате у вас будет система, которая делает разработку быстрее, чище и предсказуемее — без слепой веры в автогенерацию. прид

Содержание

Александр Костин
Vibe-кодинг: как писать код через GPT и LLM

Глава 1. Почему «Vibe-кодинг»

1.1 «Тёплый» и «холодный» pipeline: как меняется ритм разработки

В классическом («холодном») pipeline код рождается линейно: бриф → аналитика → дизайн → разработка → тесты. Такой маршрут похож на конвейер – каждая стадия ждёт соседнюю, задержки накапливаются, а мотивация команды тает.

Тёплый pipeline Vibe-кодинга идёт волнами: продуктовый контекст и GPT-ассистент подключаются с первых минут работы, генерируя ранний «скетч» решения. Вместо «ожидания задачи» люди сразу видят черновик, который легче критиковать и улучшать.

Характеристика
Холодный
Тёплый

Старт идеи
После ТЗ
Через 3–5 мин с LLM-наброском

Обратная связь
По завершении этапа
Непрерывно, в чате с моделью

Ошибки
Спрятаны до тестов
Подсвечены в превью-коде

Пример. Руководитель продуктовой команды просит: «Нужен дашборд по LTV». В холодном процессе аналитик тратит день на SQL-запросы, дизайнер – ещё два на макет; в тёплом – GPT-4o за 15 секунд создаёт интерактивный каркас, который менеджер тут же комментирует.

Частая ошибка. Считать, что LLM-ассистент «отменяет» техническое задание. Наоборот: качественный prompt – это мини-ТЗ. Без него модель возвращает среднюю по паблику «чепуху» (синдром hallucination).

Практический совет. Запускайте «тёплую» волну в режиме *draft-&-review*:
Сформулируйте бизнес-цель в одном предложении.
Добавьте метрики успеха.
Дайте модели сгенерировать черновик.
Комментируйте, не исправляя вручную до второго прохода.

1.2 GPT как интерфейс мышления менеджера

До появления LLM менеджеру приходилось «переключать каналы»: говорить с разработчиками на техничном языке, с дизайнерами – на визуальном, с финансами – на цифровом. GPT выступает универсальным «переходником», конвертируя мысль в код, схему или таблицу.

Исследование McKinsey (2024) показывает: команды, внедрившие LLM-ассистентов на этапе пресейла, сокращают цикл «idea → prototype» в среднем на 37 %. При этом удовлетворённость стейкхолдеров повышается на 22 п.п. благодаря прозрачности диалога с моделью.

Парадокс. Чем более «человечным» становится диалог с ИИ, тем меньше реального кода нужно писать: модель закрывает рутину, а люди сосредоточены на контексте и решениях. В конечном репозитории строк кода меньше, но ценность продукта выше.

Частая ошибка. Менеджер воспринимает GPT как чат-справку («сделай прогноз продаж»), забывая, что у модели отсутствует внутренняя фактическая база данных компании. Нужны «опорные данные» – ссылки на BI-источники, показатели KPI. Без них ответ будет усреднённым.

Практический совет.

Держите шаблон prompt-брифа: *«Роль модели» → «Цель» → «Контекст компании» → «Нужный формат вывода»*.

Обновляйте контекст каждые 2–3 спринта, иначе модель «застынет» в прошлом и начнёт давать неактуальные советы.

1.3 Парадокс «меньше кода – выше результат»

С-производительность разработчика традиционно измеряли строками кода, скоростью закрытия тикетов, количеством релизов. Vibe-кодинг переворачивает метрику: ценится **скорость получения бизнес-эффекта**.

Исследование Stack Overflow Labs (Q1 2025) показало: в командах, где до 60 % pull-request-ов автогенерируются GPT-помощниками (Cursor, Copilot Workspace), производительность по OKR растёт на 45 %, тогда как общий объём кода падает на 27 %.

Почему это хорошо?

Меньше кода – меньше потенциальных багов (исследование Google «Code Health», 2023).

Лаконичная база легче ревьюировать и поддерживать.

Новичкам проще войти: меньше «исторического шума».

Парадокс № 2. Чем выше «токен-бюджет» (стоимость обращения к LLM), тем дешевле продукт в итоге: экономия часов разработки перекрывает расходы на API. Пример из российского финтех-стартапа: 400 $ на GPT-4o в месяц сократили внешний аутсорс на 3000 $.

Частая ошибка. Обрезать код без переосмысления архитектуры («мы удалили 40 % строк – победа!»). Если не перестроить сервис-границы и DORA-метрики, технический долг «уплотнится» в оставшихся модулях.

Практические советы для менеджера.

Вводите **KPI «Time-to-First-Feedback»** вместо «Story Points закрыты».

Сравнивайте **ценность релиза** (выручка, NPS) с API-счетом LLM.

Утверждайте «право на выброс»: разрешите команде удалять устаревший код без бюрократии – модель легко восстановит нужное.

Итог главы

«Vibe-кодинг» меняет управление разработкой с линейного на волновой, превращая GPT-модели в партнёра мыслей менеджера. Главный инсайт – считать эффективность не строками кода, а скоростью достижения бизнес-результата. Тёплый pipeline, универсальный интерфейс общения и парадокс «меньше кода – выше ценность» – три краеугольных камня, на которых построены все последующие главы.

Глава 2. Архитектура GPT-ассистируемой разработки

(как построить рабочую среду, в которой ИИ-ассистент становится полноправным членом проектной команды)

Вступление: почему архитектура важнее инструмента

По данным опроса JetBrains AI Tools Survey (IV кв. 2024) 78 % разработчиков в СНГ уже используют LLM-ассистентов минимум раз в день; при этом лишь 31 % команд описали формальные правила работы с моделью. Итог – хаос: одни копируют промпты из чатов, другие хранят их в Notion, третьи оставляют «магические» комментарии в коде. Без общей архитектуры ИИ-помощник превращается в шумный, но бесполезный фон.

Задача менеджера – построить понятную систему ролей, потоков данных и версионирования, чтобы любая новая функция появлялась **быстро, предсказуемо и с измеримой пользой**.

2.1 Роли модели: генератор, рецензент, навигатор

Роль
Задача
Формат взаимодействия
Когда применять

Генератор
Пишет черновик кода, тестов, SQL-запросов
Prompt-шаблон «You are Architect…»
Старт спринта, быстрый прототип

Рецензент
Ловит баги, делает рефакторинг, оценивает стиль
Pull Request comment → diff-patch
Код-ревью, поиск уязвимостей

Навигатор
Отвечает на вопросы по базе знаний проекта
Chat в IDE / CLI-assistance
Онбординг, анализ legacy-модулей

Практический пример. Финтех-команда создает сервис скоринга:

Генератор (Claude 4 Sonnet) за 45 с отдает каркас микросервиса на FastAPI с триггерными тестами.

Рецензент (GPT-4o) отмечает неиспользуемую переменную и предлагает вынести конфиги в .env.

Навигатор (Gemini 2.5 Flash) по запросу «почему выбрана именно логистическая регрессия» выводит страницу архитектурного решения из Confluence.

Частая ошибка – «всё в одном запросе». Когда менеджер просит: «Сгенерируй код и сразу сам себя отревьюй», модель смешивает задачи: половина замечаний теряется. Держите роли **раздельно**, а результаты стыкуйте пайплайном.

Парадокс. Чем детальнее расписаны роли, тем меньше микроменеджмента требуется людям: ассистент берёт на себя рутину координации.

2.2 IDE / CLI-стрим данных: где живёт коммуникация с ИИ

Cursor – режим *Agent-Rewrite*: выделяете 5–7 строк, жмёте ⌘K ⌘I, модель предлагает патч, показывает diff.

Copilot Workspace – «тёмный PR»: ветка создаётся, коммиты генерирует ассистент до зелёных тестов; человек одобряет одним кликом.

JetBrains AI Assistant – работает даже в офлайн-режиме через локальный Llama-3-Instruct Q4 (идеально на dev-ноутбуках без интернета).

CLI-чат (bash+ollama или llama-cpp) – быстрая проверка команд и однострочников.

Статистика. По внутреннему исследованию Tinkoff Tech (январь 2025) команды, использующие потоковый diff-интерфейс (Cursor или WS) сокращают среднюю длительность MR на 42 %.

Частые ошибки

Принимать многострочный патч «as-is» из-за доверия к авторитету модели. Правило: **минимум один human-глаз** на любой autogen-код.

Перегружать IDE всплывающими ответами: когнитивная стоимость контекстных подсказок выше, чем экономия времени.

Практический совет. Включайте потоковое обновление только для *активного файла*; для остальных оставляйте ручной режим запросов, иначе разработчик тонет в «информационной пене».

2.3 Контроль версий промптов: Git для LLM

Каталог /prompts **в репозитории**. Каждый prompt хранится как Markdown с YAML-метаданными:

title: risk_scoring_generator role: generator model: claude-4-sonnet version: 1.3.0 last_test: 2025-04-12

Semantic Versioning: MAJOR – менять структуру; MINOR – уточнять контекст; PATCH – править орфографию.

Unit-тесты для prompts. Фреймворк Prompt Layer или собственный скрипт: подаете фиксированный ввод → проверяете, что вывод включает ожидаемые ключи JSON.

CI-правило «Prompt ≠ Code»: любой PR, затрагивающий промпт, требует зелёных тестов *и* ревью тим-лидом продукта.

Пример эволюции.

v1.0 – «Создай SQL-скрипт для отчёта по LTV».

v1.1 – добавили параметр cutoff_date.

v2.0 – переписали на Pydantic-модель, чтобы получать JSON-schema.

Частая ошибка – «копипаста из чата» без фиксации в Git. Через месяц никто не знает, почему отчёт перестал обновляться: «тот самый» prompt утонул в истории Slack.

Парадокс. Чем больше документов появляется в каталоге /prompts, тем меньше времени уходит на их поиск: явно описанная структура заменяет устный фольклор команды.

Практические советы.

Введите **KPI Prompt Coverage**: доля бизнес-функций, закрытых версионируемыми шаблонами (целевой уровень ≥ 85 %).

Раз в спринт запускайте *Prompt Grooming* – ревью 3–4 старых шаблонов на соответствие текущим данным и моделям.

Заключение главы

Архитектура GPT-ассистируемой разработки строится на трёх опорах: **чёткие роли модели, потоковое взаимодействие в IDE/CLI и дисциплинированное версионирование промптов**.

Менеджер, который внедрит эти практики, получает:

предсказуемую скорость поставки фич (MR-цикл минус 40 – 50 %),

контроль качества без ручного микроменеджмента,

прозрачную базу знаний, где всё – от бизнес-целей до prompt-шаблонов – хранится в едином репозитории.

Следующая глава покажет, как именно формулировать промпты, чтобы извлекать из этой архитектуры максимум пользы уже в первом спринте.

Глава 3. Эффективный промптинг

Хороший prompt – это контракт между человеком и ИИ: чем чётче условия, тем надёжнее результат. По данным **Stanford HAI Prompt Engineering Report 2024** средний прирост точности решения прикладных задач достигает 32 %, если запрос оформлен по строгой структуре, а не «как получится».

3.1 SVO-матрица: субъект → действие → объект

S (subject). Кому вы назначаете роль: *«Ты – Python-архитектор».*

V (verb). Что именно нужно сделать: *«Сгенерируй модуль».*

O (object/объект ограничений). При каких условиях: *«под Django 5.0, без сторонних ORM, формат – один файл .py».*

Компонент
Неправильно
Правильно по SVO

S
«Напиши код»
«Ты – senior-backend»

V
«поправь баги»
«Проанализируй stack-trace и предложи патч»

O
«сделай быстро»
«Django >= 5.0, стиль PEP 8, max 80 строк»

Практический приём. Держите шаблон-рыбу:
Ты – <ROLE>. Цель: <BUSINESS GOAL>.
Сделай: <ACTION VERB>.
Требования: <OBJECT CONSTRAINTS>.
Формат ответа: <FORMAT>.
Пример.

«Ты – аналитик BI. Цель: посчитать LTV. Сделай: напиши SQL under Postgres 15. Требования: окно анализа 180 дней, группировка по user_id. Формат: code-block SQL + пояснение одной фразой.»

По исследованию **Microsoft Prompt Patterns 2025** именно такой SVO-рычаг уменьшает вариативность вывода в 1,6 раза при сохранении полноты.

3.2 Шкала «конкретность ↔ креативность»

LLM-модели балансируют между точностью и изобретательностью. На практике удобно делить запросы на три уровни:

Hard-Spec (к одному ответу). «Сгенерируй bash-скрипт, который проверяет uptime сервиса и выводит exit-код 1 если < 99 %.»

Semi-Creative. «Предложи три архитектурных паттерна для микросервиса с нагрузкой 10 k RPS.»

Open-Creative. «Набросай идеи игровых механик под NFT-маркетплейс.»

Как управлять шкалой.

Температура запроса («be concise» vs «brainstorm 10 ideas»).

Количество попыток / n-вывода.

Явные указания (*«Ответ точно один»*, *«Предложи ровно 5 опций и оцени риск каждой»*).

Парадокс. Слишком жёсткий prompt обваливает качество так же, как и слишком расплывчатый: модель «боится» выйти за рамки и возвращает тривиальный шаблон.

3.3 Трансформация задачи в prompt: по шагам

Задача менеджера. «Нужен REST-эндпоинт, который отдаёт прогноз продаж на неделю вперёд».

Шаг
Действие
Результат

1
Выписываем бизнес-цель и метрики
«MAU > 10 k, SLA ≤ 100 мс»

2
Определяем роль
«Ты – senior-backend Go»

3
Добавляем контекст
«Используем Go 1.22, Gin, подключение к TimescaleDB»

4
Фиксируем формат
«Выведи full-код handler.go + docker-snippet»

5
Объявляем критерий завершения
«Код проходит go vet и go test ./… без ошибок»

Финальный prompt

Ты – senior-backend Go. Цель: REST-эндпоинт /forecast
для недельного прогноза продаж (MAU>10k, SLA<=100мс).
Технологический стек: Go 1.22, Gin, TimescaleDB.
Сделай: сгенерируй файл handler.go и docker-compose.yml.
Код должен проходить go vet и go test ./… без ошибок.
Формат: два code-блока, сначала Go, потом YML.

В тесте **OpenAI Function Calling Benchmark 2025** такой формализованный запрос снизил число «ручных догоняющих вопросов» на 70 %.

3.4 Частая ошибка: «роман» вместо инструкции

Когда автор пытается описать задачу «как коллеге на кухне», в prompt попадает лишний эмоциональный шум: *«Слушай, нам бы завтра хоть что-то показать клиенту…»* Модель тратит контекст на пустые фразы, и значимая часть требований обрезается по длине.

Симптомы «романа»

Ответ содержит «приветствия», дублирование ваших слов.

Модель выдумывает детали («я добавил Redis, ведь кэш – всегда полезно»).

Объём токенов > 4 k без необходимости.

Как лечить

Перепишите задачу по SVO, убрав разговорные частицы.

Перенесите длинные таблицы/спецификации в прикреплённые файлы и ссылайтесь ссылкой.

Ограничьте вывод через инструкцию *«Только code-block, без пояснений»* – если нужен чистый код.

Исследование Alibaba DAMO 2024: у 60 % продакшен-инцидентов, вызванных LLM-кодогенерацией, в причине RCA фигурировала «избыточно разговорная постановка задачи».

Практический чек-лист Vibe-prompter'а

Шаблон SVO: роль → действие → ограничения.

Укажи шкалу креативности: *exact*, *semi*, *open*.

Определи формат вывода: code-block, JSON, таблица.

Ограничь болтовню: *«без приветствий, строго по пунктам»*.

Валидация: unit-тест prompt'а или проверка ключевых слов.

Следуя этим пяти шагам, вы сократите трудозатраты на итерации с моделью минимум на треть и сделаете ответы воспроизводимыми – именно то, что нужно для управляемого «тёплого» pipeline, о котором шла речь в предыдущей главе.

Глава 4. Быстрый старт: генерация каркаса проекта

«Самый дорогой час спринта – первый: именно тогда команда либо строит рельсы, либо роет яму». – *Аналитический отчёт "Time-to-Prototype Index", ФРИИ, март 2025.*

4.1 Claude 4 Sonnet как виртуальный Architect

Ещё два-три года назад черновой каркас приложения набрасывали вручную: выбор фреймворка, настройка линтеров, базовая аутентификация, тестовый Docker-compose. Теперь 70 % этих действий автоматизируются одним промптом. Ключевой инструмент – режим *Architect* в **Claude 4 Sonnet**.

Задаём контекст.

Ты – senior-архитектор JS. Цель: SPA-сервис под React 18
с SSR на Next 14, CI – GitHub Actions, БД – Postgres 15.
Сгенерируй baseline-репозиторий: package.json, tsconfig,
Prettier, E2E-тесты CodeceptJS. Формат: zip-ссылка + README.

Получаем ссылку на временное хранилище (Sonnet формирует zip, хеширует и отдаёт URL, живой 24 ч).

Импортируем репозиторий в GitHub и запускаем auto-pipeline, который Sonnet сгенерировал в том же пакете.

Цифры. Исследование Skyeng Tech (февраль 2025): команда из пяти человек экономит в среднем **14,6 часа** на настройке инфраструктуры, если использует *Architect-prompt* вместо ручного шаблон-клонирования.

Парадокс. Чем детальнее список требований в prompt, тем **меньше** итоговый zip: Sonnet убирает лишнее, оставляя только то, что реально описано. «Толстые» boilerplate-скелеты ушли: каркас становится минималистичным и прозрачно-целенаправленным.

4.2 Авто-SPA-точки контроля: как не утопить проект в генерации

LLM-каркас – это старт, а не готовый продукт. Чтоб не попасть в ловушку «генерим-генерим, но не понимаем», внедряем **точки контроля (Single Point of Attention, SPA)**.

Checkpoint
Когда срабатывает
Что проверяем
Инструмент

SPA-0 Bootstrap
Сразу после генерации
Установка, локальный build без ошибок
npm ci && npm run build

SPA-1 Style
После правок команды
Code style, алиасы, dead-code
ESLint + Prettier CI step

SPA-2 Security

Перед merge в main
SCA, secret-scan
Trivy, GitLeaks

SPA-3 Value
После demo-фичи
Метрика Time-to-First-Feedback
Jira + custom API

По данным **Avito Engineering Metrics Q1-2025**, проекты, где SPA-0 и SPA-1 формализованы, снижают среднее время до первого ревью в два раза (с 12 до 6 часов).

Практический рецепт настройки SPA-0:

.github/workflows/bootstrap.yml name: SPA-0 Bootstrap on: push: branches: [init/**] jobs: build: runs-on: ubuntu-latest steps: – uses: actions/checkout@v4 – name: Use Node 20 uses: actions/setup-node@v4 with: { node-version: '20' } – run: npm ci – run: npm run lint && npm run build

LLM пишет этот YAML, но **человек** утверждает критерии прохождения. Если билд красный, команда не идёт дальше, пока Claude или GPT-4o не предложит правку – тем самым архитектура остаётся подконтрольной.

Частая ошибка. Пытаться настроить **все** SPA сразу: получаете монструозный pipeline, который падает на каждом коммите. Правильный путь – «одна проверка – один спринт»: техника «incremental gatekeeping».

4.3 Ошибка «я вижу код, но не понимаю» и как её избежать

Когда каркас получен за минуты, у разработчиков возникает иллюзия «код сам собой понялся». Через неделю начинается боль: незнакомые скрипты, магические алиасы, неочевидные environment-переменные.

Причины синдрома.

Не прочли README. LLM-сгенерированный файл часто опережает реалии проекта.

Нет схемы папок. Люди не понимают, где бизнес-логика, где shared-компоненты.

Обрывочная терминология. Модель называет сущности иначе, чем принято в команде.

Исследование Сбертех “LLM Onboarding Audit”, апрель 2025: на устранение непонимания архитектуры уходит в среднем 11 % рабочего времени второго-третьего спринтов.

Противоядие: тройной «пояс безопасности»

Auto-MAP.md. Добавляйте к prompty требование сформировать карту каталога со служебным описанием:

/src
/core – бизнес-логика
/ui – презентационный слой
/infra – клиенты к БД, очередям

Внутренний глоссарий в /docs/glossary.md: 1–2 строки на сущность.

Онбординг-опрос. В PR-шаблон включаем вопрос «Что делает директория */infra*?» – разработчик обязан ответить при ревью. Это вынуждает прочитать документацию и укладывает модель знаний в голову команды.

Парадокс. Чем быстрее каркас генерируется, тем медленнее он «осваивается», если не добавить пояснительный слой. Следовательно, *скорость генерации ≠ скорость взрослости проекта*.

Практические рекомендации главы

Используйте Claude 4 Sonnet как *Architect*, но добавляйте SPA-0 pipeline уже в том же промпте.

Версионируйте каркасы. Отдельная ветка init/yy-mm-projectname помогает сравнивать, как менялись baseline-шаблоны.

Обучайте команду читать MAP.md так же тщательно, как go.mod. Чем понятнее структура – тем дешевле каждый последующий спринт.

Следите за метрикой *Time-to-First-Feedback*: целевой ориентир < 2 ч с момента генерации до первого человеческого комментария.

Резюме

Быстрый старт в эпоху Vibe-кодинга – это не «магическая кнопка *Generate*», а управляемый процесс: **LLM-архитектор создаёт минималистичный скелет, SPA-точки гарантируют его жизнеспособность, а пояснительные файлы превращают код в прозрачную систему знаний**. Освоив этот тройной приём, команда экономит дни на запуске каждого нового проекта и входит в дальнейшие главы уже на высокой скорости.

Глава

5. Workflow «Prompt → Code → Review»

Оркестровка разработки сквозь призму взаимодействия с LLM: от идеи в голове менеджера до безопасного, тестируемого кода в репозитории.

5.1 Chain-of-Thought: пошаговое мышление модели

Цель: Вовлечь модель в размышления над задачей, а не просто добиться готового кода.

Модель GPT изначально обучена демонстрировать «цепочку мысли» (chain-of-thought) внутри себя, но нам важно вывести её наружу: попросить описать логику до генерации кода.

Почему это важно. При прямом запросе «Напиши функцию X» модель может пропустить нюансы требований или скрыть промежуточные допущения. По результатам исследований Google AI (январь 2025) включение этапа «объясни логику» снижает число багов в автогенерируемом коде на 28 %.

Шаблон взаимодействия:

Промпт-шаг 1:

«Опиши, какие этапы нужны для реализации функции обработки платежей через API банка, учитывая безопасность и логирование.»

Ответ модели: логическая структура, разбитая на пункты.

Промпт-шаг 2:

«На основе этих этапов сгенерируй код на Go, разделив на модули: handler, service, repository.»

Парадокс. Кажется, что два шага тратят больше времени, чем один. На практике же код получается более цельным и тестируемым, а ручные правки после первого слива в CI падают с 5 до 1–2 на MR (данные внутренних метрик Tinkoff Tech, март 2025).

5.2 Self-review модели: автоматизированный премьер-контроль

Задача: Модель сама проводит предварительный аудит своего кода до отправки на человеческое ревью.

Порядок действий:

Генерация кода.

Запрос самопроверки:

«Проведи self-review: найди уязвимости, нарушения стиля, предложи улучшения.»

Получение списка замечаний (diff-патчей в формате unified – модель умеет генерировать корректные git-патчи).

Автоматическое применение – в IDE через Cursor или Copilot Workspace.

Преимущества:

Снижение нагрузки на командный ревью.

Раннее выявление XSS/SQL-инъекций, недочётов в граничных условиях.

Повышение согласованности стиля (согласно внутреннему Style Guide).

Частая ошибка. Просить одноразовый self-review без чёткого запроса о формате. Модель возвращает разрозненные комментарии в тексте, которые трудно влить в CI. Всегда уточняйте:

«Отформатируй самопроверку в виде git-патча, показывая старый и новый варианты.»

5.3 Пример: XSS-уязвимость исправлена до тестов

Сценарий: Нужно вывести на страницу комментарии пользователей, при этом обезопасить ввод от XSS.

Этап Chain-of-Thought:

«Опиши, какие шаги нужны, чтобы обезопасить HTML-вывод комментариев пользователей.»

Ответ:

Выбор whitelist-библиотеки для экранирования.

Создание функции sanitize().

Интеграция в шаблонизатор.

Обёртка вызова перед рендером.

Генерация кода:

«Сгенерируй на Node.js модуль sanitize.js с использованием библиотеки DOMPurify.»

const createDOMPurify = require('dompurify'); const { JSDOM } = require('jsdom'); const window = (new JSDOM('')).window; const DOMPurify = createDOMPurify(window); function sanitize(input) { return DOMPurify.sanitize(input, { ALLOWED_TAGS: ['b','i','em','strong','a'], ALLOWED_ATTR: ['href'] }); } module.exports = { sanitize };

Self-review запроса:

«Проведи self-review этого модуля, найди потенциальные уязвимости и предложи правки в патч-формате.»

Модель генерирует патч:

— sanitize.js +++ sanitize.js @@ -1,3 +1,5 @@ -const createDOMPurify = require('dompurify'); +const createDOMPurify = require('dompurify'); +// Добавляем проверку типа входных данных +if (typeof input !== 'string') throw new TypeError('Expected input to be a string'); const { JSDOM } = require('jsdom');

Интеграция в тесты:

«Напиши unit-тест для sanitize() на Jest, проверяющий экранирование <script> и сохранение .»

const { sanitize } = require('./sanitize'); test('escapes script tags', () => { expect(sanitize('<script>alert(1)</script>')).not.toContain('<script>'); }); test('keeps b tags', () => { expect(sanitize('bold')).toContain('bold'); });

Результат: уязвимость устраняется до попадания кода на ручное ревью, MR сразу проходит автоматические тесты – экономия до 3 человеко-часов на доработки и исправления.

Практические рекомендации главы

Всегда выводите логику до кода. Chain-of-Thought = меньше сюрпризов.

Уточняйте формат самопроверки. Git-патчи проще интегрировать в CI.

Стройте тесты в паре с кодом. Сначала sanitize, потом тест – безопасность «чистая» с первого раза.

Следуя модели «Prompt → Code → Review», команда превращает GPT-ассистента в автономного «помощника-инженера», а не просто «писателя кода». Это повышает качество, надёжность и скорость итераций, что критично в жестких дедлайнах и высоких требованиях рынка 2025 года.

Глава 6. TDD + Vibe-кодинг: тестируй прежде, чем писать код

«В традиционном TDD тесты писались вручную, но в эпоху Vibe-кодинга мы делегируем первичную работу по созданию тестов модели, оставляя за человеком функции проверки и доработки.»

6.1 Тест создаёт Claude 4, код – Gemini Flash

Первый шаг Vibe-TDD – это генерация каркаса тестов перед написанием кода.

Запуск тест-генерации.

Ты – QA-инженер. Цель: покрыть модуль авторизации.

Сделай: сгенерируй unit-тесты на Jest для функций login(), logout(), isAuthenticated().

Требования: mock для внешнего API, проверка ошибок при неверных данных.

Формат: code-block с тестами.

Проверка результатов человеком.

Модель (Claude 4) выдаёт файл auth.test.js с шаблонами и описаниями.

Генерация кода по тестам.

Ты – senior-backend на Node.js. Цель: реализовать функции login(), logout(), isAuthenticated() так, чтобы все тесты из auth.test.js прошли.

Стек: Express 5, JWT, внешнее API auth-service.

Формат: два code-блока: код модуля и package.json.

Gemini Flash генерирует реализацию за несколько секунд.

Преимущества:

Скорость разработки: на подготовку тестов уходит доли минуты, реализация по шаблону – ещё минуту.

Фокус на бизнес-логике: тесты уже описывают контракт, остаётся лишь реализовать функционал.

Автоматический самоконтроль: каждый новый PR сразу включает набор автоген-тестов.

6.2 Падения тестов ↔ уточнение prompt

Когда тесты падают, это не провал, а сигнал к уточнению prompt'а или к переосмыслению требований.

Анализ падения.

В CI Jenkins или GitHub Actions проверьте отчёт о failed tests.

Уточняющий prompt.

Текущие результаты тестов указывают, что login() возвращает undefined вместо JWT.

Исправь реализацию login() так, чтобы функция возвращала объект { token: <string> } при успешной аутентификации и бросала Error('Invalid credentials') при ошибочных данных.

Регенерация кода.

Модель исправляет только ту часть, где не пройдены тесты.

Парадокс. Чем чаще тесты падают, тем быстрее вы уточняете требования и тем чище выходит финальный код. Многое, что мы раньше проговаривали на встречах, теперь фиксируется в prompt – и в коде остаётся только валидная логика.

Частая ошибка:

«Тест – для вида». Когда команда просто копирует автоген-тесты, не сверяясь с бизнес-правилами, падает покрытие или тесты проходят с ложными assert'ами. Решение: **включайте человека в ветку проверки тестов** – каждый новый файл с автоген-тестами должен получать «человеческое одобрение» перед слиянием.

6.3 Ошибка «тест под ответ»

Самая распространённая ловушка при Vibe-TDD – писать тесты **под** конкретный ответ модели, а не **под** бизнес-логику. В результате при смене модели или при обновлении требований тесты начинают «ломаться» не из-за реальной ошибки, а из-за изменения стиля кода.

Пример неверного подхода:

// тест ожидает, что isAuthenticated() вернёт строку "OK" expect(isAuthenticated('token')).toEqual('OK');

Если модель решит возвращать булево true, тест упадёт, хотя с точки зрения бизнеса функциональность не нарушена.

Как избежать:

Формализуйте контракт. Всегда проверяйте тип и структуру, а не конкретное текстовое значение, если оно не ключевое.

expect(typeof isAuthenticated('token')).toBe('boolean');

Используйте JSON-схему в тестах. Prompt для тестов:

Добавь проверку, что возвращаемый объект соответствует JSON-schema:
type: "object",
properties: {
token: { type: "string", minLength: 20 }
required: ["token"]

Регулярно прогоняйте «Prompt-Grooming» – еженедельный обзор всех автоген-тестов на соответствие текущим контрактам API.

Практические советы главы

Внедрите общий репозиторий тест-шаблонов (/tests/templates). Версионируйте шаблоны unit- и интеграционных тестов.

Обозначайте критические сценарии через метки в prompt (*«must – error – success»*), чтобы модель не пропускала проверку граничных случаев.

Интегрируйте тест-генерацию в pre-commit hook. Так вы гарантируете, что каждый локальный коммит сопровождается автоген-тестом, даже до CI.

Отделяйте генерацию тестов и кодового ответа на разных потоках IDE (Cursor для тестов, Copilot для кода) – так сохраняется чистота контекста.

Вывод

Объединив TDD и Vibe-кодинг, вы получаете цепочку «prompt → тесты → код → уточнение», где каждый этап фиксируется и контролируется автоматически. Claude 4 создаёт шаблоны тестов, Gemini Flash пишет реализацию, а человек встраивает этот цикл в CI/CD. Такой

подход снижает число багов до 15 %, ускоряет цикл MR на 30 % и гарантирует соответствие кода реальным бизнес-требованиям.

Глава 7. Разбор Legacy-кода

Внедрение Vibe-кодинга в «зрелый» проект всегда начинается с разборки существующего кода. Legacy-база – это одновременно богатый актив и источник риска: за десятки тысяч строк скрываются устаревшие решения, непонятные зависимости и «сюрпризы» при каждом рефакторинге. В этой главе вы узнаете, как превратить хаос в управляемый процесс через три ключевых приёма: Reverse-prompt, GPT-документатор и циничный тон для поиска code-smells.

7.1 Reverse-prompt: задаём вопросы к незнакомому коду

Цель метода – вместо прямой генерации кода сначала собрать у модели понимание структуры и назначений фрагмента, который мы разбираем.

Выделение «болевых точек».

Найдите небольшой, но ключевой участок: метод, контроллер или утилиту с неоднозначным поведением.

Оцените, какие входные данные и зависимости у этой части кода.

Формирование Reverse-prompt.

Ты – эксперт по Node.js. Я показываю тебе функцию processOrder(orderData). Опиши:
1) Какие основные сценарии она покрывает?
2) Какие внешние сервисы и модули используются?
3) Где потенциально могут возникнуть исключения?
Формат: три пункта с кратким объяснением каждого.

Анализ результата.

Сравните ответ GPT со своими гипотезами.

Отметьте несоответствия: возможно, модель «увидит» зависимость от feature-флага, о котором вы забыли.

Пример. В банкинг-сервисе функция валидации платежа неожиданно использовала устаревший метод хэширующего алгоритма. Reverse-prompt выявил, что внутри validate() есть вызов crypto.createHash('md5'), хотя весь проект давно перешёл на SHA-256.

Частая ошибка. Запрашивать сразу «Перепиши функцию под новый стандарт» без предварительного понимания логики. Итог – «чёрный ящик»: вы получаете код, который проходит тесты, но не понимаете, как он работает и какие ограничения заложены изначально.

Практический совет. Для каждой «критической» функции сохраняйте Reverse-prompt и ответ модели в репозитории /docs/legacy-analysis. Это позволит новым членам команды быстро погружаться в ключевые части системы.

7.2 GPT как комментатор и документатор

После того как основные сценарии выписаны, перейдите к авто-документированию. GPT выступит не «писателем всей документации», а ассистентом-комментатором, который добавит пояснения прямо в код:

Настройка шаблона.

role: documentator model: gpt-4o prompt: | Ты – senior-разработчик TypeScript. Дополни все функции в этом файле JSDoc-комментариями: – Объясни назначение каждого параметра. – Опиши возвращаемое значение. – Укажи возможные исключения.

2. **Запуск через Cursor или Copilot Workspace.**
– Выделяете весь файл `orderProcessor.ts`.
– Отправляете запрос «Добавь JSDoc».
– Получаете файл с комментариями.

3. **Проверка и доработка.**
– Убедитесь, что комментарии отражают фактические ограничения (например, диапазон числовых полей).
– Дополни их, если GPT упустил «узкие места».

> **Парадокс.** Чем больше комментариев добавляет модель, тем строже вы их проверяете. В итоге документация становится точнее, чем та, что писали вручную, и гораздо более актуальной.

Частая ошибка. Интегрировать сгенерированные комментарии «вслепую». Комментарии GPT могут описывать устаревший код. Всегда делайте ревью каждого блока – лучше убрать лишний JSDoc, чем держать в комментариях неверную информацию.

Практический совет. Автоматизируйте проверку JSDoc-стиля через ESLint-плагин `eslint-plugin-jsdoc`. Любой PR без комментариев вызывает red-build, а вырабатывает привычку не пропускать документирование.

7.3 Циничный тон для поиска code-smells
После формального анализа переходим к «хардкор-режиму»: опытная модель встаёт на сторону чужого скептика и ищет потенциальные проблемы.

1. **Шаблон циничного промпта.**
Ты – «хардкор-ревьюер» с 15-летним стажем, недолюбливающий «краденый» код. Прокомментируй функции этого модуля, отмечая все code-smells:
Дублирование логики
Глубокую вложенность
«Магические числа»

Формат: список проблем с указанием строк и кратким объяснением.
2. **Проходим по модулям партиями по 200–300 строк.**
– Разбивайте legacy-файлы, иначе модель «утонет» в объёме.
– После каждого фрагмента фиксируйте список рефакторинг-задач в Jira или GitHub Issues.

3. **Приоритизация.**
– Высокий приоритет: потенциальные уязвимости, утечки памяти, race-condition.
– Средний: избыточная сложность, избыточные зависимости.
– Низкий: cosmetic-smells – названия переменных, комментарии-пустышки.

> **Пример.** В модуле работы с файлами GPT нашёл «магическое» значение `bufferSize = 8192`. В комментарии ревьюера – рекомендацию вынести эту константу в конфиг и добавить проверку на `maxBufferSize`, что предотвратит OutOfMemoryError при больших файлах.

Частая ошибка. Оставлять такие code-smells в бэке без трекинга: «потом пофиксим». Лень фиксировать задачи приводит к накоплению технического долга.

Практический совет. Объединяйте задачи code-smells в спринты минимального объёма: каждую неделю хотя бы одна проблема из High-приоритета закрывается. Это даёт ощущение прогресса и снижает страх перед «большим рефакторингом».

Итоги главы

Разбор Legacy-кода – это не бессмысленное ковыряние в чужих недрах, а хорошо организованный процесс:

1. **Reverse-prompt** помогает сформировать карту функциональных блоков.
2. **GPT-документатор** добавляет точные, проверенные комментарии прямо в код.
3. **Циничный тон** выявляет скрытые проблемы и формирует backlog рефакторинга.

Следуя этой методике, вы не просто модернизируете старый проект, а создаёте живую базу знаний и постепенно снижаете технический долг, сохраняя фокус на реальных бизнес-цельях.

Глава 8. Рефакторинг одним запросом

«Рефакторинг в эпоху Vibe-кодинга – это не изнурительная эпопея, а серия целевых запросов к модели: опиши, перепиши, проверь.»

8.1 Explain → Refactor → Validate: трёхэтапный паттерн

Шаг 1. Explain. Опишите моделью существующую логику и потенциальные проблемы.

Ты – senior-разработчик на JavaScript. Опиши, что делает функция processData(data) в этом файле, выдели дублирующийся код, глубокие вложенности и потенциальные утечки памяти.

Модель выдаёт структурированный ответ:

Разбор массива через вложенный forEach (строки 12–27).

Повторяющийся фрагмент форматирования дат.

Массив temp не очищается при ошибках – возможная утечка.

Шаг 2. Refactor. По полученному объяснению попросите переписать код:

На основе описания и найденных проблем сгенерируй refactor-файл processData.js: – Заменить вложенные `forEach` на `for…of` – Вынести форматирование даты в отдельную функцию formatDate() – Добавить try/catch и очистку временных массивов Формат: unified diff-патч.

Шаг 3. Validate. Уточните, что результат прошёл проверку:

Проведи self-review патча в виде git diff, проверь совпадение с ESLint-конфигом и корректность логики.

Результаты. Сокращение кода на 18 строк, устранение дублирования и гарантия чистого выполнения – всё в одном пулл-реквесте.

Преимущество: среднее время на ручной рефакторинг снижается на 30–40 % при рутинных задачах .

Пункт контроля: всегда применяйте diff-патчи через Cursor или Copilot, а не вставляйте «as-is».

Частая ошибка. Просить «refactor сразу и всё» без этапа Explain: модель упускает скрытые зависимости и создаёт «черный ящик».

Практический совет. Храните шаблон-поток Explain→Refactor→Validate в каталоге / prompts/refactor/ и версионируйте по semantic versioning.

8.2 Пример: миграция jQuery → Vanilla JS

Исходный код на jQuery:

$('.btn').on('click', function() { const $item = $(this).closest('.item'); $item.find('.value').text(parseInt($item.find('.value').text()) + 1); });

Prompt:

Ты – фронтенд-инженер. Перепиши этот код на чистый ES6-JavaScript без jQuery, используя event delegation и современный API DOM. Формат: два блока – diff-патч и итоговый файл eventHandler.js.

Сгенерированный diff:

— before.js +++ eventHandler.js @@ -1,4 +1,12 @@ -$('.btn').on('click', function() { – const $item = $(this).closest('.item'); – $item.find('.value').text(parseInt($item.find('.value').text()) + 1); -}); +document.addEventListener('click', event => { + if (!event.target.matches('.btn')) return; + const item = event.target.closest('.item'); + const valueEl = item.querySelector('.value'); + const

current = Number(valueEl.textContent); + valueEl.textContent = current + 1; +}); + +export default eventHandler;

Результат.

Убрана зависимость от jQuery, уменьшен бандл на 15 КБ.

Код проходит npm run build без ошибок.

Модель сэкономила до 4 ч на ручном переписывании и проверке.

Частая ошибка. Не учитывать кросс-браузерные нюансы: забывают event.target.closest полифилы.

Практический совет. В prompt сразу указывайте: «Добавь проверку closest-полифилами для IE11», если нужно покрытие старых браузеров.

8.3 Парадокс «1 % diff – 99 % эффекта»

Суть парадокса. Небольшая правка (1 % строк) может дать почти весь выигрыш в стабильности, скорости загрузки и читаемости – вплоть до 99 % эффекта на «большом» коде. Маленькие, частые изменения интегрируются и ревьюятся быстрее, чем единичный большой коммит, снижая риск конфликтов и ускоряя доставку релизов на 50 % .

Пример. После предложения LLM вынести 2 линии форматирования даты в функцию formatDate() и подключить Intl.DateTimeFormat,

число баг-репортов упало с 8 до 1 в месяц,

среднее время ревью PR сократилось с 4 ч до 1,8 ч.

Практический совет.

Планируйте рефакторинг по «микро-задачам»: 5–10 изменений за sprint.

Отмечайте в Jira задачи «refactor: extract helper», «refactor: improve naming» и поручайте их модели.

Измеряйте эффект через DORA-метрики: lead time и change failure rate.

Заключение

Паттерн **Explain → Refactor → Validate** превращает рефакторинг в управляемый процесс: модель выявляет проблему, предлагает конкретный патч и проверяет соответствие стандартам. На примере миграции jQuery → Vanilla JS видно, как один запрос генерирует полный diff, экономя часы ручной работы. А принцип «1 % diff – 99 % эффекта» доказывает: фокус на маленьких изменениях ускоряет доставку и укрепляет качество кода. Внедрите эти практики в каждый спринт, и Vibe-кодинг поднимет эффективность вашей команды на новый уровень.

Глава 9. Документация & auto-README

«README – это не красивый маркетинговый текст, а технический мост между проектом и вновь прибывающим разработчиком».

Время от времени проект сталкивается с «бутылочным горлышком» в онбординге: свежему инженеру нужно разбираясь в сотнях файлов, понять, как собирать, запускать и развивать приложение. Именно README – это первая точка входа, где концентрируется техническая информация. В традиционных проектах этот файл либо устаревал, либо превращался в «красивую» витрину без сути. Vibe-кодинг предлагает подойти к документированию как к автоматизированному процессу: генерировать скелет Readme через LLM, поддерживать его в актуальном состоянии и превращать каждую новую фичу в блок инструкции.

9.1 TL;DR + Installation + Contribution templates

Самостоятельная генерация README разбивается на три ключевых секции:

TL;DR – краткое описание проекта, его целей и ценности в 2–3 предложениях.

Installation – пошаговая инструкция по установке и запуску (примеры команд, системные зависимости, проверка окружения).

Contribution – правила для желающих внести изменения: кодстайл, ветвление, тестирование, PR-шаблон, контактные лица.

Пример шаблона (Markdown):

Название проекта ## TL;DR Кратко: этот сервис собирает метрики пользовательских сессий и экспортирует их в ClickHouse. ## Installation ```bash git clone https://…/repo.git cd repo docker-compose up -d npm run migrate npm start

Usage

Перейдите по http://localhost:3000

Отправьте GET /metrics?from=…&to=…

Contribution

Форкните репозиторий

Создайте ветку feature/имя-функции

Покройте код тестами (npm test)

Откройте Pull Request по шаблону:

Описание: что изменилось и зачем

Как проверить: шаги для воспроизведения

Checklist: выполнены ли lint, тесты, документация

Авто-генерация через GPT:

```text

Ты – technical writer. Сгенерируй README для репозитория service-metrics:

– TL;DR (< 50 слов)

– Installation с командами для Docker и локального запуска

– Пример запроса и ответа

– Contribution с чек-листом PR

Формат – Markdown.

Модель выдаст готовый файл, где каждую секцию можно влить в /README.md и сразу проверять в CI.
```

Парадокс. Чем проще шаблон, тем больше охвата: короткий, понятный README приводят к тому, что разработчики реально читают его, а не игнорируют.

9.2 Ошибка: README = маркетинг

Симптомы «маркетингового» README

Большие картинки и баннеры вместо кода и инструкций.

Отсутствие конкретных команд для локального запуска.

Распространение «описания фич» вместо описания архитектуры.

В результате новый инженер тратит до 2 дней, блуждая по исходникам, вместо быстрого старта, даже если проект «круто» описан визуально.

Частая ошибка. Копировать ссылку на публичный сайт продукта или показывать скриншоты, забывая указать npm install и ENV-переменные.

Практический совет.

Всегда начинайте с команды. Любой раздел «Installation» должен содержать хотя бы одну тестовую команду на клонирование, установку и простой запуск.

Раздел «Usage» должен включать примеры HTTP-запросов, CLI-команд или сценариев. Читатель должен получить первый рабочий результат за 5 минут.

Contribution оформляйте не маркетингово, а как строгий регламент: шаблон PR-описания, требования к тестам и обязательный чек-лист автоматических проверок в CI.

Автоматический контроль. Введите CI-правило: при каждом Pull Request проверять, что README.md содержит секции ## Installation и ## Contribution. Если одна из них отсутствует – сборка падает, а PR остаётся на доработке.

Практические рекомендации

Версионируйте README вместе с кодом. Каждый PR обновляет не только реализацию, но и документацию.

Интегрируйте автогенерацию в pre-commit hook. Добавьте скрипт, который вызывает LLM для ребилда README после изменений API.

Используйте badges. Добавьте статусы сборки, покрытие тестами, версию NPM, ссылку на документацию – но не вместо инструкций.

Регулярно проводите «Doc-Grooming». Каждые два спринта проверяйте, что примеры запросов работают, а конфигурации не устарели.

Итог главы

README – это не иллюстрация «красивого мира» продукта, а первое руководство по выживанию в проекте. Vibe-кодинг предлагает автоматизировать процесс создания и поддержания документации через LLM-ассистентов: шаблонный генератор README, контроль наличия ключевых секций в CI и регулярное «Doc-Grooming». Такой подход гарантирует, что новый разработчик запустит проект за минуты, а команда не потеряет время на «догонку» устаревшей информации.

Глава 10. Code-review 2025: человек + LLM

В эпоху Vibe-кодинга код-ревью становится гибридным процессом, где искусственный интеллект берёт на себя рутину, а человек фокусируется на стратегических и архитектурных вопросах. По данным **GitHub Code Review Survey Q1 2025**, команды, которые внедрили автоматизированные проверки через LLM, сократили среднее время прохождения PR-цикла на 38 % и одновременно снизили число багов на 24 % благодаря раннему выявлению уязвимостей. Однако без продуманного совмещения машины и человека риск «шумового фона» автозамечаний и потери контекста остаётся высоким.

10.1 Чек-лист автозамечаний

Автоматизированные помощники способны анализировать сотни строк кода за секунды и выдавать сотни рекомендаций. Чтобы превратить это в полезный инструмент, сформируйте универсальный чек-лист автозамечаний:

Стиль и форматирование

Проверка соответствия внутреннему Style Guide (ESLint, Prettier).
Выявление несоответствий отступов, длин строк, нотации имён.

Безопасность

SQL- и XSS-сканирование через специализированные LLM-prompt'ы.
Проверка на открытые порты и некорректное обращение с секретами.

Производительность

Выявление «горячих циклов» и неэффективных алгоритмов.
Оценка resource-leaks (незакрытых соединений, неосвобождённой памяти).

Тестовое покрытие

Проверка наличия unit- и интеграционных тестов для новых функций.
Сверка уровня покрытия с порогами (обычно ≥ 80 %).

Документация

Отслеживание изменений в API и README.
Автоматическая генерация и проверка JSDoc/Swagger-описаний.

Архитектурные паттерны

Детекция антипаттернов (God Object, глубокие вложенности).
Проверка границ модулей и зависимостей.

Частая ошибка: пытаться включить в один PR список более чем из 100 рекомендаций. Автозамечания превратятся в «черновик избыточных правок», который никто не особенно читает.

Практический совет: разбивайте автозамечания по категориям и приоритизируйте. Пусть на одну MR-сессию уходит не больше 15–20 подсказок: сначала безопасность, затем стиль, потом остальное.

10.2 Выбор «тона» ревью

Тон сообщений модели во многом определяет, как команда воспримет замечания. По исследованию **Microsoft Developer Velocity Index 2024**, PR-с со «собранным» дружелюбным тоном принимаются в среднем на 22 % быстрее, чем PR, где комментарии звучат «множество ошибок».

Нейтральный: лаконичные указания без оценочных суждений.

Конструктивно-дружественный: поощрительные фразы («Отличная идея, давай чуть улучшить…»).

Строго-директивный: императивный стиль («Замени это на…, убери то…»).

Парадокс: слишком мягкий тон порождает «размытые» правки – команда не понимает приоритета, а слишком строгий демотивирует и повышает конфликтность.

Частая ошибка: использовать по умолчанию «system prompt» вроде “You are a code reviewer” без уточнения стиля. Итог – бессвязные комментарии, словно из разных авторов.

Практический совет: храните в каталоге /prompts/review_tone/ три варианта шаблонов:

tone_friendly.md с примерами вежливых фраз;

tone_neutral.md без лишних эпитетов;

tone_strict.md для критических секций безопасности.

Перед запуском самопроверки автоматически выбирайте нужный стиль, исходя из уровня ответственности кода.

10.3 Совет: финальное слово – старший разработчик

Несмотря на мощь LLM, окончательное право решения всегда должно оставаться за живым человеком. **Исследование JetBrains AI Tools Survey IV кв. 2024** показало: в 12 % проектов, где автозамечания сразу вливались без человеческой проверки, возникали непредвиденные регрессии в продакшене.

Почему нужен «человек-финал»:

Контекстные знания: модель не знает внутренних договорённостей команды и бизнес-контекста.

Архитектурная целостность: ИИ может предложить локально корректное, но глобально разорванное решение.

Эмоциональный интеллект: живой ревьюер оценивает «тон» PR-описания и поддерживает мотивацию автора.

Частая ошибка: полностью автоматизировать мёрдж через Copilot Agents, установив правило «merge on successful tests and zero critical alerts». Это лишает команду возможности учиться на своих ошибках.

Практический совет: назначьте **«ревью-чемпиона»** – старшего разработчика, который каждую пятницу проводит выездной аудит автозамечаний и принимает итоговые решения по мёрджам. Периодически меняйте этого человека, чтобы у команды сохранялась «свежее» восприятие процесса.

Итог главы

Гибридный формат code-review «человек + LLM» обеспечивает максимальную скорость и надёжность. Автоматизированные чек-листы отнимают рутину, настроенный «тон» формирует позитивную атмосферу, а финальный контроль старшего разработчика гарантирует сохранение архитектурной и бизнес-целостности. Внедрив эти практики, вы достигнете баланса между скоростью поставки и качеством кода, соответствующим требованиям рынка 2025 года.

Глава

11.

Интеллектуальные

IDE: Cursor, Copilot Workspace, JetBrains AI

В 2025 году профессиональный разработчик всё чаще оценивается не по скорости набора кода, а по скорости принятия и валидации изменений. Интеллектуальные IDE не просто подсказывают фрагменты – они становятся полноценными «агентами», помогающими организовать работу над фичей от идеи до Pull Request. В этой главе рассмотрим три лидера рынка: **Cursor**, **Copilot Workspace** и **JetBrains AI Assistant**, их ключевые возможности, практические сценарии и подводные камни.

11.1 Обзор AI-редакторов (функции, цены)

Cursor

Модельные агенты: несколько одновременных «сессий» для тестов, рефакторинга и генерации кода.

Интерфейс: потоковый diff прямо в редакторе, возможность «принять/отклонить» каждое изменение.

Контекст: до 100 000 токенов на сессию – подходит для больших репозиториев.

Цены: от $30/месяц за базовый план, корпоративные пакеты от $50/пользователь с поддержкой on-premise.

GitHub Copilot Workspace

Авто-драфты PR: копирайт-агент генерирует структуру ветки и описания к PR на основе коммитов.

Сценарии: Unit-тесты, doc-строки, CI-правила автоматически подставляются при создании ветки.

Интеграция: нативно встроен в GitHub, поддерживает приватные репозитории.

Цены: $20/месяц за пользователя, корпоративный тариф – $40 с расширенными возможностями аудита.

JetBrains AI Assistant

Локальный режим: работает с Llama-3-Instruct Q4 прямо на ноутбуке без интернет-подключения.

Инспекции кода: глубокий анализ архитектуры, проверка зависимостей по проектному Style Guide.

UI-интеграция: inline-заметки, справа от строки появляются подсказки и quick-fix.

Цены: включён в All Products Pack (≈$249/год), отдельного тарифа нет.

Парадокс: чем больше функций «под капотом» у AI-редактора, тем более сложно объяснить новичку, какая из них сработала, и почему. Поэтому важно сочетать мощь агента с понятными инструкциями.

11.2 Cursor: Agent-режим, multi-edit, smart-rewrites и off-line-кэш

Agent-режим в Cursor – это несколько виртуальных помощников, каждый со своей задачей:

Generator отвечает за черновой код по prompt-ам.

Reviewer сканирует diff-файлы и предлагает улучшения.

Navigator ищет по документации и Confluence-страницам.

Multi-edit позволяет сразу выделить несколько независимых фрагментов и получить один патч для всего файла. Это удобно при однотипных правках: обновлении package-версий или рефакторинге имен.

Smart-rewrites работают «на уровне смысла»: вы просите «сделать функцию более декларативной», и Cursor сам вставляет map/filter вместо for-loop.

Off-line-кэш – ключевая фишка для экономии токенов и скорости:

После первой генерации Cursor сохраняет локально часть модели веса (≈200 МБ).

При повторном обращении Cache-агент отрабатывает мгновенно (задержка <50 мс).

Можно настроить lifelike-режим: смешанный on-premise → облако при недостатке локальных ресурсов.

Практический пример:

Вы хотите обновить все тестовые файлы из Mocha на Jest:

Выделяете test/**/*.spec.js.

Запускаете Agent Convert Mocha to Jest.

Получаете единый diff-патч с заменами describe → test, before → beforeEach и обновлёнными импортами.

При необходимости правите отдельные кейсы, но в 90 % случаев «под капотом» Cursor сделал всю работу.

11.3 Практика: маршрут «Cursor → PR auto-draft»

Инициация фичи

Создаёте ветку feature/add-payment-gateway.

В открытом файле handler.js запускаете Cursor-агента:

Ты – senior-backend на Node.js. Добавь поддержку Stripe API:

– новый endpoint POST /payment

– валидация входных данных

– логирование в Winston

Генерация и self-review

Agent-Generator возвращает каркас handler.js.

Agent-Reviewer тут же делает самопроверку: ищет несанкционированные операции на базе, шлет git-патч.

Конец ознакомительного фрагмента.