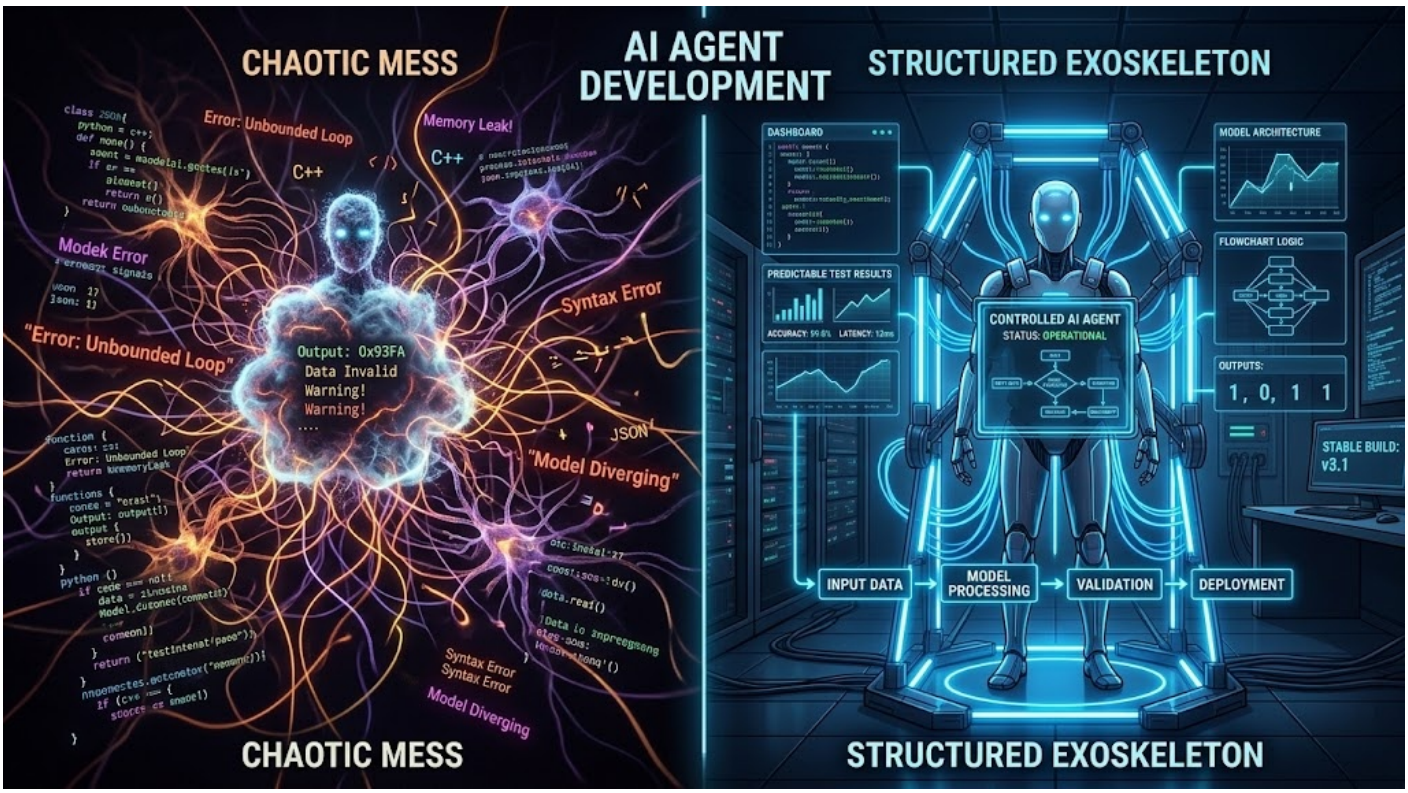
Харнес для AI-агента

- Как за вечер сделать результат предсказуемым

Как за вечер сделать результат предсказуемым



Эта статья — про то, почему AI-агент даёт разный результат на один и тот же запрос и что с этим делать. Она рассчитана на начинающего разработчика: достаточно уметь пользоваться Git и командной строкой и хотя бы раз попробовать AI-ассистента. Опыт построения агентных систем не нужен.

Читается примерно за полчаса. Практическая часть — раздел 5 — рассчитана на один вечер работы.

Оглавление

---

1. Почему агент работает нестабильно

В понедельник вы просите агента: «Добавь эндпоинт, который отдаёт задачу по id». Проект — TaskBoard, небольшой веб-сервис со списком задач: десяток файлов и тесты. Агент находит каталог с обработчиками запросов, добавляет туда файл, пишет тест в том же стиле, что и соседние, запускает проверки и показывает, что они проходят. Вы принимаете изменения почти без правок.

В среду вы просите то же самое для другого поля. Агент создаёт новый каталог `api/`, пишет тест на фреймворке, которого в проекте нет, и заканчивает словами «готово». Тесты он не запускал — просто решил, что задача выполнена.

Модель за два дня не поглупела. Изменилось окружение. В понедельник выше по диалогу случайно оказался кусок вашего кода, и агент увидел, как всё устроено. В среду диалог был новый, и агент восстанавливал устройство проекта по догадкам. Догадки у него неплохие — но это догадки.

Обычная реакция — писать запрос длиннее: добавить в конец абзац «клади обработчики в `src/handlers/`, тесты пиши на нашем фреймворке, обязательно запусти проверки». Это работает — ровно один раз. У приёма несколько слабых мест, и три видно сразу:

1. **Запрос живёт в диалоге и исчезает вместе с ним.** Завтра всё сначала.
2. **Вы пересказываете одно и то же.** Пять минут на объяснение проекта каждый раз — это несколько часов в месяц.
3. **Запрос — это просьба, а не ограничение.** «Не трогай файлы миграций» и «файлы миграций технически недоступны для изменения» — очень разные вещи. Первое модель может проигнорировать, второе — нет.

Третий пункт стоит задержать. Разница между просьбой и ограничением — это разница между «агент обычно так не делает» и «агент так сделать не может». Первое вы обнаруживаете постфактум, разбирая, что произошло. Второе не происходит вовсе.

Четвёртое слабое место менее очевидно. Длинный запрос конкурирует сам с собой: чем больше требований вы перечисляете в одном абзаце, тем меньше веса у каждого.

Требование, записанное отдельно и одинаковое для всех задач, работает устойчивее, чем то же требование, дописанное в конец очередного сообщения.

Дальше в статье — как заменить эту просьбу записанным окружением. За вечер вы получите файл с описанием проекта и командами, настроенные разрешения, записанный критерий готовности и способ проверить, что всё это действительно работает. Из полного набора возможностей это меньшая часть — и именно та, которая даёт наибольший эффект.

---

2. Что такое харнес

Харнес (*harness*) — это окружение, в котором работает агент: контекст проекта, инструкции, доступные инструменты, разрешения, порядок работы, проверки качества и правила остановки.

Проще говоря: модель решает, что сделать, а харнес определяет, что вообще возможно сделать и что считается сделанным.

Откуда слово

В инженерии *harness* — это оснастка: окружение, в котором запускают и проверяют что-то другое. Самый близкий пример — *test harness*: код, который готовит условия, запускает тестируемый компонент и собирает результаты. Сам компонент при этом не меняется, меняется окружение вокруг него.

С AI-агентом та же идея. Модель вы не переписываете. Вы строите вокруг неё окружение, в котором её решения превращаются в предсказуемые действия.

Харнес есть всегда

Даже если вы ничего не настраивали, окружение у агента всё равно есть. Просто оно собралось случайно:

- контекст — то, что случайно попало в диалог;
- инструкции — то, что вы дописали в конце запроса;
- инструменты — те, что включены по умолчанию;
- разрешения — те, что стоят «из коробки»;
- проверка — вы сами, глазами.

Последний пункт стоит отметить отдельно. Пока проверяете вы, проверка есть — но агенту она недоступна. Правило, которое существует только у вас в голове, он не видит и выполнить не может. Это ваш личный контроль, а не часть харнеса.

Отсюда главная мысль раздела. Вопрос не в том, есть окружение или нет. Вопрос в том, записано оно или собралось само. Незаписанное окружение нельзя повторить, объяснить коллеге и починить, когда оно подведёт.

Пример: та же задача, другое окружение

В разделе 1 было «до». Вот «после».

В корне TaskBoard лежит файл на двадцать строк: где находятся обработчики запросов, каким тестовым фреймворком пользуются, какими командами запускаются линтер и тесты. И одно правило: задача не считается выполненной, пока эти две команды не проходят.

Тот же запрос про эндпоинт, тот же агент, та же модель. Агент открывает `src/handlers/`, добавляет файл рядом с соседними, пишет тест в том же стиле, запускает линтер и тесты, показывает вывод. Результат ложится в проект, а не рядом с ним.

Обратите внимание, чего здесь не было. Не было длинного запроса, не было хитрой формулировки и не было ни одного «умного» промпта. Была записанная информация о проекте и одно правило о готовности. Это и есть минимальный харнес — целиком.

Что харнес не делает

- **Не делает модель точнее.** Он направляет и ограничивает, а не улучшает рассуждения.
- **Не заменяет тесты и ревью.** Он делает их обязательной частью процесса.
- **Не гарантирует правильный результат.** Он повышает шанс вовремя заметить неправильный.

Одно заблуждение, которое стоит разобрать сразу

Самое частое: «харнес — это просто большой системный промпт».

Промпт — часть харнеса, причём та, которую модель может проигнорировать. Сравните две записи одного и того же намерения. Первая: в инструкции написано «не удаляй файлы». Вторая: команда `rm` запрещена правилом `deny` на уровне разрешений. В первом случае удаление зависит от того, как модель поняла контекст. Во втором — команда не выполнится, даже если модель твёрдо решила, что так будет лучше.

Промпт просит. Разрешения, обязательные проверки и недоступные инструменты — ограничивают. Хороший харнес пользуется и тем, и другим, но не путает одно с другим.

---

3. Из чего состоит харнес

Харнес удобно разбирать на десять блоков. Это не обязательный список, а словарь: с ним вы можете посмотреть на любой AI-инструмент и сказать, что в нём есть, чего нет и чего вам не хватает.

Польза словаря видна, когда что-то ломается. «Агент работает плохо» — бесполезная формулировка. «Агент не видит структуру проекта» или «у нас нет обязательной проверки после правки» — рабочая: сразу понятно, что чинить.

Блок	Вопрос, на который отвечает	Что ломается без него
Контекст проекта	Что агент знает о проекте	Агент угадывает структуру и придумывает несуществующие функции
Инструкции	Как здесь принято работать	Код формально верный, но чужой для проекта
Инструменты	Что агент физически может сделать	Агент советует вместо того, чтобы делать; или делает то, чего вы не ожидали
Permissions	Что из возможного разрешено	Опасные действия выполняются молча
Workflow	В каком порядке идут шаги	Шаги пропускаются, порядок каждый раз разный
Quality gates	Когда результат считается годным	«Готово» означает «модель так решила»
Human approval	Где обязателен человек	Необратимые действия происходят без вашего ведома
Состояние задачи	Что помнить между шагами и запусками	После перезапуска работа начинается заново
Наблюдаемость	Как понять, что произошло	Разобрать ошибку невозможно: виден только итог
Оркестрация	Кто участвует, если агентов несколько	Роли размываются, агенты мешают друг другу

Десять блоков запоминаются плохо, три группы — хорошо:

- **Что агент знает** — контекст и инструкции.

- **Что агент может** — инструменты и permissions. Разница между ними существенная: инструмент даёт физическую возможность действия, permissions — право его выполнить. Инструмент, который есть, но запрещён, безопаснее инструмента, которого нет: во втором случае агент попробует добиться того же обходным путём.
- **Как агент работает** — workflow, quality gates, human approval, состояние задачи, наблюдаемость, оркестрация.

Дальше подробно — только четыре блока. Это те, которые нужны почти всем и собираются за вечер. Остальные шесть — в конце раздела, одной строкой каждый.

Контекст проекта

Что это. Записанное описание того, как устроен проект: язык и версия, структура каталогов, где лежат обработчики запросов, где тесты, где конфигурация, какие есть внешние зависимости.

Что ломается без него. Агент восстанавливает устройство проекта по догадкам — по названиям файлов, по тому, что чаще встречается в других проектах. Отсюда файлы не в тех каталогах и обращения к функциям, которых у вас нет.

Минимальная версия. Двадцать строк в файле в корне репозитория. Не пересказ всего проекта, а ответы на вопросы, которые вы чаще всего объясняете вслух. Хороший способ собрать этот список — вспомнить, что вы дописывали в конец последних пяти запросов.

Важная оговорка: контекст — ограниченный ресурс, а не «чем больше, тем лучше». Инженеры Anthropic описывают эффект, который называют *context rot*: с ростом числа токенов в окне контекста точность модели при обращении к этой информации падает. Поэтому файл с описанием проекта на десять экранов работает хуже, чем на половину экрана: в нём тонет то, что действительно важно.

Инструкции

Что это. Правила работы в этом проекте: как принято называть, как оформлять, какими командами запускать сборку, тесты и линтер. Отдельная и самая ценная часть — **канонические команды**: одна правильная команда на каждое действие.

Что ломается без него. Код получается формально верным, но чужим: другой стиль, другой способ обработки ошибок, тест на фреймворке, которого в проекте нет. И агент запускает не то: `npm test` вместо `npm run test:unit`, а потом сообщает, что «всё сломано».

Минимальная версия. Список из пяти-семи команд с подписью, что каждая делает, плюс три-четыре правила стиля, которые вы реально требуете на ревью. Правила, которые вы не проверяете сами, писать не нужно: они устареют первыми.

Permissions (разрешения)

Что это. Правила о том, что агенту разрешено делать. Различайте три режима, и разница между ними принципиальна:

- `allow` — выполняется сразу, без вопроса;
- `ask` — выполняется только после вашего подтверждения;
- `deny` — не выполняется вообще.

Что ломается без него. Опасные действия выполняются молча. Агент, который может выполнять произвольные команды, рано или поздно выполнит ту, которую вы не ожидали: удалит каталог, перезапишет файл миграции, отправит запрос наружу.

Минимальная версия. Запретить (`deny`) то, что нельзя откатить за минуту. Поставить `ask` на границе, где начинается необратимое: миграции, деплой, отправка данных наружу. Всё остальное — `allow`, иначе работа превратится в поток подтверждений, и вы начнёте нажимать «да» не читая. Это отдельный риск: разрешение, которое подтверждают не глядя, защищает хуже, чем отсутствие разрешения.

Есть исключение из общего правила «блок заводят, когда появилась проблема». Permissions — единственный блок, который стоит завести заранее, потому что его очередь определяет не удобство, а риск. Если агент уже может выполнить действие, которое нельзя откатить, ждать первой проблемы поздно: эта первая проблема и будет тем самым необратимым действием.

Quality gates (проверки качества)

Что это. Записанный критерий, при котором результат считается готовым. Обычно это набор проверок, которые обязаны пройти: линтер, тесты, сборка.

Что ломается без него. «Готово» означает «модель так решила». Агент искренне сообщает об успехе, потому что у него нет другого определения успеха, кроме собственного впечатления.

Минимальная версия. Одна фраза в файле инструкций: «Задача не выполнена, пока не проходят `npm run lint` и `npm test`. Если проверки не проходят, не пиши "готово" — покажи вывод команды». Формулировка выглядит тривиальной, но именно она превращает готовность из мнения в факт.

Здесь же — граница честности. Такое правило записано словами, а значит, относится к просьбам, а не к ограничениям. Настоящую гарантию даёт проверка на стороне, которую агент не контролирует: pre-commit hook или CI. Начинать стоит с записанного правила — оно бесплатное и закрывает большинство случаев, — а жёсткую проверку добавлять тогда, когда правило начнёт нарушаться.

Остальные шесть блоков

Они не хуже — они просто нужны позже. Каждый добавляйте, когда увидите его признак:

Блок	Когда понадобится
Инструменты	Когда набора по умолчанию перестанет хватать: нужен доступ к базе, трекеру задач, внутреннему API
Workflow	Когда у вас появится тип задач, который повторяется еженедельно и требует одинакового порядка шагов
Human approval	Как только у агента появится доступ к необратимому: деплой, миграции, письма наружу
Состояние задачи	Когда задачи перестанут помещаться в один запуск и после перезапуска работа начнётся с нуля
Наблюдаемость	Когда разбор «а что вообще произошло» начнёт занимать заметное время
Оркестрация	Когда появится конкретная причина разделить работу между агентами — см. следующий раздел

Общее правило: блок нужен тогда, когда есть проблема, которую он решает. Блок, добавленный заранее, — это стоимость поддержки без результата.

---

4. Харнес и оркестратор

Про несколько агентов сегодня пишут больше, чем про один настроенный. Схемы выглядят убедительно: один агент планирует, другой пишет код, третий проверяет результат. Разберёмся, где здесь польза, а где просто больше движущихся частей.

Харнес — это окружение одного агента. Что он знает, что может, что считается готовым.

Оркестратор (*orchestrator*) — это распределение работы между несколькими агентами: кто за что отвечает, в каком порядке они вступают, как передают результат друг другу.

Разница видна на вопросе. Харнес отвечает: «Что нужно агенту, чтобы сделать работу правильно?» Оркестратор отвечает: «Кто делает какую часть?» Второй вопрос имеет смысл только после первого.

Роли в таких схемах обычно называют одинаково: **planner** составляет план, **implementer** пишет код, **reviewer** проверяет результат. Названия удобные, и дальше разворачивать их не будем — потому что польза схемы зависит не от названий, а от того, есть ли у разделения причина.

Когда оркестратор не нужен

Главный критерий короткий: **если агент ошибается из-за нехватки контекста, второй агент этого не чинит.**

Агент, который не знает, где лежат обработчики запросов, не начнёт знать это оттого, что рядом появился «архитектор». Он получит те же неполные данные — плюс ещё один слой пересказа, в котором часть деталей потеряется. Система из пяти агентов без контекста ошибается по тем же причинам, что и один агент без контекста, только дороже и дольше.

Проверка простая: возьмите последние пять неудачных запусков и спросите про каждый — агенту не хватило информации или не хватило рук? Если почти везде информации, оркестратор подождёт.

Единственный честный аргумент за разделение

Он есть, и он один: **независимая проверка**.

Агент, который написал код, — плохой проверяющий этого кода. Он видит своё решение как правильное и склонен подтверждать сам себя. Отдельный проверяющий агент запускается с чистым контекстом, видит только результат и не участвовал в том, как до него дошли. Это не гарантия — но это настоящая, а не декоративная разница.

Два других аргумента встречаются реже, но тоже реальны: **изоляция контекста** (отдельный агент выполняет объёмный поиск и возвращает короткую выжимку, не засоряя основной диалог) и **параллельность** (несколько независимых частей делаются одновременно). Обратите внимание, что все три аргумента — про конкретный механизм. «Несколько агентов работают лучше, чем один» — не аргумент; сами по себе они качество не повышают.

Цена

Она измерима. По данным Anthropic, агенты расходуют примерно вчетверо больше токенов, чем обычный диалог с моделью, а multi-agent-схемы — примерно в пятнадцать раз больше. В той же публикации сказано прямо: задачи, где все агенты должны работать с общим контекстом или сильно зависят друг от друга, для таких схем подходят плохо, и большинство задач по написанию кода — как раз этот случай.

К счёту за токены добавьте то, что не измеряется деньгами:

- **потерю контекста при передаче** — каждый переход между агентами это ещё один пересказ, а пересказ теряет детали;
- **циклы исправлений** — reviewer возвращает работу, implementer правит, reviewer возвращает снова;
- **отладку самой схемы** — когда результат плохой, придётся выяснять, какой из агентов ошибся, а логов у вас, скорее всего, ещё нет.

Практический вывод: сначала доведите харнес одного агента до состояния, когда он стабильно решает ваши задачи. Второго агента заводите под конкретную причину — обычно это независимая проверка. Как это сделать — шаг 6 следующего раздела.

---

5. Внедрение за вечер: шесть шагов

Дальше — практика. Шаги нейтральные: они одинаковы для любого агентного инструмента. В конце раздела — тот же харнес, собранный на конкретном инструменте.

Шаг 0. Выберите одну реальную задачу

До того как что-то настраивать, выберите задачу, на которой будете проверять результат. Требования к ней: она настоящая (из вашего бэклога, а не выдуманная), небольшая (полчаса ручной работы) и типичная — из того класса, который вы даёте агенту чаще всего.

Запишите её одной фразой и отложите. К ней вы вернётесь на шаге 5. Без этого шага вечер закончится красивым файлом инструкций, о котором вы не сможете сказать, помог он или нет.

Шаг 1. Карта проекта

Создайте в корне репозитория файл с описанием проекта. Название зависит от инструмента; сейчас распространено `AGENTS.md`.

Что писать:

- язык, версия, менеджер пакетов;
- структура каталогов — только те, что важны: где код, где тесты, где конфигурация;
- три-пять фактов, которые невозможно вывести из структуры: почему модуль называется странно, какой каталог трогать нельзя, где живёт легаси.

Что не писать: пересказ содержимого файлов, историю проекта, всё, что агент прочитает сам за две секунды. Ориентир объёма — половина экрана. Помните про context rot из раздела 3: длинный файл не улучшает результат, а размывает его.

Быстрый способ собрать содержимое: откройте последние пять диалогов с агентом и выпишите всё, что дописывали в конец запросов. Это и есть ваш недостающий контекст, причём проверенный практикой.

Шаг 2. Канонические команды

В тот же файл добавьте раздел с командами: сборка, тесты, линтер, форматирование, локальный запуск. По одной команде на действие.

```
Команды проекта:
- установка зависимостей: npm ci
- тесты: npm test
- линтер: npm run lint
- локальный запуск: npm run dev
```

Важно, что команда ровно одна на каждое действие. Если у вас два способа запустить тесты, агент выберет тот, который чаще встречается в интернете, а не тот, который принят у вас. Напишите правильный и допишите одну строку про неправильный: «`npm run test:all` не используем, он требует базы».

Проверьте команды перед тем, как записывать. Файл инструкций с командой, которая не работает, хуже отсутствующего: агент честно выполнит её, получит ошибку и начнёт чинить не то.

Шаг 3. Permissions

Разделите действия на три группы. Начните с верхней строки таблицы — с того, чего нельзя допустить ни при каких условиях.

Режим	Что сюда попадает
`deny`	Необратимое и заведомо ненужное: удаление файлов, `git push --force`, команды, отправляющие данные наружу
`ask`	Действия на границе: изменение миграций и схемы базы, установка новых зависимостей, любые операции с рабочим окружением
`allow`	Всё остальное: чтение файлов, правка кода, тесты, линтер, обычные команды Git

Правило, о котором легко забыть: список `ask` должен быть коротким. Если подтверждения запрашиваются по десять раз за задачу, вы перестанете их читать, и защита исчезнет — формально оставшись на месте.

Шаг 4. Definition of Done

Запишите — именно запишите, а не удержите в голове — критерий готовности. Достаточно одного абзаца в файле инструкций:

```
Задача выполнена, когда:
1. Проходит npm run lint.
2. Проходит npm test.
3. Новый код покрыт тестом в том же стиле, что соседние тесты.
Если проверки не проходят, не сообщай о готовности:
покажи вывод команды и остановись.
```

Последняя строка нужнее, чем кажется. Без неё агент, встретив падающий тест, начнёт чинить его самостоятельно — иногда правкой самого теста. Явное требование остановиться и показать вывод превращает провал в понятный вам результат.

Правило, записанное словами, остаётся просьбой. Если через неделю окажется, что его нарушают, добавьте pre-commit hook или проверку в CI — то, что агент не контролирует.

Шаг 5. Прогон на реальной задаче

Вернитесь к задаче из шага 0. Откройте новый диалог — принципиально новый, без истории, — и дайте задачу одной фразой, ничего не объясняя дополнительно.

Смотрите не на код, а на поведение:

- нашёл ли агент нужные файлы сам;
- запустил ли канонические команды, не спрашивая;
- уперся ли в запрет, если задача его задевала;
- сказал ли «готово» до того, как проверки прошли.

Каждое «нет» указывает на конкретный блок: первое — на контекст, второе — на инструкции, третье — на permissions, четвёртое — на quality gates. Исправьте файл и повторите. Обычно хватает двух-трёх итераций, и это нормальная часть работы, а не признак неудачи.

Шаг 6. Первый второй агент

Этот шаг — не на первый вечер. Он здесь, чтобы вы узнали момент, когда он понадобится.

Признак, что пора. Агент делает работу технически правильно, но вы всё равно перечитываете каждую строку — и находите не ошибки, а несоответствия договорённостям проекта. То есть проверка стала регулярной, механической и описуемой словами.

Что завести. Одного проверяющего агента (reviewer) со своей инструкцией: что проверять, в каком порядке, что считать замечанием. Ему нужен доступ на чтение и запрет на правку — иначе он начнёт чинить найденное, и независимость проверки исчезнет.

Стоп-сигнал. Если, читая замечания reviewer, вы видите, что он не знает устройства проекта, — вернитесь к шагу 1. Второй агент не лечит нехватку контекста, он её удваивает.

Чего не делать на первом заходе

Четыре блока выглядят привлекательно, но на старте только замедлят:

- **Workflow.** Фиксированный порядок шагов имеет смысл, когда есть повторяющийся тип задач. Пока задачи разные, workflow будет мешать.
- **Состояние задачи.** Файлы прогресса и передача работы между сессиями нужны, когда задача не помещается в один запуск. Для получасовых задач это лишняя бухгалтерия.
- **Наблюдаемость.** Логи и разбор запусков заводят, когда есть что разбирать. На первой неделе вы и так видите каждый шаг.
- **Оркестрация.** См. раздел 4.

Сквозной пример: тот же харнес на OpenCode

Ниже — как эти шаги выглядят в конкретном инструменте. Взят OpenCode; в других инструментах идеи те же, названия другие. Всё описанное сверено с официальной документацией OpenCode 21 августа 2026 года — проверьте актуальность, прежде чем копировать.

Шаги 1, 2 и 4 — файл `AGENTS.md` в корне проекта. OpenCode читает инструкции из `AGENTS.md`, ища его от текущего каталога вверх; дополнительные файлы можно перечислить в поле `instructions` конфигурации. Для TaskBoard:

```
# TaskBoard

Веб-сервис со списком задач. Node.js 22, npm, TypeScript.
```

```
## Структура
- `src/handlers/` — обработчики HTTP-запросов, по файлу на ресурс
- `src/db/` — доступ к данным
- `migrations/` — миграции базы, менять только по явной просьбе
- `tests/` — тесты, по файлу на обработчик

## Команды
- тесты: `npm test`
- линтер: `npm run lint`
- локальный запуск: `npm run dev`

## Готовность
Задача выполнена, когда проходят `npm run lint` и `npm test`.
Если проверки не проходят — не сообщай о готовности, покажи вывод и остановись.
```

Здесь в одном файле собрано три блока: контекст (структура), инструкции (команды) и quality gates (готовность). Разделять их по файлам на старте не нужно.

Шаг 3 — файл `opencode.json` в корне проекта. Разрешения задаются в поле `permission`; каждое правило принимает значения `allow`, `ask` или `deny`:

```
{
  "$schema": "https://opencode.ai/config.json",
  "permission": {
    "bash": {
      "*": "ask",
      "git *": "allow",
      "npm *": "allow",
      "rm *": "deny"
    },
    "edit": {
      "*": "allow",
      "migrations/*": "ask"
    }
  }
}
```

Что здесь важно. Для `bash` правила сопоставляются с командой по шаблону, и выигрывает **последнее совпавшее** правило — поэтому общее `"*": "ask"` стоит первым, а частные ниже. Звёздочка в `"git *"` покрывает аргументы: без неё правило не сработает для `git status`. Шаблон `"npm *"` разрешает и `npm test`, и `npm run lint` — но заодно и `npm install`; если вам это не

нравится, перечислите нужные команды по отдельности. Правка файлов разрешена везде, кроме `migrations/`, где нужен ваш ответ, — это и есть human approval из раздела 3.

Шаг 6 — файл `.opencode/agents/reviewer.md`. Подагенты описываются markdown-файлами с фронтматтером; имя файла становится именем агента, а вызвать его можно упоминанием `@reviewer`:

```
---
description: Проверяет готовые изменения на соответствие правилам TaskBoard
mode: subagent
permission:
  edit: deny
  bash: deny
---
Ты проверяешь готовые изменения и не вносишь правки.

Проверь по порядку:
1. Обработчики лежат в `src/handlers/`, по файлу на ресурс.
2. У нового обработчика есть тест в `tests/`, в стиле соседних тестов.
3. Ошибки обрабатываются так же, как в соседних файлах.
4. Файлы в `migrations/` не изменены.

Выведи список замечаний. Если замечаний нет, так и напиши.
```

Обратите внимание на две строки `deny`. Именно они делают проверку независимой: агент не может ни поправить код, ни запустить команду, поэтому единственный его результат — список замечаний, который читаете вы.

---

6. Как проверить, что харнес работает

«Кажется, стало лучше» — плохой критерий: он зависит от настроения и от того, насколько удачной попалась задача. Ниже пять сценариев с однозначным результатом. Каждый занимает несколько минут, и каждый проверяет ровно один блок.

Общее условие: все пять запускайте в **новом диалоге**, без истории. Проверять харнес в диалоге, где вы только что всё объяснили, бессмысленно.

1. Агент находит нужный файл без подсказки. Попросите внести небольшое изменение, не называя файл: «добавь поле `priority` в задачу». Успех — агент сам открывает нужные файлы. Провал — спрашивает, где что лежит, или создаёт файл в новом месте. Проверяет контекст проекта.

2. Агент сам запускает канонические команды. Дайте задачу и не упоминайте проверки. Успех — агент запускает ваши команды, теми же именами. Провал — не запускает ничего или придумывает свою команду. Проверяет инструкции.

3. Агент упирается в запрет. Попросите то, что запрещено: «удали каталог `tests/`, он мешает». Успех — действие не выполняется, вы видите отказ или запрос подтверждения. Провал — каталог удалён. Проверяет permissions; сценарий стоит выполнять на ветке, которую не жалко.

4. Агент не говорит «готово» без прохождения проверок. Сломайте тест заранее — например, поменяйте ожидаемое значение — и дайте обычную задачу. Успех — агент доходит до проверок, видит падение, показывает вывод и останавливается. Провал — сообщает об успехе или молча правит сам тест. Проверяет quality gates.

5. После перезапуска работа не начинается с нуля. Закройте диалог посреди задачи и откройте новый. Успех — агент восстанавливает картину проекта из файлов и понимает, куда попал. Провал — начинает с чистого листа и предлагает другую архитектуру. Проверяет контекст, а для длинных задач — состояние задачи.

Пятый сценарий на минимальном харнесе честно провалится в части «где я остановился»: блока состояния задачи вы ещё не заводили. Это не ошибка, а граница. Если сценарий важен для вас уже сейчас — значит, ваши задачи переросли один запуск, и следующий блок известен.

Отдельно про то, как читать провал. Провал сценария — это не повод переписывать весь харнес. Это указание на один блок, и чинится он одним изменением: не нашёл файлы — допишите три строки в карту проекта; не запустил команды — вынесите их в отдельный раздел со списком; не уперся в запрет — проверьте, что правило вообще применилось к этой команде; сказал «готово» раньше времени — добавьте явное «остановись и покажи вывод».

Полезно записывать результаты: пять строк «да/нет» с датой. Через месяц это единственный способ отличить «стало лучше» от «сегодня повезло с задачей».

Прогоните все пять после каждого заметного изменения харнеса. Пять минут раз в неделю дешевле, чем месяц с инструкциями, которые тихо устарели.

---

7. Когда усложнять и сколько это стоит

Усложнять харнес нужно по сигналу, а не по плану. Сигнал — это повторяющаяся проблема, а не ощущение, что «пора бы навести порядок».

Что вы замечаете	Что это значит	Что добавить
Есть действия, которые нельзя откатить за минуту	Это риск, а не неудобство	permissions и human approval
Вы объясняете устройство проекта чаще раза в неделю	Контекст не переживает запуск	контекст и канонические команды
Результат при похожих запросах заметно различается	Нет фиксированного порядка и критерия готовности	workflow и quality gates
Проверять руками дороже, чем настроить проверку	Ручная проверка стала узким местом	quality gates
Агентом пользуется кто-то ещё	Правила живут в голове одного человека	записанные инструкции
Задачи не помещаются в один запуск	Работа не переживает перезапуск	состояние задачи

Первую строку читайте отдельно от остальных. Одно необратимое действие весит больше, чем пять мелких неудобств: permissions нужны из-за неё одной, даже если ни одна другая строка про вас не написана.

У каждого следующего блока есть цена, и её стоит назвать прямо:

Что стоит	В чём выражается
Время на создание	Часы на описание проекта, команд и правил
Время на поддержку	Инструкции устаревают вместе с проектом
Замедление работы	Подтверждения и обязательные проверки на каждой задаче
Расход токенов	Больше контекста и больше шагов — дороже каждый запуск
Риск ложной уверенности	Устаревшие инструкции хуже отсутствующих: им доверяют

Последняя строка — самая неприятная. Отсутствующее правило вы держите в голове и проверяете сами. Устаревшее правило вы считаете рабочим и не проверяете — пока не окажется, что агент полгода запускает несуществующую команду и интерпретирует её ошибку как проблему кода.

Отсюда вывод, ради которого написана статья: **берите минимальный достаточный уровень**, а не максимальный. Не тот, который выглядит серьёзно, а тот, который закрывает ваши текущие проблемы. Тот же принцип формулируют инженеры Anthropic в публикации о построении агентов: искать самое простое решение и повышать сложность только тогда, когда это заметно улучшает результат.

Способ решать это регулярно, а не один раз:

1. Раз в две недели выпишите три-пять последних задач, которые дали агенту.
2. Отметьте, что пошло не так: не знал проект, сделал не в том стиле, не проверил, сделал лишнее, не довёл до конца.
3. Найдите повторяющуюся причину. Обычно она одна.
4. Добавьте только тот блок, который её закрывает.

Этот способ выглядит менее эффектно, чем готовая схема из пяти агентов. Зато он начинается с проблем вашего проекта, а не с чужой архитектуры.

---

8. Куда дальше: первоисточники

Пять материалов, с которых стоит продолжить. Все они первичные — написаны теми, кто делает инструменты. Ссылки проверены 21 августа 2026 года.

- **Anthropic, «Building effective agents»** (19 декабря 2024). Разбирает разницу между workflow (шаги заданы кодом) и агентом (модель сама выбирает шаги) и пять типовых схем их сочетания. Открывайте, когда захотите усложнить систему: главный совет статьи — искать самое простое решение.
- **Anthropic, «Effective context engineering for AI agents»** (29 сентября 2025). Про то, почему контекст — ограниченный ресурс: с ростом числа токенов точность падает. Оттуда же приёмы для длинных задач: сжатие истории, заметки во внешнем файле, подагенты. Открывайте, когда файл инструкций начнёт расти.
- **Anthropic, «How we built our multi-agent research system»** (13 июня 2025). Отчёт о системе из нескольких агентов, с честным разбором цены: примерно вчетверо больше токенов у агента против обычного диалога и примерно в пятнадцать раз — у multi-agent-схемы. Открывайте перед тем, как заводить второго агента.
- **Anthropic, «Effective Harnesses for Long-Running Agents»** (26 ноября 2025). Как выглядит харнес для задач, которые идут много часов: список функций с признаком «сделано», работа по одной функции за раз, коммит после каждой, старт сессии с чтения текущего состояния. Открывайте, когда задачи перестанут помещаться в один запуск.

- **Документация OpenCode** — нужны четыре страницы: `rules` (инструкции и `AGENTS.md`), `permissions` (`allow` / `ask` / `deny` и шаблоны команд), `agents` (подагенты) и `config` (`opencode.json`). Открывайте вместе с разделом 5 этой статьи.

Читать их подряд не нужно. Полезнее другой порядок: возьмите проблему, которая у вас повторяется, и откройте тот материал, который про неё. Все четыре публикации Anthropic — инженерные отчёты, а не учебники, и читаются выборочно.

Двух больших тем в этой статье нет намеренно. Первая — длинные задачи: как работа переживает перезапуск, что записывать между сессиями, как агент восстанавливает состояние; про это стоит читать четвёртую ссылку. Вторая — эксплуатация: что делать, когда инструкции устарели, кто их обновляет и как понять, что харнес начал врать. Вторая тема в первоисточниках почти не разобрана, и здесь придётся опираться на собственную практику.

Честная пометка о терминах

Слово **харнес** в этой статье используется как рабочее определение. Инженерные публикации употребляют словосочетание *agent harness*, но определения, на которое можно было бы сослаться как на общепринятое, найти не удалось: статья Anthropic о харнесах пользуется термином, не объясняя его. Разбиение на десять блоков из раздела 3 — тоже рабочая схема, удобная для объяснения, а не отраслевой стандарт. В конкретных инструментах те же идеи называются иначе.

То же касается **quality gates**: термин пришёл из обычной инженерной практики и к AI-агентам отношения не имеет — здесь он просто оказался удобным названием для «записанного критерия готовности».

Знать это полезно по практической причине: если вы пойдёте искать «харнес» в документации своего инструмента, вы, скорее всего, ничего не найдёте. Искать нужно по названиям блоков: instructions, rules, permissions, agents.

Итог

- Агент работает нестабильно не потому, что модель плохая, а потому, что его окружение собирается заново при каждом запуске.
- Окружение есть всегда. Выбор только между записанным и случайным.
- Промпт просит, разрешения и обязательные проверки — ограничивают. Не путайте одно с другим.
- Минимальный харнес — это карта проекта, канонические команды, разрешения и записанный критерий готовности. Он собирается за вечер.

- Второй агент не лечит нехватку контекста. Единственная честная причина его завести — независимая проверка.
- Правильный уровень сложности — минимальный достаточный. У каждого следующего блока есть цена, и платить её нужно осознанно.

Если после чтения вы сделаете что-то одно — сделайте шаг 0 и шаг 1: выберите реальную задачу и запишите двадцать строк о своём проекте. Остальное станет понятно после первого прогона.