

12+

АНДРЕЙ ЛАДЫГИН

Субпрайм- кризис кода

ДОЛГОСРОЧНАЯ ЦЕНА
ИИ-АССИСТЕНТОВ



Андрей Ладыгин

**Субпрайм-кризис кода.
Долгосрочная цена
ИИ-ассистентов**

«Издательские решения»

Ладыгин А.

Субпрайм-кризис кода. Долгосрочная цена ИИ-ассистентов /
А. Ладыгин — «Издательские решения»,

ИИ ускорил написание кода, но не владение. Разница накапливается как субпрайм-ипотека на архитектуру. Книга для СТО, тимлидов и инженеров: как выстроить review, ownership и архитектурный контроль, чтобы скорость генерации не обернулась катастрофой. Практическое руководство по антихрупкости инженерных команд в эпоху ИИ. 12 принципов, метрики, чеклисты, кейсы.

© Ладыгин А.

© Издательские решения

Содержание

Введение	6
Часть I. Диагноз: почему ИИ-код — это субпрайм	11
Глава 1. Дешёвый код, дорогое владение	11
Глава 2. Анатомия субпрайм-ипотеки на архитектуру	18
Глава 3. Голоса с передовой: DOU, Reddit, HN	25
Глава 4. Пять слоёв скрытого долга	32
Глава 5. Метрики, которые скрывают катастрофу	39
Часть II. Механика отказа: как именно ИИ ломает инженерную систему	47
Глава 6. Чёрные ящики и потеря замысла	47
Глава 7. Парадокс review: генерация быстрее мышления	54
Конец ознакомительного фрагмента.	60

Субпрайм-кризис кода Долгосрочная цена ИИ-ассистентов

Андрей Ладыгин

© Андрей Ладыгин, 2026

ISBN 978-5-0071-4411-7

Создано в интеллектуальной издательской системе Ridero

Введение

«Мы купили скорость. Счёт придёт позже.»
— из разговора с СТО, который пережил rewrite

Зачем эта книга

В 2023—2024 годах индустрия пережила первую волну массового внедрения ИИ-ассистентов. Компании увидели рост velocity, сократили штат, отчитались о «повышении эффективности». Через 6—18 месяцев начали приходить сигналы: инциденты, которые не чинятся «ещё одним патчем», миграции, которые ломают всё, уход инженеров — единственных носителей контекста, rewrites, которые стоят дороже, чем вся сэкономленная зарплата.

Это не история про «ИИ — плохо». Это история про асимметрию: ИИ снизил стоимость написания кода, но не снизил стоимость владения им. Разница между этими двумя величинами накапливается, как проценты по кредиту. Мы называем это субпрайм-ипотекой на архитектуру.

Книга — попытка посчитать этот долг и дать инструменты, чтобы его не брать.

Что произошло на самом деле

Три года назад считалось, что главная проблема разработки — скорость написания кода. Медленно пишем, медленно доставляем, медленно реагируем. ИИ решил эту проблему. Сегодня разработчик с ассистентом генерирует в 3—5 раз больше кода, чем без него (оценка, зависит от контекста и языка).

Но скорость написания — не единственное узкое место. Есть ещё:

Понимание. Код, который никто не понимает, невозможно эволюционировать.

Review. 500 строк сгенерированного кода нужно прочитать, проверить, протестировать.

Ownership. Кто-то должен дежурить, чинить, отвечать.

Организационная память. Контекст решений живёт в головах, а не в модели.

Ответственность. За outage отвечает СТО, а не нейросеть.

ИИ ускорил только первую часть уравнения. Остальные остались прежними — или выросли. Это как если бы вы ускорили выдачу кредитов, но не изменили систему проверки заёмщиков. Больше кредитов — больше плохих кредитов. Больше кода — больше скрытого долга.

Ключевая метафора

Субпрайм-ипотека 2007—2008 работала так:

Кредит выдавали без проверки дохода.

Первые 2—3 года — низкая ставка (teaser rate).

Потом ставка росла.

Заёмщик не мог платить.

Фореклоужер.

С ИИ-кодом то же самое:

Этап Ипотека ИИ-код

Origination Кредит без проверки Промпт без спецификации

Teaser rate Низкий платёж 2—3 года Быстрая генерация, «работает»

Securitization CDO Merge пачками в main

Rating agencies AAA-рейтинги Velocity, LOC, «скорость доставки»

Reset Ставка выросла Инциденты, отладка, миграции

Foreclosure Дом забрали Outage, rewrite, потеря команды

Первые месяцы всё выглядит отлично. Метрики растут, руководство довольно. Потом наступает reset. И foreclosure.

Эта книга — не про то, как запретить ИИ. Она про то, как не купить дешёвый код и не получить дорогую архитектуру.

Для кого эта книга

СТО, VP Engineering, Head of Development — тем, кто принимает решения о внедрении ИИ и сокращениях.

Tech lead, staff/principal engineer, архитекторы — тем, кто отвечает за архитектуру и review.

DevOps/SRE — тем, кто дежурит и чинит последствия.

Security engineers — тем, кто ищет мины в сгенерированном коде.

Senior-разработчикам — тем, кто ревьюит ИИ-код и чувствует review-усталость.

Продуктовым лидерам — тем, кто верит, что «один с ИИ заменит пятерых».

Книга не для тех, кто ищет подтверждение, что «ИИ всё ломает». И не для тех, кто ищет подтверждение, что «ИИ всё починит». Она для тех, кто хочет посчитать реальную цену и выстроить систему, которая выдерживает скорость генерации.

Что вы найдёте внутри

Книга состоит из пяти частей:

Часть I. Диагноз: почему ИИ-код — это субпрайм. Что такое стоимость владения, как она накапливается, где именно возникает долг. Пять слоёв скрытого долга: когнитивный, архитектурный, операционный, организационный, безопасностный.

Часть II. Механика отказа: как именно ИИ ломает инженерную систему. Чёрные ящики, парадокс review, ownership-вакуум, архитектурная эрозия, production-мины, организационная слепота. Как отказ выглядит на практике.

Часть III. Принципы антихрупкости. ИИ как усилитель, а не заменитель ответственности. Контракт до генерации. Владение важнее авторства. Архитектура как бюджет. Review нового поколения. Безопасность по умолчанию. Прозрачность и provenance.

Часть IV. Практики и ритуалы для команд. AI-aware review, верификация, ритуалы (ADR/RFC, debt review, postmortem), on-call, метрики здоровья, политика внедрения, роли и оргдизайн, обучение и культура.

Часть V. Экономика и стратегия. ТСО ИИ-кода, где ИИ реально экономит, сценарии для стартапа/enterprise/аутсорса/regulated, дорожная карта на 90/180/365 дней, будущее агентов и автономных PR, манифест антихрупкой команды.

В конце — приложения: чеклисты, шаблоны, метрики, кейсы, глоссарий.

Как читать эту книгу

Если вы СТО или VP Engineering — начните с части I и части V. Часть I даст диагноз, часть V — экономику и дорожную карту.

Если вы tech lead или архитектор — читайте последовательно. Части II и III — ваши. Часть IV — практика на каждый день.

Если вы SRE или security — начните с глав 10, 17, 22. Это про мины и инциденты.

Если вы разработчик — начните с глав 6, 7, 8. Это про чёрные ящики, review и ownership.

Если вы продуктовый лидер — начните с глав 1, 11, 27. Это про экономику и сокращения.

Книгу можно читать по главам — каждая самодостаточна. Но связи между главами есть: мы ссылаемся на номера, чтобы вы могли вернуться.

Что эта книга не делает

Не обещает серебряных пуль. Нет «одного правильного способа внедрить ИИ». Есть набор практик, которые работают в разных контекстах.

Не запрещает ИИ. ИИ — усилитель. Вопрос в том, что именно он усиливает: вашу инженерную систему или ваш технический долг.

Не морализирует. Мы не говорим, что «ИИ — это плохо» или «ИИ — это хорошо». Мы показываем механику и даём инструменты.

Не выдумывает исследования. Если ссылаемся — только на общеизвестные практики (DORA, SBOM, ADR, RFC, mutation testing). Кейсы — собирательные, помечены. Числа — с пометкой «оценка», где нет источника.

Главная мысль

ИИ-ассистенты — это кредитное плечо. Плечо может ускорить рост, а может уничтожить компанию. Вопрос не в том, использовать ли ИИ, а в том, есть ли у вас инженерная система, которая выдерживает скорость генерации.

Если нет — вы берёте субпрайм-ипотеку на архитектуру. И однажды придёт форекло-ужер: outage, rewrote, потеря команды и рынка.

Книга даёт не страх, а инструменты: review, ownership, архитектурный контроль, метрики и экономику. Тогда ИИ станет усилителем, а не долговой пирамидой.

Как пользоваться книгой

Каждая глава построена одинаково:

Эпиграф — задаёт тон.

Тезис — что доказываем.

Ключевые вопросы — на что отвечает глава.

Основной текст — разделы по подглавам.

Практика — что сделать завтра.

Метрики — что мерить.

Кейс из trenches — история, которую легко узнать.

Красные флаги — цитаты, которые вы слышали.

Антипаттерн — как делать не надо.

Чеклист — 5—10 пунктов.

Что запомнить — что унести.

Что почитать — 5 источников.

Вы можете читать главы последовательно или выборочно. Но если вы СТО — рекомендуемую последовательно: логика книги выстроена как диагностика, потом лечение.

От автора

Я пишу эту книгу как инженер, который видел и хорошие, и плохие внедрения ИИ. Я не против ассистентов — я против слепой веры в них. Я не против скорости — я против скорости без ответственности.

Если после прочтения вы:

посчитаете стоимость владения кодом в своей команде;

введёте owner-а для каждого PR;

начнёте мерить не только velocity, но и defect escape rate;

перестанете сокращать команду на основе роста генерации;

введёте review для ИИ-кода;

— книга сделала своё дело.

Начинаем.

Часть I. Диагноз: почему ИИ-код — это субпрайм

Глава 1. Дешёвый код, дорогое владение

«Код — как ребёнок. Сделать легко. Воспитывать — всю жизнь.»
— из разговора двух тимлидов после трёхдневного инцидента

Тезис

ИИ снизил стоимость написания кода, но не стоимость владения им. Разница между этими двумя величинами — и есть субпрайм-ипотека на архитектуру. Владение = понимание + эксплуатация + эволюция. Пока эти три компонента не учтены, экономия от ИИ — это отсроченный платёж с процентами.

Ключевые вопросы

Почему «код работает» не равно «код готов к эксплуатации»?

Что такое владение кодом и почему ИИ его не отменяет?

Где именно возникает скрытая цена ИИ-кода?

Почему генерация ускорила, а ответственность — нет?

Как посчитать реальную стоимость одной строки кода?

1.1. Код как обязательство, а не актив

В бухгалтерии код иногда записывают в активы. В инженерии — это обязательство. Каждая строка, попавшая в `main`, требует: понимания, сопровождения, тестирования, обновления зависимостей, миграций, обработки инцидентов. Пока код жив — он генерирует работу. Чем больше кода — тем больше работы.

Код — как кот. Пока жив — жрёт, гадит и требует внимания. Только кот хотя бы мурлычет. Код не мурлычет никогда.

ИИ ускорил создание обязательств. Раньше разработчик писал 50—100 строк в день и помнил каждую. Теперь он генерирует 1500 строк до обеда и не помнит ни одной. Прогресс? Смотря для кого. Для его резюме — да. Для on-call-а через полгода — нет.

Вы взяли кредит, чтобы купить квартиру. Квартира — актив. Кредит — обязательство. Если вы берёте кредиты быстрее, чем растёт стоимость квартиры, вы не богатеете — вы тоните. С кодом то же самое: если вы генерируете код быстрее, чем команда способна его освоить, вы не ускоряетесь — вы берёте субпрайм-ипотеку.

Код — это не то, что вы создали. Это то, что вы обязаны поддерживать. Пока не посчитана стоимость поддержки — стоимость создания бессмысленна.

1.2. Что такое владение: три ночи из жизни on-call-а

Есть три сцены, которые объясняют владение лучше любого определения.

Сцена первая: понимание.

3:47 ночи. Звонит алерт. On-call открывает модуль биллинга. 800 строк. Сгенерированы ИИ восемь месяцев назад. Автор промпта уволен в рамках оптимизации. Комментариев нет. Тестов нет. Есть только код, который «работал». On-call смотрит на функцию `calculate_discount()` и не понимает, почему скидка применяется дважды. Он не может починить то, что не понимает.

Это отсутствие понимания. Без него владения нет.

Сцена вторая: эксплуатация.

Та же ночь. Через четыре часа. Инцидент не закрыт. On-call пишет в чат: «Кто-нибудь знает, что делает этот модуль?» Тишина. Он дежурит, но не владеет. Он отвечает за код, который никогда не писал и не понимает.

Это отсутствие эксплуатации в руках того, кто отвечает. Владение — это не только «понимаю», но и «готов чинить в 4 утра».

Сцена третья: эволюция.

Через месяц приходит требование: изменить логику скидок. Архитектор открывает модуль. Понимает с трудом. Начинает рефакторить — и ломает три соседних сервиса, потому что сгенерированный код использовал неочевидные side effects. Никто не знал, что они там есть.

Это отсутствие эволюции. Код, который нельзя безопасно изменить, — это не актив. Это мина с часовым механизмом.

Владение = понимание + эксплуатация + эволюция. Уберите любой компонент — и вы получите субпрайм. Платёж вносится, но дом вам не принадлежит.

ИИ не закрывает ни один из трёх компонентов. Модель не объясняет замысел. Модель не дежурит. Модель не знает, почему вчера было решено именно так. Она даёт результат — и уходит. Владение остаётся на команде. Или не остаётся ни на ком.

1.3. Три стоимости: где именно прячется долг

Компания посчитала: фича стоила два часа генерации. Дёшево. Через год та же фича стоила 60 часов инцидентов, отладки и миграций. Разница — это Own. Складываем — получаем настоящую цену.

Стоимость кода не одна. Их три.

Стоимость Что входит Кто платит Когда платит

Написание (Write) Промпт, генерация, правки, merge Разработчик Сейчас

Review Чтение, проверка, тесты, обсуждение Ревьюер, команда Сейчас

Владение (Own) Понимание, отладка, миграции, инциденты, удаление Команда, on-call, будущие инженеры Через 3—24 месяца

ИИ резко снизил Write. Review остался прежним или вырос: ревьюить 500 строк сложнее, чем 100. Own — вырос кратно, потому что сгенерированный код часто не имеет автора, который понимает замысел.

text

Total Cost of Code = Write + Review + Own

Own включает:

время на отладку (debug time);

время на инциденты (incident time × cost of downtime);

время на миграции и рефакторинг;

стоимость rewrites;

стоимость потери контекста при уходе инженера.

Числовой пример (оценка):

1000 строк сгенерированного кода.

Write: 2 часа.

Review: 4 часа.

Own (за 12 месяцев): 20—60 часов на отладку, инциденты, миграции.

Итого: 26—66 часов.

Для сравнения: 1000 строк, написанных человеком с пониманием: Write 20 часов, Review 4 часа, Own 10—30 часов. Итого: 34—54 часа.

Разница не в пользу ИИ, если код не понят командой. Если понят — ИИ выигрывает. Если нет — проигрывает. Третьего нет. Ключевое слово — «понят». Это и есть владение.

1.4. Почему генерация ускорилась, а ответственность — нет

Генерация — это функция от промпта. Ответственность — это функция от понимания, ownership и организационной структуры. ИИ ускорил первую. Вторую он не трогал.

Вот что не изменилось.

Понимание. Чтобы отвечать за код, его нужно понимать. ИИ не даёт понимания. Он даёт результат. Разница — как между «я знаю, что это работает» и «я знаю, почему это работает». Первое ломается при первом изменении. Второе — живёт.

Ownership. Кто-то должен дежурить, чинить, отвечать. ИИ не дежурит. Модель не просыпается в 3:47. Модель не пишет постмортем. Модель не объясняет бизнесу, почему упал биллинг.

Организационная память. Контекст решений живёт в головах. ИИ не помнит, почему вы выбрали именно это решение. Он не был на том созвоне, где вы спорили два часа и выбрали вариант В. Через год вариант В станет «странным кодом, который никто не понимает».

Ответственность перед бизнесом. За outage отвечает СТО, а не модель. За потерянные деньги отвечает компания, а не API-провайдер. За увольнение команды отвечает руководитель, а не нейросеть.

ИИ ускорил только одну часть уравнения. Это как если бы вы ускорили подачу заявок на кредиты, но не изменили систему проверки заёмщиков. Больше заявок — больше плохих кредитов. Больше кода — больше скрытого долга.

Скорость генерации — не узкое место. Узкое место — способность команды осваивать и владеть кодом. ИИ это узкое место не расширяет. Он его маскирует.

1.5. Почему метрики velocity это скрывают

Velocity измеряет, сколько команда выдала. Она не измеряет, сколько команда сможет поддерживать. Это как измерять качество кредита по количеству выданных кредитов. Пока рынок растёт — всё хорошо. Когда рынок падает — выясняется, что половина кредитов — мусор.

Что измеряет velocity:

количество закрытых задач;

количество PR;

story points;

время до merge.

Что она не измеряет:

стоимость владения;

частоту инцидентов;

rework rate;

orphan code ratio;

время до первого фикса;

стоимость outage.

ИИ усиливает этот эффект: генерация растёт, velocity растёт, а скрытые метрики — нет. Компания принимает решения по метрикам, которые не видят риска.

Метрика без стоимости владения — это рейтинговое агентство субпрайма. Оно показывает рост, пока не наступает фореклоужер.

Практика: что сделать завтра

Посчитайте стоимость одной строки кода. Возьмите 1000 строк сгенерированного кода. Оцените Write, Review, Own за 6 и 12 месяцев. Сравните с кодом, написанным человеком.

Введите метрику «cost of ownership» в дашборд. Она не обязана быть точной. Она должна быть видимой.

Проведите аудит: сколько кода в main не имеет owner-а? Используйте CODEOWNERS или git blame. Если git blame показывает «автор — ИИ, owner — NULL» — это orphan code.

Задайте вопрос команде: «Что ты не сможешь починить, если автор уйдёт?» Ответы — это карта рисков.

Введите правило: «нет owner-а — нет merge». Даже для ИИ-кода.

Проверьте три компонента владения для критичных модулей. Есть ли понимание? Есть ли эксплуатация? Есть ли план эволюции? Если хотя бы одного нет — модуль в зоне риска.

Метрики

Метрика Формула Что показывает

Cost per line (write) Время на написание / LOC Стоимость генерации

Cost per line (own) Время на владение / LOC Скрытая стоимость

Rework rate Переписанные строки / всего строк Доля кода, который не работает

Defect escape rate Баги в проде / всего багов Качество review

Orphan code ratio Код без owner-а / всего кода Риск владения

Формула субпрайм-процента (оценка):

text

$$\text{Subprime Rate} = (\text{Own} - \text{Write}) / \text{Write} \times 100\%$$

Если Own больше Write — вы платите проценты. Если Own в 5 раз больше Write — foreclosure не за горами.

Кейс из trenches

Собирательный кейс. Детали изменены.

Стартап, 12 разработчиков, B2B SaaS, стек: Python + React + PostgreSQL. Внедрили ИИ-ассистента в январе. К марту генерировали в 3 раза больше фич. Руководство довольно, velocity выросла на 180%.

В апреле сократили 4 разработчиков: «ИИ справляется».

В июне, в 2:47 ночи, on-call получил алерт от биллинга. Он открыл сгенерированный модуль. 800 строк. Ни комментариев, ни тестов, ни автора. Промпт писал уволенный разработчик. Через 4 дня он всё ещё не понимал, почему скидка считалась дважды. Через 5 дней уволился.

Потеряли \$40—80 тыс. (оценка) на инциденте и ещё больше — на найме замены.

В августе — второй инцидент: утечка секретов через сгенерированный конфиг. В сентябре — третий: миграция БД, сгенерированная ИИ, удалила часть данных. Восстановление — 2 недели.

К декабрю: velocity упала на 60%, команда — 6 человек (из 12), СТО уволен, проект переписывают. Стоимость rewrite — \$200—400 тыс. (оценка). Экономия на зарплатах — \$150—200 тыс. (оценка). Чистый убыток — \$250—500 тыс.

Что сделали не так: не считали стоимость владения, не вводили ownership, не проверяли числа.

Красные флаги

«Мы просто генерируем и мержим, review потом.»

«ИИ справляется, зачем нам 5 разработчиков?»

«У нас velocity выросла в 2 раза.»

«Это ИИ сгенерировал, я не знаю, как оно работает.»

«Разберёмся, когда сломается.»

«У нас всё под контролем.»

«Мы потом отрефакторим.»

Антипаттерн

Как делать НЕ надо:

Мержить ИИ-код без owner-a.

Считать velocity единственной метрикой здоровья.

Сокращать команду на основе роста генерации.

Не считать стоимость владения.

Не вводить review для ИИ-кода.

Хранить контекст только в головах уволенных.

Верить, что «ещё один патч» решит проблему.

Чеклист

- ☐ Посчитана стоимость Write, Review, Own для 1000 строк кода.
- ☐ Введена метрика cost of ownership.
- ☐ Проведён аудит orphan code.

- ☐ Каждый PR имеет owner-а.
- ☐ Команда знает, что не сможет починить без автора.
- ☐ Velocity не единственная метрика.
- ☐ Review ИИ-кода не формальность.
- ☐ Сокращения не основаны на росте генерации.
- ☐ Есть план на случай ухода ключевых инженеров.
- ☐ Стоимость владения видна руководству.

Что запомнить

Скорость генерации — не метрика здоровья. Метрика — стоимость владения. Пока она не посчитана, вы не экономите — вы берёте кредит под высокий процент.

Что почитать

DORA State of DevOps Report — метрики доставки и их ограничения.

«Accelerate» (Forsgren, Humble, Kim) — про то, какие метрики реально предсказывают здоровье команды.

«Working Effectively with Legacy Code» (Michael Feathers) — про владение кодом без автора.

Google SRE Book, глава про postmortem — как разбирать инциденты без поиска виноватых.

«The Mythical Man-Month» (Fred Brooks) — про цену отсрочки и иллюзию скорости.

Глава 2. Анатомия субпрайм-ипотеки на архитектуру

«Мы не брали кредит. Мы просто генерировали код. Разница — как между „не брал“ и „не помню, как брал“.»

— из разбора инцидента в аутсорс-компании

Тезис

Финансовая аналогия работает не как метафора, а как точная модель: origination → securitization → leverage → rating → foreclosure. Каждый этап можно найти в вашем репозитории. Если вы не видите этап — вы просто стоите в его середине. Foreclosure всегда приходит. Вопрос — к кому.

Ключевые вопросы

Где в разработке origination, securitization и foreclosure?

Почему rating agencies (метрики) врут?

Кто платит по счетам, когда субпрайм срывает?

Как провести стресс-тест архитектуры до того, как придёт foreclosure?

Почему «у нас всё под контролем» — это красный флаг, а не успокоение?

2.1. Origination: промпт как выдача кредита без проверки дохода

В 2007 году кредит выдавали так: заёмщик приходил, говорил «у меня есть доход», ему верили. Никаких документов. Никакой проверки. Кредит выдан. Дом куплен.

В разработке origination выглядит так. Разработчик открывает чат с ИИ, пишет: «Сделай мне модуль биллинга с поддержкой скидок». Модель генерирует 800 строк. Разработчик смотрит: «Выглядит нормально». Мержит. Никто не спросил: куда это встраивается? Какие ограничения? Кто owner? Какие критерии приёмки? Промпт выдан — кредит выдан.

Проблема не в том, что промпт плохой. Проблема в том, что никто не проверил заёмщика. Заёмщик в данном случае — код. Проверка — это спецификация, архитектурный review, понимание ограничений. Если их нет — вы выдаёте кредит без проверки дохода.

Бывает хуже. Промпт пишется не разработчиком, а продактом. Или тимлидом в спешке. Или джуном, который не знает контекста. Модель не задаёт вопросов. Она генерирует. А тот, кто мержит, часто не тот, кто понимает.

Origination без проверки — это не ошибка одного разработчика. Это системная практика, которая выглядит как скорость. Пока не наступает reset.

2.2. Securitization: merge-пачки как CDO

В ипотечном кризисе банки не держали кредиты у себя. Они упаковывали их в CDO (collateralized debt obligations) и продавали. Чем больше кредитов — тем больше CDO. Чем быстрее продали — тем меньше риска.

В разработке securitization выглядит так. Разработчик генерирует 10 PR за день. Ревьюер смотрит каждый по 5 минут. Мержит. К вечеру в main — 3000 строк нового кода. Никто не держит этот код «у себя» — он уже в main. Риск передан дальше.

Чем больше PR — тем меньше времени на каждый. Чем меньше времени — тем выше вероятность, что что-то пропустили. Это не халатность. Это математика: если у вас 10 PR и 8 часов, у вас 48 минут на PR. Включая чтение, понимание, проверку тестов, обсуждение.

Вот как это выглядит в жизни. Пятница, 18:40. Ревьюер открывает четырнадцатый PR за день. 600 строк. Он листает. Функции выглядят знакомо. Тесты зелёные. Он пишет «LGTM» и закрывает ноутбук. В понедельник он не вспомнит этот PR. В следующем квартале — тем более. А код останется.

Securitization — это не злой умысел. Это следствие скорости генерации. ИИ генерирует быстрее, чем команда способна ревьюить. Merge-пачки — это способ не остановить поток. Но именно они превращают отдельные кредиты в CDO.

Вы не помните, что мержили вчера. Вы не помните, что мержили утром. Вы помните только, что «закрыли 15 задач». Это и есть securitization.

2.3. Leverage: техдолг как кредитное плечо

Команда взяла техдолг, чтобы успеть к релизу. Через полгода техдолг взял команду. Вот это leverage.

Кредитное плечо — это когда вы используете заёмные деньги, чтобы увеличить доходность. Работает, пока рынок растёт. Когда падает — вы теряете больше, чем вложили.

Техдолг — это то же самое. Вы берёте «кредит» в виде быстрых решений: генерируете код без тестов, без документации, без понимания. Это позволяет быстрее доставлять фиши. Пока фиши нужны — плечо работает. Когда приходит инцидент или требование изменить логику — плечо срывает против вас.

Виды техдолга, которые создаёт ИИ-код:

Тип долга Как возникает Когда срабатывает

Когнитивный Никто не понимает замысел При первом изменении

Архитектурный Локальные оптимумы, дублирование При масштабировании

Операционный Нет тестов, нет логирования При инциденте

Организационный Нет owner-а При уходе автора

Безопасностный Секреты в коде, уязвимые зависимости При аудите или атаке

Каждый тип — это плечо. Чем больше плечо, тем выше доходность в хорошие времена и тем глубже яма в плохие.

Какое у вас плечо? Если вы не можете ответить — вы в зоне субпрайма. Если можете, но число растёт — вы в зоне субпрайма, просто пока этого не видно.

2.4. Rating agencies: velocity, LOC, «скорость доставки»

СТО смотрит на дашборд. Velocity — зелёная. Инциденты — красные. Он смотрит на velocity. Потому что её видно. Инциденты он увидит через полгода. Вот это rating agencies.

В 2008 году рейтинговые агентства ставили AAA на CDO, которые через год становились мусором. Почему? Потому что модель оценки не учитывала риск. Она учитывала только доходность.

В разработке рейтинговые агентства — это ваши метрики. Velocity, LOC, story points, «скорость доставки». Они показывают рост. Они не показывают риск.

Что измеряет velocity:

сколько задач закрыто;

сколько PR смирено;

сколько story points сожжено.

Что она не измеряет:

сколько кода никто не понимает;

сколько инцидентов будет через 6 месяцев;

сколько времени уйдёт на отладку;

сколько будет стоить rewrite;

сколько инженеров уволится.

ИИ усиливает этот эффект: генерация растёт, velocity растёт, рейтинг AAA. Пока не наступает reset.

Если в вашем дашборде есть velocity, но нет cost of ownership — вы смотрите на rating agencies. Если в отчёте для руководства есть «скорость доставки», но нет «частоты инцидентов» — вы смотрите на rating agencies. Если в ретроспективе обсуждают «как ускориться», а не «как снизить риск» — вы смотрите на rating agencies.

2.5. Foreclosure: outage, rewrite, потеря команды

Сначала алерт. Потом on-call не понимает код. Потом звонит автору. Автор уволен. Потом rewrite. Потом уход команды. Это не три сценария. Это один.

Первый алерт — в пятницу, 21:14. Биллинг считает скидки дважды. On-call открывает модуль. 800 строк. Сгенерировано ИИ восемь месяцев назад. Автор промпта уволен в рамках оптимизации. Комментариев нет. Тестов нет. On-call пишет в чат: «Кто-нибудь знает, что делает этот модуль?» Тишина.

В субботу он всё ещё не понимает. В воскресенье СТО открывает Slack и пишет: «Кто-нибудь понимает, что делает модуль биллинга?» Тишина. В понедельник СТО увольняется.

Через месяц команда решает: переписать. Переписывание стоит дороже, чем всё, что сэкономили на генерации. Плюс время. Плюс нервы. Плюс уход людей, которые устали чинить то, что не понимают.

Через полгода инженеры, которые были единственными носителями контекста, уходят. Не потому что плохие. Потому что устали. Компания остаётся с кодом и без людей.

Foreclosure не приходит внезапно. Он зреет месяцами. Сначала — «ещё один патч». Потом — «разберёмся позже». Потом — «почему это не работает?». Потом — «кто это писал?». Потом — «никто не знает».

Foreclosure — это не событие. Это процесс. И он уже начался, если вы не можете ответить на вопрос «кто владеет этим кодом?».

2.6. Кто платит: следующая команда, следующая компания, следующий инженер

В ипотечном кризисе платили не банки, которые выдали кредиты. Платили заёмщики, которые потеряли дома. Платили налогоплательщики, которые спасали банки. Платили следующие поколения.

В разработке платят тоже не те, кто генерировал.

Платят клиенты, которые ушли после третьего outage. Платят инженеры, которые уволились, потому что устали тушить пожары. Платят те, кто остался: они работают за троих и не могут найти замену, потому что рынок знает репутацию компании.

Платят следующая команда — те, кто придёт поддерживать код. Платят следующая компания — если проект продан или передан. Платят следующий инженер — тот, кто откроет файл и скажет «что это вообще?». Платят бизнес — через outage, потерю клиентов, репутацию. Платят следующий СТО — который будет объяснять, почему `gewrite` стоит миллионы.

Тот, кто генерировал код, часто уже не платит. Он ушёл. Он получил оффер. Он в другой компании, где всё «по-другому». А долг остался.

Вопрос не в том, кто виноват. Виноватых не будет. Вопрос в том, кто платит. Если ответ «следующая команда» — вы в зоне субпрайма. Если ответ «мы» — вы уже платите. Если ответ «не знаю» — foreclosure уже в процессе.

Практика: что сделать завтра

Проведите стресс-тест архитектуры. Что будет, если завтра уйдёт вся команда, писавшая ИИ-код? Кто сможет поддерживать систему? Сколько времени уйдёт на восстановление контекста?

Посчитайте `bus factor` для критичных модулей. Сколько людей понимают этот код? Если один — `bus factor` = 1. Это риск. Если ноль — foreclosure уже близко.

Проверьте, есть ли в вашем дашборде `cost of ownership`. Если нет — добавьте. Если есть — посмотрите динамику за 6 месяцев.

Проведите аудит merge-пачек. Сколько PR мержится в день? Сколько времени уходит на ревью каждого? Если меньше 30 минут на PR — вы в зоне securitization.

Задайте вопрос: «Кто платит за этот код?» Если ответ «следующая команда» — начните с ownership.

Введите метрику debt interest. Команда тратит 40% времени на поддержку кода, который никто не понимает? Это и есть проценты по субпрайму.

Метрики

Метрика Формула Что показывает

Orphan code ratio Код без owner-а / всего кода Риск владения

Bus factor Мин. число людей, знающих модуль Хрупкость команды

Debt interest Время на поддержку / общее время Проценты по субпрайму

Merge latency Время от PR до merge Скорость securitization

Review depth Время на ревью / LOC Качество проверки

Формула debt interest (оценка):

text

$\text{Debt Interest} = (\text{Время на поддержку ИИ-кода} / \text{Общее время команды}) \times 100\%$

Если debt interest > 30% — вы платите больше, чем зарабатываете. Если > 50% — foreclosure близко.

Кейс из trenches

Собирательный кейс. Детали изменены.

Аутсорс-компания, 40 разработчиков, fixed-price контракт с крупным заказчиком. Стек: Java + Spring + Oracle. Проект — система документооборота для банка. Сроки — 9 месяцев. Бюджет — фиксированный.

Чтобы уложиться, команда активно использовала ИИ-ассистента. К концу срока сгенерировали ~70% кода. Сдали проект. Заказчик принял. Аутсорс получил оплату. Все довольны.

Через год заказчик пришёл с требованием: «Нужно изменить логику согласования». Команда аутсорса к тому времени распалась — проект закончен, люди ушли на другие. Новые разработчики открыли код. 500 000 строк. Без документации. Без тестов. Без owner-а.

Новый разработчик открыл DocumentApprovalService.java. 12 000 строк. Ни одного комментария. Ни одного теста. В git log — один коммит: «initial commit from AI assistant». Он закрыл файл. Открыл резюме.

Через месяц попыток изменить логику — каскад ошибок. Сгенерированный код использовал неочевидные side effects, которые никто не задокументировал. Три месяца попыток. Два rewrite. Заказчик подал в суд. Аутсорс потерял контракт и репутацию.

Стоимость поддержки за год превысила стоимость разработки в 2,5 раза (оценка). Кто заплатил? Аутсорс — деньгами и репутацией. Заказчик — временем и нервами. Следующая команда — тем, что вошла в проект, который «уже работает».

Красные флаги

«У нас всё под контролем.»

«Мы потом отрефакторим.»

«Это ИИ сгенерировал, но работает же.»

«Кто это писал? Не помню.»

«У нас velocity выросла, значит, всё хорошо.»

«Зачем нам owner? Оно же работает.»

«Следующая команда разберётся.»

Антипаттерн

Как делать НЕ надо:

Мержить PR пачками без глубокого ревью.

Считать velocity единственной метрикой.

Не считать debt interest.

Не вводить bus factor как метрику риска.

Верить, что «работает» = «можно поддерживать».

Оставлять код без owner-а после ухода автора.

Делать rewrite, не разобравшись, почему первый вариант сломался.

Чеклист

- ☐ Проведён стресс-тест: что будет, если команда уйдёт завтра?
- ☐ Посчитан bus factor для критичных модулей.
- ☐ Введена метрика debt interest.
- ☐ Проверен merge latency: не слишком ли быстро мержим?
- ☐ Есть owner для каждого критичного модуля.
- ☐ Orphan code ratio известен и отслеживается.
- ☐ Есть план на случай foreclosure.
- ☐ Руководство знает, кто платит за субпрайм.
- ☐ Merge-пачки не заменяют review.
- ☐ Rewrite не начинается без анализа причин.

Что запомнить

Foreclosure уже начался. Вы просто ещё не получили уведомление.

Что почитать

«The Big Short» (Michael Lewis) — как финансовые модели могут быть формально верными и катастрофически ошибочными. Прямая аналогия с вашими метриками.

«Release It!» (Michael Nygard) — про production-мины и стоимость отказов.

«Team Topologies» (Skelton, Pais) — про ownership и границы команд.

DORA State of DevOps Report — метрики, которые реально предсказывают риск.

«Working Effectively with Legacy Code» (Michael Feathers) — как разбирать код без автора.

Глава 3. Голоса с передовой: DOU, Reddit, HN

«ИИ сгенерировал 500 строк. Я сгенерировал резюме.»

— из треда на Reddit про review-усталость

Тезис

Проблема не в отдельных историях. Проблема в повторяющихся паттернах. Если десять команд в разных компаниях пишут одно и то же — это не случайность, а свойство системы. Собрать паттерны — значит увидеть риск до того, как он станет инцидентом.

Ключевые вопросы

Что реально пишут инженеры, а не маркетинг?

Какие паттерны повторяются в сотнях команд?

Где хайп расходится с реальностью?

Что говорят не только жертвы, но и выжившие?

Как провести диагностику своей команды по этим паттернам?

3.1. Как читать форумы и не утонуть в anecdotes

Пятница, 23:40. Вы листаете Reddit. Тред «AI assistants ruined my codebase». 400 комментариев. Половина — «same here». Половина — «skill issue». Вы закрываете ноутбук и не понимаете, что делать.

Это не диагностика. Это шум. Один комментарий — это человек, у которого болит. Десять одинаковых в разных тредах — это система, которая болит. Верьте системе.

Как отличить сигнал от шума:

Что видите Что это значит Что делать

Один комментарий Anecdote Игнорировать

5—10 похожих в одном треде Локальный паттерн Запомнить

10+ в разных тредах за месяц Системный паттерн Проверить у себя

Паттерн + кейс с деталями Сигнал Включить в диагностику

Паттерн + цифры + повтор Диагноз Действовать

Если паттерн всплывает на DOU, Reddit и HN одновременно — это не хейт. Это реальность, которую кто-то не хочет замечать.

3.2. Паттерн 1: review-усталость

Среда, 17:20. Ревьюер открывает двенадцатый PR за день. 700 строк. Сгенерировано ИИ. Он листает. Функции выглядят знакомо. Тесты зелёные. Он пишет «LGTM» и идёт домой. В понедельник он не вспомнит этот PR. Через месяц — тем более.

Это не лень. У ревьюера 32 минуты на PR. Включая чтение. Включая понимание. Включая обсуждение. 32 минуты. Дальше — арифметика: если PR 700 строк, вы не ревьюите. Вы имитируете ревью.

В треде на DOU один тимлид написал: «Я больше не читаю PR. Я их просматриваю». Через два комментария другой добавил: «Раньше ревью занимало 20 минут. Теперь — 5, потому что иначе не успеваю». Третий ответил: «Мне кажется, я пропускаю баги, но у меня нет времени их искать». Это не три истории. Это одна система.

Что происходит дальше: баги уходят в прод. Инциденты растут. Команда удивляется: «Как мы это пропустили?» Пропустили, потому что не читали.

3.3. Паттерн 2: галлюцинированные зависимости

Разработчик пишет промпт: «Добавь библиотеку для парсинга PDF». ИИ генерирует код. Импорт: `from pdf_magic_parser import PDFWizard`. Разработчик мержит. CI падает. Библиотеки не существует. Модель её выдумала.

На Hacker News был тред, где инженер описал, как нашёл в проекте зависимость, которая существовала только в галлюцинациях модели. Документация выглядела правдоподобно. API — логично. Пакета не было. Он написал: «Теперь мы проверяем каждый импорт вручную». Второй добавил: «ИИ уверенно использует API, которого нет. И документация выглядит правдоподобно».

Галлюцинированные зависимости — это не баг. Это свойство модели. Она генерирует правдоподобный код, а не работающий. Разница — как между «выглядит как самолёт» и «летает».

Опасность не в том, что CI упадёт. Если CI упал — вы уже выиграли. Вы узнали о проблеме до прода. Настоящая опасность — если галлюцинированная зависимость существует, но делает не то, что вы думаете. Тогда CI зелёный. Прод красный. И никто не понимает, откуда инцидент.

3.4. Паттерн 3: код, который никто не понимает

Понедельник, 10:15. Архитектор открывает модуль, который нужно изменить. 1200 строк. Сгенерировано полгода назад. Автор промпта ушёл. `git blame` показывает: «AI assistant, initial commit». Архитектор читает функцию за функцией. Понимает процентов на тридцать. Остальное — «работает, но непонятно как».

Он спросил автора — тот ещё отвечал на сообщения в Telegram. «Почему здесь два индекса?» Автор ответил: «ИИ так предложил». Архитектор кивнул. Через месяц автор удалил аккаунт. Архитектор остался с двумя индексами и без ответа.

На Reddit это называют «археологией». Каждый рефакторинг — раскопки. Один разработчик написал: «Я боюсь его трогать. Он работает. Пока я не трогаю». Другой добавил: «У нас есть код, который никто не может объяснить. Но он в проде». Третий — самое честное: «Код работает. Но никто не знает, почему. И это страшнее, чем баг».

Код без замысла — это чёрный ящик. Он может работать годами. А потом одно изменение — и он рассыпается, потому что никто не знал, на чём он держится. Это не технический долг. Это когнитивный долг. Вы платите не за то, что код плохой. Вы платите за то, что никто не понимает, почему он хороший.

3.5. Паттерн 4: «ИИ сгенерировал — ИИ и чини»

Вторник, 14:30. Прод упал. Разработчик открывает чат с ИИ. Копирует ошибку. Пишет: «Почини». ИИ генерирует патч. Разработчик мержит. Через час — новый инцидент. Он снова копирует ошибку. Снова патч. Снова мерж.

Это как лечить перелом пластырем. Пластырь держится. Пока не надо бегать. Через месяц вы бежите. И узнаете, что кость срослась неправильно.

На DOU один разработчик описал это так: «Мы чиним сгенерированный код сгенерированными патчами. Это рекурсия». Другой добавил: «ИИ сгенерировал баг, ИИ сгенерировал фикс, ИИ сгенерировал новый баг». Третий — самое точное: «Я чувствую себя оператором, а не инженером».

Патч поверх патча — это не отладка. Это имитация. Вы не лечите причину. Вы закрашиваете симптом. Через месяц у вас десять патчей, которые взаимодействуют непредсказуемо. Опасность в том, что это работает. До поры. Пока не перестанет.

3.6. Паттерн 5: сокращения и потеря экспертизы

Пятница, 16:00. СТО объявляет: «ИИ справляется. Мы сокращаем четырёх из пяти разработчиков». Оставшийся кивает. Через месяц он дежурит 24/7. Через два — увольняется. Через три — компания ищет замену. Найти не может: репутация уже известна.

На Reddit был тред от разработчика, который остался один. Он написал: «Нас было пятеро. Остался один. Я не справляюсь». Через месяц он добавил: «Уволили тех, кто понимал систему. Оставили тех, кто генерирует». Ещё через месяц: «Через полгода никто не помнит, как это работает».

Сокращения под ИИ — это не экономия. Это ставка. Ставка на то, что генерация заменит экспертизу. Экспертиза — это не написание кода. Это понимание, почему он такой. Генерация этого не даёт.

Что происходит через год: команда меньше, инцидентов больше, контекста нет, нанять некого. Экономия на зарплатах — минус. Стоимость восстановления — плюс. Чистый результат — минус. И никто не считал.

3.7. Что говорят не только жертвы, но и выжившие

Форумы — это не только истории провалов. Есть команды, которые прошли через это и остались. Что они делают иначе.

Одна команда в Берлине ввела правило: «Если PR больше 300 строк — он автоматически отправляется на второй review». Через месяц defect escape rate упал на 40% (оценка). Никто не уволился. Velocity упала на 15%. Это оказалось выгодно.

Другая команда, распределённая, ввела «owner дня»: каждый день один человек отвечает за все ИИ-PR. Он не ревьюит всё, но он — точка входа. Если что-то непонятно — идут к нему. Через два месяца review latency выросла, но rework rate упал в три раза (оценка).

Третья команда, в enterprise, ввела «AI Usage Policy»: ИИ разрешён для прототипов, тестов и boilerplate. Запрещён для критичной логики, безопасности и архитектуры. Через полгода — ни одного инцидента, связанного с ИИ-кодом.

Что говорят эти команды:

«ИИ — усилитель. Но усилитель чего? Если у вас хаос, ИИ усилит хаос.»

«Мы не запрещаем ИИ. Мы запрещаем мерзить без понимания.»

«Мы перестали мерить velocity. Начали мерить defect escape rate.»

«Мы не сократили никого. Мы изменили роли.»

«ИИ не заменил инженеров. Он заменил рутину.»

Эти голоса тише. Их меньше. Но они есть. И они дают модель, а не только диагноз.

Практика: что сделать завтра

Проведите анонимный опрос в команде. «Что тебя пугает в ИИ-коде?» «Что ты не сможешь починить?» «Сколько времени у тебя на ревью?» Ответы — карта рисков.

Проверьте долю PR, которые никто не может объяснить. Возьмите 10 случайных PR за последний месяц. Спросите авторов: «Что делает этот код?» Если двое не могут ответить — у вас 20%. Если пятеро — 50%. Это unexplained PR ratio.

Посчитайте review latency. Сколько минут уходит на PR? Если меньше 30 — вы в зоне review-усталости.

Проверьте зависимости. Запустите prn audit или rip check. Если найдёте пакеты, которых нет в реестре, — это галлюцинации. Считайте долю.

Проверьте git log критичных модулей. Если там цепочка «fix», «hotfix», «fix again», «revert fix» — это patch-over-patch. Считайте долю.

Проверьте post-layoff MTTR. До сокращений инцидент закрывался за час? После — за шесть? Это и есть post-layoff MTTR. Считайте.

Метрики

Метрика Формула Что показывает

Review latency Время от PR до merge Скорость securitization

Unexplainable PR ratio PR без объяснения / всего PR Паттерн 3

Hallucinated dependency rate Найденные галлюцинации / всего зависимостей Паттерн 2

Patch-over-patch rate Патчи поверх патчей / всего фиксов Паттерн 4

Post-layoff MTTR MTTR после сокращений / MTTR до Паттерн 5

Формула unexplained PR ratio (оценка):

text

Unexplainable PR Ratio = (PR без объяснения / Всего PR) × 100%

Если > 20% — паттерн 3 в активной фазе. Если > 50% — вы в зоне субпрайма.

Кейс из trenches

Собирательный кейс. Детали изменены.

Enterprise, regulated, банк. 200 разработчиков. Стек: Java + Spring + Kafka + Oracle. Внедрили ИИ-ассистента по инициативе СТО. Цель — ускорить доставку фич. За полгода velocity выросла на 60%. Руководство довольно.

Через восемь месяцев — внутренний аудит. Комплаенс-требования: каждое изменение в критичных модулях должно быть объяснено и задокументировано.

Комплаенс-офицер открыл PaymentProcessor.java. 4000 строк. Сгенерировано ИИ. Документации нет. Он вызвал автора. Автор посмотрел на код и сказал: «Я не помню, что здесь». COMPLAINT-офицер закрыл файл. Написал в отчёте: «Требуется ручное ревью каждого изменения».

Через неделю релизы встали.

Аудит показал: 30% сгенерированного кода не имеет документации. Авторы промптов не могут объяснить логику. Часть кода использует паттерны, несовместимые с внутренними стандартами.

Катастрофы не было. Никто не уволился. Прод не упал. Но:

Релизы критичных модулей остановлены на 6 недель.

Три команды переведены на ручное ревью каждого PR.

Введён обязательный owmpet для каждого изменения.

Внедрён AI Usage Policy.

Стоимость комплаенс-аудита: \$80—120 тыс. (оценка).

Потеря времени: 6 недель релизного цикла.

Что сделали не так: не ввели ownership, не считали debt interest, не проверяли, что генерация совместима с комплаенс.

Что сделали правильно: остановились, провели аудит, ввели правила. Не стали ждать инцидента.

Кто заплатил? Банк — деньгами и временем. Команды — нервами. Но не клиенты. Пока.

Красные флаги

«Это просто хейтеры.»

«У нас не так.»

«На Reddit одни неудачники.»

«У нас всё под контролем.»

«Мы не читаем форумы. Мы делаем.»

«ИИ не может ошибаться, это модель.»

«Мы потом разберёмся.»

Антипаттерн

Как делать НЕ надо:

Игнорировать форумы, потому что «там одни жалобы».

Верить одному треду.

Не проверять паттерны у себя.

Считать review формальностью.

Не считать review latency.

Оставлять галлюцинированные зависимости в коде.

Чинить сгенерированный код сгенерированными патчами.

Сокращать команду под ИИ, не считая последствий.

Чеклист

- ☐ Проведён анонимный опрос в команде.
- ☐ Посчитан review latency.
- ☐ Проверена доля PR без объяснения.
- ☐ Проверены зависимости на галлюцинации.
- ☐ Проверена доля патчей поверх патчей.
- ☐ Проверены последствия сокращений (если были).
- ☐ Команда знает паттерны и их названия.
- ☐ Есть план реакции на каждый паттерн.
- ☐ Руководство знает о паттернах.
- ☐ Форумы читаются как диагностика, а не как шум.

Что запомнить

Один комментарий — это человек, у которого болит. Десять — это система. У вас она тоже есть. Проверьте.

Что почитать

DOU, Reddit r/programming, Hacker News — читать не как новости, а как диагностику. Ищите не истории, а совпадения. Если три команды в разных странах пишут одно и то же — это не совпадение.

«Accelerate» (Forsgren, Humble, Kim) — как отличать сигнал от шума в метриках.

«The Field Guide to Understanding Human Error» (Sidney Dekker) — почему системы ошибаются, а не люди.

Google SRE Book — как разбирать инциденты и находить паттерны.

«Thinking in Systems» (Donella Meadows) — как видеть системы, а не события.

Глава 4. Пять слоёв скрытого долга

«Мы думали, у нас один долг. Оказалось — пять. И они росли одновременно.»
— из разбора архитектурного аудита в продуктовой команде

Тезис

ИИ-код создаёт долг не в одном месте, а сразу в пяти: когнитивном, архитектурном, операционном, организационном и безопасностном. Лечить один слой бесполезно — остальные четыре продолжают расти. Слои усиливают друг друга: когнитивный порождает архитектурный, архитектурный — операционный, операционный — организационный. Безопасностный ждёт своего часа.

Ключевые вопросы

Где именно накапливается долг от ИИ-кода?

Как измерить каждый слой?

Какой слой самый опасный?

Как слои усиливают друг друга?

Как провести аудит по пяти слоям и не утонуть?

4.1. Пять слоёв: почему одного недостаточно

Вторник, 11:20. Архитектурный аудит в продуктовой команде. Тимлид открывает отчёт и говорит: «У нас технический долг». Архитектор кивает. Через час выясняется: то, что они называют «техническим долгом», — это пять разных проблем, которые выросли одновременно.

Первая: никто не помнит, зачем в модуле оплаты два индекса. Это когнитивный долг. Вторая: три сервиса дублируют логику скидок. Это архитектурный. Третья: алерты приходят, но никто не знает, что они значат. Это операционный. Четвёртая: два разработчика ушли, их модули никто не поддерживает. Это организационный. Пятая: в сгенерированном конфиге лежит ключ от прода. Это безопасностный.

Пять слоёв. Пять разных людей, которые должны их чинить. Пять разных метрик. Если вы лечите только технический долг — вы лечите симптом, а не систему.

Слой	Что включает	Кто отвечает	Когда срабатывает
Когнитивный	Потеря замысла, непонятный код	Автор, owner, команда	При первом изменении
Архитектурный	Дублирование, локальные оптимумы	Архитектор, tech lead	При масштабировании
Операционный	Нет тестов, нет логирования, алерты-загадки	SRE, on-call	При инциденте
Организационный	Ownership-вакуум, потеря экспертизы	CTO, VP Engineering	При уходе автора
Безопасностный	Секреты в коде, уязвимые зависимости	Security, комплаенс	При аудите или атаке

Когнитивный измеряется временем до контекста. Архитектурный — дублированием. Операционный — MTTR. Организационный — orphan code ratio. Безопасностный — находками сканера. Пять слоёв. Пять метрик. Одна таблица, которую никто не ведёт.

4.2. Когнитивный долг: никто не помнит замысел

Среда, 15:40. Разработчик открывает модуль, который нужно изменить. 800 строк на Go. Сгенерировано четыре месяца назад. Автор промпта ушёл — сначала в отпуск, потом в другую компанию. git blame показывает «AI assistant».

Он читает функцию ProcessPayment. Понимает процентов на тридцать. Остальное — «работает, но непонятно как». Почему здесь используется очередь вместо прямого вызова? Почему ретрай с экспоненциальной задержкой, а не фиксированной? Почему в одном месте context. WithTimeout, а в другом — time.After? Он не знает. Никто не знает.

Он идёт к тимлиду. Тимлид смотрит, пожимает плечами. Идёт к архитектору. Архитектор не видел этот модуль ни разу. Через час они втроём сидят в переговорке и гадают. Три senior-инженера, 40 лет опыта на троих, смотрят на код и не могут ответить на простой вопрос: почему?

Это когнитивный долг. Вы платите не за то, что код плохой. Вы платите за то, что никто не понимает, почему он хороший.

Когнитивный долг — самый коварный. Он не виден в метриках. Не проявляется в инцидентах. Он ждёт, пока кто-то попробует изменить код. Тогда срабатывает: изменение ломает три соседних модуля, потому что никто не знал про неочевидные связи.

Как его увидеть? Спросите нового разработчика: сколько времени ему нужно, чтобы понять модуль? Если час — долг низкий. Если день — средний. Если он приходит к вам через три дня и говорит «я не понимаю» — высокий. Это называется time to context. Если не можете назвать человека, который понимает модуль, — оценка 5.

4.3. Архитектурный долг: локальные оптимумы, дублирование

Четверг, 10:15. Архитектор смотрит на граф зависимостей. Три сервиса дублируют логику скидок. В одном — скидка считается по старой формуле. В другом — по новой. В третьем — по какой-то третьей, которую никто не помнит. Каждый сервис сгенерирован ИИ в разное время. Каждый — локальный оптимум. Глобально — хаос.

ИИ не знает, что скидка уже есть в трёх местах. Он знает про промпт. Промпт был «сделай скидку». Он сделал. Четвёртую. Пятая будет в следующем спринте.

Архитектурный долг выглядит так: дублирование логики (десять способов сделать одно и то же), dependency hell (сгенерированные зависимости, несовместимые друг с другом), несовместимые паттерны (один модуль на event-driven, другой на синхронных вызовах), нарушение границ (сервис А лезет в базу сервиса Б).

Измерить его можно через duplication rate — долю дублированного кода. Если у вас три реализации скидки, а должна быть одна, — это 200% дублирования в этой области. Если CI не проходит архитектурные проверки (fitness functions) — оценка 5.

Здесь есть соблазн: «Давайте отрефакторим». Отрефакторить три сервиса, которые никто не понимает, — это не рефакторинг. Это археология с риском обрушения.

4.4. Операционный долг: непредсказуемые отказы

Пятница, 2:47. Алерт. Модуль биллинга. On-call открывает дашборд. Метрики красные. Он открывает код. 1200 строк на Kotlin. Сгенерировано. Логирование отсутствует — вообще. Он не знает, где ставить breakpoint. Не знает, какие метрики смотреть. Не знает, что именно сломалось. Он знает только, что клиенты не могут оплатить.

В Slack — 200 алертов. Из них 5 требуют действия. Остальные 195 — шум. Через месяц никто не смотрит на алерты. Через два — алерты отключили. Через три — инцидент обнаружил клиент. Это alert-to-action ratio: доля алертов, которые приводят к действию. Если меньше 10% — вы в зоне операционного долга.

ИИ генерирует код, который делает то, что просили. Он не генерирует логирование. Не генерирует метрики. Не генерирует алерты. Не пишет runbook. Всё это — на команде. И часто этого нет.

Операционный долг — это когда код работает, но вы не можете понять, как он работает, когда он ломается. MTTR растёт. Инциденты повторяются. Если incident recurrence rate — доля повторных инцидентов — больше 20%, вы не чините. Вы закрашиваете.

4.5. Организационный долг: ownership-вакуум, сокращения

Понедельник, 9:00. Тимлид смотрит на список модулей. Пять модулей не имеют owner-a. Авторы ушли. Двое — уволены. Трое — перешли в другие команды. Модули работают. Пока.

Модуль работает. Но если что-то сломается — чинить некому. Можно, конечно, спросить у ИИ. Он сгенерирует фикс. Потом второй. Потом третий. Пока модуль не перестанет работать окончательно.

Организационный долг — это долг, который нельзя починить кодом. Только людьми. Или их отсутствием. ИИ его усиливает: раньше у кода был автор. Теперь автора нет. Формально есть тот, кто писал промпт. Но он может уйти. Или не помнить. Или не понимать. Ownership-вакуум — это когда код есть, а ответственного нет.

Измерить его можно через orphan code ratio — долю кода без owner-a. Если 20% — оценка 3. Если критичные модули знает один человек — оценка 5, независимо от процентов. Один человек — это bus factor = 1. Он в отпуске. Инцидент ждёт.

4.6. Безопасность и комплаенс: утечки, лицензии, supply chain

Среда, 14:30. Security-инженер запускает сканер. Находит ключ от прода в сгенерированном конфиге. Ключ лежит в репозитории три месяца. Кто его туда положил — неизвестно. Модель сгенерировала. Разработчик мержил. Ревьюер не заметил. Все трое уже не работают здесь. Ключ работает.

Безопасностный долг — это долг, который не проявляется, пока не станет поздно. ИИ генерирует код, который может содержать секреты в открытом виде, уязвимые зависимости, галлюцинированные пакеты (которые могут быть вредоносными), код, нарушающий лицензии, отсутствие валидации входных данных.

Он отличается от остальных. Не растёт постепенно. Срабатывает внезапно. Один инцидент — и вы теряете данные, деньги, репутацию.

Считать его можно через security findings per PR — сколько уязвимостей на PR. Если сканеры чистые и SBOM ведётся — оценка 1. Если ключи в репозитории и никто не проверяет зависимости — 5.

Здесь есть ложное успокоение: «У нас же не банк». Утечка ключа от staging — это доступ к staging. Через staging — доступ к продакшену. Через продакшен — ко всему. Банк вы или нет — неважно.

4.7. Как слои усиливают друг друга

Слои не существуют отдельно. Они связаны. И это самое опасное.

text

Когнитивный долг

↓

Никто не понимает код → нельзя безопасно изменить →

↓

Архитектурный долг

↓

Дублирование, локальные оптимумы → сложнее отладка →

↓

Операционный долг

↓

Инциденты, MTTR растёт → люди устают, уходят →

↓

Организационный долг

↓

Нет owner-а → никто не проверяет безопасность →

↓

Безопасностный долг

Пример первый. Когнитивный долг: никто не понимает модуль оплаты. Из-за этого нельзя безопасно изменить логику. Появляется дублирование: новый модуль пишется с нуля. Архитектурный долг растёт. Дублирование приводит к непредсказуемым инцидентам. Операционный растёт. Инциденты утомляют команду. Люди уходят. Организационный растёт. Без owner-а никто не проверяет зависимости. Безопасностный растёт.

Пример второй. В другой команде когнитивный долг привёл к тому, что новый разработчик сломал продакшен в первый день. Не потому что плохой. Потому что никто не объяснил, что этот модуль нельзя трогать по пятницам. Никто не помнил почему. Раньше знал один человек. Он ушёл в отпуск.

Каждый слой усиливает следующий. Если вы лечите только один — остальные четыре продолжают расти. Через год вы получаете систему, где все пять слоёв достигли максимума. Это и есть foreclosure.

Практика: что сделать завтра

Проведите аудит по пяти слоям. Оцените каждый по шкале 1—5. Запишите результат. Повторите через месяц.

Для каждого слоя назначьте ответственного. Когнитивный — tech lead. Архитектурный — архитектор. Операционный — SRE. Организационный — CTO/VP. Безопасностный — security.

Введите debt register по слоям. Одна таблица. Пять колонок. Обзор раз в спринт.

Проверьте связи между слоями. Какой слой самый слабый? Он усиливает остальные.

Начните с организационного. Если нет owner-а, остальные слои некому чинить.

Не назначайте одного человека на всё. Пять слоёв — пять ответственных. Иначе вы вернётесь к «у нас технический долг».

Метрики

Слой Метрика Формула Что показывает

Когнитивный Unexplainable PR ratio PR без объяснения / всего PR Потеря замысла

Когнитивный Time to context Время на понимание модуля Глубина долга

Архитектурный Duplication rate Дублированный код / всего кода Локальные оптимумы

Операционный MTTR Среднее время восстановления Способность чинить

Операционный Alert-to-action ratio Алерты с действием / всего алертов Шум в мониторинге

Организационный Orphan code ratio Код без owner-а / всего кода Ownership-вакуум

Организационный Bus factor Мин. число людей, знающих модуль Хрупкость команды

Безопасностный Security findings per PR Находки / PR Уязвимости

Формула Debt Score (оценка):

text

$\text{Debt Score} = (\text{Cognitive} + \text{Architectural} + \text{Operational} + \text{Organizational} + \text{Security}) / 5$

Если > 3 — вы в зоне субпрайма. Если > 4 — foreclosure близко.

Кейс из trenches

Собирательный кейс. Детали изменены.

Продуктовая команда, 8 разработчиков, mobile-приложение для e-commerce. Стек: Swift + Kotlin + Node.js + PostgreSQL + Redis. Внедрили ИИ-ассистента для ускорения фич. Через полгода velocity выросла на 70%.

В августе тимлид заметил: два разработчика ушли, их модули никто не поддерживает. Он провёл аудит по пяти слоям.

Тимлид сидел в переговорке с распечаткой. Пять колонок. Он вписывал числа и понимал: организационный — пять. Потому что два модуля не знает никто. Вообще никто. Он посмотрел на пустой стул напротив. На нём раньше сидел разработчик, который знал. Теперь стул пустой.

Когнитивный: 4/5. Сорок процентов модулей никто не может объяснить.

Архитектурный: 3/5. Дублирование логики скидок в трёх местах.

Операционный: 4/5. MTTR вырос с 40 минут до 4 часов.

Организационный: 5/5. Пять модулей без owner-а.

Безопасностный: 3/5. Нашли ключ от staging в репозитории.

Debt Score = $(4+3+4+5+3) / 5 = 3.8$. Зона субпрайма.

Что сделали:

Назначили owner для каждого модуля. Даже если owner — один человек на три модуля.

Ввели правило: «Нет owner-а — нет merge».

Провели архитектурный review. Убрали дублирование скидок.

Добавили логирование и метрики в критичные модули.

Ввели security-сканер в CI.

Провели обучение по AI-aware review.

Через три месяца:

Когнитивный: 3/5. Архитектурный: 2/5. Операционный: 2/5.

Организационный: 2/5. Безопасностный: 2/5.

Debt Score = 2.2. Вышли из зоны субпрайма.

Катастрофы не было. Никто не уволился. Прод не упал. Но если бы аудит не провели — через год был бы foreclosure. Они остановились вовремя.

Красные флаги

«У нас только технический долг.»

«Безопасность — это отдел безопасности.»

«Мы потом разберёмся с owner-ами.»

«MTTR вырос, но это из-за роста нагрузки.»

«У нас всё под контролем.»

«Мы не считаем долг по слоям, это сложно.»

«Давайте сначала починим код, потом людей.»

Антипаттерн

Как делать НЕ надо:

Считать, что долг один.

Лечить только технический долг.

Не назначать owner-a.

Игнорировать безопасность до аудита.

Не считать MTTR.

Не проверять связи между слоями.

Начинать с кода, а не с людей.

Верить, что «потом разберёмся».

Чеклист

- ☐ Проведён аудит по пяти слоям.
- ☐ Каждый слой оценён по шкале 1—5.
- ☐ Для каждого слоя назначен ответственный.
- ☐ Введён debt register.
- ☐ Проверены связи между слоями.
- ☐ Назначен owner для каждого критичного модуля.
- ☐ Введены метрики для каждого слоя.
- ☐ Руководство знает о пяти слоях.
- ☐ Есть план по каждому слою.
- ☐ Аудит повторяется раз в квартал.

Что запомнить

Долг не один. Их пять. Лечить один — значит кормить остальные четыре.

Что почитать

«The DevOps Handbook» (Kim, Debois, Willis, Humble) — про операционный долг и как его измерять.

«Building Secure and Reliable Systems» (Google) — про безопасный и операционный слои: как объединить их в одну систему.

«Software Architecture: The Hard Parts» (Ford, Richards, Sadalage, Dehghani) — про архитектурный долг и trade-off.

«The Phoenix Project» (Kim, Behr, Spafford) — про организационный долг и цену отсрочки.

«Thinking in Systems» (Donella Meadows) — про то, как слои усиливают друг друга.

Глава 5. Метрики, которые скрывают катастрофу

«Наш дашборд был зелёным. Прод — красным. Мы смотрели на дашборд.»
— из postmortem SaaS-компании

Тезис

Velocity, LOC и «скорость генерации» — это rating agencies субпрайма. Они показывают рост, пока не наступает foreclosure. Классические метрики не врут напрямую — они просто не видят того, что важно. Они измеряют то, что легко посчитать, а не то, что определяет выживание. Если метрика не включает стоимость владения — она врёт.

Ключевые вопросы

Почему классические метрики врут в эпоху ИИ?

Что мерить вместо них?

Почему DORA работает, но недостаточен?

Как cost of outage становится главной метрикой?

Как построить дашборд, который не скрывает катастрофу?

5.1. Метрики-обманки: LOC, PR count, story points

Понедельник, 10:00. Еженедельный статус. Тимлид открывает дашборд. Velocity — зелёная. LOC — растёт. PR count — рекорд за квартал. Руководство кивает. Все довольны. Через месяц — три инцидента. Через два — rewrite. Через три — СТО уволен. Дашборд не врал. Он показывал правду. Просто эта правда не имела отношения к выживанию.

Классические метрики измеряют то, что легко посчитать. Их легко посчитать, потому что они лежат на поверхности: строки, задачи, PR, story points. Но именно поэтому они не видят того, что важно.

Метрика Что измеряет Что не видит Почему обманка

LOC Объём кода Стоимость владения Больше кода ≠ больше ценности

PR count Скорость merge Качество review Больше PR ≠ лучше код

Story points Оценка сложности Реальная сложность Оценка ≠ реальность

Velocity Скорость доставки Стоимость доставки Скорость без направления — это бег в пропасть

«Скорость ИИ» Объём генерации Объём понимания Генерация ≠ понимание

Все пять метрик объединяет одно: они измеряют производство, а не владение. Производство видно сразу. Владение видно через год. Дашборд показывает первое. Второе вы узнаете из postmortem. Или из письма об увольнении.

Есть ирония. Чем быстрее вы генерируете, тем красивее дашборд. И тем ближе foreclosure. LOC растёт — и никто не понимает, что в этих строках. PR count растёт — и никто не помнит, что мержил вчера. Velocity растёт — и никто не считает, сколько будет стоить поддержка.

Дашборд — это зеркало. Если вы смотрите в зеркало и видите красивого человека, а через год у вас инфаркт — зеркало не виновато. Виноват тот, кто не пошёл к врачу.

5.2. Почему DORA работает, но недостаточен

DevOps-инженер настраивает дашборд. Четыре метрики: lead time, deployment frequency, MTTR, change failure rate. Всё по учебнику. Через месяц выясняется: lead time сократился, deployment frequency выросла, MTTR — стабилен. Но команда выгорела, инциденты повторяются, а новый разработчик не может понять код.

DORA работает. Но DORA измеряет доставку, а не владение. Она говорит, как быстро вы доставляете изменения и как быстро восстанавливаетесь после сбоев. Она не говорит, сколько стоит поддержка, сколько кода никто не понимает, сколько модулей без owner-а.

Метрика Что показывает Чего не видит

Lead time Время от коммита до прода Стоимость владения

Deployment frequency Частота релизов Качество кода

MTTR Время восстановления Причины инцидентов

Change failure rate Доля сбоев Скрытый долг

DORA хороша. Но она не создана для эпохи ИИ. Она создана для эпохи, когда код писал человек, и у кода был автор. Сейчас автора нет. DORA этого не замечает.

Если у вас DORA зелёная и вы думаете, что всё хорошо, — вы смотрите на rating agencies. Они ставят AAA. Через год будет reset.

5.3. AI-specific метрики: то, что DORA не видит

Тимлид добавил пять новых метрик в дашборд. На следующее утро финансовый директор спросил: «Что такое debt interest?» Тимлид объяснил. Финансовый директор кивнул и сказал: «Значит, мы платим проценты по кредиту, который не брали». Тимлид кивнул. Именно так.

Вот эти пять метрик — каждая через сцену.

Review latency.

Пятница, 18:40. Ревьюер открывает четырнадцатый PR за день. 600 строк. Он листает. Функции выглядят знакомо. Тесты зелёные. Он пишет «LGTM» и закрывает ноутбук. Сколько минут он потратил? Четыре. Это review latency. Если меньше тридцати — вы не ревьюите. Вы ставите печать. Через месяц баги уходят в прод. Команда удивляется: «Как мы это пропустили?» Пропустили, потому что не читали.

Rework rate.

Понедельник, 10:00. Разработчик открывает код, который сам сгенерировал неделю назад. Переписывает. Вторник — переписывает снова. Среда — снова. К пятнице он переписал 40% того, что сделал. Это rework rate. Треть того, что вы генерируете, выбрасывается. Вы платите за генерацию, за review, за отладку, за rewrite. Четыре раза за одну строку.

Orphan code ratio.

Тимлид смотрит на список модулей. Пять модулей без owner-а. Авторы ушли. Модули работают. Пока. Если что-то сломается — чинить некому. Пять из двадцати пяти. Двадцать процентов. Это orphan code ratio. Если критичные модули знает один человек — bus factor = 1. Он в отпуске. Инцидент ждёт.

Debt interest.

Команда тратит 40% времени на поддержку кода, который никто не понимает. Не на новые фишки. Не на архитектуру. На поддержку. Это debt interest. Если больше 30% — вы платите больше, чем зарабатываете. Если больше 50% — foreclosure близко.

Unexplainable PR ratio.

Спросите десять авторов: «Что делает этот код?» Двое не могут ответить. Это 20%. Пятеро — 50%. Это unexplainable PR ratio. Потеря замысла, измеренная в числах.

Все пять метрик измеряют владение, а не производство. Они не красивые. Они не растут. Они не радуют руководство. Но они говорят правду.

Можно добавить шестую: cost of incident. Сколько стоит один инцидент? Не только downtime, но и время команды, потерянные клиенты, репутация, упущенные возможности. Если cost of incident растёт быстрее velocity — вы в зоне субпрайма.

5.4. Cost of outage как главная метрика

On-call получил алерт в 3:14. Прод упал. Он открывает дашборд. MTTR — 4 часа. Через час он понимает: модуль биллинга. 1200 строк. Сгенерировано. Логирования нет. Breakpoint не поставить. Он звонит автору. Автор уволен. Звонит архитектору. Архитектор не видел модуль. Инцидент длится шесть часов. Утром СТО спрашивает: «Сколько мы потеряли?» Никто не знает. Никто не считал. Cost of outage — это то, что вы узнаете после.

Cost of outage — главная метрика, потому что она объединяет всё: качество кода, владение, review, архитектуру, безопасность. Если у вас высокий cost of outage — у вас проблемы во всех слоях.

Из чего он складывается:

потерянная выручка (сколько денег не пришло, пока прод лежал);

время команды (сколько человеко-часов ушло на восстановление);

ушедшие клиенты (сколько не вернулось);

репутация (сколько будущих сделок потеряно);

упущенные возможности (что не выпустили, пока тушили).

Формула (оценка):

text

$\text{Cost of Outage} = \text{Lost Revenue} + \text{Team Hours} \times \text{Hourly Rate} + \text{Churned Clients} \times \text{LTV} + \text{Reputation Damage}$

Reputation Damage — самая сложная часть. Её нельзя посчитать точно. Но можно оценить: сколько клиентов ушло после инцидента? Сколько не пришло? Сколько партнёров отказалось? Если хотя бы один — это уже число.

В SaaS-компании, которую я знаю, один инцидент на шесть часов стоил \$120—180 тыс. (оценка). Из них \$40 тыс. — потерянная выручка. \$30 тыс. — время команды. \$50—100 тыс. — ушедшие клиенты. \$10 тыс. — репутация. Экономия на зарплатах за год — \$200 тыс. Один инцидент съел половину.

Руководство понимает деньги. Velocity — нет. LOC — нет. «Скорость ИИ» — нет. Деньги — да. Показывайте деньги.

5.5. Как построить дашборд, который не врёт

Тимлид открывает новый дашборд. Слева — старые метрики: velocity, LOC, PR count. Справа — новые: review latency, rework rate, orphan code ratio, debt interest, cost of incident. Слева всё зелёное. Справа — жёлтое и красное. Он смотрит на правую часть. Теперь он знает, что делать.

Дашборд, который не врёт, строится на трёх вещах.

Не убирайте старые метрики. Velocity, LOC, PR count — это не ложь. Это половина правды. Оставьте их. Но добавьте вторую половину. Тогда вы увидите картину целиком. Если убрать старые — вы потеряете язык, на котором говорит руководство. Если оставить только старые — вы потеряете правду.

Метрика без владельца — мёртвая метрика. Если у review latency нет ответственного, она будет расти. Если у orphan code ratio нет ответственного, он будет расти. У каждой метрики — человек. Один. Не «команда». Человек. Иначе метрика превращается в отчёт, который никто не читает.

Метрика без действия — бесполезна. Если review latency растёт, а вы ничего не делаете — зачем измерять? Метрика должна триггерить действие. Если review latency > 30 минут — остановите merge. Если orphan code ratio > 20% — назначьте owner-а. Если debt interest > 30% — пересмотрите план.

Метрика Тип Ответственный Триггер

Lead time DORA DevOps > 3 дней

Deployment frequency DORA DevOps < 1 в день

MTTR DORA SRE > 1 часа

Change failure rate DORA SRE > 15%

Review latency AI-specific Tech lead > 30 минут

Rework rate AI-specific Tech lead > 30%

Orphan code ratio AI-specific Архитектор > 20%

Debt interest AI-specific CTO > 30%

Cost of incident Экономика CTO > \$10 тыс.

Каждая метрика — с ответственным. Каждая — с триггером. Каждая — с действием. Это не дашборд. Это пульт управления.

Практика: что сделать завтра

Замените одну vanity-метрику на метрику владения. Уберите LOC. Добавьте debt interest. Посмотрите, что изменится в разговоре с руководством.

Посчитайте cost одного инцидента за последние 6 месяцев. Lost revenue + team hours + churned clients + reputation. Сравните с экономией на зарплатах.

Добавьте AI-specific метрики в дашборд. Review latency, rework rate, orphan code ratio, debt interest. Не убирайте DORA. Добавьте к ней.

Назначьте ответственного за каждую метрику. Один человек. Не «команда». Не «все». Человек.

Определите триггер для каждой метрики. Если review latency > 30 минут — остановите merge. Если orphan code ratio > 20% — назначьте owner-а.

Покажите cost of incident руководству в деньгах. Не в процентах. Не в story points. В деньгах.

Метрики

Категория Метрика Формула Триггер

DORA Lead time Время от коммита до прода > 3 дней

DORA Deployment frequency Релизов в день < 1

DORA MTTR Среднее время восстановления > 1 часа

DORA Change failure rate Сбоев / релизов > 15%

AI-specific Review latency Время от PR до merge > 30 минут

AI-specific Rework rate Переписанный код / всего > 30%

AI-specific Orphan code ratio Код без owner-а / всего > 20%

AI-specific Debt interest Время на поддержку / общее > 30%

AI-specific Unexplainable PR ratio PR без объяснения / всего > 20%

Экономика Cost of incident Lost Revenue + Team Hours + Churned Clients + Reputation > \$10 тыс.

Формула cost of incident (оценка):

text

Cost of Incident = Lost Revenue + (Team Hours × Hourly Rate) + (Churned Clients × LTV) + Reputation Damage

Кейс из trenches

Собирательный кейс. Детали изменены.

B2B SaaS, 25 разработчиков, платформа для управления проектами. стек: Go + React + PostgreSQL + Kubernetes. Внедрили ИИ-ассистента в январе. К июню velocity выросла на 200%. Дашборд зелёный. Руководство довольно. СТО получил бонус.

В июле — первый инцидент. Прод упал на 4 часа. В августе — второй. В сентябре — третий. Каждый раз чинили дольше. Каждый раз никто не понимал код. Каждый раз on-call не спал.

В октябре СТО попросили посчитать cost of incidents. Он собрал данные.

Первый инцидент: \$45 тыс. (потерянная выручка \$20 тыс., время команды \$15 тыс., ушедшие клиенты \$10 тыс.).

Второй: \$80 тыс.

Третий: \$120 тыс.

Итого за три месяца: \$245 тыс. (оценка).

Экономия на зарплатах за год: \$300 тыс. (сократили трёх разработчиков).

СТО посмотрел на цифры. Потом на дашборд. Velocity — зелёная. Cost of incidents — \$245 тыс. Он закрыл ноутбук. Пошёл к CEO.

Что сделали:

Вернули двух разработчиков из трёх. Не потому что «надо больше людей». Потому что нужен ownership.

Ввели AI-specific метрики в дашборд. Review latency, rework rate, orphan code ratio, debt interest.

Назначили owner для каждого критичного модуля.

Ввели правило: «Нет owner-а — нет merge».

Добавили cost of incident в ежемесячный отчёт для руководства.

Через полгода:

Velocity упала на 40%. Дашборд перестал быть зелёным.

Cost of incidents: \$0 за квартал.

Debt interest упал с 45% до 20% (оценка).

Команда перестала дежурить 24/7.

CEO сказал: «Сколько мы потеряли?» СТО ответил: «\$245 тыс.» CEO кивнул. Больше не спрашивал. СТО остался на работе. Бонуса не будет.

Красные флаги

«У нас всё зелёное.»

«Метрики не врут.»

«Velocity выросла, значит, всё хорошо.»

«Cost of incident? Мы не считаем.»

«У нас DORA зелёная, зачем что-то ещё?»

«Дашборд красивый, руководство довольно.»

«Мы потом посчитаем.»

Антипаттерн

Как делать НЕ надо:

Считать velocity единственной метрикой.

Игнорировать cost of incident.

Не добавлять AI-specific метрики.

Не назначать ответственного за метрику.

Не определять триггеры.

Показывать руководству проценты вместо денег.

Верить, что «зелёный дашборд = всё хорошо».

Не считать стоимость владения.

Чеклист

- ☐ Заменена хотя бы одна vanity-метрика на метрику владения.
- ☐ Посчитан cost одного инцидента за последние 6 месяцев.
- ☐ Добавлены AI-specific метрики в дашборд.
- ☐ Назначен ответственный за каждую метрику.
- ☐ Определён триггер для каждой метрики.
- ☐ Cost of incident показан руководству в деньгах.
- ☐ DORA не убрана, но дополнена.
- ☐ Дашборд пересматривается раз в квартал.
- ☐ Команда знает, что метрики без действия бесполезны.
- ☐ Руководство понимает разницу между производством и владением.

Что запомнить

Если метрика не включает стоимость владения — она врёт. Не потому что злая. Потому что слепая.

Что почитать

DORA State of DevOps Report — база. Но читайте с вопросом: «Что эти метрики не видят?»

«Accelerate» (Forsgren, Humble, Kim) — как отличать сигнал от шума.

«How to Measure Anything» (Douglas Hubbard) — как считать то, что кажется неизмеримым. Особенно cost of incident.

«The DevOps Handbook» (Kim, Debois, Willis, Humble) — про операционные метрики и их ограничения.

«Thinking in Systems» (Donella Meadows) — про то, почему метрики без контекста врут.

Часть II. Механика отказа: как именно ИИ ломает инженерную систему

Глава 6. Чёрные ящики и потеря замысла

«Код работает. Не трогай.»

— надгробная плита на могиле архитектуры

Тезис

Проблема ИИ-кода не в багах. Проблема в отсутствии модели. Баг можно найти и починить. Потерю замысла — нельзя. Код, который никто не понимает, невозможно эволюционировать: его можно только копировать, обходить или переписывать. Чёрный ящик — это не метафора, а точное описание состояния: вход известен, выход известен, внутренности — нет. Пока это состояние сохраняется, каждое изменение — это русская рулетка.

Ключевые вопросы

Почему сгенерированный код трудно сопровождать, даже если он работает?

Что такое «потеря замысла» и почему она страшнее бага?

Почему комментарии и документация не решают проблему?

Как отлаживать код, если не знаешь логику?

Как вернуть понимание — и можно ли его вернуть вообще?

6.1. Разница между «работает» и «понятно, почему работает»

Вторник, 11:30. Модуль PricingEngine в проде три месяца. Тесты зелёные. Метрики в норме. Никто не трогает. На вопрос «как оно работает?» тимлид пожимает плечами: «Работает — и хорошо».

Через неделю продакт приходит с требованием: добавить скидку для корпоративных клиентов. Тимлид открывает модуль. 1400 строк на TypeScript. Сгенерировано в феврале. Автор промпта перешёл в другую команду. git blame показывает: «AI assistant». Тимлид читает функцию за функцией. Понимает процентов на двадцать. Остальное — «работает, но непонятно как».

Он делает изменение. Тесты проходят. Деплой. Через час — алерт. Скидка применяется дважды для клиентов с определённым тарифом. Никто не знал, что в модуле есть неочевидная связь между тарифом и типом клиента. Она была в коде. Её не было в головах.

Разница простая. Первое — это состояние. Второе — это модель. Модель позволяет предсказать, что будет, если изменить X. Состояние позволяет только наблюдать: сейчас X ведёт себя так. Что будет завтра — неизвестно.

ИИ генерирует состояние. Не модель. Он не объясняет, почему выбрал этот алгоритм, почему здесь `reduce`, а не `map`, почему в одном месте мемоизация, а в другом — нет. Он просто выдаёт результат, который проходит тесты. Модель остаётся у модели. Вам — состояние.

Это и есть чёрный ящик. Вход известен: вы даёте данные. Выход известен: вы получаете результат. Внутренности — нет. Пока вы не трогаете — работает. Как только трогаете — вы не инженер. Вы археолог, который копает в темноте.

6.2. Потеря замысла: почему это страшнее бага

Четверг, 15:20. Разработчик открывает модуль `NotificationService`. Нужно добавить новый канал — `push`. Он видит: функция `sendNotification` принимает объект `options`, в котором есть поле `channel` со значениями `email`, `sms`, `webhook`. Добавляет `push`. Мержит.

Через день — инцидент. `Push`-уведомления уходят клиентам, которые их не заказывали. Оказывается, в `options` было ещё поле `priority`, и при `priority === 'high'` `NotificationService` игнорировал `channel` и отправлял во все каналы, включая те, которые не зарегистрированы. Никто не знал про эту логику. Её не было ни в документации, ни в комментариях. Она была в коде — как мина.

Это потеря замысла. Вы не просто не понимаете код. Вы не понимаете, почему он такой. Замысел — это набор решений, которые кто-то принял. Почему `priority` перебивает `channel`? Почему это не задокументировано? Почему это не в тестах? Ответ один: ИИ так сгенерировал. А тот, кто мержил, не спросил почему.

Баг — это отклонение от замысла. Если замысел известен, баг можно найти и починить. Если замысла нет — баг неотличим от фичи. Вы не знаете, что должно было быть. Вы знаете только, что происходит.

Потеря замысла страшнее бага. Баг локален. Потеря замысла — системна. Она распространяется на все будущие изменения. Каждый раз, когда кто-то трогает код, он не может предсказать последствия. Он может только гадать. И молиться.

6.3. Почему комментарии и документация не спасают

Среда, 10:15. Тимлид решает: «Надо задокументировать». Команда садится писать комментарии к сгенерированному коду. Через неделю в модуле `PaymentProcessor` появляются строки:

```
typescript
// Обработывает платёж
async function processPayment (payment: Payment): Promise <Result> {
  // Проверяем валюту
  if (payment.currency === «USD») {
    // Конвертируем
    const amount = await convert(payment.amount);
    // Возвращаем результат
    return {status: 'ok', amount};
  }
  // Возвращаем ошибку
  return {status: 'error', code: «UNSUPPORTED_CURRENCY»};
}
```

}

Комментарии описывают что делает код. Они не объясняют почему. Почему только USD? Почему не EUR? Почему конвертация здесь, а не в вызывающем коде? Почему ошибка возвращается, а не бросается? Почему Result, а не PaymentResult?

Через месяц комментарии устареют. Код изменится, комментарии — нет. Через полгода они станут ложью, которая хуже отсутствия комментариев. Читатель поверит комментарий и ошибётся.

Тимлид попробовал три подхода. Первый — комментарии. Умерли через месяц. Второй — документация. Умерла через полгода. Третий — тесты. Выжили, потому что их нельзя забыть: они падают в CI.

Документация умирает по той же причине. Она описывает интерфейс. Замысел живёт в голове. Если головы нет — документация становится гробницей для мёртвых решений.

Что выживает:

Тесты, которые проверяют поведение, а не реализацию. Не «функция вызывает convert», а «при USD-платеже сумма конвертируется по курсу на момент запроса». Тест — это замысел, записанный на языке, который нельзя забыть.

ADR (Architecture Decision Records). Не «что делает код», а «почему мы решили так». Один файл — одно решение. Дата. Контекст. Альтернативы.

Owner, который помнит. Не документация. Человек. Пока он здесь.

Комментарии — это надгробные плиты. Тесты — это живые свидетели.

6.4. Отладка в темноте: где ставить breakpoint

Пятница, 2:47. Прод упал. Алерт в модуле RecommendationEngine. On-call открывает код. 2000 строк на Python. Сгенерировано. Логирование — одна строка: logger.info («recommendation generated»). Ни параметров, ни стека, ни контекста.

Он ставит breakpoint. Запускает. Функция проходит точку. Идёт дальше. Он ставит второй breakpoint. Та же история. К третьему он понимает: он не отлаживает. Он перебирает. Как обезьяна, которая нажимает кнопки в надежде, что загорится лампочка.

Это отладка в темноте. Вы не знаете, где искать. Вы можете только перебирать. Как в старом анекдоте: «Ищу ключи не там, где потерял, а там, где светло». Только у вас нет даже светлого места.

В сгенерированном коде нет якорей для отладки. Нет комментариев, которые подскажут «здесь важная логика». Нет логов, которые покажут путь. Нет метрик, которые укажут на аномалию. Есть только код и ошибка.

Тимлид попробовал три вещи. Первая — логирование на входе и выходе каждой значимой функции. Вторая — метрики, а не только логи. Третья — gunbook для модулей, которые падают в 3 ночи. Все три выжили. Потому что их нельзя забыть: они падают в CI или в 3 ночи.

Отладка — это не про инструменты. Это про понимание. Без понимания инструменты бесполезны.

6.5. Как вернуть понимание

Можно ли вернуть замысел, если он потерян? Частично. Полностью — нет. Но можно восстановить достаточно, чтобы код перестал быть чёрным ящиком.

Археология. Выделите один модуль. Возьмите человека, который его трогал последним — или того, кто вообще не трогал. Попросите написать на бумаге, что делает каждая функция. Не читая код. По памяти. Потом сравните с кодом. Разница покажет, где замысел потерян.

Контракты. Для каждой публичной функции напишите контракт: вход, выход, инварианты, ошибки. Не «функция обрабатывает платёж». А «функция принимает платёж в валюте X, возвращает сумму в USD, бросает `InvalidCurrency`, если валюта не поддерживается». Контракт — это минимальная модель. Он не объясняет, почему так, но объясняет, что должно быть.

Тесты на поведение. Каждый контракт превращается в тест. Не тест реализации, а тест поведения. После этого изменения можно делать безопасно: тесты поймут отклонение от модели.

ADR на каждое решение. Почему это так? Почему не иначе? Не для каждого `if`, а для каждого значимого выбора. Через год это будет единственный способ узнать замысел.

Owner. У модуля должен быть человек. Не «команда». Человек. Он не обязан помнить всё. Но он обязан отвечать. И когда он уходит — он передаёт контекст. Не «вот код, разберись». А «вот контракты, вот тесты, вот ADR, вот что я знаю».

Понимание нельзя купить. Его можно только построить. И это долго. Быстрее, чем `rewrite`. Дешевле, чем `foreclosure`. Но всё равно долго.

Практика: что сделать завтра

Проведите археологию одного модуля. Возьмите модуль, который трогали последним. Попросите разработчика написать по памяти, что он делает. Сравните с кодом. Разница — карта потерянного замысла.

Напишите контракты для трёх критичных функций. Вход, выход, инварианты, ошибки. Без этого изменения — это гадание.

Проверьте логирование в критичных модулях. Если лог — одна строка `logger.info` («done»), вы не сможете отлаживать. Добавьте вход и выход каждой значимой функции.

Введите правило: «Не мерзить код, который не можешь объяснить на пальцах». Не на языке кода. На пальцах. Человеку, который не знает контекста. Если не получается — не мерзить.

Начните ADR. Один файл на одно решение. Дата. Контекст. Альтернативы. Почему так. Не для всего. Для того, что может удивить через год.

Назначьте owner для модуля. Человек. Не команда. Он не обязан всё знать. Он обязан отвечать.

Метрики

Метрика Формула Что показывает

Unexplainable PR ratio PR без объяснения / всего PR Потеря замысла

Time to context Время на понимание модуля Глубина чёрного ящика

ADR coverage Модулей с ADR / всего модулей Восстановленный замысел

Log coverage Функций с логированием / всего функций Готовность к отладке

Runbook coverage Модулей с runbook / критичных модулей Готовность к инциденту

Формула black box ratio (оценка):

text

$$\text{Black Box Ratio} = (\text{Unexplainable modules} / \text{Всего модулей}) \times 100\%$$

Тимлид посчитал: из двадцати пяти модулей пять он не может объяснить. Двадцать процентов. Если больше — вы работаете не с системой, а с набором мин.

Кейс из trenches

Собирательный кейс. Детали изменены.

DevTools-компания, 30 инженеров, платформа для аналитики кода. Стек: TypeScript + Node.js + Rust + ClickHouse. Внедрили ИИ-ассистента год назад. К концу года 60% кода написано ИИ.

В январе новый разработчик получил задачу: добавить поддержку нового языка в парсер. Он открыл модуль LanguageDetector. 800 строк на Rust. Сгенерировано. Комментариев — три. Тестов — ноль. git blame — «AI assistant».

Он попросил помощь. Тимлид сказал: «Я тоже не знаю, как оно работает». Архитектор сказал: «Мы его не трогали год, потому что работает». Через два дня разработчик сдался и написал свой детектор — 200 строк. Работал хуже, но был понятен.

Через месяц два детектора сосуществовали. Через два — прод падал три раза, потому что они давали разные результаты для одного языка. Через три — команда решила: оставить один. Какой? Никто не знал, какой из них правильный. У сгенерированного не было модели. У нового — была, но хуже.

Тимлид сел за археологию. Восемь часов, три инженера. Разобрали LanguageDetector по функциям. Написали контракты для трёх ключевых. Написали 40 тестов на поведение. Задокументировали три ADR: почему такой алгоритм, почему такая структура данных, почему такие edge cases. Удалили второй детектор. Назначили owner.

Через полгода сгенерированный детектор работал. Не потому что стал понятнее. А потому что у него появилась модель — контракты, тесты, ADR. Модель, которую не дал ИИ. Модель, которую построила команда.

Стоимость археологии: 24 человеко-часа (оценка). Стоимость rewrite, если бы пошли этим путём: 200+ человеко-часов. Разница — в понимании.

Красные флаги

«Работает — не трогай.»

«Разберёмся потом.»

«Нам не нужно понимать, нам нужно доставлять.»

«Комментарии потом добавим.»

«Тесты — это для слабых команд.»

«У нас нет времени на археологию.»

«ИИ знает, что делает.»

Антипаттерн

Как делать НЕ надо:

Мержить код без объяснения.

Писать комментарии, описывающие «что», а не «почему».

Полагаться на документацию, которая устаревает.

Не добавлять логирование в критичные модули.

Не писать runbook.

Оставлять модуль без owner-а.

Верить, что «работает» = «можно поддерживать».

Делать rewrite, не попробовав археологию.

Чеклист

- ☐ Проведена археология хотя бы одного модуля.
- ☐ Написаны контракты для критичных функций.
- ☐ Проверено логирование в критичных модулях.
- ☐ Введено правило: «Не мержить код, который не можешь объяснить на пальцах».
- ☐ Начаты ADR.
- ☐ Назначен owner для каждого критичного модуля.
- ☐ Есть runbook для модулей, которые падают в 3 ночи.
- ☐ Black Box Ratio известен и отслеживается.
- ☐ Команда знает разницу между «работает» и «понятно, почему работает».
- ☐ Никто не говорит «ИИ знает, что делает».

Что запомнить

Код, который работает, но непонятен, — это не актив. Это мина с часовым механизмом. Взрывается не тогда, когда сломался. Взрывается тогда, когда кто-то попробовал изменить.

Что почитать

«Working Effectively with Legacy Code» (Michael Feathers) — как разбирать код без автора. Прямое руководство к археологии.

«Refactoring» (Martin Fowler) — как менять код, не ломая поведение. Но сначала — понять поведение. Без этого рефакторинг — это переписывание.

«Domain-Driven Design» (Eric Evans) — про то, как строить модель, а не состояние.

«Documenting Software Architectures» (Clements et al.) — про ADR и архитектурные решения.

«The Pragmatic Programmer» (Hunt, Thomas) — про разницу между «работает» и «правильно».

Глава 7. Парадокс review: генерация быстрее мышления

«Я не ревьюю PR. Я ставлю печать. Печать называется „LGTM“.»

— из треда на DOU про review-усталость

Тезис

ИИ пишет 500 строк, человек ревьюит 500 строк. Ревью становится узким местом — и его начинают имитировать. Review latency падает, defect escape rate растёт. Это не лень ревьюера. Это математика: если у вас 15 PR в день и 8 часов, у вас 32 минуты на PR. Включая чтение, понимание, проверку тестов, обсуждение. 32 минуты на 700 строк — это не ревью. Это имитация. И она опаснее, чем отсутствие ревью вообще, потому что создаёт иллюзию контроля.

Ключевые вопросы

Почему review не масштабируется вместе с генерацией?

Что такое rubber-stamping и как его распознать?

Почему «выглядит нормально» — это когнитивная ловушка?

Как ревьюить быстрее, не теряя качество?

Что такое risk-based review и почему он работает?

7.1. Асимметрия: генерация vs ревью

Разработчик открывает чат с ИИ. Пишет промпт: «Добавь поддержку вебхуков в модуль уведомлений». Через 30 секунд — 600 строк. Он просматривает. Выглядит логично. Мержит.

Ревьюер открывает этот PR. 600 строк. Незнакомый модуль. Сгенерировано. Он читает первую функцию. Понимает. Вторую. Понимает. Третью — начинает терять нить. Четвёртую — пропускает. К десятой он смотрит на часы: 12 минут прошло. У него ещё 14 PR в очереди. Он пишет «LGTM» и закрывает.

Генерация занимает 30 секунд. Ревью — 30 минут. Если бы ревьюер читал внимательно. Он не читал. Он ставил печать.

Вот асимметрия. ИИ ускорил генерацию в 100 раз. Ревью не ускорилось вообще. Оно осталось линейным: чем больше строк, тем больше времени. Если раньше разработчик писал 100 строк в день и ревьюер тратил на них 15 минут, то теперь разработчик генерирует 1500 строк, а ревьюер должен тратить 4 часа. У него нет 4 часов. У него 8 часов на всё.

Что происходит? Ревьюер делает то, что делает любой человек под давлением объёма: он сокращает. Сначала сокращает чтение. Потом сокращает понимание. Потом сокращает проверку тестов. Остаётся ритуал: открыть PR, пролистать, поставить галочку, закрыть.

Это не халатность. Это адаптация. Система требует от ревьюера невозможного — и он находит способ выжить. Способ называется rubber-stamping.

7.2. Rubber-stamping: как выглядит имитация review

Ревьюер открывает двенадцатый PR за день. 700 строк. Он листает. Функции выглядят знакомо. Тесты зелёные. Он пишет «LGTM» и идёт домой. В понедельник он не вспомнит этот PR. Через месяц — тем более.

Это rubber-stamping. Формально ревью было. Фактически — нет.

Как распознать rubber-stamping в своей команде:

Review latency меньше 5 минут на PR. Если PR из 500 строк ревьюится за 3 минуты, его не читали.

Review depth — меньше 1 минуты на 100 строк. Если это число падает, ревью умирает.

LGTM rate — больше 90%. Если почти все PR получают «LGTM» без комментариев, ревьюеры не вовлечены.

Нет комментариев, кроме «nit». Если единственные комментарии — про пробелы и запятые, значит, логику не смотрели.

Никто не просит изменения. Если за неделю ни один PR не был отправлен на доработку, ревью — ритуал.

Rubber-stamping опасен не тем, что пропускает баги. Он опасен тем, что создаёт иллюзию контроля. Команда думает: «У нас есть ревью». Руководство думает: «У нас есть процесс». А в реальности — печать. Печать называется «LGTM». Она не читает код. Она его штампует.

Через месяц баги уходят в прод. Команда удивляется: «Как мы это пропустили?» Пропустили, потому что не читали. Через два — инцидент. Через три — postmortem. В postmortem напишут: «Недостаточное покрытие тестами». Не напишут: «Ревьюер не читал, потому что у него было 32 минуты на 700 строк».

7.3. Психология: «выглядит нормально» как когнитивная ловушка

Четверг, 11:15. Ревьюер открывает PR. 400 строк. Сгенерировано. Он читает первую функцию. Всё логично. Вторую. Тоже. Третью. Он ловит себя на мысли: «Выглядит нормально». Четвёртую пропускает. Пятую — по диагонали. К десятой он уверен: код хороший. Он мерзит.

Через неделю — баг. В той самой функции, которую он пропустил. Он открывает код. Смотрит. Понимает: баг был очевиден. Если бы он читал.

«Выглядит нормально» — это не оценка. Это когнитивная ловушка. Мозг экономит энергию: когда первые несколько элементов паттерна совпадают с ожиданием, он перестаёт проверять остальные. Это называется «эффект первого впечатления». Работает в жизни. Не работает в ревью.

ИИ усиливает ловушку. Сгенерированный код стилистически однороден. Он выглядит «правильно». Отступы, нейминг, структура — всё как в учебнике. Мозг ревьюера расслабляется: «Это не спагетти, это нормальный код». И перестаёт искать логические ошибки. А они там есть — просто выглядят аккуратно.

Вот почему сгенерированный код опаснее рукописного. Рукописный код часто выглядит плохо — и это сигнал: «Смотри внимательно». Сгенерированный код выглядит хорошо — и это антисигнал: «Всё в порядке, можно не смотреть». А внутри — мина.

Что делать с ловушкой:

Читайте не подряд, а выборочно. Начните с конца. Потом с середины. Потом с начала. Мозг не успеет расслабиться.

Ищите не «что не так», а «что здесь может быть не так». Не оценивайте. Предсказывайте. Где здесь может быть баг? Где здесь может быть неочевидная связь?

Задавайте вопросы вслух. Даже если некому. «Почему здесь reduce? Почему не map? Почему здесь try/catch, а не throw?» Если ответа нет — это красный флаг.

Не ревьюйте больше 300 строк за раз. После 300 строк внимание падает на 40% (оценка). Разбейте PR или отложите.

«Выглядит нормально» — это не ревью. Это капитуляция.

7.4. Уровни риска: не всё ревьюить одинаково

Тимлид смотрит на очередь PR. 18 штук. Половина — сгенерированный boilerplate. Четверть — тесты. Четверть — бизнес-логика. Он понимает: если ревьюить всё одинаково, не хватит времени ни на что.

Он вводит уровни риска.

Уровень	Что входит	Кто ревьюит	Сколько времени
Критичный	Платежи, auth, персональные данные, архитектура	Минимум 2 ревьюера	30—60 минут

Средний	Бизнес-логика, интеграции, API	1 ревьюер	15—30 минут
---------	--------------------------------	-----------	-------------

Низкий	Boilerplate, тесты, документация, стиль	Автомерж или 1 быстрый ревьюер	0—5 минут
--------	---	--------------------------------	-----------

Критичный PR не мерджится без двух ревьюеров. Один — автор, второй — независимый. Оба должны понимать, что делает код. Если не понимают — PR не мерджится. Точка.

Средний PR ревьюит один человек. Но он обязан прочитать. Не пролистать. Прочитать. Если PR больше 300 строк — разбить или отложить.

Низкий PR — автомерж. Тесты, boilerplate, документация. Если CI зелёный — мержим. Если CI красный — не мержим. Никто не тратит на это время.

Risk-based review не ускоряет ревью вообще. Он ускоряет ревью критичного. Потому что освобождает время от низкого. Если вы тратите 30 минут на boilerplate, у вас нет 30 минут на платежи. Если вы тратите 2 минуты на boilerplate, у вас есть 28 минут на платежи.

Что меняется:

Review latency для критичных PR — растёт. Это нормально. Лучше 2 часа на платёж, чем 5 минут.

Review latency для низких PR — падает до нуля. Автомерж.

Defect escape rate — падает. Потому что критичное ревьюится внимательно.

Review-усталость — падает. Потому что ревьюер не тратит силы на ерунду.

Главное правило: критичный PR не мержится, пока хотя бы один ревьюер не может объяснить, что делает код. Не «выглядит нормально». А «я понимаю, что здесь происходит, и могу объяснить».

7.5. AI-aware review: чеклист и запрет на слепой merge

Тимлид пишет чеклист. Не для всех PR. Для критичных. Он вешает его в шаблон PR. Теперь каждый критичный PR проходит через семь вопросов.

Чеклист критичного PR:

Понимание. Можешь объяснить, что делает код? Если нет — не мержить.

Безопасность. Проверены ли входные данные? Нет ли секретов в коде? Нет ли уязвимых зависимостей?

Зависимости. Все ли импорты существуют? Все ли они лицензионно чистые? Нет ли галлюцинированных пакетов?

Тесты. Есть ли тесты на поведение? Покрывают ли они edge cases? Не сгенерированы ли тесты тем же ИИ, что и код?

Обработка ошибок. Что происходит при ошибке? Логируется ли? Есть ли retry? Есть ли fallback?

Логирование и метрики. Есть ли лог на входе и выходе? Есть ли метрика для мониторинга?

Стоимость. Не создаёт ли код непредсказуемых облачных расходов? Нет ли бесконечных циклов? Нет ли N+1 запросов?

Если хотя бы один пункт — «нет», PR не мержится. Не «доработаем потом». Не мержится.

Запрет на слепой merge — это не бюрократия. Это единственный способ не превратить ревью в ритуал. Если вы разрешаете merge без проверки, вы разрешаете foreclosure. Рано или поздно.

AI-aware review отличается от обычного одним: вы предполагаете, что код может быть сгенерирован. Не «автор написал и подумал». А «модель сгенерировала и не подумала». Зна-

чит, проверка должна быть жёстче. Не потому что автор плохой. Потому что у автора не было замысла — был промпт.

Практика: что сделать завтра

Посчитайте review latency за последнюю неделю. Среднее время от открытия PR до merge. Если меньше 10 минут на PR из 300+ строк — вы в зоне rubber-stamping.

Введите уровни риска. Критичный, средний, низкий. С разным временем и числом ревьюеров. Начните с критичного. Остальное — потом.

Повесьте чеклист в шаблон PR. Семь вопросов. Для критичных PR — обязательны. Если хотя бы один «нет» — не мержить.

Введите правило: «Не мержить код, который не можешь объяснить». Не «выглядит нормально». А «я понимаю, что здесь происходит». Если не понимаешь — не мержить. Точка.

Проверьте LGTM rate. Если больше 90% PR получают LGTM без комментариев — ревью умирает. Введите обязательный комментарий для критичных PR: что проверено, что нет.

Разбейте большие PR. Если PR больше 300 строк — разбить. Если нельзя разбить — отложить и ревьюить утром, на свежую голову.

Метрики

Метрика Формула Что показывает

Review latency Время от PR до merge Скорость ревью

Review depth Время на ревью / LOC Глубина ревью

LGTM rate LGTM без комментариев / всего PR Rubber-stamping

Defect escape rate Баги в проде / всего багов Качество ревью

Rework rate Переписанный код / всего кода Пропущенные проблемы

Review capacity Доступное время / объём PR Перегрузка ревьюера

Формула review capacity (оценка):

text

$\text{Review Capacity} = (\text{Доступное время на ревью} / \text{Объём PR}) \times 100\%$

Если capacity <50% — ревьюер не успевает. Если <20% — ревью имитируется. Если <10% — ревью нет.

Кейс из trenches

Собираемый кейс. Детали изменены.

Финтех-стартап, 40 инженеров, платформа для платежей. стек: Ruby on Rails + Go + PostgreSQL + Kafka. Внедрили ИИ-ассистента в марте. К июню velocity выросла на 150%. Дашборд зелёный. Руководство довольно.

В июле тимлид заметил: review latency упала до 8 минут на PR. Раньше было 45 минут. Он посмотрел на PR. 600 строк. 8 минут. Он понял: ревью умерло.

Он провёл аудит. LGTM rate — 94%. Review depth — 0.8 минуты на 100 строк. Defect escape rate вырос в три раза за квартал. Три инцидента в проде, все — из PR, которые получили «LGTM» без комментариев.

Он собрал команду. Показал цифры. Никто не удивился. Один ревьюер сказал: «У меня 15 PR в день. Я не могу читать. Я могу только штамповать».

Что сделали:

Ввели уровни риска. Критичный, средний, низкий.

Критичный PR — минимум 2 ревьюера, 30—60 минут, обязательный чеклист.

Средний — 1 ревьюер, 15—30 минут.

Низкий — автомерж, если CI зелёный.

Ввели правило: «Не мержить код, который не можешь объяснить».

Повесили чеклист из семи вопросов в шаблон PR.

Разбили большие PR. Если больше 300 строк — либо разбить, либо отложить.

Через три месяца:

Review latency для критичных PR: 2 часа. Для низких: 0.

LGTM rate: 60% (для низких — норма, для критичных — нет).

Review depth: 4 минуты на 100 строк (для критичных — 8).

Defect escape rate: упал в 2,5 раза (оценка).

Velocity: упала на 20%. Дашборд перестал быть зелёным.

СЕО спросил: «Почему velocity упала?» Тимлид ответил: «Потому что мы начали читать код». СЕО кивнул. Больше не спрашивал.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

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

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