AI-Based Development

Сопроводительный текст к презентации

Подготовлено на основе доклада Андрея Мешкова

Дата создания: 15.02.2026

Содержание

Часть 1: Теоретическая часть

Слайд 1: AI-Based Development

Здравствуйте, я Андрей Мешков, CTO и сооснователь AdGuard. Сегодня я хочу поговорить о чём-то, что касается каждого разработчика и тестировщика в этом зале — о том, как мы работаем с искусственным интеллектом. Подзаголовок этого доклада — «От AI помогает мне к AI создаёт для меня», и это отражает путь, по которому мы идём. За последние три года ИИ превратился из новинки в неотъемлемую часть того, как создаётся программное обеспечение. Но вот в чём дело — большинство из нас используют его неправильно. Мы упускаем огромные возможности для повышения производительности, потому что не обновили нашу ментальную модель того, на что способен ИИ и, что более важно, как нам нужно изменить свой рабочий процесс, чтобы в полной мере использовать его возможности. По окончании этого доклада я хочу, чтобы вы уходили отсюда с тремя вещами: чётким пониманием правильной ментальной модели работы с ИИ, практическим рабочим процессом под названием Spec-Driven Development, который вы сможете начать использовать завтра, и знанием конкретных инструментов, которые мы предоставляем здесь в AdGuard. Это не теория — мы завершим практическим воркшопом, где мы реализуем реальную функцию, используя эти методы. Давайте начнём.

Слайд 2: Why This Talk

Позвольте мне начать с того, почему этот доклад необходим. Искусственный интеллект уже коренным образом изменил способ, которым мы пишем код. Это не прогноз — это факт. GitHub сообщает, что пользователи Copilot принимают предложения, созданные ИИ, более чем в 30% их кода. Но вот в чём проблема: многие команды используют ИИ неправильно. Они относятся к нему как к красивому автодополнению или glorified StackOverflow, и когда результаты не соответствуют их ожиданиям, они делают вывод, что ИИ ещё не готов. Результат — разочарование, недоверие и потраченный впустую потенциал. Посмотрите на изображение на этом слайде — это бензопила, которая используется для резки бумаги. Инструмент невероятно мощный, но используется без понимания. Это именно то, что происходит с ИИ во многих командах разработчиков. Бензопила не сломана — человек, который её использует, просто не научился её правильно использовать. Это паттерн, который я вижу во всей индустрии, и это особенно обидно, потому что решение совсем не сложное. Оно не требует новой технологии или дорогостоящих инструментов. Оно требует изменения в том, как мы думаем об ИИ и как организуем свою работу. И это то, что мы сегодня исправим.

Слайд 4: Evolution — Section

Давайте посмотрим, как развивалась разработка с помощью ИИ за последние несколько лет. Я хочу провести вас через эту эволюцию, потому что понимание того, откуда мы пришли, помогает нам понять, где застревает большинство людей — и почему.

Слайд 5: 3 Years Ago

Три года назад наш мир выглядел вот так. Вы, разработчик, писали каждую строку кода своими руками. Ваши инструменты были ваши IDE, Google и StackOverflow. Когда вы натолкнулись на проблему, вы её искали, читали сообщения на форумах, может быть, находили фрагмент кода, который вы могли адаптировать, а затем вручную вводили его в редактор. Люди были узким местом всего процесса разработки — скорость создания программного обеспечения была напрямую ограничена тем, как быстро человек мог думать, печатать и отлаживать. Всё проходило через ваши пальцы на клавиатуре. И знаете что? Это работало нормально. Было медленно, но предсказуемо. Вы знали, что вы получите. Вы знали качество кода, потому что вы написали каждый символ самостоятельно. Вы понимали систему в деталях, потому что создавали её по кусочкам. В этом подходе была определённая удобство — прямолинейность и контроль, которые многие разработчики до сих пор упускают. Вы могли показать на любую строку кода и сказать «я точно знаю, почему это здесь». Каждое решение было преднамеренным, каждая строка — целенаправленной. Но мир был на пороге кардинальных изменений, и цепляние за эту модель вскоре стало бы конкурентным недостатком. Разработчики и команды, которые адаптировались, получили бы значительное преимущество перед теми, кто этого не сделал.

Слайд 6: 2 Years Ago

Затем, примерно два года назад, появился ChatGPT и всё изменилось. Внезапно у нас появился инструмент, который мог отвечать на вопросы по кодированию, объяснять сообщения об ошибках и генерировать фрагменты кода по требованию. Для большинства разработчиков рабочий процесс изменился одним конкретным способом: вместо поиска в Google и StackOverflow, вы спрашивали ChatGPT. Вы вставляли сообщение об ошибке, спрашивали «как мне сделать X на Python» или запрашивали фрагмент кода, а затем копировали ответ обратно в редактор. ИИ стал StackOverflow++. Это было быстрее, удобнее, и часто давало лучше направленные ответы, чем поиск через посты на форумах. Больше не нужно было просматривать пять устаревших ответов, чтобы найти тот, который действительно работает. Но вот критический момент — и я хочу, чтобы вы действительно это услышали — многие люди остаются здесь. Многие разработчики в этом зале и по всей индустрии по-прежнему используют ИИ как не более чем улучшенный поисковик. Они задают вопрос, копируют фрагмент и возвращаются к написанию кода вручную. Это как купить Tesla и использовать её как сарай для хранения. Инструмент способен на гораздо большее, но мы едва касаемся поверхности, потому что наша ментальная модель не эволюционировала вместе с технологией. Если это описывает ваш текущий рабочий процесс, не чувствуйте себя плохо — но признайте, что вы оставляете на столе огромные возможности для повышения производительности.

Слайд 7: 1 Year Ago

Примерно год назад мы вошли в эпоху agentic кодирования. Появились инструменты вроде Cursor, Windsurf и Claude Code, которые принципиально изменили отношение между разработчиком и ИИ. Вместо того чтобы задавать вопросы и копировать фрагменты, вы теперь могли дать более высокоуровневые

инструкции: «Напиши эту функцию», «Реализуй этот класс», «Отрефакторь этот файл». ИИ фактически создавал полные, работающие блоки кода — не просто фрагменты, а существенные реализации. Он мог читать ваш существующий код, понимать контекст и генерировать код, который соответствовал бы вашему проекту. ИИ перешёл от ответов на вопросы к написанию большого количества кода. Это был парадигмальный сдвиг. ИИ был уже не просто справочным инструментом — он активно создавал результаты работы. Это казалось революцией. Вы могли описать, что вы хотите на естественном языке, и получить работающий код за секунды. Многие разработчики, включая меня, были действительно взволнованы. Казалось, что производительность будет расти в небывалых темпах. Целые функции, классы, даже модули могли быть созданы из краткого описания. Но потом пришла реальность, и всё стало сложнее. Медовый месяц закончился, когда мы начали использовать эти инструменты для реальной работы в production, а не только для демонстраций и игрушечных проектов.

Слайд 8: What Happens Next?

Вот что произошло, когда люди начали полагаться на ИИ при написании существенного количества кода. Код не компилируется. Логика неправильная. Отсутствуют граничные случаи. Тесты либо слабые, либо полностью отсутствуют. Вы просили ИИ реализовать функцию, получали что-то, что выглядело правдоподобно, а потом часами отлаживали проблемы, которые ИИ не предусмотрел. Ошибки off-by-one, исключения null pointer, race conditions, неправильные предположения о модели данных — список можно продолжать. И это были не просто мелкие проблемы. Иногда весь подход был неправильным — ИИ выбрал архитектуру, которая не подходила вашей системе, или сделал предположения о ваших данных, которые были совершенно неправильны. Многие разработчики пришли к выводу на этом этапе: «ИИ всё ещё не может кодировать правильно». Вы можете видеть расстроенного разработчика на этом слайде, думающего «ChatGPT — ТУПОЙ!» И я понимаю это разочарование. Когда вы просите инструмент что-то сделать, и он многократно не справляется, естественный вывод заключается в том, что инструмент не работает. Но я хочу поставить под сомнение этот вывод, потому что он основан на фундаментальном неправильном понимании того, что пошло не так. Проблема была не в том, что ИИ неспособен — проблема была в том, как мы его использовали. Мы давали ему недостаточный контекст, неясные инструкции и не было способа проверить его собственный результат. Потом мы обвинили инструмент в результате.

Слайд 9: This Is a Mistake!

Отклонить ИИ из-за этих неудач — это ошибка, и вот почему. Вы остановились слишком рано. Подумайте о том, что фактически произошло. Вы оценили ИИ так, как если бы это был junior разработчик — вы ожидали, что он поймёт вашу базу кода, примет хорошие архитектурные решения, справится с граничными случаями и напишет production-ready код из инструкции в одну строку. Но в то же время вы относились к нему как к генератору фрагментов — вы давали ему краткое указание, ожидали идеального результата с первой попытки и отбрасывали его, когда результат был несовершенным. И критично, вы никогда не изменили свой процесс. Вы продолжали работать так же, как всегда, просто с ИИ, генерирующим код вместо вас. Те же неясные инструкции. То же отсутствие спецификаций. То же

отсутствие автоматической проверки. Это как нанять нового члена команды, дать ему нулевой onboarding, никакой документации, никакого процесса code review, а потом уволить его, когда он ошибается в первую неделю. Вы бы это не делали с человеком, почему вы это делаете с ИИ? Ключевой вывод на этом слайде вот такой: ИИ, который не справляется на первом шаге, не означает, что ИИ не способен завершить задачу. Это означает, что ваш процесс должен учитывать итерацию, контекст и обратную связь — точно так же, как и с человеческими разработчиками. ИИ нуждается в руководстве, спецификациях и циклах обратной связи для создания качественной работы. Дайте ему эти вещи, и результаты будут трансформативны.

Слайд 10: AI Agents 101 — Section

Прежде чем мы поговорим о правильном рабочем процессе, давайте убедимся, что все понимают, что такое ИИ агенты и как они работают под капотом. Это техническое понимание необходимо для того, чтобы знать, как получить лучшие результаты.

Слайд 11: What AI Agent Is?

Давайте определим, что на самом деле такое ИИ агент, потому что этот термин часто используется без точности. ИИ агент — это программа, которая имеет цикл рассуждений, работающий на базе большой языковой модели или LLM. Он читает контекст — ваш код, ваши спецификации, структуру вашего проекта. Он планирует действия на основе этого контекста. Он решает, что делать дальше. И он запрашивает или вызывает «инструменты» для выполнения этих действий. Обратите внимание, что я сказал «программа», а не «модель». LLM — это мозг, но агент — это вся система. Агент включает шаблон prompt'a, определения инструментов, управление памятью и логику оркестрации, которая всё это связывает. Cursor — это агент. Windsurf — это агент. Claude Code — это агент. Когда вы вводите «реализуй эту функцию» в Cursor, агент читает файлы вашего проекта, создаёт prompt, который включает релевантный контекст, отправляет его на LLM, интерпретирует ответ и выполняет соответствующие действия — написание файлов, запуск команд или запрос уточнений. Понимание этой архитектуры критично, потому что оно показывает вам ровно где точки рычага. Если вы хотите лучшие результаты от ИИ, вам нужно улучшить входные данные для этого цикла: лучший контекст, лучшие инструкции, лучшие инструменты. Сама LLM — это константа — вы не можете изменить, как она рассуждает. Но вы можете кардинально изменить то, о чём она рассуждает.

Слайд 12: How an AI Agent Loop Works? (Part 1)

Позвольте мне провести вас через цикл агента пошагово. Сначала программа — агент — даёт LLM prompt. Этот prompt имеет две части: инструкцию, которая это то, что вы просили сделать — «Сделай X» — и контекст, который это всё, что агент считает, что LLM должна знать для выполнения задачи. Контекст может включать релевантный код из вашего проекта, файлы документации, описания доступных инструментов и их описания, а также выходы из предыдущих вызовов инструментов и prompts. Это важно: LLM видит всё это как один большой блок

текста. У неё нет отдельной «памяти» для кода или документации или ваших инструкций. Всё это сглажено в один prompt. Шаг третий: агент рассуждает о задаче только на основе этого prompt. Это ключевое ограничение — LLM может работать только с тем, что находится в prompt. Она не знает ваши невысказанные соглашения, предпочтения вашей команды или ошибки, которые вы исправили на прошлой неделе — если эта информация не включена явно в контекст. Поэтому управление контекстом так критично, и почему инструменты вроде Cursor и Windsurf вкладывают много в интеллектуальный выбор контекста. Чем лучше контекст, тем лучше результат. Garbage in, garbage out — это более верно для ИИ, чем для любого другого инструмента в вашем наборе инструментов.

Слайд 13: How an AI Agent Loop Works? (Part 2)

Продолжая цикл. LLM отвечает одним из двух способов — либо вызовом инструмента, либо финальным ответом. Вызов инструмента означает, что LLM хочет что-то сделать: запустить тесты, прочитать файл, поискать паттерн в базе кода, написать код в файл. Финальный ответ означает, что LLM считает, что задача выполнена, и возвращает свой ответ. Когда LLM запрашивает вызов инструмента, программа — агент — выполняет его. Он запускает тест, читает файл или пишет код. Затем выход этого вызова инструмента добавляется к prompt, и всё это возвращается в LLM для следующей итерации. Этот цикл повторяется до тех пор, пока LLM не решит, что она готова, и не предоставит финальный ответ. Это agentic цикл — читай, думай, действуй, наблюдай, повтори. Это цикл — вот почему agentic кодирование настолько более мощное, чем простой чат. В чате вы задаёте один вопрос и получаете один ответ. В agentic цикле ИИ может прочитать ваш код, попробовать реализацию, запустить тесты, увидеть, что они не прошли, прочитать сообщение об ошибке, исправить код, запустить тесты снова и продолжать итерировать, пока всё не пройдёт. Это разница между тем, чтобы задать кому-то вопрос, и фактически делегировать ему задачу. Но вот загвоздка — качество этого всего цикла зависит от качества ваших первоначальных инструкций и контекста, доступного агенту. Лучшие входные данные приводят к меньшему количеству итераций и лучшим результатам.

Слайд 14: Capability vs Behavior

Это одна из самых важных концепций во всём этом докладе, так что, пожалуйста, внимательно слушайте. ИИ способен писать очень хороший код. На самом деле, он часто структурирован лучше, чем код, написанный человеком, потому что он последовательно следует паттернам, соглашениям об именовании и лучшим практикам без лени или срезаний углов. Но — и это критическое различие — способность не равняется автоматическим результатам. Просто потому, что ИИ может писать превосходный код, не означает, что он будет писать превосходный код каждый раз, когда вы просите. ИИ делает ровно то, что вы его настроили делать. Если вы дадите ему неясные инструкции, он создаст неясные результаты. Если вы дадите ему точные спецификации, он создаст точные реализации. Позвольте мне дать вам аналогию. Ferrari может делать 300 километров в час. Это невероятно способная машина. Но это не означает, что вы доберётесь до пункта назначения без карты. Без навигации, без знания дорог, без плана вашего пути, скорость Ferrari бесполезна — вы просто быстрее заблудитесь. То же самое верно и для ИИ. Способность есть. Скорость есть. Что отсутствует для большинства

разработчиков — это навигация, чёткие спецификации, определённые ограничения, структурированный рабочий процесс, который направляет способность ИИ в надёжные результаты. Эта навигация — о чём остаток этого доклада. Мы построим карту, которая превратит сырую способность ИИ в последовательный, предсказуемый, высокое качество результаты.

Слайд 16: So What's Your Role Now? — New Role

Ваша новая роль принципиально отличается от того, что это было раньше. Вместо написания кода вы определяете намерение — ясно выражая, что должна делать система и почему. Вместо борьбы с синтаксисом, вы устанавливаете ограничения — указываете границы, в которых должен работать ИИ. Вместо того, чтобы помнить API, вы разрабатываете поведение — определяете, как компоненты взаимодействуют, как обрабатываются ошибки, как данные текут через систему. И вместо отладки строка за строкой, вы проверяете результаты — оцениваете, соответствует ли результат ИИ вашим спецификациям. Посмотрите на изображение на этом слайде. Вместо того чтобы быть согнутым над клавиатурой в тёмной комнате, разработчик стоит у окна с естественным светом, рисует блок-схемы на стекле, разрабатывает системы. На столе стоит чашка чая. Это новая осанка разработки программного обеспечения. Вы архитектор, дизайнер, рецензент. Вы работаете на более высоком уровне абстракции, а ИИ обрабатывает детали реализации. Этот сдвиг требует различных навыков. Вам нужно быть лучше в коммуникации — объяснять, что вы хотите, точно. Лучше в спецификации — писать ясные, однозначные требования. Лучше в системном мышлении — понимать, как компоненты взаимодействуют и что может пойти не так. Разработчики, которые преуспевают в этой новой модели, не будут самыми быстрыми печатающими — они будут самыми ясными мыслящими. И это гораздо более награждающий способ работать.

Слайд 17: The Uncomfortable Truth

Вот неудобная истина, которую многие разработчики отрицают: «Я больше не пишу код. И вы тоже не должны». Я знаю, что это звучит провокационно, и это намеренно. Когда я говорю «я не пишу код», я не имею в виду, что я не участвую в разработке программного обеспечения. Я имею в виду, что моя роль изменилась. Я разрабатываю системы. ИИ пишет код. Я проверяю результаты. Посмотрите на мем на этом слайде — вы вероятно видели это изображение раньше. Все стоят вокруг, наблюдая, как один человек выполняет реальную работу. В старом мире разработчик был человеком в яме, копающим. В новом мире вы человек, который направляет работу — вы тот, кто решил где копать, как глубоко, и для чего нужна яма. Некоторые из вас могут чувствовать себя некомфортно с этим. Написание кода — это часть вашей личности. Это то, что вы тренировались делать, то, что вам нравится. Но я бы утверждал, что на самом деле то, что вы тренировались делать, это решать проблемы. Код был просто средой. Теперь у вас есть более мощная среда, и ваши навыки решения проблем более ценны, чем когда-либо. Разработчики, которые сопротивляются этому сдвигу — которые настаивают на том, чтобы писать каждую строку вручную — будут отставать. Не потому что они плохие в кодировании, а потому что они добровольно себя ограничивают в мире, где ИИ может выполнить реализацию в десять раз быстрее. Примите сдвиг. Ваш инженерный ум — это ценная часть — не ваша скорость печати.

Слайд 18: System Design Is Now the Main Job

Если ИИ изменил способ написания кода, что он не изменил? Он не изменил, что такое программное обеспечение. Программное обеспечение по-прежнему о решении проблем. Это по-прежнему о управлении сложностью, управлении состоянием, работе с отказами и служении пользователям. ИИ изменяет уровень реализации, но уровень проектирования полностью ваш. Ваши основные обязанности теперь: разложить проблему — разбить большие, сложные задачи на более мелкие, хорошо определённые куски, которые ИИ может реализовать независимо. Определить границы — указать ясные интерфейсы между компонентами, чтобы ИИ знал, где заканчивается один модуль и начинается другой. Выбрать абстракции — решить на правильные структуры данных, паттерны проектирования и архитектурные подходы для вашей конкретной ситуации. Определить поток данных и режимы отказа — указать, как данные движутся через систему и что происходит, когда что-то идёт не так. Вот цитата, которую я хочу, чтобы вы запомнили из всего этого доклада: «Если дизайн системы плохой, ИИ верно реализует плохую систему». ИИ — отличный исполнитель. Он будет делать именно то, что вы просите. Если ваша архитектура неправильна, ИИ построит эту неправильную архитектуру красиво, с чистым кодом, хорошими именами переменных и комплексными тестами, которые все пройдут — для неправильной системы. Качество программного обеспечения определяется качеством вашего дизайна, а не качеством кода. ИИ обрабатывает качество кода. Качество дизайна на вас.

Слайд 19: Rule #1: You Must Know How to Do It by Hand

Это приводит к первому фундаментальному правилу работы с ИИ: вы должны знать, как это делать вручную. Это может казаться противоречивым — я только что сказал вам перестать писать код, а теперь я говорю вам, что вам нужно знать как. Но нет противоречия. Если вы не понимаете, как пишется код, как строятся системы и как появляются ошибки, то вы не можете разработать систему для ИИ реализации. Вы не можете проверить, правилен ли результат ИИ. И вы не можете сказать «почти правильно» от «опасно». Подумайте об этом: если ИИ пишет функцию, которая выглядит разумно, но имеет тонкую race condition, как вы узнаете? Если ИИ выбирает $O(n^2)$ алгоритм, где существует $O(n \log n)$ решение, как вы это поймаете? Если запрос базы данных ИИ отсутствует индекс подсказки и превысит время ожидания в production, как вы это заметите? Вам нужны глубокие технические знания — не чтобы писать код самостоятельно, а чтобы быть эффективным рецензентом и дизайнером. Без понимания вы просто слепо принимаете то, что создаёт ИИ, и надеетесь, что это работает. Цитата внизу этого слайда резюмирует это: «ИИ не заменяет понимание». ИИ заменяет печать. Он заменяет поиск синтаксиса. Он заменяет boilerplate. Но он абсолютно не заменяет понимание, которое вам нужно для создания хорошего программного обеспечения. Junior разработчики, которые думают, что они могут пропустить обучение основам, потому что «ИИ это обработает», делают опасную ошибку.

Слайд 20: Rule #2: Vague Intent Is Useless

Правило номер два: неясное намерение бесполезно. Это единственная самая частая ошибка, которую я вижу, когда разработчики работают с ИИ агентами. Они дают инструкции вроде «Сделай это быстрее» или «Исправь баг» или «Улучши производительность» и потом удивляются, почему результаты посредственные. Позвольте мне объяснить, почему это не срабатывает. ИИ не может следовать неясным инструкциям, потому что неясные инструкции не содержат достаточно информации для определения правильного действия. «Сделай это быстрее» — быстрее в сравнении с чем? Быстро в смысле latency, throughput, использования памяти? Быстро для каких операций? Что приемлемо компромиссы? Хорошая инструкция специфична и измерима. Вместо «Сделай это быстрее» вы должны: определить, что означает «быстрее» — например, p95 latency менее 10 миллисекунд. Определить ограничения и компромиссы — например, мы можем использовать больше памяти, но не более 500MB на процесс. И это критично — объяснить, как ИИ может это проверить. Unit тесты здесь очень важны. Если вы даёте ИИ ясную цель производительности и тест, который проверяет эту цель, агент может итерировать, пока тест не пройдёт. Без теста, агент не имеет способа узнать, действительно ли его изменения что-то улучшили. Думайте об этом так: вы не скажете человеческому разработчику «сделай это быстрее» и не уйдёте. Вы обсудили бы требования, согласились бы на метрики и определили критерии приёмки. ИИ нуждается в том же уровне конкретности — фактически, ему нужно больше, потому что он не может читать между строк или делать предположения на основе вашего организационного контекста.

Слайд 23: Spec-Driven Development — Section

Теперь давайте поговорим о Spec-Driven Development — SDD — методологии, которая связывает всё воедино и заставляет ИИ-ассистируемую разработку действительно работать. Это самая практичная часть всей презентации.

Слайд 24: What Is Spec-Driven Development?

Spec-Driven Development, или SDD, это фундаментальный сдвиг в том, как вы взаимодействуете с ИИ. Вместо того чтобы сказать «Напиши мне функцию, которая делает X» — что неясная инструкция на уровне реализации — вы пишете спецификацию. Спецификация определяет три вещи. Первое: что должна делать система — функциональные требования, поведение, ожидаемые выходы для заданных входов. Второе: ограничения, не-цели и граничные случаи — что система не должна делать, что вне области видимости и что происходит в необычных ситуациях. Третье: критерии приёмки — как вы знаете, что реализация правильна, измеримые результаты, которые определяют успех. После того как у вас есть эта спецификация, ИИ берёт на себя реализацию. Он реализует функцию в соответствии со спецификацией. Он итерирует, когда что-то не работает. Он делает рефакторинг, когда реализация нуждается в очистке. Всё это руководствуется спецификацией, которую вы написали. Ключевой вывод в том, что спецификация — это интерфейс между человеческим намерением и ИИ исполнением. Без неё ИИ угадывает. С ней, ИИ имеет ясную цель. Думайте о SDD как о Test-Driven Development, но на более высоком уровне. В TDD вы пишете тест первым, а потом реализуете до тех пор, пока тест не пройдёт. В SDD вы пишете спецификацию первой, а потом ИИ реализует до тех пор, пока спецификация не будет удовлетворена. Спецификация более комплексна, чем тест — она включает

не просто критерии проверки, но также решения проектирования, ограничения и контекст, который руководит реализацией.

Слайд 26: Why SDD Is Mandatory with AI

Позвольте мне обосновать, почему SDD — это не просто хорошо иметь, это обязательно при работе с ИИ. Без спецификаций вот что происходит. ИИ угадывает — он делает предположения о том, что вы хотите, на основе паттернов, которые он видел в обучающих данных. Иногда эти предположения правильны, но часто они неправильные. Вы отлаживаете предположения — вы тратите своё время на выяснение, почему ИИ сделал конкретный выбор и соответствует ли это тому, что вам действительно нужно. Доверие исчезает — после нескольких раундов ИИ неправильно угадывает, вы перестаёте ему доверять и возвращаетесь к написанию кода самостоятельно. Вся работа была напрасна. Теперь, со спецификациями, всё меняется. ИИ имеет намерение — он знает, что вы хотите, потому что вы это явно указали. ИИ самокорректируется — когда что-то не соответствует спецификации, агент знает, что ему нужно попробовать другой подход. ИИ делает рефакторинг безопасно — он может делать изменения с уверенностью, потому что спецификация определяет, что должно оставаться неизменным. ИИ объясняет решения — он может ссылаться на спецификацию, чтобы обосновать свои выборы. Вот фраза, которую я хочу, чтобы вы запомнили: «Спецификации — это как вы программируете ИИ». Когда вы пишете спецификацию, вы не пишете документацию — вы программируете. Спецификация — это ваш исходный код, а ИИ — это компилятор, который превращает его в реализацию. Чем лучше ваша спецификация, тем лучше результат. Неясная спецификация создаёт неясный код, точно так же, как неясный исходный код создаёт непредсказуемое поведение.

Слайд 28: Example of SDD: Business Requirements

Позвольте мне провести вас через конкретный рабочий процесс SDD, шаг за шагом. Первый prompt для бизнес-требований. Вы можете рассказать ИИ агенту: «Пожалуйста, напиши спецификацию функции в спецификацию specs/current/spec.md для добавления функции DNS защиты к приложению. Определи пользовательские истории, идентифицируй граничные случаи, определи функциональные требования». Обратите внимание, как специфичен этот prompt. Вы рассказываете ИИ, ровно где писать спецификацию — конкретный файл путь в вашем проекте. Вы рассказываете ему что включить — пользовательские истории, граничные случаи, функциональные требования. И вы даёте ему доменный контекст — функция DNS защиты для приложения. ИИ создаст комплексный документ спецификации, опираясь на его знание DNS, безопасности и вашей базе кода. Потом идёт критический шаг: вы проверяете и уточняете спецификацию с ИИ. Вы ищете логические проблемы — содержит ли спецификация противоречия или пробелы? Вы ищите неясные заявления — есть ли требования, которые могут быть интерпретированы несколькими способами? И вот практический совет: вы можете даже начать Pull Request на этом этапе. Да, до любого кода написано. PR для спецификации позволяет вашей команде проверить дизайн, предложить изменения и достичь консенсуса до того, как реализация начнётся. Это невероятно мощно, потому что это сдвигает review upstream — вы проверяете решения проектирования, когда они дёшевы для изменения, а не детали реализации, когда они дороги для изменения.

Слайд 32: What Is AGENTS.md?

AGENTS.md — это файл инструкций проекта-уровня для ИИ агентов. Думайте о нём как о README, но специально для ИИ. В то время как README.md рассказывает человеческим разработчикам как настроить и использовать проект, AGENTS.md рассказывает ИИ агентам как работать в этом проекте. Как минимум, он описывает: как структурирован репозиторий — что находится в каждом директории, где живёт различные типы кода, как проект организован. Как система предполагается работать — архитектура, поток данных, ключевые решения дизайна и рассуждения за ними. Архитектурные границы и ограничения — какие паттерны требуются, какие паттерны запрещены, какие соглашения должны быть соблюдены. Разработка и workflow тестирования — как запустить тесты, как создать проект, как проверить изменения. Этот файл понимается всеми основными ИИ агентами — Cursor, Windsurf, Claude Code, OpenAI Codex — они все ищут и уважают AGENTS.md или подобные файлы конфигурации. Вы можете найти спецификацию и лучшие практики на agents.md. Наличие хорошо написанного AGENTS.md — это единственное наиболее важное действие, которое вы можете сделать, чтобы улучшить производительность ИИ на вашем проекте. Без него, каждый ИИ сеанс начинается с нуля — агент должен переоткрывать ваши соглашения, вашу структуру и ваш рабочий процесс. С ним, агент начинает со всего контекста, который ему нужен, чтобы быть продуктивным сразу.

Слайд 37: AI Without Guardrails Is Chaos

ИИ невероятно продуктивен — и это ровно то, что делает его опасным без guardrails. ИИ может писать много кода. В одном сеансе ИИ агент может создать сотни или даже тысячи строк кода. Он может агрессивно делать рефакторинг — переименовывать переменные, перестраивать модули, перемещать файлы. И он может изменять вещи, которые вы не просили — «улучшая» код, который работал хорошо, «очищая» паттерны, которые вы намеренно выбрали или «исправляя» вещи, которые не были сломаны. Без guardrails, regressions просачиваются. ИИ модифицирует функцию, чтобы исправить один баг и случайно ломает другое поведение. Качество становится непоследовательным — некоторые части базе кода следуют одному стандарту, другие части следуют различным паттернам, потому что различные ИИ сеансы сделали различные выборы. Вот цитата, которую я хочу, чтобы вы internalize: «Чем быстрее движется ИИ, тем сильнее должны быть guardrails». Это контринтуитивно. Вы можете думать, что более быстрая разработка означает вы должны удалить барьеры и позволить ИИ свободно работать. Противоположное верно. Когда человек пишет код медленно, они естественно ловят свои собственные ошибки через механический процесс печати и чтения. ИИ не имеет этого естественного тормоза. Он генерирует код на скорости машины, что означает ошибки также происходят на скорости машины. Без автоматических проверок, эти ошибки накапливаются быстрее, чем любой человек может их проверить. Скорость без безопасности — это просто хаос с добрыми намерениями.

Слайд 38: The Minimum Viable Guardrails

Каждый ИИ-управляемый проект должен иметь эти минимальные guardrails. Никаких исключений, никаких переговоров. Linters и formatters — они ловят

нарушения стиля, обычные ошибки и непоследовательности автоматически. ИИ-генерированный код должен пройти те же lint правила, что и код, написанный человеком. Это ваша первая линия защиты против небрежного или нестандартного результата. Unit тесты — это необсуждаемо. Я не могу достаточно это подчеркнуть. Unit тесты — это основной механизм, с помощью которого ИИ проверяет свою собственную работу. Когда ИИ пишет код и запускает тесты, он получает немедленную обратную связь о том, правилен ли его результат. Без тестов, ИИ летает вслепую — генерируя код и надеясь, что это работает. Markdownlint для спецификаций и правил — если вы делаете SDD, ваши спецификации являются важными артефактами. Они должны быть чистыми, хорошо отформатированы и согласованы. Markdownlint обеспечивает, что ваши спецификации поддерживают профессиональный стандарт. Pre-commit hook рекомендуется — pre-commit hook, который запускает linting, форматирование и тесты перед каждым коммитом, обеспечивает, что ничего не просочится. Это сеть безопасности, которая ловит проблемы до того как они входят в вашу базу кода. И вот два абсолютных требования. ИИ агент ДОЛЖЕН быть способен запустить все эти проверки из терминала. Если ваш linter требует GUI, ИИ не может его использовать. Если ваши тесты требуют ручных шагов установки, ИИ не может их запустить. ИИ агент ДОЛЖЕН быть способен создать ваш проект из терминала. Если ИИ не может создать и проверить его изменения, это генерирует код в темноте. Задокументируйте эти команды ясно в вашем AGENTS.md.

Часть 2: Воркшоп

Слайд 46: Workshop — Section

Теперь давайте применим всю эту теорию на практику. Мы будем использовать SDD для реализации реальной функции в реальном проекте, чтобы вы могли увидеть ровно как выглядит этот рабочий процесс в действии.

Слайд 47: Workshop

Для нашего воркшопа, мы будем использовать PLAT/sdd framework для реализации функции в реальном проекте — slack-bitbucket интеграционный сервис. Вы можете получить доступ к репозиторию проекта по ссылке, показанной на этом слайде. Этот проект был специально выбран, потому что он демонстрирует всё, что мы обсуждали сегодня. Он имеет хорошо поддерживаемый AGENTS.md, который определяет структуру проекта, кодовые соглашения, требования тестирования и полный рабочий процесс разработки. Он использует подход SDD для разработки функции, со спецификациями и техническими планами, хранящимися рядом с кодом в контроле версий. И он имеет надлежащие guardrails на месте — linting, автоматизированное тестирование и проверка CI. Это не toy пример или demo проект. Мы будем работать над production базой кода с реальными требованиями, реальными ограничениями и реальными последствиями если мы что-то сделаем неправильно. Цель этого воркшопа — продемонстрировать, что SDD работает на практике, не просто в теории. Вы увидите, что первоначальная инвестиция в написание спецификаций и планов на самом деле экономит время в целом, потому что фаза реализации становится намного более гладкой, когда ИИ имеет ясные, подробные инструкции для следования.

Слайд 48: What Will Be Showcased

В воркшопе мы продемонстрируем четыре отличные рабочие процесса, которые охватывают самые обычные сценарии разработки. Первое, полный SDD рабочий процесс для большой функции. Мы пройдём через полный цикл: написание бизнес-спецификации, создание технического плана реализации, имея ИИ реализовать план и проверить результаты. Вы увидите как каждый шаг строится на предыдущем и как спецификации руководят ИИ в сторону правильной реализации без гаданий. Второе, быстрый SDD рабочий процесс для небольшого улучшения. Не всё нуждается в полном четырёх-шаговом процессе, и знание когда использовать быстрый рабочий процесс против полного рабочего процесса является важным навыком. Мы покажем как обработать меньшие изменения эффективно. Третье, актуализирование документации проекта. Один из самых практичных и недооценённых использований ИИ — это держание документации в актуальном состоянии. Мы покажем как использовать ИИ для обновления AGENTS.md и других документов проекта на основе последних изменений кода, гарантируя документация остаётся в синхронизации с базой кода. Четвёртое, используя ИИ рецензента. Мы создадим PR из наших изменений воркшопа и попросим ИИ рецензента проверить нашу работу. Вы увидите как он предоставляет обратную связь, ловит проблемы и обеспечивает согласованность с установленными стандартами проекта. Этот практический раздел — это где всё из доклада

собирается вместе в полный, практичный рабочий процесс.