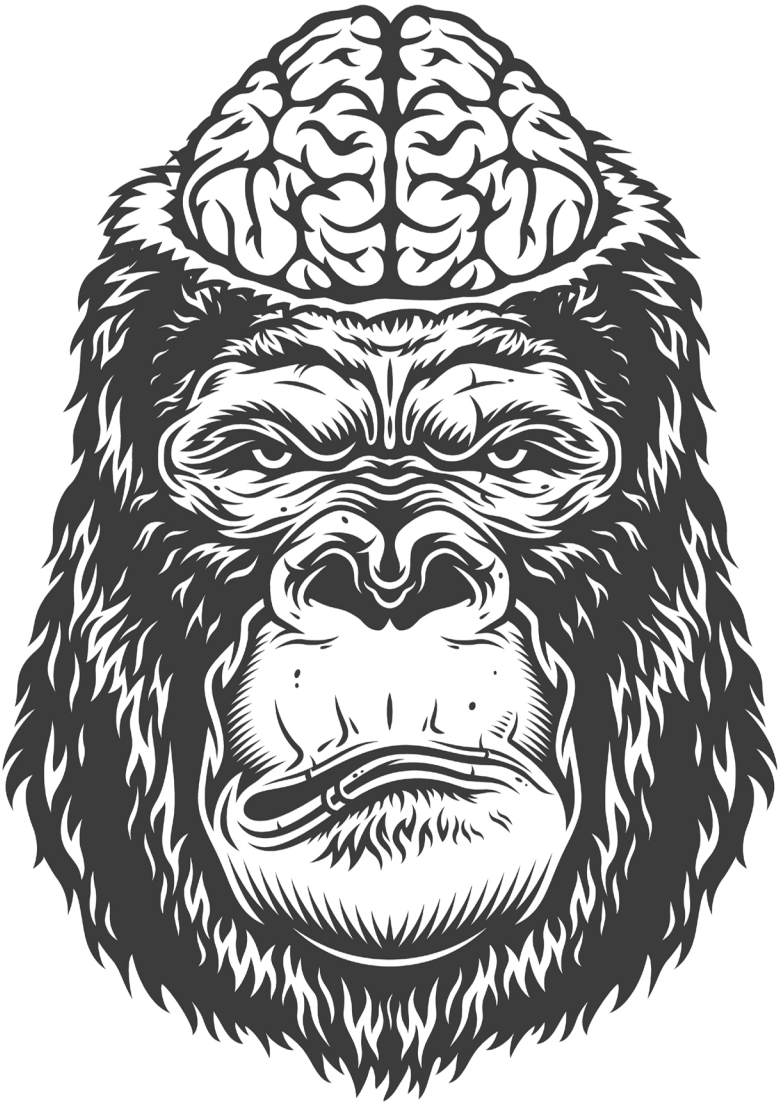
Фань Вэньцзе, Ли Цзяньчао, Янь Сяофань, Чжан Бинь



Вайб-программирование

Разработка кода в эпоху искусственного интеллекта

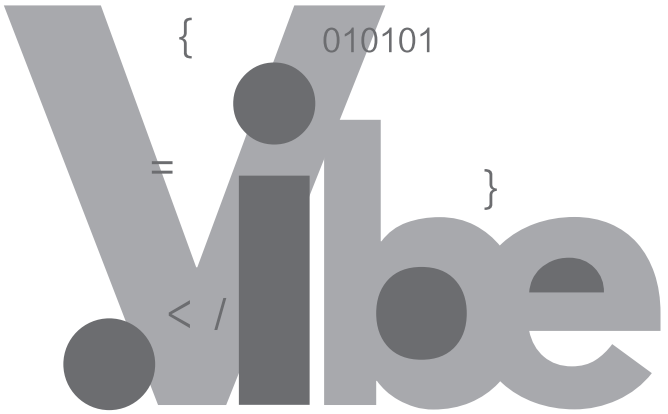
ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Ван Вэньцзе (Tecvan), Ли Цзяньчао (Isboyjc),
Ю И (Янь Сяофань), Чжан Бинь (Captain)

Вайб-программирование. Разработка кода в эпоху искусственного интеллекта

Vibe编程

探索AI时代编程新范式



范文杰（Tecvan） 李建超（Isboyjc） 尤毅（言萧凡） 张斌（Captain）◎ 著

人 民 邮 电 出 版 社
北 京

Вайб-программирование. Разработка кода в эпоху искусственного интеллекта

Ван Вэньцзе (Tecvan), Ли Цзяньчао (Isboyjc),
Ю И (Янь Сяофань), Чжан Бинь (Captain)

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Москва, 2026

УДК 004.42:004.8
ББК 32.973
В14

Ван Вэньцзе (Tecvan), Ли Цзяньчао (Isboyjc), Ю И (Янь Сяофань), Чжан Бинь (Captain)

В14 Вайб-программирование. Разработка кода в эпоху искусственного интеллекта / пер. с кит. В. И. Бахура. – М.: ДМК Пресс, 2026. – 246 с.: ил.

ISBN 978-5-93700-440-6

В книге подробно рассматриваются прикладные аспекты использования новейшего подхода к разработке ПО – так называемого вайб-программирования (Vibe Coding), также известного как «интуитивное программирование». Основная идея книги заключается в том, что интеграция ИИ должна представлять собой не усовершенствование существующих процессов, а принципиально новый способ использования моделей, по значимости не уступающий функциональному программированию.

Издание адресовано профессиональным разработчикам для совершенствования навыков построения архитектуры приложений, менеджерам по продуктам, UI/UX-дизайнерам и широкой аудитории как руководство по использованию ИИ для воплощения личных идей в реальные приложения.

УДК 004.42:004.8
ББК 32.973

Дизайн обложки разработан с использованием ресурса magnific.com

All rights reserved. First published in the Chinese language under the title 978-7-115-67971-0 Vibe编程：探索AI时代编程新范式

Russian translation rights arranged with Posts and Telecom Press Co., Ltd. through Media Solutions, Tokyo Japan (info@mediasolutions.jp)

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (кит.) 978-7-11567-971-0
ISBN (рус.) 978-5-93700-440-6

Copyright© 2025 Posts and Telecom Press Co., Ltd.
© Оформление, издание, перевод, ДМК Пресс, 2026

Оглавление

https://t.me/it_boooks/2

Предисловие от издательства 9

Об авторах 10

Предисловие 11

Для кого предназначена эта книга 12

Введение 13

Глава 1. Предпосылки 15

1.1. Введение в вайб-программирование 15

1.1.1. Воплощение идей 17

1.1.2. Креативность превыше технологий 18

1.2. Переход от командного стиля к целевому 20

1.2.1. Обзор императивного программирования 21

1.2.2. От команд к намерениям 22

1.2.3. Сравнение методов разработки 23

1.3. Основы вайб-программирования 25

1.3.1. Становление технологий больших языковых моделей 25

1.3.2. Стимулирующие факторы рыночного спроса 26

1.3.3. Ключевые проблемы разработчиков 28

1.4. Заключение 29

Глава 2. Эволюция способов программирования 30

2.1. Эволюция языков программирования 30

2.1.1. Общение с машиной 31

2.1.2. От двоичного кода к символьному языку 32

2.1.3. Возникновение языков программирования высокого уровня 36

2.1.4. Прорыв в области структурного и объектно-ориентированного программирования 39

2.1.5. Разнообразие современных моделей программирования 42

2.2. Эволюция взаимодействия при программировании 49

2.2.1. Зачатки программирования на физических носителях 49

2.2.2. Переход от строчного редактирования к полноэкранному взаимодействию 50

2.2.3. Эпоха интеграции: объединение редактирования, компиляции и отладки 52

2.2.4. Кросс-платформенная экосистема и реформа архитектуры плагинов 53

2.3. Возникновение разработки с минимальным кодом и без кода 54
2.3.1. Теоретическая система полноты по Тьюрингу 55
2.3.2. Ранние технологические исследования 56
2.3.3. Формирование концепции и первые практические шаги 58
2.3.4. Этапы развития рынка 58
2.3.5. Ограничения традиционных платформ с минимальным кодом и без кода 59
2.3.6. Использование ИИ в платформах с низким и нулевым уровнями кодирования 61
2.4. Заключение 62

Глава 3. Экосистема приложений вайб-программирования 63

3.1. Программирование с универсальными большими языковыми моделями 63
3.1.1. Вопросы и ответы с использованием больших языковых моделей 63
3.1.2. Большие языковые модели и их потенциал в программировании 65
3.1.3. Преимущества и недостатки диалогового режима программирования 66
3.2. Программирование с помощью IDE 66
3.2.1. Интеграция ИИ в плагины IDE 66
3.2.2. IDE с нативной интеграцией ИИ 68
3.2.3. Сравнение ИИ-интеграции в IDE 69
3.3. Сквозное программирование с помощью агентов 70
3.3.1. Концепция сквозного программирования с помощью агентов 70
3.3.2. Решения для сквозного программирования агентов 71
3.3.3. Механизм работы и системная архитектура 73
3.4. Будущее прикладных форм 75
3.4.1. Новые формы применения 75
3.4.2. Формы использования и сегментация пользователей 76
3.5. Заключение 78

Глава 4. Вайб-программирование: сценарии и практические примеры 79

4.1. Анализ сценариев применения 80
4.1.1. Быстрое создание прототипов продуктов 80
4.1.2. Возникновение «массового программирования» 82
4.1.3. Инструмент для новичков и препятствия для продвинутых 82
4.1.4. Автоматизация внутренних процессов компании: повышение эффективности и проблемы интеграции 83
4.2. Подробный анализ практических примеров 84
4.2.1. Примеры успеха независимых разработчиков 84
4.2.2. Опыт практического использования стартапами 88
4.2.3. Трансформация крупных предприятий 89
4.2.4. Адаптация и инновации сообщества Open Source 90
4.3. Заключение 92

Глава 5. Лучшие практические методики 93
5.1. Методики разработки промптов 93
5.1.1. Почему важны хорошие промпты 94
5.1.2. Основные принципы разработки промптов 95
5.1.3. Пример оптимизации промпта 98
5.1.4. Набор практических шаблонов подсказок 99
5.1.5. Настройка подсказок для Cursor 100
5.2. Планирование требований 108
5.2.1. Анализ требований 108
5.2.2. Составление документа с требованиями к продукту 111
5.2.3. Выбор ИИ-ориентированного технологического стека 114
5.2.4. Использование ИИ для генерации технических требований 117
5.3. Рецензирование и оптимизация кода 119
5.3.1. Ограничения ИИ 120
5.3.2. Распространенные дефекты качества 120
5.3.3. Некачественный код может привести к провалу проекта 125
5.3.4. Руководство по рецензированию кода в эпоху ИИ 126
5.3.5. Создание системы рецензирования кода 143
5.4. Инженерный подход 149
5.4.1. Введение в инженерию 150
5.4.2. Легкая инженерная система для вайб-программирования 151
5.5. Заключение 160

Глава 6. Практические примеры 161
6.1. Настройка среды 161
6.1.1. Подготовка инструментов 161
6.1.2. Структура проекта 169
6.2. Анализ требований проекта 171
6.2.1. Систематизация документации по требованиям 171
6.2.2. Составление документации по техническому проектированию 176
6.2.3. Составление документации плана реализации проекта 179
6.3. Разработка бэкенда 183
6.3.1. Принципы реализации 183
6.3.2. Разработка программы бэкенд-сервиса 185
6.3.3. Рецензирование кода 187
6.3.4. Тестирование интерфейсов 188
6.3.5. Доработка и дополнение функциональных возможностей 189
6.4. Разработка веб-систем 190
6.4.1. Подход к реализации 191
6.4.2. Разработка веб-страниц 193
6.4.3. Рецензирование кода 195
6.4.4. Вывод веб-страницы с помощью ИИ в соответствии с ожиданиями 196
6.4.5. Вызов реальных серверных интерфейсов 203
6.5. Развертывание приложения 206
6.5.1. Понимание логики развертывания кода 207

6.5.2. Развертывание приложения на Vercel ... 211
6.5.3. Реализация непрерывного развертывания с помощью GitHub Actions ... 217
6.6. Заключение ... 222

Глава 7. Ограничения и проблемы ... 224
7.1. Точка зрения потребителя ... 225
7.1.1. Обычный потребитель ... 225
7.1.2. Профессиональные разработчики ... 227
7.2. Точка зрения разработчика ... 229
7.2.1. Неудобное позиционирование продукта ... 229
7.2.2. Проблема издержек ... 230
7.2.3. Различные сценарии взаимодействия с потребителем ... 231
7.3. Революция мышления разработчиков и доступность технологий для всех ... 232
7.3.1. Смена мышления разработчиков старого поколения ... 232
7.3.2. Ключевые навыки разработчика новой эпохи ... 233
7.3.3. Глубина в одной области и охват в других: расцвет междисциплинарных навыков ... 235
7.3.4. Технологическое равенство для всех ... 236
7.3.5. Корректировка карьерного роста и образовательных траекторий ... 238
7.4. Заключение ... 239

Послесловие ... 241

Предметный указатель ... 242

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторах

Фан Вэньцзе (Tecvan). Ведущий фронтенд-архитектор одной из крупных компаний, руководитель официального аккаунта Tecvan, автор руководства «Основные принципы и практическое применение Webpack 5» для платформы Juejin, ведущий подкаста Langshuo. Специализируется в области инженерных аспектов фронтенд-разработки и практического применения ИИ в программировании. На протяжении многих лет выступает в качестве лектора на мероприятиях D2 Terminal Technology Conference, GIAC (Global Internet Architecture Conference) и Frontend Zaozao Chat.

Ли Цзяньчао (Isboyjc). Опытный разработчик полного цикла, вайб-программист. Автор канала «Несерьезный фронтенд» на стриминговой платформе Bilibili. Специализируется на веб-разработке, проектировании системных архитектур, больших языковых моделях и создании приложений с ИИ-агентами. Активно занимается просветительской работой в области технологий искусственного интеллекта.

Ю И (Янь Сяофань). Старший фронтенд-архитектор команды по трансграничной электронной коммерции. Автор пособий «Проекты NestJS на практике» и «Практики DevOps на основе Node» на платформе Juejin. Специализируется на построении систем фронтенд-инжиниринга и проектировании архитектуры клиентских приложений. Выступал в качестве приглашенного эксперта на технологическом салоне ByteDance и в качестве лектора на конференции Rare Earth Juejin.

Чжан Бинь (Captain). Опытный координатор сообщества разработчиков, ведущий подкаста Langshuo. Основатель базы знаний AGI Juejin. Активный популяризатор вайб-программирования.

Предисловие

Программирование никогда не стоит на месте и постоянно развивается на волне технического прогресса. От механических перфокарт до структурного программирования, от объектно-ориентированного подхода до функциональной парадигмы – каждая из этих трансформаций изменяла способ взаимодействия человека с машиной. Сегодня перед нами предстала совершенно новая революция. **Вайб-программирование** – это не просто технологическая тенденция, а скорее переход к новому способу мышления, который превращает программирование из «императивного» набора точных инструкций в «интенциональное» выражение вдохновения. Эта концепция, впервые предложенная Андреем Карпатым (Andrej Karpathy) и сопровождаемая расцветом технологий генеративного ИИ, с невиданной скоростью меняет подход разработчиков к созданию ПО наряду с самой структурой их образа мышления.

Суть вайб-программирования заключается в раскрепощении и расширении потенциала разработчиков. Благодаря устойчивому прогрессу в области технологий больших языковых моделей разработчикам больше не нужно вводить код строка за строкой – вместо этого они могут выражать свои намерения на естественном языке и вместе с ИИ проходить весь путь от идеи до ее воплощения. Появление таких инструментов, как ChatGPT, Claude и Cursor, делает программирование доступным не только для избранных, но и для широкой аудитории, снижая технический порог вхождения и стимулируя креативность в различных дисциплинах. От независимых разработчиков, быстро создающих прототипы продуктов с помощью ИИ, до предприятий, использующих интеллектуальные системы для автоматизации сложных рабочих процессов, вайб-программирование меняет экосистему разработки ПО благодаря своей эффективности и простоте.

Путь этой трансформации не будет легким. Проблемы отладки кода, сгенерированного ИИ, ограничения производительности, угрозы безопасности, а также сложности адаптации к сложным проектам требуют от нас осознания того, что за технологическими удобствами скрывается необходимость тщательного анализа и освоения новых профессиональных навыков. Разработчики будущего будут заниматься не столько написанием кода, сколько постановкой задач, проектированием систем и интеграцией различных областей знаний. Вайб-программирование трансформирует инструменты и процессы программирования, но также подталкивает к «революции мышления» с приоритетом всестороннего подхода, способствуя развитию широкого кругозора и открывая эпоху технологического равенства.

Не исключено, что в будущем вайб-программирование полностью устранит границы между техническими и нетехническими сферами, давая возможность стать творцами всем желающим. Это не просто обновление инструментов, а новая глава в истории сотрудничества человека и машины. Данная книга

призвана познакомить вас с сутью этой трансформации, исследовать ее технические основы, сценарии применения и практические способы реализации, а также честно рассказать о ее ограничениях и вызовах. Надеюсь, вы найдете в ней вдохновение и в симфонии кода и замыслов обретете свой собственный ритм программирования.

Чжан Бинь (Captain)
Июнь 2025 года

Для кого предназначена эта книга

Эта книга подходит как для профессиональных разработчиков с определенным техническим багажом, так и для менеджеров по продукту, UI/UX-дизайнеров и даже обычных пользователей, которые хотят разобраться в тенденциях программирования с использованием ИИ и заниматься инновациями с низким порогом входа.

Если вы профессиональный разработчик и хотите научиться эффективно и системно использовать инструменты программирования с ИИ, эта книга поможет вам сформировать концептуальную основу и избежать распространенных ошибок. Если вы являетесь менеджером по продукту или UI/UX-дизайнером, при этом имеете некоторое представление о программировании, но ваши базовые знания в этой области оставляют желать лучшего, эта книга поможет вам открыть для себя новые возможности в области разработки продуктов и повысить эффективность работы с помощью вайб-программирования. Даже если вы не занимаетесь технической работой, а просто интересуетесь влиянием ИИ на методы работы и творчества, эта книга поможет вам понять базовую логику происходящих изменений.

Эпоха вайб-программирования только начинается, и вне зависимости от ваших предпочтений – принять участие, выждать, критиковать или просто разобраться – вам обязательно нужно ознакомиться с реальной сутью этой технологической революции.

Введение

Цифровая экономика набирает обороты по всему миру. Начиная с 2023 года инновации в области технологий программирования с использованием искусственного интеллекта (ИИ) стали бесспорным движущим фактором модернизации промышленности. Вайб-программирование является новой парадигмой кодирования, в основе которой лежит модель разработки «что видишь, то и получаешь». Благодаря диалоговому взаимодействию разработчики могут быстро преобразовывать свои идеи в рабочий код, что значительно снижает барьер технической сложности для начинающих. Все больше предприятий, профессиональных разработчиков и даже обычных пользователей начинают знакомиться и использовать такие ИИ-сервисы для программирования, как GitHub Copilot, Cursor, Claude Code и Devin. Вайб-программирование постепенно выходит за пределы технического сообщества и формирует новую коммерческую экосистему.

Ярким свидетельством стремительного развития вайб-программирования являются данные статистики. По прогнозам Polaris Research, к 2032 году рынок ИИ-инструментов для программирования вырастет до $271,7 млрд и станет важнейшей инфраструктурой в сфере разработки ПО. Ожидается, что уже в 2025 году объем мирового рынка программных средств с ИИ-поддержкой будет превышать $250 млрд, а годовой совокупный темп роста сохранится на уровне более 35 %. С прикладной точки зрения, до 75 % пользователей платформы Replit никогда не писали программ до знакомства с ней, но с помощью ее ИИ-функции ghostwriter смогли успешно воплотить свои идеи в реальные продукты.

Кроме того, многие ведущие компании и стартап-команды проявляют острую прозорливость и активно переходят на вайб-программирование. Генеральный директор Google как-то сообщил, что более 25 % нового кода в компании генерируется искусственным интеллектом. Это свидетельствует о значительной роли ИИ в обширной кодовой базе Google, который обеспечивает ее эффективную работу. В известном стартап-инкубаторе из Кремниевой долины Y Combinator также сообщили, что среди участников мероприятия Winter Demo Day 2025 (W25 Demo Day) четверть стартап-команд используют ИИ для генерации 95 % своего кода. Благодаря ИИ они выходят на рынок с минимальными затратами, обеспечивают быструю итерацию продуктов и занимают выгодные позиции в условиях жесткой конкуренции.

Эта книга написана на основе личного опыта авторов, которые стали свидетелями данной трансформации. За последние два года, работая в качестве разработчиков, технических исследователей и активных пользователей продуктов ИИ, мы постоянно испытывали восторг от появления новых инструментов, идей и моделей. В то же время мы обнаружили, что многие люди по-прежнему имеют неправильное представление о вайб-программировании: кто-то счита-

ет его просто «интеллектуальным автозаполнением», кто-то теряется в многообразии терминов, а кто-то хочет начать, но не знает с чего.

Эта книга является попыткой систематически, объективно и доступно проследить эволюцию и основную логику развития данной технологии, а также продемонстрировать практические сценарии применения вайб-программирования в работе отдельных специалистов, команд и предприятий на примере реальных проектов в сфере ИИ. **Глава 1** начинается с истории появления вайб-программирования и помогает читателю по-новому взглянуть на роль искусственного интеллекта в трансформации парадигмы программирования. **Глава 2** посвящена развитию языков и инструментов программирования. **Глава 3** рассказывает о развитии экосистемы приложений вайб-программирования и ее перспективах. **Глава 4** посвящена сценариям применения и практическим примерам вайб-программирования. **Глава 5** представляет собой обзор научно обоснованных и технически проработанных методов вайб-программирования. **Глава 6** описывает создание полнофункционального ИИ-приложения с нуля и помогает читателю эффективно использовать соответствующие инструменты и методы программирования. **Глава 7** посвящена ограничениям и вызовам, с которыми сталкивается современная экосистема вайб-программирования, чтобы помочь читателю всесторонне оценить возможности этой новой парадигмы программирования.

Глава 1

Предпосылки

https://t.me/it_boooks/2

На протяжении последних десятилетий программирование строилось на командах и логике, а выполнение задач на компьютере осуществлялось с помощью построчных последовательностей кода. Однако в настоящее время в этой области наблюдается постепенный процесс изменения парадигмы. Слияние таких новых технологий, как большие языковые модели (Large Language Model, LLM), обработка естественного языка (Natural Language Processing, NLP) и интерактивное программирование (Interactive Programming), привело к появлению совершенно нового подхода к программированию, который в большей степени похож на человеческое мышление, а также более эффективен и гибок. Этим подходом стало **вайб-программирование** (Vibe Coding)[1].

Вайб-программирование – это не просто новая разновидность технологии, но и глубокая трансформация самой концепции разработки и взаимодействия между человеком и машиной. Переход от командного взаимодействия к программированию на основании замыслов, от приспособления человека к машине до понимания человеческого мышления самим компьютером – все это переопределяет границы возможностей современной программной отрасли.

В этой главе читатель шаг за шагом приобщится к сути и идеям вайб-программирования. Особое внимание будет уделено ключевым концепциям этой парадигмы, истории ее развития и предпосылкам ее роста, а также раскрытию ее потенциала и ценности как новой нормы в индустрии разработки.

1.1. Введение в вайб-программирование

В ноябре 2022 года ChatGPT, созданный компанией OpenAI, кардинально изменил устоявшиеся модели взаимодействия человека и машины, ворвавшись в массовое сознание с совершенно новой концепцией. В то время как люди все еще находились в плену стереотипов, связанных с сенсорными интерфейсами, ChatGPT впервые предложил использовать естественный язык в качестве основного средства общения между человеком и машиной, беспрецедентно сократив дистанцию между ними. Теперь человек может общаться с ИИ так же, как со своим приятелем.

---

[1] В настоящее время термин Vibe Coding переводится на русский язык как «программирование по настроению», «интуитивное программирование» «вайб-программирование» или просто «вайб-кодинг». Для обеспечения однозначности и единообразия терминологии в данной книге используется термин «вайб-программирование». – *Прим. перев*.

На момент появления ChatGPT у него все еще наблюдались явные технические недоработки, такие как ограниченный объем обучающих данных, отсутствие возможности подключения к интернету в режиме реального времени и пробелы в базе знаний, вследствие чего его функциональность в целом была еще далека от совершенства. Несмотря на это, даже в таком виде он уже обладал впечатляющими способностями восприятия и синтеза речи, вполне достаточными для работы с большинством повседневных запросов. На удивление многих экспертов, эволюция экосистемы больших языковых моделей происходила гораздо быстрее, чем ожидалось: менее чем за 3 года она трансформировалась из системы для ответов на вопросы на естественном языке в более комплексную интеллектуальную структуру, включающую мультимодальную интеграцию и агентный ИИ (agentic AI).

Все более широкое распространение диалоговых моделей на основе естественного языка постепенно меняет привязанность людей к поисковым системам, основанным на сопоставлении ключевых слов и частичном совпадении. Все больше людей предпочитают задавать вопросы непосредственно большим языковым моделям: от ранней ChatGPT от OpenAI до более поздних Doubao от ByteDance, Tongyi от Alibaba Cloud и Yuanbao от Tencent. Все эти ИИ-инструменты в корне меняют процесс получения информации. Эти ИИ-инструменты не только способны понимать приблизительные и нестандартные человеческие формулировки, но также могут создавать структурированные и релевантные ответы благодаря глубокому пониманию семантики. Такое преимущество существенно повышает эффективность поиска.

Такая беспрецедентная эффективность взаимодействия и способность к пониманию дают разработчикам возможность возлагать на ИИ более высокие ожидания: не просто получать идеи для разрешения задач или образцы кода, но и реально участвовать в креативном процессе и программировании. В связи с развитием и продвижением больших языковых моделей для программирования многие задаются вопросом: если машина сможет понять такое повседневное выражение, как «страница товара с корзиной», то можно ли будет отказаться от традиционного способа написания кода, предполагающего ввод каждой строки вручную?

Из этой идеи постепенно сформировалась совершенно новая парадигма – вайб-программирование. В феврале 2025 года Андрей Карпатый (Andrej Karpathy), один из основателей OpenAI, впервые представил эту концепцию в социальных сетях. Ее суть заключается в использовании больших языковых моделей для максимального приближения процесса разработки к формулировкам естественного языка и создания нового типа программирования, основанного на совместном творчестве человека и машины.

Говоря проще, вайб-программирование можно сравнить с тем, как будто у каждого разработчика появился универсальный «технический помощник» – что-то вроде Джарвиса (J.A.R.V.I.S.) из фильма «Железный человек» (Iron Man), который создает незримый, но чрезвычайно эффективный мост между разработчиком и требованиями к конечному продукту. Программистам больше не нужно тратить силы на разбор синтаксиса кода, выбор фреймворков или тонкости низкоуровневой реализации. Они могут сосредоточиться на ключевых вопросах: что нужно пользователям и какими функциями должен обладать продукт, доверив остальные задачи искусственному интеллекту.

Благодаря глубокому обучению на огромных массивах открытого исходного кода, системных архитектур и моделей разработки большие языковые модели не только освоили различные технологические стеки и лучшие отраслевые практики, но также приобрели способность трансформировать приблизительные пожелания в высококачественный код. С помощью вайб-программирования абстрактные творческие идеи могут быть оперативно преобразованы в практичные, работоспособные программные продукты, что значительно сокращает путь от зарождения идеи до ее воплощения. Традиционное программирование подобно исполнению конкретных команд, тогда как вайб-программирование больше напоминает общение с большой языковой моделью: разработчик задает направление и цели, модель понимает контекст и дополняет детали, чтобы в итоге создать высококачественный код совместными усилиями.

1.1.1. Воплощение идей

При использовании традиционной модели разработки ПО полный цикл создания продукта обычно включает в себя несколько этапов. Для начала команды по исследованию рынка и анализу потребностей пользователей изучают запросы и выявляют проблемные моменты, формируя предварительную концепцию продукта. Затем, на основе результатов исследования, менеджер по продукту составляет четко структурированный **документ с требованиями к продукту** (Product Requirement Document, PRD). Далее на основе этого документа специалисты по дизайну разрабатывают прототипы интерфейса, сценарии взаимодействия и визуальный дизайн, обеспечивая высокое качество взаимодействия с пользователем в дополнение к функциональности продукта. Наконец, на основе проектной документации и документа с требованиями к продукту команда разработчиков корректно интерпретирует замысел и воплощает его в виде исполняемого кода.

Чем больше этапов и участников в таком процессе, тем выше затраты на взаимодействие и тем сложнее становятся возможные сложности. Еще более серьезной проблемой является техническая ограниченность разработчиков и недостаточно глубокое понимание требований к продукту, что затрудняет полноценное и точное воплощение проектных требований в виде стабильной и работоспособной программной системы. Это зависит не только от технической зрелости разработчиков, но и от способности всей команды правильно интерпретировать, согласовывать и реализовывать задачи.

Возьмем, к примеру, такую простую функцию, как «изменение цвета при нажатии кнопки». При традиционном подходе к разработке ПО даже для реализации такой базовой функции потребуется написание нескольких строк кода вручную, что включает в себя работу со структурой HTML, стилями CSS и логикой JavaScript.

```
<button id="colorBtn">Нажмите, чтобы изменить цвет </button>
<script>
  var btn = document.getElementById("colorBtn");
  btn.addEventListener("click", function() {
    btn.style.backgroundColor = "#ff0000";
  });
</script>
```

Для технически неосведомленных или начинающих разработчиков этот простой фрагмент кода может показаться «загадочным текстом». Комплексная программная система состоит из бесчисленного множества подобных базовых функциональных модулей, которые послойно собираются и связываются в соответствии со строгой архитектурной концепцией, нормами кодирования и техническими стандартами. Именно по этой причине программисты с навыками системного программирования долгое время оставались дефицитным человеческим ресурсом.

Появление генеративного ИИ кардинально меняет эту ситуацию. Благодаря возможностям больших языковых моделей роль разработчика трансформируется из «создателя кода» в «проектировщика функциональности». Разработчику не требуется вникать в программные фреймворки, API-вызовы или детали низкоуровневой реализации, достаточно лишь четко сформулировать свои требования на естественном языке. Например, для реализации функции «изменение цвета при нажатии кнопки» разработчику не нужно понимать технические детали, такие как обработка событий или работа с **моделью объектов документа** (Document Object Model, DOM). Достаточно просто задать ИИ запрос: «Создай красную кнопку, которая при нажатии станет синей», – чтобы ИИ за несколько секунд сгенерировал высококачественный код с оптимальной структурой, соответствующий лучшим практикам, и даже автоматически добавил дополнительные функции, такие как анимационные эффекты, адаптивный макет и совместимость со средствами доступа. Эта революция полностью разрушила технические барьеры программирования, превратив разработку ПО из «технической игры для избранных» в «творческий инструмент для всех».

1.1.2. Креативность превыше технологий

Суть вайб-программирования вовсе не заключается в отрицании ценности программных технологий. Смысл в том, чтобы технологии стали «невидимой инфраструктурой». Так же, как при повседневном использовании мобильного телефона нам не нужно заботиться об архитектуре чипа и принципах его работы, в будущем при разработке ПО тоже не придется зацикливаться на деталях реализации кода. Постоянное обучение ИИ незаметно устраняет барьеры, связанные с базовыми технологиями, и превращает ИИ в эффективный механизм преобразования, соединяющий запросы и их реализацию.

Как только стоимость воплощения идей приблизится к нулю, ключевым конкурентным преимуществом разработчиков вновь станет сама креативность: умение определять потребности пользователей, проектировать интерфейс взаимодействия и планировать функциональную архитектуру. Именно в этом заключается основная трансформация, которую приносит ИИ: технологии работают на креативность, а не наоборот. Как опытные разработчики, так и обычные люди без навыков программирования смогут воплощать свои идеи в жизнь благодаря взаимодействию с ИИ на естественном языке.

Техническим специалистам ИИ может помочь в превращении из мастеров написания кода в системных архитекторов.

Специалисты в сфере технологий:
от мастеров написания кода к системным архитекторам

Опытный бэкенд-инженер Лао Чжан в прошлом затрачивал три дня на отладку интерфейса функции календаря посещаемости сотрудников. Сегодня благодаря вайб-программированию ему достаточно всего лишь сказать ИИ:

```
Создай календарь посещаемости с функцией переключения месяцев;
Обеспечь возможность просмотра записей о посещаемости при нажатии
на дату;
Отмечай опоздания сотрудников красным цветом, данные должны синхро-
низироваться с существующим бэкенд-интерфейсом.
```

ИИ не только оперативно генерирует фронтенд-код с библиотекой React-компонентов, но и автоматически согласовывает его с форматом Java-интерфейса, а также добавляет анимацию загрузки данных. Основная задача Лао Чжана теперь состоит не в построчной отладке стилей, а в оптимизации алгоритма учета рабочего времени (даже этот этап может быть реализован с помощью Vibe), что повышает эффективность разработки примерно на 60 %. Это яркий пример того, как технические специалисты совершают впечатляющий переход от рутинной работы к творческой деятельности.

Людям без опыта программирования ИИ дает возможность воплотить свои творческие идеи в реальные продукты.

Люди без опыта программирования:
от авторов идей до владельцев продуктов

Вышедшая на пенсию учительница госпожа Ван мечтала разработать приложение для семейного фотоальбома, чтобы ее дети, проживающие в разных городах, могли в режиме реального времени обмениваться фотографиями из своей повседневной жизни. Она озвучила ИИ свои общие пожелания следующим образом:

```
Возможность загружать фотографии;
После входа в систему члены семьи должны видеть фотоальбомы друг
друга в хронологическом порядке;
Также должна быть возможность оставлять комментарии.
```

ИИ, как заботливый помощник, помог ей уточнить детали:

```
Входить через WeChat или регистрировать аккаунт;
Нужны ли теги для классификации фотографий;
Создавать ли видео с воспоминаниями за каждый год.
```

Госпожа Ван сделала следующие уточнения:

```
Вход через WeChat;
Классификация по тегам «Путешествия», «Праздники» и т. д.;
Создание видео с воспоминаниями за каждый год под фоновую музыку
«Дорогой путешественник».
```

С помощью ИИ госпожа Ван всего за три дня создала и доработала полнофункциональное приложение с интеллектуальной сортировкой и функцией монтажа видео. Более того: когда ей показалось, что видео генерируется слишком медленно, она просто попросила: «Ускорьте создание видео», после чего ИИ автоматически оптимизировал алгоритм сжатия изображений. Так, не имея ни малейшего представления о программировании, она обзавелась собственным семейным приложением для общения.

Несмотря на то что ИИ пока что не в состоянии решать все сложные задачи разработки, он уже с легкостью справляется с большинством повседневных задач. Будь то создание сайта электронной коммерции, разработка приложения для ведения дневника или дизайн небольшой развлекательной игры, ИИ способен точно определить и доработать требования, а также быстро сгенерировать готовый к использованию код.

Революционная сущность вайб-программирования заключается в том, что оно перестраивает систему координат мышления разработчика. Оно не преуменьшает важность мышления, а помогает разработчику перенести акцент с «мелочей технической реализации» на «основные задачи создания ценности». Когда генерация кода станет такой же простой, как набор текста, конкурентоспособность продукта будет зависеть не от программистских навыков разработчика, а от его способности анализировать потребности клиентов и раскрывать творческий потенциал.

Вполне возможно, что дальнейшая эволюция и распространение ИИ повлекут за собой новый цикл перемен в плане повышения производительности труда в цифровую эпоху. Так же, как пакет Office сделал обработку текстов и статистический анализ данных базовыми навыками для работников офиса, ИИ, в свою очередь, может стать цифровой инфраструктурой нового поколения, которая разрушит технологические барьеры и даст каждому человеку возможность свободно воплощать свои идеи в жизнь.

1.2. Переход от командного стиля к целевому

В 50-х годах XX века программирование представляло собой чрезвычайно сложную профессию. Разработчики того времени должны были не только в совершенстве владеть логикой работы ПО, но также глубоко понимать принципы работы аппаратной платформы. В те времена программирование больше напоминало своего рода искусство работы с языком аппаратного уровня, когда с машиной велся прямой диалог с помощью строк двоичных инструкций.

По мере развития компьютерных наук и появления высокоуровневых языков программирования эффективность разработки заметно возросла. Но даже несмотря на постоянное совершенствование технологических инструментов – от ассемблера до высокоуровневых языков, от командной строки до **интегрированных сред разработки** (Integrated Developer Environment, IDE), процесс создания ПО по-прежнему строился по принципу «человек пишет код, машина выполняет». Такая модель не только оказала глубокое влияние на процесс разработки, но и сформировала привычный рабочий режим программистов.

И по сей день для создания продукта по-прежнему необходимы редактор, язык программирования, совместная работа нескольких специалистов и соблюдение определенных процессов. Разработчику необходимо преобразовать требования к продукту в бизнес-логику, затем преобразовать бизнес-логику в код, после чего реализовать его с помощью инструментов. Такая модель разработки стала нормой в сфере технологий.

Появление вайб-программирования начинает разрушать эту «норму», переопределяя отправную точку и путь разработки ПО на основе принципа «управление намерениями и сотрудничество человека и машины».

1.2.1. Обзор императивного программирования

Императивное программирование – это парадигма, в основе которой заложено явное управление потоком выполнения. Разработчику приходится описывать изменения состояния программы построчно, реализуя целевую логику с помощью переменных, условных операторов и управления памятью.

В ранних версиях императивное программирование в значительной степени зависело от низкоуровневых аспектов. Для примера рассмотрим перфокарты (см. рис. 1.1), где каждая программная инструкция должна была быть запечатлена в виде физических отверстий на карте. Это существенно снижало эффективность взаимодействия человека с компьютером, а также требовало от разработчиков глубокого понимания сложных концепций, таких как регистры, адреса памяти и логика переходов.

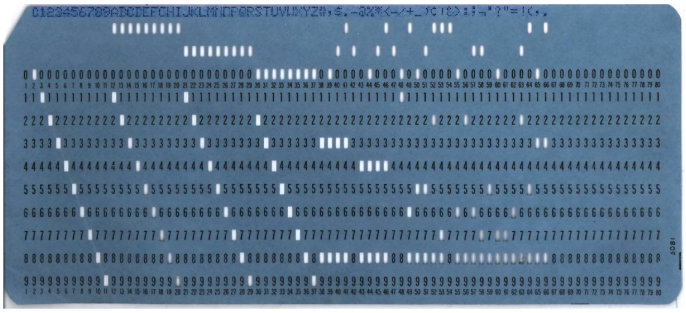


Рисунок 1-1. Перфокарта (изображение из Википедии)

По мере развития вычислительных технологий появление высокоуровневых языков программирования, таких как Fortran и COBOL, избавило разработчиков от необходимости непосредственно работать с ассемблерными командами и позволило описывать логику программы с помощью синтаксиса, более близкого к естественному языку. Впоследствии появились такие инструменты, как **интегрированная среда разработки** (IDE), системы контроля версий и **конвейеры непрерывной интеграции / непрерывной доставки** (Continuous Integration/Continuous Delivery, CI/CD), что значительно повысило эффективность разработки. Несмотря на появление множества новых технических инструментов, модель «человек пишет логику, машина выполняет» осталась неизменной на протяжении всего жизненного цикла разработки ПО. Даже в эпоху облачных и low-code технологий подавляющая часть разработки

по-прежнему строится на этой базовой модели. В крупных проектах взаимодействие переменных между модулями, стек функций вызова и многопоточная параллельность делают разработку в рамках императивного мышления чрезвычайно трудоемкой: малейшее изменение может повлечь за собой глобальные последствия. Именно в этом контексте особенность императивного программирования, которая заключается в приоритетном внимании к подробностям, постепенно стала одной из основных проблем разработчиков.

- **Высокая умственная нагрузка**: разработчику приходится постоянно отслеживать изменения статуса, что зачастую приводит к долгим процессам отладки.
- **Сложности обновления**: множество мелких элементов и нагромождение шаблонного кода, из-за чего любые изменения могут повлиять на работу всей системы.
- **Высокие затраты на согласование**: сильная зависимость от экспертных знаний, многочисленные конфликты версий кода и трудности в общении внутри команды.
- **Слабая абстракция**: сильная связность процессов, что затрудняет извлечение высокоуровневых компонентов для повторного использования.
- **Сложность тестирования**: усложненный контроль выполняемых операций приводит к резкому росту затрат на тестирование и риску возникновения непредвиденных ситуаций.

Эти проблемы повышают уровень нагрузки на разработчиков и замедляют скорость реализации проектов всей команды. С ростом сложности бизнес-задач и ускорением темпов разработки императивное программирование становится все менее пригодным для удовлетворения современных запросов в плане скорости и качества.

1.2.2. От команд к намерениям

Для решения проблем императивного программирования была разработана концепция **интенционального** (или **преднамеренного**) **программирования**, основная идея которой заключается в упоре на намерения пользователя и бизнес-цели, а не на процесс выполнения отдельных команд. Благодаря эффективному замкнутому циклу «описание намерения – генерация системы – взаимодействие человека и машины» интенциональное программирование позволяет разработчикам не углубляться в рутинные подробности реализации и сосредоточиться на самом важном вопросе: «что именно я хочу получить».

В этой парадигме разработчик больше не начинает с написания «первой строки кода», а приступает к определению цели. Например, если разработчик задает требование «мне нужен процесс, поддерживающий регистрацию и вход пользователей», система автоматически распознает лежащие в его основе структуры данных, логические правила, требования безопасности и так далее, а затем генерирует исходную реализацию. Разработчики больше не являются создателями низкоуровневой логики, они становятся конструкторами целей. Они выражают свои намерения с помощью естественного языка, графического моделирования и других средств, а система берет на себя работу по реализа-

ции деталей с помощью генерации кода, сборки компонентов и предварительного просмотра в реальном времени.

Говоря конкретнее, принципиальные изменения при интенциональном программировании проявляются в следующих аспектах:

- **фокус на цели, а не на пути**: разработка переходит от «как сделать» к «чего достичь»;
- **повышение уровня абстракции**: намерения могут быть преобразованы в модули для повторного применения и совместного использования разными командами;
- **унификация языка взаимодействия**: бизнес, продукт и разработка выравниваются на основе единой семантической структуры;
- **более оперативное реагирование**: теперь изменения начинаются не с нижнего уровня, а с настройки конфигурации или тонкой настройки намерения.

Необходимо понимать, что интенциональное программирование – это не просто инструмент для ускорения написания кода, а совершенно новая система, которая переопределяет процесс разработки, перестраивает модели мышления и способы организационного взаимодействия. Эти изменения отражают основной дух вайб-программирования: превратить «выражение намерения» в отправную точку разработки и вернуть опыт разработки во главу угла творческого процесса.

1.2.3. Сравнение методов разработки

Для более глубокого понимания различий между двумя парадигмами программирования, императивной и интенциональной, ниже на примере реализации функции «входа пользователя» сравниваются традиционный подход к разработке и вайб-подход – как с точки зрения алгоритма реализации, так и в плане процесса разработки.

1. Традиционный процесс разработки

При традиционном императивном программировании даже разработка простейшего модуля входа пользователя представляет собой довольно сложный процесс, который, как правило, включает в себя множество этапов и шагов взаимодействия:

- **анализ требований и проектирование**: определение способов аутентификации при входе (например, использование имени пользователя / пароля, OAuth и т. д.), структуры базы данных, механизмов безопасности (например, шифрование паролей, защита от атак грубой силы и т. д.), а также составление диаграмм отношений между сущностями – ER-диаграммы (Entity Relationship) и документации по интерфейсам;
- **настройка среды**: настройка локальной среды разработки, службы базы данных, веб-фреймворка, интеграция библиотек шифрования (например, BCrypt) и промежуточного ПО для аутентификации (например, Spring Security, Django REST Framework);

- **реализация кода, бэкенд**: реализация интерфейсов регистрации/авторизации пользователей, логики выдачи и проверки токенов[1], создание уровня взаимодействия с базой данных;
- **реализация кода, фронтенд**: создание пользовательского интерфейса страницы входа, обработка событий форм, интеграция с логикой авторизации;
- **тестирование и отладка**: проверка функциональности с помощью инструментов тестирования интерфейсов, проверка бизнес-логики с помощью фреймворка модульного тестирования, а также устранение типичных проблем, таких как междоменные ошибки и ошибки токенов;
- **развертывание и эксплуатация**: упаковка кода и его развертывание в производственной среде, настройка протокола SSL (Secure Socket Layer) и политик безопасности, а также последующее устранение потенциальных уязвимостей и поддержка эволюции функциональности.

Этот процесс разработки характеризуется не только длинной технологической цепочкой и множеством зависимостей, но также предъявляет высокие профессиональные требования к разработчикам. Некоторые, казалось бы, незначительные детали, такие как настройка алгоритмов шифрования и управление жизненным циклом токенов, требуют тщательной проработки, поскольку малейшая небрежность может привести к угрозам безопасности или проблемам с производительностью. В условиях совместной работы нескольких человек такие этапы, как совместная отладка фронтенда и бэкенда, изменение интерфейсов и синхронизация требований, также могут создавать препятствия для эффективной работы.

2. Процесс вайб-разработки

В рамках новой парадигмы интенционального программирования процесс разработки значительно упрощается: разработчикам больше не приходится создавать базовую логику вручную, достаточно описать задачу с помощью естественного языка и графических операций, а система сама выполнит большую часть работы по созданию кода.

- **Ввод интенций** (намерений): разработчик описывает требования на естественном языке, например «создать функцию, поддерживающую вход по имени пользователя и паролю, с использованием шифрования BCrypt; при успешном входе возвращать токен JWT».
- **Автоматическая ИИ-генерация**: платформа автоматически формирует полный набор бэкенд-интерфейсов, скриптов миграции базы данных и компонентов фронтенд-страниц, а также автоматически дополняет механизмы безопасности в соответствии с передовыми практиками.
- **Развертывание в один клик**: код может быть напрямую развернут на основных облачных платформах, при этом синхронно выполняется настройка SSL и формирование документации по интерфейсам прикладных программ (Application Program Interface, API).

[1] Используемый в этой главе термин «токен» (Token) является условным обозначением, используемым при аутентификации пользователей и проверке их прав доступа.

- **Интеллектуальная эксплуатация и обслуживание**: платформа обладает встроенным модулем мониторинга, способным обнаруживать аномальные действия при входе в систему; ИИ автоматически предлагает рекомендации по оптимизации и устраняет наиболее распространенные уязвимости.

В рамках этой новой парадигмы процесс разработки, который ранее требовал участия нескольких специалистов и занимал несколько дней, зачастую может быть выполнен одним человеком за более короткий срок. При этом обеспечивается хорошая масштабируемость: для последующего подключения таких функций, как OAuth и двухфакторная аутентификация (2FA), требуется лишь настройка конфигурации, без необходимости масштабной переработки кода.

В табл. 1-1 приведены различия между традиционным процессом разработки и процессом разработки с использованием вайб-программирования.

Таблица 1-1. Различия между традиционным и вайб-процессом разработки

Программный процесс	Традиционная разработка	Вайб-разработка
Эффективность разработки	Совместная работа нескольких человек, длительный цикл	Возможность выполнения и сдачи проекта одним человеком в короткие сроки
Технические требования	Требуется владение несколькими языками программирования и сложными фреймворками	Настройка осуществляется с помощью естественного языка и графического интерфейса
Безопасность	Шифрование и аутентификация реализуются вручную, высокий уровень риска	Автоматическое генерирование кода в соответствии с нормативными требованиями, встроенные средства защиты
Затраты на обслуживание	Для внесения изменений необходимо просматривать код по всей системе, что чревато ошибками	Обновления с помощью конфигурации, более гибкое обслуживание
Сложность совместной работы	Разделение фронтенда и бэкенда, необходимость частых совместных настроек	Единое моделирование всего стека, более эффективная совместная работа

1.3. Основы вайб-программирования

В разделе 1.2 уже была представлена концептуальная характеристика вайб-программирования – новой парадигмы разработки, в основе которой лежат «управляемость намерениями» и «сотрудничество человека и машины». В этом разделе мы сосредоточимся на трех ключевых факторах: зрелости технологий больших языковых моделей, стимулирующем влиянии рыночного спроса и основных проблемах разработчиков, а также проанализируем их совместное влияние на возникновение вайб-программирования.

1.3.1. Становление технологий больших языковых моделей

В последние годы размер больших языковых моделей увеличивался скачкообразно, с нескольких сотен миллионов до нескольких сотен миллиардов параметров, что привело к экспоненциальному росту производительности. Масштабные предварительно обученные модели (такие как GPT, PaLM и др.) демонстрируют отлич-

ные результаты в области восприятия и генерации естественного языка. Когда эти возможности переносятся в сферу разработки кода, модели способны не только автоматически дополнять простые функции на основе локального контекста, но также понимать структуру всего проекта благодаря более длинному контекстному окну. Благодаря этому они могут генерировать сложную бизнес-логику, оптимизировать существующий код и даже осуществлять перевод между языками. Такое расширение возможностей обработки длинного контекста позволяет моделям поддерживать согласованность в именовании, структуре и логической реализации в больших кодовых базах, быстро выявлять зависимости и обеспечивать существенную поддержку для гибкой разработки и непрерывной интеграции.

Уровень вычислительной мощности и платформенные возможности также постоянно возрастают: вычислительная мощность кластеров GPU/TPU неуклонно увеличивается, в то время как стоимость снижается, а аренда ресурсов в облаке на условиях «по требованию» стала нормой. Крупнейшие облачные провайдеры один за другим выпускают специализированные инстансы для обучения и инференции больших языковых моделей. Это позволяет корпоративным и индивидуальным разработчикам получать доступ к мощным вычислительным ресурсам с минимальными затратами. Совершенствование фреймворков глубокого обучения, таких как TensorFlow и PyTorch, а также инструментальных цепочек для распределенного обучения обеспечивает контролируемый и эффективный характер всего процесса, от разработки моделей до их масштабного развертывания. Еще более важным является быстрое развитие инструментальных цепочек и экосистем, в основе которых лежат эти модели. В основные IDE (такие как VS Code и серия JetBrains) теперь глубоко интегрированы интеллектуальные плагины, например Copilot, благодаря чему разработчики могут использовать возможности больших языковых моделей непосредственно в своем любимом редакторе. В то же время появились специализированные IDE для программирования с использованием ИИ, такие как Cursor и Windsurf, которые обеспечивают полную поддержку всего рабочего процесса – от интеллектуального автодополнения и рефакторинга кода до генерации тестов, в соответствии с потребностями разработчиков разного уровня. Платформы **программирования с малым использованием кода** (low-code) и инструменты для автоматизации процессов также массово подключаются к возможностям больших языковых моделей, обеспечивая полный цикл работы – от создания прототипа до развертывания в производственной среде.

Совместная эволюция программного и аппаратного обеспечения, фреймворков и инструментов способствует эффективному внедрению и быстрому распространению вайб-программирования, обеспечивая беспрецедентный рост производительности разработчиков.

1.3.2. Стимулирующие факторы рыночного спроса

В последние годы темпы обновления продуктов постоянно ускоряются: выпуск одной версии в неделю или даже нескольких версий в день стал нормой для большинства команд. В условиях столь сжатых сроков сдачи проектов важно обеспечить своевременный запуск новых функций, не жертвуя при этом качеством кода и стабильностью системы, что ставит перед разработчиками серьезные задачи по повышению эффективности. Традиционный процесс «ручное на-

писание → ручная проверка → ручное тестирование» отнимает не только много времени и трудозатрат, но также вызывает цепную реакцию задержек в последующих этапах при нарушении графика на любом из них. Если ускорить процесс слишком резко, часто возникающие ошибки в коде могут негативно повлиять на пользовательский опыт и репутацию бренда; если же чрезмерно полагаться на ручное тестирование и проверку кода, возникает риск возникновения проблем со сроками сдачи, что приводит к задержкам запуска или пустой трате ресурсов.

Одновременно с этим, по мере все большего разнообразия бизнес-сценариев, все более заметными становятся барьеры во взаимодействии между отделом разработки и бизнес-командами. Продукт-менеджер может утром получить отзывы с рынка, а уже днем ему необходимо быстро запустить в системе внеплановую акцию; после того как разработчик интерфейса скорректировал способ взаимодействия со страницей, ему необходимо срочно согласовать с фронтенд-разработчиком новую верстку; при этом команда разработчиков зачастую не может обеспечить оперативный отклик из-за нехватки кадров или различий в технологическом стеке, что в конечном итоге приводит к растягиванию цикла «идея → реализация → проверка» до нескольких дней или даже недель.

Несмотря на то что платформы с низким уровнем кодирования (low-code) или без кодирования (no-code) в определенной степени снижают порог входа для неспециалистов, они имеют значительные ограничения при глубокой настройке, в случае сложной бизнес-логики и при оптимизации производительности. В то же время при написании кода вручную в короткие сроки сложно обеспечить одновременно масштабируемость и возможность повторного использования, что при высокой частоте итераций приводит к постоянному «изобретению велосипеда».

Помимо этого, требования различных отраслей в отношении соответствия стандартам, обеспечения безопасности, производительности и т. д. значительно различаются. Например, финансовая отрасль требует строгого контроля доступа и ведения аудиторских логов, сектор электронной коммерции нуждается в гибком механизме проведения рекламных акций, а производственная отрасль требует интеграции с устройствами интернета вещей в режиме реального времени. Эти сложные сценарии еще больше усугубляют недостатки универсальных решений, заставляя команды выбирать между готовыми решениями и глубокой настройкой. Нерешительность при выборе технологий зачастую напрямую приводит к задержкам при запуске проектов и избыточным затратам.

Именно в таких рыночных условиях и возникла необходимость в вайб-программировании. Объединяя «намерения» разработчиков, продуктовых менеджеров, дизайнеров, тестировщиков и специалистов по эксплуатации в едином интеллектуальном движке, вайб-программирование позволяет использовать преимущества больших языковых моделей для анализа и генерации длинных контекстов, обеспечивая интеграцию всего рабочего процесса – от технического задания по продукту до рабочего кода, а также сценариев автоматического тестирования и развертывания. Разработчикам достаточно сосредоточиться на проектировании основной бизнес-логики, в то время как ИИ-помощник в фоновом режиме выполняет построение базовой инфраструктуры, интеграцию интерфейсов, генерацию тестовых случаев и даже обновление документации. Это позволяет существенно сократить цикл разработки и поддерживать высокие стандарты качества CI/CD. Таким образом, будь то временная рекламная акция или глубоко настраиваемое

решение для вертикальной отрасли, все это можно быстро реализовать на единой платформе с помощью готовых решений и профессиональной оптимизации, что позволяет добиться одновременно высокой скорости и стабильности.

1.3.3. Ключевые проблемы разработчиков

Именно в таких рыночных условиях и возникла необходимость в вайб-программировании. Объединяя «намерения» разработчиков, продуктовых менеджеров, дизайнеров, тестировщиков и специалистов по эксплуатации в едином интеллектуальном движке, вайб-программирование позволяет использовать преимущества больших языковых моделей для анализа и генерации длинных контекстов, обеспечивая интеграцию всего рабочего процесса – от технического задания по продукту до рабочего кода, а также сценариев автоматического тестирования и развертывания. Разработчикам достаточно сосредоточиться на проектировании основной бизнес-логики, в то время как ИИ-помощник в фоновом режиме выполняет построение базовой инфраструктуры, интеграцию интерфейсов, генерацию тестовых случаев и даже обновление документации. Это позволяет существенно сократить цикл разработки и поддерживать высокие стандарты качества CI/CD. Таким образом, будь то временная рекламная акция или глубоко настраиваемое решение для вертикальной отрасли, все это можно быстро реализовать на единой платформе с помощью готовых решений и профессиональной оптимизации, что позволяет добиться одновременно высокой скорости и стабильности.

В процессе реальной разработки неотъемлемой частью работы являются постоянно повторяющиеся задачи: от быстрого создания структуры проекта с помощью шаблонов, инициализации модулей и настройки маршрутизации до интеграции с различными сторонними сервисами, а также создания набора модульных тестов вручную для каждой новой функции. К этому добавляется необходимость обновления комментариев в коде и документации. Все эти задачи, требующие лишь базовых шаблонных действий, могут отнимать до 30–50 % рабочего времени разработчика. Хуже того: каждый раз, когда в проект внедряется новый стек технологий, происходит переход на другую среду бэкенд-языка или смены фронтенд-фреймворка на другую экосистему, а это означает, что весь цикл предварительных работ – от настройки среды и установки взаимосвязей до изучения документации – приходится начинать с нуля. Мыслительный процесс вынужденно прерывается, интеллектуальная нагрузка мгновенно возрастает, а ритм инноваций и проектирования выбивается из колеи.

Такие издержки при переключении между рабочими средами становятся еще более заметными при совместной работе нескольких команд. Например, микросервис A, за который отвечает один человек, использует Java+Spring Boot, а микросервис B, за который отвечает другой, основан на Node.js+Express. Разработчику приходится переключаться между несколькими проектными структурами, при этом ему приходится не только запоминать соответствующие команды сборки и методы отладки, но также писать соответствующие скрипты для разных тестовых фреймворков. Зачастую утром приходится настраивать формат логов микросервиса A, а днем создавать *имитационные* (mock) API для микросервиса B. Такая смена контекста приводит к резкому снижению производительности и повышению вероятности возникновения элементарных ошибок.

Во время быстрых итераций кодовой базы обновление документации обычно отстает от фактической реализации, в результате чего новые сотрудники и аутсорсинговые команды вынуждены читать большие фрагменты исходного кода для понимания бизнес-логики, что отнимает много времени и сил. Такая проблема с отставанием актуальной документации становится серьезным препятствием при командной работе. Даже опытным разработчикам приходится тратить несколько часов на поиск и устранение ошибок из-за отсутствия записей в документации, особенно когда в ключевой библиотеке крупного проекта произошли незначительные изменения, но они не были отражены в документации. Это значительно увеличивает затраты на устранение неполадок и повторное тестирование.

Для решения описанных выше проблем необходим интеллектуальный инструмент, способный автоматически фиксировать и синхронизировать технические требования проекта, документацию и код в рамках привычного рабочего процесса разработчика. Опираясь на большие языковые модели с более мощными возможностями обработки длинного контекста, вайб-программирование превращается в новый стандарт этой парадигмы, способный охватить весь проект в локальной IDE или конвейере CI/CD и объединить все этапы разработки. Помимо создания шаблонов кода, более соответствующих бизнес-задачам, наряду со стандартными командами командной строки, оно также способно автоматически корректировать вызовы интерфейсов на основе новейшей документации API, а после реализации кода – автоматически дополнять комментарии и синхронизировать техническую документацию, и даже генерировать предварительные тестовые наборы и отчеты о сканировании безопасности в каждом **запросе на изменение** (или пул-реквесте – pull request, PR). Такой сквозной механизм взаимодействия «документация + код + тестирование» с контекстной поддержкой значительно сокращает рутинную работу, позволяет синхронизировать знания команды в режиме реального времени, полностью устраняет информационные разрывы и делает каждую итерацию по-настоящему эффективным улучшением.

Зрелость технологий больших языковых моделей заложила прочную основу, сильный рыночный спрос на «быструю и стабильную» доставку создал реальные условия, а основные проблемы разработчиков стали катализатором ускорения. Сочетание этих трех факторов способствовало появлению вайб-программирования.

1.4. Заключение

Появление вайб-программирования – это не случайный технологический прорыв, а неизбежный результат длительной эволюции и накопления опыта. Сдвиг парадигмы, лежащий в его основе, обновление представлений и технологический скачок коренным образом меняют наше понимание процесса разработки ПО.

Чтобы по-настоящему понять истоки этой трансформации, нам необходимо расширить временную перспективу, вернуться к истории развития программирования и проследить его эволюцию от базовых уровней до сегодняшней сложной и многогранной экосистемы. Глава 2 познакомит вас с историей программирования и поможет по-новому взглянуть на его истоки и перспективы.

Глава 2

Эволюция способов программирования

В прошлом умение программировать считалось сложным навыком, доступным лишь узкому кругу специалистов. В настоящее время оно стремительно превращается в доступный каждому вид творчества. От первоначального побитового управления машинами до создания сложных систем с помощью языков высокого уровня, от сложных операций в интерфейсе командной строки до визуальных инструментов, работающих по принципу «что видишь, то и получаешь», и до современных платформ с низким уровнем кодирования и бескодовых платформ – методы программирования постоянно претерпевают глубокие и непрерывные изменения.

В этой главе мы рассмотрим эволюцию способов программирования с трех точек зрения. Во-первых, мы проследим историю развития языков программирования и попробуем ответить на вопрос, как дизайн языков стимулирует технологические инновации. Затем уделим внимание эволюции способов взаимодействия с программами – от клавиатурных команд до графических интерфейсов и интеллектуальных помощников, чтобы проследить за изменением способов взаимодействия человека и компьютера. Наконец, на примере технологий с низким уровнем кода или вовсе без кода мы рассмотрим влияние современной тенденции «декодификации» на перестройку экосистемы разработки и расширение возможностей участия в технологическом процессе.

Понимание этих изменений необходимо не только для анализа исторического контекста, но также для прогнозирования будущего.

2.1. Эволюция языков программирования

На самых ранних этапах истории программирования разработчики могли взаимодействовать с аппаратным обеспечением только посредством прямой передачи двоичных команд, состоящих из последовательностей нулей и единиц. В 50–70-е годы XX века появляются ассемблер, Fortran, COBOL и другие языки высокого уровня, которые заменяют двоичные инструкции на мнемонические коды, математические символы и элементы естественного языка, что значительно облегчает работу разработчиков. В 70–80-е годы концепция структурированного программирования языка Pascal и объектно-ориентированная философия Smalltalk-80 коренным образом изменили подход к написанию программного кода. После 90-х годов языки программирования Java, JavaScript,

Python и другие, в основе которых лежат принципы кросс-платформенности, асинхронности и минимализма, заложили фундамент для современной экосистемы программирования со множеством различных парадигм.

В данном разделе мы проведем обзор этого пути развития от машинного кода до современных языков программирования.

2.1.1. Общение с машиной

Эпоха **машинного языка** (machine language) пришлась на 40-е и 50-е годы XX века. Будучи языком уровня аппаратного обеспечения, он позволял напрямую управлять элементами оборудования: передачей электрических сигналов в регистрах центрального процессора, физическим отображением адресов памяти и работой портов устройств ввода-вывода. Все это осуществлялось с помощью двоичных инструкций.

Пионеры программирования вели своего рода «первобытный диалог» с электронными схемами посредством прямого управления двоичным кодом: последовательностью нулей и единиц – и составляли весь программный код. Каждое вычисление и каждая операция с памятью требовали точной настройки этих двоичных битов, что снижало эффективность и приводило к частым ошибкам. Программирование напоминало блуждание в тумане: оно было сложным и полным непростых задач.

14 февраля 1946 года в Университете Пенсильвании был запущен ENIAC (Electronic Numerical Integrator and Computer) – «электронный цифровой интегральный компьютер», см. рис. 2-1. Этот гигантский аппарат, занимавший площадь 167 кв. м, содержал 17 468 электронных ламп и мог выполнять 5000 операций сложения в секунду.

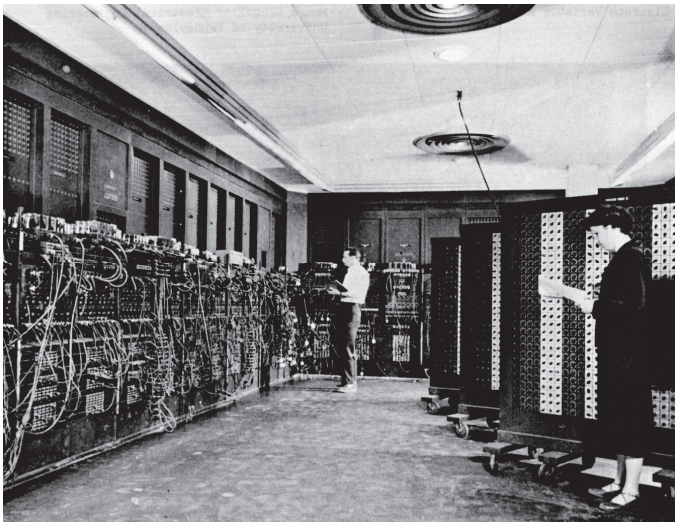


Рисунок 2-1. ENIAC (изображение взято из Википедии)

В эпоху машинного языка программистам для формирования различных каналов передачи данных приходилось вручную подключать и отключать до 2000 соединений. При выполнении сложения электрический сигнал должен

был проходить от входного регистра через электронный делитель и в конечном итоге поступать в 300-разрядный сумматор. При выполнении сложения требовалось активировать 36 независимых 4-разрядных блоков умножения, каждый из которых выполнял булевы операции через матрицу реле. Такая сложность операций на физическом уровне приводила к тому, что даже для изменения одного адреса данных требовалось несколько часов на перенастройку соединений, а малейшее отклонение приводило к сбою сигналов в регистрах и остановке всего процесса.

Система UNIVAC I (см. рис. 2-2) считается первым в мире коммерческим компьютером. По сравнению с ENIAC он стал прорывом в области хранения программ и способов ввода данных: благодаря отделению программ от аппаратного обеспечения и использованию перфоленты для однократной загрузки команд одна и та же система могла выполнять различные вычисления, что ознаменовало начало эры универсальных вычислений. Однако UNIVAC I предъявлял к оператору почти невыполнимые требования: даже для простого суммирования чисел от 1 до 100 требовалось написать более 200 двоичных инструкций, что соответствовало пробиванию более 4000 отверстий на перфорированной ленте шириной 16 колонок. Каждая программа была похожа на индивидуальный «патч» для аппаратной архитектуры, который было нереально скопировать и практически невозможно изменить. Процесс разработки был не только утомительным, но и крайне зависимым от кропотливости и терпения оператора, что делало его очень неэффективным.



Рисунок 2-2. Система UNIVAC I (изображение взято из Википедии)

2.1.2. От двоичного кода к символьному языку

В 50–70-е годы XX века языки программирования вышли из-под господства двоичного кода, официально вступив в эпоху **ассемблера** (assembly language), что стало первым серьезным шагом вперед к абстрактному уровню программирования. Благодаря введению символьной системы кодирования разработчикам больше не требовалось работать со сложным двоичным кодом напря-

мую. Вместо этого они могли использовать такие символьные команды, как `MOV A, @SRC`, а не набивать в память машинные коды. Этот переход значительно повысил эффективность программирования и позволил разработчикам перенести внимание с нюансов аппаратного уровня на более высокий уровень алгоритмов и операций с данными.

В 1955 году IBM выпустила Autocoder – один из первых ассемблеров для компьютера IBM 702. Он впервые обеспечил автоматический перевод символьных кодов в машинный код и ввел такие расширенные функции, как макрокоманды. Так, например, появилась возможность преобразования низкоуровневых операционных кодов в более читаемые команды (такие как использование `ADD` вместо `21`). Хотя это по-прежнему была машинная форма написания кода, такой механизм автоматического перевода послужил техническим прообразом для появления последующих высокоуровневых языков программирования.

В 1963 году компания IBM выпустила макроассемблер для своего полупроводникового компьютера 7094 (см. рис. 2-3). Это был первый макроассемблер с механизмом определения макросов, который поддерживал условное ассемблирование и рекурсивную обработку. Программисты могли создавать макросы с параметрами, а также использовать псевдокоманды условий (if and only if false, `IFF`) и итерации (if and only if true, `IFT`). Благодаря гибкому повторному использованию шаблонов кода удалось значительно повысить простоту обслуживания, а также возможности повторного использования программ.

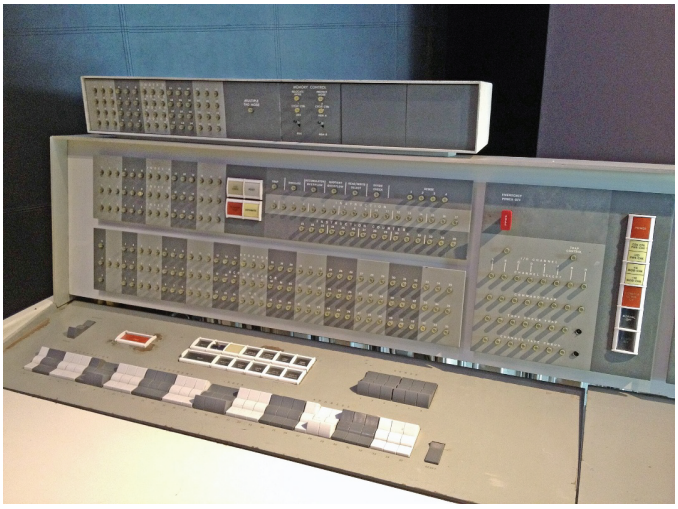


Рисунок 2-3. Полупроводниковый компьютер IBM 7094 (автор изображения: ArnoldReinhold)

Тем не менее все первые макроассемблеры имели недостаток в виде замусоривания общего пространства имен. Иными словами, макросы с одинаковыми именами в разных модулях могли накладываться друг на друга, вызывая в программе сложно отслеживаемые ошибки. Лишь в 1970 году компания DEC выпустила для своего мини-компьютера PDP-11 (см. рис. 2-4) ассемблер MACRO-11, в котором впервые было введено ключевое слово `LOCAL` для изоля-

ции области действия макросов. Этот механизм позволил эффективно решить проблему конфликтов имен и впоследствии широко использовался во многих средах, таких как компилятор NASM и ассемблер GNU. С этого момента использование ассемблера стало более практичным.



Рисунок 2-4. Мини-компьютер PDP-11 (автор изображения: Stefan Kögl)

Ранние ассемблеры для PDP-8 (такие как PAL-III, выпущенный примерно в 1965 году) были первыми, где появился ряд псевдокоманд (pseudo-instruction), включая `DECIMAL`, `FIELD`, `EXPUNGE` и `FIXTAB`. Они сами не генерировали машинный код, но использовались для поддержки процесса ассемблирования, управления счетчиком адресуемых позиций и определения констант, что позволило упростить структуру программ и процесс разработки.

Ниже приведен более наглядный и структурированный пример кода, иллюстрирующий использование псевдокоманд в ассемблере PDP-8 (на примере PAL-III) для организации кода, определения констант и установки начального адреса.

```
* Указание десятичного формата чисел (псевдокоманда DECIMAL)
DECIMAL

* Учет позиций и управление полями: установка адреса загрузки программы
в 0x1000 (FIELD), смещения внутри страницы в 0x0 (PAGE)
FIELD 1             / Выбор поля 1 (0x1000-0x17FF) 0

*200                / ORG/PAGE, установка счетчика позиций в начальный адрес
текущей страницы

* Определение констант (псевдокоманда EQU / =)
BUFFER = 4096       / Определение адреса BUFFER как 0x1000 (4096 в десятичной
системе)
```

```
* Основная часть программы
LOADI  BUFFER      / Загрузка адреса BUFFER в регистр счетчика (пример
псевдокоманды, фактическая команда зависит от ассемблера)

* Резервирование пространства, заполнение нулями
ZBLOCK  16         / Выделение блока длиной 16 байт для инициализации нулями

* Пример условной ассемблерной инструкции
IFDEF BUFFER       / Если BUFFER уже определен, то выполняется следующее
   TAD BUFFER
ENDIF

* Конец ассемблера
$
```

Такой способ символьной адресации избавляет разработчиков от необходимости ручного вычисления сложных абсолютных адресов памяти. Это позволяет повысить удобочитаемость и легкость обслуживания кода, а также закладывает основу для появления в последующих высокоуровневых языках программирования таких концепций, как переменные и область действия.

В период зарождения компьютерной индустрии программирование на ассемблере для архитектуры x86 можно было назвать экстремальным испытанием для разработчиков. Возьмем, к примеру, загрузчик MS-DOS 1.0, выпущенный в 1981 году (см. рис. 2-5): даже для написания простого кода запуска системы разработчикам приходилось с прецизионной точностью контролировать каждый сегментный регистр и смещение адреса – как при ремонте сложного оборудования, поскольку любая мелкая ошибка могла привести к сбою всей системы.



Рисунок 2-5. Интерфейс загрузчика MS-DOS 1.0 (изображение взято из Википедии)

Так, например, разработчик может написать приведенный ниже оператор для загрузки адреса сегмента данных программы в сегментный регистр `DS`, чтобы впоследствии все обращения к данным с префиксом `DS` по умолчанию смогли

правильно найти переменную. Разработчику приходилось вручную производить точные вычисления адресных смещений в памяти; малейшее отклонение могло привести к сбою в адресации глобальных переменных и, как следствие, к сбою системы. Такая жесткая зависимость от низкоуровневой архитектуры в те времена превращала разработку на ассемблере в настоящую авантюру.

```
MOV AX, @DATA_SEGMENT ; Загрузка адреса сегмента данных в регистр AX
MOV DS, AX            ; Присвоение значения регистра AX регистру сегмента данных DS
```

И только в 1985 году, после выхода процессора Intel 80386, появился **защищенный режим** (protected mode), что послужило прорывом на пути развития механизмов управления памятью. Этот режим впервые делегировал права адресации памяти операционной системе, реализовав раздельное управление сегментными и логическими адресами. В программах больше не было необходимости явно настраивать сегментные регистры, поскольку система могла автоматически выполнять распределение памяти и контроль доступа. Таким образом, разработчики избавились от утомительной работы с низкоуровневой адресацией и смогли сосредоточиться непосредственно на программной логике.

Несмотря на это, ассемблер по-прежнему был сильно ограничен низкоуровневой аппаратной архитектурой. Огромные различия в наборах команд разных процессоров существенно затрудняли перенос программ на другие платформы. Разработчикам по-прежнему приходилось в совершенстве владеть навыками настройки регистров, отображения адресов памяти и другими низкоуровневыми аспектами, что делало разработку крайне сложной и неэффективной. Несмотря на то что появление ассемблера частично решило проблему написания кода, в условиях растущих вычислительных потребностей, особенно в области научных исследований и моделирования бизнес-процессов, возникла острая необходимость в языках программирования более высокого уровня абстракции.

2.1.3. Возникновение языков программирования высокого уровня

В 1950-х годах, по мере распространения компьютерных технологий из военной сферы в научно-исследовательскую и коммерческую, потребности в программировании стремительно специализировались, а концепция языков программирования сместилась с «ориентации на машину» к «ориентации на задачу». В этот период появились языки программирования высокого уровня, которые не только освободили мышление разработчиков, но и ознаменовали вступление разработки ПО в новую эру, ориентированную на бизнес-потребности и решение задач.

1. Родоначальник научных вычислений: Fortran

В 1957 году компания IBM выпустила язык Fortran (Formula Translation), что ознаменовало официальное начало новой эры языков высокого уровня. Язык Fortran стал первым в мире широко используемым языком высокого уровня, который больше не ограничивался непосредственным управлением аппарат-

ной структурой, а впервые позволил разработчикам описывать алгоритмы и логические процессы с помощью математических выражений.

В качестве языка программирования для научных вычислений Fortran поддерживает абстрактные конструкции, такие как циклы, условия и функции, и с помощью компилятора превращает исходный код в эффективное машинное представление. Его появление коренным образом изменило суть программирования, превратив разработчиков из специалистов по «вводу команд» в профессионалов по «решению задач».

В 1962 году была выпущена версия FORTRAN IV, которая ознаменовала собой значительный прорыв в области технологий компиляции, поскольку в ней были внедрены такие методы оптимизации, как **перемещение кода с неизменными циклами** (loop invariant code motion). Приведем пример следующего исходного кода:

```
DO 10 I = 1, 100
   ; Исходный код: вычисление значений массива A
   A(I) = B + C * I
   ; В выражении B+C*I переменная B является инвариантом цикла, то есть
   ; остается неизменной на протяжении всего цикла, поэтому ее значение можно
   ; вычислить заранее
10 CONTINUE
```

Компилятор может распознавать выражения в цикле, которые не изменяются при итерации, и переносить их для выполнения вне цикла, что позволяет значительно сократить количество инструкций в самом цикле. Ниже приведен пример кода, сгенерированного после оптимизации:

```
MOV AX, B              ; Загрузка значения переменной B в регистр AX
ADD AX, C              ; Вычисление значения B+C и сохранение его в AX
MOV DX, AX             ; Сохранение неизменной величины цикла B+C в DX
DO_START:
MOV [A+I], DX          ; Тело цикла выполняет только операции умножения
и сохранения
INC I                  ; Инкремент переменной индекса
CMP I, 100             ; Сравнение переменной индекса с верхним пределом цикла
JLE DO_START           ; Если не достигнут верхний предел 100, то цикл
продолжается
```

Эта оптимизация позволила сократить количество инструкций в теле цикла примерно на 40 %. На полупроводниковом компьютере IBM 7094 время вычисления обратной матрицы сократилось с 2 часов до 72 минут.

В 1978 году в версии FORTRAN 77 была внедрена технология автоматической векторизации (auto-vectorization), позволяющая преобразовывать операции с массивами в векторные инструкции суперкомпьютера Cray-1. Это в сотни раз повысило производительность линейно-алгебраических вычислений, таких как умножение матриц, что способствовало внедрению сложных вычислений, таких как анализ методом конечных элементов, прогнозирование погоды и моделирование землетрясений, из теоретических исследований в инженерную практику. Fortran до сих пор остается основным языком в области высокопроизводительных вычислений.

2. Стандартизация в коммерческой сфере: COBOL

В 1959 году под руководством Министерства обороны США был создан язык COBOL (Common Business-Oriented Language) с целью унификации стандартов языков обработки информации в государственных учреждениях и коммерческих организациях, а также решения проблем низкой эффективности разработки коммерческих программ, сложности их обслуживания и плохой переносимости.

Эпохальное значение языка COBOL заключается в том, что в нем впервые был использован стиль естественного языка в программировании, что сделало код понятным даже для нетехнических специалистов.

```
01 CHECK-RECORD.
05 CHECK-NUMBER PIC 9(8) COMP-3.
   ; Номер счета в сжатом десятичном формате (8 цифр)
05 ACCOUNT-NUMBER PIC 9(12) COMP-3.
   ; Номер счета (12 цифр, сжатое хранение)
05 AMOUNT PIC S9(10)V99 SIGN TRAILING.
   ; Сумма со знаком, с сохранением двух знаков после запятой
05 ENDORSEMENT-AREA PIC X(30).
   ; Область передаточной подписи (эндорсмента), строка из 30 символов
```

Структурированный синтаксис COBOL точно соответствует полям бизнес-форм и широко используется в банковской, телекоммуникационной, страховой и других отраслях.

В COBOL впервые были внедрены парадигма программирования, ориентированная на данные, и строгая система определения данных, позволяющая проводить проверку формата еще на этапе компиляции. Такое внимание к надежности и стандартизации послужило непосредственным источником идей для разработки более поздних языков корпоративного уровня, таких как Java и C#.

3. От арифметики к бизнесу: эволюция языков программирования в сторону многообразия

Fortran и COBOL легли в основу языков научных вычислений и коммерческих приложений соответственно. Их появление ознаменовало появление высокоуровневых языков программирования, разработанных для решения конкретных задач. Первый из них отражал стремление к максимальной производительности и выразительности алгоритмов, а второй способствовал снижению технической сложности кода: языки программирования больше не зависели от аппаратной архитектуры и постепенно превратились в инструменты для решения абстрактных задач и системного моделирования.

Высокоуровневые языки программирования этого периода улучшили эффективность разработки и способствовали появлению таких ключевых концепций, как технологии компиляторов, абстрактные модели и структуры данных. Это заложило прочную основу для последующего развития языков программирования, таких как C, Pascal и даже Java.

2.1.4. Прорыв в области структурного и объектно-ориентированного программирования

Благодаря стремительному развитию индустрии интегральных микросхем масштабы программных систем резко увеличились, зачастую превышая 100 000 строк кода, и проблемы традиционного линейного программирования в плане читаемости и удобства техподдержки становились все более очевидными. Несмотря на то что языки высокого уровня уже обеспечивали моделирование предметных областей, перед создателями огромных и сложных программных систем возникла острая необходимость в более строгой логической структуре и более высоком уровне абстракции.

В этот период языки программирования прошли путь от процедурного подхода к структурированному и объектно-ориентированному программированию, что перестроило архитектуру программ и заложило основу современных парадигм разработки ПО.

1. Эпоха структурированного программирования: Pascal

Язык Pascal, официально представленный в 1970 году, является ярким примером концепции структурированного программирования. В Pascal особое внимание уделяется четкой структуре программы, строгим ограничениям типов и модульному дизайну, что делает его подходящим как для обучения, так и для широкого применения в реальных проектах.

Механизм статической проверки типов в Pascal позволяет выявлять большинство ошибок типов еще на этапе компиляции благодаря созданию символьной таблицы с именами переменных, типами и областями видимости. Например:

```
[LINE 15] ERROR: CANNOT ASSIGN BOOLEAN VALUE „TRUE“ TO INTEGER VARIABLE „COUNT“
```

Это сделало Pascal одним из первых языков, способных выявлять большинство типовых ошибок на этапе компиляции, что значительно повысило эффективность предварительного контроля качества GJ.

В 1983 году компания Borland выпустила Turbo Pascal, в котором была внедрена технология инкрементной компиляции. Благодаря записи временных меток изменений исходных файлов и перекомпиляции только измененных модулей время компиляции проекта объемом 100 000 строк кода сократилось с 60 до 8 минут. Этот механизм впоследствии активно использовался в компиляторах Jikes для Java и Roslyn для C#.

В Pascal примечательна модульная архитектура, которая позволяет объединять функции в отдельные модули с помощью ключевого слова `UNIT`. Например:

```
UNIT MathUtils;
INTERFACE
FUNCTION Add(a, b: INTEGER): INTEGER;  // Описание интерфейса
IMPLEMENTATION
FUNCTION Add(a, b: INTEGER): INTEGER;  // Реализация интерфейса
BEGIN
    Add := a + b;
END;
END.
```

Этот механизм инкапсуляции позволил повысить степень повторного использования кода, а также послужил прототипом для последующих заголовочных файлов в языке C и системы пакетов в Java, способствуя переходу программирования от модели разрозненного накопления логики к современному инженерному подходу модульной сборки.

2. Зарождение объектно-ориентированной архитектуры: Smalltalk

В 1980 году релиз Smalltalk-80 ознаменовал радикальный переход языков программирования от принципа «разложения на процедуры» к принципу «объектного моделирования». В Smalltalk была полностью систематизирована концепция **объектно-ориентированного программирования** (Object-Oriented Programming, OOP), основной идеей которой является принцип «все являются объектами».

Smalltalk определяет объект как «независимую единицу, инкапсулирующую состояние (данные) и поведение (методы)», взаимодействие в которой осуществляется посредством обмена сообщениями. Его основными механизмами являются следующие.

(1) **Инкапсуляция**: например, объект `BankAccount` объединяет баланс и методы перевода средств в неразделимое целое.

```
"BankAccount – это класс, инкапсулирующий баланс и методы операций"
Object subclass: #BankAccount
    instanceVariableNames: 'balance'
    "Именные переменные экземпляра: balance (баланс)"
    classVariableNames: ''
    category: 'Finance'

"Метод инициализации: установка начального баланса равным 0"
BankAccount >> initialize
    "Метод initialize: инициализация состояния счета"
    balance := 0.

"Метод deposit: добавление указанной суммы к балансу"
BankAccount >> deposit: amount
    "Метод deposit: принимает параметр amount (сумма вклада)
    и обновляет баланс"
    balance := balance + amount.

"Метод transfer: перевод средств с текущего счета и вызов
метода deposit целевого счета"
BankAccount >> transfer: amount to: otherAccount
    "Метод transfer:to:
       • amount – сумма перевода
       • otherAccount – объект BankAccount, получающий перевод
       Операция: сначала проверяется баланс, затем выполняется
       списание средств и пополнение счета получателя,
       либо генерируется ошибка"
    (balance >= amount)
        ifTrue: [
            balance := balance - amount.
            "Связанное поведение: списание средств с баланса текущего счета"
```

```
            otherAccount deposit: amount
            "Вызов метода deposit: получателя для зачисления средств"
        ]
        ifFalse: [
            self error: 'Insufficient balance'
            "Генерирует ошибку при недостаточном балансе"
        ].
```

Такой механизм контроля доступа позволяет внешнему коду обращаться к данным только через заранее определенные интерфейсы, что повышает безопасность и согласованность на программном уровне.

(2) **Наследование**: моделирование реалий посредством иерархии классов, например класс `SavingsAccount` наследует класс `BankAccount` и расширяет логику расчета процентов.

```
BankAccount subclass: #SavingsAccount
    instanceVariableNames: 'interestRate'
    "Добавлена переменная процентной ставки"
    classVariableNames: ''
    category: 'Finance'

SavingsAccount >> initiali
    super initialize.
    "Вызов метода инициализации родительского класса"

    interestRate := 0.03.
    "Установка годовой процентной ставки по умолчанию"

SavingsAccount >> calculateInterest
    balance := balance * (1 + interestRate).
    "Расширение функциональности родительского класса, расчет процентов"
```

Такое естественное моделирование отношения «is-a» (используется для обозначения отношения наследования, т. е. подкласс, или производный класс) является специализацией родительского, или базового, класса) не только повышает возможность повторного использования кода, но также обеспечивает интуитивное представление концепций предметной области на уровне классов, что значительно повышает эффективность разработки.

(3) **Полиморфизм**: благодаря абстрактному унифицированному интерфейсу родительского класса подклассы могут реализовывать дифференцированную логику в соответствии со своей семантикой, обеспечивая динамическую привязку во время выполнения. Например:

```
AbstractObject subclass: #AbstractAccount
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'Finance'

AbstractAccount >> withdraw: amount
    self subclassResponsibility.
    "Обязательная реализация метода снятия средств в подклассе"
```

```
BankAccount >> withdraw: amount
    balance := balance - amount.
    "Реализация снятия средств для обычного счета"

CreditAccount >> withdraw: amount
    (balance + overdraftLimit) >= amount
        ifTrue: [
            balance := balance - amount.
            "Реализация снятия средств при разрешенном овердрафте"
        ]
        ifFalse: [
            self error: 'Overdraft exceeded'].
```

При вызове `account withdraw: 100` система динамически определяет логику, которая подлежит выполнению, в зависимости от конкретного типа объекта. Это обеспечивает программе более высокую возможность расширения и гибкость архитектуры.

Кроме того, в Smalltalk впервые был предложен паттерн «модель–представление–контроллер» (Model-View-Controller, MVC), который впервые обеспечил ортогональную развязку данных, представления и взаимодействия, став теоретической отправной точкой для архитектур Java Swing, iOS UIKit и Web MVVM.

3. От синтаксических правил к системному мышлению

Структурное программирование обеспечило возможность создания программ с четкой логической упорядоченностью за счет модульного проектирования, проверки типов и стандартизации потока управления. Объектно-ориентированное программирование, в свою очередь, наделило языки способностью моделировать мир реальных объектов, способствуя переходу разработки ПО от процедурного подхода к объектно-ориентированному и архитектурно-ориентированному.

Языки Pascal и Smalltalk стали ключевыми достижениями в области управляемости и возможности моделирования языков программирования. Они определили базовые парадигмы современного языкового проектирования, а также обеспечили теоретическую основу и практические шаблоны для появления нового поколения языков, таких как C++, Java и Delphi. В результате языки программирования превратились из инструментов описания алгоритмов в инструменты построения сложных систем.

2.1.5. Разнообразие современных моделей программирования

Начиная с 90-х годов XX века, вместе с распространением интернета и широким внедрением распределенных вычислений, программные системы становились все более сложными, а требования к разработке стремительно менялись. Один-единственный подход к программированию уже не мог удовлетворить разнообразные сценарии, и его заменила тенденция к использованию различных моделей, сочетающих объектно-ориентированный, функциональный, событийно-ориентированный и скриптовый стили.

Языки Java, JavaScript, C++ и Python занимают доминирующее положение в таких ключевых областях, как корпоративные системы, веб-разработка, системные низкоуровневые технологии и интеллектуальный анализ данных, составляя четыре основных столпа современной экосистемы языков программирования.

1. Эпохальный лидер кросс-платформенности: Java

В 1995 году компания Sun Microsystems выпустила язык Java, который, следуя концепции «написал один раз – работает везде», установил новую парадигму разработки кросс-платформенных приложений.

(1) Архитектура байт-кода и кросс-платформенная революция. Компилятор Java превращает исходный код в независимый от платформы байт-код (файлы .class), который интерпретируется и выполняется виртуальной машиной Java (Java Virtual Machine, JVM). В сочетании с технологией компиляции «just-in-time» (JIT) наиболее часто используемый код преобразуется в машинный код. Это позволяет достичь производительности, приближающейся к 70 % от производительности C++. Например:

```
public class HelloWorld {
    public static void main(String[] args) {

        // 1. Загрузчик классов JVM (ClassLoader) загружает байт-код
        // HelloWorld.class

        // 2. Интерпретатор JVM (Interpreter) построчно выполняет
        // инструкции байт-кода

        // 3. Если System.out.println и его цепочка вызовов часто выполняются,
        // компилятор JIT преобразует эти "горячие точки"
        // в локальный машинный код, благодаря чему при последующих вызовах
        // можно напрямую запускать локальный код, что значительно повышает
        // производительность

        System.out.println(«Hello, World!»);
    }
}
```

Такая многоуровневая архитектура позволяет Java-программам беспрепятственно переключаться между операционными системами Windows, Linux, macOS и другими. Это превращает Java в предпочтительный язык для корпоративных систем и операционных систем для встраиваемых устройств.

(2) Проектные стандарты и упрощение синтаксиса. Java упрощает сложный синтаксис C++, сохраняя при этом целостность объектно-ориентированного подхода.

Механизм одноуровневого наследования плюс интерфейсы: позволяет избежать проблем ромбовидного наследования (diamond inheritance) и повышает гибкость.

Автоматическая сборка мусора (garbage collection, GC): повышает безопасность использования памяти и снижает риск потери данных.

Структурированная обработка исключений: усиливает отказоустойчивость системы с помощью механизма try-catch-finally. Например:

```
try (FileInputStream fis = new FileInputStream("data.csv")) {
    // Бизнес-логика
} catch (IOException e) {
    logger.error("Исключение при работе с файлом: {}", e.getMessage());
}  finally {
    // Независимо от возникновения исключения необходимо закрыть файловый
    // поток для освобождения ресурсов
    if (fis != null) {
        try {
            fis.close();
        } catch (IOException ex) {
            logger.error("Исключение при закрытии файла: {}", ex.getMessage());
        }
    }
}
```

(3) Основа корпоративной разработки. В Java EE (ныне Jakarta EE) были закреплены корпоративные стандарты, такие как Enterprise JavaBean (EJB), Servlet и Java Server Pages (JSP). В сочетании с механизмами инъекции зависимостей и обратного управления в рамках Spring они заложили основу для формирования доминирующей модели корпоративной архитектуры.

После выпуска Android SDK в 2008 году Java стал основным языком мобильной разработки, способствуя формированию единой языковой экосистемы для фронтенда, серверной части и мобильных устройств.

(4) Непрерывная эволюция производительности и экосистемы. В Java 8 были введены Lambda-выражения и Stream API, что позволило приблизиться к функциональной парадигме; ZGC-сборщик, представленный в Java 17, позволяет ограничить паузы при работе с большими объемами памяти до 10 мс, что актуально для финансовых, телекоммуникационных и других систем, чувствительных к задержкам.

2. От скриптов для фронтенда к полнофункциональному движку: JavaScript

Язык JavaScript, изначально задуманный как язык скриптов для браузеров, после появления Node.js в 2009 году завершил переход от фронтенда к бэкенду и стал основным языком современной полнофункциональной веб-разработки.

(1) Событийно-ориентированная модель и революция асинхронности. В Node.js используется движок V8 для выполнения кода JavaScript, а механизм цикла событий реализован с помощью библиотеки libuv, которая обеспечивает точную координацию асинхронных задач, таких как ввод-вывод (input/output, I/O), таймеры и микрозадачи, что позволяет адаптироваться к сценариям с высокой степенью параллелизма. Например:

```
// Пример событийно-ориентированного асинхронного программирования
document.getElementById(«btn»).addEventListener("click", () => {
    fetch("https://api.example.com/data")
        .then(response => response.json())
        .then(data => updateUI(data))
        .catch(error => console.error(«Ошибка запроса:», error));
});
```

Шлюз API Netflix с помощью Node.js обрабатывает 20 млрд запросов ежедневно, сокращая время отклика до 50 мс, что свидетельствует о высокой пропускной способности Node.js в системах реального времени.

(2) Единый язык для полнофункциональной разработки. JavaScript обеспечивает совместное использование языка в браузере и на сервере.

- **Фронтенд**: фреймворки React/Vue и др. создают высокопроизводительные интерактивные страницы с помощью виртуального DOM.
- **Бэкенд**: фреймворки Express, Koa и др. поддерживают создание API-сервисов с высокой параллельностью.
- **Кросс-платформенная разработка**: фреймворки Electron, React Native и др. реализуют кросс-платформенную разработку для настольных и мобильных устройств.

3. Вечная классика системного программирования: C++

C++ был разработан Бьярне Страуструпом (Bjarne Stroustrup) в 1985 году. Сохранив высокую производительность языка C, он ввел механизмы объектно-ориентированного программирования и генерализации, став представителем языков смешанной парадигмы.

(1) Полиморфизм и объектное моделирование. C++ поддерживает полиморфизм во время выполнения с помощью механизма виртуальных функций, что позволяет отделить вызов интерфейса от его реализации. Например:

```
// Абстрактный базовый класс Shape определяет интерфейс
// (чисто виртуальную функцию), подклассы должны реализовывать draw()
// Ключевое слово virtual и =0 означают, что это чисто виртуальная функция,
// что превращает Shape в абстрактный класс
class Shape {
public:
    // Интерфейс: метод рисования, точка входа полиморфизма во время выполнения
    virtual void draw() = 0;

    // Виртуальный деструктор, гарантирующий правильный вызов деструктора
    // подкласса при удалении через указатель на базовый класс
    virtual ~Shape() = default;
};

// Circle наследуется от Shape и реализует метод draw()
class Circle : public Shape {
public:
    // override явно указывает, что данный метод переопределяет виртуальную
    // функцию базового класса
```

```
    void draw() override {
        // **Конкретная реализация**: здесь рисуется круг
    }
};

// Функция рендеринга, вызывающая draw() с помощью ссылки на базовый класс
// При вызове не происходит статическая привязка, а выполняется один переход
// по указателю через vtable (таблицу виртуальных функций), динамически
// принимая решение о том, какую реализацию draw() вызвать, что обеспечивает
// развязку интерфейса от реализации
void renderShape(Shape& shape) {
    // Ищет правильный адрес функции в vtable, на которую указывает vptr,
    // и вызывает ее
    shape.draw();
}

// Схема внутреннего механизма: каждый полиморфный объект содержит в памяти
// скрытый указатель vptr, указывающий на vtable класса (массив указателей
// на функции), в vtable хранятся фактические адреса всех виртуальных функций,
// при вызове renderShape компилятор генерирует код, подобный
// (*shape.vptr->draw)()
// Затраты только на один косвенный вызов
```

(2) Шаблоны и общие типы. Механизм шаблонов в C++ позволяет генерировать специальные реализации контейнеров для различных типов на этапе компиляции, что обеспечивает нулевую нагрузку абстракции без потери производительности и без зависимости от типа.

Возьмем в качестве примера `std::vector<T>` из стандартной библиотеки шаблонов (STL): он полностью соответствует приведенному ниже упрощенному варианту `Vector<T>` с точки зрения механизма работы: при наличии в коде `Vector<int>` или `std::vector< std::string>` компилятор заменяет все `T` в теле шаблона на целевой тип и генерирует реальный машинный код.

```
// Упрощенный пример: пользовательский шаблонный класс Vector<T>
// для реализации универсального контейнера
// Использование шаблонов C++ (template) для реализации “универсального
// программирования”: код не зависит от типа T и поддерживает хранение данных
// любого типа

// Определение шаблона: T – параметр типа, заполнитель.
// При компиляции генерируется конкретная версия в зависимости
// от фактического типа
template <typename T>
class Vector {
private:
    T* data;      // Универсальный указатель: указывает на массив любого типа T
    size_t size;  // Текущее количество хранимых элементов
public:
    void push_back(const T& value) {
        // Логика универсальной вставки: поддерживает вставку элементов любого
        // типа, не зависит от конкретной реализации T
```

```
        // На практике std::vector также обрабатывает расширение емкости,
        // динамическое выделение памяти и т. д., что здесь опущено
    }
};

// Инстанцирование при компиляции: для каждого экземпляра Vector<T>
// генерируется отдельное определение класса (развертка шаблона)
// Две следующие строки кода “переводят” шаблон в классы двух конкретных типов:
// int и std::string

// Компилятор сгенерировал реализацию data as int*,
// push_back(const int&) и т. д.
Vector<int> ints;

// Компилятор сгенерировал реализацию data as string*
// push_back(const std::string&) и т. д.
Vector<std::string> strings;

// Вектор в STL является полностью универсальным шаблоном контейнера, обладающим
// расширенным набором функций и обеспечивающим безопасность

// Использование универсального шаблона для инстанциирования
// контейнера типа double
std::vector<double> dv;

// Пользовательские типы также могут использоваться в качестве параметров
// шаблона, обеспечивая типовую безопасность
std::vector<Customer> customers;

// Вывод: класс Vector, реализованный с помощью template<typename T>,
// является примером универсального контейнера.
// Типовой параметр T может быть заменен на любой тип, что обеспечивает
// гибкость повторного использования кода и типовую безопасность
```

Этот универсальный (generic) механизм позволяет программе при определении алгоритмов или структур данных не указывать конкретные типы, а реализовывать проект без привязки к типу с помощью типовых параметров. Компилятор при выполнении генерирует отдельную версию для каждого конкретного типа, что позволяет обеспечить повторное использование кода при одновременном сохранении безопасности типов и эффективности выполнения. Поскольку все замены типов выполняются на этапе компиляции, в процессе выполнения нет дополнительных нагрузок, и поэтому такой подход считается беспроигрышным с точки зрения производительности. Благодаря этому шаблоны стали основой проектирования современных библиотек алгоритмов (таких как C++ STL и Java Collections), что способствовало широкому применению универсальных контейнеров и высокоэффективных алгоритмов.

(3) Лучший выбор для системного программирования. Благодаря детализированному управлению памятью, прямому доступу к аппаратному обеспечению и абстракции с нулевыми затратами язык C++ на протяжении долгого времени занимает центральное место в таких областях разработки низкоуров-

невого и ресурсоемкого ПО, как операционные системы Windows и Linux, ядра баз данных, а также игровые движки (например, Unreal Engine).

4. Язык данных с приоритетом лаконичности: Python

В 1991 году Гвидо ван Россум (Guido van Rossum) представил язык Python, ставший предпочтительным выбором для разработки в области науки о данных и искусственного интеллекта благодаря своему синтаксису, отличающемуся изяществом, точностью и лаконичностью.

(1) Минималистичный синтаксис и выразительность. В Python используются отступы для обозначения структуры фрагментов кода. Кроме того, язык предлагает такие усовершенствованные синтаксические конструкции, как списочные выражения и генераторы, что повышает читаемость кода и эффективность разработки. Например:

```
# Списочная конструкция: вывод квадратов чисел от 1 до 10
squares = [x**2 for x in range(1, 11)]

# Генератор: ленивая (lazy) генерация больших массивов чисел, экономия памяти
large_numbers = (
    x for x in range(1, 1000000)
    if x  % 3 == 0
)
```

(2) Три основных компонента науки о данных. В Python представлен полный набор инструментов для работы с данными.

NumPy: обеспечивает высокопроизводительные матричные вычисления со скоростью, близкой к скорости выполнения в языке C. Широко используется в научных вычислениях и численном анализе.

pandas: по сравнению с традиционным методом написания циклов вручную (например, с использованием встроенных в Python списков и словарей), эффективность обработки данных может быть повышена в 3–5 раз.

Matplotlib/Seaborn: поддерживают точную настройку графиков, позволяя получать визуализацию в соответствии с требованиями к верстке публикаций.

Например, приведенный ниже код демонстрирует типичный процесс очистки данных и групповой статистики с использованием *pandas*.

```
import pandas as pd

data = pd.read_csv("data.csv")
clean_data = data.dropna().groupby("category").mean()
```

(3) Лидер экосистемы ИИ.

Изначально Jupyter Notebook развился из проекта IPython, специально разработанного для интерактивных вычислений на Python и входящего в экосистему Python. Он поддерживает не только написание и выполнение кода, но также встраивание текста, диаграмм и математических формул, создавая «исполняемые документы». Благодаря глубокой интеграции Notebook с Python комфорт его использования значительно превосходит аналогичные интерфейсы в других языках (таких как R или Julia).

Благодаря свойству Python выступать в качестве связующего языка с возможностью бесшовного вызова высокопроизводительных библиотек на C/C++ и Fortran основные фреймворки в экосистеме ИИ и машинного обучения, такие как TensorFlow и PyTorch, используют Python в качестве языка интерфейса. Благодаря сочетанию с Jupyter Notebook исследователи могут оперативно создавать, тестировать и документировать свои модели. Это обеспечивает базовую производительность для построения систем искусственного интеллекта, научных вычислений и обработки больших данных.

2.2. Эволюция взаимодействия при программировании

Способы взаимодействия между человеком и программой постоянно эволюционировали на протяжении всего развития сферы разработки ПО. От первоначальных физических методов ввода данных с помощью перфокарт и панельных переключателей до текстового интерфейса командной строки, а затем до графической среды и интегрированных платформ разработки – способы взаимодействия программиста с кодом подвергались постоянным изменениям.

Все эти трансформации не только изменили стиль написания программ, но также перестроили модель мышления программистов и рабочий процесс в целом. В этом разделе мы проследим путь этих изменений и заглянем в прошлое, чтобы проследить, как разработчики шаг за шагом продвигались к более эффективному и интеллектуальному взаимодействию человека и машины.

2.2.1. Зачатки программирования на физических носителях

На заре появления электронных компьютеров взаимодействие программистов с вычислительной техникой в значительной степени зависело от физических носителей.

- Перфокарты и программирование с помощью перфокарт. Программисту приходилось пробивать двоичные команды на бумажной ленте шириной 16 столбцов с помощью специальной перфорационной машины (см. рис. 2-6). Для выполнения программы, состоящей из 100 операций сложения, требовалось пробить более 4000 отверстий. Любое изменение логики означало необходимость изготовления новой ленты, а исправление одного адреса данных могло потребовать нескольких часов на перекомпоновку последовательности отверстий. Каждая корректировка кода сопровождалась переделкой физического носителя.
- Прямой ввод через панель управления. В компьютерах, представленных моделью IBM 650, каждая команда вводилась вручную с помощью двоичных переключателей на панели управления, при этом для каждого байта требовалось 8 переключений и подтверждение с помощью индикаторов. Этот способ был не только трудоемким и отнимал много времени, но также чрезвычайно сложным в обслуживании, поскольку по сути код становился отображением аппаратных схем.

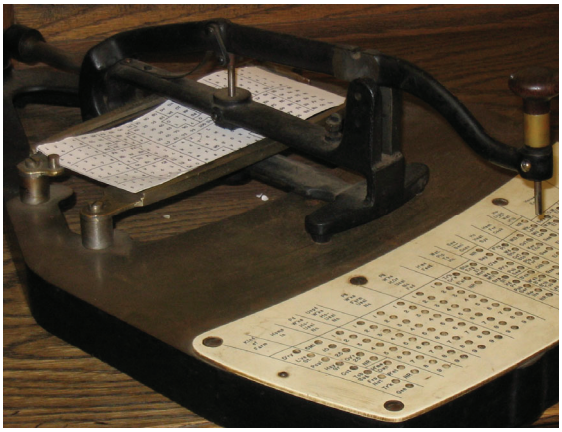


Рисунок 2-6. Специализированная перфорационная машина (изображение взято из Википедии)

На этом этапе программисты в большей степени работали с аппаратным обеспечением машины, а не занимались написанием абстрактной логики. Эффективность модификации и отладки кода практически полностью зависела от физических операций, что представляло собой большую инженерную проблему.

2.2.2. Переход от строчного редактирования к полноэкранному взаимодействию

Распространение операционных систем с разделением времени в 1960-х годах привело к появлению текстовых редакторов на основе символьного интерфейса. Редакторы в среде дисковой операционной системы (Disk Operating System, DOS) стали важным инструментом коммерческой разработки и программирования на персональных компьютерах той эпохи.

(1) Типичный представитель строчных редакторов: EDLIN. В 1980 году Тим Патерсон (Tim Paterson) написал для 86-DOS (QDOS) программу EDLIN – командно-строковый редактор, призванный заменить утомительный ввод данных с перфокарт. Впоследствии EDLIN был включен в релиз MS-DOS/PC-DOS 1.0 в 1981 году и стал первым широко используемым строчным редактором. Разработчикам приходилось вводить код построчно в формате `10 PRINT "HELLO"`, а для внесения изменений они использовали команды `E` (редактировать строку), `I` (вставить строку), `D` (удалить строку) и т. д. Так, например, для исправления логической ошибки требовалось сначала отобразить строки кода с помощью команды `LIST`, а затем ввести `60 CHANGE` для перехода к строке с ошибкой. Такой линейный подход к редактированию был продолжением мышления, привычного при работе с перфокартами: изменение кода ограничивалось уровнем строк, а для корректировки сложной логики приходилось постоянно переходить по номерам строк. Несмотря на низкую эффективность, именно этот подход заложил основу для последующих текстовых редакторов.

(2) Прорыв в области коммерческих редакторов: WordStar. Выпущенный в 1978 году WordStar (см. рис. 2-7) был первым коммерческим полноэкранным редактором на платформе DOS с поддержкой отображения текста в 80 столб-

цах и простым управлением форматированием. В WordStar была введена концепция «блочных операций». Так, например, с помощью ^KB и ^KK можно было выделить блок кода, а с помощью ^Y осуществлялось удаление всей строки. Это позволило перейти от работы с кодом на уровне отдельных символов к работе на уровне отдельных фрагментов. Кроме того, функция имитации печати (^KI) обеспечивала предварительный просмотр форматирования и была важным инструментом на ранних этапах создания документации.

Из-за ограничений механизма сегментов памяти DOS (каждая программа могла напрямую обращаться максимум к 64 КБ данных) при редактировании больших документов нередко возникали проблемы с переполнением памяти, замедлением отклика или ограничением функциональности, что было особенно заметно при выполнении таких операций, как предварительный просмотр страниц или переход по длинному документу. Пользователям приходилось чаще сохранять и разбивать документы вручную, что увеличивало риск возникновения ошибок.



Рисунок 2-7. Снимок экрана WordStar 3.0 (автор изображения: Plenz)

Чтобы справиться с этими ограничениями, в WordStar были внедрены инновационные технологии «обмена дисками» (disk swapping), позволяющие временно сохранять часть содержимого документа в буфере на диске и динамически загружать редактируемый фрагмент. Кроме того, поддерживаемые команды перехода между абзацами (такие как ^QR для перехода на страницу назад и ^QC для перехода на страницу вперед) обеспечивали эффект имитации разбиения на страницы, позволяя осуществлять быстрое позиционирование и навигацию по большим файлам. Это обеспечивало WordStar способность обрабатывать тексты объемом в сотни страниц даже на первых персональных компьютерах с ограниченными ресурсами. В результате WordStar стал предпочтительным инструментом для создания коммерческих текстов в начале 1980-х годов.

Технически эти редакторы эпохи DOS впервые реализовали на персональных компьютерах редактирование текста по принципу «что видишь, то и получаешь» (What You See Is What You Get, WYSIWYG). Несмотря на ограничения производительности ввиду низкой мощности аппаратного обеспечения, такая концепция взаимодействия оказала глубокое влияние на последующие разработки графических интерфейсов и даже на современные интегрированные среды разработки (IDE).

2.2.3. Эпоха интеграции: объединение редактирования, компиляции и отладки

С наступлением 1980-х годов, по мере распространения графических интерфейсов и персональных компьютеров, редакторы постепенно эволюционировали от простых текстовых инструментов до многофункциональных платформ.

(1) Интегрированная парадигма Turbo Pascal. В 1983 году компания Borland выпустила Turbo Pascal (см. рис. 2-8), где впервые в едином интерфейсе были объединены три основные функции: редактирование, компиляция и отладка. Пользователь мог запускать компиляцию одним нажатием клавиши **F9** и устанавливать точки останова для отладки. При отладке поддерживалось пошаговое выполнение и просмотр регистров, что значительно сократило цикл разработки. Эта версия считается прототипом современных интегрированных сред разработки (IDE).

(2) Внедрение интеллектуального распознавания: Visual Studio 6.0. В 1998 году в Visual Studio 6.0 была реализована функция IntelliSense, которая обеспечивала подсказки по синтаксису и автодополнение. Например, при вводе `"std:"`отображались все STL-контейнеры C++. Одновременно разработчики смогли просматривать адрес переменной в памяти и соответствующее отображение памяти, а также быстро переходить к месту ее определения и использования, щелкнув по ней правой кнопкой мыши. Эти интерактивные функции значительно улучшили возможности навигации между файлами и восприятия кода, сделав разработку и сопровождение крупных проектов более эффективными.

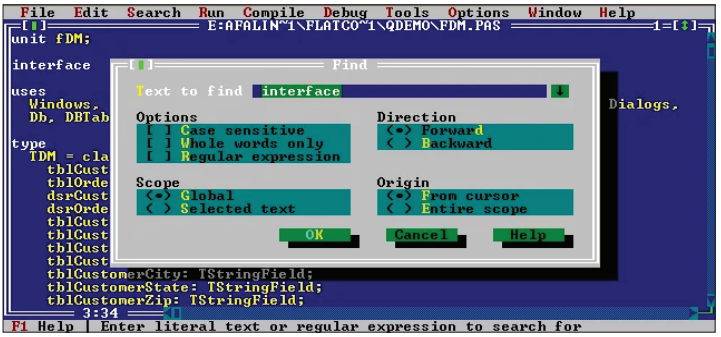


Рисунок 2-8. Turbo Pascal 2.0

(3) Веха в разработке на Java. Выпущенный в 1999 году Eclipse (см. рис. 2-9) быстро стал основным инструментом для разработки на Java благодаря своей модульной архитектуре. Его экосистема с открытым исходным кодом привлекла множество сторонних расширений, поддерживающих интегрированную разработку Java EE, баз данных, инструментов сборки и т. д. В свою очередь, NetBeans стал эталоном кросс-платформенной разработки благодаря визуальному конструктору и поддержке нескольких языков, а его модульная архитектура оставила глубокий след в истории.

Появление интегрированных редакторов дало разработчикам возможность выполнять полный цикл действий на одной платформе (редактирование–компиляция–отладка), что значительно повысило эффективность и способствовало стандартизации процессов разработки ПО.

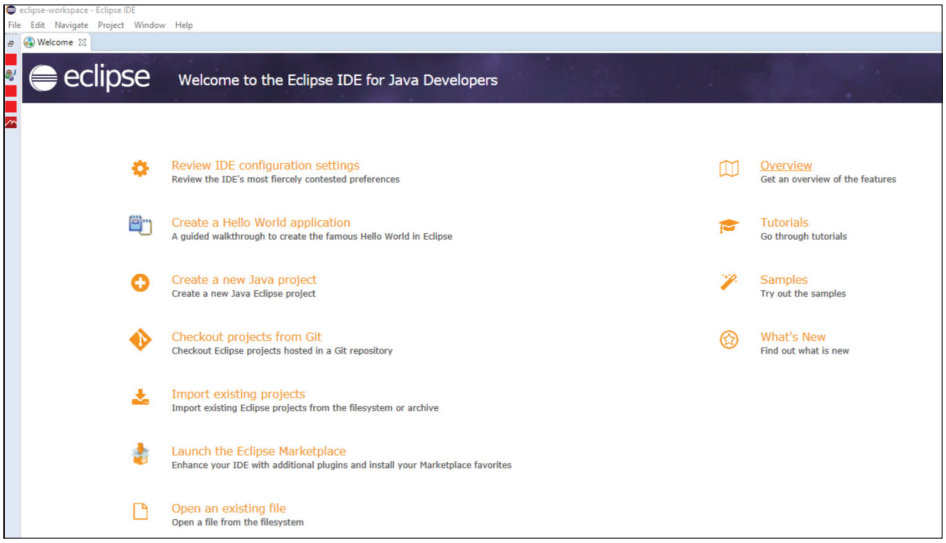


Рисунок 2-9. Редактор Eclipse

2.2.4. Кросс-платформенная экосистема и реформа архитектуры плагинов

В 2015 году с выходом Visual Studio Code (VS Code) началась новая эра редакторов: архитектура этого инструмента обеспечила беспрецедентную гибкость и возможности расширения.

(1) Протокол языкового сервера. Благодаря **протоколу языкового сервера** (Language Server Protocol, LSP), который отделяет логику обработки языка от интерфейса редактора, VS Code изначально поддерживает более ста языков. В качестве примера можно привести разработку на языке Go: в режиме реального времени обнаруживаются ошибки типа undeclared name: x (где переменная x не объявлена) и предлагаются рекомендации по исправлению, что позволяет эффективно выявлять проблемы еще на этапе кодирования и значительно повышает качество кода.

(2) Максимальная открытость архитектуры плагинов. Разработчики могут создавать плагины на базе Node.js. Например, плагин GitLens обеспечивает отслеживание истории коммитов, а плагин Docker интегрирует управление образами. По состоянию на 2023 год на рынке плагинов VS Code насчитывается более 200 000 вариантов, которые покрывают все возможные сценарии использования – от квантовых вычислений и фронтенд-фреймворков до платформ с низким уровнем кодирования. Это позволило сформировать экосистему по принципу «легкое ядро + расширение с помощью плагинов».

Являясь эталоном современных редакторов, VS Code с его лаконичным интерфейсом (см. рис. 2-10) объединяет такие функции, как редактирование кода, отладка и управление версиями, а также обеспечивает высокую степень персонализации с помощью тем и плагинов.

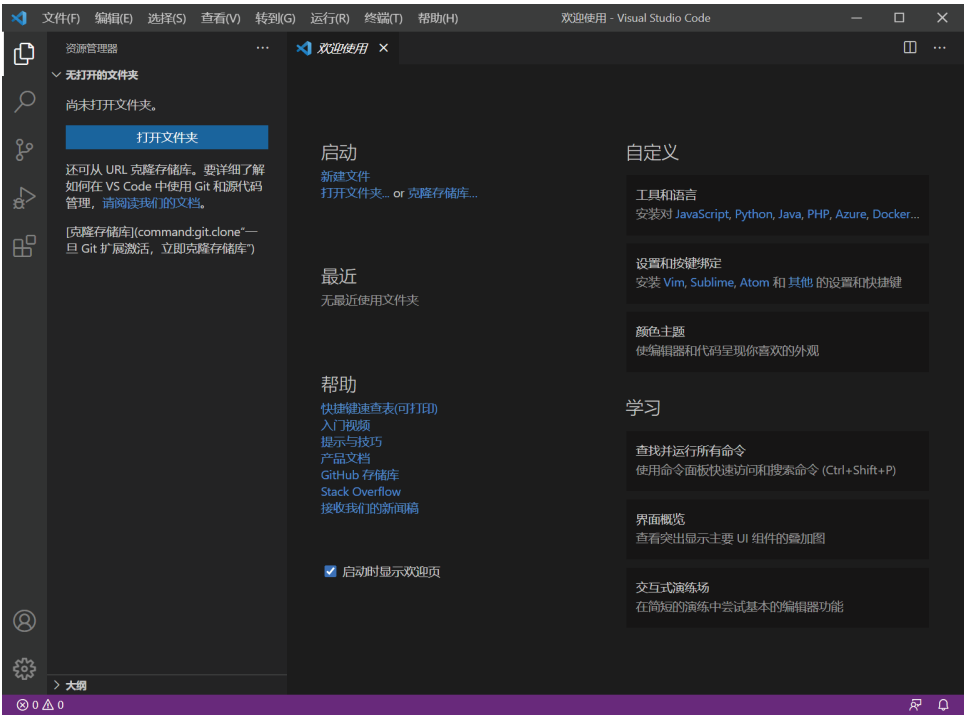


Рисунок 2-10. Редактор VS Code

Наряду с VS Code серия редакторов JetBrains продолжает углубляться в нишевые области. Java-редактор IntelliJ IDEA поддерживает такие продвинутые функции, как сворачивание кода, предварительный просмотр рефакторинга и визуализация инъекции зависимостей. PyCharm, специально разработанный для Python, интегрирует инструменты визуализации потоков и параллелизма, а также управление контейнерами Docker, став предпочтительным выбором для разработки приложений в области данных и искусственного интеллекта. Xcode, являясь официальной IDE экосистемы Apple, значительно повышает эффективность разработки мобильных приложений благодаря визуальному конструктору Interface Builder и интеграции с моделями Core ML.

На этом этапе редакторы эволюционировали от «языковых инструментов» в многоязычные, кросс-платформенные и расширяемые решения с помощью плагинов платформы разработки, став универсальной основой для разработчиков полного цикла.

2.3. Возникновение разработки с минимальным кодом и без кода

На волне цифровой трансформации предприятий традиционная модель разработки кода стала тяжким бременем, сдерживающим инновационный рост компаний. Возьмем, к примеру, систему управления средним предприятием: от анализа требований к продукту и проектирования архитектуры до написа-

ния кода и многократной отладки профессиональной команде зачастую требуется от 3 до 6 месяцев. Как только требования к продукту меняются, затраты на модификацию растут как снежный ком, а сроки сдачи проекта постоянно затягиваются. В то же время на рынке разработчиков наблюдается дефицит кадров, и из-за нехватки квалифицированных специалистов компании вынуждены вкладывать огромные средства в набор и обучение персонала, при этом по-прежнему не могут избавиться от проблемы кадрового дефицита.

Развитие облачных технологий и искусственного интеллекта способствует переходу от ручного программирования к визуальному конструированию. В результате появились платформы с низким уровнем кодирования (low-code) и без кодирования (no-code), которые значительно снижают порог входа в разработку и повышают эффективность внедрения благодаря графическому интерфейсу, операциям методом перемещения и библиотеке готовых компонентов.

В рамках этой экосистемы вайб-программирование в качестве новейшей модели разработки интеллектуальных систем завоевало популярность благодаря глубокому семантическому анализу и механизму автоматической генерации кода, что обеспечило автоматическое конвертирование описываемых требований в рабочий код.

2.3.1. Теоретическая система полноты по Тьюрингу

Глубокое понимание преимуществ и ограничений платформ с низким уровнем кодирования и без кода требует прежде всего обращения к основополагающей концепции компьютерных наук: «полноте по Тьюрингу».

В 1930-х годах британский математик Алан Мэтисон Тьюринг (Alan Mathison Turing) предложил концепцию «машины Тьюринга», схематическое изображение которой приведено на рис. 2-11. Эта абстрактная вычислительная модель используется для исследования «проблемы решаемости»: существует ли универсальный алгоритм, способный определить истинность или ложность любого математического утверждения.

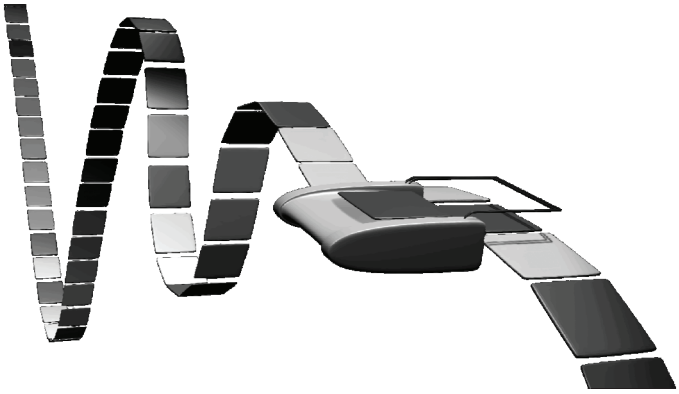


Рисунок 2-11. Наглядное представление машины Тьюринга (изображение взято из Википедии)

Машина Тьюринга состоит из бесконечной ленты, считывающего и записывающего устройства, а также контроллера состояний. С помощью простых пра-

вил она теоретически может моделировать любой вычислительный процесс. Машина Тьюринга определила границы «вычислимости», заложила теоретическую основу для развития информатики и стала ключевым стандартом для оценки возможностей вычислительных систем.

После этого, по мере развития компьютерных технологий, в 1940-х годах появился первый универсальный электронный компьютер ENIAC. Хотя ENIAC использовался для военных вычислений, он был способен выполнять многозадачную обработку; в свою очередь, архитектура фон Неймана (von Neumann architecture) представляет собой концептуальную структуру компьютерного дизайна, объединяющую память для программных инструкций и память для данных. Эта архитектура, основанная на идее «программируемой памяти», усовершенствовала дизайн компьютеров и привела их к модели машины Тьюринга.

В 1950-х годах появились языки программирования Fortran, LISP, C и другие, функциональность которых постоянно расширялась вплоть до достижения уровня полноты по Тьюрингу. В XXI веке, вместе с развитием интернета, мобильных технологий и облачных вычислений, вычислительные возможности, достигшие уровня полноты по Тьюрингу, получили широкое распространение, а исследования в области квантовых компьютеров стремятся к новым горизонтам. Системы с полнотой по Тьюрингу теоретически способны решать все возможные вычислительные задачи, однако при использовании платформ с низким уровнем кодирования и без кодирования вопрос соблюдения баланса между эффективностью и универсальностью по-прежнему остается нерешенной проблемой, требующей скорейшего решения.

2.3.2. Ранние технологические исследования

В период с 1980-х годов до начала XXI века появились **языки четвертого поколения** (Fourth Generation Language, 4GL). В отличие от традиционных языков программирования, языки 4GL предлагают декларативный стиль программирования, сосредоточенный на задаче «что делать», позволяя разработчикам писать программы на естественном языке и сокращая объем низкоуровневого кода. Например, использование упрощенных SQL-запросов вместо сложных низкоуровневых вызовов при работе с базами данных заложило теоретическую основу для разработки с использованием low-code технологий.

Дополнительная информация

Языки четвертого поколения – это языки программирования для электронных вычислительных машин, занимающие место над языками третьего поколения в классификации поколений языков программирования. Например, Clipper, SQL, SAS и MATLAB являются языками четвертого поколения.

В тот же период постепенно стали появляться языки и инструменты визуального программирования. Ранние языки программирования позволяли разработчикам создавать простые приложения с помощью визуального интерфейса, например путем перетаскивания компонентов и настройки свойств. Хотя функциональность таких языков и инструментов была относительно ограниченной, уже тогда они продемонстрировали возможность снижения по-

рога входа в разработку с помощью графических операций. Это можно считать прообразом концепции разработки с низким уровнем кодирования.

Так, например, Visual Basic (см. рис. 2-12) является классическим языком визуального программирования, который позволяет разработчикам перемещать на форму элементы управления, такие как кнопки и текстовые поля, с помощью интуитивно понятного конструктора **графического интерфейса пользователя** (Graphical User Interface, GUI), а затем быстро создавать приложения для Windows с помощью простого кода на языке Basic, управляемого событиями.

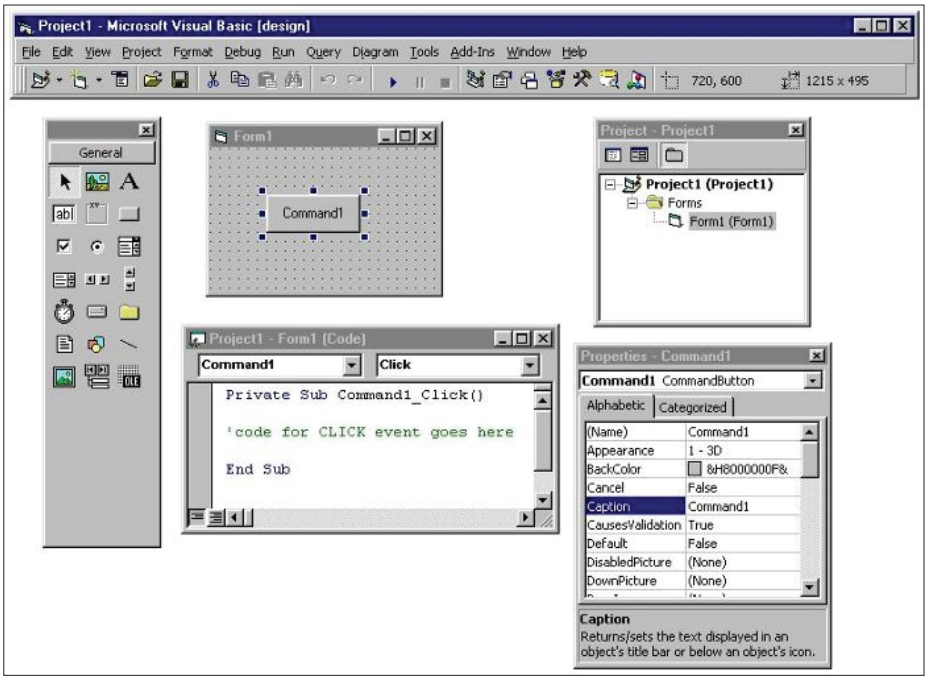


Рисунок 2-12. Редактор Visual Basic

Программа Dreamweaver (см. рис. 2-13) является типичным инструментом в области веб-разработки, поддерживающим режим редактирования как в режиме «дизайна», так и в режиме «кода». Разработчики могут размещать элементы веб-страницы и настраивать стили в визуальном интерфейсе, а также писать код HTML, CSS и JavaScript в режиме просмотра кода. Функции панели действий дают пользователям возможность реализовывать такие интерактивные эффекты, как смена изображений и проверка форм, с помощью простых настроек параметров. Пользователям не нужны глубокие знания сложной логики кода, что значительно повышает эффективность разработки веб-страниц.

Хотя в то время возможности визуальных редакторов были ограниченными, а их применение в основном сводилось к небольшим и средним проектам или специфическим областям, визуальные операции снизили порог входа в разработку, положив начало концепции разработки с минимальным кодом.



Рисунок 2-13. Рабочая область Dreamweaver

2.3.3. Формирование концепции и первые практические шаги

В 2012 году компания Gartner выступила с инициативой «разработка для всех», направленной на устранение барьеров в программировании, чтобы неспециалисты также могли участвовать в процессе создания приложений. В том же году компания OutSystems представила свою платформу для разработки с минимальным использованием кода, которая позволяла быстро создавать приложения благодаря визуальному интерфейсу и готовым компонентам. Она послужила практическим примером реализации концепции low-code.

В 2014 году исследовательская и консалтинговая компания Forrester официально сформулировала концепцию программирования с минимальным кодом: быстрая разработка, настройка и развертывание приложений с использованием минимального количества кода или без его написания. Программирование с минимальным кодом получило признание в качестве самостоятельной технологической области.

Однако с точки зрения теории полноты по Тьюрингу, при использовании в сложных алгоритмах и сценариях с высокой степенью параллелизма программирование low-code ограничено готовыми компонентами и визуальным интерфейсом, что не позволяет достичь той же эффективности, что и с помощью полноценных языков программирования. С позиции неполноты по Тьюрингу, готовые компоненты также не способны удовлетворить потребности в глубокой реализации бизнес-логики при наличии требований высокой степени кастомизации.

Таким образом, несмотря на беспрецедентную популярность сферы low-code на данном этапе, многие по-прежнему скептически относятся к этой концепции и ставят под сомнение ее эффективность.

2.3.4. Этапы развития рынка

По мере повышения требований предприятий к цифровым приложениям функции отдельных low-code платформ оказались недостаточными для решения таких задач, как системная интеграция, обмен данными и кросс-платформенное развертывание. В 2018 году компания Gartner предложила кон-

цепции «платформа приложений как услуга» (Application Platform as a Service, aPaaS) и «платформа интеграции как услуга» (Integration Platform as a Service, iPaaS), что придало новый импульс развитию концепции low-code.

aPaaS: универсальная визуальная среда разработки на основе облачных вычислений, предоставляющая богатый набор компонентов, шаблонов, а также возможности автоматизации развертывания и эксплуатации. Разработчики могут сосредоточиться на бизнес-моделировании. Например, платформа, выпущенная компанией OutSystems, поддерживает функцию перетаскивания компонентов страницы, настройку бизнес-логики, а также позволяет одним нажатием завершить настройку сервера и распределение ресурсов, что сокращает цикл разработки веб-приложений и мобильных приложений.

iPaaS: фокусируется на интеграции данных и процессов между гетерогенными системами. С помощью коннекторов, технологий обработки данных, **механизма извлечения–преобразования–загрузки** (Extract, Transform, Load, ETL) и оркестрации процессов обеспечивает обмен данными и бизнес-интеграцию между **системами планирования ресурсов предприятия** (Enterprise Resource Planning, ERP), **управления взаимоотношениями с клиентами** (Customer Relationship Management, CRM), **управления цепочкой поставок** (Supply Chain Management, SCM) и другими. В качестве примера можно привести компанию Mendix, чье решение iPaaS обеспечивает беспрепятственную интеграцию с точками доступа к сервисам (SAP) и Salesforce, реализуя сквозную оптимизацию бизнес-процессов.

В Китае появились такие платформы с низким уровнем кодирования, как JianDao Cloud и MingDao Cloud. Они соответствуют базовым потребностям в области цифровизации, таким как автоматизация делопроизводства и управление данными, и способствуют внедрению технологий с низким уровнем кодирования на внутреннем рынке. Финансовый рынок также проявляет большой интерес к платформам с низким уровнем кодирования: в 2018 году Siemens приобрела Mendix, а в том же году KKR и Goldman Sachs инвестировали в OutSystems, что обеспечило значительный приток капитала в развитие отрасли.

Однако на этой стадии специалисты и обычные пользователи сталкиваются с такими проблемами, как несоответствие функциональности платформы сложным бизнес-требованиям, технологическая привязка и зависимость от конкретных поставщиков. Несмотря на снижение порога входа в разработку в результате внедрения технологии low-code, пользователям по-прежнему необходимо обладать определенными базовыми знаниями в области программирования и практическим опытом разработки, чтобы с помощью инструментов и ресурсов платформы создавать подходящие для бизнеса специализированные приложения. Технически неопытные пользователи в процессе интеграции сложных систем и расширения функциональности с помощью этих платформ по-прежнему сталкиваются с многочисленными проблемами.

2.3.5. Ограничения традиционных платформ с минимальным кодом и без кода

Несмотря на преимущества традиционных платформ с минимальным кодом и без кода, которые вызвали революцию в моделях разработки благодаря сни-

жению барьеров для разработки и повышению эффективности внедрения, при более глубоком анализе можно обнаружить, что в практическом применении они имеют множество ограничений.

- **Глубина функциональности и обработка сложных задач**. Традиционные low-code и no-code платформы теоретически обладают определенными вычислительными возможностями. Однако из-за зависимости от библиотеки предустановленных компонентов и визуализированных операций они не могут в полной мере реализовать потенциал полноты по Тьюрингу при обработке сложной бизнес-логики, такой как финансовые модели с комплексными алгоритмами или данные в реальном времени в системах электронной коммерции с высокой степенью параллелизма. В результате они не способны удовлетворить потребности предприятий в цифровизации сложных рабочих процессов. Разработчикам зачастую приходится выходить за рамки платформы и возвращаться к традиционному программированию. В то же время традиционные платформы с минимальным кодом и без кода изначально не создавались с целью достижения полноты по Тьюрингу, а потому их возможности заведомо ограничены при реализации бизнес-сценариев со сложными циклами, рекурсией и другими нестандартными управляющими конструкциями. Они не способны предложить эффективные решения для таких задач, как реализация сложных алгоритмов в промышленной автоматизации или научные вычисления.
- **Персонализация и расширяемость**. В погоне за скоростью разработки традиционные платформы с минимальным кодом и без кода в определенной степени жертвуют гибкостью настройки. Для разработчиков со сложными требованиями к настройке ограничения платформы затрудняют реализацию эффективной и гибкой настройки. В то же время рядовые пользователи с ограниченными компетенциями не могут преобразовать сложные бизнес-требования в эффективную логику приложения, даже если платформа предоставляет определенные возможности настройки. Это приводит к появлению в создаваемых приложениях таких проблем, как отсутствие функций и нарушение логики процессов. Кроме того, с развитием бизнеса предприятия возрастает сложность интеграции платформы с новыми технологиями и системами, что еще больше усугубляет проблему масштабируемости.
- **Техническая привязка и зависимость от поставщика**. После внедрения приложений, разработанных на традиционных платформах с минимальным кодом или без кода, предприятия оказываются в глубокой зависимости от соответствующей архитектуры, инструментов разработки и экосистемы. Различия в форматах данных и стандартах разработки между различными платформами затрудняют перенос приложений. Как разработчики с полным набором навыков программирования, так и бизнес-специалисты, полагающиеся на визуальные инструменты, рискуют столкнуться с проблемой неработоспособности приложений или сложностью их обновления в случае изменения стратегии поставщика или ухудшения его финансового положения, поскольку не имеют ни возможностей, ни технической базы для самостоятельного решения таких проблем.

- **Производительность и безопасность**. В коде, генерируемом традиционными платформами с минимальным кодом и без кода, присутствует избыточность, что затрудняет проведение тонкой оптимизации производительности, как в случае с профессиональным кодом, написанным вручную. При обработке больших объемов данных и большого количества одновременных запросов возможно появление таких проблем, как большое время отклика и зависание системы. С точки зрения теории Тьюринга, традиционные low-code и no-code платформы не могут полноценно использовать вычислительные мощности и обеспечить высокую производительность; с позиции неполноты по Тьюрингу, обычные пользователи из-за недостатка знаний и понимания проблем безопасности могут создавать уязвимости, а при появлении таких уязвимостей в платформе предприятия вынуждены полагаться на исправления со стороны поставщика, что усиливает риски для данных и бизнеса.

2.3.6. Использование ИИ в платформах с низким и нулевым уровнями кодирования

ИИ обеспечивает платформы с низким и нулевым уровнями кодирования беспрецедентной эффективностью.

- **Расширение функциональных возможностей**. Интеллектуальная генерация и оптимизация кода с помощью ИИ обеспечивают платформы с низким и нулевым кодированием возможностями для более сложных бизнес-сценариев. Например, в сфере финансового риск-менеджмента ИИ может создавать и постоянно оптимизировать модели оценки рисков на основе исторических данных и бизнес-правил, постепенно приближаясь к эффективной реализации системы с полнотой по Тьюрингу.
- **Автоматическое взаимодействие при генерации кода**. В таких областях, как управление устройствами интернета вещей, технологии обработки естественного языка могут генерировать базовый код для сбора и анализа данных на основе протоколов устройств и описаний требований, восполняя недостатки платформы при реализации сложных функций.
- **Персонализированная настройка и расширение**. Опираясь на глубокий семантический анализ и машинное обучение, ИИ может преобразовывать требования, сформулированные на естественном языке, в настраиваемые компоненты и процессы в соответствии с бизнес-логикой, что значительно снижает порог вхождения для обычных пользователей.

В этом контексте вайб-программирование с его уникальной ИИ-парадигмой дополнительно расширяет описанные выше преимущества: от глубокого понимания намерений до интеллектуальной генерации кода, создавая замкнутый цикл «заявление намерения – генерация системы – взаимодействие человека и машины». Таким образом значительно повышаются эффективность и качество разработки, что позволяет по-настоящему преодолеть ограничения традиционных платформ с низким и нулевым уровнями кодирования, предлагая более эффективные и интеллектуальные решения для разработки ПО.

2.4. Заключение

Оглядываясь на историю развития методов программирования, от абстрактной эволюции машинных языков к языкам высокого уровня, от инноваций в области взаимодействия, таких как текстовые редакторы и интегрированные среды разработки, до оптимального сочетания эффективности и доступности платформ с низким и нулевым уровнями кодирования, можно говорить о постоянном поиске баланса между эффективностью, доступностью и информативностью. Каждый технологический скачок – это не только раскрытие вычислительных возможностей, но и переосмысление способов мышления разработчиков.

Эволюция языков программирования привела к появлению более выразительных средств, развитие инструментов улучшило взаимодействие при разработке, а появление платформ с низким уровнем кодирования еще больше размыло границы между «программированием» и просто «использованием». Все эти наработки в совокупности сформировали основу, на которой сегодня реализуется программирование с использованием ИИ. Технологии уже созрели, и настоящий вызов заключается в воплощении возможностей ИИ в реальных сценариях разработки для создания удобных, надежных и безопасных продуктов.

Глава 3

Экосистема приложений вайб-программирования

https://t.me/it_boooks/2

На пути от создания технологии к ее практическому применению необходимо преодолеть один важный этап – коммерциализацию. Так же, как открытие электричества не привело к немедленному переходу к эпохе электротехники, зрелость ИИ-технологий не означает автоматического наступления эпохи ИИ-программирования. Чтобы эти передовые технологии действительно стали полезными инструментами в повседневной практике программирования, необходимы подходящие производственные формы, удобные способы взаимодействия и развитая экосистема рабочих приложений.

В этой главе мы сосредоточимся на конкретных формах применения вайб-программирования в различных сценариях разработки: от ранних этапов программирования с помощью универсальных больших языковых моделей до интеграции в IDE и сквозного программирования с помощью агентов. Мы систематизируем эволюцию технологий и тенденции внедрения вайб-программирования в реальной цепочке разработки.

3.1. Программирование с универсальными большими языковыми моделями

Благодаря широкому распространению универсальных больших языковых моделей одной из первых форм взаимодействия с вайб-программированием стали диалоги. Разработчики могут общаться с большими языковыми моделями на естественном языке и без использования сложных инструментов напрямую получать рекомендации по программированию, фрагменты кода и даже готовые решения. Такое веб-ориентированное «диалоговое программирование» пока что остается базовым вспомогательным инструментом, но его вклад в повышение эффективности и улучшение опыта разработки несомненно огромен.

3.1.1. Вопросы и ответы с использованием больших языковых моделей

Самые ранние формы диалогового вспомогательного программирования в основном опирались на веб-окна чата с большими языковыми моделями, типичными представителями которых являются ChatGPT, Claude и Gemini. Эти про-

дукты отличаются лаконичным интерфейсом и поддержкой ввода на естественном языке. Пользователям не нужно осваивать сложные команды – достаточно задать технический вопрос обычным разговорным языком, и большая языковая модель в режиме реального времени сгенерирует соответствующий ответ, рекомендацию или даже непосредственно предоставит код для решения этой задачи.

Основное преимущество такого подхода заключается в том, что пользователь может пробовать и экспериментировать без каких-либо начальных навыков. Будь то новичок, который не может найти ошибку, опытный разработчик, которому нужно быстро разобраться в незнакомом API, или просто пользователь, которому нужно сгенерировать шаблон простого алгоритма с помощью ИИ, – благодаря прямому диалогу с большими языковыми моделями они могут получить необходимую информацию буквально за несколько секунд.

По сути, на ранних этапах развития многие не придавали значения генеративному ИИ, полагая, что способности хорошего разработчика заключаются исключительно в технических навыках. Однако на практике умение искать информацию является не менее важным фактором, определяющим эффективность разработки и качество решения задач.

Проще говоря, умение искать информацию означает способность быстро и точно находить нужные сведения для решения проблемы с помощью таких поисковых систем, как Google или Baidu, и затем эффективно решать эту проблему. Не стоит недооценивать этот навык. Хотя пользоваться поисковыми системами умеет каждый, немногие действительно способны использовать их для оперативного решения проблем. Зачастую вы обнаруживаете, что после долгих поисков так и не нашли нужного ответа, в то время как опытные специалисты, к которым вы обращаетесь за помощью, могут дать решение в считанные секунды. Помимо накопленного опыта, это в значительной степени обусловлено именно их умением находить и отбирать нужную информацию.

Настоящие мастера решения проблем, как правило, обладают двумя качествами. Во-первых, они способны быстро определить ключевые слова в сложных задачах. Во-вторых, они имеют богатый опыт поиска и способны с минимальными затратами отфильтровать наиболее полезные ответы из множества результатов. Умение задавать вопросы поисковым системам – это целое искусство, и с появлением генеративных ИИ этот навык претерпевает значительные изменения. Можно сказать, что большие языковые модели заменяют традиционные поисковые системы, становясь новым каналом получения знаний и решения проблем. Что касается конкретных навыков, то раньше нам нужно было выделять ключевые слова для запроса, а теперь можно использовать более естественный язык и формулировать вопросы высокого качества, чтобы ИИ понял наши намерения и ответил максимально близко к запросу. ИИ в определенной степени ослабил нашу зависимость от поисковых запросов по ключевым словам, но при этом превратил навык четкого и точного изложения вопроса в новую ключевую способность для решения проблем.

На самом деле мы наблюдаем постепенную эволюцию способов получения информации человеком: от запросов к поисковым системам до обращений к ИИ. С каждым этапом этой эволюции на первый план выходит способность формулировать вопросы и извлекать нужную информацию. Только в эпоху ИИ

этот навык получил более формальное техническое название – умение составлять промпты, то есть формулировать подсказки для ИИ.

С распространением ИИ-приложений умение эффективно и точно задавать вопросы ИИ, а также умение с помощью промптов добиваться от больших языковых моделей желаемых ответов становится обязательным навыком для разработчиков новой эпохи. Диалоговый режим программирования – это конкретное воплощение сочетания поисковых навыков и промптового мышления при поддержке больших языковых моделей.

3.1.2. Большие языковые модели и их потенциал в программировании

По мере развития технологических возможностей больших языковых моделей они все чаще демонстрируют выдающиеся результаты в сфере программирования. Различные производители конкурируют между собой на основе таких параметров, как генерация кода, техническое мышление и понимание контекста по нескольким диалогам.

На примере серии GPT, выпущенной OpenAI, можно отметить, что она не только способна генерировать код на распространенных языках программирования, таких как Python и JavaScript, но также обладает сильными логическими возможностями, может понимать сложные бизнес-требования и на их основе генерировать относительно полную структуру функций или проект решения. В частности, в ходе нескольких раундов диалога серия GPT постепенно начинает демонстрировать хорошую техническую связность и способность к постоянной оптимизации качества кода на основе контекста.

Выпущенная компанией Anthropic серия Claude уделяет больше внимания безопасности и стабильности диалога, особенно в плане соблюдения непрерывности длинных контекстов. В условиях корпоративной разработки, где действуют строгие стандарты, серия Claude более эффективно соблюдает заданные разработчиком правила и ограничения, сокращая количество ошибок в коде. Она считается лучшей в отрасли программируемой моделью и стабильно занимает первое место в различных рейтингах по программированию (таких как SWE-bench и Codeforces).

Серия Gemini от Google открыла новую парадигму программирования с использованием нескольких моделей. Благодаря технологии кросс-модальности, объединяющей текст, изображения, аудио и другую информацию, она еще больше расширила возможности больших языковых моделей в таких областях, как мультимедийная разработка, визуализация данных и пользовательский интерфейс. В отличие от обычного текстового ввода, серия Gemini предлагает новые подходы к решению более сложных и многомерных задач разработки.

Китайские большие языковые модели, такие как Wenxin Yiyan от Baidu, Tongyi Qianwen от Alibaba и DeepSeek от Huanfang Quantification, также постоянно совершенствуют свои возможности в области генерации кода, обладая уникальными преимуществами, особенно в локализованных сценариях применения и поддержке китайского языка. По мере технологического развития разрыв в способностях различных больших языковых моделей в таких областях, как анализ кода, исправление ошибок и оптимизация производительности, сокращается, а уникальные функции и интеграция экосистем становятся новыми факторами конкуренции.

3.1.3. Преимущества и недостатки диалогового режима программирования

Нельзя отрицать, что диалоговый режим программирования обладает значительными преимуществами в плане снижения барьеров для разработчиков и повышения эффективности технических экспериментов. Однако ограничения этого подхода также несомненны.

- В ранних версиях диалоговых больших языковых моделей наблюдалась явная разрозненность контекста: отсутствовало непрерывное восприятие полного контекста проекта и истории разработки, что приводило к трудностям с непосредственным внедрением сгенерированного кода в реальные проекты и зачастую требовало ручной корректировки или проверки.
- Простое веб-интерфейсное взаимодействие оторвано от основной среды разработки, и программистам приходится переключаться между разными окнами, что сказывается на целостности рабочего процесса.
- Диалоговый режим программирования не имеет механизмов структурированного взаимодействия, что делает его пригодным в основном для индивидуальных исследований и не отвечает потребностям непрерывной разработки на уровне команды или системы.

Таким образом, несмотря на то что диалоговый режим программирования стал отправной точкой для вайб-программирования, настоящее раскрытие потенциала ИИ в разработке требует эволюции к более глубокой, системной и интегрированной форме. Именно это стало причиной появления режима программирования с интеграцией в IDE.

3.2. Программирование с помощью IDE

По сравнению с программированием с помощью универсальных диалоговых моделей, полностью зависящих от веб-интерфейса, вайб-программирование глубже и эффективнее интегрируется в рабочий процесс разработчика за счет системной интеграции возможностей ИИ в IDE – основную среду разработки. Такая интеграция преодолевает фрагментацию взаимодействия разработчиков, когда им приходится выходить из IDE, чтобы задать вопрос ИИ, а затем возвращаться в IDE для продолжения разработки. Вместо этого ИИ-возможности напрямую интегрируются в рабочий процесс, объединяя программирование и интеллектуальную поддержку в единое целое. В результате формируется более целостная и продуктивная форма применения вайб-программирования.

Ранее IDE по сути разрабатывались для людей, и их функции были сосредоточены на традиционных задачах разработки, таких как редактирование кода, проверка синтаксиса и отладка. Однако на волне развития ИИ интеграция переходит от плагинов и патчей к глубокой интеграции искусственного интеллекта на уровне ядра, постепенно превращаясь в основную площадку для совместной разработки с участием ИИ.

3.2.1. Интеграция ИИ в плагины IDE

Одним из первых инструментов, представляющих интеграцию ИИ в плагины IDE, является Tabnine. Используя технологии машинного обучения, он обеспе-

чивает интеллектуальное автодополнение для распространенных синтаксических структур и вызовов функций, помогая разработчикам сэкономить время на повторяющихся операциях ввода. Хотя функционал Tabnine относительно ограничен, а его эффективность в сложных сценариях оставляет желать лучшего, он стал первым шагом в попытке объединить ИИ и IDE.

Именно выпуск GitHub Copilot в 2021 году стал тем фактором, который ускорил внедрение концепции программирования с помощью ИИ. Являясь одним из самых востребованных ИИ-плагинов для IDE на сегодняшний день, GitHub Copilot опирается на большую языковую модель OpenAI, поддерживает множество популярных языков программирования и способен генерировать код в режиме реального времени на основе комментариев на естественном языке. Кроме того, он также анализирует намерения разработчика, самостоятельно дополняет структуру функций и даже предлагает несколько вариантов решения. Такой переход от обработки фрагментов кода к пониманию сути задачи впервые позволил ИИ в IDE превратиться из простого набора инструментов в настоящего партнера разработчика, что значительно повысило эффективность и удобство работы.

В последние годы появилось множество более комплексных ИИ-помощников для разработчиков с акцентом на взаимодействие. Их функционал является весьма мощным и в некоторых аспектах не уступает IDE со встроенным ИИ. Однако поскольку они также являются «внешними» надстройками для традиционных IDE, их пользовательский интерфейс и некоторые функциональные детали уступают по своим характеристикам IDE со встроенным ИИ.

Наиболее ярким примером является Cline, представленный в 2024 году. Этот интеллектуальный помощник разработчика глубоко интегрирован с редактором VS Code и отличается высокой частотой обновлений. В отличие от традиционных плагинов для генерации кода, Cline предоставляет не только интерфейс для взаимодействия IDE с большими языковыми моделями, но также поддерживает **протокол контекста модели** (Model Context Protocol, MCP), который устраняет барьеры между естественным языком в IDE и внешними инструментами. Благодаря этому Cline может анализировать структуру и бизнес-логику больших кодовых хранилищ, поэтапно планировать и генерировать решения, одновременно отслеживая изменения в IDE в режиме реального времени и проверяя, что каждое изменение получило окончательное одобрение разработчика. Благодаря такому коллаборативному подходу к генерации кода повышается эффективность работы больших команд, а также значительно упрощается вхождение новых членов в сложные проекты. Кроме того, благодаря интеграции с рынком MCP Cline предлагает богатый выбор сторонних MCP-серверов (серверной части протокола контекста модели), и управление всеми командами осуществляется с помощью естественного языка, что еще больше сокращает время разработки. В Cline также поддерживается интеграция с API-интерфейсами поставщиков больших языковых моделей, что дает пользователям возможность использовать сторонние большие языковые модели без значительных финансовых издержек. На данный момент Cline является единственным плагином для IDE с интегрированным ИИ, который по функциональности может сравниться с Cursor, IDE с нативной интеграцией ИИ.

В 2025 году был выпущен Augment Code, который похож на Cline, но больше ориентирован на оптимизацию процессов разработки на уровне команды и обработку длинных контекстных окон. Благодаря глубокому анализу кодовой базы, системы зависимостей и лучших практик команды Augment Code помогает разработчикам быстро понять структуру системы в привычной среде IDE. Многие пользователи Augment Code характеризуют его как очень интеллектуальный инструмент: по их мнению, он превосходит Cline и даже Cursor в плане понимания контекста и организации рабочих процессов. Однако ежемесячная плата в размере $50 отпугивает многих потенциальных пользователей.

Таким образом, несмотря на различия в технической реализации интеграции ИИ в плагины IDE, в целом наблюдаются две общие тенденции: во-первых, постепенный отход от позиционирования как «точечного инструмента» и начало системного участия в работе IDE; во-вторых, расширение функционала от простого автодополнения кода до оптимизации совместной работы, передачи знаний и понимания сложных систем.

3.2.2. IDE с нативной интеграцией ИИ

Если ИИ-плагины представляют собой «внешнюю» интеграцию в традиционные IDE, то IDE с нативной интеграцией ИИ представляют собой среду разработки, переосмысленную в соответствии с концепцией вайб-программирования. Такие IDE больше не являются простым объединением функций, а представляют собой продукты полного цикла, в которых основой является ИИ – от системной архитектуры, потоков данных и управления контекстом до моделей взаимодействия человека и машины.

Среди продуктов этой категории Cursor, несомненно, является самым знаковым. Первая версия Cursor, выпущенная в 2023 году, была разработана на основе открытого редактора VS Code, однако по своим возможностям она далеко выходит за рамки традиционной модернизации. Cursor – это первая в полном смысле слова интегрированная IDE со встроенным ИИ, чья техническая концепция и принципы работы принципиально отличаются от более ранних интегрированных плагинов. Суть инновации Cursor заключается в том, что в качестве основы для ИИ используется весь кодовый репозиторий. В отличие от ранней эпохи плагинов, основанной на поверхностном понимании по принципу «текущего окна» или «локального контекста», Cursor после запуска автоматически индексирует, анализирует и векторизует весь код внутри проекта, создавая полную семантическую карту. На каждый вопрос, команду или запрос на автодополнение разработчика искусственный интеллект отвечает в контексте всего проекта. Такая глубокая интеграция обеспечивает совершенно иной уровень взаимодействия по сравнению с традиционными IDE.

Это улучшение взаимодействия не ограничивается техническими деталями, оно также значительно снижает порог входа в сферу программирования. Например, восьмилетний ребенок вице-президента Cloudflare, используя только Cursor в сочетании с ИИ, самостоятельно создал приложение-чат-бот всего за 45 минут. Хотя это единичный случай, он наглядно демонстрирует потенциал вайб-программирования в плане преодоления профессиональных барьеров и переосмысления методов разработки.

В Cursor базовыми функциями уже являются автодополнение кода в реальном времени, контекстное понимание, интеллектуальный рефакторинг, предварительное определение позиции курсора и совместная работа с несколькими файлами. В 2025 году была выпущена версия Cursor 1.0, а на момент завершения работы над этой книгой последней версией являлась Cursor 1.2. Частые итерации позволили Cursor обзавестись множеством мощных функций, таких как интеграция правил (Rules) на уровне проекта, поддержка клиента протокола контекста модели (MCP Client), функция запоминания (Memories), помощник по автоматической проверке ошибок (BugBot) и фоновый агент (Background Agent).

Появление Cursor было подобно падению камня в спокойную воду: оно быстро вызвало цепную реакцию и непосредственно способствовало росту внимания и исследований отрасли в области IDE с нативной интеграцией ИИ.

Выпущенный в 2024 году Windsurf можно рассматривать как технологическое продолжение и углубление концепции Cursor того времени. Проект был создан командой Codeium. Помимо интеграции многих преимуществ Cursor, таких как поддержка диалоговых сессий, глобальное понимание кода и автодополнение в реальном времени, Windsurf предпринял попытки прорыва по нескольким направлениям, в частности обеспечив более системную и тщательную оптимизацию взаимодействия в рамках парадигмы AI Flow (рабочий процесс с ИИ).

Наиболее примечательной особенностью Windsurf является поддержка многоэтапной совместной работы с использованием нескольких инструментов, что позволяет преодолеть ограничения традиционных генеративных ИИ с принципом работы «один запрос – один ответ». Благодаря встроенному модулю Cascade разработка задач в Windsurf разбивается на четкие режимы Write и Chat. Режим Chat предназначен для обычных диалогов и решения вопросов, а режим Write поддерживает возможность непосредственного создания файлов, их пакетного изменения и редактирования нескольких файлов одновременно. После подтверждения разработчиком изменения автоматически внедряются на уровне кода. Такой подход повышает эффективность и позволяет разработчику полностью контролировать работу над проектом. В Windsurf также реализовано динамическое определение среды разработки, позволяющее автоматически отслеживать каждое изменение в проекте. Будь то добавление файла, изменение переменной или рефакторинг метода, Windsurf мгновенно обновляет контекстную информацию для гарантированного информирования ИИ об актуальном состоянии проекта при последующих взаимодействиях.

3.2.3. Сравнение ИИ-интеграции в IDE

Благодаря установке плагинов Copilot, Cline или Augment Code в IDE, таких как VS Code или JetBrains, разработчики получают мощные возможности ИИ-поддержки без необходимости переключаться между рабочими средами. Такой подход совместим с действующей средой разработки пользователя: достаточно установить плагин в существующий редактор, что очень удобно для быстрого внедрения в команде. Но поскольку плагины работают в IDE, восприятие ИИ общей структуры проекта зависит от ручного ввода контекста, локального кеша или интеллектуального поиска плагина. Из-за этого ИИ не может постоянно и систематически воспринимать весь проект целиком, как это делает ИИ, встроенный в IDE изначально. Интеграция плагинов также

ограничена базовой архитектурой разных IDE, что препятствует свободному развертыванию некоторых низкоуровневых функций.

Такие IDE с нативной интеграцией ИИ, как Cursor и Windsurf, безусловно, обеспечивают лучшую степень осмысления проекта, более качественный пользовательский опыт и более широкие возможности расширения. Безусловно, использование глобального индекса для анализа проектного кода не является абсолютно идеальным решением. Однако по сравнению с динамическими механизмами поиска, используемыми в таких продуктах, как Cline, пользователи получают возможность осуществлять поиск и просмотр проектных файлов с более интеллектуальным и контекстно-зависимым откликом. Интеграция на нативном уровне обеспечивает более плавную конвергенцию ИИ и IDE, а также более высокую согласованность детализированного взаимодействия; самое главное, что низкоуровневая интеграция не ограничивается архитектурой IDE и обеспечивает больше возможностей для расширения, что можно увидеть на примере различных итераций настраиваемых функций Cursor. Тем не менее поскольку Cursor и Windsurf разработаны на основе ответвления VS Code, некоторым пользователям других IDE может потребоваться время для адаптации к новому способу взаимодействия. Более того, для использования мощных функций Cursor без отказа от привычной IDE некоторые пользователи вынуждены работать в режиме «двойного запуска», что заведомо не является удобным. Этот недостаток присущ всем продуктам с нативной интеграцией ИИ, таким как Cursor и Windsurf.

Следует отметить, что плагины для IDE с интегрированным ИИ, такие как Cline и Augment Code, в определенной степени заимствуют концепцию дизайна Cursor. Ранняя версия Windsurf также в некоторой степени опиралась на концепцию Cursor, но на ее основе проложила свой уникальный путь, а затем в Cursor, в свою очередь, вошли некоторые передовые решения Windsurf.

3.3. Сквозное программирование с помощью агентов

По сравнению с IDE-интеграцией для поддержки программирования, продукты для сквозной (end-to-end) разработки с помощью агентов представляют собой более совершенную форму вайб-программирования. В отличие от автозаполнения кода или ответов на вопросы, они позволяют ИИ действительно самостоятельно выполнять весь цикл разработки.

3.3.1. Концепция сквозного программирования с помощью агентов

Под «сквозным» программированием в данном случае подразумевается способность ИИ самостоятельно выполнять все ключевые этапы процесса разработки ПО в соответствии с конкретными заданными требованиями. Сюда входит не только генерация кода, но также реализация функций, исправление ошибок, добавление тестовых примеров, синхронное обновление документации и даже автоматическое развертывание приложений и запуск в производственной среде.

В отличие от традиционных интегрированных в IDE функций поддержки ИИ, сквозной агент более ориентирован на автономное взаимодействие на системном уровне. Как в случае с плагинами ИИ для IDE либо встроенными в IDE

функциями ИИ, их возможности зачастую сводятся к конкретным операциям, таким как интеллектуальное автодополнение, семантический поиск или локальный рефакторинг. Несмотря на их удобство, разработчику по-прежнему приходится контролировать всю работу над проектом в целом. В то же время сквозной (end-to-end) агент обладает более мощными возможностями общего восприятия и организации задач, способен самостоятельно анализировать и разбивать задачи в соответствии с потребностями, а также координировать их выполнение.

В связи с необходимостью сочетать мощные вычислительные ресурсы для работы с большими языковыми моделями и хранения сложных контекстов, а также интеграции с различными инструментальными цепочками большинство продуктов для программирования с использованием сквозных агентов в настоящее время развернуты в облаке и опираются на мощные облачные вычислительные ресурсы, при этом полная локализация этих продуктов пока не реализована.

3.3.2. Решения для сквозного программирования агентов

В сфере сквозного программирования агентов решение Devin, несомненно, является одним из первых в своем роде, позиционирующим себя как инструмент «ИИ-инженера», и поэтому считается отраслевым эталоном, хотя между его практическими возможностями и ожиданиями отрасли по-прежнему существует ощутимый разрыв.

В момент первоначального релиза Devin разработчики ставили перед собой очень амбициозные цели, делая ставку на «полностью автоматизированную сквозную разработку», то есть на то, чтобы ИИ мог самостоятельно анализировать требования, разбивать задачи на этапы, генерировать код, тестировать его, исправлять ошибки и даже выпускать готовый продукт. Высокая стоимость подписки в размере $500 ежемесячно, а также несовершенство самой модели отпугнули многих пользователей, а первые тестировщики в основном отмечали недостаточный уровень интеллекта Devin и даже его «неуклюжесть».

Однако по мере развития отраслевых технологий разработчики Devin оперативно скорректировали стратегию развития продукта. С одной стороны, цена подписки снизилась до $20 в месяц, и была введена модель оплаты по факту использования. С другой стороны, на уровне продукта был сделан акцент на прагматичном позиционировании «Treat Devin like a junior engineer» (относитесь к Devin как к инженеру-новичку): нужно использовать Devin как начинающего разработчика, разумно ожидая от него результатов при выполнении простых и структурированных задач, но при этом сохраняя за человеком-разработчиком право принятия сложных решений и контроля на этапе тестирования.

На самом деле нынешние возможности Devin характеризуются типичным сочетанием высокой степени исполнения и низкой интеллектуальной глубины. То есть, несмотря на средний уровень логического мышления модели, общий рабочий процесс продукта отлажен достаточно хорошо, особенно когда пользователь предоставляет адекватные подсказки и четко разбивает задачу на этапы. В таком случае Devin вполне способен выполнять разработку базовых функций, исправление простых ошибок и даже относительно сложные многоэтапные рабочие процессы.

Однако у Devin есть еще один недостаток, который нельзя игнорировать. Речь о проблеме чрезвычайно высокой стоимости использования **агентов** (Agent). Например, для обработки одной задачи (GitHub Issue) Devin может потребовать около 3 вычислительных единиц агента (Agent Compute Unit, ACU), при этом стоимость одной ACU составляет примерно $2,25. В случае более сложных задач, когда модель не распознает замысел и предпринимает многократные попытки, расход ACU может быстро увеличиться, что приведет к резкому росту издержек. Поэтому фактическая стратегия многих пользователей заключается в использовании Devin для быстрого создания первоначальной версии проекта, а впоследствии они переключаются на такие продукты, как Cursor или Windsurf, которые изначально интегрированы с ИИ, для доработки деталей.

С точки зрения технологического стека, преимущества Devin заключаются не столько в самом моделировании, сколько в системной интеграции и проектировании рабочих процессов. Это решение поддерживает прямую интеграцию с распространенными инструментами для командной работы, такими как Slack, Linear и Jira. Распределение задач, отслеживание статуса и обратная связь по результатам – все это может быть встроено в основную рабочую среду для повышения эффективности контекстной разработки.

Еще одной заслуживающей внимания функцией является представленный Devin **механизм Confidence Rating** (коэффициент уверенности для оценки готовности модели к выполнению текущей задачи и вероятности ее успешного выполнения). Эта функция систематически оценивает степень готовности к выполнению каждой задачи, помогая пользователям избежать неэффективного расходования ресурсов из-за низкого качества результатов. Ибо, помимо снижения эффективности, ошибки агента приводят к непосредственным издержкам за счет расхода вычислительных ресурсов. В некотором смысле такая конструкция с обратной связью отражает серьезные размышления команды Devin о практической реализации продуктов для сквозного программирования агентов.

Помимо первопроходца Devin, в сфере сквозного программирования агентов появилось множество продуктов для узких специализированных областей.

v0 больше от Vercel ориентирован на сценарии пользовательского интерфейса (UI), где пользователь может быстро «нарисовать» прототип интерфейса с помощью простого описания. Благодаря React и системе компонентов shadcn/ui результаты, генерируемые v0, получаются не только красивыми, но также реально пригодными для использования и могут быть бесшовно интегрированы в код проекта. В результате глубокой инженерной оптимизации, повторного использования шаблонов и тонкой настройки моделей команда Vercel создала ведущий в отрасли продукт, который понемногу расширяет сферу применения v0 до полноценной разработки с поддержкой бесшовной интеграции с GitHub. Очевидно, что у команды Vercel большие амбиции.

Bolt, Replit, Lovable и другие продукты, ориентированные на концепцию Idea to App (от идеи до приложения). Эти инструменты призваны упростить процесс разработки, объединить фронтенд и бэкенд с цепочкой развертывания, а также обеспечить предварительный просмотр в реальном времени. Их целью является дать пользователям возможность быстро создавать и запускать прототипы продуктов, руководствуясь естественным языком и собственными идеями, без необходимости углубляться в детали реализации. В частности, Bolt больше ориентирован на

удобство для разработчиков, а Replit делает ставку на универсальную облачную среду. Хотя оба продукта позиционируются как пригодные для начинающих, они все же отображают код и выводят сообщения об ошибках сборки в процессе использования. В основном они ориентированы на профессионалов с определенным техническим опытом, таких как менеджеры продуктов, UI-дизайнеры и разработчики. В свою очередь, Lovable ориентирован на пользователей с низким уровнем технической подготовки и предлагает еще более низкий порог входа.

В области программирования от конечного пользователя до агента YouWare демонстрирует более смелый и радикальный подход. Он полностью скрывает низкоуровневые технические детали: пользователь не видит код и не сталкивается с такими концепциями разработки, как ошибки сборки. Пользователю достаточно просто высказать свои идеи и поэкспериментировать с некоторыми функциями интерактивного взаимодействия, предоставляемыми YouWare. После этого ИИ автоматически сгенерирует готовый к использованию вебсайт, приложение или виджет. Кроме того, YouWare скрывает неудачные попытки, что значительно снижает психологическую нагрузку и позволяет даже «техническим новичкам» не бояться экспериментов. По сути, YouWare – это платформа на основе **концепции пользовательского ПО** (User Generated Software, UGS), которая привлекает к созданию ПО широкий круг обычных пользователей. Для тех, кто не имеет опыта программирования, это настоящая находка. Однако для профессиональных разработчиков YouWare не обеспечивает необходимый уровень управляемости. Обычные пользователи, возможно, не обращают особого внимания на технические детали, но поскольку детали генерации и реализации на платформе YouWare остаются для пользователей «черным ящиком», профессиональные разработчики не могут напрямую вызывать или настраивать базовый технологический стек. Это не позволяет им использовать свои глубокие технические знания и вынуждает программировать с помощью естественного языка, как это делают обычные пользователи.

В эпоху ИИ сосуществование программистов и вайб-кодеров (то есть людей, которые не зависят от технических деталей и создают ПО на основе творческого подхода) становится реальностью, а такие продукты, как YouWare, являются для обычных пользователей важным каналом доступа к системе производственных ресурсов ИИ. Возможно, эта платформа не сможет напрямую составить конкуренцию инструментам профессиональных разработчиков, но она вполне способна перестроить отношения между массовым пользователем, ПО и ИИ, способствуя воплощению в жизнь концепции «программирование для всех».

Смогут ли YouWare и подобные продукты стать инструментами для массового творчества, как платформы для коротких видео? Это вопрос, который еще предстоит исследовать. Однако по мере дальнейшего совершенствования платформы, расширения форматов вывода и постепенной оптимизации структуры затрат YouWare уже сегодня открывает новые возможности для сферы сквозного программирования агентов благодаря повышению привлекательности и доступности.

3.3.3. Механизм работы и системная архитектура

На первый взгляд продукты для сквозного программирования агентов кажутся просто «приложениями для генерации диалогов», но на самом деле за ними стоит целый комплекс высокоинтегрированных и систематизированных тех-

нологий. Ключом к реализации полного цикла взаимодействия – от ввода требований до выпуска продукта – является многоуровневая координация и интеллектуальные механизмы на базовом уровне.

В основе продуктов для сквозного программирования агентов лежит способность больших языковых моделей к логическому анализу. По сравнению с такими плагинами IDE, как GitHub Copilot, в системах «от начала до конца» обычно используются большие языковые модели с более глубокой цепочкой логических выводов и более сильной способностью к работе с контекстом. Эти модели не только понимают отдельные команды, но также способны вести непрерывный диалог, осуществлять цепочку логических выводов и пошаговое планирование. Например, модель в основе Devin, хотя и позиционируется как автономный «инженер-программист с ИИ», на самом деле при решении сложных задач по-прежнему полагается на несколько циклов логических выводов и динамическое планирование больших языковых моделей.

Механизм непрерывной синхронизации контекста является основой функционирования агента в целом. Такие продукты, как Replit, Devin и Bolt, обычно используют мониторинг файловой системы, индексирование в реальном времени и мониторинг состояния для поддержания полного понимания текущей среды разработки, истории операций и внешних систем. Это означает, что агент не только единовременно принимает информацию и генерирует результат, но также отслеживает изменения требований и эволюцию структуры кода в реальном времени, динамически корректируя свое поведение. Например, Devin может автоматически обновлять статус задач в соответствии с изменениями в Jira и Slack, а Replit использует структуру рабочей области для сохранения полного контекста и поддержки множественных циклов разработки и отладки.

На этом фоне способность инструментальной цепочки к планированию является ключевым элементом сквозных решений. Простые выводы больших языковых моделей могут предоставлять лишь текст или фрагменты кода, тогда как сквозные системы зачастую включают в себя полноценные интерфейсы вызова инструментов, охватывающие такие этапы, как написание кода, тестирование, инсталляция компонентов и развертывание среды. Например, Bolt и YouWare имеют встроенные интегрированные процессы развертывания для фронтенда и бэкенда, что дает пользователям возможность воплощать свои идеи в готовые продукты без необходимости переключения между средами.

В настоящее время все больше сквозных платформ пытаются внедрить архитектуру со множеством агентов. На примере Devin можно отметить, что в его структуру введены такие функции, как Knowledge Base (база знаний) и Playbook (механизм эффективного повторного использования). Это позволяет нескольким субагентам выполнять различные роли, такие как разбивка требований, написание кода, исправление ошибок и тестирование, повышая общую эффективность и надежность за счет совместного разделения обязанностей. Такая архитектура фактически имитирует логику взаимодействия человеческой команды разработчиков и отражает тенденцию постепенной эволюции продуктов агентского программирования в сторону «интеллектуальных агентов на уровне команды».

Кроме того, структурированный вывод результатов является еще одним важным аспектом повышения практичности сквозных систем. Платформы Replit, Lovable и другие обычно предоставляют предварительный просмотр

в реальном времени, а также визуальный интерактивный интерфейс. По сравнению с традиционным отображением различий в коде (то есть демонстрацией разницы между текстом кода до и после изменения) и текстовыми ответами, структурированный и графический вывод результатов значительно снижает порог вхождения и повышает степень зрелости продукта. Разумеется, все это по-прежнему опирается на мощную вычислительную базу и инженерную систему. Так, YouWare, упростив технические детали и пожертвовав частью специализированных возможностей управления, обеспечила доступный для широкой аудитории интерфейс с низким порогом входа. Компромиссы в системном проектировании различных продуктов в конечном итоге определяют различия в их позиционировании и целевых группах потребителей.

3.4. Будущее прикладных форм

Вайб-программирование все еще находится в стадии стремительной эволюции: будь то форм-факторы продуктов, сегментация пользователей, технологическая основа или концепции взаимодействия, вся отрасль находится на ранней стадии экспериментов и соперничества. Несмотря на текущую разрозненность, направление дальнейшего развития становится все более четким. По сравнению с первоначальными инструментами, ограниченными функциями интеллектуального автодополнения или генерации кода, современные продукты для программирования с помощью ИИ демонстрируют явное разделение на различные категории.

3.4.1. Новые формы применения

В сфере программирования с помощью ИИ появился ряд характерных продуктов нового типа, которые представляют собой технологические исследования в разных областях.

Claude Code – это первый инструмент для программирования агентов с приоритетом командной строки, выпущенный компанией Anthropic. Он разработан с целью безопасного внедрения ИИ в повседневный процесс инженерной разработки. В отличие от интегрированных в IDE средств помощи при программировании, Claude Code запускается непосредственно в терминале. С помощью простых команд на естественном языке разработчики могут использовать Claude Code для редактирования файлов, исправления ошибок, запуска тестов и контроля версий с помощью Git, а также вызывать инструменты веб-поиска для получения дополнительной информации. Кроме того, Claude Code поддерживает использование встроенной документации проекта (например, файла *CLAUDE.md*) в качестве контекста, что обеспечивает более точное первоначальное понимание проекта со стороны ИИ и более адекватные ответы, соответствующие структуре продукта. Представители Anthropic также подчеркивают «ориентированность на безопасность и конфиденциальность» инструмента: все взаимодействия осуществляются напрямую через локальный терминал, минуя промежуточные серверы. Инструмент поддерживает интеграцию с корпоративными облачными платформами. Разумеется, Claude Code также предоставляет возможность интеграции в виде плагинов для IDE и даже может использоваться в Cursor путем интеграции через плагин.

Gemini CLI – это открытый агент командной строки, выпущенный Google. Он построен на модели Gemini 2.5 Pro и обеспечивает поддержку MCP, поиск внешнего контента и возможность работы с несколькими средами. Разработчики могут не только генерировать, отлаживать и комментировать код посредством диалога, но также читать и записывать файлы, выполнять команды командной оболочки, создавать документацию и даже вызывать инструменты генерации изображений и видео (такие как Veo и Imagen). Преимуществами Gemini CLI являются его открытый исходный код и обширное окно контекста, а также бесплатный дневной лимит запросов, что делает его чрезвычайно привлекательным для любителей настраивать цепочки инструментов.

По сравнению с Claude Code, Gemini CLI в значительной степени освобождает пользователей от ограничений в использовании ИИ: он не привязан к IDE и не зависит от коммерческой подписки, являясь скорее расширяемым и настраиваемым помощником DevOps.

На самом деле Claude Code и Gemini CLI можно считать попыткой разработки новой формы программирования, и, судя по всему, им есть куда расти. Одной из важных тенденций в развитии продуктов для программирования с использованием ИИ в будущем станет совместная работа с несколькими модальностями – не только понимание текста и генерация кода, но и постепенная интеграция речи, изображений и взаимодействия с интерфейсом. Это позволит преодолеть традиционные границы между «редактором, консолью и браузером» и обеспечить комплексный подход к разработке, отладке и развертыванию. Одновременно границы между функциональностью настольных, веб-, мобильных и облачных платформ будут становиться все более размытыми, а продукты для программирования с помощью ИИ эволюционируют в направлении унификации всех платформ и возможности программирования «в любое время и в любом месте».

3.4.2. Формы использования и сегментация пользователей

С макроэкономической точки зрения экосистему программирования с помощью ИИ можно условно разделить на две основные формы использования.

- **Продукты для программирования с ИИ-поддержкой**. Основная функция таких продуктов заключается в поддержке процессов программирования. Они выступают в качестве «усилителей» существующих рабочих процессов разработки и призваны помочь профессиональным специалистам писать код быстрее, эффективнее и с меньшими затратами времени. Например, Cursor, Cline, GitHub Copilot и Claude Code обычно интегрируются в традиционные IDE и предоставляют такие функции, как автодополнение, генерация и рефакторинг кода. Все эти продукты также пытаются эволюционировать в сторону сквозных решений. Например, Background Agent от Cursor уже реализовал часть сквозных возможностей, что можно назвать «частично сквозным» подходом, поскольку контекстное понимание, способность к запоминанию, обработка задач и организация агентов в таких продуктах могут управляться извне (например, через организацию правил в Cursor). Профессиональные разработчики могут настраивать или заниматься оркестрацией агентов в рамках фреймворка, но для обычных пользователей это требует опре-

деленных затрат на обучение, иначе им придется использовать только те возможности, которые по умолчанию интегрированы в фреймворк.

- **Продукты для сквозного программирования с помощью агентов**. Основная особенность таких продуктов заключается в их способности самостоятельно выполнять задачи разработки. Они, как правило, не привязаны к традиционным средам разработки и стремятся превратить профессиональных разработчиков из исполнителей кода в диспетчеров задач или рецензентов кода. Например, Devin, v0, Bolt, Replit, Lovable и YouWare обычно способны самостоятельно анализировать требования, писать код, устранять ошибки, а также осуществлять управление проектами и интеграцию с системами развертывания. Для подавляющего большинства обычных пользователей использование таких продуктов для программирования является настоящим вайб-программированием, поскольку они не должны задумываться ни о чем, кроме описания своих намерений на естественном языке. Эти продукты даже не стремятся показывать сгенерированный ИИ исходный код, они сразу предоставляют конечный результат (например, YouWare). Однако для профессиональных разработчиков ограничения таких продуктов более значительны, поскольку им обычно требуется более тонкий контроль над результатами генерации.

Используя концепцию уровней автономности, мы можем классифицировать доступные на рынке продукты для программирования с помощью ИИ в соответствии с уровнями автоматизации.

(1) **Уровень 1 – диалоговый помощник**. Этот уровень представляет собой самую раннюю и наиболее широко распространенную форму программирования с помощью ИИ. Типичными продуктами являются ChatGPT, Claude и Gemini, которые с помощью вопросов и ответов на естественном языке помогают пользователям генерировать фрагменты кода, объяснять логику и давать рекомендации по разработке. Хотя продукты программирования с помощью ИИ 1-го уровня обладают определенной способностью воспринимать контекст в рамках диалога, общая модель по-прежнему представляет собой «дискретный диалог», который не позволяет непрерывно отслеживать состояние проекта. Пользователям приходится самостоятельно анализировать, интегрировать и корректировать результаты работы ИИ, что делает такие продукты инструментами незначительной поддержки.

(2) **Уровень 2 – интеллектуальное сотрудничество**. Представленные такими продуктами, как Cursor, Windsurf и Claude Code, продукты ИИ-программирования уровня 2 выходят за рамки простой модели «дискретного диалога», глубоко интегрируя ИИ в среду разработки и обладая более полноценными функциями индексации контекста на уровне проектирования, интеллектуального автодополнения, распознавания взаимосвязей и совместной разработки. По сравнению с продуктами ИИ-программирования 1-го уровня, продукты 2-го уровня уделяют больше внимания системности и внедрению в инженерную практику. Они могут помогать в IDE при создании сложных функций, интеллектуальном рефакторинге, навигации по коду и синхронизации работы команды, что делает их мощным вспомогательным инструментом программистов. Обратите внимание: ИИ-программирование 2-го уровня не означает полностью автономную разработку от начала до конца, а представляет собой скорее режим совместной работы – ИИ помогает, а программист руководит.

(3) **Уровень 3 – автономная реализация**. Этот уровень представляет собой передовые направления исследований в области программирования с ИИ. Типичными продуктами являются Devin, Bolt, v0 и YouWare. Суть заключается в том, что ИИ способен самостоятельно выполнять полный цикл работ – от разбиения требований, разработки функций, тестирования и запуска до предоставления результатов в соответствии с конкретными потребностями. Продукты программирования ИИ 3-го уровня делают акцент на автономности и системной интеграции. Некоторые из них обладают возможностями совместной работы нескольких агентов, непрерывного отслеживания контекста, взаимодействия инструментальных цепочек и интеллектуальной корректировки на основе обратной связи, пытаясь эволюционировать в направлении «ИИ-члена команды» или виртуального инженера. Хотя в настоящее время продукты программирования ИИ 3-го уровня по-прежнему сталкиваются с трудностями в плане сложности и надежности, их концепция предвещает переход роли разработчика от исполнителя к проектировщику требований. На ИИ будет возлагаться все больше задач по внедрению инженерных решений.

3.5. Заключение

Современные продукты для вайб-программирования по-прежнему находятся в стадии активного развития: от самых ранних решений, основанных на использовании универсальных больших языковых моделей для помощи в программировании, до интеграции ИИ-помощников в IDE, а затем к сквозному программированию с помощью агентов. В конечном итоге это привело к появлению интегрированной интеллектуальной системы, охватывающей разработку, тестирование, развертывание и совместную работу. Такие продукты, как Cursor и Windsurf, способствуют совместной работе на уровне команд, а Devin, Bolt и YouWare открывают новые горизонты в области сквозного автономного программирования. Продукты Claude Code и Gemini CLI также экспериментируют с различными формами прикладного использования.

Глава 4

Вайб-программирование: сценарии и практические примеры

Истинная ценность любой технологии зависит от возможностей ее применения в реальных условиях. Вайб-программирование – это новая парадигма разработки с использованием ИИ для автоматической генерации кода посредством взаимодействия на естественном языке. Результаты его практического применения должны показать, является ли эта технология лишь красивой концептуальной идеей или же она сможет стать реальным инструментом трансформации способов разработки ПО.

В условиях глобальной цифровизации экономики новаторские технологии в области программирования становятся основным двигателем модернизации промышленности. В основе вайб-программирования находится диалоговая модель разработки по принципу «что видишь, то и получаешь»: программисту достаточно использовать команды на естественном языке, чтобы мгновенно преобразовать мимолетные идеи в рабочий код. Такой подход значительно снижает технический порог вовлечения и стимулирует появление совершенно новой бизнес-экосистемы.

Рыночные данные убедительно подтверждают эту тенденцию.

- Согласно статистике Grand View Research, в 2024 году объем мирового рынка ИИ достиг $279,22 млрд, а к 2030 году, по прогнозам, превысит $1,8 трлн, при этом среднегодовой темп роста (CAGR) в 2025–2030 годах составит 35,9 %.
- Статистика Exploding Topics показывает, что по состоянию на июль 2025 года объем мирового рынка ИИ составил около $391 млрд, а среднегодовой темп роста отрасли в целом был около 35,9 %.

На уровне применения примечательны как практики новых платформ, так и ведущих компаний отрасли.

Платформа Replit: 75 % пользователей Replit до первого использования никогда не писали ни одной строчки кода, но благодаря функции Ghostwriter AI им удалось успешно преобразовать свои идеи в готовые продукты.

Google: генеральный директор Сундар Пичаи (Sundar Pichai) в ходе телеконференции по финансовым результатам за третий квартал сообщил, что более

25 % нового кода генерируется ИИ. Это свидетельствует о том, что автоматизация кода на основе естественного языка глубоко интегрирована во внутренние процессы разработки компании.

Y Combinator: управляющий партнер Джаред Фридман (Jared Friedman) на мероприятии W25 Demo Day (зима 2025 года) упомянул, что у четверти участвующих стартап-команд более 95 % основного репозитория кода было сгенерировано ИИ. Это свидетельствует о мощной поддержке, которую ИИ-программирование оказывает на ранних этапах стартапа.

На основе приведенных выше рыночных данных и практических примеров можно наглядно оценить эффективность вайб-программирования, проанализировать границы его применения в качестве методологии и предсказать перспективы дальнейшего развития.

В этой главе будет проведена системная классификация применения вайб-программирования в различных сферах и для разных участников процесса. На основе комплексного анализа типов задач, моделей сотрудничества и результатов внедрения будут раскрыты уникальные преимущества этой методологии в плане повышения эффективности разработки, снижения расходов на совместную работу и ускорения внедрения инноваций.

Мы отобрали несколько типичных примеров – от создания прототипов продуктов до автоматизации бизнес-процессов, от оптимизации оценки образования до быстрой реализации идей, чтобы продемонстрировать практическую применимость и потенциал вайб-программирования в реальных условиях.

4.1. Анализ сценариев применения

Благодаря таким характеристикам, как низкоуровневое программирование, визуализация и нацеленность на результат, вайб-программирование предлагает совершенно новый подход к разработке ПО. Оно значительно снижает порог вхождения в программирование и расширяет границы применения технологий за счет высокой эффективности разработки. Вайб-программирование демонстрирует уникальные преимущества как при создании прототипов продуктов, реализации личных идей и обучении программированию, так и при автоматизации внутренних бизнес-процессов. В этом разделе мы рассмотрим четыре типичных сценария применения.

4.1.1. Быстрое создание прототипов продуктов

На ранних этапах разработки продукта крайне важно проверить осуществимость идеи. Непроверенные идеи могут сталкиваться с техническими препятствиями или несоответствием потребностям рынка, а непосредственный переход к масштабной разработке чреват большими затратами времени и средств, а также риском провала продукта. Проверка осуществимости путем быстрого создания прототипа означает, что в случае неудачи все вложенные на ранних этапах ресурсы (включая затраты на разработку, трудозатраты, ресурсы на дизайн и т. д.) не принесут ожидаемой отдачи. Однако неудача на ранней стадии имеет и положительный аспект: она позволяет избежать продолжения работы в неверном направлении до стадии зрелости продукта, сэкономить команде более значительные издержки на последующую масштабную разработку

и продвижение на рынке, а также дает возможность быстро скорректировать стратегию и перенаправить ресурсы в более перспективное русло.

Использование технологий программирования с помощью ИИ, представленных большими языковыми моделями, позволяет значительно снизить затраты на этапе разработки прототипа. С точки зрения временных затрат, традиционный подход с ручным написанием прототипа может занять несколько недель или даже месяцев, тогда как программирование с помощью ИИ благодаря анализу команд на естественном языке и интеллектуальному созданию кода позволяет сократить цикл разработки до нескольких дней или даже часов. Что касается затрат на персонал, программирование с помощью ИИ снижает зависимость от профессиональных разработчиков, позволяя выполнять разработку людям и небольшим командам, не имеющим опыта написания кода, что сокращает расходы на аутсорсинг или найм высокооплачиваемых разработчиков.

Возьмем в качестве примера небольшую студию, специализирующуюся на разработке продуктов в сфере культуры и творчества. Эта команда планирует выпустить интерактивную открытку с изображением масок оперных персонажей, где будет использована технология дополненной реальности (Augmented Reality, AR). После сканирования рисунка маски на открытке пользователь сможет посмотреть видео с классическими ариями соответствующего персонажа оперы и ознакомиться с информацией о его истории. При использовании традиционной модели разработки потребовалось бы сформировать профессиональную команду из инженеров по AR, фронтенд-разработчиков, видеомонтажеров и других специалистов, которые бы писали код с нуля. Срок разработки составил бы 5–7 недель, а затраты на персонал могли бы достичь нескольких сотен тысяч юаней (десятков тысяч долларов. – *Прим. перев.*). По этой причине команда решила опробовать разработку с использованием большой языковой модели программирования. Для начала разработчики разбили требования к продукту на конкретные задачи в виде модулей, а затем просто ввели в интерфейс редактора запрос: «создать функцию распознавания грима на основе AR, которая после его распознавания будет воспроизводить соответствующее видео и текстовую информацию», и большая языковая модель программирования с помощью встроенного AR-фреймворка и библиотеки мультимедийного воспроизведения смогла быстро сгенерировать смешанный код на JavaScript и Python.

В процессе разработки, столкнувшись с проблемой медленной загрузки видео, разработчики задали модели вопрос: «как оптимизировать скорость загрузки видео на мобильных устройствах?» Модель мгновенно проанализировала код и предложила использовать технологию фрагментированной загрузки с компрессией битрейта видео. После завершения разработки интерфейса с помощью ИИ-инструментов команда создала прототип всего за 6 дней, сократив затраты на персонал более чем на 70 %.

Еще один пример: независимый разработчик, занимающийся созданием приложений в сфере фитнеса, использовал вайб-программирование для реализации функций распознавания движений и предоставления обратной связи, заложив тем самым основу для дальнейшего развития функциональности продукта. Несмотря на возможность дальнейшей оптимизации, этот инструмент значительно сократил время от идеи до первоначальной реализации.

Такая возможность дает индивидуальным разработчикам, небольшим командам и даже стартапам возможность опробовать свои силы на ранних этапах с минимальными затратами, что повышает вероятность успешной реализации продукта.

4.1.2. Возникновение «массового программирования»

В разделе 1.1 был описан случай, когда учительница на пенсии, госпожа Ван, разработала приложение для семейного альбома с помощью инструментов программирования на базе ИИ. Изначально она была обычным пользователем с туманными представлениями о своем проекте, но в процессе диалога и взаимодействия с ИИ постепенно уточнила свои требования – от загрузки фотографий и входа через WeChat до классификации по тегам и фоновой музыки. Инструменты программирования на базе ИИ смогли оценить ее замысел и автоматически выполнить такие этапы разработки, как создание бэкенд-интерфейсов, моделирование базы данных и генерация интерфейса пользователя.

На протяжении всего процесса госпоже Ван не пришлось писать ни одной строчки кода и не требовалось владеть сложной профессиональной терминологией. Ей достаточно было общаться с ИИ, как в повседневной жизни, и вносить предложения по доработке, например «хочу, чтобы оболочка была более уютной» или «нельзя ли ускорить генерацию видео». ИИ воспринимал смысл ее слов и автоматически оптимизировал стиль интерфейса или алгоритмы бэкенда. Всего за 3 дня она создала полнофункциональный прототип приложения.

Этот пример наглядно демонстрирует высокую эффективность инструментов программирования на основе ИИ в распознавании и интерпретации естественного языка, а также показывает, что творческие идеи «непрограммистов» могут напрямую приводить к созданию новых продуктов. Именно в этом заключается суть тенденции «всеобщего программирования»: резкое снижение технического барьера позволяет даже рядовым пользователям играть ведущую роль в процессе разработки продуктов.

4.1.3. Инструмент для новичков и препятствия для продвинутых

Интуитивно понятный визуальный интерфейс и отсутствие необходимости в написании кода сделали вайб-программирование мощным инструментом для обучения детей младшего возраста и начинающих программистов. Уникальный «конструкторский рабочий стол» оснащен интеллектуальной системой подсказок, которая поддерживает свободное перемещение модульных компонентов и настройку параметров, а также преобразует сложную логику кода в наглядные действия с помощью динамических подсказок и функции воспроизведения шагов.

Пользователю достаточно построить процесс с помощью графического интерфейса, и система автоматически сгенерирует соответствующий код, обеспечивая интересное взаимодействие по принципу «созидание как обучение», как показано на рис. 4-1.

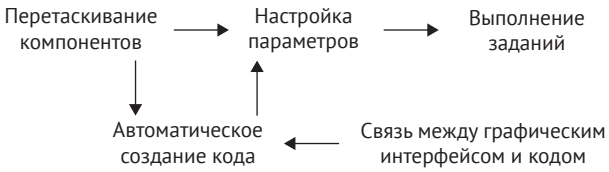


Рисунок 4-1. Замкнутый цикл «созидание как обучение»

В качестве примера можно привести кружок программирования в одной из начальных школ Шанхая, где учитель использовал как учебный пример «разработку простого электронного фотоальбома» и показал эффективность обучения с помощью вайб-программирования. В ходе 10-минутной демонстрации работы с модулями учитель с помощью визуального интерфейса перетаскивал модули отображения изображений, настраивал компоненты эффектов переключения, а также задавал параметры анимации и логику отображения. ИИ-инструмент в режиме реального времени генерировал окно предварительного просмотра кода с синхронным отображением каждого фрагмента кода, соответствующего выполненной операции. Это помогало ученикам установить взаимосвязь между графическим интерфейсом и кодом. Затем, в ходе 40-минутного практического занятия, ученик Ли импортировал фотографии семейной поездки в ИИ-инструмент. С помощью настройки размера изображений, добавления текстовых комментариев и применения эффекта плавного перехода «затухание» он завершил разработку альбома всего за 20 минут. Еще более творческим решением стало гибкое использование механизма триггеров событий: путем связывания мультимедийных компонентов он создал интерактивный эффект, при котором нажатие на фотографию приводило к синхронному воспроизведению соответствующего видео из путешествия. Такой подход, сочетающий обучение с развлечением, позволил ученикам естественным образом освоить основные концепции программирования, такие как последовательное выполнение и условные операторы, в процессе практической работы. По окончании курса 90 % учеников успешно завершили работу над своими проектами, 80 % из них выразили глубокий интерес к программированию, а некоторые даже самостоятельно пробовали добавлять такие расширенные функции, как фоновая музыка и навигационное меню.

4.1.4. Автоматизация внутренних процессов компании: повышение эффективности и проблемы интеграции

Повторяющиеся и рутинные рабочие задачи в повседневной деятельности предприятия отнимают у сотрудников много времени и сил, что снижает эффективность труда и повышает вероятность возникновения ошибок. Благодаря отсутствию необходимости в написании кода и визуальному интерфейсу вайб-программирование предлагает эффективный путь к автоматизации повседневных рабочих процессов, помогая предприятиям сократить расходы и повысить результативность.

Отдел маркетинга рекламной медиакомпании ежемесячно загружает аналитические данные из различных источников, включая социальные сети, рекламные панели поисковых систем и центры обработки данных электронных торго-

вых платформ, после чего проводит их очистку, унификацию формата и подготовку графиков для составления итоговых отчетов. Раньше на выполнение этой работы у трех сотрудников уходило два дня, при этом в процессе копирования и вставки данных, а также при расчете формул нередко возникали ошибки.

В конце 2023 года отдел приступил к автоматизации этого процесса. Руководитель отдела вместе с двумя сотрудниками, обладающими базовыми знаниями в области информатики, обратился к большой языковой модели с командой на естественном языке: «сделай скрипт для автоматической загрузки данных из TikTok, рекламной платформы Baidu и панели управления Taobao, очисти аномальные данные, преобразуй их в формат CSV, сгенерируй столбчатые и линейные диаграммы с показателями переходов и конверсии, а затем выведи ежемесячный отчет по маркетинговой аналитике». На основе этого указания большая языковая модель задействовала алгоритмы и шаблоны, связанные со сбором, очисткой и визуализацией данных, в результате чего быстро сгенерировала гибридный скрипт на Python и Shell. После небольшой доработки сотрудники смогли успешно развернуть систему автоматизации. После запуска системы работа, которая раньше занимала 2 дня, стала выполняться за 1 час, а точность данных достигла практически 100 %, что позволило сотрудникам уделять больше внимания анализу данных и разработке стратегий.

4.2. Подробный анализ практических примеров

Программирование с помощью ИИ стало необратимой тенденцией в отрасли, а вайб-программирование выступает ее представителем и первопроходцем. В этом разделе мы подробно, как под лупой, проанализируем изменения и возможности, которые приносит вайб-программирование, в контексте независимых разработчиков, стартапов и других участников рынка. От независимых разработчиков, которые с его помощью заявили о себе в этой новой сфере, до стартапов, которые благодаря этому стремительно набирают популярность, и крупных предприятий, которые с его помощью осуществляют стратегическую трансформацию, а также благодаря влиянию вайб-программирования в сообществе разработчиков ПО с открытым исходным кодом поднимается волна инноваций. Вайб-программирование всесторонне меняет ландшафт разработки ПО, создавая беспрецедентные возможности и определяя новые цели для всех участников рынка.

Обратите внимание: хотя термин «вайб-программирование» официально появился только в 2025 году, лежащая в его основе концепция постепенно получила широкое признание еще с начала применения больших языковых моделей в разработке ПО. Приведенные ниже примеры, возможно, не подпадают под это определение напрямую, но на самом деле все они исходят из одного и того же ядра: новой парадигмы разработки, в которой естественный язык играет ведущую роль, а ИИ выступает в качестве основного инструмента.

4.2.1. Примеры успеха независимых разработчиков

Появление вайб-программирования открыло для независимых разработчиков дверь в новый мир, превратив мечту о «разработке приложений без знания кода» из несбыточной в реальность. Согласно отчету GitHub Octoverse 2024,

за последний год количество проектов в области генеративного ИИ выросло на 98 %, а количество связанных с ними внедрений увеличилось на 59 %. Кроме того, количество учителей, студентов и технических специалистов, использующих Copilot, выросло вдвое. Это показывает, что благодаря взаимодействию на естественном языке и инструментам программирования на базе ИИ все больше неспециалистов вступают в ряды разработчиков, а вклад в разработку кода значительно увеличивается. Эти впечатляющие данные наглядно демонстрируют мощный эффект, который вайб-программирование оказывает на независимых разработчиков.

Далее на примере нескольких типичных случаев мы рассмотрим примеры того, как независимые разработчики, руководствуясь этой концепцией, совершают качественный скачок от идеи к реализации.

1. Первопроходец в сфере социальных сетей для знакомств

История стартапа Блейка Андерсона (Blake Anderson) является настоящим примером того, как ИИ может помочь разработчикам-одиночкам, и вполне может сравниться с захватывающей легендой о стартапе. Будучи выпускником университета 2023 года, он благодаря глубокому пониманию проблем «поколения Z» в сфере общения принял решение заняться созданием соответствующего социального сервиса. На этапе разработки он не тратил огромное количество времени на написание сложного кода, как это делают традиционные разработчики, а четко описывал функциональные требования программным инструментам на естественном языке, например «разработать модуль, способный генерировать ответы с высоким уровнем эмпатии на основе сценариев общения». При этом программные инструменты работали быстро: с помощью ИИ они автоматически генерировали соответствующий каркас кода, а затем проводили обучение и оптимизацию на основе 5 млн реальных данных общения в чатах.

Дополнительная информация

«Поколение Z» – это интернет-термин, обычно обозначающий людей, родившихся в период с 1995 по 2009 год. С самого рождения они были неразрывно связаны с сетевой информационной эпохой и находились под значительным влиянием цифровых технологий, устройств мгновенного обмена сообщениями и смартфонов.

Блейк создал приложение для знакомств Plug AI (ранее известное как RizzGPT, см. рис. 4-2). Помимо анализа диалогов в реальном времени и распознавания эмоций, Plug AI предлагает инновационную функцию «портрета личности в отношениях», которая с помощью семантического анализа эмоций подбирает пользователю индивидуальную стратегию общения, словно предоставляя ему «наставника по общению в отношениях». Такая модель глубокого взаимодействия значительно повысила привлекательность продукта: коэффициент конверсии в платную подписку достиг 35 %, что намного превышает средний показатель по отрасли. В плане бизнес-стратегии Блейк использовал комбинированную модель «бесплатные базовые функции + премиум-подписка + виртуальные товары», благодаря чему **средний доход от платного пользо-**

вателя (Average Revenue Per Paying User, ARPPU) достиг $12,8. Благодаря маркетинговой кампании с хештегом #RizzGPTChallenge в TikTok пользовательские видеоролики «Уроки любви от ИИ» быстро обрели популярность, а общее количество просмотров превысило 2,7 млрд. Это помогло компании заработать более $10 млн в первый год после запуска продукта, что стало примером успешного стартапа, созданного независимым разработчиком с помощью ИИ.



Рисунок 4-2. Plug AI

2. Из автора интернет-романов в полноценного разработчика

Не менее впечатляющим и достойным восхищения является путь трансформации китайского разработчика Чжао Чуньсяна. Переход от автора интернет-романов к полноценному разработчику – это, казалось бы, невозможное преобразование, которое с помощью вайб-программирования прошло гладко и непринужденно. Он использовал Midjourney для прототипирования пользовательского интерфейса, а затем с помощью программирования на естественном языке преобразовал эти визуальные идеи в функциональный код. Всего за 8 месяцев ему удалось разработать 23 приложения в различных вертикальных областях, продемонстрировав потрясающую креативность и работоспособность.

В частности, приложение «Книга желудка» (см. рис. 4-3) объединяет модель большого языка и граф знаний о еде, предлагая пользователям совершенно новый опыт в области кулинарии. Достаточно ввести комбинацию ингреди-

ентов, таких как «помидоры» и «яйца», и приложение «Книга желудка» на основе анализа ИИ сгенерирует оценку питательной ценности, пошаговую инструкцию по приготовлению и индивидуальные рекомендации по питанию, работая как личный диетолог. Это приложение вызвало бум ИИ-рецептов в социальной сети Xiaohongshu. Благодаря уникальным функциям и высокому качеству обслуживания это приложение продержалось в тройке лидеров рейтинга кулинарных приложений в App Store 14 дней подряд, а месячный доход от рекламы и подписок достиг $17 000.

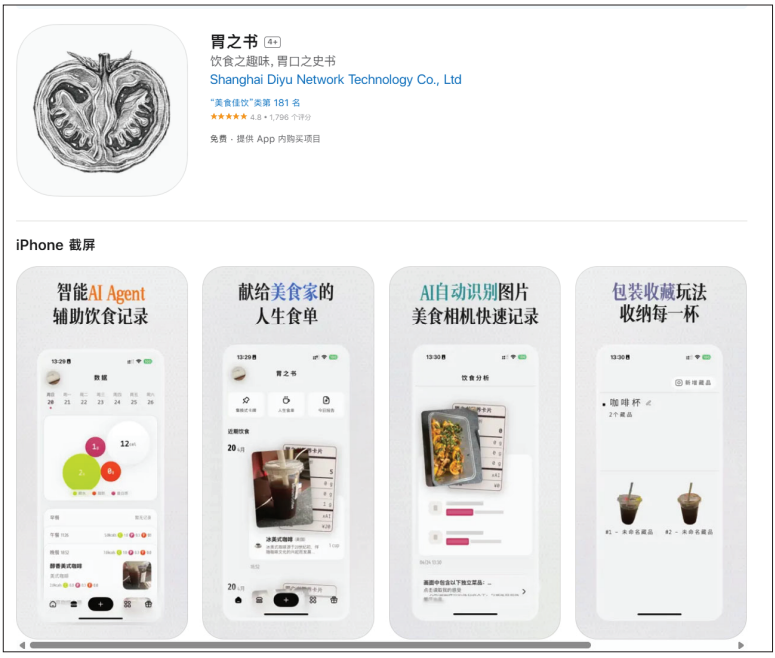


Рисунок 4-3. Приложение «Книга желудка»

3. Мировой рекордсмен разработки за несколько часов

Разработчик приложения «Кошачья подсветка» (см. рис. 4-4) Чэнь Юньфэй максимально эффективно использовал возможности вайб-программирования и получил широкую известность в профессиональном сообществе. Столкнувшись с реальной потребностью блогеров Xiaohongshu в функции подсветки, он ввел в редактор Cursor команду на естественном языке: «разработать функцию интеллектуальной подсветки с двумя областями и с поддержкой режима обработки фотографий в режиме реального времени (Live Photo)». Прототип продукта был разработан всего за один час. После запуска приложения количество ежедневных загрузок превысило 20 000, а конверсия в платную версию Pro достигла 22 %. Это позволило воплотить в жизнь легендарную концепцию «разработка за час, доход в миллионы» и наглядно продемонстрировать огромный потенциал вайб-программирования для независимых разработчиков.



Рисунок 4-4. Приложение «Кошачья подсветка»

4. Разработчик-путешественник

Голландский разработчик Питер Левелс (Pieter Levels) путешествует по миру и при этом с помощью вайб-программирования создает эффективные решения. Его продукты, такие как Nomad List (Список кочевника), помогают цифровым кочевникам выбрать идеальное место для жизни по критериям уровня жизни, климата, безопасности и скорости интернета в более чем 1000 городов из 187 стран (регионов). Приложение стало крупнейшей в мире базой данных городов для цифровых кочевников и приносит стабильную прибыль за счет подписок и рекламы. Решение Remote OK – это платформа для удаленной работы, которая впоследствии была приобретена GitLab. Приложение Photo AI ежемесячно генерирует около 1 млн ИИ-фотографий и постоянно оптимизирует продукт на основе пользовательских отзывов. Этот пример наглядно иллюстрирует гибкость творческого потенциала и возможности постоянного развития, которые дает разработчикам вайб-программирование.

4.2.2. Опыт практического использования стартапами

Вайб-программирование, подобно волшебной палочке, превращающей камень в золото, постепенно перестраивает экосистему стартапов и становится залогом успеха для начинающих компаний, реализующих концепцию «маленькая команда – большие инновации». Согласно открытым данным мероприятия Y Combinator W25, управляющий партнер Y Combinator Джаред Фридман (Jared Friedman) в одном из интервью сообщил, что у 25 % стартап-команд более 95 % кода в их основном репозитории сгенерировано с помощью ИИ. Эта поразительная цифра наглядно демонстрирует глубокое проникновение и огромную ценность вайб-программирования в сфере стартапов – подхода, основанного на использовании естественного языка и доминировании ИИ в процессе кодирования.

В области медицинских технологий один стартап, состоящий всего из одного инженера и двух менеджеров по продукту, несмотря на небольшую команду, смог продемонстрировать поразительную креативность благодаря концепции вайб-программирования. Опираясь на платформу Replit, они смогли оперативно воплотить в жизнь идею распознавания пищи по фотографиям и прогнозирования рисков для здоровья. С помощью диалогового взаимодействия члены команды отдавали программным инструментам четкие команды, например «создать модуль распознавания изображений на основе GPT-4V и реализовать анализ рисков для здоровья с помощью сети с долгой краткосрочной памятью (Long Short-Term Memory, LSTM)». Инструменты ИИ автоматически генерировали соответствующий код в соответствии с парадигмой вайб-программирования и быстро объединили все модули. В результате было разработано приложение Cal AI, точность распознавания продуктов в котором достигает 98,7 %. Еще на этапе внутреннего тестирования этот продукт получил финансирование в размере $20 млн в рамках раунда Pre-A благодаря своей выдающейся производительности и инновационной концепции. Это полностью доказало огромный потенциал предпринимательских инноваций на основе вайб-программирования и способность небольших команд добиваться успеха даже в условиях жесткой рыночной конкуренции.

Ряд инновационных решений появился и в сфере образовательных технологий. Компания GradeWiz, основанная студентами Корнельского университета Максом Бохуном (Max Bohun) и Аманом Гаргом (Aman Garg), была успешно отобрана для участия в зимнем потоке Y Combinator 2025 года. Команда разработала систему автоматической проверки заданий по математике и программированию на основе технологий ИИ и компьютерного зрения. Система уже проверила более 30 000 заданий в таких вузах, как Государственный университет Пенсильвании, Корнельский университет, Колледж Хантера при Городском университете Нью-Йорка, Калифорнийский государственный политехнический университет и Университет Сиракуз. По сравнению с привычной ручной проверкой, GradeWiz повышает эффективность работы ассистентов преподавателей примерно на 60 %, что в среднем экономит каждому из них на проверке работ около 4 часов в неделю. Благодаря эффективной и надежной автоматической оценке GradeWiz открывает новый путь в сфере оценок в образовании с помощью ИИ, привнося мощный технологический импульс в эту отрасль.

4.2.3. Трансформация крупных предприятий

Ведущие технологические компании активно внедряют вайб-программирование в свои системы стратегического планирования, вооружаясь этим мощным инструментом в жестких условиях рынка и видя в нем важный ресурс для повышения своей конкурентоспособности.

На конференции I/O 2024 компания Google сообщила, что инструменты ИИ-программирования на основе естественного языка уже играют ключевую роль в крупных корпоративных проектах. В рамках одного из проектов по переносу основного кода около 74 % изменений были выполнены автоматизированными инструментами, что сократило общее время переноса примерно на 50 %. Несмотря на отсутствие открытых данных о временных затратах на миграцию кода в рамках обычного процесса рецензирования, этот результат наглядно

подтверждает, что генерация и модификация кода с помощью взаимодействия на естественном языке способны значительно повысить эффективность работы команды над сложными проектами.

Также впечатляют результаты GitHub Copilot в контролируемых экспериментах: ускорение выполнения типичных задач кодирования составляет около 55 %. Эти открытые данные однозначно показывают, что генерация и автодополнение кода с помощью взаимодействия на естественном языке способны значительно повысить эффективность разработки сложных проектных задач. Программистам достаточно описать функциональные требования, например «создать модуль для мониторинга состояния устройств в реальном времени», и система автоматически сгенерирует готовый код. Цикл разработки ключевых функциональных модулей уменьшился с нескольких месяцев до нескольких недель, что значительно сократило время вывода продукта на рынок и повысило скорость реагирования на рыночные изменения.

Китайский технологический гигант Tencent также внедряет концепцию вайб-программирования в свои основные продукты: разработанный компанией программный помощник Tian Gong глубоко интегрирован во весь процесс разработки.

- **Уровень плагинов IDE**: встроен в модифицированную версию VS Code, разработчики могут напрямую отправлять запросы на естественном языке прямо в редакторе.
- **Конвейер CI/CD**: на каждом этапе компиляции и сборки Tian Gong автоматически проверяет, дополняет и оптимизирует скрипты и настройки, обеспечивая высокое качество конечного продукта.
- **Цепочка инструментов игрового движка**: благодаря бесшовной интеграции с внутренним движком скрипты навыков, логика взаимодействия с пользовательским интерфейсом и настройки спецэффектов генерируются одним нажатием и автоматически добавляются в пакет ресурсов.

На примере итерации нового сезона игры Honor of Kings после ввода командой в IDE фразы «генерировать код боевых навыков с крутыми эффектами» в Tian Gong сразу же были созданы полные скрипты навыков, логика взаимодействия с пользовательским интерфейсом и настройки спецэффектов, что позволило повысить эффективность итерации версий примерно на 50 %, а показатель лояльности игроков в первую неделю вырос на 8,3 %. Такая глубокая интеграция не только подтвердила превосходную эффективность вайб-программирования в крупных сложных проектах, но также помогла Tencent Games сохранить лидерство в условиях жесткой конкуренции.

4.2.4. Адаптация и инновации сообщества Open Source

Возникновение вайб-программирования добавило сообществу разработчиков с открытым исходным кодом беспрецедентную энергию, словно брошенный в спокойную гладь озера камень вызвал волны инноваций. Это способствовало глубоким структурным изменениям в экосистеме разработки Open Source.

В официальном отчете GitHub Octoverse 2024 отмечается, что в 2024 году количество ИИ-проектов и их вклад на платформе GitHub значительно выросли. Это свидетельствует о становлении ИИ в качестве важной движущей силы развития сообщества разработчиков с открытым исходным кодом.

В дополнение к этому научные исследования свидетельствуют о растущем влиянии ИИ на процесс обработки запросов на внесение изменений в код (Pull Request, PR). Например, в исследовании Generative AI for Pull Request Descriptions: Adoption, Impact, and Developer Interventions (Генеративный ИИ для описаний Pull Request: внедрение, влияние и участие разработчиков), опубликованном в 2024 году, были проанализированы 18 256 случаев автоматического создания описаний с помощью Copilot. Результаты исследования показали следующее.

- Использование Copilot for PRs (то есть функции автоматического создания описаний фиксаций) растет быстрыми темпами.
- Время рецензирования PR с описаниями, созданными с помощью ИИ, сокращается, а коэффициент слияния повышается.
- Разработчики зачастую дорабатывают автоматически сгенерированный контент вручную.

Эти исследования указывают на то, что хотя в некоторых областях попрежнему требуется ручная корректировка, ИИ существенно повышает эффективность процессов взаимодействия и становится важной составляющей при разработке инструментов для открытого сотрудничества нового уровня.

В экосистеме TensorFlow разработчики с помощью команд на естественном языке генерируют код для распределенного обучения и таким образом повышают эффективность обучения. Код, автоматически сгенерированный в рамках программного фреймворка Vibe, повышает эффективность обучения моделей примерно на 30 %, что способствует постоянному развитию этого открытого фреймворка в области оптимизации производительности и позволяет ему сохранять лидирующие позиции среди фреймворков ИИ с открытым исходным кодом во всем мире.

В то же время новые проекты с открытым исходным кодом активно исследуют новые модели взаимодействия человека и машины, способствуя постоянному расширению границ инноваций в сфере Open Source. Например, проект AutoML-Zero полностью автоматизирует генерацию кода алгоритмов машинного обучения с помощью команд на естественном языке, что кардинально меняет традиционные подходы к разработке алгоритмов и открывает новые горизонты для автоматизированного моделирования и инноваций в области машинного обучения.

Еще одной важной тенденцией является то, что на фоне быстрого развития сообщества Open Source развернулась беспрецедентно глубокая дискуссия по вопросам этики «автономной разработки ИИ». Статистика показывает, что в дискуссионных разделах проектов GitHub ежедневно появляется более 200 новых обсуждений на эту тему, которые охватывают такие аспекты, как безопасность моделей, достоверность кода и авторские права на сгенерированный контент. Это свидетельствует о том, что вайб-программирование способствует техническому развитию открытых проектов и одновременно стимулирует всю экосистему открытого исходного кода уделять внимание не только эффективности и интеллекту, но и формированию этических норм и консенсуса ценностей, способствуя постоянному развитию сообщества в направлении большей эффективности, открытости и ответственности.

4.3. Заключение

Благодаря анализу типичных сценариев и демонстрации конкретных примеров мы убедились, что вайб-программирование уже доказало свою практическую ценность во многих областях. Оно значительно снизило порог входа в разработку и изменило подход к взаимодействию человека и машины, становясь важным инструментом для инновационного развития и быстрого внедрения. Однако для полного раскрытия его потенциала недостаточно одного лишь инструмента – необходим набор научно обоснованных методов и система инженерных практик.

Поэтому в главе 5 мы более подробно рассмотрим передовые методы вайб-программирования: от разработки требований до моделей взаимодействия, от управления качеством до устойчивого развития. Мы сформулируем набор практических рекомендаций, которые помогут разработчикам более эффективно и уверенно переходить к новой парадигме вайб-программирования.

Глава 5

Лучшие практические методики

Несмотря на высокую эффективность, ИИ-генерация кода иногда не учитывает такие ключевые факторы долгосрочной стабильной эксплуатации, как качество кода, удобство обслуживания, безопасность и продуктивность. Поэтому эта глава посвящена детальному описанию методов внедрения научно обоснованных инженерных практик и процессов управления качеством в рамках модели вайб-программирования.

Для начала в этой главе представлены методы высококачественного проектирования промптов, или подсказок (prompt engineering), которые помогают разработчикам эффективнее направлять работу ИИ с помощью точных инструкций. Затем на примере планирования запросов будет показано, как четкое и структурированное описание требований становится основой для точной ИИ-генерации кода. Кроме того, в этой главе будет уделено внимание процессам рецензирования и оптимизации кода, которые помогут разработчикам выявлять и устранять различные ошибки и уязвимости в коде, сгенерированном ИИ. В заключение будет представлен упрощенный инженерный набор инструментов с поддержкой контроля версий, автоматизированного тестирования, автоматической сборки и развертывания (CI/CD), а также создания документации с помощью ИИ. Все это призвано обеспечить стабильную и долгосрочную разработку высококачественного корпоративного ПО с помощью вайб-программирования.

Из этой главы вы узнаете о лучших инженерных методиках, позволяющих обеспечить качество кода, снизить затраты на обслуживание и повысить надежность проектов при ИИ-разработке, что позволит в полной мере раскрыть потенциал вайб-программирования.

5.1. Методики разработки промптов

При вайб-программировании в теории разработчику достаточно лишь четко описать цели и требования, после чего большая языковая модель автоматически сгенерирует соответствующую логику и архитектуру кода, избавив тем самым от рутинного ручного кодирования. Таким образом, основной акцент в дальнейшей работе смещается с непосредственного кодирования на разработку промптов. Разработка промптов – это технология создания и оптими-

зации инструкций (подсказок), которые передаются ИИ с целью получения желаемого результата от большой языковой модели. Тщательно разработанный промпт может значительно повысить степень понимания ИИ намерений пользователя и качество генерируемого кода. Учитывая зависимость вайб-программирования от взаимодействия на естественном языке, овладение эффективными навыками промпт-инженерии имеет решающее значение для полного раскрытия потенциала ИИ в процессе кодирования, а также для повышения эффективности разработки и качества кода. В этом разделе мы подробно рассмотрим способы эффективного использования навыков промпт-инженерии для управления работой больших языковых моделей в практике вайб-программирования.

5.1.1. Почему важны хорошие промпты

Ключ к пониманию принципа работы больших языковых моделей заключается в осознании того, что они не «понимают» код или требования в прямом смысле, а осуществляют сопоставление шаблонов и генерацию контента на основе огромных массивов обучающих данных и вероятностных алгоритмов. Основной механизм работы больших языковых моделей заключается в последовательном прогнозировании: модель принимает входной текст (т. е. промпт) и, с учетом уже сгенерированного контента, прогнозирует следующий наиболее вероятный токен[1]. Этот цикл повторяется до тех пор, пока не будет достигнута заданная длина или не будет обнаружен маркер завершения. Общая логика процесса показана на рис. 5-1.

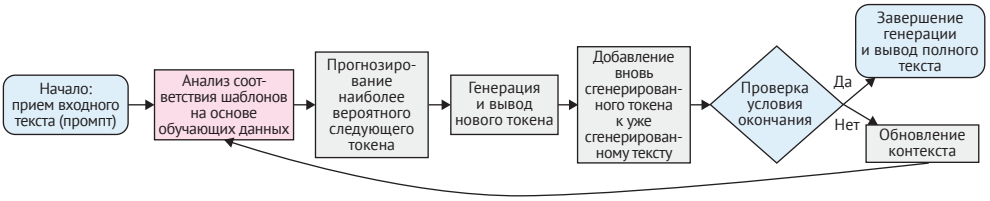


Рисунок 5-1. Процесс работы большой языковой модели

Несмотря на общепризнанную эффективность базовой логики на основе последовательного прогнозирования, у больших языковых моделей по-прежнему существуют системные недостатки, наиболее заметным из которых является отсутствие способности понимать контекст так же, как это делает человек. Поэтому качество их вывода в значительной мере зависит от качества и разнообразия обучающих данных. Если в обучающих данных присутствуют искажения или отсутствуют некоторые сценарии, большие языковые модели могут генерировать предвзятые, неточные или даже полностью ошибочные результаты.

Для устранения этого недостатка был разработан **механизм внимания** (attention mechanism), который можно рассматривать как способ модели имитировать присущую человеку практику придания различной важности разным словам при чтении с динамическим фокусированием на наиболее релевантных

[1] В области обработки естественного языка токен (token) – это минимальная семантическая единица текста.

частях при обработке входной последовательности. В частности, каждое слово входной последовательности преобразуется в три вектора: запрос (query), ключ (key) и значение (value). Модель вычисляет оценку внимания путем расчета сходства вектора запроса одного слова с векторами ключей всех слов во входной последовательности (как правило, с помощью скалярного произведения). Эти оценки после нормализации (например, с помощью функции Softmax) используются для взвешенного суммирования векторов значений, что позволяет сгенерировать для каждого слова новое отображение с учетом контекстуальной информации.

Этот механизм позволяет модели улавливать долгосрочные зависимости между словами и тонкие семантические связи. Например, в предложении «The cat sat on the mat because it was warm» модель определяет, что «it» относится к «the mat», а не к «the cat». Именно эта способность фокусироваться на ключевой информации посредством динамической регулировки внимания позволяет большим языковым моделям генерировать более семантически связный, контекстно-зависимый текст и код, который лучше соответствует особенностям человеческого языка.

Исходя из сказанного выше, можно сделать вывод, что качественные подсказки играют ключевую роль в управлении процессом. Когда мы предоставляем модели промпт, она активирует связанные с ним внутренние шаблоны и контекст, на основе которых генерирует последующий текст с учетом соответствующей вероятности. Четкие, ясные и контекстуально насыщенные промпты помогают модели более эффективно использовать механизм внимания, направляя ее «запрос» на «значения», связанные с ожидаемым выходом. В результате модель генерирует более точные результаты. Напротив, расплывчатые подсказки заставляют модель «угадывать» в пространстве вероятностей, что влечет за собой расхождение результатов с ожидаемыми показателями.

5.1.2. Основные принципы разработки промптов

В эпоху ИИ навык написания качественных промптов стал одним из наиболее востребованных. Несмотря на обилие материалов по этой теме в интернете, с учетом специфики вайб-программирования можно выделить следующие основные принципы разработки промптов.

1. **Ясность, конкретность, однозначность**. Используйте лаконичный и четкий язык, ясно излагайте задачу и ожидаемый результат, избегайте использования расплывчатых или общих инструкций, чтобы уменьшить пространство для догадок модели. Например, конкретизируйте «написать алгоритм суммирования» до «реализовать в Python функцию, которая может считывать файл *CSV* и вычислять сумму значений в каждой строке»; или детализируйте «создать аналитическую панель» до «создать аналитическую панель с темной темой, которая должна содержать линейные графики, фильтры, кнопку экспорта и другие функциональные и интерактивные элементы, с использованием HTML+Tailwind CSS, насколько это возможно, а также предоставить полное решение, выходящее за рамки базовых функций».

Точные инструкции и описание целей могут значительно повысить точность генерируемого ИИ кода, предотвращая вывод нерелевантного или не соответствующего требованиям контента.

2. **Структурированные промпты**. Разработайте промпт в виде комбинации модулей с логической иерархией, обычно включающей модули «роль», «контекст», «задача» и «требования к выводу». Такая структурная организация помогает ИИ более точно понимать контекст и отвечать в соответствии с заданным форматом. Например, можно воспользоваться шаблонами промптов, такими как LangGPT, и разделить промпт на модули: **роль** (role), **профиль** (profile), **правила** (rule), **рабочий процесс** (workflow) и т. д. Сначала определитесь с ролью и навыками ИИ, затем предоставьте необходимые исходные данные, далее опишите конкретные задачи, которые необходимо выполнить, и, наконец, укажите формат или стиль вывода. Ниже приведен пример промпта веб-приложения для воспроизведения музыки:

```
【Роль】
Вы – опытный помощник фронтенд-разработчика, специализирующийся на создании UI-
компонентов в стиле минимализма и темной темы. Вы владеете HTML, Tailwind CSS
и JavaScript и способны быстро реализовывать фронтенд-модули с единым стилем
высокой доступности

【Описание】
Я разрабатываю веб-приложение для прослушивания музыки, специально
предназначенное для использования в ночное время. Общий стиль пользовательского
интерфейса – темный фон с неоновыми акцентами, лаконичный макет и мягкие
динамические эффекты на страницах. Целевая аудитория – разработчики, которые
поздно ночью пишут код под музыку. Общий стиль ориентирован на темный режим
Spotify/Apple Music

【Правила】
– Использовать Tailwind CSS для построения интерфейса
– Не использовать фронтенд-фреймворки, такие как React или Vue
– Весь код должен быть встроен непосредственно в HTML-файл и быть исполняемым
– Результат должен быть представлен по модулям, код должен быть полностью
  работоспособным
– Не предоставлять пояснений или описаний, кроме кода
– Цветовая палитра ограничена черным, серым, белым, синим и фиолетовым, стиль
  должен быть единообразным
– Компоненты должны поддерживать адаптивный дизайн и быть совместимыми
  с настольными и мобильными устройствами

【Рабочий процесс】
Разработать и создать для этой страницы верхнюю панель навигации (Header),
содержащую следующее:
1. Место для логотипа (LOGO) слева;
2. Поле поиска в центре;
3. Кнопки управления воспроизведением справа (предыдущий трек, воспроизведение/
   пауза, следующий трек);
4. Общая ширина макета должна адаптироваться к размеру экрана, выравнивание –
   по центру.

Окончательный результат должен быть представлен в виде полного кода HTML +
Tailwind CSS + JS
```

Такой структурированный подход, напоминающий заполнение проектной документации, позволяет четко разграничить все элементы информации в промпте, эффективно снизить степень неоднозначности и тем самым помочь ИИ более точно понять наши намерения и организовать вывод результатов.

3. **Предоставьте небольшое количество примеров**. Включение небольшого количества высококачественных примеров в промпт позволяет значительно повысить точность и согласованность результатов, выдаваемых ИИ. Метод обучения на **небольшом количестве примеров** (few-shot learning) путем демонстрации желаемых пар «вход-выход» помогает ИИ имитировать стиль и формат примеров при построении ответа, что особенно эффективно для задач с определенными требованиями к формату или фиксированному стилю. Например, если требуется, чтобы ИИ генерировал комментарии к коду в определенном стиле, можно сначала предоставить один или два фрагмента кода с уже отформатированными комментариями в качестве примеров. При запросе на вывод таблиц или данных JSON предоставьте заранее шаблон формата, и ИИ сможет более строго следовать этому шаблону.

Исследования подтверждают, что тщательно разработанные примеры могут эффективно повысить точность и согласованность вывода ИИ. Таким образом, если инструкции трудно выразить однозначно за один раз, предоставление «шаблона» для ИИ в качестве ориентира может значительно упростить его работу.

4. **Добавление ограничений**. Четко укажите конкретные требования или ограничения при выполнении задачи: язык программирования (например, «использовать Python 3.9»), библиотеки (например, «реализовать эту функцию с помощью библиотеки pandas, избегая использования других сторонних библиотек»), стиль кода, длину вывода (например, «код не должен превышать 50 строк») и т. д. Эти ограничения устанавливают четкие рамки для работы ИИ, позволяя ему генерировать выходные данные, более соответствующие реальным потребностям, и избегая создания решений за пределами этих рамок или неприменимых решений. Ограничения также можно использовать для форматирования, например указав, что «выводимый код должен содержать аннотации типов» или «результат должен быть представлен в формате Markdown». Типичным примером успешного применения такого подхода является компания Anthropic, которая в настройках модели Claude задала системный промпт с требованием всегда выводить код в формате Markdown, что обеспечивает единообразие стиля представления кода.

В целом установка четких правил, границ и других ограничений для ИИ позволяет значительно повысить контролируемость и надежность генерируемого контента.

5. Итеративная оптимизация и непрерывное накопление опыта. Рассматривайте взаимодействие с ИИ как непрерывный итеративный процесс, а не как попытку получить идеальный результат за один раз. Когда первоначальный результат не соответствует ожиданиям, следует корректировать или дополнять промпты, чтобы помочь ИИ внести изменения и усовершенствовать результат. Например, можно использовать такие уточняющие инструкции, как «оптимизируй следующий код, чтобы сделать его более эффективным» или «переработай функцию из предыдущего абзаца, чтобы повысить ее наглядность». Это позволит превратить процесс разработки в командную дис-

куссию, постепенно приближаясь к идеальному решению. Такая диалоговая итерация повышает качество конечного кода и помогает ИИ поэтапно калибрировать понимание требований. Кроме того, разработчикам настоятельно рекомендуется создать и поддерживать библиотеку качественных промптов, собирая и систематизируя наиболее эффективные из них для последующего многократного использования.

5.1.3. Пример оптимизации промпта

В этом разделе на простом примере показан процесс оптимизации обычного промпта до высококачественного для получения лучшего результата. Предположим, что нам нужно, чтобы ИИ написал функцию факториала. Обычный промпт может выглядеть так:

```
Напиши функцию факториала
```

Этот промпт слишком краток, в нем не указаны язык программирования и детали реализации. ИИ может самостоятельно догадываться о способе реализации, например по умолчанию выдать код на Python, но без обработки исключений. Результат может выглядеть примерно так:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Хотя приведенный выше код и позволяет вычислить факториал, в нем есть несколько недостатков. Во-первых, мы не просили использовать Python, поэтому язык реализации не определен. Во-вторых, отсутствует обработка недопустимых входных данных. В-третьих, нет никаких комментариев с пояснениями. Очевидно, что эта подсказка содержит недостаточно информации, в результате чего ИИ может предложить только общий ответ.

Далее мы напишем улучшенную подсказку, чтобы дать ИИ больше указаний:

```
Роль: вы – опытный инженер, программирующий на языке C
Задача: напишите **функцию на языке C** с именем factorial для вычисления
факториала целых чисел
Требования:
1. Использовать рекурсивный алгоритм;
2. Учитывать исключительные ситуации при вводе, например при вводе
   отрицательного числа функция должна возвращать сообщение об ошибке;
3. Код должен содержать соответствующие комментарии, объясняющие логику;
4. Вывод кода должен соответствовать стандартному формату языка C, длина строки
   не должна превышать 80 символов
```

Благодаря этой тщательно продуманной серии подсказок мы четко определили роль (опытный инженер по языку C), язык и название функции (функция `factorial()` на языке C), конкретные детали реализации (использование рекурсивного алгоритма, необходимость обработки отрицательных входных

данных), а также требования к формату (комментарии и ширина каждой строки). ИИ получил более четкое представление о требованиях и сгенерировал следующий код (отрывок):

```
#include <stdio.h>

/* Вычисление факториала целого числа n.
   Если n отрицательно, возвращается -1, обозначающий ошибку. */
int factorial(int n) {
    if (n < 0) {
        // Обработка ошибок: для отрицательных входных данных
       // возвращается код ошибки
        return -1;
    }
    if (n == 0 || n == 1) {
        return 1;
    }
    // Рекурсивное вычисление факториала n
    return n * factorial(n - 1);
}
```

Из примера видно, что улучшенные подсказки позволили ИИ вывести полную функцию на языке C, включающую рекурсивную логику, обработку ошибок и четкие комментарии, что соответствует всем нашим требованиям. Этот пример иллюстрирует несколько упомянутых ранее принципов: четкое указание языка и ограничений (добавление условий ограничения), включение требований к деталям реализации и комментариям (ясность, конкретность), а также структурированное изложение ролей, введения и правил (структурированные подсказки). При необходимости мы даже можем добавить в промпты примеры ввода и вывода, чтобы еще больше сократить неоднозначность.

5.1.4. Набор практических шаблонов подсказок

В этом разделе собраны некоторые практичные подсказки, которые часто используются в повседневной разработке. Читатели могут напрямую применять эти шаблоны в зависимости от конкретной ситуации.

(1) Режим быстрого написания кода:

```
Мне нужно быстро разработать XX. Требования: (1) минималистичный стиль; (2)
полностью черный фон; (3) кнопки с эффектом подсветки. Необходимо выдать
готовый код HTML+CSS+JS
```

(2) Усиление эстетики интерфейса:

```
Мне нужна боковая панель с эффектом погружения в ночную атмосферу, похожая на
терминал macOS. Разработай ее и выведи CSS-код
```

(3) Дизайн анимации с эффектом движения:

```
Напиши анимацию загрузки с эффектом движения и ритма, реализованную с помощью
чистого CSS, в стиле neon cyberpunk
```

Дополнительная информация

Neon cyberpunk (неоновый киберпанк): стиль фантастического будущего, сочетающий в себе высокие технологии и эстетику антиутопии. Его основными элементами являются неоновые огни, ночные городские пейзажи, кибернетические имплантаты и маргинальные слои общества. Этот стиль отражает мир с высокоразвитыми технологиями, но дегуманизированным обществом.

(4) Режим совместной разработки:

```
В дальнейшем я буду постоянно добавлять требования к UI-компонентам. Ваша
задача как помощника фронтенд-разработчика заключается в сохранении стилевой
целостности и эффективном выполнении задач
```

(5) Дизайн в мрачном стиле:

```
Напиши компонент карточки в мрачном стиле: фон – темно-сине-серый, текст –
мягкий белый, при наведении курсора мыши должен появляться легкий ореол
```

(6) Режим программирования Flow:

```
Сейчас я хочу спокойно писать код в режиме потока сознания. Возвращай только
код, комментарии и объяснения не требуются
```

5.1.5. Настройка подсказок для Cursor

Большие языковые модели не способны запоминать информацию, и после завершения сеанса все предыдущие взаимодействия исчезают. Однако функция **Rules** в Cursor предоставляет механизм сохранения и повторного использования подсказок, который можно рассматривать как решение для сохранения контекста кода, настроек или рабочих процессов. Для одного проекта можно настроить несколько правил, и как только они применяются, их содержание всегда включается в начальную часть контекста модели. Это дает ИИ последовательные инструкции, превращая его из универсального помощника в программировании в «экспертную систему», которая глубоко понимает требования проекта и обладает определенными знаниями в конкретной области. Например, в примере Cursor Rules, показанном на рис. 5-2, различные правила описывают нормативные требования для разных сценариев кодирования, что позволяет обеспечить более точный контроль.

В Cursor доступны два основных типа правил.

- **Правила пользователя** (user rule): глобальные правила пользователя, определяемые в настройках Cursor, применяются ко всем проектам на локальном компьютере пользователя. Недостатком является невозможность отправки в Git и отсутствие возможности совместного использования в команде. Правила пользователя обычно служат для настройки личных предпочтений, таких как язык программирования, стиль или часто используемые фрагменты кода, способы импорта библиотек и т. д. Правила пользователя поддерживают только формат простого текста, при этом файлы *.mdc* (компоненты Markdown) не поддерживаются.

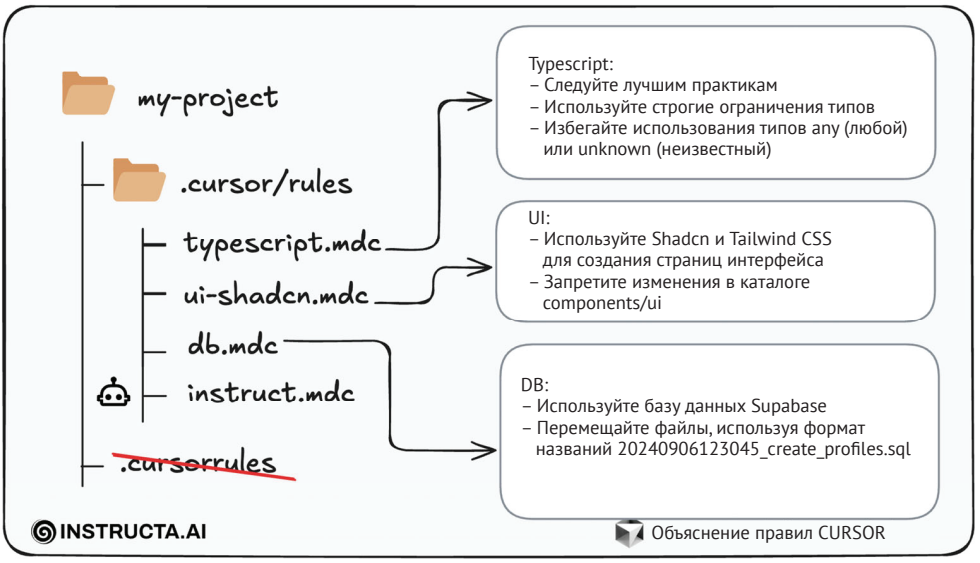


Рисунок 5-2. Пример правил Cursor
(изображение взято с официального сайта INSTRUCTA.AI)

- **Правила проекта** (project rule): правила проекта хранятся в каталоге *.cursor/rules* в корневом каталоге проекта. Каждое правило представляет собой отдельный файл *.mdc*, который обычно отправляется в систему Git и используется для совместной работы членов команды. Правила проекта идеально подходят для кодирования определенных областей знаний, стандартизации рабочих процессов или архитектурных решений, характерных для каждого проекта, а также для сохранения единообразия стиля кода.

1. Глобальные подсказки

В настройках Cursor пользователь может добавить глобальные подсказки, которые действуют как скрытые инструкции, заданные заранее для каждого диалога. В частности, на странице Cursor **Settings** (Установки) в разделе **Rules** (Правила) необходимо ввести текст подсказки в качестве правила системного уровня (см. рис. 5-3). На этом рисунке в качестве примера правил (в красной рамке) приведены следующие указания: «Вы являетесь ведущим архитектором и ведущим программистом. При проведении анализа архитектуры, анализа функциональных модулей, а также при написании кода следуйте приведенным ниже правилам.

1. При анализе задачи, проектировании архитектуры и компоновке модулей кода руководствуйтесь “принципом первоочередности”.
2. При написании кода следуйте фундаментальным принципам программирования: **DRY** (Don’t Repeat Yourself – “Не повторяйся”), **KISS** (Keep It Simple, Stupid – “Делай проще, глупец”), **SOLID** (пять принципов объектно-ориентированного программирования: **S** –Single Responsibility, принцип единственной ответственности; **O** – Open/Closed, принцип открытости/закрытости; **L** – Liskov Substitution, принцип подстановки

Барбары Лисков (объекты программы должны быть заменяемы на экземпляры их подтипов без нарушения работы программы); **I** – Interface Segregation, принцип разделения интерфейса; **D** – Dependency Inversion, принцип инверсии зависимостей) и **YAGNI** (You Ain't Gonna Need It – "Вам это не понадобится", т. е. не стоит тратить время на код, который может и не пригодиться в будущем).

3. Если отдельный метод, функция или файл кода превышает 500 строк, его следует разделить, декомпозировать и разбить на части, при этом в процессе разделения, декомпозиции и разбиения следуйте приведенным выше принципам».

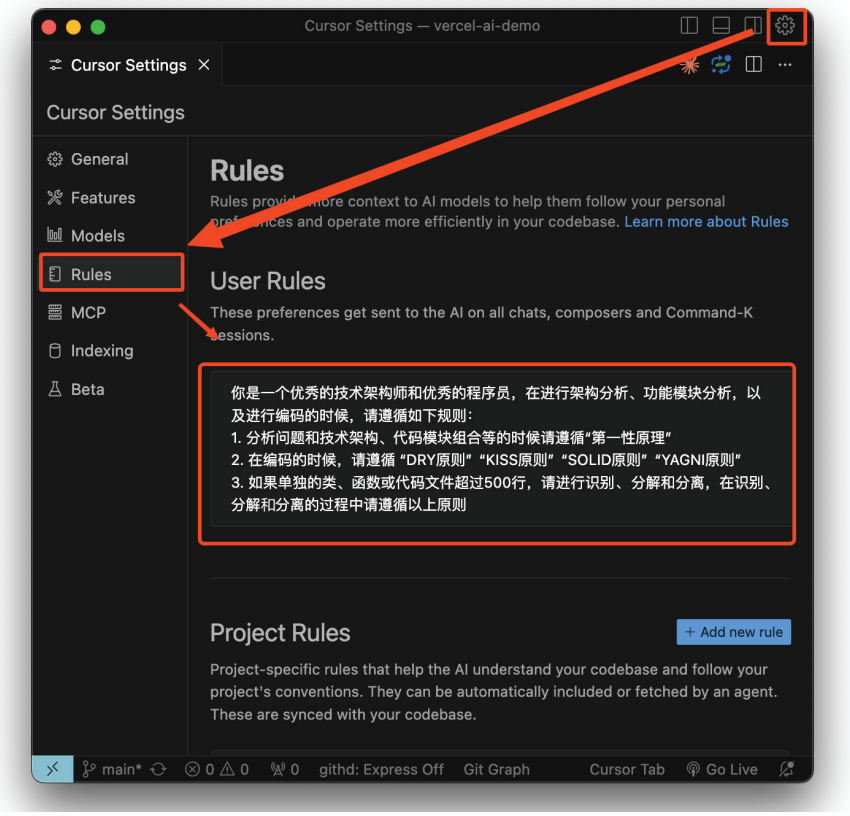


Рисунок 5-3. Настройка глобальных пользовательских подсказок

В официальной документации Cursor также указано, что написание подходящих системных подсказок помогает модели лучше понимать свои задачи и предпочтения пользователя, что позволяет ей давать более точные ответы. Например, фраза «вы – опытный помощник по кодированию, и все выводимые результаты должны соответствовать внутреннему стандарту стиля кода компании» эквивалентна определению для модели долгосрочного образа и кодекса поведения: «играть определенную роль и соблюдать определенные правила». Здесь вы также можете добавить указания по стандартам кода ко-

манды, руководствам по стилю и т. п., чтобы модель отвечала на вопросы в соответствии с едиными правилами вне зависимости от конкретного проекта.

2. Подсказки на уровне проекта

Основной элемент правил проекта представляет собой файл *.mdc*. Это облегченный формат файла, позволяющий одновременно включать в него метаданные (metadata) и содержание правил. Типичная структура файла *.mdc* выглядит следующим образом.

Метаданные представляют собой блок YAML-кода в верхней части файла, заключенный в символы `---`. Метаданные определяют свойства и поведение правил проекта и поддерживают три поля: `description`, `globs` и `alwaysApply`. Их назначение и примеры использования приведены в табл. 5-1.

Таблица 5-1. Назначение и примеры полей метаданных

Поле метаданных	Назначение	Пример
`description`	Предоставляет краткое описание задачи правила. ИИ использует это описание для динамического определения необходимости применения данного правила в конкретной ситуации	`description: Ensures all React components use functional syntax` (использование функционального синтаксиса во всех компонентах React)
`globs`	Указывает один или несколько шаблонов glob (массив строк) для сопоставления путей к файлам. Это правило применяется в случае совпадения	`globs: ["src/components/**/*.tsx ", "src/hooks/*.ts "]`
`alwaysApply`	Булево значение. Если `true`, то определяется правило `Always`, которое всегда включается в контекст модели данного проекта	`alwaysApply: true`

Обратите внимание, что поле `alwaysApply` следует использовать с осторожностью. Оно предназначено в основном для универсальных инструкций по проекту (например, «все бэкенд-интерфейсы в этом проекте используют Python 3.10 и Django REST Framework»). Злоупотребление `alwaysApply: true` может привести к преждевременному переполнению контекстного окна, что повлияет на производительность генерируемого ИИ кода.

Содержание правила следует сразу за метаданными и представляет собой основную часть правила, обычно написанную в формате Markdown. Содержание правила может включать текстовые инструкции, примеры кода и ссылки на другие файлы проекта в качестве контекста с помощью синтаксиса *@filename.ext*.

В зависимости от различных комбинаций упомянутых выше метаданных правила проекта можно разделить на 4 типа, как показано в табл. 5-2. Для каждого типа предусмотрены свои механизмы активации и сценарии использования.

Независимо от способа активации правила, его основной механизм работы остается неизменным: при применении правила его содержание из файла *.mdc* включается в начальную часть промпта, отправляемого в большую языковую модель. Большая языковая модель «читает» эти правила перед выдачей ответа, создавая эффект инициализации.

Таблица 5-2. Классификация типов правил проекта

Тип правила проекта	Ключевые метаданные	Механизм активации	Основные сценарии использования	Пример
Always	`alwaysApply: true`	Всегда включается в контекст модели	Универсальные рекомендации на уровне проекта, такие как специфические спецификации фреймворков или языков	"Весь код на Python должен соответствовать руководству стилей PEP 8"
Auto Attached	`globs: ["pattern"]`, `alwaysApply: false` (или опущено)	Автоматически включается при совпадении ссылки на файл с шаблоном *globs*	Контекстные инструкции по типу файла или местоположению, например правила для каталога React-компонентов	"При обработке файлов в *src/api/* подключайте Zod для проверки всех данных"
Agent Requested	`description: "..."`, `alwaysApply: false`	ИИ на основании диалога и описания правила решает, включать ли его; описание обязательно	Правила, полезные в определенных сценариях, но не всегда необходимые, позволяют ИИ вводить их динамически	"Руководство по рефакторингу классов-компонентов в функциональные компоненты"
Manual	`alwaysApply: false` (или опущено), без `globs`	Включается только при явном упоминании правила в диалоге с помощью *@ruleName* (ruleName – это имя файла)	Вызов конкретных, нечасто используемых правил по требованию	*@generate-express-service-template* (для генерации шаблона сервиса Express)

Таким образом, правила Cursor являются по сути инструментом для создания промптов путем сопоставления конфигурационных файлов. Разработчикам не нужно вручную вводить длинные команды при каждом взаимодействии с ИИ; вместо этого они могут заранее задать эти команды в качестве промптов с помощью структурированных файлов *.mdc*. Это обеспечивает согласованность содержания промптов на уровне проекта.

3. Примеры правил проекта: создание системы правил для форм

Как правильно формулировать правила? Для конкретных задач в отрасли уже существует множество готовых решений (например, каталог правил Cursor: https://cursor.directory/rules), а также автоматизированные методы генерации правил с помощью ИИ (например, функция *Cursor /Generate Cursor Rules*). Эти методы будут подробно рассмотрены в конце данной главы. В этом разделе мы рассмотрим подход к разбиению правил на конкретном примере.

Представим себе следующую ситуацию: нам нужно разработать набор правил проекта для функции «форма настроек». Эта форма включает в себя не-

сколько полей ввода, а также такие функции, как проверка данных, взаимодействие с API (получение и сохранение настроек) и управление состоянием. Таким образом, суть задачи заключается в гарантии того, что ИИ будет следовать спецификациям проекта при генерации или модификации кода, связанного с формой настроек пользователя. Кроме того, функция «форма настроек пользователя» включает в себя технологический стек, состоящий из React-компонентов, библиотек обработки форм (таких как React Hook Form), библиотек проверки данных (таких как Zod, Yup), API-сервисов, средств управления состоянием (таких как Zustand, Redux Toolkit) и т. д. Исходя из этого, нам необходимо сформулировать как минимум следующие правила.

- *user-settings-form-structure.mdc* (тип правила проекта: `Auto Attached`): описывает правила написания кода форм с целью унификации способов их реализации, в частности интеграции логики управления состоянием и валидации.

```
---
description: "Руководство по структуре компонента User Settings Form
и использованию React Hook Form".
globs: ["src/features/user-settings/**/*Form.tsx"]
alwaysApply: false
---

- Для управления состоянием формы и отправки данных необходимо использовать
React Hook Form
- Правила проверки формы должны быть определены с помощью Zod (или Yup)
и сохранены в каталоге src/features/user-settings/schemas/
- Компонент должен содержать понятную обработку состояний загрузки и ошибок
- Кнопка отправки должна быть отключена, если форма недействительна или
находится в процессе обработки

Пример: как использовать функцию useForm, как регистрировать ввод, как
обрабатывать отправку и ошибки
...
```

- *user-settings-api.mdc* (тип правила проекта: `Agent Requested`): определяет шаблоны взаимодействия с API, включая использование библиотек для извлечения данных, инкапсуляцию пользовательских хуков и организацию файлов API-сервисов.

```
---
description: "Правила взаимодействия с конечными точками API настроек
пользователя, включая хуки для извлечения и изменения данных".
alwaysApply: false
---

- Для получения настроек пользователя следует использовать пользовательский
хук useUserSettings() в обертке useQuery
- Для сохранения настроек пользователя следует использовать настраиваемый
хук useUpdateUserSettings() в обертке useMutation
- Эти два хука должны быть определены в файле src/features/user-settings/
hooks/api.ts
- Функции API-сервиса (такие как fetchUserSettings() и saveUserSettings())
должны находиться в файле src/services/userSettingsApi.ts
- Подробное описание структуры запросов/ответов API: см. @/docs/api/user-
settings.md
```

- *zod-validation-patterns.mdc* (тип правила проекта: `Always`): глобальное правило для объявления лучших практик использования Zod в проекте с целью гарантии согласованности и удобства обслуживания логики валидации.

```
---
description: "Лучшие практики использования схем валидации Zod в масштабе
всего проекта".
alwaysApply: true
---

- Все схемы Zod должны экспортировать свои построенные типы TypeScript
(например, export type UserSettings-Schema = z.infer<typeof
userSettingsSchema>;)
- Примеры распространенных правил валидации (email, надежность пароля,
минимальная/максимальная длина)
- Способы организации сложных вложенных схем и т. д.
```

- *form-accessibility.mdc* (тип правила проекта: `Auto Attached`): соответствие требованиям **A11y** (сокращение от Accessibility), обеспечивающее доступность и удобство использования веб-сайта или приложения для всех, в том числе для людей с ограниченными возможностями:

```
---
description: "Обеспечивает соблюдение лучших практик доступности для всех
форм".
globs: ["**/*Form.tsx"]
alwaysApply: false
---

- Все поля ввода в формах должны иметь связанный атрибут <label>
- Сообщения об ошибках валидации необходимо связывать с полями ввода
с помощью атрибута aria-describedby или аналогичного механизма
- Необходимо обеспечить возможность полноценной навигации и управления
формой с помощью клавиатуры
```

Эти типы файлов правил в конечном итоге формируют набор правил разработки, представленный на рис. 5-4.



Рисунок 5-4. Правила разработки формы настроек пользователя

На основании этих правил при запросе ИИ на создание или изменение формы настроек пользователя будет выполняться следующая логика загрузки и применения правил проекта.

1. *zod-validation-patterns.mdc* (`Always`) всегда предоставляет общие рекомендации Zod.

2. Если путь к обрабатываемому файлу соответствует *src/features/user-settings/**/*Form.tsx*, то *user-settings-form-structure.mdc* (`Auto Attached`) активируется автоматически, определяя общую структуру формы, использование React Hook Form и точки интеграции с валидацией Zod. Одновременно файл *form-accessibility.mdc* (`Auto Attached`) также активируется благодаря совпадению имени файла с **/*Form.tsx*, чтобы обеспечить соответствие элементов формы стандартам доступности.
3. Когда содержание диалога касается загрузки данных настроек пользователя из бэкенда или их сохранения в бэкенд, ИИ может активировать файл *user-settings-api.mdc* (`Agent Requested`) в соответствии со своим описанием для получения конкретных инструкций по реализации вызова API, инкапсуляции хуков доступа к данным, а также обработке данных API.

Таким образом, эти разбитые на части и выполняющие свои функции правила образуют единое целое: они активируются на разных уровнях и в разные моменты времени, сообща направляя ИИ на создание функции формы настроек пользователя с четкой структурой, полным набором функций, строгой проверкой данных, стандартизированным взаимодействием с API и удобным доступом. Такой модульный подход к разработке правил повышает точность промптов, а также упрощает поддержку и расширяет возможности набора правил.

4. Автоматическое создание правил с помощью команды /Generate Cursor Rules

Правила Cursor очень полезны, но их написание может быть сложным. К счастью, разработчикам не всегда приходится создавать правила с нуля: в сообществе уже сформировались отличные репозитории правил Cursor, которые предлагают множество готовых файлов правил для различных технологических стеков и сценариев. Их можно использовать как источник вдохновения или применять напрямую.

- *Awesome CursorRules*: это популярная подборка различных файлов *.cursorrules* (в основном в формате *.mdc*), охватывающих такие области, как фронтенд-фреймворки, бэкенд-технологии, мобильная разработка, CSS-стили, управление состоянием и многое другое.
- *Cursor Discovery*: также является репозиторием правил Cursor, но его содержание более полное, чем у *Awesome CursorRules*.

Помимо этого, разработчикам рекомендуется напрямую использовать команду *Cursor /Generate Cursor Rules* для создания конкретных правил, как показано на рис. 5-5.

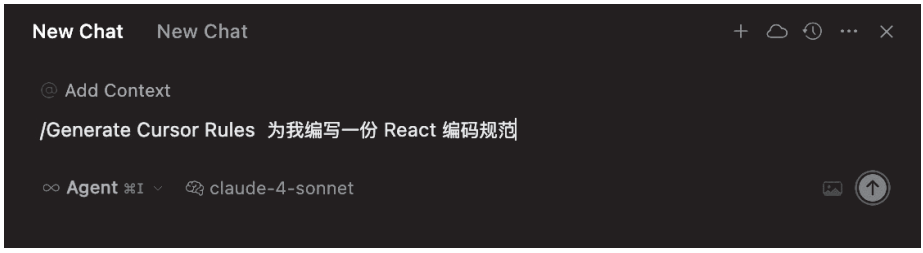


Рисунок 5-5. Автоматическое создание кодовых стандартов React

Эта команда очень гибкая: вы можете ввести задачу вручную (см. рис. 5-5) или сгенерировать новые кодовые стандарты на основе взаимодействий в текущей сессии, как показано на рис. 5-6.



Рисунок 5-6. Обобщение правил кодирования на основе контекста

Эта функция значительно упрощает процесс создания правил и особенно подходит для тех шаблонов кодирования или проектных стандартов, которые формируются постепенно в ходе реальной разработки на основе устоявшихся соглашений. Она дает ИИ возможность самостоятельно определять правила управления, формируя ИИ-поддержку на уровне метаданных.

5.2. Планирование требований

Термин «вайб-программирование» может создать ложное впечатление легкости процесса почти без каких-либо усилий. На самом деле хотя «вайб» подразумевает свободный и плавный процесс разработки с низкой степенью структурирования, который может ослабить значение тщательного предварительного планирования запросов, эффективная работа больших языковых моделей основана на четком, ясном и насыщенном контекстом вводе. Таким образом, «легкость» этапа выполнения (написание кода ИИ) обычно является результатом значительных (хотя и различающихся по форме) усилий, приложенных на предварительном этапе для точного определения содержания. Поэтому для достижения идеального вайб-режима требуется более тщательное предварительное планирование требований: вайб-программирование не означает отказ от структуры, а скорее корректировку и усиление процесса планирования требований с целью их адаптации к взаимодействию с ИИ.

5.2.1. Анализ требований

В области программной инженерии существует множество моделей жизненного цикла разработки ПО (таких как модель гибкой разработки, каскадная модель и т. д.). Несмотря на значительные различия в конкретной реализации, все они имеют одну и ту же отправную точку – анализ требований. Это на самом деле вполне понятно: без четкого определения целей и требований любое программирование подобно строительству высокой башни в песках – бесполезное и тщетное занятие. Задачи анализа требований включают:

- определение правильного вектора развития программного проекта: при отсутствии четких целей все усилия могут привести к результату, противоположному ожидаемому;

- повышение эффективности совместной работы: согласование целей разработчиков, дизайнеров, менеджеров по продукту и других специалистов для минимизации переделок;
- предотвращение расширения объема работ (scope creep): четкие границы являются «брандмауэром», защищающим от необоснованного расширения требований;
- единственный критерий оценки результатов: четкое определение требований предопределяет успех проекта.

Этот принцип применим и в методологии вайб-программирования, где особое внимание уделяется «иммерсивному, эффективному и осознанному» процессу разработки. Предпосылкой для такого вида разработки является четкое и ясное определение направления проекта. Если же большая языковая модель запускается без адекватных указаний, она с высокой вероятностью будет «выдумывать» информацию или генерировать код, который хотя и будет функционально правильным, но не будет соответствовать общему замыслу проекта или его ограничениям. До появления первой строки кода необходимо однозначно определить следующее: какую проблему нужно решить, для кого она решается и какие проблемы не будут решаться. Процесс анализа требований показан на рис. 5-7.

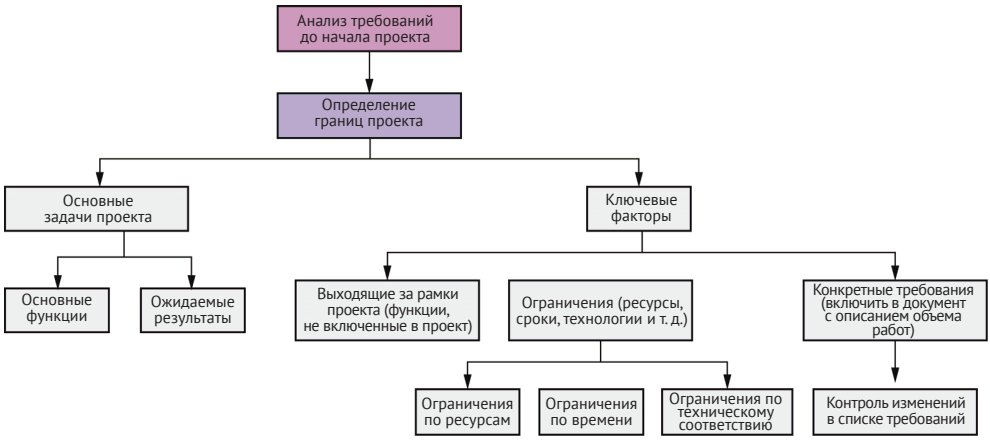


Рисунок 5-7. Процесс анализа требований

На этапе анализа требований необходимо провести углубленное обсуждение с заинтересованными сторонами (заказчиками) для выявления реальных проблем, которые должен решить проект, а также четкого определения ключевых задач проекта и ожидаемых бизнес-целей. В частности, это включает в себя два следующих аспекта.

- **Основные функции**: работа, которую необходимо выполнить в рамках проекта. Это основа существования проекта – решение, разработанное для определенной задачи.
- **Ожидаемые результаты**: критерии оценки достижения целей проекта. Ожидаемые результаты могут представлять собой конкретные бизнес-

показатели (например, повышение конверсии на X %, сокращение числа жалоб на Y %) или стратегические результаты (например, улучшение имиджа бренда, выход на новые рынки).

Помимо этого, с академической точки зрения программной инженерии, необходимо дополнительно выделить другие ключевые элементы, такие как четкое определение не подлежащих выполнению задач (вне цели) и существующих ограничений (ограничения), что не менее важно, чем определение основных задач. Это помогает команде сконцентрировать усилия на основных целях и предотвратить расширение объема работ.

- **Вне цели**: функции или требования, которые явно не входят в проект. Например, проект, ориентированный на мобильное приложение, явно не включает веб-версию или версию для настольных компьютеров.
- **Ограничения**: условия, налагаемые на проект в плане ресурсов, времени, технологий, соответствия нормативным требованиям и так далее. Например, необходимость завершения в рамках определенного бюджета, сдачи до определенной даты, использования определенного набора технологий или соблюдения конкретных законов о конфиденциальности данных.
- **Конкретные действия**: четкое указание «не целей» и ограничений в документе о рамках проекта (scope document) и их строгое соблюдение на протяжении всего жизненного цикла проекта. При появлении новых требований необходимо сверять их со списком «не целей» для недопущения разрастания объема работ.

Например, в проекте по созданию электронной торговой платформы перед началом разработки необходимо как минимум определить следующие основные функции и ключевые нефункциональные требования.

Основные функции:

- просмотр и поиск товаров (поиск по ключевым словам, фильтрация по категориям товаров, фильтрация по брендам и т. д.);
- оформление заказов и оплата (механизмы выдачи купонов, начисления бонусных баллов, проведения рекламных акций и т. д.);
- система пользовательских аккаунтов (регистрация, вход в систему, управление заказами, процедура возврата и обмена товаров и т. д.);
- система гарантийного обслуживания (служба поддержки, система оценок, канал обратной связи и т. д.).

Задачи за рамками проекта (то, что однозначно не будет реализовано в текущей версии):

- не будет поддержки зарубежных пользователей;
- не будет функций подключения продавцов и управления несколькими продавцами в административной панели;
- не будет поддержки синхронизации запасов с офлайн-магазинами.

Четкое определение целей и границ проекта – это отправная точка любого успешного проекта. Как традиционные процессы разработки ПО, так и более гибкая методология вайб-программирования придают большое значение четкому определению задач, целевой аудитории и ограничений еще до начала

кодирования. Такое системное мышление на раннем этапе помогает команде прийти к единому мнению, определить границы проекта и избежать рисков, благодаря чему он не теряется в хаосе разнородных требований и уверенно движется в заданном направлении. Прежде чем писать код, сначала четко определите направление – это отправная точка лучших практик.

5.2.2. Составление документа с требованиями к продукту

Нечеткие входные данные приводят только к нечетким результатам, а ясные и строгие требования обычно дают более точный код, более качественные результаты и сокращают затраты на многократные переговоры в процессе разработки. Поэтому цель этапа планирования требований – составить полный документ с требованиями к продукту (Product Requirement Document, PRD), четко определив его прототип или проблемы, требующие решения, а также обязательные функции и целевую аудиторию. Однако по сравнению с традиционным подходом, вайб-программирование более терпимо к нечетким запросам, и описание требований может быть более лаконичным: приоритет отдается ключевым функциям, а конкретные детали можно постепенно доработать в последующих итерациях. Конкретные различия между ними приведены в табл. 5-3.

Таблица 5-3. Различия между планированием требований при традиционном и вайб-программировании

Характеристики	Традиционное планирование требований	Планирование требований для вайб-программирования
Основной приоритет	Подробное предварительное проектирование, минимизация изменений на поздних этапах	Четкое определение целей и направления итераций, готовность к изменениям
Уровень детализации	Подробная спецификация с охватом всех известных сценариев	Приоритет основных функций и пользовательских историй, доработка деталей в ходе итераций
Результаты	Объемные документы, такие как спецификация требований к ПО (Software Requirements Specification, SRS)	Необременительные документы (например, документ с требованиями к продукту, список пользовательских историй), в основном в форме понятных для ИИ подсказок
Темп итераций	Медленный, каскадный или поэтапный подход	Очень быстрый, непрерывная интеграция и выпуск, быстрое реагирование на обратную связь
Инструменты и технологии	Инструменты CASE, языки моделирования (например, UML)	Вспомогательные инструменты для больших языковых моделей, обработка естественного языка, инженерия промптов, системы контроля версий (для отслеживания промптов и кода)
Роль человека	Аналитики требований, архитекторы, дизайнеры и разработчики в качестве основных создателей	Разработчики требований, консультанты по ИИ, рецензенты кода и системные интеграторы, с уклоном в сторону управления и проверки результатов работы ИИ

Окончание табл. 5.3

Характеристики	Традиционное планирование требований	Планирование требований для вайб-программирования
Уровень допустимой неопределенности	Низкий, стремление устранить все неопределенности	На начальном этапе допускается некоторый уровень неопределенности, требования уточняются постепенно в процессе взаимодействия с ИИ
Стоимость изменений	Довольно высокая, особенно на поздних этапах	Относительно низкая, ИИ может быстро повторно сгенерировать или изменить код

В реальных проектах участникам команды рекомендуется выработать навык выражения требований в структурированной форме: будь то составление заданий на разработку, создание запросов (тикетов) или взаимодействие с ИИ, формулировка запросов должна строиться на основе ключевой структуры: «контекст – цель – подход – ограничения». Такой стандартизированный подход к формулировке значительно повысит согласованность и удобство сопровождения проекта, особенно в условиях командной работы и постоянных обновлений. Разбить требование можно по следующим четырем критериям.

(1) **Контекст требования**: используется для обоснования причин разработки данной функции, то есть для описания реальных проблем, которые она должна решить, или бизнес-требований/потребностей пользователей, которые она должна обеспечить. Описание контекста должно быть лаконичным и содержательным, с акцентом на мотивацию и значимость. Это поможет команде понять причины запуска проекта и позволит ИИ быстрее сориентироваться в целях разработки. Например, для функции «Система автоматического создания README» контекст требования можно сформулировать следующим образом:

```
# Система автоматического создания README

Эта функция предназначена для статического анализа исходного кода указанного
пакета, извлечения его основных функций и способов использования, а также
для вызова большой языковой модели с помощью Vercel AI SDK с целью генерации
README-документа с четкой структурой и понятным содержанием, что должно
повысить наглядность и удобство использования проекта с открытым исходным кодом
```

(2) **Описание функции**: подробное описание ее действий и ожидаемых результатов, включая принцип работы системы, входные и выходные данные, а также соответствующие элементы пользовательского интерфейса. Для повышения наглядности и удобства можно использовать формат пользовательских историй (например, «Как зарегистрированный пользователь, я хочу...»):

```
– Предоставляется команда generate, предназначенная для полноценного
  сканирования имеющегося кода и последующего создания документа README. Эта
  команда поддерживает следующие параметры:

– --project: модуль, для которого необходимо сгенерировать README; после
  получения описания проекта с помощью интерфейса @arch/monorepo-kits адрес
  каталога пакета извлекается из поля project.projectFolder, затем проводится
  анализ кода;
```

- --locale: язык, на котором необходимо сгенерировать README; поддерживаются en (английский) и zh (китайский), по умолчанию используется zh;
- --file: имя выходного файла, по умолчанию README.md; этот файл в конечном итоге будет сохранен в каталоге project.projectFolder

(3) **Инструменты и технологический стек**: описание технологических фреймворков, набора инструментов, а также установленных технических ограничений или рекомендуемых решений, необходимых для реализации данной функции:

- См. документацию https://github.com/jehna/readme-best-practices для создания содержательного README
- Использование Vercel AI SDK для вызова больших языковых моделей; соответствующие интерфейсы описаны в документации .cursor/api/vercel-ai-sdk.mdc
- Написание кода на TypeScript
- Использование пакета commander для реализации взаимодействия с командной строкой; можно добавить подходящие инструменты командной строки, такие как ora
- Использование пакета @coze-arch/rush-logger для вывода логов;
- Использование пакета @coze-arch/monorepo-kits для анализа зависимостей проектов monorepo; соответствующие интерфейсы см. в infra/utils/monorepo-kits/docs/llms.txt
- Запрет на использование ts-node
- Не нужно писать модульные тесты, необходимо сосредоточиться на разработке функциональности
- Не устанавливать зависимости, а записывать их в файл package.json, чтобы пользователи могли заняться ими самостоятельно
- Предпочтительно использовать стиль кодирования FP

(4) **Основной алгоритм** (опционально): если данное требование касается ключевого алгоритма или сложной логики, рекомендуется описать эту часть отдельно для облегчения технической реализации и совместной работы. Содержание может включать описание алгоритма и концепцию проектирования, определение входов/выходов, ожидаемую производительность и граничные условия и т. д. При необходимости можно использовать Mermaid для визуального описания сложной структуры процессов. Однако в большинстве случаев эту часть может выполнить ИИ:

1. После запуска проекта с параметром --project анализируется путь к проекту, для которого необходимо сгенерировать README.
2. Проводится анализ исходного кода проекта с помощью команды git ls-files (из-за возможного наличия в коде постороннего контента, такого как node_modules, требуется фильтрация).
3. Исходный код просматривается по частям и отправляется в большую языковую модель, которая обобщает функции и принципы реализации каждого файла и сохраняет результаты в кеше в виде объекта map.
4. По результатам анализа обобщаются все сгенерированные языковой моделью резюме файлов, после чего модель вновь вызывается для объединения и анализа функций, назначения, характеристик и т. д. данного пакета, а итоговые данные сводятся в файл README.

Благодаря стандартному описанию по этим четырем параметрам разработчики могут эффективнее общаться с командой и обеспечить хорошую базу для

взаимодействия с ИИ-агентом: четкое определение требований является отправной точкой для технической реализации и залогом эффективного взаимодействия человека и ИИ.

Обратите внимание, что при создании документации с требованиями к продукту следует учитывать реальные условия и подходить к делу максимально гибко. Например, если какая-то функция не связана со сложной логикой, описание алгоритма можно упростить, а если технический стек уже определен, можно сократить раздел с описанием инструментов. В любом случае, независимо от степени упрощения, «ясность» остается самым важным принципом.

Кроме того, вайб-программирование не означает «передачу всего на откуп ИИ», а заключается в том, чтобы на основе понятных и четких формулировок требований, с помощью способностей ИИ к генерации, пониманию и логическому выводу, повысить креативность и производительность труда программистов. Чем лучше написаны требования, тем эффективнее будет помощь ИИ.

5.2.3. Выбор ИИ-ориентированного технологического стека

В разделе 5.2.1 говорилось о необходимости определения ряда технологических стеков и технических ограничений для ИИ, и это очень важный момент. Технологический стек – это совокупность технологий, фреймворков, библиотек и инструментов, используемых при создании программного продукта. Выбор технологического стека влияет на эффективность разработки и производительность продукта. При выборе технологического стека необходимо учитывать его совместимость с существующими системами и степень поддержки ИИ-агента.

Если с самого начала не определить используемый стек технологий, ИИ может принимать некорректные или неприемлемые решения, создавая так называемый **технический долг** (technical debt), что со временем может привести к сложностям в обслуживании системы. Например, ИИ может выбрать для создания веб-приложения нативный стек технологий HTML+CSS+JS, отходя от современных систем фронтенд-разработки (таких как Webpack, React и т. д.). Это усложнит чтение кода, а также возможности его масштабирования и поддержки. Хотя это может дать удовлетворительный результат здесь и сейчас, в будущем система неизбежно будет испытывать трудности.

Поэтому на этапе выбора технологий следует тщательно фильтровать варианты вручную, выбирая сочетание технологий с максимальным соответствием навыкам команды, с возможностью масштабирования и с высокой степенью совместимости с ИИ. Одной из наиболее сложных задач является поиск подходящего сочетания технологий для работы с ИИ. В настоящее время в сообществе об этом говорится не так много, однако с учетом характеристик больших языковых моделей можно обозначить некоторые ориентировочные критерии. Возьмем в качестве примера фронтенд: Tailwind CSS превосходит нативный CSS, Less и т. д.; TypeScript превосходит JavaScript, CoffeeScript и т. д.; фреймворки MVVM (model-view-viewmodel), такие как React или Vue, превосходят нативный JS+HTML+CSS; GraphQL превосходит RESTful API и т. д.

Почему между технологическими стеками могут возникать такие различия в степени совместимости с ИИ? Ключевая причина заключается в том, что на базовом уровне большие языковые модели генерируют контент на основе вероятностного вывода, и качество результатов зависит от множества факторов,

таких как калибровка модели, обучающий набор данных, полнота контекста, промпты и т. д. Если говорить исключительно о сценарии программирования с поддержкой ИИ, то чем полнее понимание технологического стека со стороны большой языковой модели, тем лучше результат; чем четче структура кода, тем меньше информационного шума при выводе, тем лучше результат; чем меньше специфических сценариев в бизнес-реализации и чем больше общих правил, тем менее объемную информацию требуется обработать языковой модели, и, как правило, результаты будут лучше, и т. д.

С учетом этих факторов мы сформулировали в данном разделе несколько простых правил оценки того, насколько тот или иной технологический стек подходит для сценариев вайб-программирования.

- **Активность сообщества**: чем активнее сообщество и чем больше пользователей, тем более насыщенными будут обсуждения и техническая документация по данной технологии, а значит, и более полными информационные ресурсы, доступные для индексации при обучении или работе больших языковых моделей, что, в свою очередь, повышает вероятность получения качественных результатов. Например, предположим, что вы столкнулись с какой-то специфической и сложной проблемой. Если кто-то уже сталкивался с ней и опубликовал в интернете анализ ее причин и способов решения, то при работе большая языковая модель сможет найти эту статью и на ее основе выдаст окончательное решение. Если же такой информации в сети нет, то с учетом ограниченной способности больших языковых моделей к сложным логическим выводам вероятность получения эффективного решения будет низкой.
- **Структурированность**: чем выше уровень структурированности и модульности самого технологического стека, тем четче его информационная выраженность и тем легче он воспринимается большими языковыми моделями, что облегчает генерацию кода. Хорошим примером является атомарный CSS (например, Tailwind CSS). Нативный CSS выражает визуальный эффект элементов страницы с помощью конкретных пар элементов «ключ–значение», в то время как атомарный CSS выражает набор правил стилей определенного типа с помощью имен атомарных классов, что делает информацию более сфокусированной и легкой для восприятия ИИ. Кроме того, под влиянием правил наложения при использовании нативного CSS стили элементов могут зависеть от стилевых правил глобального уровня, уровня родительских элементов, различных селекторов и т. д. Для больших языковых моделей это означает, что конкретная информация разбросана по разным частям проекта, и для вывода правильного результата требуется обработка и понимание большего количества контекста. В то же время в рамках атомарного CSS большая часть информации о стилях сосредоточена в списке соответствующих элементам классов, что обеспечивает высокую степень фокусировки информации, снижает затраты на вывод и делает результаты более надежными.
- **Универсальные правила предпочтительнее специализированного дизайна**: чем более универсальны спецификации технологического стека, тем легче их понять большим языковым моделям и тем больше они подходят для сценариев вайб-программирования. Например, GraphQL

явно превосходит RESTful, поскольку GraphQL предоставляет набор универсальных языковых правил для описания сущностей и связей между ними, достаточный для выражения подавляющего большинства стандартных бизнес-операций, таких как сохранение, извлечение, удаление и изменение данных. Таким образом, большой языковой модели достаточно понять этот набор универсальных языковых правил в сочетании с сущностями и взаимосвязями между ними в конкретной предметной области, чтобы гибко программировать различную логику работы с данными на основе GraphQL. Напротив, спецификация RESTful в большей степени сосредоточена на сущностях. Помимо нескольких базовых операций с данными, при работе со сложными структурами данных из соображений практичности и производительности обычно приходится прибегать к специализированному проектированию и разработке. Однако для больших языковых моделей такая специализация может оказаться слишком узконаправленной, что приводит к увеличению сложности контекста и количества шума, а значит, затрудняет вывод правильного ответа.

- **Автоматизированный контроль качества**: чем мощнее инструменты контроля качества в технологическом стеке, тем раньше и более основательно можно выявлять проблемы с качеством. В результате снижаются побочные риски при использовании больших языковых моделей, что делает их более пригодными для сценариев вайб-программирования. Например, TypeScript обладает более мощной системой типизации по сравнению с JavaScript и позволяет выявлять многие проблемы несоответствия типов еще на этапе статического анализа кода. Таким образом, даже если большая языковая модель сгенерирует код, не соответствующий типам, встроенный в Cursor инструмент linter сможет быстро обнаружить проблему до запуска кода, что значительно снизит издержки на исправление ошибок.

Безусловно, по мере итеративного развития больших языковых моделей специфические правила будут неизбежно дополняться, удаляться или модифицироваться. Это не имеет большого значения. Важно то, что при выборе технологий разработчики должны уделять больше внимания адаптируемости технологического стека к ИИ, а то и вовсе руководствоваться этим принципом в первую очередь, по возможности выбирая технологии и инструменты, оптимизированные для работы с ИИ. Это позволит большим языковым моделям более эффективно и точно содействовать выполнению различных задач разработки, полноценно интегрироваться в повседневную работу и способствовать повышению общей эффективности как отдельных сотрудников, так и всей команды. Ниже представлено несколько наиболее популярных в настоящее время комбинаций технологических стеков, которые читатели могут выбрать в соответствии со своими потребностями.

- **Разработка веб-фронтенда**: React (или Vue) + TypeScript + Tailwind CSS. Преимущества: четкая структура, унифицированный код, простота понимания ИИ.
- **Разработка мобильных приложений**: React Native (или Flutter) + TypeScript (или Dart). Преимущества: кросс-платформенная разработка, структурированность, облегчающая генерацию кода ИИ.

- **Разработка бэкенд-сервисов**: Node.js (или Python) + TypeScript (или Python) + GraphQL. Преимущества: универсальные правила, легкость вывода и исправления ошибок с помощью ИИ.
- **Анализ и визуализация данных**: Python (pandas) + ECharts (или Plotly) + Jupyter Notebook. Преимущества: четкая структура данных, легкость анализа и представления с помощью ИИ.
- **ИИ-агенты и автоматизация**: Python + LangChain (или AutoGPT) + API OpenAI (или Hugging Face). Преимущества: специально разработано для ИИ, легко интегрируется и масштабируется, дружественно для ИИ.
- **Выбор системы хранения данных**: Supabase, PostgreSQL или MongoDB. Преимущества: четкая структура данных, обширная поддержка сообщества, удобство для работы и восприятия ИИ.

5.2.4. Использование ИИ для генерации технических требований

Изложенное выше может показаться сложным и не совсем соответствующим духу эпохи ИИ. Более эффективная стратегия заключается в делегировании задачи составления технического задания искусственному интеллекту. Давайте обратимся к примеру:

```
Напиши для меня: техническое задание проекта "Интерактивная стена с волшебными
частицами". Оно должно содержать следующее:
1. Введение: краткое описание контекста и целей проекта, то есть "что нужно
   сделать";
2. Описание функциональности: перечисление функций и интерфейсов, которые будет
   предоставлять система (описание внешне видимых действий с точки зрения
   пользователя);
3. Ограничения: четкое определение технологического стека, требований
   к производительности, стандартов кодирования и других ограничивающих
   условий.
```

Использование GPT-4o даст следующий результат:

```
# Техническое задание "Интерактивная стена с волшебными частицами"”

## 1. Введение

**Предпосылки проекта**: С развитием технологий мультимедийных презентаций
интерактивные визуальные эффекты стали важным элементом технологических
выставочных залов, выставок цифрового искусства и образовательных сценариев.
"Интерактивная стена с волшебными частицами" предназначена для создания с помощью
веб-технологий визуально впечатляющей и интерактивной анимационной стены
с частицами, которая обеспечит пользователям иммерсивный интерактивный опыт.

**Цели проекта**: Создать интерактивную визуальную систему, работающую на веб-
платформе, в которой пользователь может взаимодействовать с динамикой частиц
на экране с помощью мыши или сенсорного управления, реализуя такие волшебные
эффекты, как перемещение частиц кончиками пальцев, следование за частицами,
слияние и рассеивание.

---
```

```
## 2. Описание функций

Ниже приведены основные функции "Интерактивной стены с волшебными частицами",
описанные с перспективы пользователя:
Дополнительные ресурсы: проверочный код 252139
— **Анимация движения частиц по умолчанию**
  - При запуске системы автоматически воспроизводится плавный эффект движения
    частиц, создающий динамический фон с футуристическим эффектом.

— **Эффекты взаимодействия с пользователем**
  - Когда курсор мыши или точка касания приближаются к частицам, частицы
    начинают уклоняться или притягиваться.
  - Взаимодействие путем перетаскивания может вызвать изменение траектории
    частиц, имитируя визуальный эффект "волшебного перетаскивания".
  - Взаимодействие с частицами приводит к динамическим изменениям, таким как
    появление текста, графических элементов или ряби.

— **Переключение между различными темами**
  - Предоставляется несколько наборов визуальных стилей частиц (например,
    звездное небо, электрический ток, чернильные пятна, светлячки и т. д.),
    которые могут переключаться пользователем.

— **Механизм адаптации производительности**
  - Система динамически регулирует количество частиц и детали рендеринга
    в зависимости от мощности устройства, обеспечивая плавную работу.

— **Два режима: полноэкранный и встраиваемый**
  - Можно переключиться в полноэкранный иммерсивный режим или встроить
    в страницу в качестве динамического фона.

---

## 3. Ограничения

### Ограничения по технологическому стеку

— **Фронтенд-фреймворк**: HTML5 + CSS3 + JavaScript, для рендеринга частиц
  в основном используется Canvas или WebGL.
— **Библиотеки взаимодействия**: для более разнообразного управления анимацией
  можно использовать GSAP, three.js или pixi.js.
— **Требования к фреймворку**: предпочтительна нативная реализация; если
  требуется фреймворк, рекомендуется использовать React.

### Требования к производительности

— Стабильная работа с частотой 60 FPS на устройствах среднего и высокого уровня
  (i5/8 ГБ ОЗУ и выше).
— На мобильных устройствах количество частиц должно регулироваться
  автоматически и обеспечивать не менее 30 FPS.
— Время загрузки страницы не должно превышать 3 секунд.

### Стандарты кодирования
```

```
– Соблюдение стандартов ESLint, форматирование кода с помощью Prettier.
– Модульная структура проекта, обеспечивающая возможность удобного
  обслуживания.
– Все графические элементы и анимация частиц должны иметь адаптивную верстку,
  совместимую с распространенными разрешениями.

---

**Примечание**: В дальнейшем возможны такие расширения функционала, как
распознавание жестов, звуковое взаимодействие, запись поведения пользователей
и т. д.
```

На первый взгляд, приведенный выше документ с техническим заданием достаточно полно описывает требования к «Интерактивной стене с волшебными частицами»: в нем есть четко сформулированные предпосылки и цели, конкретное описание функционала, а также требования к технологическому стеку, производительности и стандартам кодирования. Теоретически его можно использовать в реальном проекте для генерации кода с помощью ИИ.

Однако на самом деле это всего лишь демонстрационная версия (Demo). Например, в разделе «Предпосылки проекта», несмотря на подробное описание, не указаны целевые пользователи и значение проекта. В разделе «Описание функций» представлены, казалось бы, достаточно полные сведения о предлагаемых функциях, но до реальных ожиданий еще далеко: не определен начальный рисунок страницы, цвета изображений, способы динамической настройки и т. д.

На самом деле невозможно заставить ИИ создать полноценный и соответствующий требованиям контент с помощью единственной фразы. К счастью, мы можем доработать результаты вручную. Например, в приведенном выше случае можно добавить более подробное описание контекста проекта, уточнить целевую аудиторию и перечислить проблемы, которые будут для нее решены, и т. д. То же самое касается и других частей технического задания. Такой подход гораздо эффективнее, чем написание текста с нуля. Однако главное заключается в том, что даже в эпоху ИИ по-прежнему приходится выполнять много «черновой работы».

5.3. Рецензирование и оптимизация кода

Основная идея вайб-программирования заключается в предоставлении разработчикам возможности взаимодействовать с большими языковыми моделями с помощью подсказок на естественном языке, при этом ИИ-агент быстро генерирует предварительный код. Хотя это значительно повышает эффективность кодирования и позволяет разработчикам уделять больше внимания формулировке требований и определению проблем, вайб-программирование не является столь строгим подходом. Такой подход к программированию, в большей степени опирающийся на интуицию, по сути отклоняется от строгих инженерных практик. Код, сгенерированный ИИ, не всегда отличается одинаковым стилем, структурой и надежностью и без тщательного рецензирования и оптимизации неизбежно приведет к накоплению технического долга и появлению потенциальных ошибок.

5.3.1. Ограничения ИИ

Обычно ИИ оптимизируется под конкретную задачу или промпт (т. е. локальная оптимизация), но ему не хватает понимания более широкого архитектурного контекста или долгосрочных последствий (т. е. глобального контекста). ИИ не воспринимает детали программных систем, а просто генерирует код на основе статистических моделей, без учета системной архитектуры, внутренних соглашений или существующих функций. Это приводит к тому, что, несмотря на отличные показатели ИИ-инструментов программирования в плане синтаксиса и семантики, их контекстная компетентность оставляет желать лучшего.

Обученные большие языковые модели могут генерировать, казалось бы, корректный код на основе промптов и непосредственного контекста, но модели не могут «продумать» общий дизайн, удобство обслуживания и взаимодействие между модулями. Большие языковые модели хорошо справляются с сопоставлением шаблонов и предсказанием токенов в ограниченном окне, но не обладают способностью к рассуждению на уровне всей системы. В результате сгенерированный код может нормально работать в изолированной среде, но при интеграции на системном уровне привести к сбоям, проблемам с производительностью, архитектурным конфликтам и нарушениям стандартов кодирования. Такие архитектурные и системные проблемы по-прежнему требуют контроля со стороны человека.

Массивы данных для обучения больших языковых моделей обширны и разнородны, обычно включают огромное количество открытого кода, в котором нередко встречаются устаревшие практики, уязвимости, пристрастность или образцы низкого качества. Большие языковые модели являются мощными механизмами сопоставления шаблонов, и если в их обучающих данных присутствуют недостатки, ИИ будет учиться и воспроизводить эти недостатки. Это означает, что без тщательного руководства и проверки код, сгенерированный ИИ, с большей вероятностью будет содержать подобные проблемы, что представляет собой серьезный риск для предприятий и может привести к уязвимостям в безопасности, диспропорциональным результатам или низкой производительности.

5.3.2. Распространенные дефекты качества

В связи с упомянутыми выше ограничениями ИИ нередко генерирует фрагменты кода со значительными дефектами качества, которые в целом можно разделить на следующие типы.

(1) **Спагетти-код** (spaghetti code) – это беспорядочный код, в котором переплетаются различные функции и который трудно читать и поддерживать. Например:

```
<canvas id="cv"></canvas>
<script>
// ---- Глобальные переменные и магические числа ----
var data   = [];                        // Без какого-либо инкапсулирования
var colors = ['#f66', '#6f6', '#66f'];  // «Магические константы»
var cvs    = document.getElementById('cv');
var ctx    = cvs.getContext('2d');
```

```
cvs.width  = 800;
cvs.height = 400;

// ---- Основной поток: вся логика сконцентрирована в одном месте ----
function loop () {
  // (1) Удаленное получение данных
  var xhr = new XMLHttpRequest();
  xhr.open('GET', '/api/values?ts=' + Date.now());
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        // (2) Предварительная обработка данных
        var raw = JSON.parse(xhr.responseText);
        for (var i = 0; i < raw.length; i++) {
          data.push(+raw[i]);          // Неявное преобразование типов
          if (data.length > 30) data.shift();
        }

        // (3) Сортировка пузырьком (O(n²)), два вложенных цикла
        for (var i = 0; i < data.length; i++) {
          for (var j = 0; j < data.length; j++) {
            if (data[i] > data[j]) {
              var t = data[i]; data[i] = data[j]; data[j] = t;
            }
          }
        }

        // (4) Рисование гистограммы
        ctx.clearRect(0, 0, cvs.width, cvs.height);
        for (var k = 0; k < data.length; k++) {
          ctx.fillStyle = colors[k  % colors.length];
          ctx.fillRect(k * 25, cvs.height - data[k] * 3, 20, data[k] * 3);

          // Вставка 'пасхального яйца'
          if (data[k] > 80) {
            ctx.strokeStyle = '#000';
            ctx.strokeRect(k * 25, cvs.height - data[k] * 3, 20, data[k] * 3);
          }
        }

        // (5) Рекурсивный вызов (продолжительность определяется случайным числом)
        setTimeout(loop, Math.random() > 0.7 ? 500 : 300);
      } else {
        console.log('ошибка сервера');
        setTimeout(loop, 1000);     // логика повторной попытки также сгруппирована
      }
    }
  };
  xhr.send();
}
loop();   // запуск
</script>
```

В приведенном выше коде функция `loop()` одновременно выполняет задачи удаленного извлечения данных, их предварительной обработки, сортировки методом «пузырьков», построения гистограммы и рекурсивных вызовов, при этом никакого разделения обязанностей не происходит; переменные `data`, `ctx` и другие элементы изменяемого состояния находятся в глобальной области доступности, и любое выражение может их изменить, что приводит к тесному переплетению логики и данных. Поток управления напоминает спагетти: асинхронные обратные вызовы `xhr`, двойной цикл сортировки пузырьком и рекурсивные вызовы `setTimeout` со случайными интервалами накладываются друг на друга, делая путь выполнения извилистым и трудным для отслеживания. Разбросанные повсюду «магические числа» (такие как 25, 3, 30) и побочные эффекты (`ctx.fillRect`, `console.log` и т. д.) приводят к тому, что любое незначительное изменение может повлиять на весь процесс. В приведенном выше коде отсутствует структура, обеспечивающая абстракцию, модульность и возможность повторного использования. Если разработчик хочет найти ошибку или изменить какую-либо функцию, ему приходится распутывать этот клубок с самого начала до конца. Именно такая сложная, запутанная, непонятная и сложная в обслуживании форма и послужила причиной появления термина «спагетти-код».

ИИ склонен генерировать спагетти-код в силу своей способности к локальному сопоставлению шаблонов при отсутствии глобального восприятия. Он может найти фрагмент кода, который, казалось бы, решает локальную проблему, и вставить его, но не будет понимать, как он вписывается в общую структуру. Это приводит к хаосу во взаимосвязях и запутанности потока управления, как если бы вы пытались отследить одну отдельную нить в тарелке спагетти. Поэтому код, сгенерированный ИИ, обычно требует значительных временных затрат на проверку. Иногда такие проверки заканчиваются безрезультатно, поскольку конкретная реализация не была продуманным выбором, а просто сгенерирована случайно, и пользователь принял это предложение.

(2) **Накопление технического долга**. Технический долг – это скрытые затраты на будущую доработку, возникающие в результате выбора простого, но ограниченного решения (часто называемого «временным») вместо долгосрочного решения, которое требует больше времени и средств, но дает лучший результат. Это похоже на использование упрощений при строительстве дома (например, установка дешевых, но протекающих труб): на первых порах это может показаться быстрее, но впоследствии потребует дорогостоящего ремонта.

ИИ ускоряет накопление технического долга, поскольку он уделяет приоритетное внимание быстрой генерации, зачастую генерируя избыточный или повторяющийся код, без долгосрочного учета соображений технической поддержки. ИИ еще не научился долгосрочному планированию, и цена такой быстрой первоначальной реализации может постепенно становиться очевидной на более поздних этапах развертывания, эксплуатации и доработки.

(3) **Повторение кода, потеря кода и уязвимость кода**. Исследование платформы GitClear, специализирующейся на программной инженерии, показало, что при использовании инструментов программирования с поддержкой ИИ количество повторяющихся блоков кода увеличивается в 8 раз. Повторение кода, как следует из названия, означает появление одинаковых или похожих фрагментов кода в нескольких местах программы. Например, в приложении для электронной коммерции необходимо проверять формат введенного поль-

зователем адреса электронной почты в нескольких местах: при регистрации, изменении личных данных, в панели администратора и т. д. ИИ может генерировать логику «проверки формата адреса электронной почты» отдельно для каждого места, что приводит к повторению кода.

Причина заключается в том, что при генерации кода ИИ, не имея полного представления о существующих функциях всего проекта или стремясь быстро отреагировать на конкретный запрос, может заново генерировать логический код вместо повторного использования уже имеющихся в проекте блоков кода, способных выполнять аналогичные задачи. Например, если разработчик неоднократно запрашивает у ИИ проверку введенных пользователем данных в разных модулях, ИИ может каждый раз генерировать новый, слегка отличающийся код проверки вместо подсказки разработчику использовать или создать универсальную функцию проверки. Дублирование кода – главный враг программной инженерии, оно приводит к следующим последствиям.

- **Рост издержек на техническое обслуживание**: когда правила проверки (например, разрешение новых доменов верхнего уровня) требуют обновления, разработчики вынуждены находить и синхронно изменять весь повторяющийся код проверки; пропуск хотя бы одного места может привести к несогласованному поведению системы.
- **Повышенный риск возникновения ошибок**: если в исходном коде присутствует скрытая ошибка, она будет присутствовать во всех копиях. При исправлении, если не удастся исправить все копии, ошибка попрежнему сохранится в некоторых частях системы.
- **Избыточность кода**: ненужное дублирование кода делает всю программную систему более громоздкой, сложной для понимания и управления.
- **Сложность понимания**: когда другие разработчики читают код, они могут быть сбиты с толку множеством незначительно отличающихся реализаций одной и той же функции, что затрудняет понимание.

Термин **текучесть кода** (code churn) относится к ситуации, когда код вскоре после написания подвергается значительным изменениям, замене или удалению. Это похоже на то, как строительная бригада в спешке возводит леса, но вскоре обнаруживает, что они неустойчивы или расположены неверно, и вынуждена немедленно их демонтировать и возводить заново. Высокий уровень текучести обычно указывает на низкое качество исходного кода или на неточное понимание требований.

Если качество генерируемого ИИ кода низкое, он плохо интегрируется, не соответствует реальным требованиям или содержит сложно исправляемые ошибки. Разработчикам, скорее всего, придется его переписать или отказаться от него. ИИ ставит во главу угла быстрое получение результата, и порой его вывод может оказаться лишь черновиком, похожим на рабочий вариант. Но при столкновении со сложностями реальных сценариев и проблемами интеграции может потребоваться значительное количество изменений, что резко повышает уровень текучести кода и, в свою очередь, вызывает множество технических проблем.

- **Растрата ресурсов разработки**: написание и рецензирование кода, который вскоре будет отвергнут, является пустой тратой времени и сил разработчиков.

- **Задержки в реализации проекта**: частые переписывания и исправления сбивают график разработки, приводя к задержкам реализации проекта.
- **Неблагоприятные сигналы о качестве**: высокий уровень отбраковки кода обычно свидетельствует о проблемах с качеством исходного кода или нечеткости требований, что является тревожным сигналом о плохом состоянии проекта.
- **Снижение морального духа команды**: если результаты работы разработчиков (или с помощью ИИ) постоянно отвергаются и переделываются, это может подорвать мотивацию членов команды.

Уязвимость кода означает, что при внесении изменений в одну часть ПО могут возникнуть сбои или непредвиденное поведение в других, казалось бы, не связанных частях системы. Это похоже на точную, но хрупкую стеклянную скульптуру: легкое прикосновение к одной точке может привести к появлению трещин или разрушению всей конструкции.

При генерации кода с помощью ИИ, из-за отсутствия полного понимания архитектуры всей системы и сложных взаимозависимостей между модулями, могут создаваться скрытые, нестабильные взаимосвязи или зависимости. ИИ может изменять общие компоненты для решения локальных проблем или вводить модели, несовместимые с другими частями системы, что приводит к снижению общей стабильности всего проекта. В отчете DORA от Google за 2024 год отмечается, что использование ИИ привело к снижению стабильности релизов на 7,2 %, что косвенно подтверждает повышение потенциальной уязвимости, которая, в свою очередь, вызывает следующие проблемы.

- **Непредсказуемые цепные реакции**: небольшие изменения могут вызвать широкомасштабные и труднопредсказуемые сбои в системе, что значительно затрудняет отладку.
- **Резкий рост затрат на обслуживание**: разработчикам приходится тратить много времени на понимание и устранение этих цепных реакций вместо работы над новыми функциями.
- **Усложнение тестирования**: сложно гарантировать, что тестирование охватывает все части, которые могут быть затронуты даже незначительными изменениями.
- **Снижение надежности системы**: частые непредвиденные сбои серьезно влияют на удобство эксплуатации и доверие к системе.

Эти три проблемы – дублирование кода, его утрата и уязвимость, как правило, не существуют изолированно; они тесно связаны между собой и совместно влияют на качество и поддержку ПО. Например, если повторяющийся код (дубликаты), сгенерированный ИИ, имеет низкое качество или содержит дефекты, он с большей вероятностью будет впоследствии подвергнут значительным изменениям или удален (текучесть). Когда необходимо исправить дефектный блок повторяющегося кода, отсутствие синхронного обновления всех копий приводит к несогласованному поведению системы, в результате чего изменения в одной части кода могут неожиданно повлиять на другие функции, использующие эти копии (уязвимость). Этот порочный круг, усугубляемый использованием ИИ, в конечном итоге приводит к созданию системы, сложной в обслуживании, полной скрытых угроз и тормозящей дальнейшее развитие.

(4) **Уязвимости безопасности и несоответствие нормативам**. Если ИИ обучается на небезопасных примерах или в подсказках не указаны четкие требования безопасности, он может генерировать код с известными уязвимостями. Сам по себе ИИ не разбирается в юридических или отраслевых нормах соответствия (таких как правила обращения с медицинскими данными или правила проведения финансовых операций), поэтому его результаты могут не соответствовать этим важным требованиям. Исследования Университета Стэнфорда показывают, что инструменты программирования на базе ИИ могут генерировать небезопасный код. ИИ также может предлагать библиотеки криптографических операций без учета конкретных требований соответствия.

Безопасность и соблюдение нормативных требований в значительной степени зависят от конкретного контекста и требуют понимания конкретных нормативных актов и моделей угроз, а это, как правило, не входит в сферу компетенции ИИ. Использование ИИ для написания критически важного кода без экспертной проверки может привести к серьезным рискам безопасности.

(5) **Трудно обнаруживаемые дефекты и проблемы с производительностью**. Цель ИИ обычно заключается в генерации работоспособного кода с учетом заданных подсказок, а не обязательно кода с оптимальной производительностью. Оптимизация производительности обычно требует более глубокого понимания алгоритмов, структур данных и системной архитектуры. ИИ может выбрать простой и распространенный алгоритм, пригодный для использования, но очень медленный при обработке больших объемов данных; либо сгенерировать код с ненужными шагами, потребляющий больше ресурсов сервера и снижающий скорость работы приложения. Также он может игнорировать исключительные ситуации (крайние случаи), что может привести к сбою программы или непредсказуемому поведению с появлением скрытых ошибок, которые разработчикам сложнее обнаружить и исправить.

5.3.3. Некачественный код может привести к провалу проекта

История развития программной отрасли уже бесчисленное количество раз демонстрировала, что сложные в обслуживании продукты быстро теряют рынок. Некачественный код, сгенерированный ИИ, может приводить к появлению ошибочных сообщений в критически важных приложениях, снижению доверия рынка и, как следствие, к серьезным последствиям для бизнеса: увеличению затрат на разработку и обслуживание, задержкам реализации проектов, появлению уязвимостей безопасности, ухудшению репутации, уходу клиентов и даже провалу проекта, что приводит к потере инвестиций и упущенным рыночным возможностям.

Поэтому проблема качества кода является не только абстрактной задачей для разработчиков, но также напрямую приводит к коммерческим рискам и экономическим потерям, и руководители должны четко осознавать эту взаимосвязь.

- Когда качество генерируемого ИИ кода низкое, на его исправление и обслуживание требуется больше времени, что увеличивает издержки на оплату труда. По сути, в отрасли уже сложилось общее мнение, что в будущем, скорее всего, 70 % рабочего времени придется тратить на отладку 30 % генерируемого ИИ кода.

- Уязвимости безопасности могут привести к дорогостоящим компенсациям за утечку данных и штрафам со стороны регулирующих органов.
- Проблемы с производительностью могут привести к потере пользователей, а системные недоработки чреваты подрывом репутации компании.

Для предотвращения или решения этих проблем в отрасли в настоящее время широко признанным подходом является возвращение к рациональному «ручному контролю».

5.3.4. Руководство по рецензированию кода в эпоху ИИ

Для эффективного решения уникальных проблем, связанных с генерацией кода ИИ, и обеспечения его конечного качества рецензирование кода должно системно охватывать следующие семь ключевых направлений.

1. Удобство чтения

«Удобство чтения» означает, что код ясно передает замысел разработчика и может быть легко понят другими людьми (в том числе и самим разработчиком в будущем). Код с высоким уровнем удобства чтения – это более чем просто стремление к визуальной привлекательности; его целью является снижение затрат на взаимодействие и уменьшение вероятности ошибок. Именно об этом говорил мастер программной инженерии Мартин Фаулер (Martin Fowler): «Любой дурак может написать код, понятный компьютеру, но только хороший программист может написать код, понятный человеку».

Код с высокой читаемостью, как правило, более прост в отладке и обслуживании. Разработчики могут быстро понять логику кода, что делает поиск ошибок или добавление новых функций более эффективным и безопасным процессом. Кроме того, удобство чтения влияет на уровень доверия: если код сложен для понимания и тестирования, доверие разработчиков к его корректности снижается. Таким образом, качество кода напрямую влияет на эффективность командной работы и надежность ПО.

В коде, сгенерированном ИИ, могут возникать следующие проблемы с удобством чтения.

- **Неясные имена переменных**: ИИ может использовать неоднозначные или слишком обобщенные имена переменных (например, `x`, `temp` и т. д.) вместо семантических имен. Такое неясное именование затрудняет понимание значения переменных. В качественном коде следует использовать конкретные и значимые имена, например `employeeName` вместо `fn` для обозначения имени сотрудника.
- **Сложность структуры кода**: сгенерированный ИИ код может содержать сложные логические конструкции, необходимые для выполнения задачи, такие как чересчур глубокие вложенные циклы, условные операторы или чрезмерно длинные функции. Повышенная сложность структуры затрудняет отслеживание потока кода, а также чтение и отладку.
- **Неупорядоченный стиль**: из-за разнообразия обучающих данных стиль сгенерированного ИИ кода может быть непоследовательным, например в отношении отступов, расположения скобок или стиля

именования. Такая стилистическая несогласованность затрудняет понимание, увеличивает сложность восприятия и вероятность возникновения ошибок.

- **Злоупотребление сложными приемами**: иногда ИИ использует слишком изощренные или нестандартные методы реализации (например, массовое использование вложенных тернарных операторов, битовых операций, минималистских цепочек вызовов и т. д.), пытаясь реализовать функциональность с помощью минимального количества кода. Такой «умный» код зачастую приносит в жертву удобство чтения: код становится запутанным и непонятным. Безусловно, компьютер способен быстро провести его анализ, но человеку придется приложить значительные усилия для его прочтения. Для сравнения, немного более длинная, но понятная реализация будет более удобной для последующего сопровождения.

Например, для реализации функции фильтрации и объединения данных можно написать следующий код:

```
// Исходная версия: плохая читаемость (короткий, непонятный)
function extractDataFromResponse(response) {
  const [Component, props] = response;
  const resultsEntries = Object.entries({ Component, props });
  // Фильтрация и объединение данных
  const assignIfValueTruthy = (o, [k, v]) => (v ? { ...o, [k]: v } : o);
  return resultsEntries.reduce(assignIfValueTruthy, {});
}
```

В приведенном выше коде фильтрация и объединение данных выполнены в стиле функционального программирования. Хотя количество строк кода невелико, его смысл неочевиден, и читателю приходится прилагать усилия для понимания его работы. Улучшенная версия выглядит следующим образом.

```
// Улучшенный вариант: повышенная читаемость (интуитивность, ясность)
function extractDataFromResponse(response) {
  const [Component, props] = response;
  const output = {};
  if (Component) {
    output.Component = Component;
  }
  if (props) {
    output.props = props;
  }
  return output;
}
```

Улучшенная версия четко выражает логику с помощью явных условий. Несмотря на несколько большее количество строк кода, каждый шаг операции очевиден, и читателю не нужно дополнительно размышлять над замыслом автора кода. Подобный подход с приоритетом удобства чтения значительно снижает сложность восприятия и обслуживания кода. После замены прежней короткой и непонятной реализации на интуитивную и понятную рецензенты

кода (code reviewers) и последующие разработчики будут с большей легкостью читать и редактировать этот код.

В отрасли существует множество проверенных на практике стандартов кодирования, которые фокусируются на повышении удобства чтения и согласованности кода, например PEP 8 (руководство по стилю Python) и Google Java Style Guide (руководство по стилю Java от Google). Эти стандарты охватывают множество подробных требований, от именования до форматирования, и призваны сделать код более унифицированным, читабельным и удобным для поддержки. Мы вкратце обобщим некоторые ключевые моменты стандартов кодирования.

- **Именование**: используйте понятные, самоочевидные имена, не используйте одиночные буквы или аббревиатуры. Например, используйте customer_id вместо простого x или неясного сокращения. Это относится к именам переменных, функций и классов. Удачное именование позволяет коду самому играть роль документации.
- **Отступы и компоновка кода**: придерживайтесь единых правил отступов и компоновки кода. В разных языках существуют разные соглашения, например PEP8 требует использования отступов в 4 пробела и запрещает смешанное использование табуляции и пробелов. Грамотные отступы позволяют отобразить иерархию кода, делая вложенную логику понятной с первого взгляда. В команде следует договориться о едином формате, чтобы весь код выглядел стилистически единообразно.
- **Ограничение длины строк**: избегайте чрезмерно длинных строк кода, которые затрудняют чтение. Большинство стандартов рекомендуют не делать строки кода слишком длинными, обычно ограничивая их длину 80–100 символами. Разумное разбиение на строки позволяет отображать код на экране целиком, что облегчает просмотр и рецензирование, а также способствует написанию более лаконичных выражений.
- **Длина функций**: рекомендуется создавать короткие и целенаправленные функции. Каждая функция должна выполнять только одну задачу, а объем кода следует по возможности ограничивать несколькими десятками строк. Слишком длинные функции рекомендуется разбивать, так как это часто означает, что они берут на себя слишком много задач, что затрудняет понимание и повторное использование. Соблюдение принципа «одна функция – одна задача» повышает ясность и удобство обслуживания кода.

Перечисленные выше пункты – лишь некоторые из правил. Кроме того, необходимо правильно использовать комментарии (ни слишком много, ни слишком мало), соблюдать единый стиль кодирования (правила использования заглавных и строчных букв, переносы в скобках и т. д.) и т. д. Эти основные стандарты кодирования служат разработчикам в качестве ориентира, и их основная цель – повысить удобство чтения и согласованность кода, тем самым снизив затраты на взаимодействие и поддержку.

В целом при проверке генерируемого ИИ кода следует уделять внимание стандартам удобства чтения, чтобы гарантировать, что ИИ всегда генерирует код высокого качества, легко поддающийся доработке. Это укрепит доверие команды к качеству кода и обеспечит более эффективный жизненный цикл ПО.

2. Удобство поддержки

Под поддержкой кода обычно понимается степень его понятности, модифицируемости и возможности расширения на последующих этапах. Код с высокой степенью поддержки позволяет последующим разработчикам легко понять его внутреннюю логику и при необходимости быстро исправлять ошибки или добавлять новые функции без риска возникновения дополнительных проблем. Этот аспект имеет решающее значение для долгосрочного успеха ПО, поскольку большая часть затрат на его разработку приходится не на первоначальные этапы, а на постоянное обслуживание и обновление на протяжении всего жизненного цикла. Обеспечение поддержки является залогом жизнеспособности ПО на протяжении всего его жизненного цикла.

Многие отраслевые отчеты отмечают снижение общего качества и поддержки генерируемого ИИ кода. Это связано с тем, что ИИ, как правило, стремится лишь к решению текущей задачи и пренебрегает многими долгосрочными принципами проектирования, которых придерживаются программные инженеры. В генерируемом ИИ коде нередко встречаются перечисленные ниже проблемы.

- **Монолитная структура**: ИИ имеет тенденцию группировать функции в один огромный блок кода или в отдельный файл, в результате чего пропадают четкие границы между модулями. Такой «монолитный» код похож на машину, все детали которой прочно приварены друг к другу: изменение одной функции может привести к непредвиденной цепочке сбоев и даже вызвать уязвимости в функциях, изначально не имеющих к ней отношения. По мере масштабирования приложения каждое обновление становится похожим на «разборку бомбы»: малейшая оплошность может вызвать цепную реакцию проблем.
- **Плотное взаимодействие**: под этим понимается слишком глубокая зависимость между модулями кода и отсутствие независимости. В генерируемом ИИ коде нередко встречаются ситуации, когда модули напрямую вызывают внутренние функции друг друга или используют общие глобальные переменные, что приводит к эффекту «потяни за нитку – и все развалится». В таких системах с плотным взаимодействием изменение одного модуля зачастую требует синхронного изменения нескольких других, иначе это приведет к нарушению работы всей системы. Например, иногда сгенерированный ИИ код бизнес-логики жестко прописывает подробности доступа к данным, в результате чего даже небольшое изменение структуры базы данных приводит к необходимости значительной переработки этой бизнес-логики. Плотное взаимодействие также подразумевает невозможность локального повторного использования или замены кода: если одна функция связана с другой, ее трудно использовать отдельно в других проектах, и даже обособленное тестирование или отладка будут затруднены. Такой дизайн с плотным взаимодействием, несомненно, повышает вероятность ошибок при модификации и расширяет зону их воздействия.
- **Низкая связность**: под этим понимается ситуация, при которой модуль выполняет слишком много не связанных между собой задач и не имеет единой цели. Генерируемый ИИ код иногда заставляет одну функцию

или класс выполнять множество не связанных между собой действий, что приводит к их смешению. Прямым следствием низкой связности является сложность понимания и поддержки кода: если модуль часто изменяется по разным причинам, это означает, что он не сосредоточен на выполнении одной конкретной задачи. Например, если функция «обработка заказа» содержит как вывод лога, так и обновление пользовательского интерфейса, то при изменении политики ведения логов или макета интерфейса придется изменять эту функцию, что является типичным примером низкой связности. В таком случае специалистам технической поддержки бывает сложно предсказать, на какую часть логики модуля повлияет изменение, и модификация одного требования может непреднамеренно нарушить работу другого, что значительно увеличивает неопределенность и риски при реализации.

Для решения упомянутых выше проблем в программной инженерии были сформулированы три основных принципа проектирования: модульность, низкая связность и высокая однородность, которые помогают нам писать код, более удобный для обслуживания.

- **Модульность**: концепцию модульности можно сравнить со сборкой конструктора или Лего. Каждый модуль подобен отдельному блоку с четко определенной формой и интерфейсом. Эти «блоки» соединяются между собой через стандартные интерфейсы, но в обычных условиях остаются независимыми друг от друга, что дает возможность свободно их разбирать, комбинировать и модернизировать. Хорошая модульность означает, что если нам нужно заменить или усовершенствовать какой-либо функциональный модуль, то это будет похоже на замену кубика Лего: достаточно просто извлечь его и заменить новым, не разбирая всю конструкцию.
- **Низкая связность**: низкая связность модулей похожа на работу бытовых приборов, каждый из которых выполняет свою функцию, не мешая другим. Например, холодильник, стиральная машина и кондиционер – это независимые приборы, и поломка одного из них не приводит к выходу из строя другого. В контексте кода низкая связность означает, что модули взаимодействуют через прозрачные интерфейсы, а их внутренняя реализация недоступна для других. Таким образом, при изменении одного модуля другие модули не требуют модификации и даже не замечают этих изменений (точно так же, как замена холодильника не влияет на работу стиральной машины).
- **Высокая однородность**: это четкое распределение обязанностей между модулями, при котором каждый выполняет свою задачу. Модули с высокой однородностью можно сравнить со специализированными отделами в компании: финансовый отдел занимается только финансами, отдел кадров занимается только кадровыми вопросами; у каждого отдела есть своя единственная обязанность, тесно связанная с его задачами, и не связанные с ним дела не обрабатываются в одном отделе. Это гарантирует, что все элементы внутри каждого модуля работают над достижением одной цели, а взаимодействие внутри модуля тесное, но сфокусировано на одной задаче. Что касается кода, высокая однородность требует, что-

бы модуль выполнял по возможности только одну задачу, что упрощает и его анализ, и его сопровождение. Если класс или функция называется «Генерация отчетов», то он должен отвечать только за генерацию отчетов (одна функция), а не заниматься загрузкой файлов или рисованием окон. Такая однозначная ответственность обеспечивает высокую внутреннюю согласованность модуля: при изменении какой-либо функции достаточно найти соответствующий модуль, что значительно снижает риск непреднамеренного влияния на другие функции.

Модульность предоставляет способ разбиения системы, низкая связность гарантирует свободу независимой замены и развития каждого модуля, а высокая однородность обеспечивает упорядоченность и четкость целей внутри каждого модуля. Сочетание этих трех принципов делает ПО похожим на конструкцию из стандартизированных деталей, где каждая часть имеет четкое назначение, но при этом может быть собрана в единое целое с помощью подходящих интерфейсов. Когда система соответствует этим принципам, можно с большей уверенностью вносить изменения в отдельные части, без опасений, что это повлияет на всю систему в целом. И наоборот, если в генерируемом ИИ коде мы не видим такого «модульного» дизайна, следует проявить бдительность.

3. Документация и комментарии

Хорошая документация и комментарии к коду повышают читаемость кода, облегчают совместную работу команды, ускоряют последующее обслуживание и помогают рецензентам понять цель кода и замысел его разработчиков. В случае сложной бизнес-логики комментарии не только рассказывают о выполненных действиях, но и объясняют их причины. Фиксируя в коде ключевые проектные решения и намерения, документация и комментарии предоставляют читателям кода контекстную информацию, позволяя не увязнуть в ситуации, когда видны только шаги, но отсутствует понимание их смысла.

Документация и комментарии полезны не только для людей, но и для больших языковых моделей. Существующие большие языковые модели обладают функцией мультимодального понимания кода, что позволяет им одновременно обрабатывать данные из разных источников и моделировать семантику кода, приближаясь к способу понимания человека-программиста. Корреляция между документацией, комментариями и кодом также является неотъемлемой частью этого процесса, и в некоторых сценариях значение комментариев может даже превышать значение соответствующего кода.

Однако в настоящее время ИИ-инструменты для программирования зачастую не позволяют автоматически генерировать содержательную документацию и комментарии. В генерируемом ИИ коде нередко отсутствуют комментарии или же присутствуют лишь поверхностные пояснения, например строка «Проверяем, меньше ли общее количество 50» перед условием `if total < 50`. Такие комментарии лишь механически повторяют логику кода, не предоставляя никакой дополнительной информации. Еще более серьезной проблемой является отсутствие в генерируемом ИИ коде объяснений причин выбора того или иного решения. Даже при использовании нестандартных или сложных методов генерируемый код обычно не сопровождается обоснованием причин, а лишь предоставляет результат, не фиксируя процесс решения задачи.

Тем не менее пути решения этой проблемы существуют: при проверке генерируемого ИИ кода в контексте вайб-программирования рекомендуется уделять особое внимание следующим аспектам.

- Есть ли объяснения сложных фрагментов? Предоставлены ли соответствующие комментарии или документация, объясняющие назначение сложных алгоритмов или непонятных фрагментов кода? Хорошие комментарии помогают сотрудникам с разным техническим опытом быстро разобраться в логике и решениях, лежащих в основе кода. Если в коде реализованы нестандартные приемы или алгоритмы, рецензент должен проверить наличие комментариев, объясняющих их суть. Если таких комментариев нет, следует потребовать их добавления, иначе последующие читатели, скорее всего, не смогут понять эту часть кода.
- Понятно ли, почему код написан именно так? Документация и комментарии должны помогать рецензенту понять намерения разработчика. Например, если в какой-то функции используется особый вариант реализации, комментарий должен объяснять причину или преимущества такого подхода. При рецензировании следует искать в коде комментарии, отражающие процесс мышления разработчика при решении задачи. Если код на первый взгляд работает, но его цель неочевидна, это означает, что документация недоработана, и рецензент должен указать на это и потребовать дополнений.
- Зафиксированы ли ключевые слова и вручную внесенные изменения? При рецензировании генерируемого ИИ кода обращайте внимание на наличие записей о ключевых изменениях, внесенных разработчиком в генерируемый ИИ код. Например, если такой код прошел ручную доработку (была исправлена скрытая ошибка или оптимизирована производительность), эти изменения лучше всего отметить в комментариях к коду или в описании ревизии, чтобы последующие читатели могли сразу понять, какие именно части были изменены и по какой причине. При отсутствии таких записей рецензент должен задать вопросы и убедиться, что все особые моменты в коде имеют четкое обоснование.

Суть в том, что ИИ обычно не объясняет свои дизайнерские решения, и человеку необходимо дополнить код не только описанием выполненных действий, но и объяснением причин. Только добавление контекстных пояснений и комментариев с обоснованием к генерируемому ИИ коду позволит последующим читателям понять его назначение. Ниже приведены общепринятые в отрасли рекомендации по документированию и комментированию, которые помогут обеспечить понятность и удобство обслуживания кода.

- **Написание Docstring (документационных строк) для функций**: основные стандарты требуют, чтобы все открытые модули, функции, классы и методы имели Docstring. Хороший Docstring обычно включает в себя обзор функциональности (описание назначения в одном предложении), более подробные пояснения (если необходимо), объяснение параметров и возвращаемых значений, а также возможные варианты использования. Документация Docstring упрощает понимание предназначения кода и позволяет любому пользователю быстро понять назначение функции при ее просмотре в будущем. Например:

```
/**
 * Проверка того, является ли положительное целое число степенью числа 2
 *
 * ### Предпосылки и принцип работы
 * В двоичном представлении степени числа 2 имеют только один бит,
 * равный 1 (например, 1 → 1, 2 → 10, 4 → 100)
 * Для таких чисел:
 * ```
 * n & (n - 1) === 0
 * ```
 * Поскольку n - 1 сбрасывает единственный бит 1 в 0 и устанавливает все биты
 * справа от него в 1,
 * результат операции равен 0.
 *
 * @param n - положительное целое число, которое необходимо проверить
 * @returns Если n является степенью числа 2, возвращается true;
 * в противном случае возвращается false
 * @throws TypeError В случае, если n не является безопасным положительным
 * целым числом, генерируется исключение
 * @example
 * ```ts
 * isPowerOfTwo(8);  // => true
 * isPowerOfTwo(7);  // => false
 * ```
 */
export function isPowerOfTwo(n: number): boolean {
  if (!Number.isSafeInteger(n) || n <= 0) {
    throw new TypeError(„n должно быть положительным целым числом“);
  }
  return (n & (n - 1)) === 0;
}
```

- **Написание полного документа README**: на уровне проекта документ README должен содержать основную информацию о проекте и помогать новичкам быстро освоить и запустить проект. Как правило, полноценный README включает как минимум введение в проект (цель и обзор его функций), зависимости среды, инструкции по установке, базовые инструкции по использованию или примеры кода. Кроме того, в зависимости от характера проекта он должен включать примеры использования или скриншоты, контактные данные разработчика, ответы на часто задаваемые вопросы, а также руководство по внесению изменений, лицензионное соглашение и т. д.
- **Принцип комментариев «объясняйте “почему”, а не описывайте “что”»**: комментарии в коде должны служить для объяснения причин и намерений, лежащих в основе кода, а не просто описывать его операционную часть. Как упоминалось выше, в случае сложных алгоритмов, важных бизнес-правил или нестандартных реализаций комментарии должны объяснять причины такой реализации, принципы алгоритма и соображения, лежащие в основе проекта. И наоборот, для простого кода, понятного с первого взгляда, не нужно излишне добавлять коммен-

тарии, объясняющие «что сделано», чтобы избежать бессмысленности комментариев. Например, не следует писать самоочевидные комментарии типа «i = i + 1 // i плюс 1»; вместо этого следует сосредоточиться на назначении кода, например: «// Используем i плюс 1 для прохождения по индексу массива, потому что... (причина)». В целом критерием для определения необходимости комментария является следующий вопрос: может ли читатель легко понять назначение кода по его содержанию? Если нет, то следует добавить комментарий, объясняющий мотивацию и причину.

- **Своевременно обновляйте документацию и комментарии**: документация и комментарии должны развиваться синхронно с кодом. При каждом изменении кода или корректировке логики соответствующие комментарии также должны оперативно обновляться, чтобы избежать появления вводящих в заблуждение устаревших комментариев. Устаревшие или ошибочные комментарии наносят больший вред, чем их отсутствие, поскольку они могут привести к неверному пониманию рецензентов и разработчиков, занимающихся поддержкой. Поэтому при рецензировании кода следует проверять, точно ли комментарии отражают текущее поведение кода. Рекомендуется включить проверку документации и комментариев в процесс рецензирования кода в качестве обязательного элемента, чтобы гарантировать их постоянную достоверность и надежность.
- **Удаляйте устаревший или ненужный код, избегайте обширных комментариев**: в репозитории кода не следует долго хранить «мертвый» код с комментариями. Если какой-то фрагмент кода больше не нужен, его следует немедленно удалить, а не оставлять с комментариями. Современные системы контроля версий позволяют отслеживать историю кода и при необходимости восстанавливать старые версии. Сохранение ненужного кода в репозитории только усложняет чтение и заставляет рецензентов ломать голову над тем, почему этот код был помечен комментарием. Большие фрагменты кода с комментариями, как правило, больше не используются и должны быть удалены; если же планируется их повторное использование, в комментарии необходимо четко указать причину и срок.

Таким образом, при проверке кода рецензент должен учитывать наличие полной документации и грамотных комментариев как один из важных показателей качества кода. Четкая документация и поясняющие комментарии помогают другим понять сложную логику, а также передают идеи и цели разработчиков. Напротив, отсутствие документации или небрежные комментарии значительно затрудняют понимание кода и создают риски при его обслуживании. В эпоху возрастающей роли ИИ в программировании нам особенно важно серьезно относиться к работе с документацией: использовать преимущества человека для объяснения «почему» и дополнять комментариями и документацией генерацию кода ИИ. Только так рецензирование кода сможет по-настоящему раскрыть свой потенциал и обеспечить не только работу кода, но и его понятность и возможность дальнейшего улучшения.

4. Стратегия ведения логов

По умолчанию после запуска программная система переходит в состояние «черного ящика», и заглянуть в ее внутреннее состояние можно только с помощью специальных инструментов отладки в особых случаях. Однако если проблема возникает на стороне клиента или в среде без инструментов отладки, то без точной и полной информации в логах отследить источник проблемы будет невозможно. Ключ к решению этой задачи заключается в создании полноценной системы ведения логов программы.

Лог (или **журнал**) – это запись ключевых событий и данных во время выполнения программы. Такие записи помогают в отладке и отслеживании проблем: когда в программе происходит сбой, логи позволяют разработчику быстро локализовать проблему. Они также фиксируют подробные данные о производительности (такие как время отклика и использование ресурсов), помогая команде оптимизировать эффективность системы. Кроме того, логи могут использоваться для аудита безопасности и отслеживания соответствия нормативным требованиям: благодаря записи ключевых операций и истории изменений они помогают при анализе безопасности и выполнении нормативных требований. Одним словом, логи обеспечивают визуализацию состояния системы, а это помогает команде разработчиков более эффективно контролировать и улучшать приложение.

Однако ИИ не обладает интуитивным пониманием бизнес-логики, как человек, и не может определить наиболее важные узлы. По этой причине он чаще всего не справляется с обработкой системных логов. Иногда ИИ добавляет в код слишком много логов, генерируя огромный объем нерелевантной информации, в результате чего важные сведения теряются в потоке шума. В других случаях логов бывает недостаточно, и ключевые шаги не фиксируются, что затрудняет поиск и устранение неполадок. Еще более серьезной проблемой является неправильная политика ведения логов: генерируемый ИИ код может записывать в лог-файлы личные данные пользователей или конфиденциальную информацию, создавая угрозу безопасности.

Поэтому в отношении генерируемых ИИ модулей кода рекомендуется проводить тщательную проверку содержания выводимых логов. Необходимо обеспечить достаточно подробную запись логов для ключевых узлов и одновременно внимательно проверять их на наличие избыточных записей или утечки конфиденциальной информации, уделяя особое внимание следующим аспектам.

- **Ключевые данные**: запись входящих параметров интерфейса, промежуточных переменных и ключевых параметров конфигурации (режимы, флаги свойств и т. д.).
- **Ветвления операционной логики**: вывод логов в местах ветвления (`if/else` и т. п.) с указанием пути выполнения (например, «отклонить транзакцию, если баланс пользователя недостаточен»).
- **Внешние интерфейсы**: запись запросов к внешним сервисам и результатов их выполнения (конфиденциальная информация должна быть обезличена).
- **Обработка исключений**: при перехвате исключений следует записывать информацию об ошибке и стек вызовов; также следует вести логи для операционных исключений.

Кроме того, для более целенаправленного использования логов их следует разделить на различные типы, каждый из которых будет ориентирован на определенные аспекты данных, и следовать соответствующим рекомендациям.

- **Логи производительности**: используются для мониторинга показателей производительности системы, записи времени выполнения кода, задержки запросов, использования памяти, загрузки центрального процессора, а также коэффициента использования кеша и другой информации для оценки эффективности системы, например { `"latency_ms ": 123, "endpoint ": "/recommend "`}. Логи производительности помогают разработчикам выявлять узкие места и проводить оптимизацию.
- **Логи поведения пользователей**: фиксируют ключевые действия и пути поведения пользователей, используются для оптимизации продукта и бизнес-анализа. Записывают такие события, как, например, клики, поиски, конверсии и т. д., и содержат такие поля, как ID пользователя, тип действия, целевой объект и т. д. ({`"user_id ": "abc123 ", "action ": "click ", "item_id ": "xyz "`}). Такие журналы представляют собой трек пользовательского поведения и фиксируют данные, генерируемые при каждом взаимодействии пользователя (посещения, просмотры, поиски, клики и т. д.). Анализ логов пользовательского поведения позволяет улучшить качество продукта, проверить эффективность функций и выявить аномальные модели использования.
- **Логи ошибок**: отслеживают системные ошибки и аномальные ситуации, подробно фиксируя коды ошибок, сообщения об ошибках, трассировку стека и принятые меры по переходу на резервный вариант (например, { `"error_code ": "E502 ", "message ": "timeout ", "action ": "fallback_used "`}). Используются для устранения неисправностей и повышения отказоустойчивости системы.

Использование приведенной выше классификации помогает применять различные стратегии сохранения и анализа к разным типам логов. Например, логи производительности можно направлять в систему мониторинга для получения оповещений в режиме реального времени, логи поведения пользователей можно агрегировать для анализа данных, а логи ошибок можно направлять в службу поддержки для устранения неполадок.

5. Обработка ошибок

Обработка ошибок – это процесс реагирования и устранения проблем, возникающих при исполнении кода в результате непредвиденных ситуаций, сбоев или неверных входных данных. Правильная обработка гарантирует нормальную работу ПО при различных предельных входных данных без сбоев и ошибочных результатов. Неправильная обработка, напротив, может привести к сбою программы или непредвиденным системным последствиям. Иными словами, обработка ошибок напрямую влияет на надежность приложения и удобство пользователей, а также определяет способность системы работать без сбоев в нестандартных ситуациях.

Предположим, что код должен преобразовать пользовательский ввод в число и выполнить вычисление:

```
# Код без обработки ошибок
result = 10 / int(user_input)
print("Результат вычисления:", result)
# Если user_input не является числом или равен 0, генерируется исключение
# и происходит сбой
```

Указанный выше код при встрече с недопустимым вводом сразу выдает ошибку и заканчивает работу. Код с добавлением надлежащей обработки ошибок выглядит следующим образом:

```
# Код с обработкой ошибок
try:
    result = 10 / int(user_input)
    print("Результат вычисления:", result)
except (ValueError, ZeroDivisionError) as e:
    # Перехват ошибки преобразования или ошибки деления на ноль
    print("Произошла ошибка:", e, ". Использован результат по умолчанию 0.")
    result = 0  # Использование значения по умолчанию в качестве меры
                # по уменьшению проблем
```

В улучшенном коде мы используем блок `try...except` для перехвата возможных исключений: если ввод не является числом или равен нулю, вызванная ошибка будет перехвачена. Программа не завершит работу, а выведет сообщение об ошибке, а затем продолжит работу со значением по умолчанию. Таким образом, мы не только предотвращаем прерывание работы программы из-за исключений, но также даем пользователям и разработчикам четкое представление о характере ошибки и о принятых системой мерах по ее устранению.

В сценарии вайб-программирования ИИ может быстро генерировать код на основе описания разработчика, но в плане обработки критических ситуаций нередко можно наблюдать две крайности.

- **Отсутствие обработки ошибок**: в коде отсутствуют необходимые проверки и перехват исключений. При возникновении ошибки (например, файл не найден, сетевой запрос завершился неудачей или введен неверный ввод) программа завершит работу из-за необработанного исключения. В таком случае пользователь увидит внезапный сбой программы или отсутствие отклика. Например, если код напрямую выполняет вычисления по пользовательскому вводу без проверки его корректности, недопустимые значения ввода могут привести к сбою всего приложения. Отсутствие обработки ошибок серьезно угрожает стабильности системы, делая ПО уязвимым в любой непредвиденной ситуации.
- **Чрезмерная обработка ошибок**: это означает, что код содержит слишком много перехватов исключений, в том числе для практически невозможных ситуаций. На первый взгляд, код со множеством конструкций `try...catch` или `try...except` кажется надежным, но чрезмерная обработка ошибок может привести к явлению «тихого сбоя», когда возникшая в программе проблема ничем не проявляется: не генерируется ни исключение, ни предупреждение, а программа продолжает работать с использованием значений по умолчанию или с пустым результатом. Хотя

при таком подходе программа не дает сбоев, ее фактическая функциональность ухудшается, а пользователи и разработчики этого не замечают, что создает огромные трудности при отладке и обслуживании.

Поэтому при рецензировании генерируемого ИИ кода следует уделять особое внимание корректности обработки ошибок, в частности обращая внимание на следующие моменты.

- **Проверка аномальных входных данных**: обратите внимание на наличие в коде проверки корректности входных данных. Например, проверяется ли наличие файла перед его чтением, проверяется ли, не равен ли делитель нулю перед выполнением вычислений и т. д. Если код напрямую использует входные данные без условных проверок или перехвата исключений, то при несоответствии входных данных ожидаемым значениям может ли система выйти из строя или зависнуть? Если такая логика проверки отсутствует, это означает, что обработка ошибок, вероятно, отсутствует, и при получении некорректных данных программа может аварийно завершиться.
- **Резервный план на случай сбоя**: проверьте наличие в коде **резервного плана** (fallback) на тот случай, если вызов какого-либо модуля ИИ завершится сбоем или вернет аномальный результат. Хороший код при сбое основной операции переходит на альтернативный путь, например повторяет запрос, использует значения по умолчанию или переключается на резервную функцию, а не просто прерывает работу программы. Если в коде отсутствует такой резервный план, то при возникновении ошибки вся система может оказаться выведенной из строя.
- **Ясность сообщений об ошибках**: проверьте обработку сообщений об ошибках при перехвате исключений. Являются ли сообщения об ошибках удобными для пользователя и полезными для разработчика? В идеале при возникновении проблемы программа должна выдавать четкое сообщение, например «Сбой сетевого соединения, попробуйте позже», а также записывать подробную информацию об исключении в логи для отладки разработчиком. Если обработка исключений в коде представляет собой пустой блок catch или выводит только неясную информацию типа «Error», это является неприемлемым. Слишком простые или урезанные сообщения об ошибках не дают пользователю представления о происходящем и затрудняют разработчику поиск и устранение проблемы.

Таким образом, обработка ошибок определяет надежность генерируемого ИИ кода в реальных условиях. Отсутствие обработки ошибок приводит к сбою программы при возникновении исключения, а чрезмерная или некорректная обработка может скрыть проблему, приводя к скрытому сбою. Для вайб-программирования, ориентированного на эффективность, особенно важно уделять внимание этому уязвимому месту в коде. На практике следует проводить тщательное рецензирование и тестирование кода, чтобы гарантировать, что генерируемый ИИ код будет правильно обрабатывать различные нештатные ситуации и не будет скрывать ошибки. Только так система сможет оставаться стабильной и надежной в любых условиях, обеспечивая высокое качество обслуживания конечных пользователей.

6. Безопасность

Безопасность кода означает его способность противостоять злонамеренным атакам и уязвимостям, то есть гарантировать, что приложение не позволит злоумышленникам воспользоваться дефектами кода или получить доступ к конфиденциальным данным. Исследования показывают, что в генерируемом ИИ коде, где не уделяется особого внимания безопасности, от 40 % до 90 % кода содержат уязвимости (данные из отчета Backslash). Ниже перечислены наиболее распространенные проблемы безопасности.

- **Повторное использование небезопасных шаблонов кода**: ИИ обучается на большом объеме готового кода и может непреднамеренно заимствовать небезопасные шаблоны кодирования из обучающих данных. Другими словами, если в общедоступных репозиториях кода широко распространены реализации с уязвимостями [такими как незащищенная SQL-инъекция (обман базы данных с помощью специальных входных данных для выполнения несанкционированных операций) или межсайтовые скрипты (XSS-атаки, вставка вредоносных скриптов в браузер для их выполнения)], то генерируемый ИИ код может повторять эти небезопасные шаблоны кодирования. Например, ИИ может просто скомпоновать строку SQL-запроса в соответствии с обучающими образцами и не учитывать, что это приведет к уязвимости SQL-инъекции. Точно так же генерируемый ИИ код веб-страниц может напрямую вставлять пользовательский ввод в HTML-скрипты, что создает риск XSS-атак.
- **Использование устаревших или подверженных высокому риску библиотек**: ИИ иногда рекомендует использовать устаревшие или содержащие известные уязвимости сторонние библиотеки, поскольку его обучающий корпус содержит большое количество исторического кода и устаревших практик, в результате чего генерируемый код может включать функции, алгоритмы или версии библиотек, которые сообщество уже признало небезопасными. Например, ИИ может порекомендовать библиотеку функций шифрования старой версии, в которой уже обнаружены уязвимости. Использование таких устаревших зависимостей повышает риски безопасности, поскольку злоумышленники могут воспользоваться известными уязвимостями старых версий.
- **Жесткое кодирование конфиденциальной информации**: во многих примерах генерируемого ИИ кода конфиденциальная информация, такая как пароли к базам данных, ключи API и т. д., жестко встраивается напрямую в код. Такой подход очень опасен, поскольку жестко закодированные конфиденциальные данные легко могут быть прочитаны посторонними или попасть в открытый доступ. В прошлом уже неоднократно имели место случаи, когда разработчики непреднамеренно загружали код с паролями или токенами в общедоступные репозитории. Согласно недавнему отчету GitGuardian, в 2024 году на GitHub было непреднамеренно раскрыто почти 24 млн различных ключей, а уровень утечки конфиденциальной информации из репозиториев кода, созданного с помощью ИИ-инструментов, превысил 40 % (данные из отчета infisical). Из этого видно, что ИИ-генерируемый код зачастую не учитывает не-

обходимость защиты конфиденциальной информации, поэтому для хранения таких данных следует использовать переменные среды или безопасные конфигурации.

- **Отсутствие проверки входных данных**: из-за чрезмерной озабоченности реализацией функциональности в генерируемом ИИ коде нередко упускается необходимая проверка входных данных и предельных значений. Иными словами, код может безоговорочно доверять предоставленным пользователем данным, не проверяя их формат, длину или корректность. Это приводит к различным проблемам безопасности: злоумышленники могут использовать непроверенные входные данные для SQL-инъекций или XSS-атак либо вызвать сбой программы с помощью аномальных значений (например, передавая чрезмерно большие числа или специальные символы, приводящие к ошибке). Безопасный код должен тщательно проверять и очищать пользовательские входные данные, а исходный ИИ-код обычно требует добавления таких защитных мер.

При рецензировании ИИ-кода можно использовать следующий чек-лист безопасности для выявления потенциальных проблем.

- Вставляются ли пользовательские входные данные напрямую в SQL-запросы?
- Используются ли библиотеки сторонних разработчиков в последних версиях и не содержат ли они известных уязвимостей?
- Закодированы ли в коде пароли, API-ключи и другая конфиденциальная информация?
- Проводится ли необходимая проверка и контроль предельных значений для внешних входных данных?

Наслаждаясь повышением эффективности разработки, которое дает ИИ, мы должны оставаться бдительными в отношении безопасности кода. Понимание основных принципов безопасности кода, распознавание распространенных проблем безопасности в генерируемом ИИ коде, а также целенаправленная проверка и исправление кода при рецензировании позволяют свести к минимуму риски безопасности, возникающие при использовании ИИ. Только так можно одновременно наслаждаться атмосферой быстрой разработки, обеспечивая при этом безопасность и надежность приложения.

7. Производительность

Под **производительностью кода** подразумевается эффективность работы программы. Это очень многогранная концепция, включающая такие аспекты, как скорость выполнения (время, необходимое для выполнения кода), потребление ресурсов (использование ресурсов процессора, памяти и т. д.), а также масштабируемость (способность стабильно работать при увеличении нагрузки). Код с высокой производительностью обеспечивает плавную работу приложения, эффективное использование ресурсов и плавное масштабирование по мере роста потребностей. Не будет преувеличением сказать, что производительность напрямую влияет на успех или провал программного проекта.

- **Пользовательский опыт**: производительность ПО напрямую влияет на степень удовлетворения пользователей. Медленно работающие или зависающие приложения вызывают у пользователей разочарование, что приводит к снижению их лояльности и даже к уходу. Например, если загрузка веб-страницы задерживается на несколько сотен миллисекунд, пользователь может просто покинуть сайт, что приводит к значительному снижению конверсии на сайтах электронной коммерции. Бесперебойная работа является важной составляющей конкурентоспособности продукта, а оптимизация производительности обычно повышает уровень удержания пользователей и улучшает репутацию продукта.
- **Стабильность системы**: код с низкой производительностью более подвержен сбоям при высокой нагрузке. Когда система не справляется с резким увеличением количества пользователей или данных, могут возникнуть задержки отклика или даже сбои. Например, в условиях одновременного доступа большого количества пользователей на сервере из-за неэффективного алгоритма может возникнуть чрезмерная нагрузка на центральный процессор (CPU), что может привести к задержкам в обработке запросов и вызвать цепную реакцию сбоев. Узкие места в производительности также могут тормозить работу всей системы, приводя к сбоям в работе других компонентов. Напротив, хорошая производительность и масштабируемость позволяют приложению стабильно работать даже в часы пиковой нагрузки без ущерба для бизнес-процессов.
- **Эксплуатационные расходы**: неэффективный код зачастую означает необходимость в большем количестве оборудования и более высоких затратах на обслуживание. Если для выполнения задачи требуется вдвое больше ресурсов, то компании приходится вкладывать средства в дополнительные серверы и память. Кроме того, сбои из-за проблем с производительностью требуют незамедлительного устранения инженерами, что увеличивает затраты на персонал и повышает трудозатраты. В случае с облачной архитектурой плохая производительность напрямую отражается на более высоких счетах за облачные ресурсы. Таким образом, оптимизация производительности кода не только повышает удовлетворенность пользователей, но также экономит деньги.

Проблема заключается в том, что, несмотря на способность быстро генерировать код, ИИ обычно уделяет приоритетное внимание точности реализации функциональности, а не лучшим практикам в области производительности. Многие исследования уже доказали, что генерируемый ИИ код характеризуется снижением производительности (данные из статьи «Оценка производительности генерируемого ИИ кода: тематическое исследование GitHub Copilot» – Assessing the Performance of AI-Generated Code: A Case Study on GitHub Copilot). В генерируемом ИИ коде могут встречаться следующие скрытые угрозы производительности.

- **Неэффективные алгоритмы**: генерируемый ИИ код может использовать неоптимальные алгоритмы, что приводит к излишней сложности. Например, для решения задачи, которую можно было бы решить по алгоритму линейной сложности, используются двойные или даже многоуровневые вложенные циклы, что повышает временную сложность до

$O(n^2)$ или выше. Реальный пример: использование вложенных циклов при удалении дубликатов из матриц или списков, что гораздо менее эффективно, чем использование таких структур данных, как хеш-таблицы или массивы.

- **Избыточные вычисления**: в генерируемом ИИ коде часто встречаются повторяющиеся вычисления или вызовы; иногда внутри циклов повторно вычисляются одни и те же значения или многократно вызываются ресурсоемкие функции без кеширования. Например, пересчет длины массива при каждой итерации, повторные запросы к базе данных или файловый ввод-вывод приводят к потере производительности. В некоторых случаях в циклах повторно создаются временные объекты или структуры данных, при этом при каждой итерации заново выделяется память, что увеличивает нагрузку на систему очистки памяти.
- **Неэффективное управление ресурсами**: генерируемый ИИ код может игнорировать эффективное управление ресурсами, в том числе своевременное освобождение неиспользуемой памяти или сетевых соединений, частое выполнение ресурсоемких вычислений на стороне клиента или неумение использовать высокопроизводительные возможности языка программирования. Например, однократное считывание больших объемов данных без использования потоковой обработки или ручная реализация функций, уже имеющихся в высокопроизводительных библиотеках. Такие подходы приводят к утечкам памяти, резкому росту загрузки центрального процессора и другим проблемам, снижая производительность и стабильность системы.

Эти проблемы носят скрытый характер: генерируемый ИИ код может вызывать на первый взгляд безобидные операции, которые тем не менее из-за слишком высокой частоты или неэффективного подхода создают проблемы с производительностью. Поэтому важно тщательно анализировать сгенерированный код, проверять его корректность и избегать его слепого использования в сценариях, чувствительных к производительности. В частности, рецензенты кода должны уделять особое внимание следующим аспектам производительности.

- **Анализ алгоритмов и сложности**: обратите внимание на наличие эффективных алгоритмов в коде. Проверьте, нет ли ненужных вложенных циклов или реализаций с экспоненциальной сложностью. Если вы обнаружите тройные циклы, частые сортировки или глубокую рекурсию, оцените их необходимость. Рецензент может попробовать оценить время выполнения в худшем случае и поискать более оптимальные алгоритмы или структуры данных.
- **Выявление повторяющихся вычислений**: при просмотре кода обращайте внимание на повторяющиеся вычисления. Типичные признаки включают вызов ресурсоемких функций внутри циклов или повторную обработку неизменяемых данных. В таких случаях рекомендуется кешировать результаты вычислений или выносить за пределы цикла те операции, которые можно выполнить вне него. Например, следует избегать повторного вычисления длины массива в каждой итерации или повторного чтения одного и того же файла.

- **Проверка использования ресурсов и памяти**: наблюдайте за моделями использования ресурсов в коде. Например, опасными сигналами являются создание большого количества объектов в цикле или выделение больших блоков памяти. При проверке можно предложить повторно использовать объекты, применять технологии пулов или по возможности создавать объекты вне цикла. Также необходимо проверить корректность закрытия файлов, подключений к базам данных и других ресурсов, чтобы предотвратить снижение производительности из-за утечки ресурсов. Что касается кода веб-фронтенда, следует обратить внимание на частоту операций с DOM или выполнение большого количества синхронных вычислений, так как все это может блокировать отклик страницы.
- **Использование языковых возможностей и лучших практик**: иногда генерируемый ИИ код не использует эффективные возможности выбранного языка программирования. По этой причине при рецензировании кода следует выявлять места, где можно заменить его более эффективными методами. Например, для объединения строк можно использовать шаблонные строки или операции слияния массивов, а для сложных математических вычислений могут быть доступны встроенные функции. Точно так же обращайте внимание на наличие неиспользуемых переменных, «мертвого» кода и тому подобного: они не только ухудшают читаемость, но также могут указывать на проблемы с производительностью (например, когда результаты ненужных вычислений не используются).

Благодаря описанным выше проверкам рецензент может выявить проблемы с производительностью на этапе проверки кода, не допуская их появления в производственной среде. Это повышает качество кода и формирует в команде понимание важности оптимизации производительности. На практике рекомендуется включить проверку производительности в чек-лист рецензирования кода, чтобы гарантировать, что внимание факторам эффективности уделяется при каждой рецензии.

5.3.5. Создание системы рецензирования кода

ИИ может генерировать код, который правильно выполняет функции, но не соответствует стратегическим приоритетам или архитектурным принципам, в то время как люди способны понимать бизнес-цели и долгосрочную стратегию продукта, а также вытекающие из них технические решения. Поэтому в настоящее время ручное рецензирование по-прежнему остается оптимальным способом обеспечения качества кода. Для этого необходимо создать некоторые процессуальные механизмы проверки кода, такие как несколько упомянутых ниже стратегий.

1. Проверка на начальном этапе

При использовании инструментов ИИ, таких как Cursor, для генерации кода не следует спешить с принятием генерируемого ИИ кода, а сначала тщательно его прочитать и понять, чтобы по возможности уменьшить количество проблем еще на начальном этапе. Для этого рецензенту необходимо выполнить следующие действия.

- **Внимательное чтение и понимание**: относитесь к генерируемому ИИ коду так, если бы он был написан начинающим инженером (например, стажером): на первый взгляд он может выглядеть синтаксически правильным и упорядоченным, но за этим может скрываться недостаточное внимание к нюансам рабочего процесса. Рецензент должен проверять логику кода строка за строкой, чтобы убедиться в его соответствии ожидаемой функциональности и требованиям.
- **Проверка основной логики и граничных условий**: генерируемый ИИ код зачастую охватывает только «нормальный путь», но не учитывает обработку исключений или граничных условий. Перед утверждением кода следует вручную проверить ключевые функции и модули, например введя различные граничные значения, чтобы оценить правильность вывода, а также убедиться в достаточной обработке исключений. По возможности следует запустить небольшие фрагменты кода или написать простые тесты для проверки его поведения и убедиться, что ИИ не упустил ключевые сценарии.
- **Проверка стиля и интеграции**: поскольку ИИ учится различным стилям из интернета, сгенерированный им код может не соответствовать стандартам кодирования команды или архитектуре проекта. Следует проверить соответствие именования стандартам и единообразие стиля кода, а также внести необходимые корректировки перед принятием кода. Рецензент также должен проверить точки интеграции кода ИИ с существующим кодом, убедиться, что ссылки на классы, функции или библиотеки присутствуют в проекте и совместимы со своими версиями, чтобы избежать проблем с интеграцией из-за ошибочных предположений ИИ.
- **Первичная проверка безопасности и производительности**: еще на начальном этапе рецензент должен выявить очевидные уязвимости безопасности и проблемы с производительностью. Например, необходимо проверить наличие в коде ИИ прямого использования непроверенных пользовательских вводов, жесткого кодирования конфиденциальной информации или использования неэффективных алгоритмов. Хотя на последующих этапах будет проводиться углубленная проверка безопасности и производительности, выявление таких проблем на этапе первичной проверки позволит разработчикам немедленно потребовать исправления от ИИ или внести исправления вручную до приемки кода, что предотвратит распространение проблем на последующие этапы процесса.

В любом случае, код можно утверждать только после проверки всех этапов и отсутствия явных проблем.

2. Приглашение рецензентов

Помимо авторов кода, в коммерческих проектах обычно привлекаются дополнительные стороны для совместной проверки кода. Строго говоря, рецензирование кода – это не технический, а управленческий процесс, и конкретные методы его проведения могут быть гибкими и разнообразными. Некоторые команды в зависимости от уровня детализации требований периодически организуют очные встречи по рецензированию кода, на которых члены команды собираются для обсуждения реализации изменений, и только после одобрения большинством участ-

ников изменения могут быть включены в основную версию. Другие команды по завершении разработки кода требуют создать Pull Requests (запросы на слияние) в системах GitLab или GitHub, и только после одобрения рецензентом эти изменения могут быть включены в основную версию, как показано на рис. 5-8.

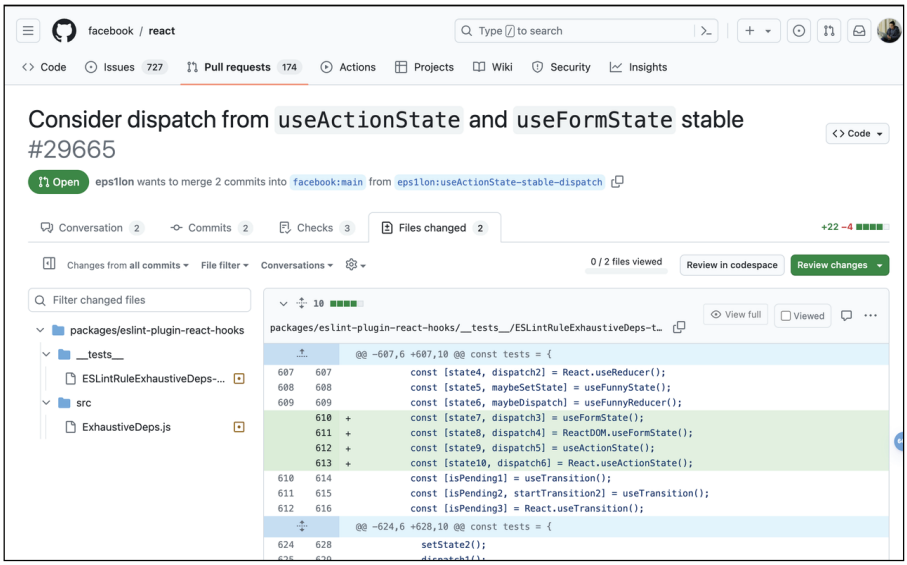


Рисунок 5-8. Пример функции Pull requests в GitHub

Независимо от выбранного способа проверки кода, в процессе должны участвовать квалифицированные сторонние специалисты, а сама проверка должна стать обязательным этапом рабочего процесса: код может быть интегрирован только после успешного прохождения проверки. Кто же является подходящим рецензентом? В реальных проектах существует множество конкретных критериев оценки, и все они различны, однако есть несколько полезных для ориентира аспектов.

- **Технический эксперт**: сотрудник команды с высоким уровнем технической компетенции, богатым опытом и высокими требованиями к качеству кода. Такой эксперт способен оценить преимущества и недостатки реализации кода, его читаемость и потенциальные риски с сугубо технической точки зрения. Например, он может выявить случаи повторного вычисления данных из-за отсутствия вызова useMemo; обнаружить проблемы качества в новых сторонних зависимостях и предложить более оптимальные решения; выявить повторяющиеся шаблоны кода в репозитории и инициировать их преобразование в универсальные базовые модули.
- **Бизнес-эксперт**: сотрудник команды с глубоким пониманием конкретных бизнес-правил. Обычно эту роль выполняет руководитель команды или ключевой специалист по конкретному направлению. Бизнес-эксперт может подтвердить с коммерческой точки зрения соответствие кода ожиданиям предприятия, в частности обоснованность обработки некоторых граничных сценариев. Например, проверка наличия обработки взаимодействий

в определенных аварийных сценариях; соответствие правил проверки полей форм бизнес-ограничениям; возможность повторного использования части логики кода из существующих модулей репозитория и т. д.

- **Эксперт по функциональному модулю**: сотрудник, наиболее хорошо знакомый с модулем, в который внесены изменения в данной версии. Как правило, это автор модуля или его основной разработчик. Технические и бизнес-эксперты умеют оценивать обоснованность изменений на макроуровне, но при переходе к конкретному модулю могут не дать точной обратной связи из-за отсутствия информации о предыстории. В этом случае эксперт по модулю может предоставить более обоснованные рекомендации благодаря знанию исторического контекста.

Обратите внимание, что эти три типа ролей представляют собой лишь приблизительную модель распределения обязанностей, границы между которыми не всегда четко очерчены, и в разных ситуациях роли часто могут меняться местами. Главное заключается в том, что при рецензировании кода мы стремимся привлекать как можно больше специалистов с разными навыками, чтобы провести обоснованную проверку кода с разных точек зрения и в конечном итоге получить более многогранные и конструктивные рекомендации, которые помогут повысить качество каждого изменения.

3. Формирование культуры рецензирования кода

Можно сказать, что рецензирование кода – это этап методологии вайб-программирования, требующий наиболее глубокого понимания кода. Проведение рецензирования кода позволяет своевременно выявлять такие проблемы, как нерациональный дизайн, неупорядоченная структура, нарушение архитектурных стандартов и дублирование кода, а также способствует эффективному техническому обмену между членами команды. Все эти преимущества широко признаны и приняты в отрасли. Поэтому многие команды готовы вкладывать время и усилия в разработку стандартов рецензирования кода. Однако, к сожалению, в большинстве команд, с которыми довелось поработать, выполнение этой задачи не давало ожидаемых результатов.

Причина проста: рецензирование кода приносит в основном долгосрочные, неочевидные и не поддающиеся количественной оценке выгоды, но требует значительных затрат времени и усилий здесь и сейчас. Проблемы, выявленные в ходе рецензирования кода, «возможно», могут повысить качество, но это также означает, что в данный момент необходимо потратить время на их исправление, что, в свою очередь, может повлиять на темпы запуска и даже привести к задержкам. Это типичная проблема возврата инвестиций: будущая выгода может проявиться только в будущем, что является неопределенным, в то время как текущие инвестиции являются реальными издержками. Так что при возникновении проблем с кадрами, временем или графиком рецензирование кода обязательно станет первым этапом, от которого придется отказаться.

Именно с этой проблемой сталкиваются многие команды разработчиков: как обеспечить строгое выполнение процедуры рецензирования кода членами команды и своевременное выявление проблем в коде? Проблема заключается в людях, и ключ к ее решению также лежит в людях. Помимо использования вайб-программирования для облегчения умственной нагрузки, следует

рассматривать такие характеристики, как читаемость, удобство поддержки, аккуратность и элегантность, в качестве ключевых требований к качеству, а не просто нагромождать код для удовлетворения функциональных потребностей. Если в команде более 20 % людей обладают такими качествами, как стремление к совершенству и перфекционизм в отношении кода, а также если культура команды достаточно открыта и толерантна, допускает и даже поощряет высказывание мнений по техническим вопросам, то общая техническая атмосфера будет достаточно сильной, и, следовательно, большее внимание будет уделяться требованиям к качеству, выходящим за рамки функциональных потребностей, а проверка кода будет выполняться с большей строгостью. Таким образом, если позволяют условия, следует по возможности привлекать в команду больше высококлассных инженеров и поощрять их активно высказывать свои сомнения; со временем это естественным образом сформирует строгую и прагматичную инженерную культуру.

4. Помощник по рецензированию на базе ИИ

Раз уж ИИ может генерировать код, то, естественно, он может и проверять его. В действительности в отрасли уже создано множество инструментов для этой цели.

CodeRabbit: платформа для проверки кода с помощью ИИ с поддержкой репозиториев. Она способна анализировать изменения в коде и предоставлять отзывы, подобные тем, что дают опытные инженеры, а также интегрировать эти отзывы в GitHub Pull Requests и локальные IDE. По заявлению разработчиков, CodeRabbit использует весь репозиторий кода и контекст информации для выявления мелких уязвимостей и улучшения практик. Кроме того, она поддерживает проверку с помощью ИИ в режиме реального времени в таких редакторах, как VS Code, что позволяет автоматически проверять каждую фиксацию и выявлять проблемы на ранней стадии.

Amazon CodeGuru: сервис проверки кода на основе машинного обучения компании Amazon. Сервис обучен на многолетнем опыте внутреннего кода и передовых практик Amazon, способен автоматически выполнять проверку кода и давать рекомендации по улучшению производительности приложения. В CodeGuru реализована функция анализа кода во время выполнения, позволяющая выявлять узкие места в производительности во время работы приложения.

Cursor: хотя Cursor в первую очередь ориентирован на повышение продуктивности разработчиков, его функции Q&A также позволяют проводить проверку кода.

Эти ИИ-помощники для проверки кода также работают на основе больших языковых моделей: изучая огромные объемы существующего кода и известные шаблоны неисправностей, они определяют наличие нарушений установленных передовых практик или типичных уязвимостей в новых фрагментах кода. По сути, это автоматические рецензенты, запомнившие правила, способные механически и с высокой скоростью сканировать код. Хотя эти ИИ-помощники по рецензированию не понимают истинного замысла и семантики кода, внедрение таких инструментов дает немало преимуществ.

- **Автоматизация и согласованность**: ИИ-инструменты могут автоматизировать большую часть рутинной работы по рецензированию, значительно снижая нагрузку на рецензентов. Они оценивают каждую строку

кода по единым стандартам, не допускают упущений из-за усталости или различий в стиле, тем самым обеспечивая единообразие кода.

- **Эффективный первичный анализ**: после предоставления кода ИИ-помощник может выступать в качестве фильтра первого уровня, быстро сканируя сотни строк кода для выявления распространенных проблем (таких как утечка ресурсов, базовые уязвимости безопасности или неэффективные алгоритмы), и способен выявлять эти проблемы быстрее и стабильнее, чем человек.
- **Снижение усталости при проверке**: поскольку базовые проблемы заранее выявляются машиной, человеку-рецензенту не нужно тратить драгоценное время на поиск простых ошибок, и он может сосредоточить свое внимание на более сложных и значимых аспектах, таких как архитектурные решения, поддержка кода и проверка бизнес-логики. ИИ отвечает за механическую проверку, а человек фокусируется на абстрактном мышлении. Такое разделение труда повышает общую эффективность и глубину проверки.

Тем не менее, как уже упоминалось в данном разделе, у больших языковых моделей есть собственные недостатки в плане генерации кода, и эти проблемы также проявляются в сценарии рецензирования кода.

- **Отсутствие понимания контекста**: ИИ не обладает глубоким пониманием бизнес-контекста и архитектурных замыслов в коде, поэтому не может, как человек, комплексно учитывать изначальные цели проектирования и бизнес-семантику. Это приводит к тому, что некоторые предложения по исправлениям, выданные ИИ, могут казаться разумными в локальном контексте, но на самом деле противоречат общему дизайну системы или требованиям продукта.
- **Ложные срабатывания и пропуски**: ограничения сопоставления шаблонов приводят к тому, что при проверке ИИ неизбежно возникают ложные срабатывания (ошибочная оценка правильного и безобидного кода как проблемного) и пропуски (упущение реально существующих дефектов). Ложные срабатывания отвлекают внимание разработчиков, заставляя тратить время на решение несуществующих проблем, что со временем может снизить доверие разработчиков к инструменту. Еще более серьезны пропуски: когда ИИ упускает скрытые ошибки или уязвимости, у команды разработчиков может возникнуть ложное чувство безопасности, и они могут ослабить бдительность, полагая, что код прошел проверку.
- **Проблемы бизнес-логики и интерпретации намерений**: многие проблемы качества кода связаны не с синтаксическими ошибками, а с целями проекта или бизнес-правилами. ИИ не может понимать, почему тот или иной бизнес-процесс должен быть реализован именно таким образом, и не может оценить архитектурные компромиссы. Поэтому ИИ может игнорировать код, который не нарушает общих правил, но на самом деле работает некорректно.
- **Риск чрезмерной зависимости**: если разработчики будут чрезмерно полагаться на проверку ИИ и бездумно принимать предложенные ИИ

изменения, это может негативно повлиять на качество проверки всей команды. В конце концов, ИИ не является авторитетным арбитром, его рекомендации иногда могут быть ошибочными или ограниченными, поэтому разработчикам важно сохранять здоровый скептицизм. Если разработчики привыкнут не проверять выводы ИИ, то в случае ошибки в его оценке проблема попадет прямо в репозиторий кода.

Не зная внутренних деталей генерируемого ИИ кода, разработчики, полагающиеся при рецензировании исключительно на ИИ, создают замкнутый цикл с чрезвычайно высоким потенциальным риском: код генерируется ИИ, рецензирование также поручается ИИ, а человек отвечает лишь за нажатие кнопки, не вникая в суть работы кода. В такой ситуации, если ИИ упустит ключевую проблему, последствия могут быть неутешительными. Чем больше кода генерирует ИИ, тем больше требуется высококачественная проверка со стороны человека для обеспечения надежности ПО. Если позволить ИИ выступать и в роли создателя, и в роли проверяющего, это будет равносильно отсутствию необходимой страховки в виде человека-эксперта. Только глубокое вовлечение рецензента-человека может разорвать этот замкнутый круг, предлагая отсутствующие у ИИ рациональное суждение и критический подход.

5.4. Инженерный подход

Вайб-программирование – это метод разработки ПО, в значительной степени полагающийся на ИИ, при котором скорость ИИ-генерации кода значительно превышает возможности ручной проверки. Однако в реальных коммерческих проектах вайб-программирование пока не обходится без ручной проверки. Это приводит к тому, что под давлением дедлайнов инженеры склонны пропускать этап проверки, чем создают множество потенциальных проблем для проекта.

- **Качество и надежность кода**: ИИ может генерировать «галлюцинации» в виде кода, который на первый взгляд кажется корректным, но на самом деле имеет скрытые ошибки, низкую эффективность или семантические искажения.
- **Высокие затраты на отладку и обслуживание**: недостаточное понимание генерируемого ИИ кода со стороны человека приводит к сложности выявления проблем на поздних этапах и резкому росту затрат на их устранение.
- **Угрозы безопасности**: код, не прошедший тестирование на безопасность, может содержать потенциальные уязвимости, что увеличивает системные риски.

Основные преимущества вайб-программирования заключаются в низком пороге входа и скорости разработки, но именно по этой причине особенно важно использовать инженерные методы для снижения рисков. В противном случае удобство вайб-программирования превратится в долгосрочную угрозу в плане сложности обслуживания и стабильности проекта. Чтобы вайб-программирование действительно прижилось, важно внедрить его в инженерную систему с четкой структурой, контролируемым процессом и возможностью проверки.

5.4.1. Введение в инженерию

Программная инженерия – это дисциплина, применяющая инженерные принципы для проектирования, разработки, тестирования и обслуживания ПО. Она использует систематизированные, научные и стандартизированные методы для решения сложных задач в процессе разработки ПО, обеспечивая высокое качество и надежность конечного продукта. В эпоху вайб-программирования значение инженерного подхода только усилилось: благодаря автоматизации с его помощью можно быстро проверять качество кода и обеспечить долгосрочную поддержку, что, в свою очередь, еще больше снижает зависимость процесса разработки от человеческого фактора.

В частности, инженерная система позволяет эффективно решать следующие ключевые задачи.

- **Обеспечение корректности и надежности кода**: в случае «галлюцинаций» или других потенциальных недостатков, которые могут возникать при ИИ-генерации кода, инженерные процессы проводят строгую проверку с помощью автоматизированного тестирования, статического анализа кода и инструментов рецензирования, гарантируя корректность и надежность кода.
- **Повышение удобства чтения и обслуживания**: генерируемый ИИ код может быть недостаточно понятным и слабо согласованным. Инженерные практики позволяют обеспечить единый стиль кода, структурированные комментарии и подробную документацию. Это помогает разработчикам глубже понять логику и концепцию генерируемого ИИ кода, что облегчает последующую отладку и сопровождение.
- **Повышение безопасности**: в инженерную систему обычно встроены меры безопасности, такие как автоматическое сканирование, статический анализ безопасности, выявление рисков и рецензирование кода. Это помогает выявлять потенциальные уязвимости на ранней стадии и обеспечивает стабильность и безопасность производственной среды.

Разработка ПО не ограничивается исключительно написанием кода. Само по себе кодирование составляет лишь небольшую часть общего объема работ по проекту. Хотя ИИ может эффективно решать задачи по генерации кода, существуют более обширные аспекты, такие как проверка требований, контроль качества, развертывание систем и мониторинг производственной среды, которые являются ключевыми этапами в корпоративных проектах по разработке ПО, требующими тщательного анализа и проектирования. Инженерные подходы позволяют эффективно решать сложные задачи в таких сценариях.

Можно сказать, что преимущества вайб-программирования без инженерной поддержки проявляются только в краткосрочных или небольших проектах. А с помощью инженерной системы подход к разработке ПО на основе ИИ может перейти от быстрых экспериментов к долгосрочным стабильным релизам корпоративного уровня, по-настоящему раскрывая свой огромный потенциал в качестве акселератора производительности.

5.4.2. Легкая инженерная система для вайб-программирования

Несмотря на неоценимую важность инженерного подхода для обеспечения качества и долгосрочной стабильности ПО, полноценная инженерная система сама по себе является обширной и сложной темой, выходящей далеко за рамки этой книги. Поэтому в данном разделе будет представлен набор простых, облегченных и готовых к практическому внедрению инженерных решений, которые читатель может рассматривать в качестве минимального набора мер для обеспечения необходимого качества, позволяющего с минимальными затратами добиться эффективного контроля качества и удобства обслуживания.

Управление версиями

Система контроля версий (Version Control) – это инструмент для ведения и управления историей изменений кода, также известный как **система управления исходным кодом** (Source Code Management) или **система контроля ревизий** (Revision Control). Система контроля версий обеспечивает управление версиями различных файлов в процессе разработки ПО (таких как код, документация, данные, изображения) и помогает эффективно устранять конфликты при совместной работе над кодом. В программной инженерии система контроля версий выполняет следующие функции.

- **Безопасный откат**: система контроля версий работает как страховка, позволяя разработчикам в любой момент вернуться к любой версии исторического архива. Если генерируемый ИИ код содержит ошибки или нестабильные элементы, можно оперативно восстановить предыдущую надежную версию и далее без опасений вносить изменения и проводить итерации без риска необратимых сбоев.
- **Эффективное совместное использование**: механизм ветвления инструментов управления версиями (таких как Git) позволяет нескольким разработчикам одновременно и независимо друг от друга разрабатывать новые функции или исправлять проблемы, а затем, после проверки кода, объединять изменения в основной ветке в упорядоченном виде, тем самым избегая конфликтов при разработке и проблем с наложением.
- **Основа практик DevOps**: управление версиями является основой современных процессов DevOps. Без поддержки управления версиями такие практики DevOps, как автоматическая сборка, CI/CD и т. д., будут неэффективны.

Приведем практический пример. Допустим, проект успешно запущен и находится в фазе стабильной работы. В этот момент вы с помощью ИИ сгенерировали новый код и провели несколько итераций, но случайно внесли скрытую ошибку, которую не удалось вовремя обнаружить в ходе ручной проверки. В результате проблема попала в финальную версию. Если в проекте не используется контроль версий, вам придется потратить много времени на точную локализацию проблемы в коде, ее исправление и повторный релиз. Весь этот процесс отнимает много времени и сил, а проблема при этом продолжает существовать в производственной среде. Но если вы уже используете систему

контроля версий, вы можете быстро вернуться к предыдущей стабильной версии с помощью команд git revert или git reset, немедленно минимизировать влияние проблемы на рабочую среду, а затем асинхронно выявить и исправить проблему, что значительно сокращает убытки для бизнеса.

Применение системы контроля версий уже сегодня является одним из обязательных навыков программиста. Ниже приведены часто используемые команды Git.

- **Установка Git**. В операционных системах macOS или Linux, как правило, Git уже предустановлен. Если он не установлен, его можно инсталлировать с помощью Homebrew (команда `brew install git`) или системного менеджера пакетов. Пользователи Windows могут скачать установочный пакет с официального сайта Git и инсталлировать его.
- **Настройка пользовательских данных**. После установки задайте имя пользователя и адрес электронной почты для идентификации автора изменений в коде:

```
git config --global user.name "ваше имя"
git config --global user.email "ваше имя@example.com"
```

- **Инициализация репозитория**. В каталоге проекта для создания нового репозитория Git выполните следующую команду:

```
git init
```

- **Проверка состояния.** Просмотрите изменения в проекте и определите, какие файлы необходимо добавить в стабильный репозиторий, зафиксировать или игнорировать:

```
git status
```

- **Добавьте файлы в область стабильного хранения**. Обозначьте файлы как часть следующей фиксации:

```
git add [имя файла]
git add . # Добавьте все измененные файлы в текущем каталоге
```

- **Зафиксируйте изменения**. Сохраните моментальный снимок проекта и опишите внесенные изменения с помощью содержательного сообщения о фиксации:

```
git commit -m «ваше сообщение о фиксации»
```

- **Просмотрите историю**. Ознакомьтесь с историей фиксаций, чтобы понять ход работы над проектом:

```
git log
```

- **Работа с ветками**: одна из самых мощных функций Git – это работа с ветками. Ветки дают пользователям возможность независимо работать над разными частями проекта, не затрагивая основной репозиторий кода. Это особенно полезно при разработке новых функций или исправлении ошибок. Ниже приведены наиболее распространенные команды.

- Создание новой ветки: `git branch` [имя ветки].
- Переключение ветки: `git checkout` [имя ветки].
- Слияние ветки: `git merge` [имя исходной ветки].
- Сравнение ветки: git diff branch1 branch2.

В современной среде разработки с ИИ вам не нужно механически запоминать все команды Git: просто сообщите ИИ свои требования, и он точно сгенерирует код и выполнит соответствующие операционные инструкции. Например, в Cursor можно использовать сочетание клавиш **Ctrl** + `` ` `` для открытия терминала, потом нажать **Ctrl** + **k** для запуска AI Chat, ввести запрос (например, "Переключиться на ветку: *chore/test*, добавить все текущие изменения в стабильный репозиторий, затем зафиксировать код и отправить на удаленный сервер"), как показано на рис. 5-9[1].



Рисунок 5-9. Взаимодействие с ИИ в терминале

Cursor автоматически сгенерирует Git-команды, соответствующие данному запросу:

```
git checkout -b chore/test && git add . && git commit -m "chore: инициализация
тестовой ветки" && git push origin chore/test
```

Подытоживая, можно сказать, что разумное использование системы контроля версий позволяет лучше организовать совместную разработку и обеспечить безопасность кода, поэтому рекомендуется включить ее в стандартный процесс разработки.

2. Автоматическое тестирование

Система контроля версий дает возможность в любой момент вернуться к предыдущим версиям, но для быстрого подтверждения стабильности кода также необходим механизм автоматического тестирования, который гарантирует, что новый генерируемый ИИ код не нарушит нормальную работу уже существующих функций, обеспечит качество, стабильность и удобство обслуживания кода и в конечном итоге позволит обеспечить его надежную отправку пользователям. Для этого необходимо внедрить технологии автоматического тестирования.

---

[1] Надпись в красной рамке на рисунке гласит: «Переключиться на ветку: *chore/test*, добавить все предыдущие изменения в стек Git, затем зафиксировать код и отправить его на удаленный сервер». – *Прим. перев.*

Автоматическое тестирование (Automated Testing) – это практика, при которой с помощью написания тестовых скриптов или тестовых примеров автоматически проверяется соответствие функций ПО заявленным требованиям. Автоматическое тестирование имеет следующие преимущества по сравнению с ручным.

- **Гарантия корректности функций**: автоматическое тестирование позволяет эффективно проверять поведение функций в различных условиях, обеспечивая целостность и корректность логики кода.
- **Быстрое выявление и устранение проблем**: в процессе разработки автоматическое тестирование позволяет быстро выявить первопричину проблемы, что значительно сокращает временные затраты на ручную отладку.
- **Мгновенная обратная связь**: если изменения в коде приводят к появлению проблем, автоматическое тестирование может быстро их обнаружить и сообщить об этом, помогая разработчикам оперативно исправлять ошибки и предотвращать дальнейшее распространение проблем.

Комбинируя автоматические тесты с генерируемым ИИ кодом, можно создать эффективный и надежный контур обратной связи в разработке, значительно сократить объем ручного тестирования и отладки, а также быстро проверить стабильность и корректность генерируемого ИИ кода, что в итоге ведет к быстрым и надежным итерациям.

Для разных языков существуют разные инструменты тестирования. Их сложно перечислить, но с помощью ИИ можно быстро создать среду автоматического тестирования для различных стеков разработки и написать тестовые наборы. Например, для автоматизации тестирования в стеке технологий JavaScript с помощью Vitest можно следовать следующему процессу.

(1) **Инициализация среды**: для быстрой инициализации среды автоматизированного тестирования Vitest можно использовать следующий промпт:

```
Это проект на JavaScript. Необходимо настроить среду автоматизированного
тестирования Vitest, которая должна поддерживать тестирование кода JavaScript
и React.
```

ИИ автоматически изменит проектный файл package.json, добавит соответствующие зависимые библиотеки и сгенерирует необходимые конфигурационные файлы. Необходимо лишь выполнить команду инициализации (например, `npm install`), чтобы быстро завершить настройку среды.

(2) **Добавление тестовых случаев**: после завершения настройки среды можно продолжить использование ИИ для генерации тестовых случаев для конкретных модулей:

```
Выполнить подробный анализ кода @xxx.ts и сгенерировать для него тестовые
случаи. Требуется покрытие модульных тестов не менее 80 %, необходимо
имитировать все модули нижестоящего уровня и обеспечить независимость тестовых
случаев
```

В идеальном случае ИИ автоматически сгенерирует соответствующий код модульных тестов, и разработчику достаточно выполнить команду `npx vitest --run` для завершения автоматического тестирования.

Однако процесс автоматического создания тестового кода не всегда проходит гладко: сложность исходного кода находится в прямой зависимости от сложности генерации модульных тестов. Чем проще исходный код, тем понятнее его логическая структура и реализация функций и тем проще сгенерировать модульные тесты. Чем сложнее исходный код, чем больше ветвей и чем сложнее связанные с ним нижестоящие модули, тем сложнее становятся граничные условия, обработка исключений и логика взаимодействия, что требует большего количества и более сложных тестовых случаев, а сложность генерации модульных тестов ИИ соответственно возрастает.

В связи с этим, если ИИ не может успешно сгенерировать модульные тесты, можно предварительно оптимизировать структуру исходного кода, снизив его сложность:

```
Исходный код модуля @xxx.ts имеет чрезмерную сложность, что не позволяет
правильно сгенерировать модульные тесты. Необходимо разбить модуль
в соответствии с принципами «один модуль – одна задача», «низкая связность»
и «высокая однородность», чтобы обеспечить удобство для модульных тестов.
```

Это позволит улучшить структуру кода и упростить тестирование модулей, а также повысить эффективность генерации тестовых наборов с помощью ИИ, что в конечном итоге обеспечит более быстрый и надежный процесс разработки ПО.

3. CI/CD

Непрерывная интеграция и непрерывная доставка/развертывание (Continuous Integration/Continuous Delivery, CI/CD) – это набор методов разработки и тестирования ПО, реализуемых с помощью автоматизированной конвейерной системы. При выполнении определенных условий она автоматически выполняет такие действия по обеспечению качества, как интеграция кода, сборка, тестирование и развертывание. Только после успешного прохождения всех автоматических проверок проект может перейти к следующему этапу. Значение терминов «непрерывная интеграция» (CI) и «непрерывная доставка/развертывание» (CD) приведено ниже.

- **Непрерывная интеграция**: перед слиянием каждой ветки с основой выполняются сборка, модульное тестирование, статическая проверка кода и другие операции, обеспечивающие стабильность и качество основной ветки.
- **Непрерывная доставка/развертывание**: запускается автоматически после внесения изменений в основную ветку и включает в себя более строгие проверки качества, такие как сборка, тестирование и проверка производительности, что в конечном итоге гарантирует стабильность кода для развертывания в производственной среде.

Основная цель CI/CD состоит в повышении эффективности поставляемого ПО за счет автоматизации инструментов контроля качества, сокращения риска человеческих ошибок и обеспечения быстрой и надежной реализации ПО. Это позволяет быстрее реагировать на запросы рынка и пожелания пользователей, а также ускорить общий ритм итераций проекта. В контексте вайб-

программирования гибкий процесс CI/CD позволяет более эффективно проверять качество генерируемого ИИ кода, ускорить скорость внедрения и снизить риски итераций.

На практическом уровне существует несколько систем, предоставляющих полный набор возможностей CI/CD. В качестве примера возьмем GitHub Actions, где мы обычно проектируем следующие две конвейерные линии.

- **Конвейер непрерывной интеграции**: запускается при каждом запросе на слияние. Выполняет базовые операции, такие как установка зависимостей, сборка кода, модульное тестирование и статическая проверка кода. Только после прохождения всех проверок качества код может быть включен в основную ветвь. Обратите внимание: в крупных командах во избежание чрезмерного удлинения конвейера (обычно его продолжительность ограничивается 10 минутами) на этом этапе рекомендуется использовать относительно «легкие» правила проверки качества. Пример подсказки:

```
Добавить конвейер GitHub Actions, который запускается при каждом создании
PR, выполняет установку зависимостей, сборку кода, модульное тестирование
и статическую проверку кода. Только после прохождения всех проверок код может
быть включен в основную версию.
```

- **Конвейер непрерывной доставки**: запускается при изменении главной ветки. По сравнению с этапом «непрерывной интеграции» здесь выполняется более полное тестирование. После прохождения проверки качества вызывается интерфейс облачной платформы для развертывания и запуска. Конкретные варианты реализации этого этапа будут описаны в разделе 6.5, поэтому здесь они не будут рассматриваться более подробно.

4. Создание документации с помощью ИИ

Помимо перечисленного выше, в долгосрочной перспективе необходимо, с точки зрения удобства обслуживания, одновременно с завершением кода дополнять документацию с описанием кода. С одной стороны, это способствует повышению читаемости кода, позволяя последующим пользователям (или будущим разработчикам) быстрее понять логику кода. С другой стороны, документация помогает ИИ более точно распознавать структуру и функции кода, что, в свою очередь, позволяет более эффективно генерировать новый код.

С точки зрения программной инженерии, ПО включает в себя не только исходный код, но и сопутствующую документацию. К этой категории относятся традиционные руководства по эксплуатации программы, файлы README в проектах с открытым исходным кодом и руководства для разработчиков в корпоративных проектах. Четкая и полная система документации выполняет следующие функции.

Объяснение функций и способов использования: подробное объяснение функций ПО и способов его использования, помогающее разработчикам и пользователям лучше понять общую архитектуру проекта и замысел его разработчиков.

Снижение затрат на обучение новых членов команды: сокращение времени обучения и упрощение процесса адаптации новых членов команды, что позволяет быстро повысить их производительность.

Предоставление истории проекта: документация фиксирует принятые решения, концепции проектирования и детали реализации, что облегчает будущее обслуживание и проверку, эффективно предотвращает повторение прошлых ошибок и, в частности, помогает ИИ быстро восстанавливать исторический контекст и поддерживать логическую связность.

Упрощение обслуживания и предотвращение дефектов: высокое качество документации значительно снижает затраты на обслуживание кода и позволяет эффективно предотвращать появление дефектов.

Безусловно, не все содержимое документации необходимо создавать вручную. В среде вайб-программирования для быстрого создания, обслуживания и обновления документации можно эффективно использовать ИИ. Например:

```
Ты являешься профессиональным инженером по технической документации, обладаешь
хорошим опытом в области разработки ПО и четкими коммуникативными навыками.
На основе приведенной ниже информации составь для одного из программных проектов
исчерпывающий и четко структурированный проектный документ для использования
разработчиками и пользователями, написанный лаконичным и точным языком.

Убедись, что документ охватывает следующие 4 аспекта, каждый из которых должен
содержать соответствующие разделы. При необходимости можно добавить примеры
кода или иллюстрации.

---

## Техническая документация (Technical Documentation)

- Введение в проект и описание контекста (задачи проекта, целевая аудитория,
  основные функции)
- Описание основных алгоритмов или ключевых реализаций (например, стратегии
  синхронизации данных, алгоритмы поиска путей)
- Проект API и примеры вызова (включая описание параметров, возвращаемые
  значения, обработку ошибок и т. д.)
- Описание структур данных (например, определение узлов дерева, модель
  интерфейса)

## Системная документация (System Documentation)

- Обзор системной архитектуры (текст + схема архитектуры)
- Обязанности и взаимосвязи между модулями/сервисами (например, фронтенд,
  бэкенд, база данных, очереди сообщений)
- Выбор технологий (например, использование React/Vue, Node.js, Redis и т. д.,
  а также обоснование этого выбора)
- Оптимизация производительности и меры безопасности (стратегия кеширования,
  ограничение пропускной способности, схемы аутентификации и т. д.)

## Документация для разработчиков (Developer Documentation)

- Установка и настройка среды разработки (версия Node, установка зависимостей,
  настройки .env и т. д.)
- Описание структуры каталогов кода (src, components, api, tests и т. д.)
- Процесс локальной разработки и отладки
```

```
- Стандарты внесения изменений (Conventional Commits) и стратегия управления
  ветками (например, Git Flow)
- Фреймворки для модульного и интеграционного тестирования (например, Vitest,
  Jest, Playwright)

## Документация для пользователей (User Documentation)

- Краткое руководство по началу работы (как запустить проект, как войти
  в систему / зарегистрироваться / использовать основные функции)
- Иллюстрированное руководство по эксплуатации (с описанием основных процессов
  взаимодействия с помощью скриншотов интерфейса)
- Инструкции по настройке и установке (например, системные требования,
  совместимость с браузерами, локальное развертывание)
- Часто задаваемые вопросы (FAQ)

---

**Дополнительные требования:**

- Все блоки кода должны быть корректно размечены в формате Markdown
- Стиль документации должен быть единообразным и подходить для публикации
  в README на GitHub
- При необходимости указания пути или командной строки используй формат
  разметки блоков кода: ```

**Примеры:**

- Быстрый старт:

```bash
git clone https://github.com/yourname/yourproject.git
cd yourproject
npm install
npm run dev
```

- Пример API:

```js
GET /api/user?id=123

Ответ:
{
  "id": 123,
  "name": "Alice"
}
```
```

Теперь мы можем эффективно создавать документацию с высоким качеством и постоянными обновлениями в режиме реального времени, что позволяет еще больше снизить затраты на долгосрочное обслуживание проекта. Ниже приведен фрагмент сгенерированной документации.

```
# Генератор контента на базе ИИ XiaoHongShu

## 🎯 **Основные инновации и технические особенности**

### 💡 **Интеллектуальная система создания контента на базе ИИ**
- **Стратегия объединения нескольких моделей**: поддержка динамического
  переключения между OpenAI GPT-4o, Gemini Pro и Claude-3 на основе единого
  абстрактного уровня Vercel AI SDK
- **Движок семантического анализа**: реализует сквозной конвейер семантического
  разбиения абзацев, извлечения ключевой информации и анализа тональности
- **Алгоритм переноса стиля**: 6 предварительно обученных стилевых
  моделей (Xiaohongshu/Профессиональный/Научный/Литературный/Свободный/
  Минималистичный), поддержка переноса стиля без обучающих данных

###   **Движок рендеринга Pure SVG**
- **Векторный рендеринг без привязки**: без использования Canvas/WebGL, чистый SVG
  + CSS для генерации высококачественных карточек (стандарт Xiaohongshu 960×1280)
- **Адаптивная система дизайна**: адаптивная верстка на основе CSS Grid +
  Flexbox, поддержка динамического масштабирования с золотым соотношением 3:4
- **Конвейер предварительного просмотра в реальном времени**: WebSocket
  + Server-Sent Events для обратной связи по рендерингу с точностью до
  миллисекунд

### ⚡ **Архитектура корпоративного уровня**
- **Монорепо в стиле Nx**: pnpm workspace + TypeScript Project References для
  совместного использования кода и обеспечения типовой безопасности
- **Микросервисная архитектура**: разделение Express Router и Service Layer,
  поддержка горизонтального масштабирования и независимого развертывания
- **Система мониторинга**: комплексное решение для управления
  производительностью приложения, включающее структурированные логи, мониторинг
  производительности и отслеживание ошибок

### 🔄 **Потоковая передача и работа в реальном времени**
- **Потоковое создание контента**: реализация фрагментированной обработки на
  основе ReadableStream API, сокращение времени загрузки первой страницы на 60 %+
- **Интеллектуальная стратегия кеширования**: двухуровневое кеширование Redis +
  Memory Cache, время отклика API < 100 мс
- **Механизм восстановления после ошибок**: Circuit Breaker + Retry with
  Exponential Backoff для обеспечения стабильности сервиса


## 📋 **Быстрая навигация**

|   [Техническая документация](#-техническая-документация) |   [Архитектура
системы](#-архитектура-системы) | 💻 [Руководство для разработчиков]
(#-документация-для-разработчиков) | 📖 [Руководство пользователя]
(#-документация-для-пользователей) |
|:---:|:---:|:---:|:---:|
| Проектирование API · Реализация алгоритмов · Структуры данных |
Проектирование архитектуры · Взаимосвязь модулей · Выбор технологий | Настройка
среды · Процесс разработки · Тестовая среда | Быстрый старт · Руководство по
эксплуатации · Ответы на вопросы |

---
```

```
## 🚀 Введение в проект

### Цель проекта
Генератор контента для Xiaohongshu - это интеллектуальный инструмент
для создания контента на основе ИИ, призванный помочь авторам быстро
преобразовывать длинные статьи в визуальные карточки, подходящие для платформы
Xiaohongshu. Благодаря ИИ-технологиям, которые автоматически анализируют
и извлекают ключевую информацию, а затем генерируют визуальные карточки,
эффективность создания контента значительно повышается.

...
```

5.5. Заключение

В этом разделе представлена комплексная практическая методика решения инженерных задач в рамках модели вайб-программирования – от методов разработки промптов и структурированного планирования требований до рецензирования и оптимизации кода. Эта методика гарантирует, что генерируемый ИИ код не только будет «работать», но также сохранит долгосрочную поддержку.

Кроме того, в данной главе представлены такие облегченные инженерные практики, как системы контроля версий, автоматическое тестирование, CI/CD и создание документации с помощью ИИ. Эти практики позволяют обеспечить высокое качество результатов при минимальных затратах. Они создают стабильную инфраструктуру для вайб-программирования, благодаря чему разработка с помощью ИИ может использоваться и для быстрого создания прототипов, и для поддержки корпоративных проектов, предназначенных для производственной среды. Только при условии применения надежных инженерных методов можно в полной мере раскрыть преимущества ИИ в области эффективности разработки.

Глава 6

Практические примеры

В этой главе на примере проекта «Генератор контента для Xiaohongshu» подробно рассказывается обо всех этапах создания полнофункционального ИИ-приложения с реальной практической ценностью на основе методологии вайб-программирования. Материал главы охватывает полный цикл работ: от настройки среды, анализа требований к проекту, разработки бэкенда и фронтенда до финального развертывания. В ходе описания каждого этапа уделяется особое внимание ключевым шагам и приводятся ценные рекомендации по построению промптов для взаимодействия моделей, а также по отладке и организации рабочего процесса.

В отличие от традиционного процесса разработки, в этой главе делается акцент на том, как создать среду разработки, ориентированную на ИИ, где ведущую роль играют разработчики-люди, а ИИ используется для более эффективной разработки приложений. Весь соответствующий код уже загружен в репозиторий по адресу https://github.com/Tecvan-fe/xiaohongshu-generator, и читатели могут скачать его самостоятельно для дальнейшего изучения.

6.1. Настройка среды

Хороший мастер всегда начинает с подготовки инструментов. Перед началом работы над проектом необходимо настроить ряд базовых компонентов среды разработки, включая Cursor (редактор кода со встроенной поддержкой ИИ), Node.js (среду выполнения JavaScript), pnpm (эффективный менеджер пакетов), учетную запись ChatGPT, токен API OpenAI и так далее, а также создать среду разработки проекта.

Для упрощения задачи в данном проекте выбран язык с низким порогом входа – JavaScript, при этом основное внимание уделяется изучению различных подсказок и приемов, используемых в процессе разработки. Читатели, использующие другие технологические стеки, не столкнутся с трудностями при ознакомлении с этим материалом. Читатели, у которых уже установлены соответствующие инструменты, могут пропустить раздел 6.1.1 и сразу перейти к разделу 6.1.2, посвященному описанию структуры проекта.

6.1.1. Подготовка инструментов

Для запуска примера проекта следует предварительно установить несколько необходимых программ для разработки и выполнить базовую настройку.

1. Установка и настройка Cursor

Cursor – это редактор кода, изначально разработанный для ИИ на базе Visual Studio Code (VS Code). Он позволяет напрямую интегрировать возможности генеративного ИИ в рабочий процесс разработки и предоставляет такие функции, как контекстные подсказки по коду, автоматическая рефакторизация и генерация кода в реальном времени. В настоящее время Cursor считается одной из самых эффективных ИИ-сред разработки (IDE) на рынке, и в этой книге для вайб-программирования рекомендуется использовать именно этот инструмент.

Процесс установки Cursor довольно прост. Сначала перейдите на официальный сайт Cursor (как показано на рис. 6-1), затем нажмите кнопку **Download** (загрузка). Программа установки, подходящая для операционной системы вашего компьютера, загрузится автоматически. По завершении загрузки запустите программу установки и дождитесь окончания процесса инсталляции.

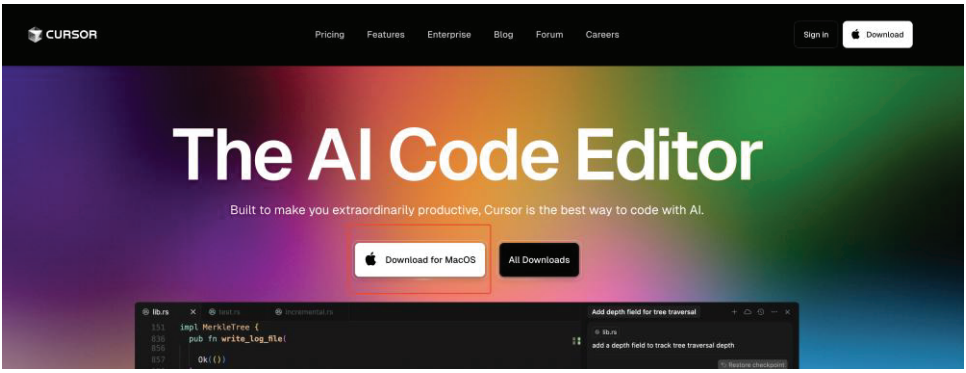


Рисунок 6-1. Официальный сайт Cursor

Способы установки для основных операционных систем приведены в табл. 6-1.

Таблица 6-1. Способы установки Cursor для разных операционных систем

Операционная система	Способ загрузки / Тип файла	Основные шаги установки (краткое описание)	Пример команды (если есть)
Windows	Файл *.exe*	Запустите загруженный файл *.exe*, следуя инструкциям мастера установки	Нет
macOS	Файл *.dmg*	Дважды щелкните файл *.dmg* и перетащите приложение в каталог «Приложения»	Нет
Linux	Файл *.AppImage*	Сделайте файл *.AppImage* исполняемым, а затем запустите его	chmod +x .Cursor-version.AppImage ./Cursor-version.AppImage

После завершения установки необходимо выполнить дополнительные настройки для дальнейшего повышения эффективности Cursor.

- **Пополните счет и активируйте подписку**. В бесплатной версии предусмотрен пробный лимит, однако ее производительность ниже, чем у версии Pro. Даже при разнице в производительности всего в 10 % в условиях мультипликативного эффекта разница в эффективности будет весьма значительной, поэтому рекомендуется сразу использовать версию Pro.
- **Включите режим конфиденциальности**. Многие операции обработки моделей внутри Cursor происходят в облаке, поэтому неизбежно возникает необходимость отправлять локальный код на серверы Cursor. Если вас беспокоят риски, связанные с соблюдением нормативных требований, вы можете включить режим конфиденциальности. В этом режиме Cursor гарантирует, что ваш код не будет сохраняться без вашего ведома. Чтобы включить этот режим, нажмите кнопку настроек в правом верхнем углу (⚙), в диалоговом окне **Cursor Settings** (Настройки) выберите вкладку **General** (Главная) и установите для параметра **Privacy Mode** (Приватный режим) в значение **Enabled** (Включено), как показано на рис. 6-2.

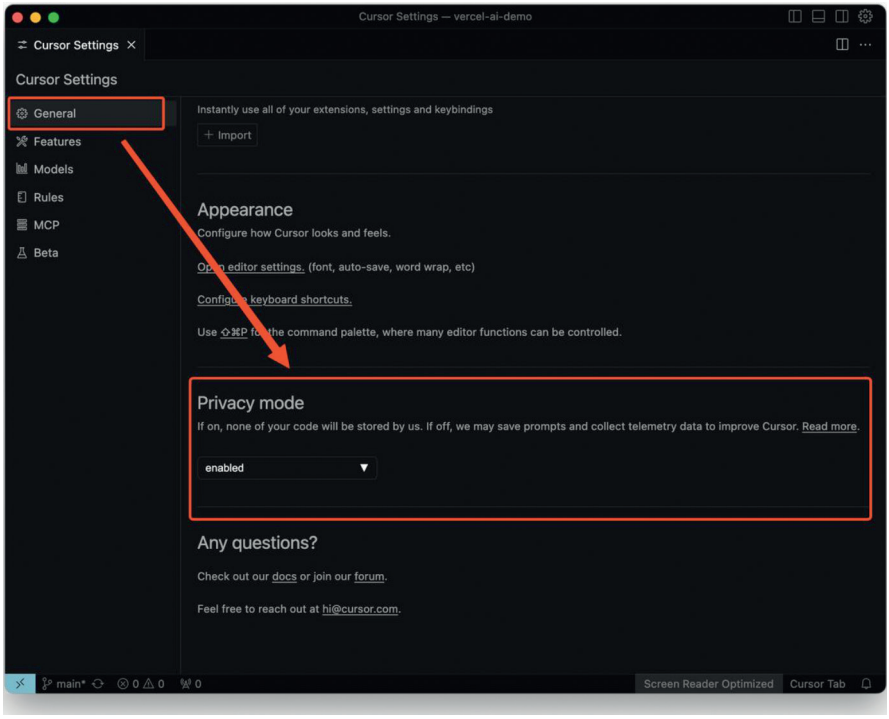


Рисунок 6-2. Включение режима конфиденциальности

- **Включение режима автозапуска**. Во время работы Cursor Agent может потребоваться запуск ряда локальных командных инструментов, таких как npm, Vitest и т. д. Из соображений безопасности Cursor по умолчанию не запускает такие программы самостоятельно, а выводит диалоговое окно, где пользователь может принять решение о необходимом действии. Для оптимизации работы рекомендуется настроить автоматический запуск

некоторых инструментов с низким уровнем риска, дабы Cursor мог запускать их напрямую, минуя диалоговое окно. Чтобы включить этот режим, нажмите кнопку настроек (⚙) в правом верхнем углу, в диалоговом окне **Cursor Settings** (Настройки) выберите **Features** (Функции), перейдите в интерфейс настройки функций и установите флажок под пунктом **Enable auto-run mode** (Включить режим автозапуска), как показано на рис. 6-3.

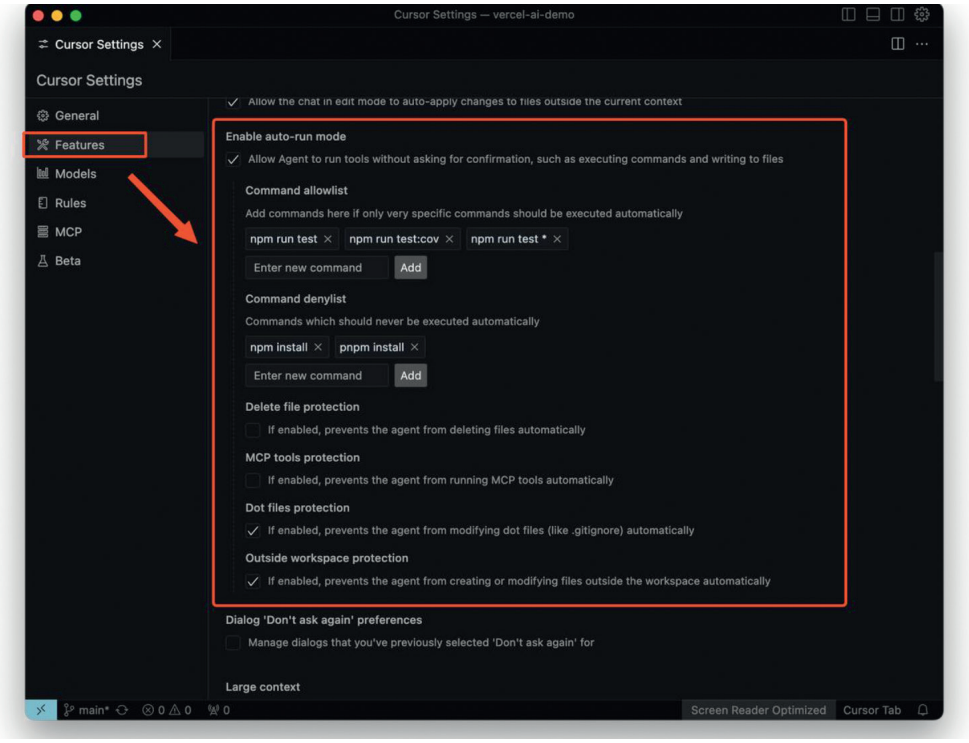


Рисунок 6-3. Включение режима автозапуска

Белый список разрешенных команд (Command allowlist): фронтенд-разработчикам рекомендуется добавить в него такие распространенные инструменты, как npm, pnpm, Vitest, eslint, webpack и т. д.

Включите индексирование кодовой базы (Codebase Indexing). Индексирование кодовой базы – одна из ключевых функций Cursor. Проще говоря, эта функция выполняет векторизацию всего содержимого проекта и шифрует его для хранения в облачной базе данных Cursor. При последующих запросах к большим языковым моделям можно извлечь векторные знания из этой базы данных, благодаря чему формируется более глубокое понимание кода репозитория, а генерируемый Cursor код будет более тесно связан с существующим кодом (например, схожий стек технологий, стиль кодирования и т. д.). Чтобы включить эту функцию, нажмите кнопку настроек в правом верхнем углу (⚙), в диалоговом окне **Cursor Settings** нажмите **Features**, перейдите в интерфейс настроек функций, прокрутите вниз, найдите пункт **Codebase Indexing** и включите опцию **Enabled**, как показано на рис. 6-4.

Отключите функцию Auto. Начиная с версии Cursor 0.47 появилась возможность автоматического выбора подходящей модели для выполнения задачи в зависимости от ее типа во время сеанса Agent или Chat, однако практические тесты показали, что эта функция работает не очень хорошо. Поэтому для конкретных сценариев рекомендуется выбирать фиксированную модель. Например, для задач, связанных с программированием, рекомендуется выбирать claude-4-sonnet, для задач общего характера – gpt-4o, а для задач по написанию текстов на китайском языке – deepseek и т. д. Способ переключения модели показан на рис. 6-5.

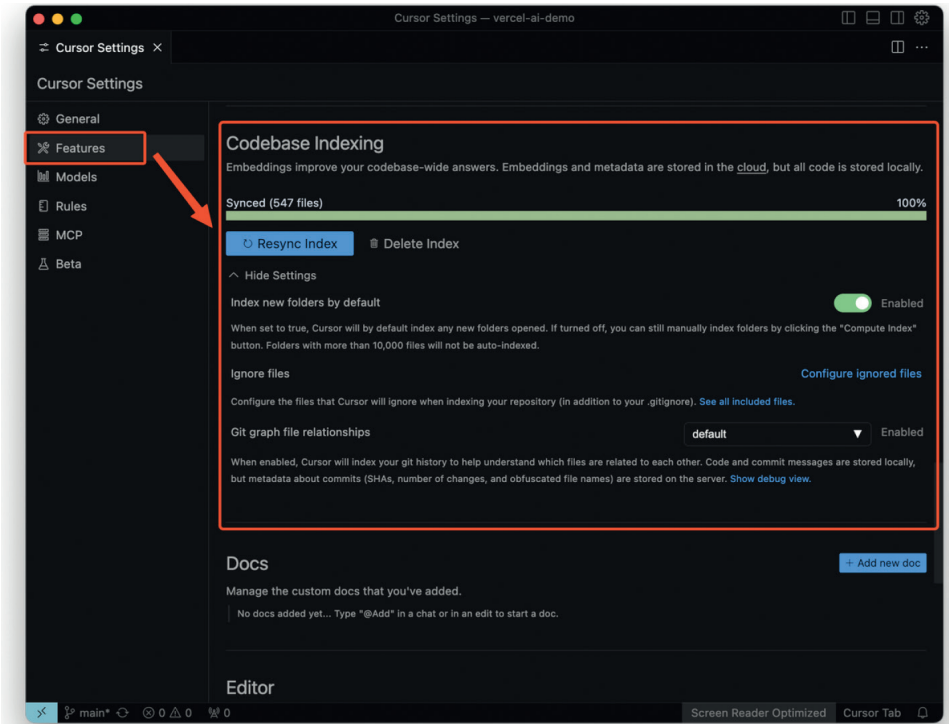


Рисунок 6-4. Включение индексирования кодовой базы



Рисунок 6-5. Переключение модели

Импорт настроек VS Code. Если вы уже используете VS Code, настоятельно рекомендуется воспользоваться командой **Import VS Code Extensions** (Импортировать расширения VS Code) и **Import VS Code Settings** (Импортировать настройки VS Code), как показано на рис. 6-6.

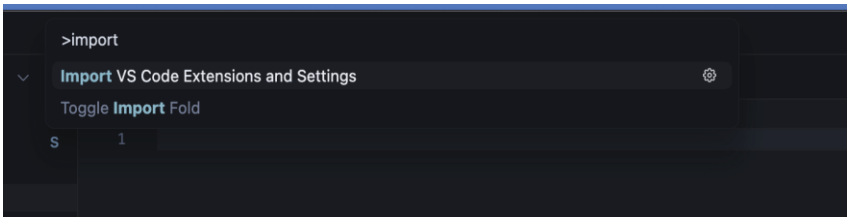


Рисунок 6-6. Импорт настроек VS Code

2. Установка Node.js

Node.js – это открытая кросс-платформенная среда выполнения JavaScript, построенная на основе движка V8 для Google Chrome. Одним из ее основных преимуществ является возможность одновременной разработки при помощи JavaScript как клиентских, так и серверных приложений. Такой унифицированный технологический стек упрощает процесс разработки, позволяя разработчикам использовать один язык (JavaScript/TypeScript) для создания всего стека приложения – от базы данных до пользовательского интерфейса, что значительно снижает сложность разработки и отлично подходит для генерации унифицированного кода с помощью ИИ.

Чтобы использовать Node.js, сначала необходимо установить среду выполнения. В операционных системах macOS или Linux установка выполняется следующим образом.

(1) Для установки инструмента nvm выполните следующую команду.

```
curl -o-
https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

(2) После установки nvm закройте и снова откройте терминал или загрузите файл конфигурации оболочки (например, *source ~/.bashrc или source ~/.zshrc*), а затем установите Node.js с помощью следующей команды.

```
nvm install --lts
```

В операционной системе Windows нельзя сразу использовать nvm, вместо этого необходимо запустить *nvm-windows*. Шаги по установке среды выполнения приведены ниже.

(1) Установите *nvm-windows*. Загрузите файл *nvm-setup.zip* со страницы последних выпусков репозитория *nvm-windows* на GitHub. После загрузки разархивируйте файл и запустите файл *nvm-setup.exe*. Мастер установки *Setup-NVM-for-Windows* поможет вам выбрать каталоги для установки *nvm-windows* и Node.js.

(2) Отобразите список доступных версий Node.js. После установки *nvm-windows* откройте *PowerShell* (желательно с правами администратора) и вы-

полните команду `nvm list available`, чтобы убедиться в корректной работе *nvm-windows*.

(3) Установите последнюю версию среды выполнения Node.js с помощью следующей команды.

```
nvm install latest
```

3. Установка pnpm

pnpm (performant node package manager) – это эффективный менеджер пакетов Node.js с экономным использованием дискового пространства. По сравнению с официальным инструментом npm от Node.js, pnpm претерпел глубокую оптимизацию, что привело к значительному улучшению скорости инсталляции, эффективности использования дискового пространства и других показателей. Кроме того, для ведения достаточно строгой вложенной структуры `node_modules` в pnpm используются жесткие и символьные ссылки, что полностью исключает проблему «призрачных зависимостей», присутствующую в npm. Таким образом, pnpm демонстрирует более высокую общую эффективность и по этой причине рекомендуется к использованию.

После установки Node.js для установки pnpm достаточно одной команды.

```
npm install -g pnpm@latest
```

После успешной инсталляции можно убедиться в ее работоспособности с помощью следующей команды:

```
pnpm --version
```

4. Установка Git

Git – это распределенная система контроля версий, которая позволяет тщательно отслеживать все изменения в проекте и является одним из незаменимых инструментов для эффективного управления проектами. Каждый раз, когда ИИ генерирует код, возникает риск снижения качества, однако правильное использование Git позволяет своевременно фиксировать последние доступные версии и в любой момент откатиться к предыдущему состоянию кода, что помогает снизить подобные риски. Поэтому Git также является одним из необходимых инструментов в методологии вайб-программирования.

В большинстве операционных систем macOS и Linux инсталляция Git происходит по умолчанию. Проверить наличие Git и его версию можно в терминале или командной строке с помощью следующей команды:

```
git --version
```

Если команда выдает номер версии Git (например, `git version 2.9.2`), это означает, что Git уже установлен. Если Git не установлен, выполните следующие шаги для его установки.

В операционной системе Windows установка осуществляется следующим образом.

(1) Загрузите установщик: перейдите на официальный сайт Git и загрузите последнюю версию установщика Git для Windows.

(2) Запустите установщик: дважды щелкните загруженный файл *.exe*, чтобы запустить мастер установки Git.

(3) Следуйте инструкциям мастера для завершения установки: для большинства пользователей подходящими будут настройки по умолчанию, если только нет особых причин для их изменения.

(4) Проверка установки: после завершения инсталляции откройте командную строку (если во время установки вы выбрали вариант «Не использовать Git в командной строке Windows», откройте Git Bash), введите команду `git --version` и нажмите клавишу **Enter**, чтобы подтвердить успешную установку Git.

В операционной системе macOS шаги по установке следующие.

(1) Загрузите установщик (рекомендуется): перейдите на официальный сайт Git и загрузите последнюю версию установщика Git для macOS.

(2) Запуск установщика: запустите установщик и следуйте инструкциям, чтобы завершить процесс инсталляции.

(3) Проверка установки: откройте терминал, введите команду `git --version` и нажмите клавишу **Enter** для проверки корректности установки Git.

В операционной системе Linux (на примере Debian/Ubuntu и Fedora) установку можно легко выполнить с помощью встроенного диспетчера пакетов. Шаги установки следующие.

(1) Обновление списка пакетов: откройте терминал и выполните следующую команду.

```
sudo apt-get update
```

(2) Установка Git: выполните следующую команду, чтобы установить Git.

```
sudo apt-get install git-all
```

(3) Проверка установки: введите команду `git --version` и нажмите клавишу **Enter**, чтобы проверить, удалась ли установка. После установки Git необходимо своевременно настроить имя пользователя и адрес электронной почты.

```
git config --global user.name "Ваше имя"
git config --global user.email "your_email@example.com"
```

5. Регистрация учетной записи ChatGPT

В этой книге в качестве основного инструмента большой языковой модели выбран ChatGPT, но читатель также может выбрать другую подходящую модель в зависимости от конкретной ситуации, например Doubao, Claude и т. д. Перед использованием API ChatGPT необходимо зарегистрироваться и получить токен API OpenAI. Для этого выполните следующие действия.

(1) Перейдите на платформу управления OpenAI по адресу https://platform.openai.com/.

(2) Нажмите кнопку **Sign Up** (Зарегистрироваться) в правом верхнем углу, чтобы зарегистрировать учетную запись. Можно войти с помощью учетной записи Google, Apple и т. д.

(3) После успешной регистрации и входа перейдите на страницу управления API-ключами и нажмите кнопку **+Create new secret key** (Создать новый секретный ключ) в правом верхнем углу, чтобы создать API-ключ, как показано на рис. 6-7.

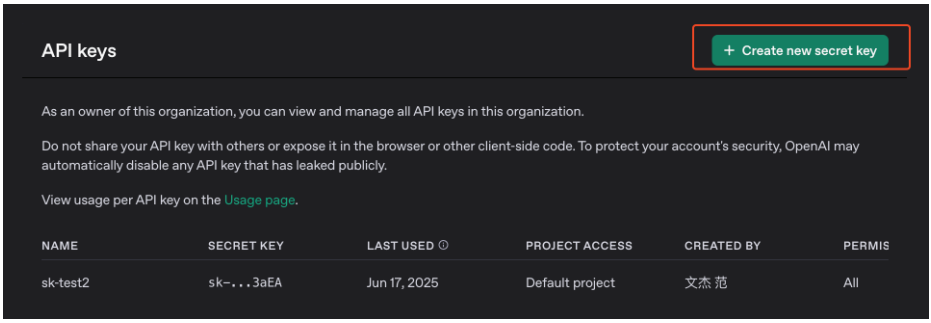


Рисунок 6-7. Создание токена API OpenAI

(4) Скопируйте новый ключ и сохраните его; в дальнейшем он понадобится для вызова API.

(5) [Опционально] После регистрации и получения ключа API можно пополнить счет на определенную сумму для последующих вызовов API.

На этом предварительная подготовка завершена. Далее мы рассмотрим структуру проекта для быстрого перехода к процессу разработки.

6.1.2. Структура проекта

В настоящее время технологии разработки ПО развиваются стремительными темпами, и для реализации программного проекта можно выбрать множество различных технологических стеков. Мы собрали здесь комплект легких шаблонов на базе JavaScript, подходящих для вайб-программирования по критериям удобства, стабильности и комфорта разработки. Этот набор шаблонов также послужит основой для разработки проекта в данной главе. Читатель может клонировать его в командной строке с помощью следующей команды:

```
git clone -b scaffold git@github.com:Tecvan-fe/xiaohongshu-generator.git
```

Затем выполните команду `cd xiaohongshu-generator`, чтобы перейти в каталог проекта, и еще раз выполните команду `pnpm install`, чтобы завершить инициализацию проекта. Структура проекта после инициализации выглядит примерно следующим образом.

```
xiaohongshu-generator/
├── 📁 docs/                          # Документация проекта
│   ├── ...
│
├── 📁 packages/                      # Каталог исходного кода Monorepo
│   ├── 📁 server/                    # Бэкенд-сервис
│   │   ├── src/
│   │   │   ├── ...                   # Исходный код сервера
│   │   ├── package.json              # Конфигурация зависимостей бэкенда
│   │   └── README.md                 # Описание бэкенд-сервиса
│   │
│   ├── 📁 web/                       # Фронтенд-приложение
│   │   ├── src/
│   │   │   └── ...                   # Исходный код веб-приложения
│   │   ├── package.json              # Настройки зависимостей фронтенда
│   │   ├── vite.config.ts            # Настройки сборки Vite
│   │   ├── tailwind.config.js        # Настройки Tailwind CSS
│   │   └── postcss.config.js         # Настройки PostCSS
│   │
│   ├── 📁 logger/            # Пакет инструментов для ведения логов
│   │   ├── src/
│   │   │   └── ...
│   │   ├── package.json
│   │   └── tsconfig.json
│   │
│   └── 📁 utils/             # Пакет вспомогательных функций
│       ├── src/
│       │   └── ...
│       ├── package.json
│       └── tsconfig.json
│
├── 📄 Файлы конфигурации
│   ├── package.json          # Корневой файл package.json (конфигурация Monorepo)
│   ├── pnpm-workspace.yaml   # Конфигурация рабочего пространства pnpm
│   └── ...
```

Ключевые каталоги включают:

- *packages/server* – для хранения кода бэкенда Node.js, в основном реализующего взаимодействие с ChatGPT;
- *packages/web* – для хранения кода веб-интерфейса, в основном реализующего взаимодействие с пользователем;
- *packages/utils* и *packages/logger* – библиотеки утилит для хранения кода фронтенда и бэкенда.

Все приведенные выше каталоги кода объединены через рабочую область pnpm, образуя структуру Monorepo. Это модель управления кодом, который настоятельно рекомендуется в контексте вайб-программирования и обладает следующими преимуществами.

- **Упрощенное управление кодом**: весь код хранится в одном репозитории, что облегчает отслеживание изменений в разных проектах, обеспечение единообразия стандартов и практик кодирования, а также надлежащий контроль версий.

- **Совместный и повторно используемый код**: единый репозиторий значительно упрощает совместное использование и повторное применение кода в разных проектах или компонентах, а также облегчает получение полного контекста кода для больших языковых моделей, что, в свою очередь, повышает эффективность помощи пользователям при генерации правильного кода.
- **Снижение затрат на проверку**: благодаря централизации всего кода в одном репозитории и наличию соответствующих инструментов контроля качества (таких как статический анализ кода, автоматическое тестирование и т. д.) можно быстро проверить эффективность любых изменений в глобальном масштабе, что очень помогает быстро проверять правильность результатов, генерируемых большими языковыми моделями.

На этом этапе мы уже настроили необходимую локальную среду и теперь переходим к этапу анализа требований проекта.

6.2. Анализ требований проекта

Вайб-программирование – это отнюдь не программирование вслепую. Как и в традиционной разработке, перед официальным запуском проекта необходимо тщательно проанализировать его функциональные и технические требования. Как минимум следует определить следующие моменты:

- ключевые функции проекта и основные процессы взаимодействия;
- планируемый стек технологий, а также соответствующие технические и нетехнические ограничения;
- приоритеты разработки и примерный план реализации.

Уточнение описанных выше моментов на ранней стадии проекта обеспечит целенаправленность процесса разработки и позволит избежать значительного отклонения от первоначальных ожиданий. Разумеется, такие документы можно постепенно составлять и дорабатывать с помощью ИИ-инструментов (таких как ChatGPT и др.). В этой главе будет показано, как, отталкиваясь от простой идеи, с помощью различных ИИ-инструментов шаг за шагом детализировать проект, чтобы в конечном итоге сформировать исчерпывающую документацию по требованиям, техническому проекту и плану реализации проекта.

6.2.1. Систематизация документации по требованиям

На ранних этапах проекта у нас обычно есть только смутное представление о том, что нужно сделать, например «разработать систему, которая будет автоматически преобразовывать введенный пользователем контент в карточки в стиле Xiaohongshu с сопроводительным текстом». Хотя эта идея и является многообещающей, ее недостаточно для непосредственного использования в качестве руководства при проектировании и разработке конкретной системы. Необходимо постепенно детализировать ее с помощью систематизированного процесса анализа требований, чтобы заложить прочную основу для последующей инженерной работы.

Для эффективного продвижения этого процесса мы рекомендуем использовать возможности больших языковых моделей и сформулировать четкие

и профессиональные подсказки с целью получения структурированного проекта требований. Ниже приведен рекомендуемый шаблон подсказки.

Вы являетесь профессиональным специалистом по анализу требований к программному обеспечению. Мы планируем разработать веб-систему, которая будет преобразовывать вводимый пользователем контент в карточки и тексты для платформы Xiaohongshu. Необходимо составить подробное техническое задание с учетом данного требования, которое должно содержать как минимум следующие разделы.

- Введение: кратко описать контекст и цели проекта, то есть предполагаемые действия.
- Функционал: перечислить функции и интерфейс, которые будет предоставлять система (описать видимые действия с точки зрения пользователя).
- Ограничения: четко определить ограничения, такие как технологический стек, требования к производительности, стандарты кодирования и т. д.
- Прочие детали (опционально) - для особо сложных или требующих строгого контроля алгоритмов/реализаций в техническом задании можно указать ключевые детали или особые требования.

Приведенные выше промпты имеют четкую структуру и направленность. Во-первых, путем определения роли большая языковая модель позиционируется как «эксперт по анализу требований к ПО», что помогает модели четко определить границы своих обязанностей и стиль вывода. Затем предоставляются четкие инструкции по заданию с требованием составить полный документ с описанием требований в соответствии с конкретной целью. Наконец, путем перечисления ключевых элементов в документе (таких как введение в проект, описание функций, технические ограничения и т. д.) ограничивается объем и структура выходных данных, что обеспечивает лаконичность, удобство использования и читабельность результата.

Введите эту подсказку в знакомую вам систему больших языковых моделей, например ChatGPT, и в нормальных условиях система вернет более полное описание требований, как показано в примере ниже.

```
# Система генерации карточек и текстов для Xiaohongshu -
# документ с описанием требований

## I. Введение

### Предпосылки проекта
В сфере создания контента Xiaohongshu стала важной платформой для брендового
маркетинга и самовыражения. Качественный графический и текстовый контент
является основным фактором привлечения пользователей и повышения эффективности
распространения информации. Однако многие авторы не обладают навыками дизайна
и написания текстов, что ограничивает скорость публикации и качество их контента.

### Цели проекта
Настоящий проект направлен на разработку веб-системы, в которой пользователю
достаточно ввести описание основного содержания, чтобы получить автоматически
сгенерированные картинки для карточек Xiaohongshu, выполненные в едином стиле
и с эстетичным оформлением, а также сопровождаемые подходящим текстом. Это
поможет пользователям оперативно создавать качественный графический контент
и автоматизировать весь процесс - от ввода данных до публикации.

---
```

```
## II. Описание возможностей

### 2.1 Интерфейс ввода данных
— Поддерживает ввод текста, включая описание продукта, описание сценария
  использования, ключевые слова, подчеркивающие преимущества продукта, и т. д.
— Возможность выбора типа создаваемого контента (например, рекламный текст,
  обзор, записи о повседневной жизни и т. д.).
— Возможность выбора стиля карточки (например, лаконичный, технологичный,
  в формате ежедневника и т. д.).
— Возможность загрузки изображения или выбора способа генерации изображения
  (например, использование встроенных в систему пиктограмм/иллюстраций
  в формате SVG и т. д.).

### 2.2 Модуль генерации текста
— Интеллектуальная генерация текста в стиле Xiaohongshu на основе введенного
  контента со следующими характеристиками:
  - высокая степень реалистичности и вовлеченности;
  - разговорный стиль и резонирующее начало;
  - завершение с хештегами (автоматическое добавление #хештегов).
— Поддержка регулировки длины контента (три уровня: короткий, средний,
  длинный).
— Поддержка генерации и переключения между несколькими вариантами текста.

### 2.3 Модуль генерации карточек
— Автоматическая генерация изображений карточек для Xiaohongshu, поддержка
  следующих элементов контента:
  - основной заголовок (например, название продукта);
  - подзаголовок (например, ситуация использования);
  - значок или фоновая иллюстрация (опционально);
  - стилизованный макет и цветовая гамма.
— Формат вывода – PNG или SVG высокого разрешения, с возможностью прямого
  сохранения или загрузки.

### 2.4 Предварительный просмотр и экспорт
— Предварительный просмотр текста и эффекта компоновки карточки в режиме
  реального времени.
— Поддержка загрузки изображения карточки и копирования текста одним нажатием.
— Возможность экспорта в формат черновика Xiaohongshu (JSON или Markdown).

### 2.5 Учетная запись и история (опционально)
— Поддержка регистрации и входа пользователей (с помощью электронной почты или
  учетной записи в социальной сети).
— Просмотр истории созданных записей и избранных текстов.
— Вторичное редактирование и повторное создание старых записей.

---

## III. Ограничения

### 3.1 Технические ограничения
— Фронтенд: React + Tailwind CSS + TypeScript.
— Бэкенд: Node.js (маршрутизация API с помощью Express или Next.js).
```

```
- ИИ-интерфейс: OpenAI API (GPT-4o) или Claude 3 для генерации текстов.
- Генерация изображений: рендеринг динамических карточек на основе SVG (без
  использования DALL·E или MJ).
- База данных (при необходимости): SQLite/PostgreSQL (управление через Prisma).

### 3.2 Требования к производительности
- Задержка при генерации текста не должна превышать 5 секунд (при нормальном
  состоянии сети).
- Время рендеринга карточки не должно превышать 1 секунду.
- Система должна выдерживать нагрузку в 1000 запросов на генерацию ежедневно.

### 3.3 Стандарты кодирования и требования к разработке
- Использование ESLint + Prettier для унификации стиля кода.
- Модульность всех компонентов с соблюдением принципа однозначности
  ответственности.
- Поддержка автоматического развертывания в среде разработки (например,
  Vercel).
- Унифицированная структура ответов всех API (включая code/message/data).
- Документирование интерфейсов фронтенда и бэкенда в соответствии со
  спецификацией OpenAPI.

---

## IV. Прочие детали и пояснения по специфике реализации

### 4.1 Создание подсказок для генерации текста
Чтобы гарантировать соответствие генерируемого текста контексту Xiaohongshu,
при разработке подсказок необходимо:
- четко требовать имитации стиля блогеров Xiaohongshu (например, "я недавно
  обнаружил…", "настоятельно рекомендую…");
- указывать стиль текста (например, рекомендация, впечатления, сравнение);
- вставлять определенные ключевые слова и эмоциональную направленность
  (например, "нежность", "технологичность", "повышение ощущения счастья").


### 4.2 Требования к визуализации SVG-карточек
- Использовать библиотеку пользовательских компонентов для инкапсуляции модулей
  карточек, с поддержкой параметрической передачи заголовков, подзаголовков,
  пиктограмм и т. д.
- Использовать шрифты с открытым исходным кодом для китайского языка (например,
  Alibaba Puhui).
- Обеспечить адаптацию к экранам мобильных устройств (пропорциональное
  масштабирование при ширине 750 пикселей).
```

Общая структура предварительной версии документа с описанием требований, генерация которой осуществляется с помощью большой языковой модели, как правило, уже обладает определенной полнотой и структурной целостностью, охватывая такие ключевые элементы, как описание возможностей, ограничения и т. д. Несмотря на кажущуюся целостность структуры документа, в нем все же могут присутствовать многочисленные проблемы, связанные с точностью и практической применимостью, что требует тщательной проверки и доработки вручную. В приведенном примере в текущей версии документа все еще присутствуют следующие заметные недостатки.

- **Отсутствие описания основного пользовательского процесса**. В документе отсутствует четкое описание основного сценария действий пользователя в системе, что затрудняет понимание принципа работы системы с точки зрения пользователя и не способствует согласованности между фронтендом и бэкендом при реализации функциональности. В связи с этим необходимо дополнить документ полным описанием основного процесса работы пользователя, включая то, как пользователь входит в систему, как отправляет контент, как система вызывает модель большого языка для обработки пользовательского ввода, как результаты анализируются и отображаются в виде карточек и т. д. Пример подсказки:

  ```
  Необходимо дополнить полное описание основного рабочего процесса
  пользователя, в котором должно быть в полной мере объяснено, как
  пользователь, войдя в систему, отправляет контент, как система вызывает
  модель большого языка, как затем возвращаются результаты и отображаются
  в виде карточек на интерфейсе и т. д.
  ```

- **Требования к производительности не имеют практического значения**. В разделе 3.2 документа изложены требования к производительности системы, однако на ранних этапах проекта оптимизация производительности не является приоритетной задачей, поэтому этот раздел можно при необходимости удалить во избежание ненужной предварительной оптимизации.
- **Отсутствие требований к пользовательскому интерфейсу**. Целью проекта на данном этапе является предоставление облегченного инструмента, не требующего системы учетных записей пользователей, поэтому описания таких функций, как регистрация, вход в систему и добавление в избранное, также можно полностью удалить.

Приведенные выше примеры являются лишь некоторыми типичными проблемами. В реальных сценариях их можно дополнительно уточнить и пересмотреть с учетом конкретных бизнес-потребностей. Например, если принято решение использовать рендеринг SVG вместо вызова моделей генерации изображений, таких как DALL·E, для рисования карточек в документации можно четко обозначить выбор этого решения и его обоснованность (например, более высокая степень управляемости, отсутствие зависимости от внешних сервисов и т. д.). Однако следует осознавать, что подобные ограничения дизайна могут впоследствии ограничить возможности расширения системы. Читатели могут по своему усмотрению гибко корректировать или удалять такие ограничения.

В целом проектная документация должна не только быть формально полной, но и отличаться точностью, понятностью и лаконичностью. В ней следует как можно подробнее изложить ключевые моменты, такие как общая концепция системы, основные задачи, функциональные процессы, выбор технологий и ограничения по внедрению. Чем конкретнее и четче будет документация, тем меньше будет количество переделок и недоразумений в ходе реализации проекта, а общая эффективность и качество разработки значительно повысятся.

6.2.2. Составление документации по техническому проектированию

Следующим шагом после создания предварительной проектной документации является составление документации по техническому проектированию на ее основе. Важно отметить, что проектная документация фокусируется на вопросе «что делать», в то время как суть документации по техническому проектированию заключается в ответе на вопрос «как это сделать». Первая определяет цели системы и ее взаимодействие с внешней средой, вторая определяет пути реализации и инженерные ограничения. Чтобы обеспечить постоянное соответствие между кодом, генерируемым большими языковыми моделями в процессе вайб-программирования, и способом реализации, необходимо четко ограничить варианты технической реализации с помощью документации по техническому проектированию.

Например, техническая документация может использоваться для ограничения конкретных способов реализации, таких как указание обязательного использования определенной технологии (например, технологии A) или запрет на использование технологий, не соответствующих архитектурным принципам или позиционированию проекта (например, технологии B). Либо она может ограничивать общий архитектурный стиль системы, например четко определяя границы ответственности фронтенда и бэкенда, структуру сервисов без сохранения состояния и т. д., чтобы избежать расхождений при автоматическом создании подробностей реализации моделью.

Как и в случае с техническим описанием требований, техническая документация также может быть сгенерирована с помощью больших языковых моделей. Рекомендуется использовать интегрированную среду разработки с ИИ (например, Cursor), которая обеспечивает глубокое восприятие контекста кода и документации, а также может автоматически переключаться в режим Agent для выполнения структурированного технического планирования. Достаточно просто скопировать содержание технического описания требований, сгенерированного в разделе 6.2.1, в Cursor и сохранить его в файл (например, *requirements.md*), а затем в режиме Cursor Agent ввести следующий промпт:

```
В соответствии с требованиями документа @requirements.md необходимо тщательно
проанализировать и сформулировать технологическую концепцию реализации.
Документ должен содержать описание основного функционального процесса,
архитектурный проект, а также детализированный выбор технологического стека.
Результаты следует сохранить в файле atech.md.
```

В итоге будет сформирована подробная техническая проектная документация; ниже приведены некоторые ключевые моменты.

```
# Генератор контента для Xiaohongshu - Техническая концепция и план реализации

## Основная техническая концепция

### 1. Процесс взаимодействия с пользователем

```mermaid
```
```

```
graph TD
    A[«📝 Ввод пользователем»] --> B{Тип контента}
    B -->|Текст| C["📄 Поле ввода текста <br/> с поддержкой Markdown"]
    B -->|Файл| D["📎 Загрузка PDF путем перетаскивания"]
    ...
```

Описание основного взаимодействия

📥 Этап ввода
...

🤖 Этап интеллектуальной обработки
...
...

2. Общая архитектура

```
┌──────────────────┐      ┌──────────────────┐      ┌──────────────────┐
│ Веб-интерфейс    │      │   Веб-бэкенд     │      │   ИИ-сервис      │
│                  │      │                  │      │                  │
│ React + TS       │◄──►  │  Node.js API     │◄──►  │ OpenAI GPT-4o    │
│ Tailwind CSS     │      │  Express/Vercel  │      │ (через ИИ SDK)   │
│ Zustand          │      │ Анализ PDF       │      │                  │
│ Framer Motion    │      │ Генерация SVG    │      │                  │
└──────────────────┘      └──────────────────┘      └──────────────────┘
```

2. Выбор технологического стека

Фронтенд-стек

— **Фреймворк**: ...

Бэкенд-стек

— **Среда выполнения**: ...

ИИ и обработка данных

— **Основная модель**: OpenAI GPT-4o
— **Альтернативные модели**: Gemini, Claude (переключение через настройки)
— **Процесс обработки**: анализ текста → извлечение абзацев → генерация заголовков → рендеринг шаблонов SVG

3. Проектирование основных модулей
...

4. Проектирование бэкенд-API

На основе существующей реализации в packages/server бэкенд предоставляет следующие REST API.

```
#### 4.1 Основные интерфейсы
...

#### 4.5 Обработка ошибок
...
#### 4.6 Безопасность и ограничения
...
## V. Схема развертывания

### 5.1 Развертывание фронтенда (Vercel)
...

### 5.2 Развертывание бэкенда (Railway/Render)
...

### 5.3 Оптимизация производительности
...

---

## VI. Стандарты разработки
...
```

Хотя с помощью больших языковых моделей можно быстро получить структурированный черновой вариант технического решения, на практике в первоначальном варианте черновика зачастую остаются многочисленные несоответствия реальному графику разработки, упущенные детали или вводящие в заблуждение сведения. Поэтому рекомендуется систематически оценивать и оптимизировать содержание документации по техническому проектированию с учетом трех критериев: завершенности, точности и целесообразности. Например, в данном примере присутствуют следующие очевидные проблемы.

Преждевременное введение требований к тестированию и документированию. На ранних этапах проекта, за исключением случаев использования модели **тестируемого программирования** (Test Driven Development, TDD), нет надобности спешить с внедрением полного объема тестирования и подробной документации по разработке. Такие материалы следует постепенно дополнять на более поздних этапах проекта, когда код станет относительно стабильным. Поэтому рекомендуется на время удалить соответствующие описания, чтобы не нарушить ритм и приоритеты разработки.

Слишком ранняя нежелательная оптимизация производительности. Ранние этапы проекта должны быть посвящены созданию работоспособного ядра функциональности. До ее стабилизации любые описания по оптимизации производительности, оценке нагрузки или распределению ресурсов являются преждевременными и должны быть удалены, чтобы техническая документация оставалась сфокусированной на ключевых задачах.

Отсутствие спецификаций экспорта изображений, ухудшающее качество конечного продукта. В нынешней версии документации отсутствуют четкие требования к разрешению экспортируемых изображений, что может привести к несоответствию их качества стандартам отображения целевой платформы (например, Xiaohongshu). В связи с этим следует добавить следующие спецификации.

```
**Система макетов**
Изображения для экспорта должны соответствовать следующим требованиям
к разрешению.
- Версия в формате постера: 750×1334 (9:16).
- Квадратный формат: 750×750 (1:1).
- Горизонтальный формат: 1080×608 (16:9).
```

- **Отсутствие четких спецификаций структуры кода фронтенда и бэкенда**. Если не установить строгие ограничения по структуре проекта, при дальнейшем создании кода с помощью больших языковых моделей могут возникнуть проблемы со смешением логики фронтенда и бэкенда, а также со степенью взаимодействия модулей, что серьезно повлияет на удобство обслуживания. Поэтому необходимо добавить четкое описание структуры проекта.

  ```
  Это архитектура проекта Monogeро на основе pnpm Workspace. Необходимо
  добавить правила и четко указать, что код бэкенда следует сохранять
  в пакете packages/server, код фронтенда в пакете packages/web, а общий для
  обеих частей код в пакете packages/utils.
  ```

- **Другие потенциальные проблемы и пути оптимизации**. Помимо перечисленных выше проблем, необходимо проанализировать по пунктам следующее:
  - соответствует ли архитектурный дизайн текущим функциональным целям (например, нет ли чрезмерной абстракции);
  - является ли выбор технологического стека практически осуществимым, есть ли зависимости с высоким уровнем риска;
  - существуют ли в вызовах сторонних сервисов (например, API моделей) неясные интерфейсы или ограничения по правам доступа;
  - имеются ли «расширения» или «резервные модули», не соответствующие позиционированию проекта; если да, необходимо тщательно оценить целесообразность их сохранения.

Подводя итог, можно сказать, что цель написания документации по техническому проектированию заключается в создании согласованной и пригодной для практического применения базы для последующей автоматизированной разработки и совместной работы. Следует избегать избыточности, устранять двусмысленности и ограничивать количество условий с целью обеспечить точность, лаконичность и конкретику. Тщательно проработанная документация по техническому проектированию является основой для высококачественной разработки, а также значительно снижает затраты на повторные коммуникации и доработки в процессе вайб-программирования.

6.2.3. Составление документации плана реализации проекта

После подготовки предварительной спецификации требований и документации по техническому проектированию необходимо составить третий документ: план реализации проекта. Здесь необходимо учесть один важный момент: окно контекста (Context Window) больших языковых моделей имеет строгие ограничения по длине. По мере продвижения разработки, увеличения объема кода и постоянного расширения содержания диалога состояние контекста, поддерживаемое ИИ-агентом, будет постепенно приближаться к максимально поддерживаемой

моделью длине. Как только предел будет превышен, это приведет к урезанию контекста, в результате чего модель будет терять важную исходную информацию или даже прекратит выполнение задачи.

Хотя в таких инструментах ИИ-программирования, как Cursor, уже встроены определенные стратегии для решения этой проблемы, например своевременный вызов модели Cursor Small для семантической компрессии старого контекста с целью снижения загрузки памяти, риск сбоя контекста при выполнении длительных и сложных задач разработки по-прежнему остается высоким. При сбое сеанса пользователю приходится прерывать предыдущую задачу и запускать новый диалог с агентом, при этом знания, накопленные в ходе предыдущего диалога, полностью теряются, а качество генерации в новом сеансе значительно снижается. В этот момент Cursor (или другой ИИ-агент для программирования) полностью утрачивает память о первоначальной задаче и целиком утрачивает всю предысторию контекста.

Поэтому требуется какой-то механизм долговременной памяти, позволяющий быстро «вспомнить» задачи проекта, определить актуальный прогресс и восстановить состояние разработки даже в случае «потери памяти» модели или прерывания сеанса. Эту задачу также можно решить с помощью больших языковых моделей.

```
В документах @tech.md и @requirements.md необходимо тщательно ознакомиться
с информацией и помочь мне составить документ с планом выполнения проекта.
В первую очередь необходимо завершить разработку системы бэкенд-интерфейсов,
а затем постепенно приступить к разработке фронтенда. Кроме того, статус
выполнения каждой задачи должен быть отмечен с помощью [ ]. В ходе последующего
диалога как только какая-либо задача будет выполнена, ее необходимо пометить
как [x] и сохранить результат в документе docs/plan.md.
```

Ниже приведены два небольших совета для выполнения этой задачи.

- Необходимо четко указать, что для обозначения статуса задачи следует использовать синтаксис Markdown [ ]. В этом случае Cursor перейдет в особый режим, позволяющий быстрее отслеживать ход выполнения задач.
- Необходимо четко указать, что сначала следует завершить разработку системы бэкенд-интерфейсов, поскольку в нашем проекте бэкенд относительно прост в реализации и, следовательно, его можно завершить быстрее.

Введя вышеуказанные подсказки в Cursor, можно сгенерировать документ с планом реализации проекта. Пример приведен ниже.

```
# Система генерации карточек и текстов для Xiaohongshu – план реализации
проекта

## Обзор проекта
...

---

## Этап 1: инициализация проекта и настройка среды
```

```
### 1.1 Базовые настройки проекта
...

### 1.2 Настройка среды разработки
...

---

## Этап 2: разработка ядра бэкенда

### 2.1 Создание базовой инфраструктуры

– [ ] Инициализация приложения Express и настройка промежуточного ПО
- ...


### 2.2 Интеграция ИИ-сервисов

– [ ] Установка зависимостей, связанных с Vercel AI SDK
– ...

### 2.3 Основные функции генерации текстов

– [ ] Разработка и реализация системы шаблонов подсказок
– [ ] ...

### 2.4 Разработка API
...

---

## Этап 3: разработка ядра фронтенда

### 3.1 Инфраструктура фронтенда
...

---

## Этап 4: интеграционное тестирование и оптимизация

### 4.1 Интеграция фронтенда и бэкенда
...

---

## Этап 5: развертывание и запуск

### 5.1 Подготовка к развертыванию
...

---
```

```
## Этап 6: последующие итерации и расширение

### 6.1 Расширение функциональности
...

---

## Основные этапы

– **Первый этап**: Инициализация проекта (1–2 дня)
– **Второй этап**: Разработка бэкенда (5–7 дней)
– **Третий этап**: Разработка фронтенда (5–7 дней)
– **Четвертый этап**: Интеграционное тестирование (2–3 дня)
– **Пятый этап**: Развертывание и запуск (1–2 дня)
– **Шестой этап**: Итеративная оптимизация (непрерывно)

**Общий предполагаемый срок разработки**: 2–3 недели

---

## Важные примечания

1. **Безопасность API-ключа**: ...

---
```

Полученные таким образом результаты также необходимо тщательно проверять для исключения нецелесообразного или лишнего содержания. Например, в этом документе с планом выполнения проекта по-прежнему присутствуют некоторые существенные проблемы.

- **Заведомо ненужные оценки сроков**. В документе представлено несколько вымышленных временных показателей, таких как «Общий предполагаемый срок разработки: 2–3 недели» и т. п. В рамках модели вайб-программирования подобная информация лишена практической ценности и, напротив, может ввести в заблуждение ИИ при построении логики управления ходом работ, поэтому все эти показатели следует удалить.

  ```
  Удалить все конкретные временные показатели, например указанные в документе
  **«Общий предполагаемый срок разработки»: 2–3 недели**.
  ```

- **Необоснованные предположения о дальнейшем развитии**. Например, в разделе «6.3. Техническое обновление» (опущен в книге) приводится ряд возможных направлений оптимизации и вариантов выбора технологий на будущее. На данном этапе такие предположения не имеют никакого практического значения и лишь усложняют контекст, поэтому рекомендуется их полностью удалить.

После внесения описанных выше изменений вы получите документ с планом реализации проекта с четкой структурой и четко сформулированными целями. В сочетании с упомянутыми выше документами по описанию требо-

ваний и техническому проектированию подготовительные работы по разработке можно считать в основном завершенными, и теперь можно переходить к этапу вайб-программирования.

6.3. Разработка бэкенда

После завершения работы по систематизации требований к проекту, описанной в разделе 6.2, нами были подготовлены следующие три основных документа.

- *requirements.md* (**документ с описанием требований**): используется для определения целей продукта, границ функциональности, основных процессов взаимодействия, а также ключевых технических и нетехнических ограничений.
- *tech.md* (**документ технического проекта**): определяет выбор технологического стека, основную архитектурную структуру и технические ограничения в процессе реализации.
- *plan.md* (**документ плана реализации проекта**): разбивает этапы разработки, определяет порядок выполнения, предоставляет ИИ-агенту контекстные подсказки, обеспечивая непрерывность и контролируемость выполнения задач.

На этом подготовка к проекту в основном завершена, и далее мы можем перейти к этапу программирования. Целью данного проекта является реализация «генератора контента для Xiaohongshu». Система включает в себя следующие две части.

- **Фронтенд-система**: отвечает за взаимодействие с пользователем, предоставляет интерфейс для ввода контента и предварительного просмотра карточек.
- **Серверная система**: отвечает за взаимодействие с большой языковой моделью, обрабатывает запросы на генерацию текста и рендеринг изображений.

С архитектурной точки зрения эти две части развязаны на уровне кода и взаимодействуют исключительно через стандартный HTTP-интерфейс. Принимая во внимание относительную простоту бэкенд-сервиса, основной упор при реализации будет сделан на построение логики взаимодействия с большой языковой моделью, поэтому в данной разработке приоритет будет отдан созданию бэкенд-сервиса. Далее мы сосредоточимся на бэкенде, постепенно реализуя основные интерфейсы и ключевые функциональные модули.

6.3.1. Принципы реализации

В соответствии с проектным решением, изложенным в разделе 6.2, бэкенд-сервис будет реализован с использованием следующего технологического стека.

- **Язык и среда выполнения**. Написано на JavaScript, работает в среде Node.js.
- **Серверный фреймворк**. Используется Express. Хотя в современной веб-разработке Express уже не является самой передовой технологией, его

долгая история, обширное сообщество и подробная документация означают, что при обучении больших языковых моделей будет больше соответствующих корпусов данных, ИИ будет лучше их понимать, а качество генерируемого кода Express будет выше.

- **Фреймворк для ИИ-интерфейсов**. Выбран Vercel AI SDK. Этот фреймворк является одним из немногих специализированных наборов средств разработки ПО (Software Development Kit, SDK) для ИИ в современной экосистеме JavaScript. По сравнению с LangChain он имеет более понятную и четкую структуру интерфейса, что позволяет быстро создавать логику вызова больших языковых моделей. Обратите внимание: поскольку Vercel AI SDK является относительно новым фреймворком, в обучающих данных для больших языковых моделей недостаточно информации о нем, поэтому на начальном этапе результаты генерации могут быть неточными.

Для решения этой проблемы рекомендуется использовать практический прием – механизм документа *llms.txt*. Это специальный канал для ввода знаний, разработанный для взаимодействия с большими языковыми моделями. Читатель может перейти по ссылке https://ai-sdk.dev/llms.txt (пример содержания см. на рис. 6-8), скопировать содержимое файла в локальный каталог проекта (рекомендуемый путь: *docs/llms/vercel-ai-sdk.md*). В дальнейшем при взаимодействии с ИИ посредством ссылки на файл можно помочь модели разобраться в использовании Vercel AI SDK, что значительно повысит точность и согласованность генерируемого кода.

````
---
title: Node.js HTTP Server
description: Learn how to use the AI SDK in a Node.js HTTP server
tags: ['api servers', 'streaming']
---

# Node.js HTTP Server

You can use the AI SDK in a Node.js HTTP server to generate text and stream it to the client.

## Examples

The examples start a simple HTTP server that listens on port 8080. You can e.g. test it using `curl`:

```bash
curl -X POST http://localhost:8080
```

<Note>
  The examples use the OpenAI `gpt-4o` model. Ensure that the OpenAI API key is
  set in the `OPENAI_API_KEY` environment variable.
</Note>

**Full example**: [github.com/vercel/ai/examples/node-http-server](https://github.com/vercel/ai/tree/main/examples/node-http-server)

### Data Stream

You can use the `pipeDataStreamToResponse` method to pipe the stream data to the server response.

```ts filename='index.ts'
import { openai } from '@ai-sdk/openai';
import { streamText } from 'ai';
import { createServer } from 'http';

createServer(async (req, res) => {
  const result = streamText({
````

Рисунок 6-8. Пример llms.txt

На основе приведенной выше технической архитектуры и в соответствии с требованиями к проектированию интерфейсов, изложенными в *tech.md*, бэкенд-сервис должен предоставлять следующие интерфейсы.

- Базовые сервисные интерфейсы:
  - *GET/health* – проверка работоспособности сервиса;
  - *GET/api* – получение базовой информации об API и доступных конечных точках.
````

- Интерфейсы обработки контента (*/api/content*):
  - *POST/api/content/parse-text* – разбор текстового контента, извлечение структуры абзацев и метаданных;
  - *POST/api/content/parse-pdf* – анализ PDF-файлов, извлечение текстового содержания и анализ структуры.
- Интерфейсы ИИ-анализа (*/api/ai*):
  - *POST/api/ai/analyze* – анализ текстового содержания с помощью ИИ, генерация карточек абзацев в стиле Xiaohongshu;
  - *POST/api/ai/titles* – генерация нескольких вариантов заголовков в стиле Xiaohongshu на основе текстового содержания;
  - *POST/api/ai/cards* – преобразование проанализированных данных абзацев в формат визуальных карточек.
- Интерфейс экспорта контента (*/api/export*):
  - *POST/api/export/markdown* – экспорт данных карточек в файл формата Markdown;
  - *POST/api/export/json* – экспорт данных карточек в файл формата JSON.

Общая логика вызова интерфейсов показана на рис. 6-9.

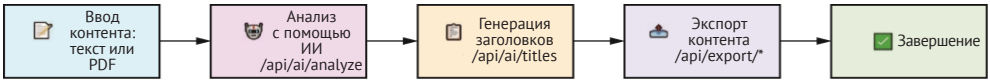


Рисунок 6-9. Последовательность вызова интерфейсов

После тщательного изучения функций бэкенд-сервиса и системы интерфейсов можно приступать к этапу программирования.

6.3.2. Разработка программы бэкенд-сервиса

Далее нам нужно лишь ввести в Cursor соответствующие подсказки для управления процессом разработки.

```
Нужно использовать файлы @requirements.md и техническую документацию в каталоге
docs для пошагового написания кода в соответствии с документом плана проекта
@plan.md. Необходимо фокусироваться на написании бизнес-кода, не нужно писать
код для модульных тестов. Код следует сохранять в каталоге packages, при этом
приоритет должен отдаваться разработке серверной части.
```

Далее Cursor автоматически выполнит ряд ключевых операций разработки, среди которых:

- инициализация системной структуры (например, создание конфигурационных файлов *package.json*, *tsconfig.json* и т. д.);
- пошаговое построение системы в соответствии с планом, изложенным в *tech.md*, включая определение интерфейсов, реализацию бизнес-логики, инкапсуляцию инструментальных модулей и логов и т. д.

Эти ключевые операции разработки обычно занимают несколько минут, и в результате, как правило, получается базовый код с полной структурой и полностью готовый к запуску. На этом этапе можно попробовать инициализиро-

вать проект для проверки работоспособности системы. Перед запуском необходимо выполнить следующие два шага.

(1) Выполнить команду `pnpm i` для установки зависимостей проекта.

(2) Ознакомиться с определениями команд в файле *package.json* и определить команду запуска. В примере этой главы командой запуска является `dev`, как показано на рис. 6-10.

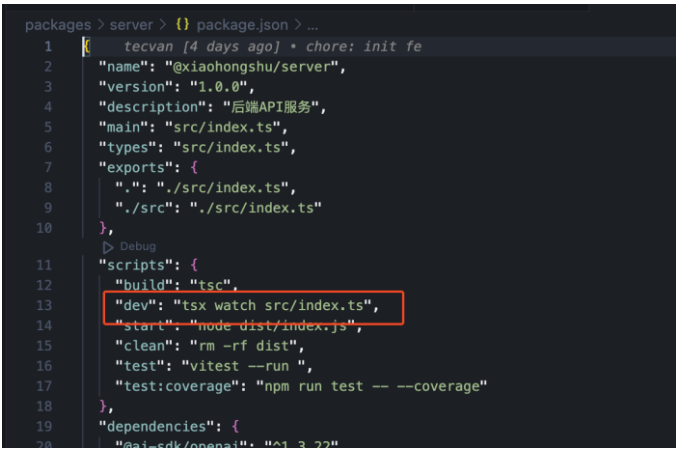


Рисунок 6-10. Пример файла package.json

После определения команды запуска для запуска системы достаточно ввести в командной строке `OPENAI_API_KEY=sk-proj-xxx npm run dev`, где значение `OPENAI_API_KEY` – это ключ API ChatGPT, полученный в разделе 6.1.1. В случае сбоя при запуске командная строка обычно выдает подробное сообщение об ошибке, как показано на рис. 6-11.



Рисунок 6-11. Команда Add to Chat в командной строке

Обратите внимание, что сгенерированный ИИ контент может быть неполным или содержать программные ошибки, синтаксические ошибки и т. д. Это может привести к сбою запуска. Конкретные причины ошибок разнообразны, и невозможно перечислить их все, но мы можем использовать Cursor для автоматического исправления таких ошибок. Нажмите кнопку **Add to Chat** в правом верхнем углу окна ошибок командной строки, показанного на рис. 6-11, а затем введите следующую подсказку на панели **Agent**.

```
Исправить ошибку командной строки.
```

Далее Cursor попытается проанализировать и исправить соответствующие ошибки. Если исправление не удалось с первого раза, можно повторить операцию несколько раз для постепенного сужения круга проблем. Если после нескольких попыток проблема не решена, можно рассмотреть возможность перехода на модель с более высокой производительностью (например, версию Max модели Claude Sonnet 4) или найти решение в логе ошибок для ручного исправления.

6.3.3. Рецензирование кода

После первоначального создания кода необходимо провести его первоначальное рецензирование. Это крайне важно, так как в ходе рецензирования обычно выявляется немало проблем. В разделе 5.3 уже были приведены многие правила и приемы рецензирования кода, поэтому здесь нет необходимости повторяться. Рекомендуется сосредоточиться на следующих аспектах при рецензировании серверного кода.

- Проверьте, насколько исчерпывающими являются логи, например включают ли различные ключевые процессы в себя достаточно подробные записи, раскрывающие подробную информацию о работе. Если подобные проблемы обнаружены, можно использовать для оптимизации такую подсказку.

  ```
  Внимательно изучить актуальный код и добавить логи в ключевые места
  основного процесса, чтобы полностью раскрыть состояние работы системы.
  ```

- Проверьте, соответствуют ли фактически сгенерированные интерфейсы запланированным в *tech.md*. Этот шаг обязателен: интерфейсов может быть много, но их не должно быть недостаточно, иначе в контексте возникнут информационные неточности, что повлияет на качество последующего кода. Если подобные проблемы обнаружены, можно использовать для оптимизации такую подсказку:

  ```
  Провести оптимизацию интерфейса xxx, строго следуя плану в документе @tech.md,
  и использовать путь xxx для предоставления сервисного интерфейса.
  ```

- Проверьте, является ли обработка ошибок корректной. Особенно следует избегать чрезмерной защищенности: если необходимо вызвать исключение, его нужно непременно вызвать, чтобы система могла быстро прервать работу и минимизировать влияние на последующее состояние бизнес-процессов. При наличии подобных проблем для оптимизации можно использовать такую подсказку:

  ```
  Оптимизировать обработку ошибок в файле xxx, напрямую вызывая исключение.
  ```

- Проверьте, соответствует ли выбор технологического стека плану в *tech.md*. В частности, не следует добавлять ненужные уровни сложности. Например, нередко встречается ситуация, когда Cursor самостоятельно вводит библиотеку `fs-extra`, хотя на самом деле использование нативных библиотек `fs/promises` гораздо проще. При наличии подобных проблем для оптимизации можно использовать следующую подсказку:

```
Удалить библиотеку xxx и запретить ее использование в дальнейшем; вместо
нее следует использовать библиотеку xxx.
```

- Проверьте, соответствует ли код вашим техническим предпочтениям. Это также очень распространенная проблема: в процессе разработки мы обычно не устанавливаем слишком много предварительных стандартов, и ИИ действует по своему усмотрению, а проблемы обнаруживаются только при проверке. В этом случае рекомендуется не идти на компромисс, а выполнить следующие два действия.
  - Используйте команду *Cursor /Generate Cursor Rules* для генерации нового документа с техническими спецификациями, как показано на рис. 6-12.

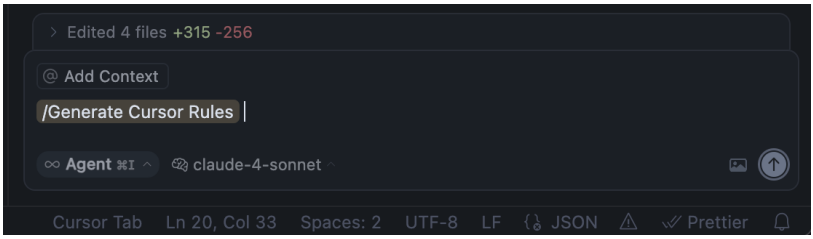


Рисунок 6-12. Автоматическая генерация Cursor Rules

  - Потребуйте от Cursor оптимизировать актуальный код в соответствии с новым документом технических стандартов. Используйте следующую подсказку:

```
Оптимизировать код в соответствии с правилом @.cursor/rules/xx-rule.mdc,
уменьшив количество всех случаев xxx.
```

Обратите внимание: если проверка кода будет слишком тщательной, в реальных сценариях будет обнаружено очень много проблем. Если решать их одну за другой, то значительное количество времени уйдет на оптимизацию кода, который на ранних этапах еще нестабилен, что нецелесообразно. Советуем читателям делать разумный выбор: если проблема не является критической, ее можно пропустить, уделяя на ранних этапах больше внимания разработке и тестированию функциональности.

Важно помнить, что этап рецензирования кода находится на ранней стадии проекта, и в дальнейшем неизбежно будет проведена масштабная переработка и оптимизация. Поэтому на данном этапе нет необходимости уделять чрезмерное внимание качеству кода.

6.3.4. Тестирование интерфейсов

В случае нормального запуска и работы кода, сгенерированного на предыдущих этапах, перед нами встанет следующая задача: как убедиться, что функциональная логика этого кода работает в соответствии с предполагаемыми ожиданиями? Поскольку к настоящему моменту визуальный веб-интерфейс еще не создан, мы не можем вызывать интерфейсы напрямую и, следовательно, не имеем возможности непосредственно наблюдать за поведением кода.

В области программной инженерии существует концепция «сдвига влево», которая, по сути, заключается в необходимости проводить проверку качества на как можно более ранних этапах жизненного цикла ПО, поскольку чем раньше будут обнаружены проблемы, тем ниже будет стоимость их устранения. Это правило применимо и к методологии вайб-программирования, поэтому перед завершением разработки веб-системы нам необходимо провести максимально полное тестирование качества интерфейсов. Для этого можно продолжить использование Cursor, чтобы сгенерировать тестовые скрипты для проверки поведения основных интерфейсов. Для этого нужно ввести следующую подсказку:

```
Внимательно ознакомиться с кодом в каталоге packages/server, разобраться в его
основных интерфейсах, а затем написать скрипт вызова интерфейсов для проверки
их поведения и стабильности.
```

После запуска серверной программы необходимо выполнить подобные тестовые скрипты, сгенерированные Cursor, и по записям в логах скриптов, равно как и по сопоставлению входов и выходов интерфейсов, определить их работоспособность. Вы также можете продолжить взаимодействие с Cursor для генерации дополнительных тестовых сценариев до полного понимания действующей логики работы системы.

Однако тестирование на этом этапе проводится не только ради самой проверки, но в большей степени для изучения логики работы системы. Поэтому не стоит вводить слишком много логики автоматизированного тестирования, так как в дальнейшем все равно будут проводиться масштабные итерации и оптимизации, поэтому проведенные на данном этапе автоматизированные тесты и оптимизация производительности, скорее всего, потребуют доработки. Здесь действует та же логика, что и при вайб-программировании: на ранних этапах проекта следует уделять больше внимания итерациям функциональности, а тестирование и оптимизацию отложить на более поздний срок.

Кроме того, не рекомендуется использовать модель TDD. Это методология разработки с акцентом на тестировании, позволяющая продвигаться от результата к реализации, что в принципе подходит для вайб-программирования. Однако сама концепция TDD слишком абстрактна, и ее сложно освоить тем, кто не имеет глубоких знаний в области программирования, из-за чего она вряд ли принесет ощутимый эффект. Поэтому более целесообразным все же будет использование традиционных подходов к разработке.

6.3.5. Доработка и дополнение функциональных возможностей

Процесс разработки сложных проектов никогда не происходит за один подход. Несмотря на то что большие языковые модели позволяют быстро генерировать рабочий прототип кода, первоначальный результат зачастую не соответствует реальным требованиям проекта. Это особенно актуально в случаях с высокой сложностью требований, когда между намерениями, воспринятыми ИИ, и реальными ожиданиями разработчиков возникают заметные расхождения.

Первая версия серверного кода, несмотря на полную структуру и наличие всех необходимых интерфейсов, все же содержала серьезные недостатки в ключевых

моментах. Например, при реализации «Генератора контента для Xiaohongshu» интерфейс */api/ai/cards* изначально был разработан для дальнейшей обработки структуры проанализированных абзацев с целью генерации контента для карточек в стиле Xiaohongshu со всей семантической целостностью, визуальной логикой и стилистической согласованностью. Однако при первоначальной реализации ИИ провел лишь простое семантическое разбиение и реорганизацию исходного текста абзаца, не задействовав для каждой карточки большую языковую модель с целью дальнейшей генерации высококачественного текста или визуальных указаний. Это привело к тому, что сгенерированный контент оказался лишенным оригинальности и контекстуальной адаптивности, что явно отклоняется от поставленных нами ключевых целей.

Подобные ситуации достаточно распространены при разработке с использованием ИИ. Искусственный интеллект может выполнять задачу формально и при этом упускать более глубокую рабочую логику на семантическом уровне. Поэтому требуется дальнейшая итеративная оптимизация.

```
При внимательном анализе кода интерфейса /api/ai/cards видно, что в нынешней
реализации не происходит вызов большой языковой модели для генерации контента
карточек. Необходимо оптимизировать этот процесс. В соответствии с рабочим
процессом на основе пользовательского ввода сначала следует определить
необходимое количество карточек и содержание каждой из них, а затем вызвать
большую языковую модель для оптимизации вывода контента.
```

Здесь важно понимать, что ИИ выступает лишь в роли эффективного исполнителя, тогда как люди берут на себя более важную функцию планировщиков и контролеров, отвечая за предоставление четких требований, проектных замыслов и ограничений, а также за проверку и контроль генерируемого ИИ кода. Это означает, что любые упущения или неточности в предварительном описании требований могут привести к появлению ошибок, поэтому всегда нужно проявлять терпение и выявлять проблемы с помощью рецензирования кода, тестирования интерфейсов и других методов, особенно в отношении функциональных упущений, и далее постоянно дорабатывать систему по мере ее создания.

6.4. Разработка веб-систем

По завершении разработки серверных интерфейсных сервисов необходимо продолжить итерационную разработку веб-систем пользовательского интерфейса. В рамках модели вайб-программирования сложность разработки веб-систем, как правило, значительно превышает сложность создания серверных систем. Эта сложность обусловлена не столько самой бизнес-логикой, сколько двумя следующими факторами.

- Веб-стек сам по себе характеризуется высокой степенью специализации инструментов и фрагментацией экосистемы. Современная разработка веб-приложений включает использование языков HTML, CSS и JavaScript, а также требует взаимодействия и настройки на нескольких уровнях: инструментов сборки, модульных систем, фреймворков компонентов, шаблонов стилей и моделей адаптивного дизайна. Все эти элементы требуют большого объема контекстуальной информации, что само по себе приводит к значительной информационной нагрузке.

- Поведение бэкенд-интерфейсов можно протестировать с помощью команды `curl`, инструмента Postman или вывода бэкенд-логов, тогда как тестирование поведения веб-интерфейса в большей степени зависит от интерфейса взаимодействия человека с машиной, а также визуальной обратной связи в браузере. На данном этапе развития технологий инструменты на основе больших языковых моделей (такие как генераторы кода или ИИ-агенты) пока еще не могут точно распознавать и обрабатывать события на странице. В результате эффективность использования больших языковых моделей в веб-разработке не так высока, как в сценариях серверной разработки.

Подытоживая сказанное выше, можно сказать, что для более эффективной разработки веб-систем нам необходимо использовать больше инструментов и более сложные подсказки.

6.4.1. Подход к реализации

Прежде чем приступить к разработке, давайте кратко обрисуем логику взаимодействия со страницей. В отношении примера проекта «Генератор контента для Xiaohongshu», с учетом документов, рассмотренных в разделе 6.2, в данном проекте используется набор современных веб-технологий с целью достижения баланса между удобством обслуживания, скоростью отклика и эффективностью разработки. Основные технические компоненты и их функции в основном технологическом стеке приведены ниже.

Tailwind CSS: атомарный CSS-фреймворк для построения гибкой и унифицированной системы стилей страниц. По сравнению с традиционными решениями CSS или CSS-in-JS, Tailwind CSS отличается большей простотой и удобством комбинирования, что способствует быстрому созданию прототипов пользовательского интерфейса в режиме вайб-программирования.

Vite: инструмент для сборки веб-приложений нового поколения со сверхбыстрой загрузкой и компиляцией по требованию. Значительно повышает эффективность разработки, особенно подходит для сценариев совместной работы с несколькими модулями и разработки с предварительным просмотром компонентов.

React: фреймворк для создания визуальных компонентов, отвечающий за декларативное представление пользовательского интерфейса страницы. Система React Hooks в сочетании с DOM обеспечивает эффективную поддержку динамических взаимодействий с управлением состояниями.

shadcn/ui: библиотека UI-компонентов на основе Radix и Tailwind, обладающая преимуществами хорошей доступности и высокой стилевой согласованности. Она позволяет снизить нагрузку при реализации интерфейса и повысить визуальную согласованность.

Zustand: облегченная библиотека управления состоянием, подходящая для управления глобальным состоянием и состоянием страниц в проектах среднего и малого масштаба. В данном проекте она в основном используется для управления подсказками, статусом загрузки и совместного управления генерируемым контентом.

TypeScript: подмножество языка JavaScript с расширенными типами, которое обеспечивает более строгие типовые ограничения и возможности автодополнения. В контексте ИИ-кодирования TypeScript помогает эффективно уменьшить семантические отклонения в сгенерированном коде.

Опираясь на эти компоненты, мы планируем создать интерфейс для диалогового взаимодействия человека с системой. Общая логика взаимодействия будет основана на модели использования ChatGPT: входные данные пользователя служат триггером, а сгенерированный контент является обратной связью, образуя замкнутый цикл. Основной процесс взаимодействия состоит из следующих трех этапов.

(1) **Страница в исходном состоянии**. После загрузки страницы по умолчанию отображается поле ввода диалога, как показано на рис. 6-13. Пользователь может ввести подсказку и запустить запрос на генерацию контента[1].



Рисунок 6-13. Исходное состояние веб-системы

(2) **Промежуточная страница.** После ввода запроса пользователем страница должна перейти в режим загрузки (см. рис. 6-14), что в основном служит для информирования о состоянии работы системы и предотвращения ситуаций с отсутствием результатов.

[1] Надпись в выделенной рамке гласит: «Стиль “Маленький красный куст”. “Живая жидкость”: отличный вариант для обзоров и рекомендаций. Девочки, обязательно посмотрите! Ребята, спешите!» – *Прим. перев.*



Рисунок 6-14. Страница в режиме загрузки

(3) **Страница отображения результатов**. После генерации контента на стороне сервера страница должна отобразить полученные результаты в структурированном виде. Отображаемый контент включает в себя карточки контента в стиле Xiaohongshu, сгенерированные заголовки и тексты и т. д. (см. рис. 6-15). Эта страница также должна поддерживать функцию экспорта контента, предоставляя пользователям возможность хранения или повторного использования материалов.



Рисунок 6-15. Страница результатов

Благодаря взаимосвязи между описанными выше тремя этапами образуется полный замкнутый цикл взаимодействия «подсказка → загрузка → результат», что обеспечивает хорошую структурную основу для последующего расширения функциональности и улучшения диалога с моделью.

6.4.2. Разработка веб-страниц

После завершения предварительного проектирования логики взаимодействия страниц и выбора технологий можно приступить к разработке контента веб-

страниц с помощью Cursor. Например, для запуска разработки можно использовать следующие подсказки:

```
С помощью @requirements.md и технической документации в каталоге docs необходимо
начать пошаговое написание кода в соответствии с планом @plan.md. Основное
внимание следует уделить функциональному коду, при этом нет необходимости писать
код для модульных тестов. Код следует сохранять в каталоге packages.
После завершения разработки веб-страниц следует в первую очередь создать
статические страницы, временно используя имитационные данные вместо всех
внешних данных. Фокус должен быть сделан на разработке визуальных аспектов,
таких как статический вид страниц и переходы.
```

Перед официальной интеграцией с бэкенд-интерфейсами рекомендуется сначала создать статические страницы с использованием имитационных данных. Такой подход позволяет эффективно контролировать сложность разрабатываемой задачи, уменьшить количество потенциально влияющих на результат факторов и повысить эффективность разработки.

Далее Cursor постепенно создаст код страниц в соответствии с планом разработки из файла *plan.md*. В ходе этого процесса разработчику достаточно наблюдать за генерацией кода и при необходимости вносить небольшие корректировки или давать указания. Весь процесс обычно занимает несколько минут. В результате получается готовая к запуску базовая структура проекта с четкой организацией.

После генерации кода можно запустить локальную среду разработки и просмотреть рендеринг страниц в браузере, выполнив следующие два шага.

(1) **Установка зависимостей**: для установки зависимостей необходимо выполнить следующую команду в командной строке.

```
pnpm install
```

(2) **Проверка команды запуска**: откройте файл *package.json* и ознакомьтесь со скриптом запуска, определенным в проекте. В данном примере выполните команду, показанную на рис. 6-16, чтобы запустить процесс разработки `npm run dev`.



Рисунок 6-16. Пример команды запуска

Кроме того, при первом запуске может произойти сбой из-за синтаксических ошибок, неустановленных зависимостей или неверных путей. В этом случае можно воспользоваться процедурой, описанной в разделе 6.3: нажмите кнопку **Add to Chat** в правом верхнем углу интерфейса командной строки для отправки лог-файла ошибок в окно чата и отправьте запрос на автоматическое исправление с помощью Cursor.

После успешного запуска сервера разработки достаточно перейти в браузере по адресу, выведенному в командной строке (см. рис. 6-17), чтобы в режиме реального времени просмотреть результат рендеринга страницы.

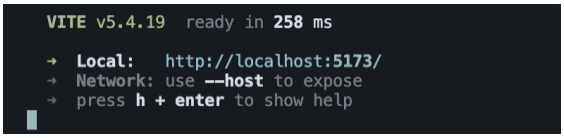


Рисунок 6-17. Результат запуска

6.4.3. Рецензирование кода

Как и в случае с разработкой бэкенда, настоятельно рекомендуется провести систематическую экспертизу кода до перехода к этапу отладки и запуска страниц. Этот процесс помогает выявить потенциальные ошибки на ранней стадии и способствует повышению стабильности структуры страниц, а также облегчает их последующее обслуживание.

Подобно проверке структуры лог-файлов и обработки ошибок в бэкенд-разработке, при создании веб-страниц также возникает ряд характерных проблем. Ниже перечислены наиболее распространенные структурные неполадки, которые рекомендуется тщательно проверить на данном этапе.

Хаотичная иерархия компонентов. ИИ склонен собирать всю логику в одном большом компоненте без четкой иерархии, смешивая в одном компоненте логику рендеринга визуализации, управления состоянием и асинхронной обработки, что приводит к ухудшению удобства чтения и сложности обслуживания. Простой критерий для определения нарушения иерархии компонентов: если код компонента превышает 300 строк, это означает, что компонент нарушает «принцип единственной ответственности» (Single Responsibility Principle, SRP). Для оптимизации рекомендуется использовать следующую подсказку.

```
Компонент @xxx.tsx слишком сложен, необходимо разбить его на дочерние
компоненты для соблюдения принципа единственной ответственности.
```

- **Отсутствие обработки предельных значений**. В области программной инженерии широко распространено утверждение: не доверяйте никаким внешним источникам данных. При генерации кода визуализации искусственный интеллект зачастую по умолчанию исходит из стабильности данных и не обрабатывает предельные случаи, такие как `null` (пустое значение) или `undefined` (не определено), что может привести к сбоям в отображении страницы или даже к сбою системы. Поэтому после получения внешних данных или ответа асинхронного интерфейса обязательно добавьте проверку на пустое значение и обработку ошибок, а в случае исключительного состояния вернитесь к заполняющему контенту или сообщению об ошибке. Пример подсказки приведен ниже.

  ```
  В компоненте @xxx.tsx переменные xx и yy могут быть пустыми. Необходимо
  добавить логику проверки на пустое значение и отобразить уведомление
  о пустом состоянии или заполняющее изображение по умолчанию.
  ```

- **Злоупотребление CSS**. Нативный CSS использует пары «ключ–значение» конкретных свойств для выражения визуальных эффектов элементов страницы, но этот метод имеет высокую сложность контекста для ИИ и подвержен возникновению ошибок. Напротив, Tailwind использует

различные атомарные имена классов для выражения наборов правил определенного стиля, что делает информацию более сфокусированной и легкой для понимания ИИ. Поэтому в техническом планировании проекта рекомендуется по возможности использовать Tailwind для выражения стилевых эффектов. Пример подсказки приведен ниже.

```
Доля кода CSS в файле @xxx.css слишком велика, рекомендуется по возможности
перейти на реализацию с помощью Tailwind.
```

- **Отсутствие промежуточных состояний**. Удобство пользовательского интерфейса основано не только на корректном отображении контента, но и на разумной обработке промежуточных и аварийных состояний. Поэтому при переходе страницы в состояние загрузки или при возникновении ошибок (ошибки API, ошибки данных, ошибки загрузки ресурсов и т. д.) необходимо отображать на странице промежуточные переходные эффекты, чтобы пользователь лучше понимал актуальное состояние системы. Пример подсказки приведен ниже.

```
Оптимизировать логику обработки состояний компонента @xxx.tsx: добавить
базовый экран при первом входе, включить состояние загрузки при частичной
загрузке данных, а при ошибках интерфейса или данных использовать всплывающие
окна или компоненты представления для отображения сообщений об ошибках.
```

Подобно разработке на стороне сервера, на данном этапе проверки не следует стремиться к идеальному коду, а также нет необходимости переделывать или абстрагировать все детали. Основные цели этого этапа заключаются в следующем:

- обеспечить разумную структуру и четкую логику, облегчающие последующие итерации и совместную работу;
- определить четкие функции компонентов и обеспечить стабильный поток данных;
- сохранить уровень сложности в пределах управляемого диапазона и избежать преждевременной оптимизации.

Другими словами, разработчики должны уделять внимание тому, насколько код написан «понятно», а не «сложно». Если структура ясна, а логика целостна, ИИ сможет и в дальнейшем помогать совершенствовать код в процессе технического обслуживания.

6.4.4. Вывод веб-страницы с помощью ИИ в соответствии с ожиданиями

По завершении первоначальной проверки кода и оптимизации структуры процесс разработки переходит к этапу отладки страниц. Основная задача на этом этапе состоит в запуске сгенерированных страниц, проверке их фактического отображения в реальной среде браузера и внесении исправлений на основе полученных результатов.

Однако процесс отладки не всегда проходит гладко, и в контексте вайб-программирования особенно часто возникают проблемы визуального характера. Например, в ранних версиях проекта «Генератор контента Xiaohongshu»

при первом запуске страницы отображался довольно грубый результат: все компоненты выглядели как стандартные элементы веб-браузера, без какого-либо макета и эстетического оформления. Вся страница выглядела как статическая HTML-структура без какого-либо оформления, где отсутствовали все необходимые элементы пользовательского интерфейса, такие как цветовое оформление, верстка и отступы, как показано на рис. 6-18.



Рисунок 6-18. Содержимое страницы ранней версии проекта «Генератор контента Xiaohongshu»

Первопричина таких проблем заключается не в отсутствии функциональности, а в том, что сгенерированный код не был правильно связан с системой стилей: дефекты рендеринга вызваны техническими деталями, такими как нарушение настроек Tailwind или отсутствие ожидаемых стилевых ресурсов в структуре компонентов.

В традиционном процессе разработки такие ошибки стилей обычно удается быстро обнаружить и исправить с помощью инструментов разработчика в браузере. Однако сценарии вайб-программирования предполагают, что задачу исправления лучше поручить ИИ-инструменту. Сложность заключается в том, что ИИ-инструменты, такие как Cursor, и браузеры представляют собой две совершенно разрозненные системы: Cursor не может напрямую «видеть» результат рендеринга в браузере, как это делает разработчик, и тем более не способен автоматически распознавать проблемы визуального оформления. Поэтому если вы хотите использовать Cursor для исправления подобных проблем с интерфейсом, основная задача заключается в том, как сообщить ему о текущем состоянии браузера, чтобы Cursor получил достаточно информации для анализа и принятия решений. В частности, можно рассмотреть два следующих способа реализации такого информационного канала.

1. Использование скриншота браузера

Лучший и наиболее простой способ передать Cursor информацию о рабочем состоянии браузера – отправить ему скриншот. Основываясь на изображении, Cursor сможет определить структуру страницы и выявить проблемы, а затем исправить код. Хотя этот метод ограничен по объему информации, он чрезвы-

чайно прост в использовании, не требует дополнительных затрат на обучение и подходит для первоначальной диагностики проблем со стилями.

Например, в приведенном выше примере с аномалией мы можем просто вставить скриншот страницы с отсутствующими стилями, показанный на рис. 6-18, в диалоговое окно Cursor (см. рис. 6-19), чтобы он определил существование таких проблем, как отсутствие импорта файла стилей, неподключенные компоненты или аномалии в структуре DOM, и попытался сгенерировать исправление.

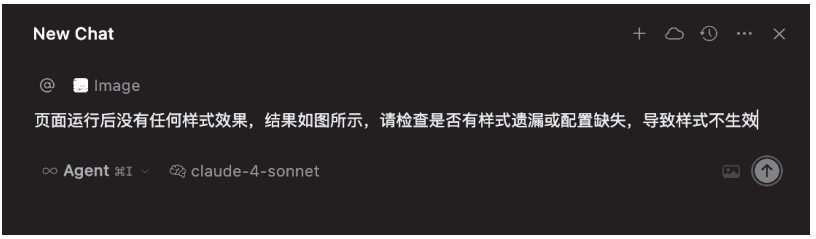


Рисунок 6-19. Использование скриншота браузера

Впрочем, поскольку Cursor на базовом уровне опирается на большие языковые модели, такие как Claude и GPT, эти модели, хотя и обладают значительными преимуществами в обработке естественного языка, построении логической структуры и анализе текстового контекста, все же имеют изначальные недостатки в области распознавания изображений и анализа графической семантики. Даже если модель обладает начальными способностями по обработке изображений, ее восприятие деталей страницы, состояния компонентов и даже сбоев в макете будет гораздо менее точным, чем результаты, полученные реальным разработчиком с помощью инструментов отладки.

По сути, взаимодействие с изображениями является средством подсказки с «низкой семантической плотностью», подходящим для решения базовых проблем с четкой структурой и ясными типами ошибок, таких как недоработка стиля кнопок, пустая страница, неправильный цвет шрифта и другие явные аномалии. Однако при возникновении проблем более высокого уровня, таких как сбой состояния компонентов, отказ сложной логики или нарушение адаптивной верстки, использование одних лишь скриншотов не позволяет решить задачу, а может даже ввести модель в заблуждение и привести к ошибочным выводам, что в итоге увеличивает затраты на отладку.

Поэтому можно сказать, что метод скриншотов браузера подходит для «быстрой обратной связи по аномалиям статических страниц», но не подходит для решения «динамических логических ошибок со сложным поведением». Его можно сравнить с небольшим гаечным ключом в инструментарии разработчика, но он не является панацеей. В сложных сценариях рекомендуется использовать MCP для получения информации о странице, чтобы помочь модели глубже проанализировать контекст страницы.

2. Использование службы MCP браузера

Наряду со скриншотами существует более эффективный и структурированный способ передачи информации о рабочем состоянии браузера в Cursor: считывание структурных данных страницы через протокол контекста модели (Model Context

Protocol, MCP). Основное преимущество этого подхода заключается в отсутствии зависимости от распознавания изображений или выводов на основе скриншотов. Вместо этого он использует нативный интерфейс браузера для прямого доступа к структурированному контенту страницы – объектной модели документа (Document Object Model, DOM). Это специальный стандарт для описания структуры контента браузера, который позволяет представить узловые элементы страницы, состояние стилей и поведение скриптов в виде иерархического дерева.

Поскольку большие языковые модели хороши в обработке структурированного текстового контента, их способность понимать входные данные типа DOM – частично структурированные, частично семантические – намного превосходит их способность анализировать изображения. Поэтому использование MCP для предоставления контекста страницы обеспечивает более высокую точность и более достоверный отклик, что особенно удобно для отладки страниц со сложными взаимодействиями, многоуровневой вложенностью компонентов или переключением между несколькими состояниями.

В настоящее время на рынке доступно несколько реализаций MCP для браузеров, среди которых можно выделить BrowserTools. Это промежуточное ПО для управления браузером, специально разработанное для ИИ-приложений, которое поддерживает стандартизированную коммуникацию между моделью и реальной пользовательской средой браузера. В отличие от автоматизированных браузеров, таких как Playwright, в BrowserTools можно напрямую использовать уже авторизованную пользовательскую среду браузера. Это означает, что ИИ может выполнять операции непосредственно на реальных страницах без необходимости моделирования или авторизации, такие как переход по навигации, заполнение форм, извлечение данных, проверка стилей и даже отладка производительности.

В то же время процесс развертывания BrowserTools является несколько трудоемким. Для его использования необходимо выполнить такие шаги, как загрузка плагина, установка в браузере, запуск службы из командной строки и настройка MCP в Cursor. Хотя при первом использовании BrowserTools требуются некоторые усилия для изучения и настройки, после установления соединения его мощные функции в сценариях отладки реальных проектов значительно повысят эффективность разработки. Процесс настройки BrowserTools включает следующие 6 шагов.

(1) **Загрузка плагина**. Перейдите на страницу релизов BrowserTools на GitHub (https://github.com/AgentDeskAI/browser-tools-mcp/releases) и загрузите плагин для браузера, как показано на рис. 6-20.



Рисунок 6-20. Загрузка пакета плагинов BrowserTools

(2) **Установка плагина**. Разархивируйте загруженный пакет плагинов, откройте страницу *chrome://extensions/* в браузере Chrome (как показано на рис. 6-21) и нажмите кнопку **Load unpacked** (Загрузить в распакованном виде) для инсталляции плагина.

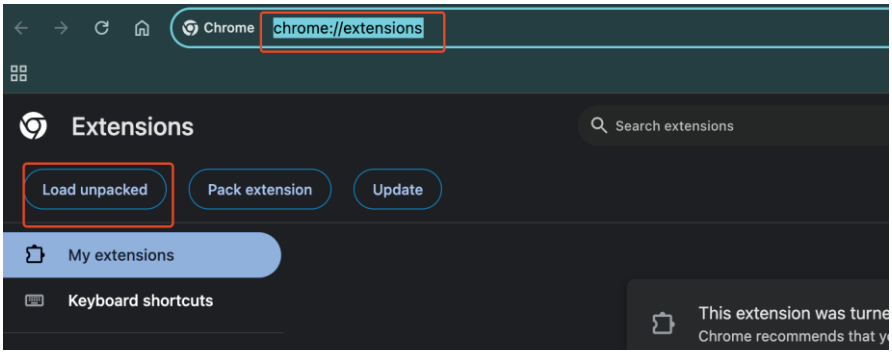


Рисунок 6-21. Установка плагина для браузера

(3) **Запуск MCP**. Откройте терминал командной строки и введите следующую команду:

```
npx -y @agentdeskai/browser-tools-server@latest
```

(4) **Настройка MCP**. Перейдите на страницу настроек **Cursor MCP**, нажмите **Add Custom MCP** (Добавить пользовательский MCP), как показано на рис. 6-22, и в появившемся окне редактора введите следующие настройки.

```
{
  "mcpServers": {
    "browsertools": {
      "command": "npx",
      "args": ["-y", «@agentdeskai/browser-tools-mcp@latest»]
    }
  }
}
```

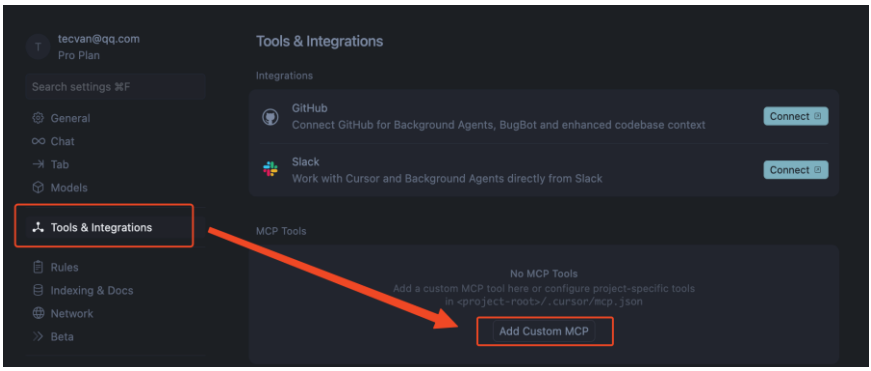


Рисунок 6-22. Настройка MCP

(5) **Тестирование**. Во вкладке браузера, которую необходимо протестировать, нажмите клавишу **F12** для запуска инструмента разработчика, как показано на рис. 6-23.

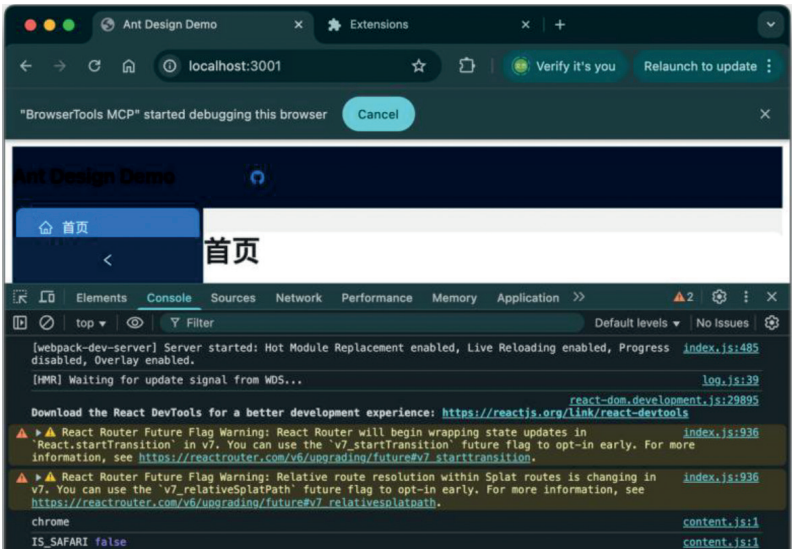


Рисунок 6-23. Пример инструмента разработчика Chrome

(6) **Активация**. В Cursor для активации действий MCP используйте такие ключевые слова, как «браузер», например:

```
Открыть в браузере страницу http://localhost:5173, провести анализ причин
неработоспособности стилей страницы и исправить их.
```

На этом настройка BrowserTools завершена. Теперь можно использовать ключевое слово «browser» для управления курсором и выполнения действий в браузере, как приведено в примере ниже.

- **Анализ производительности страницы**. Как упоминалось ранее, большая языковая модель с помощью MCP может считывать данные Lighthouse о производительности страницы и т. д., а после тщательного анализа предоставлять сводную информацию о производительности. На этом основании можно даже попросить большую языковую модель выполнить исправления.

  ```
  Проанализировать производительность страницы, открытой в браузере,
  проанализировать данные Lighthouse о производительности и определить
  проблемные места.
  ```

- **Исправление проблем со стилями**. Модель может напрямую считывать настройки отступов (`margin`) элемента `body` в DOM страницы, цепочку наследования стилей и фактический результат рендеринга, быстро выявляя проблемы с переопределением или неработающими стилями, а также предлагая точные варианты исправления.

  ```
  На странице у элемента body есть лишний отступ (margin), необходимо исправить.
  ```

- **Выявление причин сетевых сбоев**. ИИ может автоматически извлекать заголовки и содержимое неудачных запросов, определять тип ошибки и прогнозировать возможные сбои на стороне сервера или проблемы с конфигурацией, предоставляя разработчикам направления для дальнейшего поиска и устранения неисправностей.

```
На странице наблюдается множество сетевых сбоев. Необходимо тщательно
проанализировать информацию о запросах и ответах, определить первопричину
проблемы и принять меры по ее устранению.
```

В процессе реальной разработки именно благодаря предоставленной BrowserTools возможности наблюдения за браузером удалось постепенно оптимизировать страницу из состояния полной потери стилей и беспорядочного наложения компонентов до состояния структурированного оформления и плавной работы современного интерфейса. Путем постоянного ввода в модель данных о реальном состоянии страницы, корректировки стратегии рендеринга и структуры стилей в конечном итоге была создана веб-страница с четкой структурой и контролируемым оформлением, как показано на рис. 6-24.



Рисунок 6-24. Веб-страница с четкой структурой и управляемым стилем

В этом и заключается реальное преимущество сочетания инструментов структурированной отладки с большими языковыми моделями: это не просто инструмент для написания кода, это помощник разработчика с возможностью синхронизации, распознавания состояния и автоматической итерации.

Таким образом, сложность этапа отладки заключается не только в выявлении проблем, но и в правильной формулировке этих проблем для модели – именно в этом и состоит огромное отличие вайб-программирования от традиционных моделей разработки. Нам нужно не только овладеть техниками отладки, но и научиться эффективно общаться с ИИ, чтобы модель могла принимать правильные решения и дополнять код в ограниченном контексте.

6.4.5. Вызов реальных серверных интерфейсов

После предварительного построения структуры и последовательных итераций мы получили код страницы с нормальным отображением в браузере. Однако возможности страницы, ограниченные лишь статическим отображением, недостаточны для поддержки полноценного бизнес-процесса. Для запуска в действие всей системы необходимо выполнить еще одну ключевую задачу: подключить интерфейсы бэкенд-сервисов и наладить обмен данными между фронтендом и бэкендом. Этот этап включает в себя следующие три основных шага.

(1) Преобразование серверных интерфейсов в более структурированную форму определений типов TypeScript. Прежде всего в плане технической архитектуры фронтенд и бэкенд всегда являются двумя полностью независимо работающими системами, которые обмениваются данными только через общие сетевые протоколы, такие как HTTP. Сам по себе HTTP является протоколом слабой привязки, а соглашения об интерфейсах обычно представлены в виде текста или документации и не имеют строгой проверки типов. Такой механизм слабой связанности приводит к частому возникновению ошибок интерпретации при генерации кода большими языковыми моделями, особенно при определении путей интерфейсов, структуры параметров или возвращаемых значений. Эти ошибки, хотя и незначительные, но критичные, в конечном итоге могут повлиять на стабильность всей системы.

Для решения данной проблемы можно предложить большой языковой модели сначала выполнить структурированное моделирование серверных интерфейсов с помощью следующего промпта.

```
Необходимо провести тщательный анализ интерфейсов из каталога packages/
server/src, включая URL, HTTP-методы, входные и выходные параметры и т. д.,
и преобразовать их в четко определенные структуры типов TypeScript. Одновременно
следует написать функции с использованием axios для вызова этих интерфейсов.
```

Сохраните код в каталоге *packages/web/src/api*.

После этого Cursor сгенерирует для каждого интерфейса соответствующие определения типов в каталоге *packages/web/src/api*, а также сгенерирует функции вызова, как показано на рис. 6-25.

Компоненты на странице интерфейса не зависят от подробностей запроса. Достаточно вызвать эти функции, чтобы инициировать полностью контролируемое взаимодействие с реальными данными. По сути, такая структура представляет собой типичную модель многоуровневой архитектуры: благодаря выделению логики вызова интерфейсов за пределы уровня представления улучшаются границы модулей и их внутренняя согласованность, повышается удобство чтения, удобство обслуживания и масштабируемость кода. Такой подход широко применяется в современных архитектурах при разработке ПО.

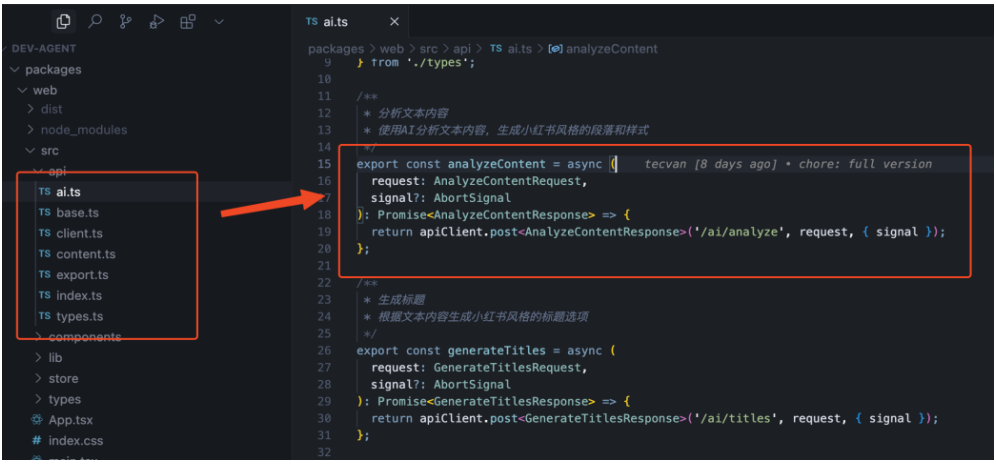


Рисунок 6-25. Пример определения типов интерфейсов

(2) Модифицируйте код фронтенда, удалите всю логику имитации данных и замените ее вызовами реальных интерфейсов. После завершения абстрагирования кода уровня API удалите всю логику имитации (mock) на странице и реализуйте взаимодействие с реальными данными.

```
Необходимо изменить код в packages/web/src, чтобы удалить всю логику имитации
интерфейсов и заменить ее вызовами реальных интерфейсов, предоставляемых
в packages/web/src/api.
```

После этого можно перезапустить страницу и провести отладку всего процесса взаимодействия с данными. На этом этапе основной акцент при отладке делается не на стилях или расположении компонентов, а на соответствии ожиданиям по инициации и ответам на сетевые запросы. Для анализа рекомендуется использовать встроенные инструменты разработчика браузера. Например, в браузере Chrome достаточно нажать сочетание клавиш **F12**, чтобы открыть панель отладки, показанную на рис. 6-26, перейти на вкладку **Network** (Сеть) и в режиме реального времени отслеживать запуск различных запросов к интерфейсам, код состояния, ответные данные и задержки на странице.

Инструменты разработчика браузера представляют собой комплексный и сложный набор средств для отладки веб-приложений, освоение которого требует немало времени. Здесь мы рекомендуем простой способ их использования, как показано на рис. 6-26: достаточно перейти на вкладку **Network** в DevTools, а затем проверить, отправляет ли страница сетевой запрос в нужное время и получает ли она нормальный ответ. При возникновении каких-либо проблем можно снова воспользоваться Cursor и BrowserTools для доработки кода, например:

```
Необходимо проанализировать сетевые журналы страницы в браузере. После нажатия
пользователем кнопки «Отправить» должен немедленно отправляться запрос analyze.
Необходимо тщательно проверить код и исправить проблему.
```

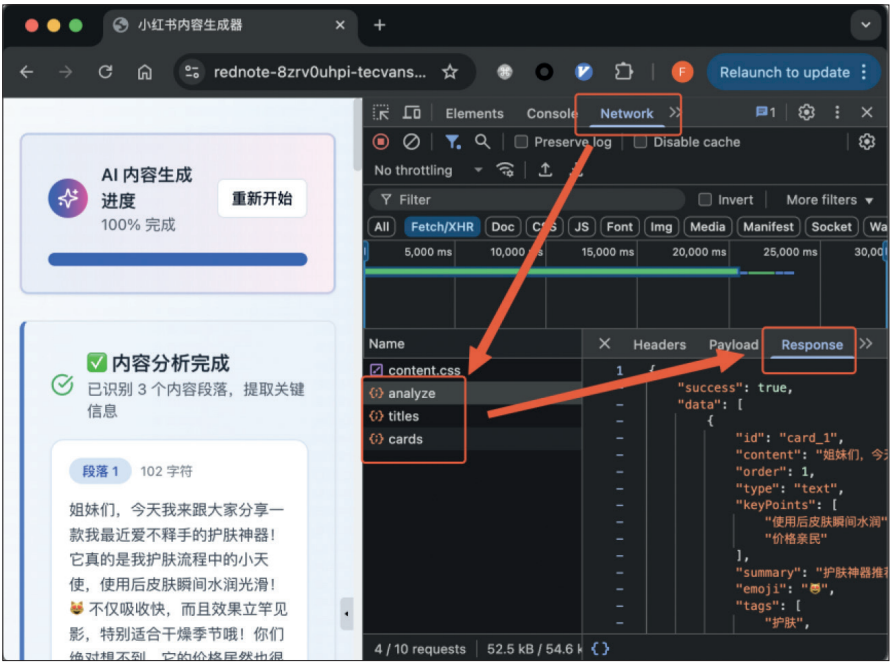


Рисунок 6-26. Панель отладки сетевых запросов

(3) Экспорт логов сеансов. По завершении разработки системы и успешной отладки функциональности остается еще один важный, но часто упускаемый из виду этап: экспорт логов взаимодействия с большой языковой моделью. Поскольку большие языковые модели не обладают «долговременной памятью», каждый новый диалог начинается с нуля (то есть существующий контекст теряется). Если в дальнейшем вам понадобится продолжить развитие системы на основе существующего кода, исправить детали или добавить новые функции, понадобится заново загрузить в систему существующий контекст. Экспорт записей взаимодействий позволяет полностью сохранить ключевые диалоги, ответы модели, исправления ошибок, архитектурные решения и другие важные сведения, накопленные в ходе всего процесса разработки.

Экспорт записей диалогов в Cursor очень прост: нажмите кнопку «...» в правом верхнем углу панели **Agent**, как показано на рис. 6-27, выберите в появившемся меню опцию **Export Chat** (Экспорт чата), и полная история чата будет экспортирована в локальный файл. Рекомендуется сохранять этот файл в корневом каталоге проекта и управлять им вместе с репозиторием кода.

Если в будущем понадобится возобновить процесс разработки, достаточно импортировать этот файл записей в Cursor, и большая языковая модель сможет восстановить весь контекст действующей системы, включая заданные вами вопросы, пути пояснений модели, предложения по изменению на каждом этапе и окончательный вариант реализации, что обеспечит контекстную связность и поведенческую согласованность нового цикла разработки.

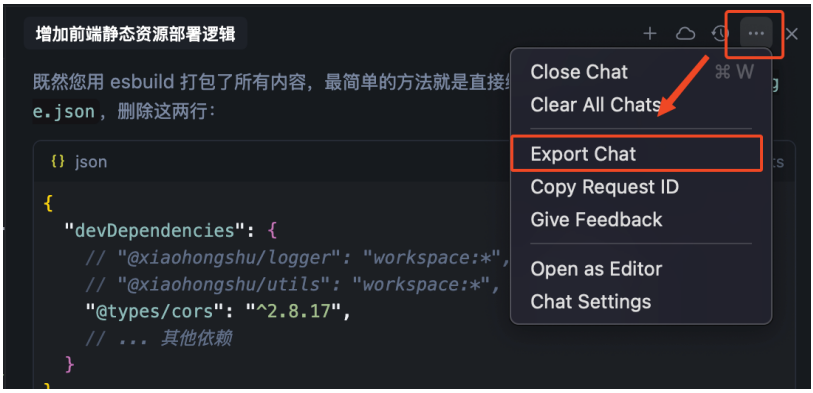


Рисунок 6-27. Экспорт записей диалога

6.5. Развертывание приложения

После завершения разработки бэкенд-интерфейса и веб-системы, описанных в разделах 6.3 и 6.4, мы получили практически готовое полнофункциональное веб-приложение. Однако завершение разработки – это лишь первый шаг. Далее предстоит еще одна важная задача – развертывание приложения. Нам необходимо упаковать локально разработанную программу и разместить ее в облаке, чтобы она была доступна через интернет и могла предоставлять услуги большему количеству пользователей.

Однако с точки зрения программной инженерии развертывание – это далеко не просто операция загрузки. В действительности это зачастую один из самых сложных и технически трудоемких этапов всего жизненного цикла ПО. Чтобы гарантировать стабильную, эффективную и безопасную работу приложения в реальной производственной среде, при разработке схемы развертывания обычно необходимо учитывать следующий комплекс системных факторов:

- техническая архитектура проекта и структура зависимостей;
- рабочая среда и ограничения ресурсов целевой платформы развертывания;
- географическое распределение пользователей и их поведение при доступе;
- политика безопасности, включая настройки сетевого брандмауэра, контроль прав доступа и шифрование данных;
- механизмы мониторинга в режиме реального времени и восстановления после сбоев для онлайн-систем;
- управление ветвями кода и стратегии отката в условиях совместной работы;
- автоматизированные процессы CI/CD.

Чтобы создать качественную систему развертывания, требуется профессиональная команда DevOps, которая будет заниматься ее долгосрочным развитием и обслуживанием. Инженерные затраты и техническая сложность такого проекта значительно выходят за рамки содержания и целей этой книги. Поэтому, чтобы сосредоточиться на практической и применимой стороне вопроса, в этой главе на примере проекта «Генератор контента Xiaohongshu» и с учетом

современных практик ведущих облачных платформ будет представлена вполне работоспособная и относительно простая схема развертывания, которая даст читателям возможность освоить метод запуска вайб-проектов в кратчайшие сроки. Это решение не нацелено на создание сложной или полнофункциональной системы развертывания, вместо этого оно ориентировано на построение базового замкнутого цикла – от локальной среды до онлайн-среды в качестве отправной точки для последующего расширения инженерных возможностей.

В этом разделе также будет представлена система GitHub Actions с подробным объяснением процесса построения непрерывного развертывания, позволяющего автоматически публиковать каждое изменение кода в онлайн-среде. Это обеспечит реальную автоматизацию, бесперебойность и надежность развертывания кода и приложений.

6.5.1. Понимание логики развертывания кода

Развертывание полнофункционального полного стека приложения в онлайн-среде – это больше, чем просто загрузка кода на сервер. Сюда входят упаковка, распространение и запуск фронтенд-ресурсов и бэкенд-сервисов, а также ряд системных задач, таких как контроль доступа, настройка среды и политики безопасности.

Суть развертывания заключается в устойчивом, эффективном и безопасном предоставлении услуг реальным пользователям в облаке на основе локально разработанной системы. Поэтому разработчику необходимо обладать базовыми навыками в следующих трех областях:

- четкое понимание модели ресурсов и жизненного цикла фронтенд- и бэкенд-кода;
- знакомство с распространенными архитектурами развертывания, понимание особенностей различных сред выполнения и сценариев их применения;
- достаточное понимание вопросов безопасности и наличие практических навыков управления конфиденциальной информацией (например, API-токены).

1. Понимание модели развертывания фронтенда

Несмотря на то что в процессе разработки фронтенда мы используем современные инженерные инструменты, такие как TypeScript, Tailwind CSS и Vite, их роль по-прежнему сосредоточена в основном на абстрагировании и повышении эффективности на этапе разработки. При фактическом развертывании фронтенд-код компилируется (build) в набор стандартных статических ресурсов, включая файлы HTML, CSS и JavaScript, а также медиаматериалы, такие как шрифты и изображения. Главная особенность этих ресурсов заключается в их независимости от вычислительной логики во время выполнения, поэтому они могут храниться и распространяться в неизменном виде. Если не учитывать сложные сценарии, такие как серверная прорисовка (Server Side Rendering, SSR), основной целью развертывания фронтенда является эффективная передача этих статических ресурсов в браузер пользователя.

Существует два основных способа достижения этой цели: первый – обеспечение логики распространения статических ресурсов на стороне сервера; второй – использование сети доставки контента (Content Delivery Network, CDN) для кеширования и ускорения в разных регионах. Для упрощения процесса развертывания в нашем проекте используется первый способ, то есть обеспечение функции распространения статических ресурсов фронтенда на стороне сервера. Конкретные шаги реализации приведены ниже.

(1) В серверном проекте необходимо добавить скрипт сборки *build.sh*, который последовательно выполняет операции сборки бэкенд- и фронтенд-ресурсов и копирует результаты сборки фронтенда в каталог *packages/server/public*.

(2) На уровне маршрутизации сервера необходимо добавить логику совместимости с одностраничными приложениями (SPA): для запросов, не начинающихся с */api*, сначала выполняется поиск соответствующего статического ресурса в каталоге *public*. Если поиск не дал результатов, возвращается файл ресурса по умолчанию *index.html*.

Соответствующие текстовые промпты приведены ниже.

```
Необходимо добавить логику развертывания статических ресурсов фронтенда. Для
начала следует добавить в серверный проект скрипт build.sh, который выполняет
следующие действия:
1. сборка бэкенд-ресурсов;
2. сборка фронтенд-ресурсов и копирование результатов сборки в каталог
   packages/server/public.

Затем необходимо изменить логику маршрутизации в packages/server, чтобы
реализовать сервис одностраничного приложения: для запросов, не начинающихся
с /api, сначала нужно найти соответствующий статический ресурс в каталоге
public. Если он найден, то его нужно сразу вернуть, в противном случае следует
вернуть файл ресурса по умолчанию index.html.
```

После этого нужно запустить вновь созданный скрипт *packages/server/build.sh*, чтобы скомпилировать код фронтенда в файлы статических ресурсов и скопировать их в нужное место. Пример кода приведен ниже.

```
#!/bin/bash

set -ex

echo "1. Сборка бэкенд-ресурсов..."
pnpm --filter @xiaohongshu/server build

echo "2. Сборка фронтенд-ресурсов..."
pnpm --filter @xiaohongshu/web build

echo "3. Создание каталога статических ресурсов..."
mkdir -p packages/server/public

echo "4. Копирование результатов сборки фронтенда в каталог статических
ресурсов бэкенда..."
cp -r packages/web/dist/* packages/server/public/

echo "Сборка завершена! Ресурсы фронтенда скопированы в каталог packages/
server/public"
```

2. Понимание модели развертывания бэкенда

По сравнению с развертыванием фронтенда, развертывание серверной части является более сложным процессом. Серверные программы требуют не только настройки среды выполнения, но еще и обеспечения непрерывной работы, асинхронной диспетчеризации, сохранения данных и взаимодействия с внешними сервисами. В зависимости от среды выполнения современные серверные системы в основном подразделяются на следующие три модели развертывания.

Виртуальная машина (Virtual Machine). Обеспечивает полноценную среду операционной системы, обладает максимальной степенью контроля и удобства настройки, но требует значительных ресурсов и имеет высокие затраты на обслуживание.

Контейнеры (Container). На примере Docker: предоставляют легкий и легко портируемый механизм упаковки приложений, обладают такими преимуществами, как быстрый запуск, низкое потребление ресурсов и стабильность работы.

Бессерверные вычисления (Serverless Computing). Разработчикам не нужно заботиться о базовых серверах: достаточно загрузить код функции, и облачная платформа автоматически выполнит планирование, масштабирование и запуск. Обладает такими преимуществами, как оплата по факту использования, упрощенное развертывание и высокая эластичность.

Для такого серверного проекта, как описанный в книге «Генератор контента для Xiaohongshu», который реализован на Node.js, имеет средний масштаб и четко определенные функции, бессерверные функции являются в настоящее время оптимальным вариантом развертывания. Это избавляет от сложностей управления серверами и позволяет автоматически масштабировать ресурсы в зависимости от нагрузки, что значительно снижает затраты на эксплуатацию и обслуживание.

Обратите внимание: вне зависимости от выбранной модели развертывания перед запуском бэкенд-код необходимо скомпилировать в исполняемую версию. Для обеспечения удобства обслуживания и масштабируемости в данном проекте используется инструмент esbuild, который упаковывает бэкенд-код в единый исполняемый файл *bundle.js*. Этот подход особенно удобен для модульной организации кода в архитектуре Monorepo: даже если впоследствии сервис будет разбит на несколько пакетов, их можно будет скомпилировать и развернуть как единое целое.

Соответствующие настройки можно выполнить с помощью следующих промптов.

```
Необходимо изменить способ сборки бэкенда и использовать esbuild для
упаковки серверного кода в один файл bundle.js, что упростит распространение
и развертывание.
```

После добавления этого шага при развертывании для запуска достаточно загрузить на облачную платформу сгенерированный файл *bundle.js* и необходимые настройки, без необходимости добавлять избыточный исходный код или зависимые компоненты. Результат упаковки показан на рис. 6-28.

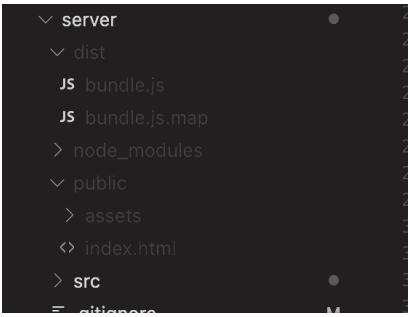


Рисунок 6-28. Пример упаковки серверного кода

3. Управление конфиденциальной информацией

Помимо кода фронтенда и бэкенда, есть еще одна категория источников угроз, которую очень легко упустить из виду, но которая имеет чрезвычайно важное значение. Это API-ключи (или токены), например токен OpenAI API, о котором мы говорили в разделе 6.1. Такие ресурсы являются своего рода «цифровыми удостоверениями», позволяющими приложению обращаться к внешним сервисам (таким как базы данных, сторонние сервисы, ИИ-модели и т. д.). В ситуации их утечки в лучшем случае это приведет к материальным убыткам, а в худшем может повлечь за собой злонамеренное изменение данных третьими лицами, что напрямую затрагивает безопасность данных всего приложения и даже его коммерческие результаты.

Поэтому при работе с такими конфиденциальными данными необходимо проявлять особую бдительность, принимать все необходимые меры безопасности на этапах разработки и развертывания, а также соблюдать следующие правила.

- **Избегайте жесткого кодирования**. Ни в коем случае не вписывайте API-ключ непосредственно в исходный код: в случае утечки кода ключ будет полностью раскрыт, что создаст серьезную угрозу безопасности.
- **Избегайте раскрытия на стороне клиента**. Ни в коем случае не размещайте API-ключ в клиентской среде, такой как браузер или мобильное приложение. Злоумышленники могут получить ключ посредством анализа клиентского кода и отправить запрос от вашего имени, что может привести к непредвиденным расходам или утечке данных. Все запросы, связанные с конфиденциальными API-ключами, должны проходить через безопасный сервер брокера.
- **Избегайте раскрытия в логах**. Ни в коем случае не выводите в логах приложения информацию об API-ключах, иначе злоумышленники могут взломать их после кражи логов.
- **Избегайте отправки ключей в репозитории кода**. Даже в приватных репозиториях следует избегать прямой отправки API-ключей. Злоумышленники постоянно сканируют публичные репозитории кода в поисках учетных данных, и даже кратковременное раскрытие может привести к утечке.

Как же тогда можно обеспечить безопасную передачу такой конфиденциальной информации в приложение? В настоящее время основной подход заключается в использовании переменных среды для передачи ключей. Например:

```
OPENAI_API_KEY=sk-proj-xxx node dist/bundle.js
```

Приведенный выше код вызывается через переменную среды `process.env.OPENAI_API_KEY`. Этот механизм изначально поддерживается всеми крупными облачными платформами, он безопасен и удобен в обслуживании. В разделе 6.5.3 будут описаны настройки переменных среды на реальных платформах развертывания для автоматизации управления лексическими единицами.

6.5.2. Развертывание приложения на Vercel

В настоящее время на рынке существует множество облачных платформ, которые можно использовать для хостинга современных веб-приложений, включая, помимо прочего, Vercel, Netlify, Render, Cloudflare Pages и AWS Amplify. Каждая из них имеет свои особенности механизмов развертывания, управления ресурсами и возможностей масштабирования, но общий процесс развертывания у них очень схож.

В данном разделе в качестве демонстрационной платформы выбран Vercel. Мы подробно расскажем об установке полнофункционального проекта «Генератор контента для Xiaohongshu» в облаке. Благодаря поддержке современных фронтенд-технологий (таких как React, Next.js, Vue и т. д.), упрощенному процессу развертывания и функциям CDN платформа Vercel в последние годы стала одной из наиболее популярных среди разработчиков. Она отлично подходит для интегрированных проектов с фронтендом и бэкендом, построенных на Node.js.

Концепция развертывания на Vercel очень четкая: снизить сложность развертывания за счет автоматизации процессов, чтобы разработчики могли сосредоточиться на бизнес-логике, а не на управлении инфраструктурой. Конкретный процесс развертывания выглядит следующим образом.

1. Регистрация и настройка

Сначала необходимо посетить официальный сайт Vercel и пройти процесс регистрации учетной записи. Платформа поддерживает вход в систему одним щелчком мыши с помощью учетных записей GitHub, GitLab и других платформ, что упрощает последующую синхронизацию репозиториев кода.

После завершения регистрации система предложит вам создать «команду» (Team). Все управление проектами, развертывание, контроль прав доступа и т. д. будут осуществляться в рамках этого командного пространства. Чтобы войти в панель управления командой, нажмите на иконку в правом верхнем углу, как показано на рис. 6-29, выберите в появившемся меню **Create Team** (Создать команду) и заполните обязательные поля.

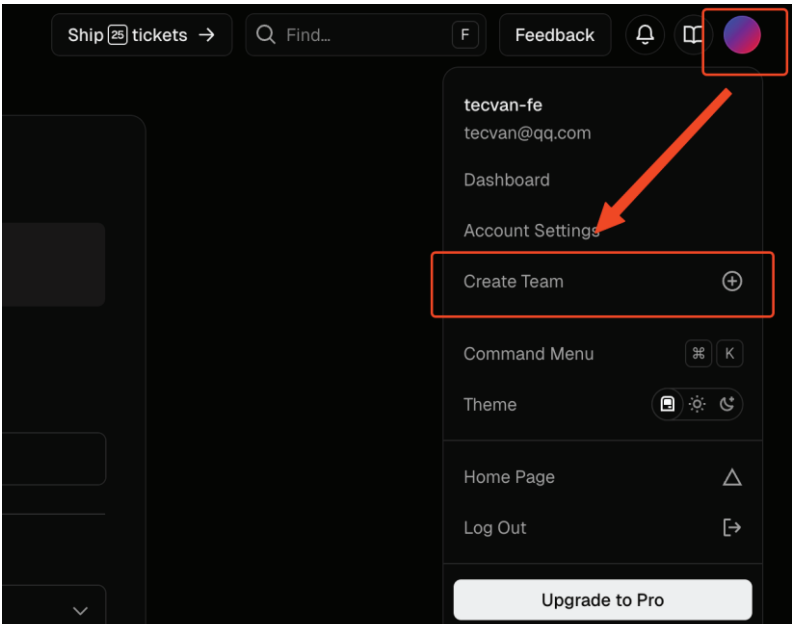


Рисунок 6-29. Панель управления командой

2. Установка инструмента командной строки

Для установки инструмента командной строки Vercel CLI, предоставляемого Vercel, чтобы обеспечить последующее развертывание и управление проектом, выполните в локальном терминале следующую команду:

```
npm i -g vercel
```

Затем выполните команду входа в систему:

```
vercel login
```

Пройдите авторизацию в соответствии с подсказками, после чего можно будет начать использовать Vercel CLI, как показано на рис. 6-30.

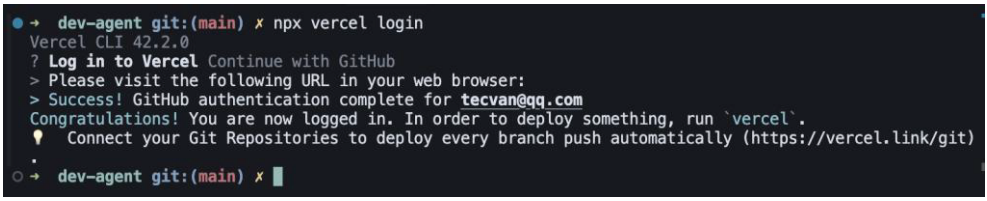


Рисунок 6-30. Вход в учетную запись Vercel

3. Настройка структуры развертывания

В корневом каталоге проекта создайте новый файл *vercel.json* и введите в него следующую конфигурацию.

```
{
  "$schema": "https://openapi.vercel.sh/vercel.json",
  "framework": null,
  "version": 2,
  "builds": [
    {
      "src": "packages/server/dist/bundle.js",
      "use": "@vercel/node"
    },
    {
      "src": "packages/server/public/**",
      "use": «@vercel/static"
    }
  ],
  "routes": [
    { "src": "/api/(.*)", "dest": "packages/server/dist/bundle.js" },
    { "src": "/health", "dest": "packages/server/dist/bundle.js" },
    { "src": "/(.*)", "dest": "packages/server/public/$1" },
    { "handle": "filesystem" },
    { "src": "/(.*)", "dest": "packages/server/public/index.html" }
  ]
}
```

Эта конфигурация выполняет такие функции:

- размещение скомпилированного бэкенд-сервиса в архитектуре бессерверных функций;
- сопоставление статических ресурсов фронтенда с общедоступным каталогом;
- поддержка маршрутизации обратного перехода в одностраничных приложениях;
- обеспечение четких точек входа для интерфейсов и проверки работоспособности.

4. Привязка и сборка проекта

Перейдите в корневой каталог проекта и выполните следующую команду для загрузки и привязки конфигурации проекта Vercel:

```
vercel pull
```

Обратите внимание: при первом выполнении этой команды (см. рис. 6-31) в опции **Link to existinct project?** (Связать с существующим проектом?) обязательно выберите **N** (Нет). В этом случае Vercel автоматически создаст новый проект и привяжет его к локальному каталогу, что позволит избежать ручной настройки.

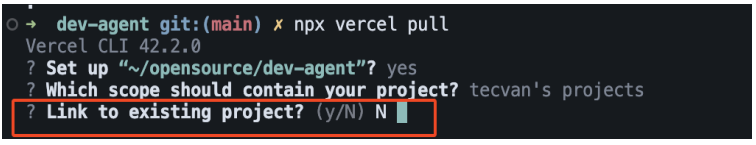


Рисунок 6-31. Отказ от привязки к существующему проекту

После успешной привязки выполните следующие команды для начала сборки ресурсов проекта вручную:

```
bash ./packages/server/build.sh
vercel build --prod
```

Этот шаг позволит скомпилировать код бэкенда и статические ресурсы фронтенда, а также сгенерировать файлы развертывания, которые по умолчанию сохраняются в каталоге *.vercel/*.

5. Окончательное развертывание

Для развертывания результатов сборки в производственную среду выполните следующую команду:

```
vercel deploy --prebuilt --prod --archive=tgz
```

По завершении развертывания Vercel выведет два важных URL-адреса, как показано на рис. 6-32.

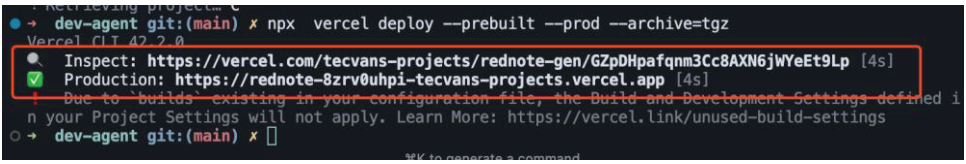


Рисунок 6-32. Генерация временной ссылки для доступа после завершения развертывания

На рис. 6-32 представлены две ссылки, значение которых поясняется ниже.

- Первая ссылка предназначена для доступа к панели управления развернутым экземпляром, как показано на рис. 6-33.
- Вторая ссылка ведет на временный адрес доступа к экземпляру в производственной среде, как показано на рис. 6-34.

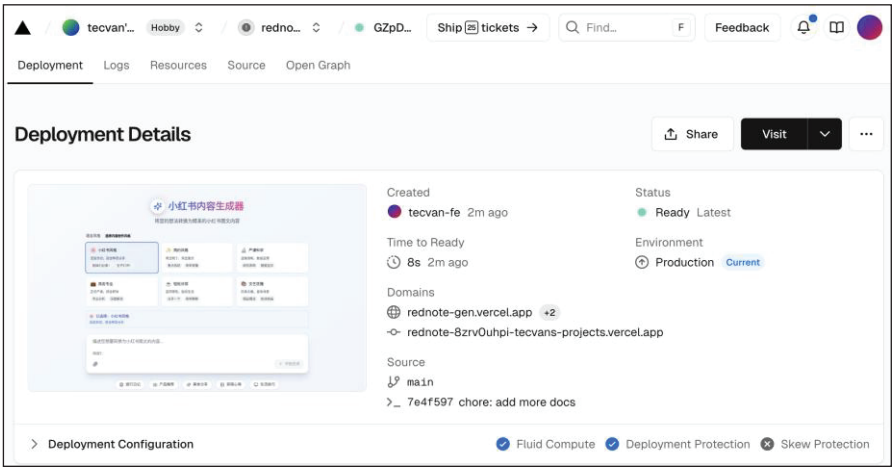


Рисунок 6-33. Панель управления экземпляром развертывания



Рисунок 6-34. Результат обращения

Теперь можно оценить фактический результат развертывания, провести тестирование страниц и отладку интерфейсов.

6. Настройка переменных среды

Если после развертывания при работе системы возникают ошибки, чаще всего это связано с отсутствием критически важных настроек (например, API-токен OpenAI). Поэтому в панели управления Vercel следует добавить переменные среды. Для этого необходимо выполнить следующие действия.

(1) Перейдите на главную страницу Vercel (см. рис. 6-35) и нажмите на соответствующий проект в разделе **Project Management** (Управление проектами), как показано на рис. 6-36.

(2) Нажмите на пункт меню **Settings** (Настройки) в верхней панели меню и выберите команду **Environment Variables** (Переменные среды).

(3) В диалоговом окне перейдите на вкладку **Create new** (Создать новую), введите ключ переменной среды (например, `OPENAI_API_KEY`) и ее значение.

(4) Нажмите кнопку **Save** (Сохранить) для сохранения и повторно выполните локально команду развертывания `vercel deploy --prebuilt -- prod --archive=tgz`. Переменные среды будут автоматически внесены в экземпляр развертывания, и системная среда перейдет в рабочее состояние.

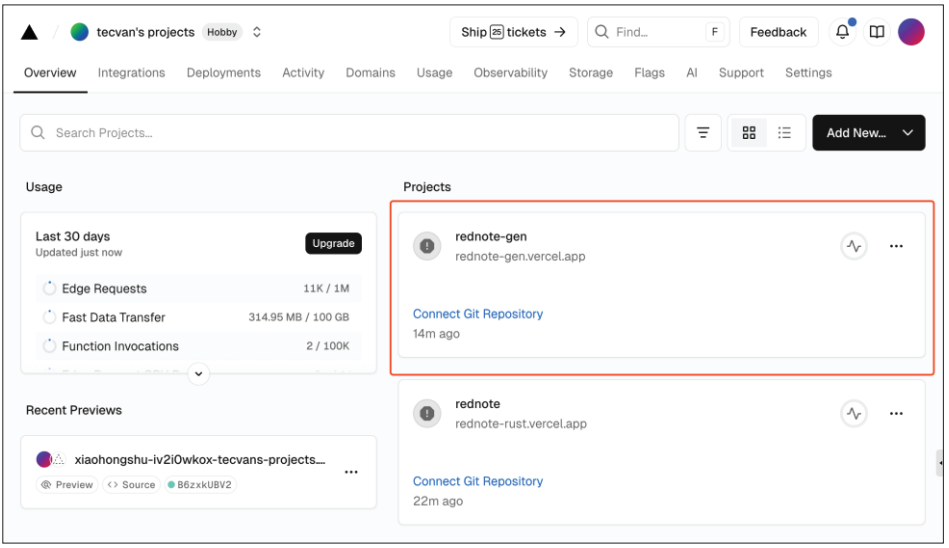


Рисунок 6-35. Главная страница Vercel

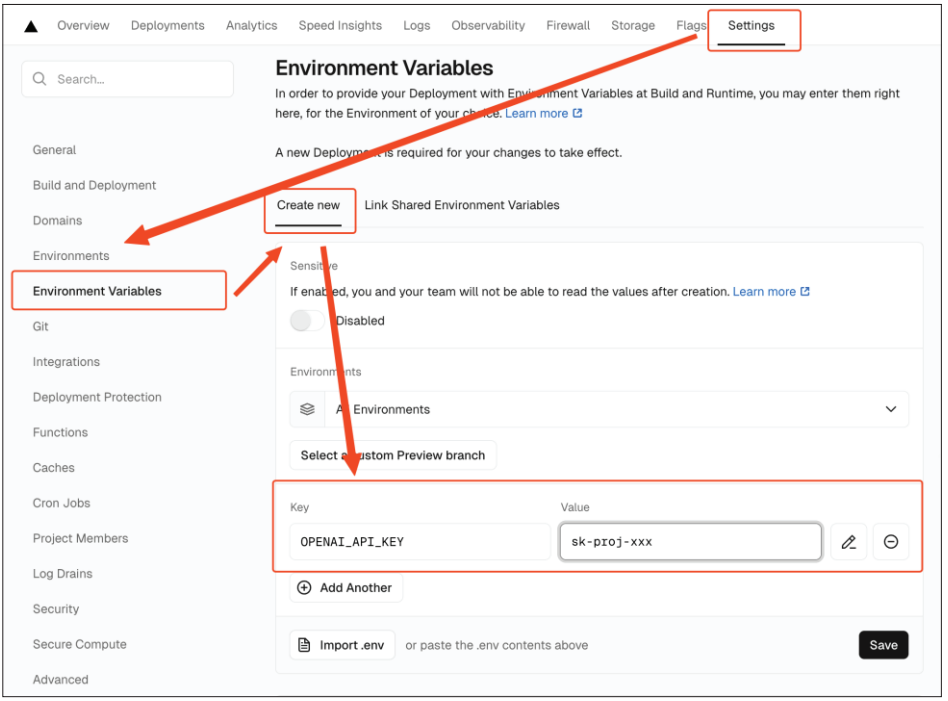


Рисунок 6-36. Страница управления проектом

6.5.3. Реализация непрерывного развертывания с помощью GitHub Actions

Несмотря на то что описанный в разделе 6.5.2 способ ручного развертывания позволяет осуществить полный процесс развертывания от локальной среды до облака, в реальной производственной среде непрерывное развертывание является более эффективным, безопасным и подходящим методом для командной работы. В частности, в проектах, предполагающих совместную работу нескольких человек, частое обновление кода или высокие требования к доступности онлайн-систем, выполнение операций вручную не только снижает эффективность, но и создает риск ошибок или утечки конфиденциальной информации.

С другой стороны, создание конвейера автоматического развертывания с помощью инструментов CI/CD, таких как GitHub Actions, позволяет значительно повысить степень автоматизации и надежность развертывания. Разработчику достаточно отправить изменения кода в репозиторий Git, и система автоматически выполнит всю цепочку операций: установку зависимостей, сборку, упаковку и развертывание в облаке.

Среди множества способов для реализации автоматического развертывания мы особенно рекомендуем использовать GitHub Actions в сочетании с Vercel CLI. Такой подход обеспечивает полный контроль над логикой развертывания, а также тесную интеграцию с репозиторием кода без привязки к дополнительным платформам, что обеспечивает более высокую адаптивность.

1. Загрузка кода в репозиторий GitHub

Для активации GitHub Actions необходимо разместить локальный проект на GitHub. Для этого выполните следующие действия.

(1) Зайдите на официальный сайт GitHub, зарегистрируйте аккаунт и создайте новый репозиторий, после чего скопируйте несколько команд git, отображаемых на странице успешного создания проекта, как показано на рис. 6-37.

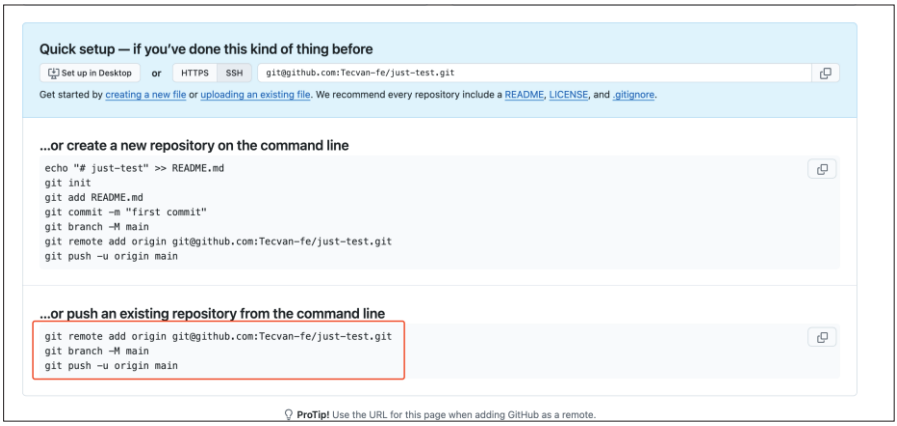


Рисунок 6-37. Подсказка с командами после успешного создания проекта

(2) Для отправки локального проекта на сервер GitHub запустите в терминале следующие команды:

```
git remote add origin https://github.com/your-username/your-repo-name.git
git add .
git commit -m "init"
git push -u origin main
```

После завершения отправки код будет передан в систему управления версиями и будет готов к запуску автоматического развертывания.

2. Создание и настройка токена Vercel

В процессе автоматического развертывания необходимо предоставить GitHub Actions учетные данные для получения доступа к API Vercel. Для этого нужно сгенерировать токен на платформе Vercel. Процедура выглядит следующим образом.

(1) Перейдите на страницу управления токенами Vercel по адресу https://vercel.com/account/settings/tokens.

(2) Нажмите **Create Token** (Создать токен), введите имя и подтвердите создание токена, как показано на рис. 6-38.

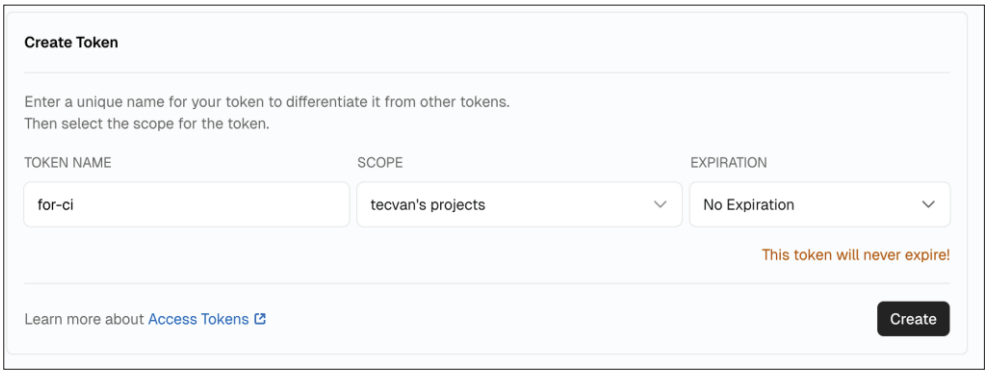


Рисунок 6-38. Создание токена

(3) На странице появится одноразовый токен (как показано на рис. 6-39). Обязательно скопируйте и сохраните его в локальном хранилище, так как эта информация больше не будет доступна для просмотра.

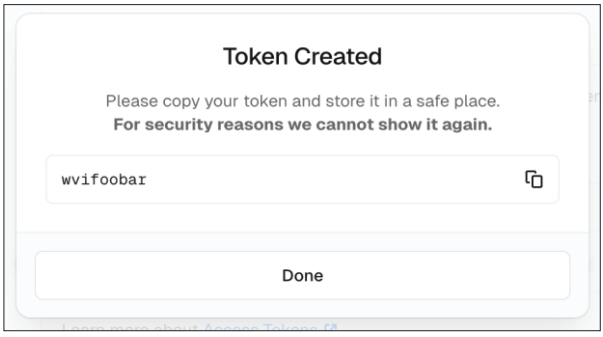


Рисунок 6-39. Копирование токена

(4) Откройте меню **Settings** (Установки) проекта GitHub и в подменю **Secrets and variables** (Конфиденциальные данные и переменные) выберите команду **Actions** (Действия), как показано на рис. 6-40.

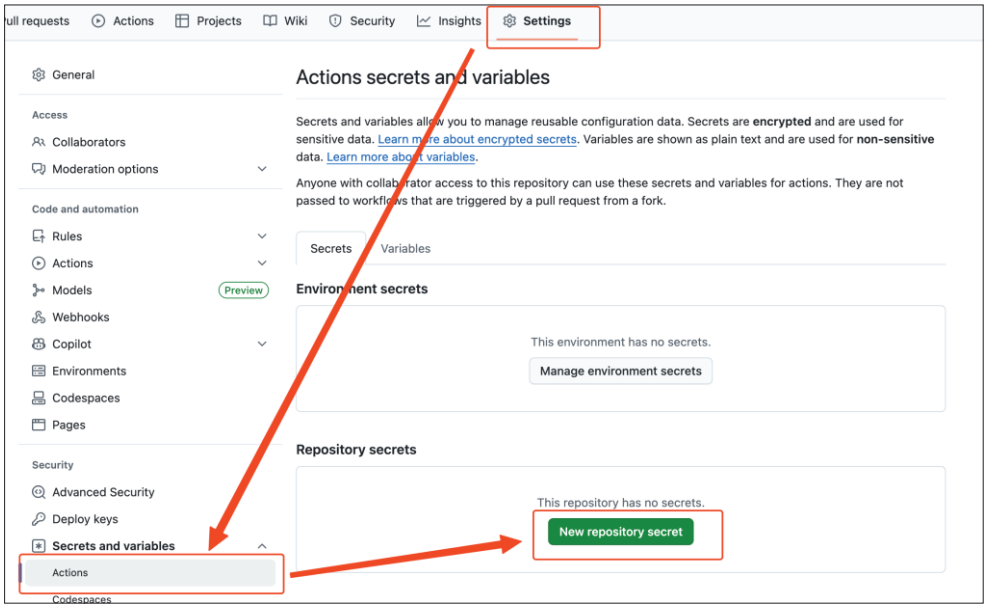


Рисунок 6-40. Добавление секретного ключа в GitHub Actions

(5) В появившемся диалоговом окне нажмите кнопку **New repository secret** (Новый секретный ключ репозитория).

(6) В появившемся диалоговом окне введите название секретного ключа `VERCEL_TOKEN`, вставьте только что сгенерированное значение в поле **Secret** (Секретный ключ) и нажмите **Add secret** (Добавить секретный ключ), как показано на рис. 6-41.

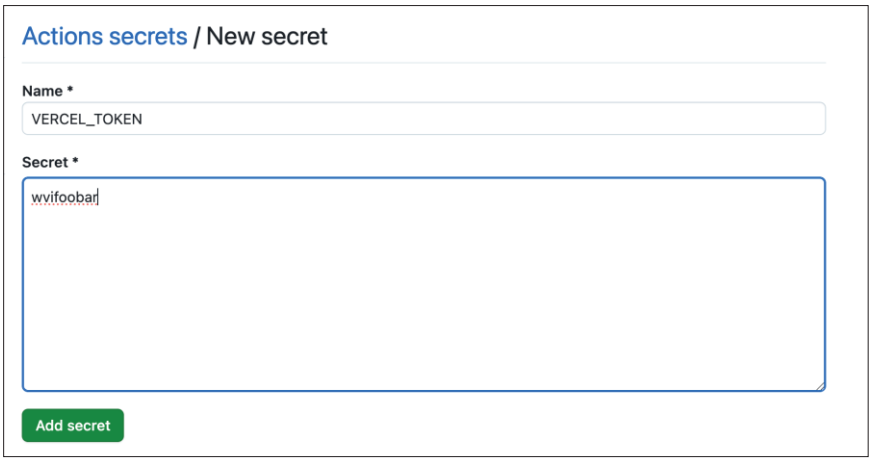


Рисунок 6-41. Настройка секретного ключа

После выполнения описанных выше действий репозиторий GitHub сможет безопасно использовать эти учетные данные для развертывания в конвейере.

3. Создание конвейера развертывания GitHub Actions

После подготовки конфигурации среды нам необходимо написать конвейер GitHub Actions, используя следующие подсказки, для реализации автоматического развертывания.

```
Для непрерывного развертывания необходимо добавить конвейер GitHub Actions,
основная логика которого заключается в следующем:
1. развертывание запускается только при изменении кода в ветке main;
2. установка зависимостей с помощью pnpm;
3. развертывание с помощью команд vercel link + pull + build + deploy,
с использованием VERCEL_TOKEN в качестве токена при развертывании.

Код необходимо сохранить в файле .github/workflows/deploy.yml.
```

После этого Cursor сгенерирует код конвейера развертывания, содержание которого будет примерно следующим.

```
name: Deploy to Vercel

on:
  push:
    branches: [main,]
  workflow_dispatch: # Разрешить ручной запуск

jobs:
  deploy:
    name: Deploy to Vercel
    runs-on: ubuntu-latest

    # Проверка названия репозитория
    if: github.repository == „Tecvan-fe/xiaohongshu-generator“

    steps:
      - name: Checkout code
        uses: actions/checkout@v4
        with:
          fetch-depth: 1

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: „22“

      - name: Setup pnpm
        uses: pnpm/action-setup@v2
        with:
          version: latest
          run_install: false
```

```
      - name: Install dependencies
        run: pnpm install --frozen-lockfile

      - name: Build project
        run: bash packages/server/build.sh
        env:
          NODE_ENV: production

      - name: Install Vercel CLI
        run: npm install --global vercel@latest

      - name: Run Deploy Script
        run: |
          vercel link
          --yes
          --project rednote-gen
          --token=${{ secrets.VERCEL_TOKEN }}

          vercel pull
          --yes
          --environment=production
          --token=${{ secrets.VERCEL_TOKEN }}

          vercel build
          --prod
          --token=${{ secrets.VERCEL_TOKEN }}

          vercel deploy
          --prebuilt
          --prod
          --archive=tgz
          --logs
          --token=${{ secrets.VERCEL_TOKEN }}

          echo "🚀 Развертывание успешно завершено!"

          echo "✅ Проект успешно развернут на Vercel"
```

У этой конвейерной линии есть следующие основные функции:

- автоматический запуск при обновлении ветки main;
- установка Node.js и среды зависимостей pnpm;
- сборка бэкенд- и фронтенд-продуктов (осуществляется на основе пользовательского скрипта сборки *packages/server/build.sh*, сгенерированного в разделе 6.5.1);
- выполняет развертывание с помощью Vercel CLI, что включает привязку проекта, извлечение конфигурации, сборку и запуск:
  - **vercel link**: привязка к конкретному проекту в личной учетной записи, при этом параметр `--project` необходимо заменить на имя проекта, созданного в Vercel;
  - **vercel pull**: извлечение конфигурации проекта Vercel;

- **vercel build**: запуск сборки Vercel с предварительной упаковкой контента для публикации на локальном компьютере;
- **vercel deploy**: упаковка результатов, полученных с помощью vercel build, и публикация в производственной среде.

Обратите внимание, что все связанные с Vercel команды требуют передачи аргументов, то есть значений, сгенерированных в системе Vercel в ходе описанных выше 4 шагов.

После этого изменения в ветке main запускают эту конвейерную линию, которая в режиме реального времени публикует код в среду Vercel. Благодаря этому конвейеру больше не нужно запускать несколько сложных команд локально, а главное, при командной работе не нужно передавать различные значения параметров или права доступа к проекту – любой член команды может в любой момент развернуть новую версию, что значительно облегчает совместную работу.

6.6. Заключение

В этой главе на примере проекта «Генератор контента для Xiaohongshu» мы рассмотрели полный цикл разработки полнофункционального ИИ-приложения: от настройки среды, анализа требований к проекту и распределения задач между фронтендом и бэкендом до развертывания приложения. Каждый шаг ориентирован на реальные задачи и сочетает в себе методы разработки на основе больших языковых моделей, что позволяет постепенно сформировать набор методов совместной работы, пригодных для повторного использования и переноса на другие проекты.

По итогам практических занятий в этой главе можно выделить несколько полезных прикладных рекомендаций.

- При написании документации для ИИ ключевое значение имеет выбор подсказок, определяющих путь поведения. ИИ не является потребителем документации, он сам выступает в роли разработчика. При составлении документации следует уделять первостепенное внимание структуре выполнения задач и не стремиться к эстетичному оформлению, а сосредоточиться на четкой структуре и конкретности формулировок.
- Для построения контекста состояния проекта следует использовать файл plan.md. Механизм `[ ]→[x]` в Markdown позволяет ИИ в ходе диалогового взаимодействия запоминать задачи и отслеживать ход выполнения проекта, что эффективно снижает риск «забывания» контекста.
- Проведение предварительного моделирования – это не компромисс, а средство повышения эффективности. Моделируемые данные позволяют изолировать факторы неопределенности в бэкенде и являются ключевым инструментом для быстрой проработки подробностей взаимодействия и подтверждения структуры на этапе веб-разработки, выступая в качестве защитного механизма при взаимодействии с ИИ.
- В первую очередь разрабатывайте бэкенд-интерфейсы и определяйте границы веб-компонентов с помощью данных. Выполнение предварительного абстрагирования интерфейсов и определения типов помогает сгенери-

ровать четкие цели для веб-компонентов, что способствует уменьшению ошибочных решений ИИ и понижению уровня связности кода.

- Отладка должна основываться на структурированных входных данных, а не на ваших «словесных описаниях». ИИ не может «видеть» браузер, поэтому ему необходимо предоставлять структурированную информацию о странице через скриншоты или MCP, чтобы он мог оценивать состояние рендеринга на веб-стороне, и только тогда отладка будет иметь смысл.
- При проверке придерживайтесь стратегии низкой стоимости и толерантности. На начальном этапе важна проработка логики, а элегантность кода и его оптимальная структура не являются первостепенными вопросами. Генерируемые ИИ результаты приемлемы, если они работают, поддаются отладке и позволяют двигаться дальше.
- Используйте Git для кеширования неопределенности ИИ. Каждое изменение ИИ может привести к изменению структуры, поэтому частые фиксации являются чрезвычайно эффективным «моментальным снимком версии» в программировании с ИИ, что позволяет быстро вернуться к предыдущему состоянию в случае ошибки.

В целом суть вайб-программирования заключается не в том, чтобы заставить ИИ писать код за вас, а в том, чтобы создать рабочий процесс и контекстную структуру, благоприятные для работы ИИ.

Глава 7

Ограничения и проблемы

Благодаря быстрому развитию различных продуктов для ИИ-программирования и постоянному появлению новых инструментов новости об этих технологиях активно обсуждаются в социальных сетях: кто-то представляет инновационный продукт, кто-то добавляет новые функции, а кто-то с помощью ИИ-инструментов создает приложения без программирования и успешно запускает коммерческие продукты. Теперь разработка приложений, судя по всему, больше не ограничивается кругом людей с профессиональным образованием в области программирования. Означает ли это, что у широкой публики постепенно появляются возможности и навыки для самостоятельной разработки продуктов?

Вайб-программирование – это будущее. Эта революция в парадигме программирования в корне меняет способы работы и мышления разработчиков и даже влияет на экосистему всей индустрии разработки ПО. Тем не менее, как и любая технологическая революция, которая несет с собой как возможности, так и вызовы, вайб-программирование не является панацеей. Любой здравомыслящий наблюдатель должен признать, что все существующие на сегодняшний день продукты для программирования с помощью ИИ по-прежнему обладают множеством ограничений. Нам не следует относиться к технологии со слепым оптимизмом только из-за ее «волшебства», но также не стоит впадать в уныние из-за существующих у нее проблем.

Безусловно, не только неподготовленные пользователи, но и многие программисты зачастую теряются в догадках: в чем разница между этими постоянно появляющимися продуктами для ИИ-программирования, на что они способны, а на что нет, и как их правильно использовать. Несомненно, все это требует определенного «мастерства» (то есть технических навыков), но если лишь наблюдать за происходящим со стороны, нам никогда не удастся по-настоящему понять значение этой революции. В данной главе мы проанализируем фактическое состояние дел с вайб-программированием с двух ракурсов: с точки зрения потребителя и с точки зрения разработчика. Мы честно обозначим его ограничения и объективно оценим его потенциал. Наша цель состоит в том, чтобы каждый читатель, независимо от опыта в программировании, смог более эффективно раскрыть свой творческий потенциал после ознакомления с вайб-программированием, дабы затем самостоятельно придумывать и создавать инструменты для решения реальных задач.

7.1. Точка зрения потребителя

Рассматривая ограничения и вызовы вайб-программирования, в первую очередь необходимо исходить именно с точки зрения потребителя. Люди с разным опытом сталкиваются с совершенно разными проблемами и вызовами при использовании инструментов ИИ-программирования.

7.1.1. Обычный потребитель

Для обычного пользователя (то есть человека без опыта программирования) истории успеха о «разработке приложений без кода», которые часто встречаются в социальных сетях, нередко оказываются вводящими в заблуждение. Эти истории обычно демонстрируют только конечный результат, упуская из виду сопутствующие затраты на обучение, приобретение опыта методом проб и ошибок, а также огромный объем работы по корректировке нюансов. На самом деле за этими успешными примерами часто скрывается множество скрытых от посторонних глаз экспериментов и неудавшихся попыток.

Реальность такова, что даже при использовании самых передовых и удобных инструментов ИИ-программирования людям без специальной подготовки по-прежнему необходимо овладеть базовыми концепциями информатики, пониманием архитектуры ПО, а также навыками разбиения задач на составляющие и отладки кода. Точно так же, как использование программ для автоматического перевода не делает человека профессиональным переводчиком, использование инструментов ИИ-программирования не делает человека профессиональным разработчиком ПО. Эти инструменты действительно снижают порог входа в профессию, но не устраняют необходимость обучения.

Что еще более важно, большинство успешных примеров разработки «без кода» сосредоточены на относительно простых сценариях, таких как создание базовых веб-сайтов, написание простых скриптов для обработки данных, создание демонстрационных прототипов или разработка приложений-инструментов. Когда речь заходит о сложной бизнес-логике, проектировании баз данных, оптимизации производительности, вопросах безопасности и т. п., одного использования инструментов ИИ-программирования явно недостаточно. Эти сложные сценарии чаще всего требуют глубоких технических знаний и системного мышления, и в данных областях инструменты ИИ-программирования пока не могут полностью заменить человека.

Многие обычные пользователи возлагают на инструменты ИИ-программирования завышенные ожидания, полагая, что достаточно описать требования на естественном языке, и ИИ автоматически сгенерирует идеальный программный продукт. Такие ожидания обычно возникают из-за недооценки сложности разработки ПО, ведь для описания требований необходимо четкое понимание архитектуры системы. Разработка ПО – это не только написание кода, но и анализ требований, проектирование системы, учет пользовательского опыта, оптимизация производительности, обеспечение безопасности, тестирование, развертывание и сопровождение, а также ряд других сложных задач. Даже если инструменты ИИ-программирования помогают генерировать код, для выполнения остальных задач по-прежнему требуются профессиональные знания и практический опыт.

Многие обычные пользователи, а тем более люди, не связанные с интернет-сферой, не осознают, что создание качественного программного продукта чаще всего требует многократных итераций и оптимизации. Первая версия минимально жизнеспособного продукта (Minimum Viable Product, MVP) может быть лишь началом, а по-настоящему ценный продукт требует постоянного совершенствования с учетом обратной связи от пользователей. В этом процессе такие навыки, как понимание логики кода, устранение ошибок и оптимизация производительности, по-прежнему остаются незаменимыми.

Даже если мы можем четко описать требования на естественном языке, понимание кода, сгенерированного ИИ-инструментами, остается серьезной проблемой. Ведь даже если не требуется писать код с нуля, понимание его базовой логики по-прежнему необходимо для отладки и модификации. Кроме того, генерируемый ИИ код не всегда идеален, поэтому нужно уметь находить ошибки и направлять ИИ-инструмент на их исправление.

Когда мы преодолеваем все проблемы и действительно разрабатываем веб-сайт или приложение, вопрос о его развертывании и запуске может стать «последней каплей, переполнившей чашу». Многие недооценивают сложность процесса развертывания. Для разных платформ, операционных систем и сред требуются разные стратегии развертывания. Даже если инструменты ИИ-программирования способны генерировать фронтенд- и бэкенд-код или базовую логику приложения, такие операции, как настройка сервера, управление доменами, конфигурирование баз данных и получение SSL-сертификатов, по-прежнему требуют обширных технических знаний.

В случае разработки приложения при его публикации необходимо пройти процесс проверки в магазине приложений, что обычно включает в себя разработку политики конфиденциальности, определение возрастного рейтинга и обеспечение соответствия контента установленным требованиям. Магазины Apple App Store и Google Play имеют строгие стандарты проверки, и эти проблемы не могут быть решены просто с помощью кода, сгенерированного ИИ.

Даже в случае успешного развертывания или выпуска последующая эксплуатация и обслуживание остаются не менее важными и сложными задачами. Мониторинг производительности приложения, обработка отзывов пользователей, исправление ошибок, обновление функционала, обеспечение безопасности данных и т. д. требуют постоянных затрат времени и усилий, и эти трудности зачастую не учитываются непрофессиональными пользователями на начальном этапе разработки проекта.

Генерируемый ИИ код может показаться безупречным во время демонстрации возможностей, но при взаимодействии с большим количеством реальных пользователей могут проявиться проблемы с производительностью, стабильностью, безопасностью и другими аспектами, а их решение требует профессиональных технических знаний.

Кроме того, необходимо учитывать конкурентную среду. Хотя ИИ-инструменты снижают порог входа в разработку, это также означает появление большего числа конкурентов на рынке. В таких условиях различия в техническом качестве продукта, удобстве использования и оптимизации производительности становятся ключевыми факторами успешности.

7.1.2. Профессиональные разработчики

Для профессиональных разработчиков инструменты ИИ-программирования являются одновременно помощником и проблемой. С одной стороны, они могут значительно повысить эффективность кодирования, автоматически генерировать код, помогать решать сложные задачи и снижать нагрузку от повторяющихся операций. С другой стороны, это требует от профессиональных разработчиков изменения подхода к работе, освоения навыков эффективного использования инструментов ИИ-программирования и налаживания эффективного взаимодействия с ними, не позволяя им заменить себя.

Профессиональные разработчики обычно более критично относятся к инструментам ИИ-программирования, поскольку их повседневная работа состоит во взаимодействии с различными специалистами (продукт-менеджерами, дизайнерами и т. д.) для выполнения разнообразных задач по разработке ПО. Обычно они предъявляют более высокие требования к программным продуктам, которые, как правило, являются более сложными.

Нельзя отрицать, что даже у самых передовых на сегодняшний день больших языковых моделей все еще существуют явные ограничения возможностей в области программирования. Существующие инструменты ИИ для программирования зачастую не справляются с решением сложных системных задач. Например, в случае крупного проекта со множеством микросервисов, проектированием баз данных и оптимизацией производительности ИИ-инструменты с трудом справляются с архитектурным проектированием и принятием решений на глобальном уровне. Они могут генерировать отличные фрагменты кода, но им не хватает глубокого понимания сложности всей системы. Это похоже на ситуацию, когда человек не видит леса за деревьями: он может быть мастером своего дела, но не сможет выполнять роль архитектора.

Сложность заключается в том, что инструменты ИИ-программирования не справляются с задачами в профессиональных сценариях, требующих глубоких знаний в конкретной области. Например, при разработке в таких специализированных областях, как решения для количественного трейдинга на финансовых рынках, ПО для управления медицинским оборудованием или системы управления в режиме реального времени для аэрокосмической отрасли, профессиональным разработчикам требуются не только навыки программирования, но и глубокое понимание контекста соответствующей предметной области. Инструменты ИИ-программирования могут уметь писать синтаксически правильный код, но они не способны понять сложность финансовых рынков и не могут нести ответственность за безопасность жизни людей при разработке медицинского ПО.

Для опытных профессиональных разработчиков созданный ИИ-инструментами код имеет одну общую проблему: красиво выглядит, но сложно использовать. Такой код может отлично работать на этапе демонстрации, но при долгосрочном обслуживании в нем обнаруживается множество проблем. Созданный ИИ код в разное время и в разных контекстах может использовать совершенно разные стили программирования и шаблоны проектирования. Это похоже на ситуацию, когда в одной команде работают несколько разработчиков с совершенно разными стилями, но между ними отсутствуют коммуникация и координация. Такая несогласованность может быть незаметна в небольших приложениях, но в крупных проектах она серьезно ухудшает читаемость

и удобство обслуживания кода. Инструменты ИИ-программирования зачастую склонны генерировать код, который выглядит весьма «профессионально», с использованием сложных шаблонов проектирования и структур, даже в простых сценариях. Это похоже на «стрельбу из пушки по воробьям»: показывая техническую мощь, такие решения добавляют ненужную сложность. Для проектов с долгосрочной поддержкой лаконичность и простота зачастую ценнее, чем сложность и «демонстрация мастерства».

Для профессиональных разработчиков корпоративных приложений использование инструментов ИИ-программирования также сопряжено с серьезными рисками в плане безопасности. Эти риски связаны как с техническими ограничениями самих инструментов, так и со специфическими требованиями корпоративной среды. Обучающие данные моделей ИИ содержат большое количество открытого кода, в котором неизбежно присутствуют уязвимости. Инструменты ИИ-программирования могут унаследовать эти уязвимости и даже создавать новые угрозы безопасности в новых контекстах. Опаснее всего то, что эти уязвимости могут быть трудноуловимыми и обнаруживаются только в ходе профессионального аудита безопасности.

Еще одной серьезной проблемой является конфиденциальность корпоративных данных. Когда профессиональные разработчики используют облачные сервисы ИИ-программирования, их код, комментарии и даже структура проекта могут быть загружены в облако. Для проектов, связанных с коммерческой тайной или персональными данными, такой риск утечки данных является неприемлемым. Хотя многие поставщики услуг обещают не хранить пользовательские данные, такие заверения не всегда могут полностью развеять опасения компаний.

Для профессиональных разработчиков современные инструменты ИИ-программирования по своей сути представляют собой средства распознавания и реорганизации шаблонов на основе существующего кода. Они отлично справляются с известными задачами, но могут оказаться беспомощными перед новыми техническими вызовами. Эти ограничения особенно заметны в передовых технологических областях или в нишевых сценариях применения. Когда профильным разработчикам необходимо исследовать совершенно новые алгоритмы, проектировать инновационные архитектуры, решать беспрецедентные технические задачи или разрабатывать нишевые продукты, инструменты ИИ-программирования зачастую могут предложить лишь рекомендации на основе существующих знаний. Они не способны генерировать подлинно оригинальные идеи. Это похоже на то, как эрудированный библиотекарь может быстро найти нужные материалы, но вряд ли способен проводить самостоятельные научные исследования.

Традиционное мышление программиста заключается в подходе «я знаю, как это сделать, и теперь я это реализую», тогда как для профессионального разработчика, работающего с ИИ-инструментами, этот подход выглядит так: «я знаю, что мне нужно, и теперь я направляю ИИ на его реализацию». Такой переход требует от профессиональных разработчиков более развитых навыков формулирования требований, проверки кода, а также умения взаимодействовать с ИИ. Для некоторых профессиональных разработчиков такая смена роли может оказаться более сложной, чем изучение нового языка программирования, поскольку не все разработчики обладают достаточными коммуникативными навыками.

7.2. Точка зрения разработчика

С точки зрения разработчиков и дизайнеров продуктов для ИИ-программирования, современные инструменты в этой области испытывают более сложные и глубокие проблемы. Речь не только о технической реализации, но и о таких аспектах, как позиционирование продукта, коммерческие затраты и удобство для пользователя.

7.2.1. Неудобное позиционирование продукта

Основной целевой аудиторией продуктов для ИИ-программирования являются профессиональные разработчики, в то время как полностью интегрированные продукты для программирования с помощью ИИ-агентов ориентированы в большей степени на обычных пользователей.

Для профессиональных разработчиков такие продукты для ИИ-программирования, как Cursor, GitHub Copilot и Claude Code, действительно помогают повысить эффективность кодирования, но в случае по-настоящему сложных или нишевых бизнес-сценариев они оказываются бессильны. Это скорее «умные помощники», чем «профессиональные партнеры по разработке». Когда перед профессиональным разработчиком возникает задача, требующая глубокого анализа и нестандартного решения, ему все равно приходится возвращаться к традиционным методам разработки. В результате возникает парадоксальная ситуация: чем опытнее профессиональный разработчик, тем лучше он может использовать эти инструменты, но в то же время тем очевиднее для него их ограничения.

Для обычных пользователей такие продукты для полного цикла программирования агентов, как v0 и YouWare, действительно сильно снижают порог входа, но при этом не устраняют сложности самих процессов написания кода. При маркетинговом продвижении на рынке для многих таких продуктов чрезмерно подчеркиваются такие характеристики, как «без кода» и «программирование для всех», однако при реальном использовании пользователи часто обнаруживают, что для создания действительно сложных и полезных приложений попрежнему требуются значительные усилия для изучения и отработки навыков. Например, генерируемый v0 код в принципе работает, но для его эффективной модификации и настройки пользователю зачастую нужно понимание технологического стека, включающего React, Tailwind CSS и т. д. Кроме того, подобные продукты для сквозного программирования агентов часто позволяют разрабатывать лишь легкие и простые веб-продукты, но с их помощью нелегко создать полноценное приложение для мобильных или настольных систем. Такой разрыв между ожиданиями и реальностью нередко приводит к разочарованию и оттоку потребителей. Обычным пользователям, желающим создавать более сложные продукты различного типа, все равно приходится прибегать к продуктам с ИИ-поддержкой, таким как Cursor, и много учиться.

В связи с быстрым развитием ИИ-технологий позиционирование продуктов также непрерывно меняется. Например, Cursor стремится развиваться из «вспомогательного инструмента» в «независимого агента», но такой переход чреват неудобством для пользователей и перегруженностью набора функций. Помимо освоения навыков эффективного взаимодействия с ИИ-агентом, пользовате-

лям также приходится осваивать традиционные навыки управления проектами и рецензирования кода. Такое сочетание требований к профессиональным навыкам на самом деле повышает, а не снижает порог вхождения. Сохранение четкого позиционирования продукта в быстро меняющейся технологической среде – это общая проблема для всех продуктов ИИ-программирования.

7.2.2. Проблема издержек

Продукты для ИИ-программирования сталкиваются еще с одной проблемой, которой нет у подавляющего большинства традиционных продуктов: затраты на большие языковые модели. В отличие от традиционных программных продуктов, каждое использование ИИ-инструментов влечет за собой существенные расходы на вычисления, которые зачастую в десятки, а то и сотни раз превышают затраты на традиционное ПО. Высокая стоимость использования больших языковых моделей затрудняет создание устойчивой бизнес-модели для таких продуктов.

Изменение структуры затрат приводит к дилемме при выборе ценовой стратегии. В GitHub Copilot используется модель с ежемесячной оплатой, однако активность пользователей значительно различается; Cursor предоставляет определенный бесплатный лимит, а затем взимает плату в зависимости от фактического использования, но такая модель вызывает у пользователей опасения по поводу превышения лимита. Если цена слишком высока, это ограничит частоту использования и снизит готовность пользователей оплачивать услугу; если цена слишком низкая, разработчики продукта могут оказаться в сложной ситуации, когда увеличение числа пользователей приводит лишь к росту убытков.

Из-за существенных различий в возможностях этих продуктов для ИИ-программирования пользователи зачастую комбинируют несколько продуктов, но это приводит к возникновению серьезной проблемы. Например, после оплаты услуги Claude пользователь может использовать принадлежащий ей сервис Claude Code, но при использовании Cursor, несмотря на продолжение работы с моделью Claude, ему все равно необходимо оплачивать услуги Claude, предоставляемые Cursor. Хотя Cursor действительно предоставляет отличный сервис на основе Claude, пользователь все равно частично дублирует расходы за модель Claude, и эта проблема особенно заметна при использовании множества разных продуктов.

Еще одна сопутствующая проблема связана с зависимостью от поставщиков моделей. Большинство продуктов для ИИ-программирования используют API-сервисы больших языковых моделей от таких провайдеров, как OpenAI и Anthropic. Это означает, что структура затрат данных продуктов в значительной степени зависит от ценовой политики этих провайдеров. Например, когда OpenAI корректирует цены на GPT-4, производители всех продуктов для ИИ-программирования на основе этой модели вынуждены пересматривать свою бизнес-модель. Если провайдеры повышают цены в значительной степени, эти продукты могут сразу же оказаться на грани выживания; если же провайдеры снижают цены, все предыдущие инвестиции в оптимизацию затрат могут мгновенно обесцениться. Такая неопределенность делает долгосрочное бизнес-планирование чрезвычайно сложной задачей. Более того, рыночная конкуренция может заставить провайдеров категорически отказаться от об-

служивания продуктов нижестоящего уровня: например, компания Anthropic отказалась предоставлять поддержку модели Claude для Windsurf, что стало косвенной причиной потери Windsurf значительного числа пользователей. Еще хуже то, что когда поставщики верхнего уровня решают выйти на рынок ИИ-программирования (например, Anthropic запустила Claude Code), производители приложений нижнего уровня могут оказаться в сложной ситуации, когда провайдеры становятся их конкурентами.

7.2.3. Различные сценарии взаимодействия с потребителем

Разработка отличного продукта для ИИ-программирования намного сложнее, чем может показаться. Традиционный дизайн программного интерфейса подразумевает относительно четкие модели взаимодействия и требования пользователей, тогда как инструменты ИИ-программирования требуют решения проблем, связанных с неопределенностью, разнообразием и сложностью ИИ.

В первую очередь речь идет о балансе способов взаимодействия. Взаимодействие на естественном языке, хотя и является интуитивным, зачастую отличается неточностью и зависит от формулировок пользователя. Традиционный графический интерфейс, хотя и обеспечивает высокую точность, может ограничивать возможности ИИ. Мы можем наблюдать различные подходы к решению этой задачи: v0 и Bolt используют диалоговый интерфейс в сочетании с предварительным просмотром в реальном времени, пытаясь дать пользователю возможность визуально отслеживать процесс преобразования естественного языка в код. Тогда как Cursor в большей степени интегрирован в традиционную IDE и использует сочетание горячих клавиш и боковой панели для обеспечения баланса между естественным взаимодействием и точным управлением. В свою очередь, Devin выбрал более сложный интерфейс управления задачами с целью дать пользователю возможность контролировать и направлять рабочий процесс ИИ. Найти баланс между этими различными подходами – задача, над которой работают все участники рынка. Ситуацию еще больше усложняют совершенно разные предпочтения и способности к адаптации у разных категорий потребителей с разным опытом, что практически исключает вероятность успеха какого-либо единого универсального решения.

Второй аспект – это разработка механизмов обработки ошибок и обратной связи. ИИ неизбежно допускает ошибки, и одна из ключевых задач при создании пользовательского интерфейса заключается в разработке способов их быстрого выявления, анализа и исправления. В GitHub Copilot используется относительно сдержанный подход, когда пользователь самостоятельно оценивает качество сгенерированного кода. В Claude Code сделана попытка предложить более подробное объяснение и описание процесса вывода, однако такая детализированная обратная связь порой только запутывает пользователя. Ошибки в традиционном ПО обычно связаны с конкретными причинами и имеют четкие решения, тогда как ошибки ИИ чаще всего характеризуются неопределенностью и неоднозначностью. Разработка системы, которая одновременно защищает пользователя от воздействия ошибок и не нарушает его рабочий процесс, требует достижения тонкого баланса между несколькими противоречивыми целями.

И наконец, вопрос управления кривой обучения. Качественный продукт для ИИ-программирования обязан адаптироваться к широкому кругу пользователей – от новичков до профессиональных разработчиков, поэтому он должен обладать высокой адаптивностью и широкими возможностями настройки. В YouWare снижение порога вхождения достигается за счет участия сообщества и использования шаблонов, однако такой подход может ограничивать возможности специалистов. Напротив, высокая степень персонализации Cursor, хотя и пользуется популярностью у профессионалов, требует слишком больших усилий для освоения новичками. В то же время чрезмерная сложность может сделать продукт труднодоступным – это типичный конфликт между удобством использования и функциональностью.

7.3. Революция мышления разработчиков и доступность технологий для всех

Несмотря на многочисленные ограничения и вызовы, с которыми сталкивается вайб-программирование, не следует упускать из виду глубокие изменения, которые несет с собой эта технологическая революция. Подобно тому, как промышленная революция изменила не только способы производства, но и переформировала всю социальную структуру, появление вайб-программирования приносит не просто изменение методов программирования, но также влечет за собой коренные изменения модели мышления, структуры навыков, системы образования и даже социального разделения труда. Эти преобразования несут с собой не только трудности и вызовы, но также создают беспрецедентные возможности и перспективы.

7.3.1. Смена мышления разработчиков старого поколения

Для разработчиков традиционной школы появление вайб-программирования стало настоящим потрясением. Проблема заключается даже не в изменении инструментария, но в пересмотре самой сути концепции «что такое программирование».

Традиционное мышление в программировании представляет собой процесс построения «снизу вверх»: разработчик начинает с базовых структур данных и алгоритмов, постепенно создавая сложные программные системы. Эта модель мышления предполагает точный контроль над низкоуровневыми деталями: разработчик должен глубоко понимать механизм действия каждой строки кода. В рамках этой парадигмы навыки программирования обычно проявляются в виде глубокого знания синтаксиса, точного понимания алгоритмов и аккуратного проектирования системной архитектуры. Ценность разработчика заключается в его способности преобразовывать абстрактные бизнес-требования в точные компьютерные команды, и этот процесс предполагает накопление обширных технических знаний и практического опыта.

Появление вайб-программирования полностью перевернуло такое традиционное представление. Программирование превратилось из задачи «как реализовать» в задачу «что реализовать». В рамках этой новой парадигмы разработчикам больше не приходится уделять внимание конкретным синтаксическим деталям и путям реализации, а достаточно сосредоточиться на точной

формулировке требований и оценке качества результата. Для многих разработчиков старой школы, обладающих глубокими техническими знаниями, такой переход стал одновременно и облегчением, и проблемой.

С одной стороны, это освобождает их от необходимости тратить огромное количество времени на рутинную работу по написанию кода, позволяя уделять больше внимания творческому и стратегическому мышлению. Разработчик с десятилетним опытом в Java, которому раньше, скорее всего, потребовалась бы неделя на реализацию сложного модуля обработки данных, теперь может за несколько часов сгенерировать базовый код с помощью ИИ-инструментов, а затем полностью сосредоточиться на оптимизации архитектуры и доработке бизнес-логики. Такое повышение эффективности позволяет разработчикам браться за более значимую работу, превращаясь из простых создателей кода в специалистов, принимающих технические решения.

Однако проблемы тоже очевидны: многие разработчики обнаруживают, что технические навыки, которыми они раньше гордились (такие как глубокое знание того или иного языка программирования или умелое использование конкретных фреймворков), теряют свою значимость на фоне ИИ. Психологический шок от обесценивания этих навыков огромен: это если бы опытный ремесленник внезапно столкнулся с индустриальной производственной линией. Разработчикам необходимо осваивать совершенно новые навыки: как эффективно общаться с ИИ, как оценивать качество генерируемого ИИ кода, как проявить уникальную ценность в рамках взаимодействия человека и машины.

Суть такого сдвига в мышлении заключается в переходе от роли «контролера» к роли «координатора». В традиционном программировании разработчик обладает полным контролем над кодом: каждый вызов функции и каждое определение переменной тщательно обдумываются. В эпоху вайб-программирования разработчик больше похож на менеджера проекта или дирижера, которому необходимо научиться направлять ИИ в качестве высококлассного помощника на выполнение конкретных задач разработки. Это требует от них более сильных коммуникативных навыков, более широкого кругозора, отточенного восприятия и оценки качества.

Залог адаптации к этим изменениям заключается в осознании сущностных изменений в программировании. Оно больше не является чисто технической деятельностью, а представляет собой комплексную работу, включающую анализ потребностей, разработку решений, контроль качества и командную работу. Те разработчики, которые способны быстро адаптироваться к этим изменениям, как правило, получают больше возможностей для развития в новой эпохе; в то время как те, кто упорно придерживается традиционных навыков, могут обнаружить, что их ценность постепенно снижается.

7.3.2. Ключевые навыки разработчика новой эпохи

В новую эпоху, определяемую вайб-программированием, происходит фундаментальная перестройка ключевых конкурентных преимуществ разработчиков. Традиционные навыки программирования, хотя и остаются важными, уже не являются решающим фактором; разработчикам новой эпохи необходимо развивать совершенно новый набор компетенций.

- Определение проблемы и анализ требований станут одними из ключевых навыков. В условиях, когда ИИ эффективно генерирует код, умение точно понимать и формулировать бизнес-требования становится важнее, чем умение реализовывать эти требования. Это требует от разработчика глубокого понимания возможностей и ограничений технической реализации, а также умения эффективно общаться с бизнес-командой, менеджерами по продукту и конечными пользователями, чтобы преобразовать расплывчатые бизнес-описания в четкие технические требования. Например, когда компания, занимающаяся электронной коммерцией, заявляет о потребности в «интеллектуальной системе рекомендаций», традиционный разработчик, вероятно, сразу начнет думать о выборе алгоритма, обработке данных и оптимизации производительности. В то же время разработчики нового поколения в первую очередь должны глубоко проанализировать истинные причины этого запроса: требуется ли повысить конверсию покупок или увеличить время пребывания пользователей на сайте? Разные цели приведут к совершенно разным техническим решениям. Только после точной интерпретации бизнес-задачи с помощью технического языка инструменты ИИ-программирования смогут сгенерировать действительно ценные решения.
- В новую эпоху навыки проектирования системной архитектуры станут еще более важными, однако акценты в этой области несколько изменятся. Традиционное проектирование архитектуры в большей степени сосредоточено на оптимизации структуры на техническом уровне, тогда как в условиях разработки с использованием ИИ программистам необходимо уделять больше внимания вопросам проектирования архитектуры, которая будет подходить для ИИ-генерации и одновременно обеспечивать удобство последующего обслуживания и расширения. Для этого им необходимо не только понимать традиционные шаблоны проектирования и принципы архитектуры, но также глубоко понимать механизмы работы и ограничения ИИ-инструментов. Хороший разработчик должен уметь определять, какие части кода можно поручить ИИ, а какие требуют тщательного проектирования вручную. Например, стандартные операции создания, удаления, запроса и изменения данных, а также обработка типичной бизнес-логики отлично подходят для ИИ-генерации, тогда как ключевые модули, связанные с основными алгоритмами, оптимизацией производительности и средствами безопасности, требуют кропотливого проектирования человеком. Такой подход к проектированию архитектуры с разделением задач между человеком и машиной становится ключевым навыком разработчиков новой эпохи.
- Навыки инженерии промптов быстро становятся обязательными для каждого разработчика. Это требует умения писать эффективные команды для нейросети и навыков эффективного взаимодействия с ИИ. Хороший инженер по промптам должен понимать особенности и ограничения различных моделей ИИ, а также уметь направлять нейросеть на создание качественного кода с помощью правильного контекста, примеров и ограничений. Более глубокие навыки инженерии промптов проявляются в понимании внутренних механизмов работы нейросети. Напри-

мер, при решении сложных задач Claude предпочитает сначала разбивать проблему на части, а затем постепенно решать их. В то же время GPT-4 в некоторых сценариях стремится сразу же предложить готовое решение. Понимание этих различий и соответствующая адаптация стратегии взаимодействия – это навык высокого уровня для инженера по промптам. Развитие данного навыка требует обширной практики и анализа полученного опыта, а также глубокого понимания принципов работы ИИ.

- В эпоху ИИ все большее значение приобретают навыки рецензирования кода и оценки его качества. Традиционное рецензирование кода в основном сосредоточено на технических показателях, таких как синтаксическая точность, логическая обоснованность и потенциальные проблемы с производительностью. В то же время при рецензировании ИИ-кода необходимо оценивать удобство его обслуживания, безопасность и согласованность с системой в целом. Еще большую сложность представляет то, что ИИ-код зачастую на первый взгляд выглядит идеальным, но может содержать скрытые логические ошибки или недочеты проектирования. Хороший рецензент кода должен уметь быстро распознавать типичные проблемы в ИИ-коде, такие как склонность к избыточной инженерии, отсутствие обработки граничных условий и несогласованные стратегии обработки ошибок. Для этого нужен богатый практический опыт и глубокое понимание особенностей ИИ-инструментов. Кроме того, рецензент должен уметь четко формулировать обнаруженные проблемы для ИИ, чтобы обеспечить эффективную итеративную оптимизацию.
- Все большее значение приобретают навыки работы с различными технологическими стеками. В традиционной модели разработки разработчики, как правило, специализируются на одном технологическом стеке, например на фронтенд-разработке, бэкенд-разработке или разработке мобильных приложений. Однако в среде ИИ-разработки границы между технологическими стеками постепенно стираются. Специалист по фронтенд-разработке с помощью инструментов ИИ-программирования может относительно легко выполнить разработку бэкенд-API, а специалист по веб-разработке может быстро освоить разработку мобильных приложений. Эти изменения требуют от программистов нового поколения более широкого технического кругозора и более мощных навыков адаптации и обучения. От них не требуется досконального знания деталей реализации каждой технологии, но необходимо понимание сценариев применения, преимуществ, ограничений и взаимосвязей между различными технологиями. Такая Т-образная структура компетенций (то есть глубокая специализация в одной области и базовые знания в нескольких смежных областях) становится основным требованием рынка.

7.3.3. Глубина в одной области и охват в других: расцвет междисциплинарных навыков

Одной из наиболее заметных особенностей эры вайб-программирования является беспрецедентное внимание к глубине знаний. В эпоху традиционного программирования ключевым фактором для успешной карьеры чаще всего

была глубокая специализация: администратор баз данных (Database Administrator, DBA), владеющий тонкостями оптимизации баз данных, или системный инженер, знакомый с сетевым программированием, обладали незаменимой ценностью в своих профессиональных областях. Однако в новых условиях, когда ИИ способен обрабатывать большую часть технических деталей внедрения, глубоких знаний в одной области уже недостаточно для сохранения профессиональной конкурентоспособности.

Причина этих изменений кроется в «демократизации» инструментов ИИ. Когда ChatGPT может помочь психологу написать скрипт для анализа данных, а Claude ассистирует биологу в создании инструмента для анализа генетических последовательностей, традиционные технические барьеры становятся гораздо ниже. В таких условиях универсальные специалисты с технической базой и профессиональными знаниями в конкретной области, как правило, способны создавать более значимую ценность.

Можно заметить, что в наше время понимание принципов важнее конкретных деталей реализации. Раньше от хорошего инженера по машинному обучению требовалось умение работать с такими инструментами, как NumPy, pandas и TensorFlow, а также владение конкретными методами реализации различных алгоритмов оптимизации. В нынешних условиях гораздо важнее понимание более фундаментальных концепций, таких как области применения различных алгоритмов, важность предварительной обработки данных и научные методы оценки моделей. Конкретную реализацию кода можно поручить ИИ, но такие задачи, как выбор алгоритма, настройка параметров и интерпретация результатов, по-прежнему требуют участия экспертов.

После снижения барьеров для технической реализации истинное конкурентное преимущество обычно исходит из глубокого понимания области применения. Разработчик, разбирающийся как в программировании, так и в финансах, может более точно понять требования к системам алгоритмического трейдинга и более эффективно взаимодействовать с ИИ для разработки решений в соответствии с финансовым законодательством. Точно так же разработчик с образованием в области психологии может предложить более эффективные технические решения при разработке удобных и понятных пользователям продуктов.

Важность таких междисциплинарных навыков проявляется на нескольких уровнях. Во-первых, это точность понимания потребностей: когда разработчики обладают глубоким знанием области применения, они могут более точно уловить реальные потребности пользователей и избежать отклонения технической реализации от бизнес-целей. Во-вторых, это инновационные решения: междисциплинарный багаж знаний обычно способствует появлению уникальных подходов к решению задач, которые трудно придумать чисто техническим специалистам. Наконец, это эффективность взаимодействия: разработчики с профессиональными знаниями в конкретной области эффективнее общаются с бизнес-экспертами, менеджерами по продукту и конечными пользователями, что минимизирует искажения информации и снижает риски непонимания.

7.3.4. Технологическое равенство для всех

Настоящая революционная значимость вайб-программирования заключается не в повышении скорости написания кода, а в предоставлении возможности

участвовать в создании ПО всем, кто не является профессиональным разработчиком. Это новый виток технологического равенства в эпоху ИИ.

При традиционной модели разработки ПО специалисты в конкретных областях (например, врачи, учителя и т. д.) для воплощения своих профессиональных идей в технический продукт обычно вынуждены проходить через сложную цепочку коммуникаций с участием таких участников, как руководитель проекта, менеджер по продукту, дизайнер и разработчик. Этот процесс малоэффективен и часто сопровождается неточностями в понимании требований. Вайб-программирование сокращает путь от идеи до продукта, позволяя специалистам участвовать в создании ПО без посредников.

Не только эксперты в своей области, но и каждый из нас обладает собственными уникальными идеями и замыслами, однако раньше большинство из таких задумок оставались лишь в виде мечтаний из-за сложностей с технической реализацией. Вайб-программирование делает возможным их воплощение. Будь то концепция нового мобильного приложения или уникальный проект визуализации данных, любой желающий может быстро проверить жизнеспособность своей идеи с помощью ИИ-инструментов и даже сразу разработать рабочий прототип продукта.

ИИ-программирование также снижает порог входа в предпринимательство, поскольку позволяет индивидуальным бизнесменам проверять коммерческие идеи и разрабатывать продукты до стадии MVP с минимальными затратами. Обычный человек со знанием рынка может на досуге разработать приложение для конкретных целей, а затем быстро вывести его на рынок для апробации. Такая возможность быстрого отбора идей путем проб и ошибок порождает совершенно новую экосистему индивидуального предпринимательства. Мы наблюдаем все большее распространение феномена «компании одного человека», когда отдельный предприниматель использует ИИ-инструменты для выполнения работы, которую традиционно выполняла целая команда технических специалистов. Например, владелец зоомагазина с обширными знаниями в области ухода за домашними животными разработал приложение для управления здоровьем питомцев; опытный фитнес-тренер создал инструмент для подготовки индивидуальных тренировочных планов. Эти продукты, как правило, отличаются высокой степенью специализации и направленности, позволяя максимально эффективно удовлетворять потребности конкретных групп пользователей.

Безусловно, не следует забывать, что такое технологическое равенство не означает исчезновения всех технических барьеров. Обычным пользователям, работающим с инструментами вайб-программирования, по-прежнему необходимо развивать базовые технические навыки, включая логическое мышление, умение разбивать задачи на составляющие и навыки оценки качества. Кроме того, им нужно научиться эффективно взаимодействовать с инструментами ИИ, формулировать свои потребности и оценивать полученные результаты.

На самом деле для подавляющего большинства обычных пользователей главная ценность ИИ-программирования заключается в быстрой реализации идей. Хотя у ИИ-программирования есть свои проблемы, это не мешает нам использовать его в качестве вспомогательного средства для воплощения спонтанных идей в реальные продукты – конечно, при условии, что мы действительно умеем этим пользоваться. В конечном итоге даже если разработка

получилась не очень удачной или недостаточно совершенной, главное – чтобы идея получила практическое воплощение и жизнеспособность этой идеи можно быстро проверить. После этого можно либо отказаться от этой идеи и перейти к следующей, либо продолжить итерацию с помощью ИИ, а можно обратиться к профессионалам для доработки продукта – все эти решения можно принять на основе полученных данных.

Равноправие в доступе к технологиям, обеспеченное вайб-программированием, является историческим прорывом: оно расширяет круг создателей ПО и позволяет вносить свой уникальный вклад людям с разным опытом и знаниями. Увеличение разнообразия принесет более богатые возможности для инноваций всей индустрии разработки ПО и обществу в целом.

7.3.5. Корректировка карьерного роста и образовательных траекторий

Перед лицом глубоких изменений, вызванных вайб-программированием, традиционные траектории карьерного роста разработчиков и система образования требуют соответствующей корректировки. Этот процесс заключается не в добавлении нескольких курсов об ИИ, а в переосмыслении концепций и методов подготовки кадров.

В плане карьерного роста традиционные пути развития разработчиков зачастую относительно однообразны: от младшего разработчика к старшему, а затем к техническому эксперту или техническому руководителю. Хотя некоторые разработчики и переходят с технических должностей на должности менеджеров по продукту и т. п., но если брать за основу все сообщество разработчиков, то число тех, кто выбирает такую карьерную траекторию, на самом деле невелико. В новых условиях разработки с помощью ИИ карьерный рост разработчиков станет более разнообразным. Часть разработчиков выберут углубленное изучение области ИИ-инженерии и станут специалистами по инженерии промптов, разработчиками агентов, архитекторами ИИ-приложений или даже евангелистами ИИ-приложений, поскольку сообщество разработчиков обладает преимуществами в сфере ИИ, которых нет у других отраслей. Другая часть разработчиков выберут развитие в бизнес-направлении и станут техническими менеджерами по продуктам, экспертами по решениям или техническими консультантами. Есть и те, кто выберет междисциплинарное развитие, объединив навыки программирования с другими профессиональными компетенциями для становления в статусе специалистов широкого профиля.

Главное в таком разностороннем развитии – найти свою уникальную нишу. В условиях, когда ИИ способен выполнять большую часть стандартизированных задач программирования, разработчики, умеющие сочетать технические навыки с опытом в конкретной отрасли, инновационным мышлением и коммуникативными способностями, как правило, получают лучшие возможности для карьерного роста. Например, разработчик с медицинским образованием может найти свою уникальную нишу в разработке медицинских приложений на базе ИИ; разработчик с дизайнерским мышлением может сыграть важную роль в оптимизации взаимодействия с пользователем с помощью ИИ.

В сфере образования классическое обучение разработке ПО обычно является поэтапным: освоив один язык или фреймворк, можно довольно долго использовать эти знания в работе. Однако в эпоху стремительного развития ИИ «период полураспада» знаний значительно сократился, и разработчикам необходимо выработать систему и привычку непрерывного обучения. Эта система должна включать несколько уровней: технологический уровень с фокусом на новых ИИ-инструментах и технологиях; прикладное обучение, направленное на применение технологий ИИ в различных бизнес-сценариях; обучение на уровне мышления, предполагающее обобщение опыта и уроков взаимодействия человека и машины для повышения эффективности совместной работы. И что наиболее важно, разработчикам необходимо научиться использовать инструменты ИИ для ускорения собственного процесса обучения.

В сфере образования преподавание информатики не может больше ограничиваться исключительно изучением алгоритмов и структур данных. Необходимо уделять больше внимания развитию навыков решения проблем, системного мышления и междисциплинарной интеграции. Некоторые образовательные учреждения уже начали включать в свои учебные программы такие темы, как инженерия промптов, взаимодействие человека и машины и этика ИИ, одновременно увеличивая количество междисциплинарных проектов и возможностей для практической работы. Более глубокие изменения отражаются в изменении методов обучения. Традиционное преподавание программирования акцентирует внимание на изучении языка с нуля и постепенном создании сложных приложений. В среде разработки с поддержкой ИИ студентам может потребоваться сначала научиться взаимодействовать с ИИ для выполнения сложных задач, а затем вникнуть в базовые технические принципы. Такой «нисходящий» подход к обучению требует от педагогов пересмотра структуры учебных программ и методов преподавания. При этом построение обучения на основе потребностей и проблем часто способствует повышению интереса и эффективности учебного процесса. Например, когда студент использует ИИ для выполнения задачи и, столкнувшись с определенной проблемой, изучает необходимые знания, он одновременно и приобретает знания, и решает проблему.

7.4. Заключение

Вайб-программирование является фундаментальной технологической революцией, которая перестраивает индустрию разработки ПО. В этой главе мы подробно проанализировали актуальные вызовы, стоящие перед ИИ-инструментами для программирования, как с точки зрения конечных пользователей, так и с точки зрения разработчиков. Для обычных потребителей между красивой идеей «разработки без кода» и сложной технической реальностью по-прежнему существует огромная пропасть. Для профессионалов инструменты ИИ-программирования, хотя и помогают повысить эффективность, все же недостаточны для решения сложных системных задач и передовых технологических проблем. С точки зрения разработки продуктов, дальнейшее развитие инструментов ИИ-программирования сдерживают такие глубокие проблемы, как ограничения технической архитектуры, трудности с позиционированием и неэффективная структура расходов.

Вайб-программирование – это не обновление инструментария, а глубокая трансформация способов мышления, навыков и социального разделения труда. Вайб-программирование приносит с собой новый уровень технологического равенства: оно наносит удар по тем, кто увлеченно «зацикливается» на технологиях, и одновременно открывает широкие возможности для тех, кто не является специалистом, но хочет войти в этот мир.

В условиях происходящих перемен мы не должны впадать в слепой оптимизм из-за удивительных возможностей технологий, но также не должны поддаваться пессимизму и разочарованию из-за существующих проблем. Истинная ценность вайб-программирования заключается в переосмыслении двух фундаментальных вопросов: «кто может программировать» и «как программировать». В этом процессе те люди и организации, которые смогут активно адаптироваться к изменениям, проявить уникальные возможности человека и эффективно сотрудничать с ИИ, найдут свое место в новой эпохе и смогут созидать еще более значимые ценности. Это и вызов, и шанс – главное, как его осознать, оценить и использовать.

Вполне может быть, что истинное технологическое равенство заключается не в том, чтобы дать всем возможность «писать код», а в том, чтобы вернуть код к его сущности: он никогда не был целью, а лишь инструментом для создания новой ценности. После устранения синтаксических барьеров с помощью ИИ мы, напротив, можем еще четче увидеть, что размышления о системном проектировании, понимание потребностей пользователей и владение бизнес-логикой, возможно, и являются пропуском в будущее цифровой эпохи ИИ.

Послесловие

Спасибо за то, что вы дочитали до конца и вместе с нами погрузились в исследование этой революции, которую принесло вайб-программирование и которая сегодня полностью меняет мир разработки ПО. Будучи новой парадигмой кодирования, вайб-программирование представляет собой не просто технологическое обновление, но и глубокий сдвиг в мышлении разработчиков, а также в их рабочих методах. Оно позволяет нам заново построить отношения между человеком и кодом, а также между человеком и ИИ, открывая безграничные возможности для разработки ПО будущего.

Изначально мы задумывали эту книгу с целью раскрыть суть происходящих изменений и помочь вам лучше осознать, приспособиться и управлять данной революцией. Независимо от того, являетесь ли вы техническим разработчиком, менеджером по продукту или просто исследователем, увлеченным программированием ИИ, мы надеемся, что эта книга станет для вас надежным помощником в обучении и на практике.

Будущее полно неопределенности, но именно поэтому его исследование становится еще более значимым. Желаем вам найти свой ритм и вдохновение в мире вайб-программирования, постоянно внедрять инновации и радоваться изменениям. С нетерпением ждем будущего, чтобы вместе стать свидетелями появления новых возможностей.

Предметный указатель

А

Автоматическая сборка мусора (GC) 43
Автоматическое тестирование 154
Агент 72
Агентный ИИ 16
Алан Мэтисон Тьюринг 55
Анализ требований проекта 171
Андрей Карпатый 11, 16
Архитектура фон Неймана 56
Ассемблер 32

Б

Белый список разрешенных команд 164
Бессерверные вычисления 209
Большая языковая модель 15, 25, 63, 65, 74, 84, 93, 103, 119, 147, 164, 168, 176, 178, 198, 205

В

Вайб-программирование 11, 16, 24, 25, 61, 84, 93, 108, 119, 149, 171, 183, 223
Вайб-программирование, навыки 233
Вайб-программирование, определение 13, 15
Вайб-программирование, основы 25
Вайб-программирование, разработка промпта 93
Вайб-программирование, сценарий 79, 80
Вайб-программирование, экосистема приложений 63
Виртуальная машина Java (JVM) 43
Виртуальная машина (VM) 209
Вычислительная единица агента (ACU) 72

Г

Генеративный ИИ 18, 64, 85, 91, 162
Генеративный ИИ, ограничения 69
Графический интерфейс пользователя (GUI) 57

Д

Дисковая операционная система (DOS) 50
Документация по техническому проектированию 176
Документ с требованиями к продукту 17
Долгая краткосрочная память (LSTM) 89
Дополненная реальность (AR) 81

З

Загрузка кода в репозиторий GitHub 217
Запрос на изменение (PR) 29
Защищенный режим 36

И

Императивное программирование 21
Импорт настроек VS Code 166
Индексирование кодовой базы 164
Инкапсуляция 40
Интеграция ИИ в плагины IDE 66
Интегрированная среда разработки (IDE) 20, 51, 66, 162
Интенциональное (преднамеренное) программирование 22
Использование службы MCP браузера 198

К

Карьерный рост и образовательная траектория 238
Ключевой навык разработчика новой эпохи 233
Компиляция just-in-time (JIT) 43
Компоненты основного технологического стека 191
Конвейер непрерывной интеграции / непрерывной доставки (CI/CD) 21, 155
Контейнер 209
Концепция пользовательского ПО (UGS) 73

М
Мартин Фаулер 126
Машина Тьюринга 55
Машинный язык 31
Механизм Confidence Rating 72
Механизм внимания 94
Механизм извлечения–преобразования–загрузки (ETL) 59
Механизм непрерывной синхронизации контекста 74
Минимально жизнеспособный продукт (MVP) 226
Моделирование отношения is-a 41
Модель объектов документа (DOM) 18

Н
Наследование 41
Настройка подсказок для Cursor 100
Непрерывное развертывание с помощью GitHub Actions 217

О
Обработка естественного языка 15
Обработка ошибок 136
Объектная модель документа (DOM) 199
Объектно-ориентированное программирование 40
Ограничение ИИ 120
Ограничения диалогового режима программирования 66
Окно контекста 179
Оптимизация промпта 98

П
Паттерн модель–представление–контроллер (MVC) 42
Перемещение кода с неизменными циклами 37
План реализации проекта 179
Платформа с низким уровнем кодирования 59
Платформа с низким уровнем кодирования (low-code) 26, 27, 55, 56, 58, 59
Полиморфизм 41, 45
Предварительная проектная документация 171
Принцип единственной ответственности (SRP) 195
Принцип «что видишь, то и получаешь» (WYSIWYG) 51
Проблема ромбовидного наследования 43
Программирование без кодирования (no-code) 27, 55
Программирование с помощью IDE 66
Программная инженерия 150
Производительность кода 140
Протокол SSL 24
Протокол контекста модели (MCP) 199
Протокол контекста модели (Model Context Protocol, MCP) 67
Протокол языкового сервера (LSP) 53
Псевдокоманда 34
Псевдокоманда условий 33

Р
Развертывание приложения на Vercel 211
Развертывание стека приложения онлайн 207
Различие между традиционным и вайб-процессом разработки 25
Различия между планированием требований при традиционном и вайб-программировании 111
Разработка веб-страницы 193
Регистрация учетной записи ChatGPT 168

С
Серверная прорисовка (SSR) 207
Сеть доставки контента (CDN) 208
Система контроля версий 151
Система контроля ревизий 151
Система планирования ресурсов предприятия (ERP) 59
Система полноты по Тьюрингу 55
Система рецензирования кода 143
Система управления взаимоотношениями с клиентами (CRM) 59
Система управления исходным кодом 151
Система управления цепочкой поставок (SCM) 59
Сквозное программирование с помощью агентов 70
Создание и настройка токена Vercel 218
Создание конвейера развертывания GitHub Actions 220

Составление документа
с требованиями к продукту 111
Спагетти-код 120
Сравнение ИИ-интеграции в IDE 69
Сравнение методов разработки 23
Средний доход от платного
пользователя (ARPPU) 86
Стандартная библиотека шаблонов
(STL) 46
Стратегия ведения логов 135
Структура проекта 169
Структурированная обработка
исключений 44
Структурное программирование 42
Сценарий взаимодействия
с потребителем 231

Т

Текучесть кода 123
Тестирование интерфейса 188
Тестируемое программирование
(TDD) 178, 189
Технический долг
(technical debt) 114, 122
Технологическое равенство
для всех 236
Токен API OpenAI 168
Токен (token) 94
Традиционная модель разработки ПО
17, 23, 25, 81, 110, 189, 233, 235, 237

Ч

Чжао Чуньсян 86

Э

Эволюция взаимодействия
при программировании 49
Экосистема разработки Open Source 90

Я

Язык программирования 30, 36, 38
Язык четвертого поколения (4GL) 56

A

A11y (Accessibility) 106
Amazon CodeGuru 147
Android SDK 44
aPaaS 59
Augment Code 68
Autocoder 33
AutoML-Zero 91

B

Bolt 72
BrowserTools 199

C

C++ 45
Cal AI 89
ChatGPT 15, 168
Claude 65, 168
Claude Code 75
Cline 67
COBOL 38
CodeRabbit 147
Core ML 54
Cray-1 37
Cursor 68, 147, 161
Cursor, установка и настройка 162

D

DeepSeek 65
Devin 71
Docstring 132
Doubao 16, 168
Dreamweaver 57

E

Eclipse 52
EDLIN 50
ENIAC 31
Enterprise JavaBean (EJB) 44

F

Fortran 36

G

Gemini 65
Gemini CLI 76
Ghostwriter AI 79
Git 152
GitHub Actions 217
GitHub Copilot 67
Git, установка и настройка 167
Google Java Style Guide 128

I

IDE с нативной
интеграцией ИИ 68
Intel 80386 36
IntelliJ IDEA 54
iPaaS 59
IPython 48

J
Java 43
Java EE (Jakarta EE) 44
JavaScript 44
Java Server Pages (JSP) 44
JianDao Cloud 59
Jupyter Notebook 48

L
Lambda-выражение 44
LISP 56
Lovable 72

M
MACRO-11 33
Matplotlib/Seaborn 48
Midjourney 86
MingDao Cloud 59
Monorepo 170
MS-DOS 1.0 35

N
NASM 34
Neon cyberpunk 100
NetBeans 52
Node.js 53, 161
Node.js, установка и настройка 166
Nomad List 88
NumPy 48

O
OutSystems 58

P
pandas 48
Pascal 39
PDP-11 33
PEP 8 128
Photo AI 88
Plug AI 85
pnpm 161
pnpm, установка и настройка 167
PyCharm 54
Python 48
PyTorch 26

R
React 191
README 133
Remote OK 88
Replit 72, 89

S
shadcn/ui 191
Smalltalk-80 30, 40
Stream API 44

T
Tabnine 66
Tailwind CSS 191
TensorFlow 26, 91
Tian Gong 90
Tongyi 16
Tongyi Qianwen 65
Turbo Pascal 39, 52
TypeScript 191, 203

V
v0 72
Vercel AI SDK 184
Vercel CLI, установка 212
Visual Basic 57
Vite 191
Vitest 154
VS Code 53, 67, 166

W
Wenxin Yiyan 65
Windsurf 69
WordStar 50

X
Xcode 54

Y
YouWare 73
Yuanbao 16

Z
Zustand 191

Книги издательства «ДМК Пресс» можно купить оптом и в розницу
на складе издательства по адресу:
Москва, ул. Электродная, д. 2, стр. 12, офис 7,
тел. +7 (499) 322–19–38,
а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ

Ван Вэньцзе (Tecvan), Ли Цзяньчао (Isboyjc),
Ю И (Янь Сяофань), Чжан Бинь (Captain)

Вайб-программирование. Разработка кода в эпоху искусственного интеллекта

Главный редактор *Яценков В. С.*
editor@dmkpress.com

Перевод *Бахура В. И.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 19,99. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

«Основная идея книги заключается в том, что интеграция ИИ – это не усовершенствование существующих процессов, а принципиально новый способ использования моделей, по значимости не уступающий функциональному программированию. Сейчас зарождается новая парадигма разработки с использованием ИИ, которая полностью изменяет подход к созданию, развертыванию и обслуживанию кода программ».

Ван Шу @i5ting, эксперт по фронтенд-технологиям

В книге подробно рассматриваются технологическая эволюция и прикладные аспекты использования новейшего подхода к разработке ПО – так называемого вайб-программирования (Vibe Coding), также известного как «интуитивное программирование».

Основные темы книги:

- анализ эволюции подходов к программированию;
- три основных направления вайб-программирования;
- приемы составления промптов;
- приемы рецензирования и оптимизации кода;
- пример полного цикла разработки приложения;
- оценка современных технологических ограничений;
- новые возможности для непрофессиональных пользователей.

Издание адресовано как профессиональным разработчикам для совершенствования навыков построения архитектуры приложений, так и менеджерам по продуктам, UI/UX-дизайнерам и широкой аудитории как руководство по использованию ИИ для воплощения личных идей в реальные приложения.

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-93700-440-6
9 785937 004406 >