Claude Code

Полное руководство по AI-ассистенту для разработчиков



Ранас Мукминов

Claude Code: Полное руководство по AI-ассистенту для разработчиков

«Автор»
2026

Мукминов Р.

Claude Code: Полное руководство по AI-ассистенту для разработчиков / Р. Мукминов — «Автор», 2026

включает описание книги и перечень ключевых тем: написание кода, рефакторинг, отладка, тесты, MCP, DevOps, безопасность, реальные проекты.

© Мукминов Р., 2026
© Автор, 2026

Содержание

Предисловие 5
Часть I. Введение и основы 7
Глава 1. Что такое Claude Code и зачем он нужен 7
Глава 2. Установка и настройка окружения 18
Глава 3. Первые шаги с Claude Code 33
Глава 4. Архитектура и принципы работы 50
Глава 5. Эффективное общение с AI-ассистентом 65
Часть II. Практическое использование 80
Глава 6. Написание кода с нуля 80
Глава 7. Рефакторинг и улучшение существующего кода 93
Глава 8. Отладка и исправление ошибок 105
Глава 9. Написание тестов 116
Глава 10. Работа с базами данных 129
Глава 11. Документирование кода и проекта 138
Глава 12. Git и совместная разработка 147
Часть III. Продвинутые возможности 155
Глава 13. MCP: Model Context Protocol 155
Глава 14. Автоматизация рабочих процессов 164
Глава 15. DevOps и инфраструктура 173
Глава 16. Безопасность 180
Глава 17. Мультиязычная разработка 187
Глава 18. Предметные области: от веба до IoT 194
Часть IV. Реальные проекты 199
Глава 19. Проект: SaaS-приложение с нуля 199
Глава 20. Проект: REST API для маркетплейса 204
Глава 21. Проект: Telegram-бот на Python 209
Глава 22. Проект: CLI-утилита на Go 213
Глава 23. Проект: Chrome-расширение 218
Глава 24. Проект: Миграция легаси-кода 222
Часть V. Лучшие практики и будущее 227
Глава 25. Лучшие практики и паттерны 227
Глава 26. Ограничения и подводные камни 232
Глава 27. Будущее AI-разработки 237
Приложения 241
Приложение А. Шпаргалка по командам Claude Code 241
Приложение Б. Шаблоны CLAUDE.md 243
Приложение В. Словарь терминов 247

Ранас Мукминов
Claude Code: Полное руководство по AI-ассистенту для разработчиков

Предисловие

О чём эта книга

Уважаемый читатель, перед вами – наиболее полное русскоязычное руководство по Claude Code, революционному AI-ассистенту для разработчиков, созданному компанией Anthropic. Эта книга родилась из простого наблюдения: мир программирования переживает фундаментальный сдвиг, сопоставимый по масштабу с появлением объектно-ориентированного программирования, открытого программного обеспечения или облачных вычислений. Искусственный интеллект больше не является экзотической технологией из научно-фантастических фильмов – он превратился в повседневный инструмент, способный кардинально изменить то, как мы пишем, отлаживаем и поддерживаем программный код.

Claude Code занимает особое место среди AI-инструментов для разработки. В отличие от простых автодополнителей кода, Claude Code представляет собой полноценного агентного ассистента, способного понимать контекст вашего проекта, работать с файловой системой, запускать команды в терминале, анализировать ошибки и самостоятельно планировать сложные многоэтапные задачи. Это не просто «умная подсказка» – это настоящий напарник по программированию, который может взять на себя существенную часть рутинной работы.

Для кого эта книга

Книга адресована широкому кругу разработчиков:

– **Начинающим программистам**, которые хотят ускорить свой рост, используя AI как наставника и помощника. Claude Code может объяснить сложные концепции, предложить лучшие практики и помочь быстро прототипировать идеи.

– **Опытным разработчикам**, стремящимся повысить свою продуктивность. Если вы тратите часы на рутинный рефакторинг, написание шаблонного кода или отладку типичных ошибок, Claude Code поможет автоматизировать эти задачи.

– **Тимлидам и архитекторам**, заинтересованным во внедрении AI-инструментов в рабочие процессы команды. Книга содержит практические рекомендации по настройке, стандартизации и масштабированию использования Claude Code.

– **Менеджерам и принимающим решения**, которым важно понимать возможности и ограничения AI-ассистентов для оценки ROI и планирования трансформации процессов разработки.

Как читать эту книгу

Книга организована в пять логических частей, которые можно читать последовательно или использовать как справочник:

Часть I закладывает фундамент: от установки и первых шагов до глубокого понимания архитектуры Claude Code и искусства эффективного общения с AI.

Часть II погружает в практику: написание кода, рефакторинг, отладка, тестирование, работа с базами данных, DevOps и Git – все основные аспекты ежедневной работы разработчика.

Часть III раскрывает продвинутые техники: MCP (Model Context Protocol), автоматизацию, безопасность, мультиязычную разработку, архитектурное проектирование и работу с API.

Часть IV представляет шесть полноценных проектов, которые вы можете воспроизвести от начала до конца: SaaS-платформа, мобильное приложение, CLI-утилита, Telegram-бот, Data Pipeline и микросервисная система.

Часть V выходит за рамки технологии: командная работа, сравнение инструментов, этические вопросы, оптимизация продуктивности и взгляд в будущее.

Каждая глава содержит практические примеры, фрагменты кода, советы и предупреждения. Я рекомендую держать под рукой терминал с установленным Claude Code и пробовать всё на практике.

Благодарности

Эта книга не появилась бы без открытости и прозрачности команды Anthropic, создавшей Claude и Claude Code. Отдельная благодарность – сообществу разработчиков, которые активно делятся опытом использования AI-ассистентов и формируют лучшие практики.

Надеюсь, после прочтения этой книги вы не просто научитесь использовать Claude Code, но и переосмыслите свой подход к разработке программного обеспечения. Добро пожаловать в эру AI-ассистированного программирования!

Автор, февраль 2026 года

Часть I. Введение и основы

Глава 1. Что такое Claude Code и зачем он нужен

1.1. Краткая история Anthropic и линейки Claude

Чтобы понять место Claude Code в современном ландшафте разработки, необходимо начать с истории компании, которая за ним стоит. Anthropic была основана в 2021 году группой исследователей, ранее работавших в OpenAI. Среди основателей – Дарио Амодей, занимавший пост вице-президента по исследованиям в OpenAI, и его сестра Даниэла Амодей, отвечавшая за операционное управление. Вместе с ними компанию покинули несколько ключевых исследователей, объединённых общей идеей: создать AI-системы, которые будут не только мощными, но и безопасными, предсказуемыми и управляемыми.

С момента основания Anthropic выбрала иной путь развития, нежели большинство конкурентов. Вместо гонки за размером моделей и бенчмарками компания сфокусировалась на том, что её основатели называют «конституционным AI» (Constitutional AI, или CAI). Суть подхода заключается в том, что модель обучается следовать набору принципов, сформулированных на естественном языке, – своего рода «конституции», определяющей границы допустимого поведения. Это позволяет создавать системы, которые не просто генерируют текст, но делают это ответственно, с пониманием контекста и последствий.

Первая публичная модель Anthropic – Claude – появилась в марте 2023 года. Она сразу привлекла внимание сообщества качеством генерации текста, способностью к анализу и рассуждению, а также заметно более аккуратным обращением с потенциально вредоносными запросами. В июле 2023 года вышла Claude 2, продемонстрировавшая значительный скачок в качестве кода, математических способностях и работе с длинными документами – контекстное окно расширилось до ста тысяч токенов.

Модель Claude 3 стала переломным моментом. Выпущенная в марте 2024 года в трёх вариантах – Haiku, Sonnet и Opus, – она показала результаты, сопоставимые и зачастую превосходящие GPT-4 от OpenAI на множестве бенчмарков. Claude 3 Opus заняла лидирующие позиции в задачах, требующих глубокого рассуждения и анализа, а более компактный Sonnet нашёл оптимальный баланс между скоростью, стоимостью и качеством.

Линейка продолжила развитие: Claude 3.5 Sonnet, выпущенная летом 2024 года, установила новые стандарты в программировании, превзойдя более дорогие модели-конкуренты на задачах генерации и анализа кода. Именно с этой модели началась эпоха Claude как серьёзного инструмента для профессиональной разработки. Но настоящая революция произошла с выпуском Claude Code.

Claude Code – это не просто API для генерации текста и кода. Это агентный инструмент командной строки, способный работать непосредственно в вашей среде разработки. Он читает файлы, пишет код, запускает тесты, анализирует ошибки, работает с системой контроля вер-

сий, управляет зависимостями и взаимодействует с внешними сервисами. Claude Code превращает мощную языковую модель в полноценного участника процесса разработки, который «живёт» в вашем терминале и имеет доступ к вашему проекту.

1.2. Место Claude Code среди AI-инструментов для разработки

Рынок AI-инструментов для разработки к 2026 году стал чрезвычайно насыщенным. Чтобы понять уникальную позицию Claude Code, необходимо рассмотреть основные категории существующих решений.

Автодополнители кода – первая и наиболее массовая категория. GitHub Copilot, появившийся в 2021 году, стал пионером этого направления. Принцип прост: вы пишете код, а AI предлагает продолжение – от одной строки до целого блока. К этой же категории относятся Codeium (бесплатная альтернатива), TabNine (использующий как облачные, так и локальные модели) и Amazon CodeWhisperer. Автодополнители интегрируются непосредственно в редактор и работают на уровне «следующей строки». Они хороши для ускорения набора типового кода, но их возможности ограничены: они не понимают архитектуру проекта в целом, не могут выполнить сложную многошаговую задачу и не имеют доступа к файловой системе или терминалу.

AI-ориентированные IDE – вторая категория, представленная прежде всего Cursor. Cursor взял код Visual Studio Code и перестроил его вокруг AI: модель имеет доступ ко всей кодовой базе, может предлагать изменения в нескольких файлах одновременно, понимает контекст проекта и поддерживает диалог в чате. Windsurf (ранее Codeium IDE) и Void предлагают аналогичный подход. Эти инструменты мощнее простых автодополнителей, но привязаны к конкретному редактору и его парадигме взаимодействия.

Чат-боты для кода – третья категория. ChatGPT, Gemini, Perplexity и другие универсальные AI-системы могут генерировать код, объяснять концепции и помогать с отладкой. Однако они работают в изоляции от вашего проекта: вы вручную копируете код в чат и обратно, AI не видит вашу файловую систему и не может запустить тест или проверить результат своей работы.

Агентные инструменты – четвёртая и новейшая категория, к которой относится Claude Code. Отличие агентных инструментов от всех предыдущих категорий фундаментально: они не просто генерируют код, а _действуют_. Агентный AI-ассистент может:

– Самостоятельно исследовать кодовую базу, открывая и анализируя нужные файлы
– Планировать последовательность шагов для решения сложной задачи
– Создавать и модифицировать файлы непосредственно на диске
– Запускать команды в терминале и анализировать их вывод
– Работать с Git: создавать ветки, коммитить изменения, разрешать конфликты
– Запускать тесты и исправлять ошибки на основе результатов
– Взаимодействовать с внешними сервисами через MCP (Model Context Protocol)
– Открывать браузер для проверки веб-приложений

Claude Code работает из командной строки, что делает его универсальным: ему не нужен конкретный редактор, он функционирует на любой системе с Node.js и терминалом. Вы можете

использовать его с Vim, Emacs, VS Code, JetBrains или любым другим редактором. Вы даже можете использовать его на удалённом сервере через SSH.

Принципиальное отличие Claude Code от конкурентов можно проиллюстрировать простым примером. Допустим, вам нужно добавить аутентификацию в веб-приложение. Автодополнитель поможет сгенерировать отдельные функции. Чат-бот покажет примерный код, который вы скопируете вручную. AI-ориентированный IDE может внести изменения в несколько файлов. Но Claude Code проанализирует вашу существующую архитектуру, определит нужные файлы, создаст модели, маршруты, middleware, напишет тесты, обновит зависимости, запустит тесты для проверки, исправит обнаруженные ошибки и предложит закоммитить результат – всё это в рамках одного диалога, без переключения контекста.

1.3. Подробное сравнение с основными конкурентами

Давайте рассмотрим каждого конкурента более детально, чтобы понять сильные и слабые стороны Claude Code в контексте каждого из них.

GitHub Copilot

GitHub Copilot – безусловно, самый массовый AI-инструмент для разработки. По состоянию на начало 2026 года он насчитывает миллионы активных пользователей. Copilot интегрируется в популярные IDE и предлагает автодополнение кода на основе моделей GPT от OpenAI. С выходом Copilot Chat и Copilot Workspace Microsoft расширила функциональность, добавив диалоговый интерфейс и возможность работы с несколькими файлами.

Преимущества Copilot:

– Глубокая интеграция с экосистемой GitHub
– Огромная база пользователей и отработанный продукт
– Поддержка множества языков и редакторов
– Относительно низкая стоимость подписки
– Copilot Workspace для планирования изменений

Преимущества Claude Code по сравнению с Copilot:

– Полноценный агентный режим с доступом к файловой системе и терминалу
– Значительно более длинное контекстное окно (200K+ токенов vs ~8K у Copilot для автодополнения)
– Способность к сложному рассуждению и планированию
– Не привязан к конкретному редактору или экосистеме
– MCP для интеграции с любыми внешними сервисами
– Лучшее понимание архитектуры и паттернов проектирования
– Возможность работы в полностью автономном режиме

Cursor IDE

Cursor – это, пожалуй, наиболее близкий конкурент Claude Code по философии подхода. Cursor перестроил VS Code вокруг AI, предоставив модели доступ ко всей кодовой базе и возможность вносить изменения в несколько файлов одновременно. Cursor поддерживает несколько моделей, включая, иронично, Claude Sonnet от Anthropic.

Преимущества Cursor:

– Полноценная IDE с визуальным интерфейсом
– Встроенный diff-просмотрщик для изменений AI
– Возможность выбора между разными моделями
– Удобная работа с множественными файлами
– Tab-completion и встроенный чат

Преимущества Claude Code по сравнению с Cursor:

– Не требует смены редактора – работает из терминала
– Полноценное выполнение команд и анализ результатов
– MCP-экосистема для расширения возможностей
– Более глубокое понимание контекста проекта
– Лучшая поддержка автоматизации и скриптинга
– Работа на удалённых серверах через SSH
– Прозрачная модель ценообразования (API-вызовы)

Amazon CodeWhisperer (Q Developer)

Amazon переименовала CodeWhisperer в Q Developer, расширив функциональность за пределы простого автодополнения. Q Developer предлагает анализ безопасности, генерацию тестов и интеграцию с сервисами AWS.

Преимущества Q Developer:

– Бесплатный для индивидуального использования
– Глубокая интеграция с AWS
– Встроенный анализ безопасности кода
– Поддержка многих языков

Преимущества Claude Code:

– Значительно более мощная базовая модель
– Агентные возможности
– Универсальность: не привязан к облачному провайдеру
– MCP для интеграций
– Лучшее качество генерации кода на большинстве языков

Google Gemini Code Assist

Google предлагает Gemini Code Assist как инструмент для разработки, интегрированный с IDE и облачной платформой Google Cloud. Используя модели семейства Gemini, инструмент обеспечивает автодополнение, чат и анализ кода.

Преимущества Gemini Code Assist:

– Огромное контекстное окно (до 1M+ токенов)
– Интеграция с Google Cloud
– Бесплатный уровень использования
– Мультимодальность (понимание изображений и скриншотов)

Преимущества Claude Code:

– Агентный режим с реальным выполнением действий
– Лучшее качество кода в большинстве сценариев
– MCP-экосистема
– Не привязан к облачному провайдеру
– Более зрелая архитектура безопасности

1.4. Философия Claude Code: AI как напарник

За Claude Code стоит определённая философия разработки, которую важно понять для эффективного использования инструмента. Anthropic позиционирует Claude Code не как замену разработчика, а как «напарника» – равноправного участника процесса разработки, который берёт на себя выполнение задач, но оставляет контроль и принятие решений человеку.

Это отличается от двух крайних подходов, которые можно наблюдать на рынке:

Подход «умное автодополнение» – AI как прокачанная версия IntelliSense. Он просто предлагает следующую строку кода, не вмешиваясь в процесс. Минимальный риск, но и минимальная трансформация рабочего процесса.

Подход «полная автономия» – AI как самостоятельный разработчик, который получает задачу и возвращает готовый результат. Максимальная автоматизация, но и максимальные риски: потеря контроля, непредсказуемые решения, сложность отладки.

Claude Code занимает промежуточную позицию. Он достаточно автономен, чтобы самостоятельно планировать и выполнять сложные задачи, но при этом:

– **Запрашивает подтверждение** перед потенциально опасными операциями (удаление файлов, установка зависимостей, выполнение неизвестных команд)
– **Объясняет свои действия** – каждый шаг сопровождается описанием того, что делается и зачем
– **Сохраняет прозрачность** – вы видите все инструменты, которые вызывает AI, и все изменения, которые он вносит

– **Допускает итерации** – вы можете скорректировать направление в любой момент
– **Уважает границы** – система разрешений позволяет точно определить, что AI может и не может делать

Эта философия выражена в концепции «HITL» (Human-In-The-Loop) – человек остаётся в цикле принятия решений. Вы не просто «запускаете и забываете» – вы руководите процессом, делегируя рутину и концентрируясь на стратегических решениях.

Практические следствия этой философии:

– **Claude Code лучше работает в диалоге.** Вместо того чтобы пытаться описать задачу одним огромным промптом, эффективнее вести итеративный диалог: поставить задачу, посмотреть промежуточный результат, скорректировать направление.

– **Claude Code уважает ваш стиль.** Он анализирует существующий код и старается следовать вашим соглашениям по именованию, форматированию и архитектурным паттернам.

– **Claude Code признаёт ошибки.** Если что-то пошло не так, он не будет настаивать на неверном решении – он переоценит ситуацию и предложит альтернативу.

– **Claude Code учится из контекста.** Файл CLAUDE.md в корне проекта позволяет задать AI постоянные инструкции: стилевые соглашения, архитектурные решения, бизнес-контекст.

1.5. Сценарии использования: от обучения до продакшена

Claude Code применяется в удивительно широком спектре ситуаций. Вот основные сценарии, каждый из которых подробно раскрывается в последующих главах:

Быстрое прототипирование

Вы придумали стартап и хотите за вечер собрать MVP? Claude Code – идеальный инструмент. Вы описываете идею, и ассистент генерирует структуру проекта, настраивает зависимости, пишет базовый функционал. За несколько часов работы можно получить рабочее веб-приложение, которое обычно потребовало бы дни или недели разработки.

Пример: «Создай веб-приложение для управления задачами с аутентификацией через Google OAuth, drag-and-drop досками в стиле Kanban, и возможностью назначать задачи участникам команды. Используй Next.js, Prisma и PostgreSQL».

Claude Code не просто сгенерирует код – он создаст файловую структуру, напишет схему базы данных, реализует API, фронтенд, настроит аутентификацию и напишет базовые тесты. Разумеется, результат потребует доработки, но как отправная точка он бесценен.

Рефакторинг и модернизация

Если у вас есть legacy-проект на устаревшем стеке, Claude Code поможет с миграцией. Он может:

– Перевести JavaScript-код на TypeScript, добавив типы
– Мигрировать с Class Components на React Hooks
– Обновить Python 2 код до Python 3
– Заменить устаревшие API на современные альтернативы
– Реорганизовать файловую структуру проекта
– Извлечь повторяющийся код в переиспользуемые модули

При этом Claude Code анализирует весь проект целиком, а не отдельные файлы, что позволяет ему делать согласованные изменения и не ломать зависимости.

Отладка сложных проблем

Когда вы столкнулись с багом, который не можете найти уже несколько часов, Claude Code может стать свежим взглядом. Он анализирует stack traces, лог-файлы, исходный код и часто находит причину проблемы за считанные минуты. Особенно ценно это для:

– Race conditions в многопоточном коде
– Memory leaks в долго работающих процессах
– Сложных зависимостей и конфликтов версий
– Проблем конфигурации в DevOps-стеке
– Ошибок в SQL-запросах и схемах данных

Написание тестов

Генерация тестов – одна из наиболее впечатляющих возможностей Claude Code. Разработчики часто откладывают написание тестов как скучную, рутинную работу. Claude Code может:

– Написать unit-тесты для существующего кода
– Создать интеграционные тесты для API
– Сгенерировать E2E-тесты для веб-приложений
– Добавить тесты для граничных случаев, которые вы могли упустить
– Настроить CI-пайплайн для автоматического запуска тестов

Документирование

Claude Code отлично справляется с задачами документирования:

– Генерация JSDoc/Docstring/KDoc комментариев для функций

– Создание README.md для проектов
– Написание API-документации
– Создание руководств для разработчиков
– Генерация CHANGELOG на основе коммитов
– Документирование архитектурных решений (ADR)

DevOps и инфраструктура

Настройка инфраструктуры – область, где Claude Code проявляет себя особенно ярко:

– Написание Dockerfile и docker-compose.yml
– Создание Kubernetes-манифестов и Helm-чартов
– Настройка CI/CD пайплайнов (GitHub Actions, GitLab CI, Jenkins)
– Terraform-конфигурации для облачных ресурсов
– Настройка веб-серверов (Nginx, Apache, Caddy)
– Конфигурация мониторинга (Prometheus, Grafana)

Обучение и менторство

Для начинающих разработчиков Claude Code служит терпеливым наставником:

– Объясняет сложные концепции на понятном языке
– Показывает лучшие практики на примерах
– Разбирает паттерны проектирования
– Помогает понять чужой код
– Предлагает альтернативные подходы с объяснением плюсов и минусов

1.6. Ограничения и реалистичные ожидания

Было бы нечестно и непрофессионально представлять Claude Code как волшебную палочку, решающую все проблемы. У инструмента есть объективные ограничения, которые важно понимать с самого начала:

Ограничение контекстного окна

Несмотря на впечатляющие 200 000 токенов контекстного окна, для крупных проектов этого может быть недостаточно. Проект с тысячами файлов и сотнями тысяч строк кода физически не помещается в контекст целиком. Claude Code решает эту задачу с помощью умного поиска и выборочного чтения файлов, но иногда может не увидеть важные зависимости или связи между далёкими частями кодовой базы.

Рекомендация: для больших проектов разбивайте задачи на локальные, затрагивающие ограниченное количество файлов. Используйте CLAUDE.md для предоставления высокоуровневого контекста об архитектуре.

Галлюцинации

Все языковые модели подвержены так называемым галлюцинациям – ситуациям, когда AI генерирует правдоподобный, но неверный ответ. Claude Code может:

– Предложить несуществующую функцию стандартной библиотеки
– Неверно описать поведение API
– Сгенерировать код с тонкими логическими ошибками
– Уверенно утверждать неверный факт

Рекомендация: всегда проверяйте критически важный код. Запускайте тесты, проводите code review, не принимайте сгенерированный код на веру. Claude Code – это черновик, который нужно проверять, а не готовый продукт.

Стоимость

Claude Code использует API Anthropic, который тарифицируется по количеству токенов. Интенсивная разработка может обойтись в существенные суммы. Особенно затратны операции с большим контекстом – чем больше файлов загружается, тем дороже каждый запрос.

Рекомендация: используйте команду `/compact` для сжатия контекста, избегайте загрузки ненужных файлов, планируйте задачи заранее, чтобы минимизировать количество итераций.

Актуальность знаний

Модель Claude обучена на данных до определённой точки отсечения. Это означает, что она может не знать о самых новых библиотеках, недавних изменениях в API или свежих уязвимостях безопасности. Claude Code частично компенсирует это, читая актуальный код и документацию из вашего проекта, но не имеет доступа к интернету для проверки информации (если не настроен MCP-сервер для этого).

Рекомендация: для работы с новейшими технологиями предоставляйте Claude Code актуальную документацию – добавляйте её в проект или передавайте через промпт.

Специфические домены

Claude Code лучше всего работает с мейнстрим-технологиями: Python, JavaScript/TypeScript, Go, Java, Rust, C/C++. Для нишевых языков (COBOL, Fortran, Erlang, Haskell) и специализированных фреймворков качество генерации может быть ниже.

Рекомендация: для редких технологий предоставляйте больше контекста – примеры кода, документацию, конвенции именования.

1.7. Что вы получите от этой книги

Прочитав эту книгу и применив описанные в ней техники на практике, вы:

– **Освоите Claude Code от установки до продвинутых сценариев.** Вы будете уверенно использовать все возможности инструмента, включая MCP, hooks, автоматизацию и командную работу.

– **Научитесь эффективно общаться с AI.** Составление промптов – это навык, и эта книга научит вас формулировать задачи так, чтобы получать максимально качественный результат с минимальным количеством итераций.

– **Построите шесть полноценных проектов.** Четвёртая часть книги – это практикум, где вы шаг за шагом создадите SaaS-платформу, мобильное приложение, CLI-утилиту, Telegram-бота, Data Pipeline и микросервисную систему.

– **Поймёте архитектурные принципы.** Claude Code – отличный инструмент для изучения архитектуры программного обеспечения, и книга использует его именно в этом контексте: каждый проект иллюстрирует определённые архитектурные паттерны.

– **Сможете внедрить Claude Code в команде.** Главы о командной работе, безопасности и стандартизации дадут вам необходимые инструменты для этого.

– **Получите стратегическое видение.** Последние главы помогут вам оценить, как AI-ассистенты изменят профессию разработчика в ближайшие годы, и подготовиться к этим изменениям.

1.8. Структура глав и условные обозначения

Каждая глава этой книги следует единой структуре:

– **Введение** – постановка задачи и мотивация
– **Теоретическая часть** – концепции и принципы
– **Практическая часть** – пошаговые инструкции с примерами кода
– **Советы и предупреждения** – типичные ошибки и лучшие практики
– **Итоги** – краткое резюме ключевых моментов

В тексте используются следующие условные обозначения:

Блоки кода содержат фрагменты, которые вы можете вводить в терминал или редактор. Комментарии внутри блоков кода объясняют ключевые моменты.

Промпты – примеры запросов к Claude Code – выделены курсивом и предваряются символом «>».

Совет – полезная рекомендация, которая поможет работать эффективнее.

Предупреждение – потенциальная ловушка или типичная ошибка.

Заметка – дополнительная информация для углублённого понимания.

Итоги главы

В этой главе мы рассмотрели:

– Историю Anthropic и эволюцию линейки Claude
– Место Claude Code среди AI-инструментов для разработки
– Подробное сравнение с GitHub Copilot, Cursor, Q Developer и Gemini Code Assist
– Философию Claude Code: AI как напарник, а не замена
– Основные сценарии использования: от прототипирования до DevOps
– Объективные ограничения и реалистичные ожидания
– Структуру книги и условные обозначения

В следующей главе мы перейдём к практике: установим Claude Code, настроим окружение и сделаем первые шаги.

Глава 2. Установка и настройка окружения

2.1. Системные требования

Прежде чем приступить к установке Claude Code, убедимся, что ваша система соответствует минимальным требованиям. Claude Code – это инструмент командной строки, построенный на платформе Node.js, что обеспечивает его кроссплатформенность, но накладывает определённые ограничения.

Аппаратные требования

Claude Code сам по себе не требует мощного оборудования – все вычисления происходят на серверах Anthropic. Однако комфортная работа предполагает:

– **Оперативная память:** минимум 4 ГБ (рекомендуется 8 ГБ и более). Claude Code использует память для хранения контекста диалога и индексов проекта.
– **Дисковое пространство:** около 500 МБ для самого инструмента и его зависимостей, плюс место для ваших проектов.
– **Процессор:** любой современный процессор. CPU не является узким местом, поскольку основные вычисления выполняются в облаке.
– **Интернет-соединение:** стабильное подключение обязательно. Без интернета Claude Code работать не будет, так как все запросы к модели отправляются через API. Рекомендованная скорость – не менее 10 Мбит/с для комфортной работы.

Программные требования

– **Node.js:** версия 18.0 или выше. Claude Code написан на TypeScript и выполняется в среде Node.js. Рекомендуется использовать LTS-версию (Long Term Support) для максимальной стабильности.
– **npm:** версия 9.0 или выше. Поставляется вместе с Node.js.
– **Операционная система:**
– macOS 12 (Monterey) или новее
– Ubuntu 20.04 / Debian 11 или новее
– Windows 10/11 с WSL2 (Windows Subsystem for Linux)
– Другие дистрибутивы Linux с glibc 2.31+
– **Терминал:** любой эмулятор терминала. Рекомендуются: iTerm2 (macOS), Windows Terminal (Windows), Alacritty, Kitty или WezTerm (кроссплатформенные).
– **Git:** версия 2.30 или выше (для работы с контролем версий).

Проверка требований

Перед установкой выполните следующие проверки в терминале:

```
node –version
npm –version
git –version
```

Если Node.js не установлен или версия устарела, рекомендуется использовать менеджер версий – nvm (Node Version Manager):

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/
v0.39.7/install.sh | bash
nvm install –lts
nvm use –lts
```

Для Windows рекомендуется nvm-windows:

```
winget install CoreyButler.NVMforWindows
nvm install lts
nvm use lts
```

2.2. Установка Claude Code

Установка через npm (рекомендуемый способ)

Основной и рекомендуемый способ установки Claude Code – через npm, менеджер пакетов Node.js:

```
npm install –g @anthropic–ai/claude–code
claude –version
```

Глобальная установка (`–g`) делает команду `claude` доступной из любого каталога в терминале. Это важно, поскольку Claude Code работает в контексте текущего каталога – того проекта, с которым вы хотите работать.

Если вы столкнулись с ошибками прав доступа при глобальной установке на Linux/macOS, есть несколько решений:

```
sudo npm install –g @anthropic–ai/claude–code
mkdir –p ~/.npm–global
npm config set prefix '~/.npm–global'
echo 'export PATH=~/.npm–global/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
npm install –g @anthropic–ai/claude–code
npx @anthropic–ai/claude–code
```

Установка через Homebrew (macOS)

На macOS доступна установка через Homebrew:

```
brew install claude-code
brew tap anthropic-ai/tap
brew install anthropic-ai/tap/claude-code
```

Установка на Windows через WSL2

Claude Code официально поддерживает Windows только через WSL2. Вот пошаговая инструкция:

```
wsl -install
sudo apt update && sudo apt upgrade -y
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/
v0.39.7/install.sh | bash
source ~/.bashrc
nvm install -lts
npm install -g @anthropic-ai/claude-code
```

Важная деталь: при работе в WSL2 убедитесь, что ваш проект находится внутри файловой системы WSL (например, `/home/username/projects/`), а не на диске Windows (`/mnt/c/...`). Работа с файлами на диске Windows через WSL значительно медленнее из-за трансляции файловых систем.

Установка в Docker

Для изолированной среды или CI/CD можно использовать Docker:

```
FROM node:20-slim
RUN npm install -g @anthropic-ai/claude-code
WORKDIR /app
COPY . .
CMD ["claude", "-print", "Analyze this project"]
```

Обновление

Claude Code активно развивается, и новые версии выходят регулярно. Для обновления:

```
npm update -g @anthropic-ai/claude-code
npm install -g @anthropic-ai/claude-code@latest
```

Рекомендуется обновляться регулярно – как минимум раз в две недели. Каждое обновление может содержать улучшения качества, новые инструменты и исправления ошибок.

2.3. Аутентификация и API-ключи

Claude Code взаимодействует с моделями Anthropic через API, для чего необходим ключ аутентификации. Существует несколько способов настроить доступ.

Способ 1: Прямой вход через Anthropic

При первом запуске Claude Code предложит авторизоваться через веб-браузер:

```
claude
```

Откройте предложенную ссылку в браузере, войдите в свой аккаунт Anthropic и подтвердите авторизацию. После этого Claude Code сохранит токен авторизации локально и будет использовать его для последующих сессий.

Этот способ наиболее удобен для индивидуального использования: вам не нужно вручную управлять ключами, а токен автоматически обновляется.

Способ 2: API-ключ через переменную окружения

Для автоматизации, CI/CD или серверного использования удобнее задавать API-ключ через переменную окружения:

```
export ANTHROPIC_API_KEY="sk-ant-…"
echo 'export ANTHROPIC_API_KEY="sk-ant-…"' >> ~/.bashrc
source ~/.bashrc
```

Для более безопасного хранения ключей используйте менеджер секретов:

```
echo 'export ANTHROPIC_API_KEY="sk-ant-…"' > .envrc
direnv allow
export ANTHROPIC_API_KEY=$(op read "op://Development/Anthropic
API Key/credential")
aws-vault exec dev - claude
```

Никогда не храните API-ключи в файлах, которые могут попасть в Git-репозиторий. Добавьте `.envrc` в `.gitignore`.

Способ 3: AWS Bedrock

Для организаций, использующих AWS, Claude доступен через Amazon Bedrock:

```
export CLAUDE_CODE_USE_BEDROCK=1
export AWS_REGION=us-east-1
export AWS_ACCESS_KEY_ID="AKIA..."
export AWS_SECRET_ACCESS_KEY="..."
```

Преимущества использования Bedrock: единый биллинг через AWS, SLA на корпоративном уровне, соответствие требованиям compliance (SOC2, HIPAA, FedRAMP).

Способ 4: Google Vertex AI

Аналогично, Claude доступен через Google Cloud Vertex AI:

```
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=us-east5
export ANTHROPIC_VERTEX_PROJECT_ID="your-project-id"
```

Проверка аутентификации

После настройки убедитесь, что всё работает:

```
claude -print "Hello, world!"
```

Если возникает ошибка аутентификации, проверьте:

– Правильность API-ключа
– Наличие достаточного баланса на аккаунте
– Доступность серверов API (status.anthropic.com)
– Корректность сетевых настроек (прокси, VPN, файрвол)

2.4. Конфигурационные файлы

Claude Code использует систему конфигурационных файлов для настройки поведения. Понимание этой системы критически важно для эффективной работы.

Иерархия конфигурации

Claude Code применяет иерархическую систему настроек с несколькими уровнями приоритета:

– **Аргументы командной строки** – высший приоритет
– **Переменные окружения**
– **Проектная конфигурация** (`.claude/` в каталоге проекта)
– **Пользовательская конфигурация** (`~/.claude/`)

– Системные настройки по умолчанию – низший приоритет

Файл CLAUDE.md

Файл `CLAUDE.md` – один из самых мощных инструментов настройки Claude Code. Это файл в формате Markdown, расположенный в корне проекта (или в любом подкаталоге для более гранулярной настройки). Claude Code автоматически читает его и использует содержимое как постоянные инструкции.

Пример `CLAUDE.md` для типичного Python-проекта:

```
    ## Описание
    REST API для приложения управления задачами. Python 3.12,
FastAPI, SQLAlchemy, PostgreSQL.
    ## Архитектура
    - `app/` - основной код приложения
    - `api/` - маршруты и эндпоинты
    - `models/` - SQLAlchemy модели
    - `schemas/` - Pydantic схемы
    - `services/` - бизнес-логика
    - `core/` - конфигурация, безопасность, зависимости
    - `tests/` - тесты (pytest)
    - `alembic/` - миграции базы данных
    ## Соглашения по коду
    - Используй type hints везде
    - Используй async/await для всех эндпоинтов
    - Именование: snake_case для функций, PascalCase для классов
    - Формат строк: f-strings
    - Максимальная длина строки: 100 символов
    - Используй ruff для форматирования и линтинга
    ## Команды
    - Запуск: `uvicorn app.main:app --reload`
    - Тесты: `pytest -v`
    - Линтинг: `ruff check .`
    - Миграции: `alembic upgrade head`
    ## Важно
    - Не используй print() - всегда logging
    - Все эндпоинты должны иметь описание и теги для OpenAPI
    - Бизнес-логика НЕ должна быть в маршрутах - выноси в services
    - Пароли хэшировать через bcrypt
    - JWT-токены со сроком жизни 30 минут
```

Claude Code будет следовать этим инструкциям во всех своих действиях. Это как README для вашего AI-напарника – чем подробнее вы опишете проект, тем лучше Claude Code будет с ним работать.

Каталог .claude/

Каталог `.claude/` в корне проекта содержит дополнительные конфигурационные файлы:

```
.claude/
+- settings.json # Настройки проекта
+- commands/ # Пользовательские команды
| +- review.md # /project:review
| +- deploy.md # /project:deploy
+- .cursorrules # Совместимость с Cursor
```

Файл `settings.json` управляет поведением Claude Code на уровне проекта:

```
{
"permissions": {
"allow": [
"Read",
"Write",
"Bash(npm run *)",
"Bash(pytest *)",
"Bash(git *)"
],
"deny": [
"Bash(rm -rf *)",
"Bash(sudo *)",
"Bash(curl * | bash)"
]
},
"model": "claude-sonnet-4-20250514",
"maxTurns": 50
}
```

Пользовательская конфигурация (~/.claude/)

Домашний каталог содержит глобальные настройки:

```
~/.claude/
+- settings.json # Глобальные настройки
+- commands/ # Глобальные команды
| +- standup.md # /user:standup
+- credentials.json # Токены авторизации (не редактировать)
+- projects/ # Кэш и состояние проектов
```

Глобальные настройки применяются ко всем проектам и могут быть переопределены проектными:

```
{
"theme": "dark",
"verbose": false,
"autoApprove": ["Read"],
"notifications": true,
"defaultModel": "claude-sonnet-4-20250514"
}
```

Переменные окружения

Claude Code поддерживает множество переменных окружения для тонкой настройки:

```
ANTHROPIC_API_KEY # API-ключ Anthropic
CLAUDE_CODE_USE_BEDROCK # Использовать AWS Bedrock
CLAUDE_CODE_USE_VERTEX # Использовать Google Vertex AI
CLAUDE_MODEL # Модель по умолчанию
HTTP_PROXY # HTTP прокси
HTTPS_PROXY # HTTPS прокси
NO_PROXY # Исключения для прокси
CLAUDE_CODE_MAX_TURNS # Максимальное количество шагов
CLAUDE_CODE_DISABLE_TELEMETRY # Отключение телеметрии
```

2.5. Интеграция с редакторами кода

Хотя Claude Code работает из терминала и не зависит от конкретного редактора, существуют интеграции, которые делают работу более удобной.

Visual Studio Code

VS Code – самый популярный редактор среди разработчиков, и для него доступно расширение Claude Code:

```
code -install-extension anthropic.claude-code-vscode
```

Расширение добавляет:

– Панель Claude Code в боковой панели
– Возможность отправлять выделенный код в Claude Code
– Отображение изменений, внесённых Claude Code, в diff-просмотрщике
– Подсветку файла CLAUDE.md
– Статус-бар с информацией о текущей сессии

Альтернативный подход – использовать встроенный терминал VS Code. Откройте терминал (`Ctrl+` или Cmd+` `), перейдите в каталог проекта и запустите

claude`. Оба окна – редактор и Claude Code – будут работать бок о бок, и вы сможете видеть изменения в файлах в реальном времени.

Полезные настройки VS Code для работы с Claude Code:

```
// settings.json
{
"files.autoSave": "afterDelay",
"files.autoSaveDelay": 1000,
"editor.formatOnSave": true,
"git.autofetch": true,
"terminal.integrated.scrollback": 10000
}
```

JetBrains IDE (IntelliJ IDEA, PyCharm, WebStorm)

Для семейства JetBrains IDE существует плагин Claude Code:

– Откройте Settings -> Plugins -> Marketplace
– Найдите «Claude Code»
– Установите и перезапустите IDE

Плагин интегрируется в панель инструментов и позволяет:

– Отправлять выделенный код или файл в Claude Code
– Просматривать изменения через встроенный diff-просмотрщик JetBrains
– Использовать Claude Code из встроенного терминала

Как и в случае с VS Code, самый простой способ – открыть встроенный терминал и запустить `claude` из каталога проекта.

Vim / Neovim

Для пользователей Vim/Neovim рекомендуется работать с Claude Code через разделение терминала (tmux, screen или встроенная функция терминального мультиплексора):

```
tmux new-session -d -s dev
tmux split-window -h
tmux send-keys -t dev:0.0 'vim .' Enter
tmux send-keys -t dev:0.1 'claude' Enter
tmux attach -t dev
```

Для Neovim существуют плагины интеграции:

```
-- lazy.nvim
{
"anthropic/claude-code.nvim",
```

```
config = function()
require("claude-code").setup({
auto_open = false,
split = "vertical",
})
end,
}
```

Emacs

Для Emacs доступен пакет `claude-code.el`:

```
;; Через use-package
(use-package claude-code
:ensure t
:config
(setq claude-code-auto-approve-reads t))
```

Или просто используйте `M-x shell` или `M-x vterm` для запуска Claude Code в буфере терминала внутри Emacs.

2.6. Настройка рабочего окружения

Для максимально продуктивной работы с Claude Code рекомендуется настроить ряд дополнительных инструментов.

Терминальный мультиплексор

tmux или screen позволяют работать с несколькими панелями терминала одновременно:

```
brew install tmux
sudo apt install tmux
set -g mouse on
set -g history-limit 50000
set -g default-terminal "screen-256color"
bind | split-window -h
bind - split-window -v
```

Типичная раскладка для работы с Claude Code:

– Левая панель: редактор кода (Vim/Neovim)
– Правая верхняя панель: Claude Code
– Правая нижняя панель: терминал для запуска команд вручную

Инструменты для работы с файлами

Claude Code активно работает с файлами, и полезно иметь инструменты для их просмотра:

```
brew install bat
sudo apt install bat
brew install fd # или apt install fd-find
brew install ripgrep # или apt install ripgrep
brew install fzf # или apt install fzf
brew install eza # или cargo install eza
brew install git-delta # или cargo install git-delta
```

Настройка Git для работы с delta:

```
[core]
pager = delta
[interactive]
diffFilter = delta --color-only
[delta]
navigate = true
side-by-side = true
line-numbers = true
```

Настройка Shell

Удобная оболочка ускоряет работу:

```
sh -c "$(curl -fsSL https://raw.githubusercontent.com/ohmyzsh/
ohmyzsh/master/tools/install.sh)"
plugins=(
git
docker
kubectl
node
npm
python
fzf
zsh-autosuggestions
zsh-syntax-highlighting
)
alias cc='claude'
alias ccp='claude -print'
alias ccr='claude -resume'
alias ccc='claude -continue'
```

2.7. Первый запуск и ориентация

После установки и настройки давайте выполним первый запуск и познакомимся с интерфейсом.

Запуск

Откройте терминал, перейдите в каталог проекта и запустите Claude Code:

```
cd ~/projects/my-project
claude
```

При первом запуске вы увидите приветственное сообщение и приглашение ввести запрос:

```
+============================================+
| Welcome to Claude Code! |
| |
| Model: claude-sonnet-4-20250514 |
| Context: 200,000 tokens |
| Working directory: ~/projects/my-project |
+============================================+
Type your request or /help for commands.
>
```

Основные команды

Claude Code поддерживает набор слэш-команд для управления:

```
/help – Справка по командам
/clear – Очистка диалога и сброс контекста
/compact – Сжатие контекста для экономии токенов
/status – Информация о текущей сессии
/cost – Стоимость текущей сессии
/model – Смена модели
/permissions – Управление разрешениями
/config – Настройки
/bug – Отправка отчёта об ошибке
/quit или Ctrl+C – Выход
```

Проверка работоспособности

Попробуйте простой запрос:

```
    > Покажи список файлов в текущем каталоге и опиши структуру
проекта
    Claude Code выполнит команду ls или find, прочитает ключевые
файлы
    (package.json, requirements.txt, README.md и т.п.)
    и выдаст описание структуры проекта.
```

Если Claude Code успешно выводит ответ, значит установка прошла корректно.

2.8. Типичные проблемы при установке и их решение

Ошибка «command not found: claude»

Наиболее частая проблема. Причины и решения:

```
npm list -g @anthropic-ai/claude-code
echo $PATH
export PATH=$(npm config get prefix)/bin:$PATH
```

Ошибка «EACCES: permission denied»

```
mkdir -p ~/.npm-global
npm config set prefix '~/.npm-global'
export PATH=~/.npm-global/bin:$PATH
nvm install -lts
```

Ошибка аутентификации (401 Unauthorized)

```
echo $ANTHROPIC_API_KEY
claude logout
claude login
```

Ошибка «rate limit exceeded»

Anthropic ограничивает количество запросов для защиты от злоупотреблений. Решения:

– Подождать несколько минут и повторить
– Перейти на более высокий тарифный план
– Использовать `/compact` для уменьшения размера запросов
– Разбить задачу на более мелкие части

Проблемы с прокси/VPN

```
export HTTP_PROXY=http://proxy.company.com:8080
export HTTPS_PROXY=http://proxy.company.com:8080
export NO_PROXY=localhost,127.0.0.1
export NODE_EXTRA_CA_CERTS=/path/to/corporate-ca.crt
```

Проблемы на Windows/WSL

```
wsl -list -verbose
wsl -set-version Ubuntu 2
nslookup api.anthropic.com
echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf
```

2.9. Управление затратами

Claude Code использует API-тарификацию Anthropic, и понимание модели ценообразования поможет контролировать расходы.

Модель ценообразования

Стоимость рассчитывается за количество токенов:

– **Входные токены** (ваш промпт + контекст): тарифицируются по одной ставке
– **Выходные токены** (ответ Claude): тарифицируются по более высокой ставке
– **Кэшированные токены**: тарифицируются со скидкой (если контекст не изменился)

Цены различаются в зависимости от модели:

– Claude Sonnet – оптимальный баланс цены и качества
– Claude Opus – дороже, но лучше для сложных задач
– Claude Haiku – дешевле, подходит для простых задач

Стратегии экономии

– **Используйте `/compact` регулярно.** Эта команда сжимает историю диалога, сохраняя ключевую информацию и уменьшая количество входных токенов.

– **Начинайте с Sonnet.** В большинстве случаев Sonnet обеспечивает достаточное качество. Переключайтесь на Opus только для особо сложных задач.

– **Будьте конкретны.** Чем точнее ваш промпт, тем меньше итераций потребуется. Вместо «исправь баги» скажите «в файле auth.py функция login_user возвращает 500 при невалидном email, а должна возвращать 422 с подробным описанием ошибки».

– **Используйте CLAUDE.md.** Предоставление контекста через CLAUDE.md снижает необходимость в повторных объяснениях и уменьшает количество операций чтения файлов.

– **Разбивайте задачи.** Одна большая задача может потребовать загрузки всей кодовой базы в контекст. Несколько мелких задач работают с меньшим контекстом.

– **Мониторьте расходы.** Используйте команду `/cost` для отслеживания расходов текущей сессии и панель управления на console.anthropic.com для общей статистики.

Бюджетирование

Для оценки бюджета учитывайте:

– Активный разработчик тратит в среднем 10-50 долларов в месяц
– Интенсивное использование (целый рабочий день каждый день) может достигать 100-200 долларов
– Для команд удобнее использовать централизованный биллинг через AWS Bedrock или Google Vertex

Итоги главы

В этой главе мы:

– Проверили системные требования для Claude Code
– Установили Claude Code через npm, Homebrew или в Docker
– Настроили аутентификацию через веб-вход, API-ключ или облачного провайдера
– Изучили систему конфигурационных файлов: CLAUDE.md, .claude/, ~/.claude/
– Настроили интеграцию с VS Code, JetBrains, Vim/Neovim и Emacs
– Оптимизировали рабочее окружение: tmux, утилиты командной строки, алиасы
– Разобрали типичные проблемы при установке и их решения
– Научились управлять затратами на API

В следующей главе мы сделаем первые практические шаги: создадим проект, научимся основным паттернам взаимодействия и напишем первый код с помощью Claude Code.

Глава 3. Первые шаги с Claude Code

3.1. Запуск и интерфейс командной строки

Итак, Claude Code установлен, аутентификация настроена, и вы готовы к работе. Откройте терминал и перейдите в каталог с проектом – или созданим новый:

```
mkdir -p ~/projects/my-first-project
cd ~/projects/my-first-project
claude
```

Claude Code запускается и представляет лаконичный интерфейс командной строки. В отличие от громоздких графических приложений, здесь нет кнопок, менюю или панелей инструментов – только текстовое поле для ввода и область для отображения ответов. Эта минималистичность обманчива: за ней скрывается мощная система, способная выполнять сложнейшие операции.

Интерфейс Claude Code состоит из нескольких элементов:

Строка приглашения (prompt) – символ >, за которым вы вводите свои запросы. Здесь вы общаетесь с AI на естественном языке: задаёте вопросы, ставите задачи, просите объяснить код.

Область вывода – пространство, где Claude Code отображает свои ответы, включая текст, код, визуализации прогресса и результаты выполнения инструментов.

Индикаторы инструментов – при выполнении операций (чтение файлов, запуск команд, запись в файлы) Claude Code показывает, какой инструмент используется и с какими параметрами. Это обеспечивает прозрачность: вы всегда видите, что именно делает AI.

Строка состояния – внизу экрана отображается информация о текущей модели, стоимости сессии и количестве использованных токенов.

Режимы работы

Claude Code поддерживает несколько режимов работы:

Интерактивный режим (по умолчанию) – вы ведёте диалог с Claude Code, поочерёдно отправляя запросы и получая ответы. Это основной режим для повседневной работы.

```
claude # Запуск в интерактивном режиме
```

Однократный режим (print mode) – Claude Code обрабатывает один запрос и завершает работу. Удобно для скриптов и автоматизации.

```
claude –print "Объясни, что делает файл main.py"
```

Режим продолжения – возобновление предыдущего диалога:

```
claude –continue # Продолжить последний диалог
claude –resume # Выбрать из списка предыдущих диалогов
```

Pipe-режим – Claude Code принимает входные данные через стандартный ввод:

```
cat error.log | claude –print "Проанализируй эти логи ошибок"
git diff | claude –print "Проведи code review этих изменений"
```

Pipe-режим – невероятно мощная возможность, потому что он позволяет встраивать Claude Code в любой конвейер Unix-команд. Представьте:

```
grep –rn "TODO:" src/ | claude –print "Расставь приоритеты для этих задач"
git log –oneline –20 | claude –print "Напиши changelog из этих коммитов"
npm audit –json | claude –print "Объясни найденные уязвимости и предложи решения"
```

3.2. Анатомия диалога

Каждое взаимодействие с Claude Code можно разложить на несколько фаз. Понимание этих фаз поможет строить более эффективные диалоги.

Фаза 1: Получение запроса

Вы вводите запрос на естественном языке. Claude Code принимает запрос, добавляет к нему системный контекст (содержимое CLAUDE.md, информацию о проекте, историю диалога) и отправляет на обработку.

Фаза 2: Планирование

Claude Code анализирует запрос и определяет, какие действия необходимы для его выполнения. Для сложных задач он может сформулировать явный план:

```
> Добавь аутентификацию через JWT в наш API
Claude Code:
Для добавления JWT-аутентификации мне потребуется:
1. Установить необходимые зависимости (PyJWT, passlib)
2. Создать модель User
3. Написать функции хеширования паролей
```

```
4. Реализовать эндпоинты регистрации и входа
5. Создать middleware для проверки токенов
6. Обновить существующие эндпоинты
Начинаю с зависимостей…
```

Фаза 3: Выполнение инструментов

Claude Code последовательно вызывает инструменты для реализации плана. Каждый вызов отображается в интерфейсе:

```
Tool: ReadFile
File: requirements.txt
Tool: Bash
Command: pip install PyJWT passlib[bcrypt]
Tool: WriteFile
File: app/models/user.py
[показ содержимого файла]
Tool: WriteFile
File: app/core/security.py
[показ содержимого файла]
```

Фаза 4: Запрос подтверждения (при необходимости)

Для потенциально опасных операций Claude Code запрашивает ваше подтверждение:

```
Tool: Bash
Command: pip install PyJWT passlib[bcrypt]
Allow? [Y/n/always]
```

Варианты ответа:

– `Y` или Enter – разрешить выполнение
– `n` – отклонить
– `always` – разрешить и запомнить для этого типа операций

Фаза 5: Подведение итогов

После выполнения всех шагов Claude Code подводит итог:

```
Готово! Я добавил JWT-аутентификацию:
[+] Установлены PyJWT и passlib
[+] Создана модель User с хешированием пароля
[+] Реализованы эндпоинты /auth/register и /auth/login
[+] Создан middleware verify_token
[+] Обновлены защищённые маршруты
```

```
Для тестирования:
curl -X POST http://localhost:8000/auth/register \
-H "Content-Type: application/json" \
-d '{"email": "user@example.com", "password": "secret123"}'
```

3.3. Базовые команды и навигация

Слэш-команды

Claude Code поддерживает набор слэш-команд для управления сессией. Разберём каждую подробно.

/help – Справка

```
> /help
Доступные команды:
/help Показать эту справку
/clear Очистить контекст диалога
/compact Сжать контекст (экономия токенов)
/status Информация о сессии
/cost Стоимость текущей сессии
/model Переключить модель
/permissions Управление разрешениями
/config Настройки
/bug Отправить отчёт об ошибке
/quit Выход
```

/clear – Сброс контекста

Команда `/clear` полностью сбрасывает контекст диалога. Используйте её, когда:

– Переключаетесь на совершенно другую задачу
– Контекст стал слишком большим и замедляет работу
– Claude Code «запутался» и даёт неадекватные ответы

После `/clear` Claude Code забудет всё, что обсуждалось ранее, но сохранит конфигурацию проекта (CLAUDE.md, settings и т. д.).

/compact – Сжатие контекста

`/compact` – одна из наиболее полезных команд. Она заставляет Claude Code сжать историю диалога, сохранив ключевую информацию, но уменьшив объём контекста. Это экономит токены и деньги.

```
> /compact
Контекст сжат:
До: 45 832 токена
```

```
После: 12 447 токенов
Сэкономлено: 33 385 токенов (~73%)
```

Рекомендуется вызывать `/compact` после каждого крупного блока работы – например, после завершения реализации конкретной функции.

Вы также можете указать фокус при сжатии:

```
> /compact Сфокусируйся на работе с базой данных
Claude Code сожмёт контекст, но сохранит детали, связанные
с базой данных, и удалит менее релевантную информацию.
```

/status – Информация о сессии

```
> /status
Session Info:
Model: claude-sonnet-4-20250514
Context used: 23,451 / 200,000 tokens (12%)
Messages: 14
Files read: 8
Files written: 3
Commands executed: 5
Duration: 12m 34s
Cost: $0.47
```

/cost – Детализация стоимости

```
> /cost
Cost Breakdown:
Input tokens: 187,234 ($0.32)
Output tokens: 41,892 ($0.15)
Cached tokens: 124,891 ($0.04)
Total: $0.47
```

Горячие клавиши

Claude Code поддерживает стандартные горячие клавиши терминала:

– `Ctrl+C` – отменить текущую операцию или выйти
– `Ctrl+D` – выйти из Claude Code
– `Up/Down` – навигация по истории запросов
– `Tab` – автодополнение слэш-команд
– `Ctrl+L` – очистить экран (без сброса контекста)
– `Esc` – отменить ввод текущего промпта

Многострочный ввод

Для длинных промптов используйте многострочный ввод. Нажмите \ в конце строки для продолжения на следующей, или используйте Shift+Enter (в поддерживающих терминалах):

```
> Создай REST API со следующими эндпоинтами: \
GET /users – список пользователей, \
POST /users – создание пользователя, \
GET /users/:id – получение пользователя, \
PUT /users/:id – обновление, \
DELETE /users/:id – удаление. \
Используй Express.js и MongoDB.
```

3.4. Управление контекстным окном

Контекстное окно – это «рабочая память» Claude Code. Всё, что вы обсудили, все прочитанные файлы и результаты выполненных команд хранятся в этом окне. Когда окно заполняется, старая информация начинает вытесняться, что может привести к потере контекста.

Размер контекстного окна

В зависимости от модели, контекстное окно Claude Code составляет:

– Claude Sonnet: ~200 000 токенов
– Claude Opus: ~200 000 токенов
– Claude Haiku: ~200 000 токенов

Один токен – это примерно 4 символа для английского текста и 1-2 символа для русского. Файл на 1000 строк занимает примерно 10 000-20 000 токенов.

Стратегии управления контекстом

1. Будьте избирательны. Не просите Claude Code «прочитай все файлы в проекте». Вместо этого укажите конкретные файлы или каталоги:

```
> Проанализируй весь проект
> Проанализируй файлы в каталоге src/auth/ – мне нужно
понять текущую систему аутентификации
```

2. Используйте /compact регулярно. Как упоминалось ранее, сжатие контекста высвобождает место для новой информации.

3. Разбивайте задачи. Вместо одного длинного диалога ведите несколько коротких. Завершили реализацию одной функции – начните новый диалог для следующей.

4. Используйте CLAUDE.md. Информация из CLAUDE.md загружается автоматически и занимает меньше места, чем повторные объяснения в диалоге.

5. Следите за индикатором. Команда `/status` показывает, какой процент контекстного окна использован. При приближении к 70-80% подумайте об оптимизации.

Что происходит при переполнении

Когда контекстное окно заполняется, Claude Code применяет алгоритм сжатия:

– Старые сообщения сжимаются в краткие резюме
– Результаты выполненных команд усекаются
– Содержимое прочитанных файлов может быть частично удалено
– Системные промпты и CLAUDE.md сохраняются в приоритете

Вы можете заметить, что Claude Code «забыл» что-то, что обсуждалось ранее. Это нормально – просто напомните ему нужный контекст.

3.5. Первый проект: создание веб-приложения

Давайте создадим полноценный проект, чтобы увидеть Claude Code в действии. Мы построим простое, но функциональное веб-приложение – менеджер заметок.

Шаг 1: Инициализация проекта

```
> Создай новый проект – менеджер заметок.
Стек: Node.js, Express, SQLite, EJS для шаблонов.
Базовая функциональность: создание, чтение, редактирование,
удаление заметок. Каждая заметка имеет заголовок, содержимое
и дату создания/обновления.
```

Claude Code начнёт работу:

– Создаст `package.json` с зависимостями
– Инициализирует npm-проект
– Создаст структуру каталогов:

```
notes-app/
+- package.json
+- server.js
+- database.js
+- routes/
| +- notes.js
+- views/
```

```
| +- layout.ejs
| +- index.ejs
| +- note.ejs
| +- edit.ejs
+- public/
| +- style.css
+- README.md
```

– Напишет код сервера, маршруты, модели и шаблоны
– Установит зависимости
– Проверит работоспособность

Шаг 2: Проверка и запуск

После генерации проверим, что всё работает:

```
> Запусти приложение и проверь, что главная страница открыва-
ется
```

Claude Code выполнит:

```
node server.js
```

И проверит доступность через curl или встроенный инструмент.

Шаг 3: Доработка

Теперь добавим функциональность:

```
> Добавь поиск по заметкам: поле поиска на главной странице,
которое фильтрует заметки по заголовку и содержимому.
Поиск должен быть нечувствительным к регистру.
```

Claude Code изменит несколько файлов:

– Добавит маршрут `/search` в `routes/notes.js`
– Добавит SQL-запрос с `LIKE` в `database.js`
– Обновит шаблон `index.ejs`, добавив форму поиска
– Обновит стили в `style.css`

Шаг 4: Итерация

Этот процесс можно продолжать сколько угодно:

```
> Добавь категории для заметок
```

```
> Добавь markdown-рендеринг в содержимом заметки
> Добавь экспорт заметок в PDF
> Добавь тёмную тему
```

Каждый запрос – это итерация, которая наращивает функциональность проекта. Claude Code помнит контекст предыдущих шагов и строит новые функции на существующей основе.

3.6. Работа с файлами

Работа с файлами – одна из основных возможностей Claude Code. Разберём ключевые операции.

Чтение файлов

Claude Code может читать файлы различными способами:

```
> Покажи содержимое файла server.js
> Прочитай все файлы в каталоге src/
> Что находится в package.json?
```

При чтении файла Claude Code отображает его содержимое и подтверждает, что файл загружен в контекст. Это позволяет модели анализировать код и предлагать изменения.

Создание файлов

```
> Создай файл config.js с настройками приложения:
порт, строка подключения к БД, секретный ключ для JWT
```

Claude Code создаст файл с осмысленным содержимым:

```
// config.js
const config = {
port: process.env.PORT || 3000,
database: {
url: process.env.DATABASE_URL || 'sqlite:./data/app.db',
},
jwt: {
secret: process.env.JWT_SECRET || 'change-me-in-production',
expiresIn: '24h',
},
app: {
name: 'Notes Manager',
version: '1.0.0',
},
};
```

```
module.exports = config;
```

Редактирование файлов

Claude Code может вносить точечные изменения, не перезаписывая весь файл:

```
> В файле server.js добавь middleware для логирования
запросов с помощью morgan
```

Claude Code прочитает файл, определит нужное место для вставки и внесёт минимальные изменения:

```
// Добавлено:
const morgan = require('morgan');
// ...
app.use(morgan('dev'));
```

Удаление файлов

```
> Удали файл temp.js – он больше не нужен
```

Claude Code запросит подтверждение перед удалением.

Переименование и перемещение

```
> Переименуй database.js в db/connection.js
```

Claude Code создаст каталог, переместит файл и обновит все import/require-ссылки в проекте.

Массовые операции

Одно из преимуществ агентного подхода – способность выполнять массовые операции:

```
> Добавь JSDoc-комментарии ко всем экспортируемым функциям
в каталоге src/utils/
```

Claude Code последовательно прочитает каждый файл, проанализирует функции, добавит документацию и сохранит изменения.

3.7. Работа с терминалом

Claude Code может выполнять команды в терминале – это даёт ему возможность непосредственно взаимодействовать с операционной системой, инструментами разработки и инфраструктурой.

Безопасность выполнения команд

Прежде чем обсуждать возможности, важно подчеркнуть: Claude Code не выполняет произвольные команды без вашего ведома. Система разрешений работает следующим образом:

– **Безопасные команды** (чтение файлов, ls, cat, find, grep) – выполняются без подтверждения
– **Команды разработки** (npm run, pytest, go build) – зависят от настроек
– **Потенциально опасные** (установка пакетов, изменение конфигурации) – требуют подтверждения
– **Опасные** (rm, sudo, curl | bash) – всегда требуют подтверждения

Вы можете настроить уровень доверия через разрешения в `.claude/settings.json`.

Типичные сценарии выполнения команд

Установка зависимостей:

```
> Установи необходимые зависимости для проекта
Claude Code: Для этого мне нужно выполнить:
npm install express ejs better-sqlite3 morgan
Allow? [Y/n/always]
```

Запуск тестов:

```
> Запусти тесты и исправь падающие
Claude Code выполнит `npm test`, проанализирует вывод,
найдёт причины ошибок и исправит код.
```

Сборка проекта:

```
> Собери проект для продакшена
Claude Code выполнит `npm run build`, проверит результат,
исправит проблемы при необходимости.
```

Работа с Git:

```
> Создай коммит с осмысленным сообщением
```

```
Claude Code проанализирует diff, сформулирует сообщение
и выполнит `git add` + `git commit`.
```

Анализ системы:

```
> Покажи использование дискового пространства в проекте
> Какие процессы занимают порт 3000?
> Проверь, запущен ли Docker
```

Конвейеры команд

Claude Code может строить сложные конвейеры команд и анализировать их результаты:

```
> Найди все TypeScript файлы с неиспользуемыми импортами
Claude Code может выполнить:
npx ts-prune | head -50
И затем проанализировать результат, показав вам список
файлов с неиспользуемыми экспортами.
```

3.8. Навигация по проекту

Одна из мощных возможностей Claude Code – способность самостоятельно навигировать по проекту. Вы не обязаны указывать точные имена файлов и пути – Claude Code найдёт нужное самостоятельно.

Поиск файлов

```
> Где определена функция calculateDiscount?
Claude Code использует grep/ripgrep для поиска:
rg "function calculateDiscount|calculateDiscount =" src/
И найдёт: src/utils/pricing.js:42
```

Понимание зависимостей

```
> Какие файлы импортируют модуль database.js?
Claude Code найдёт все файлы, которые используют
require('./database') или import from './database'.
```

Анализ структуры

```
> Опиши архитектуру проекта
Claude Code прочитает ключевые файлы (package.json,
структуру каталогов, основные модули) и составит
```

```
описание архитектуры.
```

Поиск по содержимому

```
> Найди все места, где используется deprecated API
из библиотеки axios
Claude Code выполнит поиск по шаблонам и покажет
результаты с контекстом.
```

3.9. Мультимодальность: работа с изображениями

Claude Code (через модели Claude) поддерживает мультимодальность – он может анализировать изображения. Это полезно в нескольких сценариях:

Реализация дизайна

Если у вас есть макет интерфейса в виде изображения, вы можете передать его Claude Code:

```
> Вот макет главной страницы [скриншот].
Реализуй этот дизайн в HTML/CSS.
```

Claude Code проанализирует изображение и создаст HTML/CSS-код, максимально приближенный к макету.

Анализ скриншотов ошибок

```
> Вот скриншот ошибки в браузере [изображение].
Что может быть причиной?
```

Claude Code прочитает текст ошибки с изображения и предложит решение.

Диаграммы и схемы

```
> Вот диаграмма архитектуры [изображение].
Создай Docker Compose файл на основе этой схемы.
```

3.10. Паттерны эффективного взаимодействия

За время работы с Claude Code сообщество выработало ряд паттернов, которые значительно повышают эффективность.

Паттерн «Контекст -> Задача -> Ограничения»

Структурируйте запросы в три блока:

```
> [Контекст] Мы разрабатываем e-commerce платформу на Django.
Сейчас работаем над модулем корзины покупок.
[Задача] Нужно реализовать функцию применения промокодов.
Промокод может давать процентную скидку (10%, 20%) или
фиксированную (500 руб, 1000 руб). Один промокод на заказ.
[Ограничения] Используй существующую модель CartItem.
Промокоды хранятся в таблице Promo. Валидацию делай на
уровне сервиса, не в модели.
```

Паттерн «Обзор -> Детализация -> Реализация»

Для сложных задач разбивайте работу на три этапа:

```
> Предложи архитектуру системы уведомлений.
Поддержка: email, push, SMS, in-app.
> Отлично, давай детализируем вариант с очередями.
Как будет выглядеть интерфейс NotificationService?
> Реализуй NotificationService согласно обсуждённому плану.
```

Паттерн «Покажи перед изменением»

Перед внесением изменений в критический код попросите Claude Code показать план:

```
> Покажи, как ты собираешься изменить auth middleware,
но пока не меняй файлы. Я хочу посмотреть план.
```

Это даёт возможность скорректировать направление до того, как будут внесены изменения.

Паттерн «Исправь и проверь»

После каждого блока изменений просите запустить проверку:

```
> Исправь ошибку в парсере и запусти тесты,
чтобы убедиться, что ничего не сломалось.
```

Этот паттерн помогает ловить регрессии на ранней стадии.

Паттерн «Объясни как учителю»

Когда вы хотите не просто получить код, но и понять решение:

```
> Реализуй алгоритм LRU-кэша и объясни каждый шаг,
как будто я джуниор-разработчик. Почему именно
такая структура данных? Какие альтернативы?
```

3.11. Управление сессиями

Claude Code поддерживает сохранение и возобновление диалогов, что позволяет продолжить работу с того места, где вы остановились.

Продолжение последнего диалога

```
claude -continue
claude -c
```

Это загрузит последний диалог со всей историей и контекстом. Полезно, когда вы закрыли терминал и хотите продолжить работу.

Возобновление конкретного диалога

```
claude -resume
```

Откроет список предыдущих диалогов с описаниями, из которого вы можете выбрать нужный.

Организация диалогов

Для организации работы рекомендуется:

– Один диалог – одна задача (фича, баг, рефакторинг)
– Используйте `/clear` перед началом совершенно новой задачи
– Используйте `--continue` для продолжения незавершённой работы
– Делайте `/compact` перед сохранением длинного диалога

3.12. Практическое упражнение: от нуля до деплоя

Закрепим полученные знания комплексным упражнением. Мы создадим, настроим и подготовим к деплою веб-приложение за один диалог.

Задание

Создайте приложение «Книжная полка» – каталог книг с возможностью добавления, редактирования, удаления, поиска и фильтрации по жанрам.

Рекомендуемая последовательность промптов

```
    1. > Создай приложение "Книжная полка" на Express.js + SQLite
+ EJS.
    Модель книги: название, автор, жанр, год, рейтинг, обложка
(URL).
    CRUD для книг, поиск, фильтрация по жанру.
    Красивый, современный интерфейс.
    2. > Запусти приложение и проверь, что всё работает.
    3. > Добавь 10 тестовых книг из русской классики с реальными
данными.
    4. > Добейся, чтобы приложение красиво выглядело на мобильных
устройствах.
    Используй responsive дизайн.
    5. > Добавь пагинацию: по 6 книг на страницу.
    6. > Напиши тесты для API-эндпоинтов.
    7. > Создай Dockerfile для деплоя.
    8. > Напиши README.md с инструкциями по запуску и деплою.
```

После выполнения этого упражнения у вас будет полноценное приложение, готовое к развёртыванию, – и глубокое понимание рабочего процесса с Claude Code.

Итоги главы

В этой главе мы:

– Освоили запуск Claude Code и интерфейс командной строки
– Изучили режимы работы: интерактивный, однократный, продолжение и pipe
– Поняли анатомию диалога: от запроса через планирование и выполнение к итогам
– Разобрали базовые слэш-команды: /help, /clear, /compact, /status, /cost
– Научились управлять контекстным окном для оптимальной работы
– Создали первый проект – менеджер заметок
– Освоили работу с файлами: чтение, создание, редактирование, удаление
– Разобрали работу с терминалом и систему разрешений
– Изучили навигацию по проекту: поиск файлов, зависимостей, структуры
– Узнали о паттернах эффективного взаимодействия
– Научились управлять сессиями
– Выполнили практическое упражнение: приложение от нуля до деплоя

В следующей главе мы погрузимся глубже – в архитектуру и принципы работы Claude Code: как он понимает код, какие инструменты использует, и как устроена его система безопасности.

Глава 4. Архитектура и принципы работы

4.1. Как Claude Code понимает вашу кодовую базу

Когда вы запускаете Claude Code в каталоге проекта, происходит нечто большее, чем простое чтение файлов. Claude Code использует многоуровневую систему для построения понимания вашего проекта. Разберём каждый уровень этой системы.

Уровень 1: Метаданные проекта

Первое, что делает Claude Code при старте – анализирует метаданные проекта. Это файлы, которые описывают проект на высоком уровне:

– **package.json / pyproject.toml / go.mod / Cargo.toml** – определяет язык, зависимости, скрипты
– **README.md** – описание проекта, инструкции по запуску
– **CLAUDE.md** – специальные инструкции для AI-ассистента
– **.gitignore** – какие файлы исключены из VCS (и, следовательно, менее важны)
– **Dockerfile / docker-compose.yml** – инфраструктура
– **tsconfig.json / eslint.config.js / ruff.toml** – конфигурация инструментов

Эти файлы дают Claude Code «карту местности» – общее представление о технологическом стеке, структуре и конвенциях проекта. Без них Claude Code всё равно может работать, но с ними – значительно эффективнее.

Уровень 2: Структура файловой системы

Claude Code анализирует иерархию каталогов проекта. Структура каталогов – это, по сути, архитектурная диаграмма проекта. Например, структура:

```
src/
+- controllers/
+- models/
+- services/
+- repositories/
+- middleware/
+- utils/
+- config/
```

Немедленно говорит Claude Code, что перед ним MVC-приложение с чёткой слоистой архитектурой. Это влияет на то, куда он поместит новый код и как будет именовать файлы.

Другой пример:

```
src/
+- features/
| +- auth/
| | +- components/
| | +- hooks/
| | +- api/
| | +- types/
| +- dashboard/
| +- settings/
+- shared/
```

Здесь Claude Code распознаёт feature-based архитектуру и будет организовывать новый код по тому же принципу – группируя по функциональности, а не по типу файла.

Уровень 3: Индексация кода

При необходимости Claude Code просматривает содержимое файлов, строя ментальную модель кодовой базы. Он не загружает все файлы в контекст одновременно (это было бы слишком затратно), а использует стратегию «ленивой загрузки»:

– **Сначала** загружаются метаданные и структура
– **По запросу** загружаются конкретные файлы, релевантные текущей задаче
– **При поиске** используются инструменты вроде `grep` и `find` для локализации нужного кода
– **Перекрёстные ссылки** отслеживаются через анализ импортов и экспортов

Этот подход позволяет работать с проектами любого размера. Вам не нужен проект из десяти файлов – Claude Code успешно работает с монорепозиториями, содержащими тысячи файлов.

Уровень 4: Семантическое понимание

На этом уровне Claude Code использует возможности языковой модели для семантического понимания кода. Это включает:

– **Распознавание паттернов проектирования:** Singleton, Factory, Observer, Repository, Decorator и другие
– **Понимание бизнес-логики:** Claude Code не просто видит функцию `calculateTax`, а понимает, что она связана с налоговыми расчётами, и может предложить добавить обработку различных налоговых юрисдикций
– **Анализ потока данных:** как данные проходят через систему – от входа (API-запрос) через обработку (сервисный слой) до хранения (база данных)
– **Выявление потенциальных проблем:** race conditions, утечки памяти, SQL-инъекции, небезопасное обращение с секретами

4.2. Инструменты (Tools): как Claude Code действует

В основе агентных возможностей Claude Code лежит система инструментов (tools). Каждый инструмент – это атомарное действие, которое Claude Code может выполнить во внешней среде. Модель сама решает, какие инструменты вызвать, в каком порядке и с какими параметрами.

Файловые инструменты

ReadFile – чтение содержимого файла:

```
Input: path: "src/utils/helpers.ts"
Output: содержимое файла (текст)
```

Claude Code использует этот инструмент для загрузки файлов в контекст. При чтении больших файлов он может запросить только определённый диапазон строк.

WriteFile – создание или перезапись файла:

```
Input: path: "src/models/user.ts"
content: "export interface User { … }"
Output: файл создан/обновлён
```

EditFile (MultiEdit) – точечное редактирование файла:

```
Input: path: "src/server.ts"
edits: [
{ oldText: "port: 3000", newText: "port: config.port" },
{    oldText:    "//    TODO:    add    auth",    newText:
"app.use(authMiddleware)" }
]
Output: файл обновлён (только указанные фрагменты)
```

Отчие инструмента EditFile от WriteFile – он вносит минимальные изменения, не перезаписывая весь файл. Это снижает риск потери данных и делает изменения более предсказуемыми.

Терминальные инструменты

Bash – выполнение команд оболочки:

```
Input: command: "npm test"
Output: stdout + stderr + exit code
```

Это «швейцарский нож» Claude Code. Через Bash он может:

– Устанавливать зависимости
– Запускать тесты
– Компилировать код
– Управлять Git
– Работать с Docker
– Запускать линтеры и форматтеры
– Анализировать систему

Поисковые инструменты

Search (grep/ripgrep) – поиск по содержимому файлов:

```
Input: query: "function calculateDiscount"
path: "src/"
Output: список файлов с совпадениями и контекстом
```

Find (fd) – поиск файлов по имени:

```
Input: pattern: "*.test.ts"
path: "src/"
Output: список путей к файлам
```

Веб-инструменты

WebSearch – поиск в интернете для получения актуальной информации. Этот инструмент помогает Claude Code находить документацию, решения проблем и актуальные версии библиотек.

ReadURL – чтение содержимого веб-страницы. Полезно для анализа API-документации, StackOverflow-ответов и README на GitHub.

MCP-инструменты

Model Context Protocol (MCP) позволяет подключать произвольные внешние инструменты. Подробнее о MCP – в Главе 13. Примеры:

– **Jira** – чтение и обновление задач
– **Slack** – отправка сообщений
– **GitHub** – управление пулл-реквестами
– **Database** – выполнение SQL-запросов
– **Kubernetes** – управление кластером

Координация инструментов

Ключевая способность Claude Code – не просто вызывать отдельные инструменты, а координировать их в последовательность для достижения комплексной цели. Рассмотрим пример:

Задача: «Исправь падающий тест test_user_creation»

Последовательность действий Claude Code:

```
1. Bash("npm test - test_user_creation")
-> Получим stack trace ошибки
2. ReadFile("tests/test_user_creation.py")
-> Прочитаем тест, чтобы понять, что он проверяет
3. ReadFile("src/services/user_service.py")
-> Прочитаем код, который тестируется
4. Search("UserRepository.create", "src/")
-> Найдём реализацию метода create в репозитории
5. ReadFile("src/repositories/user_repository.py")
-> Прочитаем репозиторий
6. EditFile("src/services/user_service.py", [
{ oldText: "user.save()",
newText: "await user_repository.create(user)" }
])
-> Исправим ошибку (вызывался deprecated метод)
7. Bash("npm test - test_user_creation")
-> Проверим, что тест теперь проходит
```

Каждый следующий шаг зависит от результатов предыдущего. Claude Code анализирует вывод каждого инструмента и адаптирует план.

4.3. Модель разрешений и безопасность

Безопасность – одна из ключевых характеристик Claude Code. Поскольку инструмент имеет доступ к файловой системе и терминалу, необходимы надёжные механизмы контроля.

Иерархия разрешений

Claude Code использует многоуровневую систему разрешений:

Уровень 1: Встроенные ограничения модели

Сама модель Claude обучена отказываться от потенциально вредоносных действий. Она не будет:

– Удалять системные файлы
– Устанавливать вредоносное ПО
– Отправлять конфиденциальные данные на внешние серверы
– Выполнять команды, которые могут повредить операционную систему

Уровень 2: Категории инструментов

Каждый инструмент имеет категорию безопасности:

Категория | Примеры | Требует подтверждения
Read-only | ReadFile, Search, Find | Нет (по умолчанию)
Write | WriteFile, EditFile | Да (можно настроить)
Execute | Bash | Да (можно настроить)
Network | WebSearch, ReadURL | Да (можно настроить)
Destructive | Bash(rm, sudo) | Всегда

Уровень 3: Настраиваемые разрешения

Через файл `.claude/settings.json` вы можете точно определить, что разрешено:

```
{
"permissions": {
"allow": [
"Read",
"Write(src/**)",
"Write(tests/**)",
"Bash(npm run *)",
"Bash(pytest *)",
"Bash(git add *)",
"Bash(git commit *)"
],
"deny": [
"Write(*.env)",
"Write(.claude/settings.json)",
"Bash(rm -rf *)",
"Bash(sudo *)",
"Bash(curl * | *)"
]
}
}
```

В этом примере:

– Чтение доступно для всех файлов
– Запись ограничена каталогами `src/` и `tests/`
– Разрешены конкретные bash-команды
– Запрещены опасные операции и модификация секретов

Уровень 4: Интерактивное подтверждение

Для действий, не покрытых настройками, Claude Code запрашивает подтверждение в реальном времени:

```
Claude Code wants to execute:
pip install requests==2.31.0
Options:
[Y] Allow once
[n] Deny
[a] Always allow 'pip install' commands
[d] Always deny 'pip install' commands
```

Выбор «Always allow» сохраняется в `.claude/settings.json` для текущего проекта.

Аудит действий

Все действия Claude Code можно просмотреть через историю сессий. Каждая операция логируется:

```
[2026-02-16 10:15:23] Tool: ReadFile(src/server.ts) -> OK
[2026-02-16 10:15:25] Tool: Bash(npm test) -> EXIT 1
[2026-02-16 10:15:28] Tool: ReadFile(tests/app.test.ts) -> OK
[2026-02-16 10:15:31] Tool: EditFile(src/server.ts) -> OK (2
edits)
[2026-02-16 10:15:34] Tool: Bash(npm test) -> EXIT 0
```

Это позволяет отследить все изменения, внесённые Claude Code, и при необходимости откатить их через Git.

Работа с секретами

Claude Code категорически не должен добавлять секреты (API-ключи, пароли, токены) в файлы, которые могут попасть в Git. Вот рекомендуемые практики:

– **Используйте переменные окружения.** Claude Code понимает файлы `.env` и будет использовать `process.env.API_KEY` вместо хардкода ключей.

– **Добавьте `.env` в `.gitignore`.** Убедитесь, что файлы с секретами не попадут в репозиторий.

– **Запретите запись в .env.** В настройках разрешений добавьте `"deny": ["Write(*.env)"]`.

– **Используйте .env.example.** Просите Claude Code создавать файлы-шаблоны без реальных значений.

4.4. Агентный режим: планирование и выполнение

Термин «агентный режим» описывает способность Claude Code самостоятельно планировать и выполнять многоэтапные задачи. Это отличает его от простых чат-ботов, которые лишь отвечают на вопросы.

Цикл AGI (Act-Grounded-Iterate)

Внутренний цикл работы Claude Code можно описать так:

```
1. THINK (Подумать)
- Проанализировать запрос пользователя
- Определить необходимые шаги
- Оценить, какая информация нужна
2. ACT (Действовать)
- Вызвать нужный инструмент
- Выполнить операцию
3. OBSERVE (Наблюдать)
- Проанализировать результат
- Есть ли ошибки?
- Достигнута ли цель?
4. ITERATE (Итерировать)
- Если цель достигнута -> вернуть результат
- Если нет -> скорректировать план и вернуться к шагу 1
```

Этот цикл может выполняться десятки раз в рамках одной задачи. Ключевое отличие от «линейной» генерации – Claude Code адаптируется к результатам каждого шага.

Пример агентного выполнения

Рассмотрим реальный сценарий:

```
> Добавь систему логирования в проект. Используй winston.
Все модули должны использовать единый логгер.
Настрой ротацию логов и формат JSON для продакшена.
```

У Claude Code появляется следующий внутренний план:

```
Step 1: Проанализировать текущую структуру проекта
Step 2: Установить winston и winston-daily-rotate-file
Step 3: Создать модуль логгера (src/utils/logger.ts)
Step 4: Настроить разные транспорты для dev и prod
Step 5: Заменить console.log на logger во всех файлах
Step 6: Обновить .env.example с новыми переменными
Step 7: Проверить работоспособность
```

На каждом шаге Claude Code может столкнуться с неожиданностями:

– Шаг 1: обнаружить, что в проекте уже используется pino -> скорректировать план
– Шаг 2: ошибка зависимостей -> добавить `--legacy-peer-deps`
– Шаг 5: обнаружить 50 файлов с console.log -> предложить постепенную миграцию
– Шаг 7: тесты падают -> исправить моки для логгера

Каждая неожиданность обрабатывается внутри цикла без дополнительного вашего вмешательства.

Ограничение количества шагов

По умолчанию Claude Code ограничивает количество автоматических шагов (maxTurns). Это защита от «бесконечных циклов», когда AI пытается исправить одну и ту же ошибку снова и снова.

Настройка:

```
// .claude/settings.json
{
"maxTurns": 50 // Максимум 50 автоматических шагов
}
```

Если лимит достигнут, Claude Code остановится и попросит ваших инструкций.

Режим расширенного планирования

Для особо сложных задач можно явно попросить Claude Code потратить больше времени на планирование:

```
> Подумай внимательно и составь детальный план перед началом работы.
Мне нужно мигрировать REST API на GraphQL. Проект использует
Express + PostgreSQL + Sequelize.
```

Слова «подумай внимательно» активируют режим расширенного размышления – Claude Code потратит больше токенов на анализ и планирование, прежде чем начнёт действовать.

4.5. Модели Claude: Opus, Sonnet, Haiku

Claude Code поддерживает несколько моделей семейства Claude, каждая из которых имеет свои характеристики.

Claude Opus

Характеристики:

– Самая мощная модель в линейке
– Лучшее качество рассуждения и генерации кода
– Самая высокая стоимость за токен
– Контекстное окно: 200K токенов

Когда использовать:

– Архитектурное проектирование
– Сложный рефакторинг legacy-кода
– Задачи, требующие глубокого анализа
– Работа с неизвестными фреймворками или языками
– Code review критически важного кода

Claude Sonnet

Характеристики:

– Оптимальный баланс качества и стоимости
– Модель по умолчанию для Claude Code
– Контекстное окно: 200K токенов
– Высокая скорость генерации

Когда использовать:

– Повседневная разработка
– Написание нового кода
– Отладка стандартных ошибок
– Генерация тестов
– Рефакторинг
– Работа с документацией

Claude Haiku

Характеристики:

– Самая быстрая и дешёвая модель
– Контекстное окно: 200K токенов
– Подходит для простых задач

Когда использовать:

– Простое автодополнение
– Генерация boilerplate-кода
– Ответы на простые вопросы
– Форматирование и линтинг
– Массовые операции, где стоимость критична

Переключение моделей

```
> /model claude-opus-4-20250514
Или в .claude/settings.json:
{
"defaultModel": "claude-sonnet-4-20250514"
}
```

Практический совет: начинайте работу с Sonnet. Если результат недостаточно качественный – переключитесь на Opus для конкретной задачи,затем вернитесь к Sonnet. Haiku используйте для автоматизации и массовых операций, где стоимость важнее качества.

4.6. Контекстный протокол: что знает Claude Code

При каждом обращении к модели Claude Code формирует контекст – совокупность информации, которую модель «видит». Понимание структуры контекста помогает оптимизировать работу.

Структура контекста

```
+-+
| Системный промпт (Instructions) |
| - Кто ты (Claude Code) |
| - Доступные инструменты |
| - Правила безопасности |
| - Форматирование ответов |
+-+
| CLAUDE.md (Project instructions) |
| - Описание проекта |
| - Архитектурные решения |
| - Конвенции кода |
| - Команды разработки |
+-+
| История диалога |
| - Предыдущие сообщения пользователя |
| - Ответы Claude Code |
| - Результаты вызовов инструментов |
+-+
| Текущий запрос |
```

```
| - Ваш последний промпт |
+-+
```

Приоритеты контекста

Не вся информация в контексте имеет одинаковый вес:

– **Высший приоритет:** текущий запрос и системные инструкции
– **Высокий приоритет:** CLAUDE.md, последние файлы, последние результаты
– **Средний приоритет:** ранние части диалога, ранее прочитанные файлы
– **Низкий приоритет:** устаревшая информация из начала длинного диалога

При сжатии контекста (`/compact`) информация с низким приоритетом удаляется первой.

Оптимизация контекста

Несколько стратегий для максимально эффективного использования контекста:

1. Предоставляйте релевантную информацию рано

```
> В нашем проекте используется монорепозиторий с Turborepo.
Бэкенд на NestJS (apps/api/), фронтенд на Next.js (apps/web/).
Общие пакеты в packages/shared/.
Добавь валидацию email в пакет shared и используй её в API.
> Добавь валидацию email.
```

2. Ссылайтесь на конкретные файлы

```
> В файле src/services/payment.ts функция processPayment
не обрабатывает ошибку от Stripe API. Добавь try-catch.
> Где-то есть баг с оплатой, исправь.
```

3. Используйте CLAUDE.md вместо повторений

Если вы обнаружили, что повторяете одни и те же инструкции в каждом диалоге – добавьте их в CLAUDE.md:

```
## Правила оформления кода
- Все строки не длиннее 100 символов
- Используй single quotes в JavaScript
- Каждый публичный метод должен иметь JSDoc
- Все ошибки должны наследоваться от AppError
```

4.7. Кэширование промптов

Anthropic поддерживает кэширование промптов (Prompt Caching) – механизм, который снижает стоимость повторных запросов с похожим контекстом.

Как работает кэширование

Когда Claude Code отправляет запрос к API, неизменная часть контекста (системный промпт, CLAUDE.md, прочитанные файлы) может быть кэширована. При следующем запросе, если эта часть не изменилась, Anthropic не будет тарифицировать её по полной стоимости.

Практический эффект: если вы ведёте долгий диалог, каждый новый запрос обходится значительно дешевле, чем первый, потому что большая часть контекста уже кэширована.

Как максимизировать кэширование

– **Не меняйте CLAUDE.md часто.** Каждое изменение инвалидирует кэш.
– **Работайте с одним набором файлов.** Если Claude Code прочитал 5 файлов, следующий запрос к тем же файлам будет дешевле.
– **Не используйте /clear без необходимости.** Сброс контекста инвалидирует весь кэш.
– **Предпочитайте /compact вместо /clear.** Сжатие сохраняет достаточно контекста для кэширования.

4.8. Границы возможностей

Понимание границ – ключ к эффективному использованию любого инструмента. Вот что Claude Code не может (или не должен) делать:

Технические ограничения

– **Нет постоянной памяти между сессиями.** Каждый новый диалог начинается с чистого листа (кроме CLAUDE.md и конфигурации). Claude Code не «помнит», что вы обсуждали вчера.

– **Нет фонового выполнения.** Claude Code не работает, пока вы не активны. Он не может мониторить систему или реагировать на события автономно.

– **Нет прямого доступа к GUI.** Claude Code работает в терминале. Он не может кликать по кнопкам в браузере или IDE (хотя инструмент Browser позволяет навигировать по веб-страницам).

– **Ограничение длительности команд.** Команды bash имеют таймаут. Claude Code не может запустить процесс на 10 часов и ждать результат.

Концептуальные ограничения

– Claude Code не понимает бизнес-контекст по умолчанию. Он не знает вашей предметной области, если вы не объясните. Медицинское ПО, финтех, игры – всё требует разного подхода.

– Claude Code не заменяет тестирование. Сгенерированный код может содержать ошибки. Автоматические тесты и ручная проверка остаются необходимыми.

– Claude Code не гарантирует безопасность. Он может написать код без очевидных уязвимостей, но не является сертифицированным инструментом безопасности. Для критических систем необходим аудит безопасности.

– Claude Code не оптимизируется автоматически. Он напишет рабочий код, но не обязательно самый эффективный. Для высоконагруженных систем требуется профилирование и оптимизация.

4.9. Архитектура расширяемости

Claude Code спроектирован с учётом расширяемости. Вот основные механизмы:

Пользовательские команды (Slash Commands)

Вы можете создавать собственные слэш-команды:

```
<!- .claude/commands/review.md ->
Проведи code review файла $ARGUMENTS:
1. Проверь code style
2. Найди потенциальные баги
3. Оцени производительность
4. Предложи улучшения
Используй формат: файл:строка – описание
```

Использование:

```
> /project:review src/services/payment.ts
```

Hooks

Hooks позволяют выполнять действия до или после вызовов инструментов:

```
// .claude/settings.json
{
```

```
"hooks": {
"preToolUse": [
{
"tool": "Write",
"command": "echo 'Файл будет изменён: $FILE_PATH'"
}
],
"postToolUse": [
{
"tool": "Bash",
"command": "echo 'Команда выполнена: $COMMAND'"
}
]
}
}
```

MCP (Model Context Protocol)

MCP – это протокол для подключения внешних инструментов и источников данных. Подробнее – в Главе 13.

Итоги главы

В этой главе мы рассмотрели:

– Как Claude Code понимает кодовую базу: от метаданных до семантического анализа
– Систему инструментов: файловые, терминальные, поисковые, веб и MCP
– Координацию инструментов: как Claude Code строит цепочки действий
– Модель разрешений и безопасность: четыре уровня защиты
– Агентный режим: цикл Think-Act-Observe-Iterate
– Модели Claude: Opus, Sonnet, Haiku – выбор для каждой задачи
– Контекстный протокол и оптимизацию контекста
– Кэширование промптов для экономии
– Границы возможностей: что Claude Code может и не может
– Архитектуру расширяемости: команды, hooks, MCP

В следующей главе мы сосредоточимся на искусстве эффективного общения с AI-ассистентом – навыке, который определяет разницу между посредственным и выдающимся использованием Claude Code.

Глава 5. Эффективное общение с AI-ассистентом

5.1. Искусство промпт-инжиниринга для разработки

Промпт-инжиниринг – это не просто модное словосочетание, а практический навык, который определяет качество результатов при работе с любым языковым моделям, включая Claude Code. Разница между хорошим и плохим промптом может быть разницей между рабочим решением за пять минут и бесконечным циклом исправлений.

В контексте разработки промпт-инжиниринг приобретает специфические черты. Вы общаетесь не с универсальным чат-ботом, а с агентом, который способен выполнять реальные действия: создавать файлы, запускать тесты, устанавливать зависимости. Следовательно, ваш промпт – это не просто вопрос, а задание для исполнителя. В нём важны точность, полнота и однозначность.

Рассмотрим три уровня зрелости промптов:

Уровень 1: Наивный промпт

```
> Сделай логин
```

Этот промпт слишком расплывчат. Какой логин? Для веб-приложения или мобильного? По email или через OAuth? Какой стек? Какие требования к безопасности? Claude Code попытается угадать – и, скорее всего, угадает неоптимально.

Уровень 2: Структурированный промпт

```
> Реализуй страницу входа для нашего React-приложения.
Форма: email + пароль.
Валидация: email формат, минимум 8 символов в пароле.
При успешном входе – редирект на /dashboard.
При ошибке – показать сообщение.
Используй наш AuthService из src/services/auth.ts.
```

Лучше: есть конкретика, контекст и ограничения. Но можно ещё детальнее.

Уровень 3: Экспертный промпт

```
> Реализуй страницу входа (src/pages/LoginPage.tsx).
## Требования к функциональности
- Форма с полями email и password
- Клиентская валидация: email (RFC 5322), пароль (8 символов,
```

```
    хотя бы 1 цифра и 1 заглавная буква)
    - Кнопка "Показать пароль" (toggle visibility)
    - Ссылка "Забыли пароль?" -> /forgot-password
    - Ссылка "Регистрация" -> /register
    - При успехе: сохранить токен в AuthContext,
    редирект на returnUrl или /dashboard
    - При ошибке 401: "Неверный email или пароль"
    - При ошибке 429: "Слишком много попыток, повторите через N
минут"
    - При сетевой ошибке: "Ошибка связи, попробуйте позже"
    ## Технические требования
    - Использовать react-hook-form + zod для валидации
    - Вызывать AuthService.login() из src/services/auth.ts
    - Стили: использовать наш UI Kit из src/components/ui/
    - Добавить тест: src/pages/__tests__/LoginPage.test.tsx
    - Тест должен покрывать: успешный вход, ошибку 401, валидацию
    ## Ограничения
    - Не менять AuthService - он уже протестирован
    - Не добавлять новых зависимостей
    - Следовать стилю существующих страниц (посмотри
RegisterPage.tsx)
```

Этот промпт не оставляет места для интерпретаций. Claude Code получает всю необходимую информацию для качественной реализации. Обратите внимание: экспертный промпт длиннее, но экономит время в целом, потому что результат получается правильным с первой попытки.

5.2. Принципы эффективных промптов

Принцип конкретности

Каждый элемент промпта должен быть максимально конкретным.

```
    > Сделай красивый дизайн
    > Используй цветовую схему: primary #6366f1, background
#0f172a.
    Карточки с rounded-2xl и тенью shadow-xl.
    Шрифт: Inter для текста, JetBrains Mono для кода.
    Анимация при наведении: scale(1.02) transition 200ms.
```

Принцип контекста

Предоставляйте релевантный контекст, но не перегружайте.

```
    > Исправь баг
```

```
> Наша компания была основана в 2015 году, мы занимаемся…
[500 слов нерелевантной информации]
…исправь баг в функции парсинга
> Функция parseCSV в src/utils/parser.ts падает при обработке
файлов с кириллицей. Ошибка: "Invalid byte sequence in UTF-8".
Файл загружается через multer, кодировка – windows-1251.
```

Принцип разделения

Разбивайте сложные задачи на этапы.

```
> Создай полноценный интернет-магазин с каталогом, корзиной,
оплатой, личным кабинетом, админкой и аналитикой
> Этап 1: Создай базовую структуру проекта и модель Product
> (после выполнения)
> Этап 2: Реализуй каталог товаров с фильтрацией
> (после выполнения)
> Этап 3: Добавь корзину покупок
```

Принцип примеров

Показывайте примеры ожидаемого результата.

```
> Напиши функцию для форматирования цены.
Примеры:
formatPrice(1000) -> "1 000 "
formatPrice(1500.5) -> "1 500,50 "
formatPrice(0) -> "0 "
formatPrice(1000000) -> "1 000 000 "
formatPrice(99.9) -> "99,90 "
```

Принцип ролевой модели

Задание роли помогает Claude Code настроить стиль и уровень ответа.

```
> Выступи как старший DevOps-инженер и проведи аудит
нашего Dockerfile. Обрати внимание на:
- Размер образа
- Безопасность (непривилегированный пользователь)
- Кэширование слоёв
- Мультистейдж-сборка
- Healthcheck
```

5.3. Типы промптов для разных задач

Промпты для генерации кода

```
> Создай [что] для [цель] используя [технологии].
Входные данные: [описание]
Выходные данные: [описание]
Граничные случаи: [описание]
Обработка ошибок: [описание]
```

Пример:

```
> Создай middleware для rate limiting HTTP-запросов
используя Redis.
Входные данные: IP-адрес и путь запроса
Выходные данные: пропуск запроса или HTTP 429
Конфигурация: 100 запросов в минуту на IP,
отдельные лимиты для /api/auth (10 req/min)
Граничные случаи: Redis недоступен -> пропускать запросы
Обработка ошибок: логировать ошибки Redis,
не падать при недоступности
```

Промпты для отладки

```
> Ошибка: [описание ошибки]
Файл: [путь к файлу]
Воспроизведение: [шаги]
Ожидаемое поведение: [что должно быть]
Фактическое поведение: [что происходит]
Уже пробовал: [что не помогло]
```

Пример:

```
> Ошибка: WebSocket-соединение разрывается через 30 секунд
Файл: src/websocket/server.ts
Воспроизведение:
1. Открыть ws://localhost:3001
2. Подождать 30 секунд
3. Соединение закрывается с кодом 1006
Ожидаемое: соединение должно жить неограниченно
Фактическое: disconnect через ~30 секунд
Пробовал: увеличить pingTimeout до 60000 – не помогло
```

Промпты для рефакторинга

```
> Отрефактори [файл/модуль]:
Цели:
- [конкретная цель 1]
- [конкретная цель 2]
Ограничения:
- [что менять нельзя]
- [обратная совместимость]
Паттерн: [если есть конкретный паттерн]
```

Пример:

```
> Отрефактори src/services/order.ts:
Цели:
- Разделить на мелкие модули (сейчас 800 строк)
- Извлечь расчёт стоимости в PricingService
- Извлечь отправку уведомлений в NotificationService
- Извлечь валидацию в OrderValidator
Ограничения:
- Не менять публичный API (OrderService.createOrder/
cancelOrder)
- Не менять типы из types/order.ts
- Тесты должны продолжать проходить
Паттерн: Dependency Injection через конструктор
```

Промпты для code review

```
> Проведи code review [файл/PR]:
Фокус:
- [конкретные аспекты]
Контекст:
- [что делает этот код]
- [почему так сделано]
Формат ответа:
- [как ты хочешь получить результат]
```

Пример:

```
> Проведи code review файла src/api/payments.ts
Фокус:
- Безопасность (обработка платёжных данных)
- Обработка ошибок (Stripe API может быть недоступен)
- Идемпотентность (повторные запросы)
Контекст:
- Это эндпоинт для обработки платежей через Stripe
- Вызывается из мобильного приложения
```

```
    - Должен быть идемпотентным (используем idempotency key)
    Формат:
    - Для каждой проблемы: строка, серьёзность (critical/warning/
info),
    описание, предложение исправления
```

Промпты для документирования

```
> Документируй [файл/модуль/API]:
Аудитория: [кто будет читать]
Формат: [JSDoc/Docstring/README/ADR]
Включить: [что должно быть]
Исключить: [чего не нужно]
```

5.4. CLAUDE.md: память проекта

Файл CLAUDE.md – это, пожалуй, самый недооценённый инструмент в арсенале Claude Code. Давайте рассмотрим его возможности подробно.

Зачем нужен CLAUDE.md

CLAUDE.md решает фундаментальную проблему: контекст теряется между сессиями. Каждый новый диалог с Claude Code начинается с чистого листа. Без CLAUDE.md вам придётся каждый раз объяснять:

– Какой стек используется
– Как организован проект
– Какие команды для запуска, тестирования, деплоя
– Какие соглашения по коду приняты
– Какие архитектурные решения были приняты и почему

CLAUDE.md хранит всё это в одном месте и загружается автоматически при каждом запуске.

Структура эффективного CLAUDE.md

Вот шаблон, который можно адаптировать под любой проект:

```
## Описание
[1-2 предложения о том, что это за проект]
## Технологический стек
- Backend: [язык, фреймворк, версии]
- Frontend: [фреймворк, версии]
- Database: [СУБД, ORM]
```

```
- Infrastructure: [Docker, K8s, облако]
## Архитектура
[Описание архитектуры: слои, сервисы, основные компоненты]
### Структура каталогов
[Описание ключевых каталогов и что в них находится]
## Команды разработки
- Запуск: `[команда]`
- Тесты: `[команда]`
- Линтинг: `[команда]`
- Сборка: `[команда]`
- Миграции: `[команда]`
## Соглашения по коду
[Стандарты именования, форматирования, организации кода]
## Важные решения
[Ключевые архитектурные решения и их обоснования]
## Чего НЕ делать
[Антипаттерны, запрещённые подходы, known pitfalls]
```

Продвинутый пример CLAUDE.md

```
## Описание
SaaS-приложение для управления задачами в стиле Kanban.
Мультитенантное, поддерживает команды до 50 человек.
## Стек
- Frontend: Next.js 14 (App Router), TypeScript 5.4,
Tailwind CSS 3.4, React Query, Zustand
- Backend: Next.js API Routes + tRPC
- DB: PostgreSQL 16 + Prisma ORM
- Auth: NextAuth.js v5 (Google, GitHub OAuth)
- Realtime: Pusher
- Hosting: Vercel + Supabase (DB)
- CI/CD: GitHub Actions
## Архитектура
### Frontend (src/app/)
- `(auth)/` - страницы авторизации (layout без sidebar)
- `(dashboard)/` - основное приложение (layout с sidebar)
- `boards/[boardId]/` - доска с колонками и карточками
- `settings/` - настройки аккаунта и команды
- `api/` - API routes (через tRPC router)
### Backend (src/server/)
- `routers/` - tRPC роутеры (board, card, user, team)
- `services/` - бизнес-логика
- `db/` - Prisma client и seed-данные
### Shared (src/lib/)
- `utils/` - утилитарные функции
- `types/` - TypeScript типы и Zod-схемы
- `hooks/` - React-хуки
```

```
    - `components/ui/` - базовый UI Kit (Button, Input, Modal,
etc.)
    ## Команды разработки
    - `pnpm dev` - запуск dev-сервера (localhost:3000)
    - `pnpm test` - запуск тестов (Vitest)
    - `pnpm test:e2e` - E2E тесты (Playwright)
    - `pnpm lint` - ESLint + Prettier проверка
    - `pnpm lint:fix` - автоисправление
    - `pnpm db:push` - применить схему Prisma
    - `pnpm db:migrate` - создать миграцию
    - `pnpm db:seed` - заполнить тестовыми данными
    - `pnpm db:studio` - открыть Prisma Studio
    ## Соглашения
    ### Именование
    - Компоненты: PascalCase (BoardCard.tsx)
    - Хуки: camelCase с prefixом use (useBoard.ts)
    - Утилиты: camelCase (formatDate.ts)
    - Типы: PascalCase с суффиксом (BoardInput, CardOutput)
    - tRPC роутеры: camelCase (boardRouter)
    - Prisma модели: PascalCase singular (Board, Card, User)
    ### Организация кода
    - Один компонент - один файл (с co-located стилями)
    - Бизнес-логика в services, НЕ в роутерах
    - Валидация через Zod-схемы (НЕ вручную)
    - Ошибки через собственный TRPCError (НЕ throw new Error)
    - Все запросы к БД через Prisma (НЕ raw SQL)
    ### Git
    - Формат коммитов: conventional commits (feat:, fix:, chore:,
etc.)
    - Ветки: feature/*, bugfix/*, hotfix/*
    - PR требует 1 approve
    ## Важные решения
    1. tRPC вместо REST - type safety между клиентом и сервером
    2. App Router вместо Pages - для улучшенной серверной рендеринг
    3. Zustand вместо Redux - простота для нашего масштаба
    4. Pusher вместо WebSocket - managed service, не нужно думать
о масштабировании
    ## Чего НЕ делать
    - НЕ использовать `any` в TypeScript (используй `unknown` +
type guard)
    - НЕ хранить секреты в коде (только .env.local)
    - НЕ делать прямые запросы к Prisma из компонентов (через tRPC)
    - НЕ добавлять CSS в global.css (используй Tailwind)
    - НЕ коммитить .env.local
    - НЕ обращаться к Prisma client напрямую - через service layer
```

Иерархические CLAUDE.md

Claude Code поддерживает CLAUDE.md на разных уровнях вложенности:

```
project/
+- CLAUDE.md # Глобальные инструкции
+- packages/
| +- api/
| | +- CLAUDE.md # Инструкции для API
| +- web/
| +- CLAUDE.md # Инструкции для фронтенда
+- scripts/
+- CLAUDE.md # Инструкции для скриптов
```

Локальный CLAUDE.md дополняет (не заменяет) глобальный. Это позволяет иметь общие правила для всего проекта и специфические – для каждого пакета.

5.5. Антипаттерны: чего не стоит делать

Опыт сообщества выявил ряд антипаттернов – типичных ошибок, которые снижают эффективность работы с Claude Code.

Антипаттерн 1: «Рыхлый промпт»

```
> Сделай что-нибудь с кодом, он какой-то медленный
```

Антипаттерн 2: «Микроменеджмент»

```
> Открой файл server.js. Теперь найди строку 42.
Замени "const" на "let". Сохрани.
> В server.js переменная config на строке 42
используется в двух местах – внутри if и после if.
Это ошибка? Если да, исправь.
```

Антипаттерн 3: «Вся кодовая база за раз»

```
> Прочитай все файлы в проекте и скажи, что можно улучшить
> Какие проблемы с обработкой ошибок в каталоге src/api/?
```

Антипаттерн 4: «Слепое доверие»

Антипаттерн 5: «Бесконечные исправления»

```
> Исправь ошибку -> … -> Не работает, исправь -> … ->
Всё ещё не работает -> … -> (20-й раз)
```

Антипаттерн 6: «Переключение контекста»

```
> Давай добавим кеширование. А кстати, ещё нужно
поправить стили на странице логина. И вообще,
давай перейдём на TypeScript.
```

5.6. Итеративная разработка: уточнение и корректировка

Работа с Claude Code – это не one-shot генерация, а диалог. Даже с идеальным промптом первый результат часто нуждается в доработке. Вот как эффективно итерировать.

Техника «Да, но»

```
> [Первый промпт] Создай REST API для блога
Claude Code генерирует базовый API.
> Да, хорошо, но добавь пагинацию с cursor-based подходом
вместо offset-based
> Да, но ещё нужна фильтрация по тегам и автору
> Хорошо, теперь добавь кеширование ответов через Redis
```

Каждая итерация добавляет новый слой, сохраняя предыдущую работу.

Техника «Расширение горизонта»

Начните с MVP и постепенно расширяйте:

```
> 1. Создай простую форму обратной связи (имя, email, сообщение)
> 2. Добавь валидацию полей
> 3. Добавь защиту от спама (reCAPTCHA)
> 4. Добавь отправку уведомлений на email
> 5. Добавь сохранение в базу данных
> 6. Добавь админку для просмотра обращений
```

Техника «Критик»

Попросите Claude Code сам оценить свою работу:

```
> Проведи code review кода, который ты только что написал.
Найди потенциальные проблемы и предложи улучшения.
```

Это удивительно эффективный приём: модель часто находит недочёты в собственной генерации, которые не заметила при создании.

Техника «Сравнение»

```
> Реализуй этот модуль двумя способами:
1. С использованием callback-стиля
2. С использованием async/await
Сравни оба подхода по читаемости, производительности
и обработке ошибок. Какой рекомендуешь для нашего проекта?
```

5.7. Продвинутые техники промптинга

Chain-of-Thought (Цепочка рассуждений)

Попросите Claude Code рассуждать пошагово:

```
> Подумай пошагово: почему функция processOrder может
вызывать deadlock при конкурентных запросах?
1. Какие ресурсы она блокирует?
2. В каком порядке?
3. Может ли другой вызов заблокировать ресурсы в
обратном порядке?
4. Как это исправить?
```

Few-Shot Learning (Обучение на примерах)

Покажите несколько примеров желаемого формата:

```
> Напиши тесты для валидатора в таком формате:
describe('EmailValidator', () => {
it('should accept valid email', () => {
expect(validate('user@example.com')).toBe(true);
});
it('should reject email without @', () => {
expect(validate('userexample.com')).toBe(false);
```

```
});
});
Теперь напиши аналогичные тесты для PhoneValidator
с тем же стилем.
```

Negative Prompting (Негативные инструкции)

Иногда эффективнее сказать, чего НЕ делать:

```
> Реализуй кэширование результатов API.
НЕ используй:
- глобальные переменные для хранения кэша
- node-cache (слишком простой)
- файловую систему для кэша
Используй Redis через ioredis.
НЕ устанавливай никакие дополнительные пакеты
кроме ioredis.
```

Conditional Prompting (Условные инструкции)

```
> Проанализируй package.json этого проекта.
- Если используется Jest -> настрой test coverage
и добавь скрипт "test:coverage"
- Если используется Vitest -> аналогично,
но через vitest.config.ts
- Если тест-фреймворка нет -> предложи Vitest
и настрой с нуля
```

5.8. Работа с большими кодовыми базами

Большие проекты (100+ файлов, 50K+ строк) требуют особого подхода.

Стратегия «Фокусная линза»

Вместо работы со всей кодовой базой, фокусируйтесь на конкретной области:

```
> Сейчас мы работаем ТОЛЬКО с модулем оплаты.
Релевантные файлы:
- src/services/payment/
- src/api/routes/payment.ts
- src/models/payment.ts
- tests/payment/
Не трогай другие модули.
```

Стратегия «Проводник»

Используйте Claude Code для навигации по незнакомому проекту:

```
> Я новый разработчик на проекте. Объясни:
1. Как обрабатывается HTTP-запрос от входа до ответа?
2. Где определены маршруты?
3. Как работает аутентификация?
4. Как данные сохраняются в БД?
```

Стратегия «Карта зависимостей»

```
> Построй карту зависимостей для модуля UserService:
- Какие модули он импортирует?
- Кто импортирует его?
- Какие внешние сервисы вызывает?
- Какие таблицы БД использует?
```

5.9. Многоязычные промпты

Claude Code одинаково хорошо понимает промпты на разных языках. Однако есть нюансы:

Русский язык

Claude Code отлично понимает русский и может генерировать русскоязычные комментарии, документацию и пользовательские интерфейсы. Однако:

– Код (названия переменных, функций, классов) лучше писать на английском – это стандарт индустрии
– Комментарии можно писать на русском, если это принято в проекте
– README и документацию – на языке целевой аудитории

```
> Добавь комментарии на русском языке к модулю
src/services/billing.ts. Объясни бизнес-логику
расчёта стоимости для новых разработчиков.
```

Смешанные промпты

Вы можете свободно смешивать языки в одном промпте:

```
> Реализуй endpoint POST /api/invoices.
Request body:
```

```
    {
    "clientId": "uuid",
    "items": [{ "description": "string", "amount": "number" }]
    }
    Валидируй amount – должен быть положительным числом.
    При ошибке возвращай {"error": "Сумма должна быть положитель-
ной"}.
```

5.10. Создание собственных шаблонов промптов

Со временем вы обнаружите, что используете одни и те же типы промптов снова и снова. Создайте шаблоны и сохраните их как пользовательские команды.

Шаблон для создания компонентов

```
<!- .claude/commands/component.md ->
Создай React-компонент $ARGUMENTS:
1. Создай файл в src/components/
2. Используй TypeScript
3. Добавь props interface
4. Используй наш UI Kit из src/components/ui/
5. Добавь Storybook story
6. Добавь unit-тест
7. Экспортируй из src/components/index.ts
```

Шаблон для исправления бага

```
<!- .claude/commands/fix-bug.md ->
Исправь баг: $ARGUMENTS
Шаги:
1. Найди файл с ошибкой
2. Проанализируй причину
3. Предложи 2-3 варианта исправления
4. Реализуй лучший вариант
5. Напиши regression-тест
6. Запусти все тесты, чтобы убедиться в отсутствии регрессий
7. Создай коммит с сообщением fix: [описание]
```

Шаблон для code review

```
<!- .claude/commands/review.md ->
Проведи code review: $ARGUMENTS
Проверь:
- [ ] Корректность логики
```

```
- [ ] Обработка ошибок и edge cases
- [ ] Безопасность (SQL injection, XSS, CSRF)
- [ ] Производительность
- [ ] Тестирование
- [ ] Документация
- [ ] Соответствие code style
- [ ] Нет ли дублирования кода
Формат ответа:
[!] Critical: [описание] (строка N)
[~] Warning: [описание] (строка N)
[i] Info: [описание] (строка N)
[+] Хорошие практики: [что сделано хорошо]
```

Использование:

```
> /project:review src/api/routes/payments.ts
```

Итоги главы

В этой главе мы освоили:

– Три уровня зрелости промптов: от наивного к экспертному
– Пять принципов эффективных промптов: конкретность, контекст, разделение, примеры, роль
– Типы промптов для разных задач: генерация, отладка, рефакторинг, review, документирование
– CLAUDE.md как «память проекта»: структура, примеры, иерархия
– Антипаттерны: шесть типичных ошибок при работе с AI
– Итеративную разработку: техники «Да, но», расширения горизонта, критики, сравнения
– Продвинутые техники: Chain-of-Thought, Few-Shot, Negative Prompting
– Стратегии работы с большими кодовыми базами
– Создание собственных шаблонов промптов

С этой главой мы завершаем первую часть книги – «Введение и основы». Вы освоили фундамент: установку, первые шаги, архитектуру и эффективное общение. Во второй части мы перейдём к практическому применению Claude Code в реальных задачах разработки.

Часть II. Практическое использование

Глава 6. Написание кода с нуля

6.1. От идеи до рабочего кода: методология

Создание программного обеспечения с нуля – задача, в которой Claude Code раскрывается в полной мере. Вместо того чтобы начинать с пустого файла и чистого редактора, вы начинаете с описания того, что хотите получить, и Claude Code превращает это описание в работающий код.

Однако «просто описать, что хочешь» – это упрощение. На практике существует методология, которая превращает генерацию кода из лотереи в предсказуемый процесс. Вот она:

Этап 1: Описание (Specification)

Прежде чем просить Claude Code писать код, опишите проект. Не код, а проект – на человеческом языке:

```
> Мне нужно приложение для управления личными финансами.
Пользователь может:
- Регистрировать доходы и расходы по категориям
- Видеть баланс по месяцам
- Просматривать аналитику: графики расходов по категориям,
тренд доходов/расходов за год
- Устанавливать бюджеты по категориям и получать
предупреждения при превышении
Это веб-приложение, однопользовательское (пока),
данные хранятся локально.
```

На этом этапе Claude Code может задать уточняющие вопросы или предложить технологический стек.

Этап 2: Архитектура (Architecture)

Попросите Claude Code продумать архитектуру перед кодированием:

```
> Прежде чем писать код, предложи архитектуру проекта:
- Технологический стек (обоснуй выбор)
- Структуру файлов
- Модели данных
- Основные компоненты и их взаимодействие
```

Claude Code предложит план, который вы можете обсудить и скорректировать. Это значительно дешевле, чем переделывать готовый код.

Этап 3: Скаффолдинг (Scaffolding)

Создание «скелета» проекта:

```
> Отлично, мне нравится план. Создай скелет проекта:
- Инициализируй npm, установи зависимости
- Создай структуру каталогов
- Настрой конфигурацию (TypeScript, ESLint, Prettier)
- Создай базовый сервер, который запускается и
показывает пустую страницу
```

Этап 4: Реализация (Implementation)

Последовательная реализация функций:

```
> Теперь реализуй первую функцию: добавление транзакций.
Модель транзакции уже определена в types/transaction.ts.
Нужен:
- API endpoint POST /api/transactions
- Форма добавления на фронтенде
- Сохранение в SQLite
```

Этап 5: Интеграция и тестирование

```
> Запусти приложение, добавь тестовые транзакции
и проверь, что всё работает корректно.
Если есть ошибки – исправь.
```

Эта пятишаговая методология применима к проектам любого масштаба и на любом технологическом стеке.

6.2. Создание веб-приложений

Фронтенд: SPA на React

Создание Single Page Application – один из самых распространённых сценариев. Вот как выглядит процесс:

```
> Создай React-приложение с Vite. TypeScript, React Router,
```

```
Zustand для стейт-менеджмента.
Страницы:
- / - главная (дашборд)
- /projects - список проектов
- /projects/:id - страница проекта
- /settings - настройки
Общий layout с sidebar навигацией.
Тёмная тема по умолчанию.
```

Claude Code создаст:

```
project/
+- index.html
+- package.json
+- tsconfig.json
+- vite.config.ts
+- src/
| +- main.tsx
| +- App.tsx
| +- router.tsx
| +- store/
| | +- useProjectStore.ts
| | +- useSettingsStore.ts
| +- components/
| | +- layout/
| | | +- Sidebar.tsx
| | | +- Layout.tsx
| | +- ui/
| | +- Button.tsx
| | +- Card.tsx
| +- pages/
| | +- Dashboard.tsx
| | +- Projects.tsx
| | +- ProjectDetail.tsx
| | +- Settings.tsx
| +- types/
| | +- index.ts
| +- styles/
| +- globals.css
```

Каждый файл будет содержать осмысленный код, а не заглушки. Компоненты будут использовать общую стилистику, маршрутизация будет настроена, стейт-менеджмент – подключён.

Фронтенд: SSR на Next.js

```
> Создай Next.js 14 приложение (App Router) для блога:
- Главная: список постов с превью
```

```
- /post/[slug]: страница поста (SSG)
- /admin: CMS для создания постов
- Посты в формате MDX
- SEO: мета-теги, Open Graph, sitemap.xml
- RSS-лента
```

Claude Code использует возможности App Router: layout-ы, loading-состояния, error boundaries, серверные компоненты и метаданные.

Фронтенд: Vue.js

Claude Code отлично работает и с Vue:

```
> Создай Vue 3 приложение с Composition API.
Менеджер задач в стиле Todoist:
- Проекты с цветовыми метками
- Задачи с приоритетами и датами
- Drag and Drop для сортировки
- Фильтры: сегодня, предстоящие, завершённые
Используй Pinia для стейта, Vue Router, Naive UI.
```

Полноценный стек: фронтенд + бэкенд

Для создания полного стека в одном запросе:

```
> Создай приложение для управления инвентарём.
Backend (Express + PostgreSQL + Prisma):
- CRUD для продуктов (название, SKU, количество, цена)
- Категории продуктов
- История изменений количества
- API аутентификация через JWT
Frontend (React + Vite + React Query):
- Таблица продуктов с сортировкой и фильтрацией
- Форма добавления/редактирования
- Dashboard с метриками: всего товаров, суммарная стоимость,
товары с низким остатком
- Авторизация: страница входа
Docker Compose для локальной разработки.
```

6.3. Бэкенд-разработка

Python + FastAPI

FastAPI – один из самых популярных фреймворков для создания API на Python, и Claude Code работает с ним превосходно:

```
> Создай API на FastAPI для сервиса коротких ссылок.
Функциональность:
- POST /api/links - создание короткой ссылки
Request: { "url": "https://example.com/long-path" }
Response: { "short_url": "https://s.hr/abc123", "clicks": 0 }
- GET /{short_code} - редирект на оригинальный URL
- GET /api/links/{short_code}/stats - статистика переходов
Response: { "url": "…", "clicks": 42,
"created_at": "…", "last_clicked": "…" }
Технические требования:
- SQLAlchemy + SQLite для хранения
- Алгоритм генерации: base62 от autoincrement ID
- Rate limiting: 10 запросов в минуту на IP
- CORS: разрешить все origins
Структура проекта:
app/
+- main.py
+- models.py
+- schemas.py
+- crud.py
+- database.py
+- config.py
```

Claude Code создаст все файлы с полной реализацией. Пример сгенерированного кода (упрощённо):

```
from fastapi import FastAPI, Depends, HTTPException
from fastapi.responses import RedirectResponse
from fastapi.middleware.cors import CORSMiddleware
from sqlalchemy.orm import Session
from . import crud, models, schemas
from .database import engine, get_db
models.Base.metadata.create_all(bind=engine)
app = FastAPI(title="URL Shortener", version="1.0.0")
app.add_middleware(
CORSMiddleware,
allow_origins=["*"],
allow_methods=["*"],
allow_headers=["*"],
)
```

```
    @app.post("/api/links", response_model=schemas.LinkResponse)
    def create_link(link: schemas.LinkCreate, db: Session =
Depends(get_db)):
    return crud.create_link(db, link.url)
    @app.get("/{short_code}")
    def redirect(short_code: str, db: Session = Depends(get_db)):
    link = crud.get_link(db, short_code)
    if not link:
    raise HTTPException(status_code=404, detail="Link not found")
    crud.increment_clicks(db, link)
    return RedirectResponse(url=link.original_url)
    @app.get("/api/links/{short_code}/stats",
    response_model=schemas.LinkStats)
    def get_stats(short_code: str, db: Session = Depends(get_db)):
    link = crud.get_link(db, short_code)
    if not link:
    raise HTTPException(status_code=404, detail="Link not found")
    return link
```

Node.js + Express

```
    > Создай RESTful API на Express.js для управления задачами
(Task Manager).
    Модели:
    - User: id, email, name, password_hash
    - Project: id, name, description, owner_id
    - Task: id, title, description, status, priority,
    assignee_id, project_id, due_date
    Endpoints:
    - Auth: register, login, refresh token
    - Projects: CRUD + список участников
    - Tasks: CRUD + смена статуса + назначение исполнителя
    Middleware:
    - Аутентификация (JWT)
    - Валидация (Joi)
    - Обработка ошибок
    - Логирование (Morgan)
    - Rate Limiting
    Database: PostgreSQL + Knex.js
    Организация: controller -> service -> repository
```

Go API

```
    > Создай REST API на Go с использованием Chi router.
    Функциональность: API для блог-платформы.
    Структура проекта (Clean Architecture):
    cmd/api/main.go
```

```
internal/
+- handler/ - HTTP handlers
+- service/ - бизнес-логика
+- repository/ - работа с БД
+- model/ - доменные модели
+- middleware/ - middleware
База данных: PostgreSQL через pgx.
Миграции: golang-migrate.
Конфигурация: через переменные окружения (envconfig).
Логирование: slog (stdlib).
```

Django

```
> Создай Django-проект для системы управления контентом (CMS).
Django 5.0, Django REST Framework, PostgreSQL.
Приложения:
- accounts: расширенная модель User, профили
- articles: статьи с категориями, тегами, черновиками
- media: управление изображениями
- api: сериализаторы и viewsets для всех моделей
Фичи:
- Админка с кастомизированным интерфейсом
- Полнотекстовый поиск (PostgreSQL FTS)
- Кеширование через Redis
- Celery для фоновых задач
- API документация через drf-spectacular
```

6.4. Работа с фреймворками

Claude Code знает десятки фреймворков и может генерировать идиоматичный код для каждого из них. Вот примеры для некоторых популярных фреймворков:

NestJS (Node.js)

```
> Создай NestJS модуль для системы уведомлений.
Module: NotificationsModule
Entities:
- Notification: id, user_id, type, title, body,
is_read, created_at
- NotificationPreference: user_id, channel, enabled
Services:
- NotificationService: отправка, чтение, пометка прочитанным
- EmailNotifier: отправка через SendGrid
- PushNotifier: отправка через FCM
Controllers:
```

```
- GET /notifications – с пагинацией и фильтрами
- PATCH /notifications/:id/read
- GET /notifications/unread-count
- PUT /notifications/preferences
Используй TypeORM, class-validator, class-transformer.
Guards: AuthGuard для всех endpoints.
```

Flask (Python)

```
> Создай Flask-приложение – агрегатор RSS-лент.
Blueprints:
- auth: регистрация, вход, выход (Flask-Login)
- feeds: добавление/удаление RSS-источников
- reader: отображение статей, пометка прочитанным
Модели (Flask-SQLAlchemy):
- User, Feed, Article, UserArticle (read status)
Фоновые задачи (Celery + Redis):
- Периодическая загрузка новых статей (каждые 15 минут)
- Очистка старых статей (старше 30 дней)
Шаблоны: Jinja2, Bootstrap 5
Парсинг: feedparser
```

Spring Boot (Java)

```
> Создай Spring Boot микросервис для управления каталогом про-
дуктов.
Java 21, Spring Boot 3.2.
Layers:
- Controller -> Service -> Repository
- DTOs с MapStruct
- Validation: Jakarta Bean Validation
Endpoints:
- CRUD для Product
- Категории (древовидная структура)
- Поиск с фильтрацией и пагинацией (Spring Data Specifications)
Database: PostgreSQL + Spring Data JPA + Flyway
Дополнительно:
- Spring Security (JWT)
- OpenAPI 3.0 документация (springdoc)
- Actuator для health checks
- Testcontainers для интеграционных тестов
```

6.5. Генерация API

REST API Design

Claude Code следует лучшим практикам REST API дизайна:

```
> Спроектируй REST API для сервиса доставки еды.
Покажи только спецификацию (без реализации):
Ресурсы: рестораны, меню, заказы, курьеры, пользователи
Для каждого endpoint покажи:
- HTTP метод, URL
- Request/Response body
- Коды ответов
- Заголовки аутентификации
```

Claude Code сгенерирует продуманную спецификацию:

```
## Authentication
All endpoints require `Authorization: Bearer <token>` except
noted.
## Restaurants
### List Restaurants
GET /api/v1/restaurants?lat=55.75&lng=37.62&radius=5km
&cuisine=japanese&rating_min=4.0
&page=1&per_page=20
Response 200:
{
"data": [{
"id": "rest_abc123",
"name": "Суши Мастер",
"cuisine": ["japanese", "asian"],
"rating": 4.7,
"delivery_time_min": 25,
"delivery_time_max": 40,
"minimum_order": 500,
"delivery_fee": 99,
"is_open": true,
"distance_km": 1.2
}],
"pagination": {
"page": 1,
"per_page": 20,
"total": 47,
"total_pages": 3
}
}
### Get Restaurant Detail
```

```
GET /api/v1/restaurants/:id
Response 200:
{
"id": "rest_abc123",
"name": "Суши Мастер",
"description": "Authentic Japanese cuisine…",
"address": "ул. Тверская, 12",
"phone": "+7 (495) 123-45-67",
"working_hours": {
"mon-fri": "10:00-22:00",
"sat-sun": "11:00-23:00"
},
"menu_categories": ["Суши", "Роллы", "Горячее", "Напитки"]
}
…
```

GraphQL Schema

```
> Создай GraphQL-схему для социальной сети:
- Пользователи с профилями
- Посты (текст + изображения)
- Комментарии (вложенные)
- Лайки (на посты и комментарии)
- Подписки (follower/following)
Включи: Query, Mutation, Subscription типы.
Реализуй N+1 проблему через DataLoader.
```

Claude Code создаст полную схему с оптимизированными резолверами.

6.6. Паттерны генерации кода

Паттерн «Слой за слоем»

Генерируйте код послойно, от фундамента к поверхности:

```
1. > Создай модели данных (Prisma schema)
2. > Создай repository layer
3. > Создай service layer с бизнес-логикой
4. > Создай API endpoints
5. > Создай фронтенд-компоненты
6. > Добавь тесты
```

Каждый слой опирается на предыдущий, и Claude Code использует уже сгенерированные типы и интерфейсы.

Паттерн «От тестов к коду» (TDD)

```
> Напиши тесты для модуля расчёта скидок.
Бизнес-правила:
- Скидка 5% при заказе от 5000
- Скидка 10% при заказе от 10000
- Скидка 15% для VIP-клиентов
- Скидки не суммируются – применяется максимальная
- Скидка не может превышать 20%
ТОЛЬКО тесты, код пока не пиши.
```

После генерации тестов:

```
> Теперь напиши реализацию, которая проходит все эти тесты.
```

Паттерн «Скопируй стиль»

```
> Посмотри, как реализован модуль src/modules/users/.
Создай аналогичный модуль src/modules/products/
с такой же структурой и стилем кода.
Product: id, name, description, price, category_id,
is_active, created_at
```

Claude Code проанализирует существующий модуль и создаст новый в том же стиле.

Паттерн «Прототип -> Продакшен»

```
> Создай простейший чат на WebSocket.
Минимум кода, без базы данных, всё в памяти.
> Теперь доработай до продакшен-версии:
- Сохранение сообщений в PostgreSQL
- Аутентификация через JWT
- Комнаты (rooms)
- Типизация (TypeScript)
- Обработка ошибок
- Reconnection logic на клиенте
- Тесты
```

6.7. Генерация конфигурационных файлов

Отдельно стоит отметить способность Claude Code генерировать конфигурационные файлы – область, где разработчики тратят несоразмерно много времени.

TypeScript конфигурация

```
> Настрой TypeScript для монорепозитория:
- Общий tsconfig.base.json
- tsconfig.json для пакета api/ (Node.js backend)
- tsconfig.json для пакета web/ (React frontend)
- tsconfig.json для пакета shared/ (общие типы)
Strict mode, path aliases, incremental compilation.
```

ESLint + Prettier

```
> Настрой ESLint 9 (flat config) + Prettier для
TypeScript React проекта.
Правила:
- Запрет any (error)
- Запрет console.log (warning)
- Хуки React (правила exhaustive-deps)
- Сортировка импортов
- Максимальная длина строки: 100
Игнорировать: dist/, node_modules/, .next/
```

Docker + Docker Compose

```
> Создай Docker-окружение для разработки:
Сервисы:
- app: Node.js 20 (Alpine), hot reload
- db: PostgreSQL 16
- redis: Redis 7
- adminer: GUI для БД
Volumes для persistence данных БД.
Health checks для всех сервисов.
.dockerignore с исключениями.
```

CI/CD

```
> Создай GitHub Actions workflow:
Triggers: push to main, pull_request
Jobs:
1. Lint: ESLint + Prettier check
2. Test: unit + integration (Vitest)
3. Build: production build
4. Deploy (только для main): deploy to Vercel
Cache: node_modules, .next/cache
Matrix: Node 18, 20
```

```
Secrets: VERCEL_TOKEN, DATABASE_URL
```

6.8. Распространённые ошибки при генерации

Зная типичные ошибки, можно их предотвращать:

1. Устаревшие API. Claude Code может использовать устаревшие версии библиотек или deprecated функции. Решение: указывайте версии в промпте.

```
> Используй Next.js 14 App Router (НЕ Pages Router).
Используй server actions (НЕ API routes для мутаций).
```

2. Избыточный код. Генерация может производить больше кода, чем необходимо. Решение: указывайте принцип минимализма.

```
> Минимальная реализация. Никаких лишних абстракций.
YAGNI – не добавляй то, что не просил.
```

3. Несогласованность стилей. В длинном диалоге стиль может «дрейфовать». Решение: используйте CLAUDE.md и ссылайтесь на существующий код.

4. Неполные edge cases. AI часто реализует happy path, забывая о граничных случаях. Решение: явно перечисляйте edge cases.

```
> Обработай edge cases:
- Пустой ввод
- Очень длинная строка (10000+ символов)
- Спецсимволы и emoji в имени
- Конкурентные запросы к тому же ресурсу
- Потеря соединения с БД
```

Итоги главы

В этой главе мы рассмотрели:

– Методологию генерации кода: от описания через архитектуру к реализации
– Создание веб-приложений на React, Next.js, Vue.js
– Бэкенд-разработку на FastAPI, Express, Go, Django
– Работу с фреймворками: NestJS, Flask, Spring Boot
– Проектирование и генерацию REST API и GraphQL
– Паттерны генерации: послойный, TDD, копирование стиля, прототип->продакшен
– Генерацию конфигурационных файлов: TypeScript, ESLint, Docker, CI/CD
– Типичные ошибки и способы их предотвращения

В следующей главе мы научимся работать с существующим кодом: рефакторинг, модернизация и улучшение того, что уже написано.

Глава 7. Рефакторинг и улучшение существующего кода

7.1. Понимание чужого кода: первый контакт

Каждый разработчик рано или поздно сталкивается с необходимостью разбираться в чужом коде – или в своём собственном, написанном год назад. Claude Code превращает этот мучительный процесс в структурированное исследование.

Первичный анализ

Начните с общего обзора:

```
> Проанализируй этот проект. Мне нужно понять:
1. Что это за приложение и что оно делает
2. Какой технологический стек используется
3. Как организована кодовая база
4. Какие основные модули и как они связаны
5. Есть ли документация и тесты
```

Claude Code методично исследует проект: прочитает package.json / requirements.txt / go.mod, просмотрит структуру каталогов, откроет ключевые файлы (точки входа, конфигурацию, основные модули) и составит связное описание.

Анализ конкретного модуля

После общего обзора можно углубиться:

```
> Объясни, как работает модуль авторизации (src/auth/).
Покажи:
- Flow аутентификации (от запроса до ответа)
- Какие middleware задействованы
- Как хранятся и проверяются токены
- Какие роли и разрешения существуют
```

Визуализация зависимостей

```
> Построй карту зависимостей между модулями в src/.
Какие модули от каких зависят?
Есть ли циклические зависимости?
```

Claude Code проанализирует import/require и нарисует картину зависимостей, выявив потенциальные проблемы.

Поиск «скелетов в шкафу»

```
> Найди в проекте:
- TODO и FIXME комментарии
- Закомментированный код
- Неиспользуемые переменные и импорты
- Дублирование кода
- Файлы без тестов
- Захардкоженные значения (magic numbers/strings)
```

7.2. Code Smells: обнаружение и устранение

Code smells – это признаки потенциальных проблем в коде. Они не являются ошибками сами по себе, но указывают на области, которые выиграют от рефакторинга.

Обнаружение code smells

```
> Проанализируй файл src/services/orderService.ts на наличие
code smells. Для каждого найденного smell укажи:
- Название (по каталогу Мартина Фаулера)
- Строки кода
- Почему это проблема
- Как исправить
```

Claude Code может обнаружить следующие распространённые проблемы:

God Object (Божественный объект) – класс, который делает слишком много:

```
// До: класс на 800 строк
class UserManager {
createUser() { /* … */ }
deleteUser() { /* … */ }
sendEmail() { /* … */ }
processPayment() { /* … */ }
generateReport() { /* … */ }
updateSubscription() { /* … */ }
handleWebhook() { /* … */ }
migrateData() { /* … */ }
}
```

```
> Этот класс UserManager слишком большой (800 строк) и делает
слишком много. Разбей его на несколько классов по принципу
Single Responsibility.
```

Claude Code создаст:

– `UserService` – CRUD операции с пользователями
– `EmailService` – отправка писем
– `PaymentService` – обработка платежей
– `ReportService` – генерация отчётов
– `SubscriptionService` – управление подписками
– `WebhookHandler` – обработка вебхуков

И обновит все файлы, которые зависят от `UserManager`.

Long Method (Длинный метод):

```
> Функция processOrder в src/services/order.ts занимает
150 строк. Разбей её на осмысленные подфункции.
Каждая подфункция – одно действие.
```

Feature Envy (Зависть к функциональности):

```
> В классе OrderService много обращений к полям Customer.
Вынеси эту логику в CustomerService, где ей место.
```

Shotgun Surgery (Дробная хирургия):

```
> При добавлении нового типа уведомления нужно менять
5 файлов. Отрефактори так, чтобы изменения были локализованы
в одном месте (паттерн Strategy или Registry).
```

Систематический аудит кода

Для комплексного аудита:

```
> Проведи полный аудит кода в каталоге src/services/.
Проверь на:
1. Сложность (цикломатическая > 10)
2. Дублирование (> 5 строк повторяющегося кода)
3. Длинные функции (> 50 строк)
4. Глубокая вложенность (> 3 уровня)
5. Магические числа и строки
6. Нарушение DRY
7. Нарушение SRP
8. Отсутствие обработки ошибок
Формат: таблица с колонками [Файл, Проблема, Серьёзность, Рекомендация]
```

7.3. Автоматический рефакторинг

Переименование

Переименование – кажущаяся простой операция, которая может затрагивать десятки файлов:

```
> Переименуй класс UserDTO в UserResponse во всём проекте.
Обнови все импорты, ссылки и комментарии.
```

Claude Code найдёт все упоминания, обновит их и проверит, что код компилируется.

Извлечение функций

```
> В файле src/utils/validators.ts есть функция validateForm(),
которая занимается валидацией email, телефона, адреса и
кредитной карты. Извлеки каждую валидацию в отдельную
функцию и используй их в validateForm.
```

Результат:

```
// До
function validateForm(data: FormData): ValidationResult {
// 150 строк с валидацией всего подряд
}
// После
function validateEmail(email: string): ValidationError | null
{ … }
function validatePhone(phone: string): ValidationError | null
{ … }
function validateAddress(address: Address): ValidationError |
null { … }
function validateCreditCard(card: CardInfo): ValidationError
| null { … }
function validateForm(data: FormData): ValidationResult {
const errors = [
validateEmail(data.email),
validatePhone(data.phone),
validateAddress(data.address),
validateCreditCard(data.card),
].filter(Boolean);
return { isValid: errors.length === 0, errors };
}
```

Извлечение классов

```
> Класс PaymentProcessor содержит логику для Stripe, PayPal
и банковских переводов. Примени паттерн Strategy:
- Общий интерфейс PaymentGateway
- StripeGateway implements PaymentGateway
- PayPalGateway implements PaymentGateway
- BankTransferGateway implements PaymentGateway
- PaymentProcessor использует нужный Gateway через DI
```

Inline рефакторинг

Иногда код чрезмерно абстрагирован:

```
> Класс StringHelper содержит метод isEmpty, который просто
проверяет !str || str.trim().length === 0. Замени все
вызовы StringHelper.isEmpty(str) на прямую проверку
и удали класс.
```

7.4. Модернизация legacy-кода

Работа с legacy-кодом – одна из самых ценных применений Claude Code. Модернизация может включать:

Миграция на современный синтаксис

```
> Модернизируй файл src/legacy/dataProcessor.js:
- var -> const/let
- function declarations -> arrow functions (где уместно)
- callbacks -> async/await
- string concatenation -> template literals
- for loops -> map/filter/reduce (где уместно)
- require -> import
- module.exports -> export
Сохрани семантику кода, меняй только синтаксис.
```

Пример трансформации:

```
// До (ES5)
var fs = require('fs');
var path = require('path');
function readConfig(configPath, callback) {
var fullPath = path.join(__dirname, configPath);
fs.readFile(fullPath, 'utf8', function(err, data) {
```

```
    if (err) {
    callback(err, null);
    return;
    }
    try {
    var config = JSON.parse(data);
    callback(null, config);
    } catch (parseErr) {
    callback(parseErr, null);
    }
    });
    }
    module.exports = {
    readConfig: readConfig
    };

    // После (Modern TypeScript)
    import { readFile } from 'fs/promises';
    import { join } from 'path';
    interface Config {
    [key: string]: unknown;
    }
    export const readConfig = async (configPath: string):
Promise<Config> => {
    const fullPath = join(__dirname, configPath);
    const data = await readFile(fullPath, 'utf8');
    return JSON.parse(data) as Config;
    };
```

Миграция JavaScript -> TypeScript

```
    > Мигрируй проект с JavaScript на TypeScript.
    Стратегия поэтапной миграции:
    1. Добавь tsconfig.json с allowJs: true
    2. Переименуй файлы в src/models/ в .ts
    3. Добавь типы для моделей
    4. Переименуй файлы в src/services/ в .ts
    5. Добавь типы для сервисов
    6. Продолжи с остальными каталогами
    Для каждого файла:
    - Добавь интерфейсы для параметров и возвратных значений
    - Замени any на конкретные типы
    - Используй strict mode
```

Миграция между фреймворками

```
    > Мигрируй React компонент с Class Components на Hooks.
```

```
Файл: src/components/Dashboard.jsx
Преобразуй:
- this.state -> useState
- componentDidMount -> useEffect
- componentDidUpdate -> useEffect с зависимостями
- componentWillUnmount -> useEffect cleanup
- this.setState -> setState functions
- event handlers -> useCallback
- computed values -> useMemo
```

Обновление зависимостей

```
> Обнови зависимости проекта до актуальных версий.
1. Покажи текущие версии и доступные обновления
2. Для каждого major-обновления укажи breaking changes
3. Обнови package.json
4. Исправь breaking changes в коде
5. Запусти тесты и исправь ошибки
```

7.5. Оптимизация производительности

Профилирование через Claude Code

```
> Проанализируй функцию generateReport в src/reports/
generator.ts.
Она работает медленно при большом количестве данных (>10000
записей).
Найди узкие места:
1. Алгоритмическая сложность (O(n²) или хуже?)
2. Лишние итерации по данным
3. Неоптимальные запросы к БД (N+1?)
4. Отсутствие кэширования
5. Блокирующие операции
Предложи оптимизации с оценкой ожидаемого ускорения.
```

Оптимизация SQL-запросов

```
> Этот SQL-запрос выполняется 5 секунд на таблице с 1M записей.
Оптимизируйте его:
SELECT o.*, u.name, u.email,
(SELECT COUNT(*) FROM order_items WHERE order_id = o.id) as
item_count,
(SELECT SUM(price) FROM order_items WHERE order_id = o.id) as
total
```

```
FROM orders o
JOIN users u ON o.user_id = u.id
WHERE o.created_at > '2024-01-01'
ORDER BY o.created_at DESC;
Также предложи индексы, которые ускорят запрос.
```

Claude Code предложит:

```
-- Оптимизированная версия
SELECT o.*, u.name, u.email,
oi.item_count, oi.total
FROM orders o
JOIN users u ON o.user_id = u.id
LEFT JOIN (
SELECT order_id,
COUNT(*) as item_count,
SUM(price) as total
FROM order_items
GROUP BY order_id
) oi ON oi.order_id = o.id
WHERE o.created_at > '2024-01-01'
ORDER BY o.created_at DESC;
-- Индексы
CREATE INDEX idx_orders_created_at ON orders(created_at DESC);
CREATE INDEX idx_order_items_order_id ON
order_items(order_id);
```

Оптимизация бэкенда

```
> Оптимизируй API endpoint GET /api/products:
Текущая проблема: время ответа 2-3 секунды
Ожидание: < 200ms
Проверь:
1. N+1 запросы (eager loading vs lazy loading)
2. Отсутствие пагинации
3. Отсутствие кэширования
4. Избыточные данные в ответе (select only needed fields)
5. Отсутствие индексов
Реализуй: Redis-кэширование с инвалидацией при изменении.
```

Оптимизация фронтенда

```
> Проанализируй производительность React-приложения:
1. Найди ненужные ре-рендеры (пропущенные React.memo)
2. Найди тяжёлые вычисления без useMemo
3. Проверь размер бандла (можно ли что-то lazy import)
4. Проверь изображения (оптимизация, lazy loading)
```

```
5. Предложи code splitting
```

7.6. Декомпозиция монолита

Один из самых сложных видов рефакторинга – разбиение монолитного приложения на модули или микросервисы.

```
> Проанализируй монолит в src/ и предложи разделение
на модули (bounded contexts):
1. Определи основные домены
2. Выяви зависимости между ними
3. Предложи границы модулей
4. Оцени сложность разделения
5. Предложи порядок декомпозиции (начни с наименее связанного)
```

Claude Code проанализирует код и может предложить:

```
Предлагаю выделить 4 bounded context:
1. User Management (низкая связность, легко отделить)
- src/models/user.ts
- src/services/userService.ts
- src/controllers/userController.ts
-> отдельный модуль user/
2. Billing (средняя связность)
- src/models/subscription.ts, payment.ts
- src/services/billingService.ts, stripeService.ts
- Зависит от User (user_id)
-> отдельный модуль billing/
3. Content (высокая связность с User)
- src/models/article.ts, comment.ts
- src/services/contentService.ts
- Зависит от User (author_id)
-> отдельный модуль content/
4. Analytics (только на чтение)
- src/services/analyticsService.ts
- Зависит от всех модулей (read-only)
-> отдельный модуль analytics/
Порядок декомпозиции: Analytics -> User -> Billing -> Content
```

7.7. Улучшение обработки ошибок

```
> Проведи аудит обработки ошибок в проекте:
1. Найди все try-catch блоки и оцени, правильно ли они
обрабатывают ошибки (не глотают ли молча?)
2. Найди async функции без try-catch
3. Найди промисы без .catch()
4. Проверь, есть ли централизованная обработка ошибок
```

```
5. Предложи архитектуру обработки ошибок:
- Иерархия Error-классов
- Middleware для обработки ошибок
- Формат ответа об ошибке для клиента
- Логирование ошибок
```

Claude Code создаст систему обработки ошибок:

```
// src/errors/AppError.ts
export class AppError extends Error {
constructor(
public statusCode: number,
public code: string,
message: string,
public isOperational = true,
) {
super(message);
Object.setPrototypeOf(this, AppError.prototype);
}
}
export class NotFoundError extends AppError {
constructor(resource: string, id: string) {
super(404, 'NOT_FOUND', `${resource} with id ${id} not found`);
}
}
export class ValidationError extends AppError {
constructor(
public errors: Record<string, string[]>,
) {
super(422, 'VALIDATION_ERROR', 'Validation failed');
}
}
export class AuthenticationError extends AppError {
constructor(message = 'Authentication required') {
super(401, 'UNAUTHORIZED', message);
}
}
export class AuthorizationError extends AppError {
constructor(message = 'Insufficient permissions') {
super(403, 'FORBIDDEN', message);
}
}
```

7.8. Практический кейс: рефакторинг реального проекта

Рассмотрим полный цикл рефакторинга на примере. Допустим, вам достался проект – Express.js API, написанный в стиле «всё в одном файле».

Исходное состояние

```
app.js (1200 строк) – весь код в одном файле:
– Express конфигурация
– Подключение к MongoDB
– 15 route handlers
– Валидация
– Аутентификация
– Бизнес-логика
– Отправка email
```

Шаг 1: Анализ

```
> Проанализируй app.js и предложи план рефакторинга.
Цель: разделить на модули, следуя MVC-архитектуре.
Не пиши код – только план.
```

Шаг 2: Структура

```
> Создай файловую структуру проекта (пустые файлы):
src/
+- config/
+- middleware/
+- models/
+- routes/
+- controllers/
+- services/
+- utils/
```

Шаг 3: Последовательный вынос

```
> Вынеси конфигурацию из app.js в src/config/
> Вынеси модели в src/models/
> Вынеси middleware в src/middleware/
> Вынеси маршруты в src/routes/
> Вынеси контроллеры в src/controllers/
> Вынеси бизнес-логику в src/services/
```

Шаг 4: Проверка

```
> Запусти приложение и проверь, что всё работает как раньше.
Запусти тесты, если есть.
```

Шаг 5: Улучшение

```
> Теперь улучши:
- Добавь TypeScript
- Добавь валидацию через Joi/Zod
- Добавь централизованную обработку ошибок
- Добавь логирование через winston
- Обнови зависимости
```

Этот подход можно назвать «рефакторинг как миграция»: каждый шаг переносит часть функциональности, и на каждом шаге приложение остаётся работоспособным.

Итоги главы

В этой главе мы освоили:

– Технику первичного анализа чужой кодовой базы
– Обнаружение code smells и систематический аудит
– Автоматический рефакторинг: переименование, извлечение, инлайнинг
– Модернизацию legacy-кода: синтаксис, миграция на TypeScript, обновление фреймворков
– Оптимизацию производительности: SQL, бэкенд, фронтенд
– Декомпозицию монолита на модули
– Улучшение обработки ошибок
– Практический кейс полного рефакторинга

В следующей главе мы погрузимся в отладку – искусство поиска и устранения ошибок с помощью Claude Code.

Глава 8. Отладка и исправление ошибок

8.1. Философия отладки с AI-ассистентом

Отладка – это детективная работа. Вы расследуете инцидент: программа ведёт себя не так, как ожидалось. У вас есть улики (логи, stack traces, симптомы), подозреваемые (код, конфигурация, данные) и инструменты (отладчик, тесты, мониторинг).

Claude Code добавляет в этот арсенал нечто принципиально новое – «свежий взгляд» с глубоким знанием технологий. Когда вы часами смотрите на один и тот же код, вы перестаёте видеть ошибку. Claude Code не страдает от «туннельного зрения» – он анализирует код непредвзято.

Но у AI-отладки есть своя специфика. Claude Code не может запустить отладчик и пошагово пройти по коду, наблюдая значения переменных. Вместо этого он анализирует код статически, формулирует гипотезы и проверяет их через запуск тестов или диагностических команд.

Эффективная отладка с Claude Code следует циклу из четырёх шагов:

– **Описание** – чёткая формулировка проблемы
– **Гипотеза** – предположение о причине
– **Проверка** – эксперимент для подтверждения или опровержения
– **Исправление** – устранение корневой причины

8.2. Анализ Stack Traces

Stack trace – первый и часто главный ключ к пониманию ошибки. Claude Code превосходно их анализирует.

Пример с Python

```
    > Приложение падает при обработке заказа. Вот stack trace:
    Traceback (most recent call last):
    File "/app/services/order_service.py", line 87, in
process_order
    discount = self.calculate_discount(order)
    File "/app/services/order_service.py", line 124, in
calculate_discount
    promo = self.promo_repo.get_active(order.promo_code)
    File "/app/repositories/promo_repo.py", line 31, in get_active
    return self.session.query(Promo).filter(
    File "/usr/local/lib/python3.12/site-packages/sqlalchemy/orm/
query.py"
```

```
    …
    sqlalchemy.exc.OperationalError: (psycopg2.OperationalError)
    connection to server at "db" (172.18.0.2), port 5432 failed:
    Connection refused
```

Claude Code проанализирует и скажет:

```
    Проблема: Потеря соединения с PostgreSQL при расчёте скидки.
    Корневая причина: Соединение с БД разорвано (Connection
refused).
    Это может быть вызвано:
    1. БД перезагружена, а пул соединений не обновился
    2. Пул соединений исчерпан (все connections заняты)
    3. Сетевой сбой между app и db контейнерами
    Давайте проверим…
```

И далее Claude Code самостоятельно проверит:

– Доступность БД (`pg_isready`)
– Конфигурацию пула соединений
– Количество активных соединений
– Настройки таймаутов

Пример с Node.js

```
    > UnhandledPromiseRejectionWarning: TypeError: Cannot read
    properties of undefined (reading 'map')
    at       UserController.getUsers       (/app/src/controllers/
user.ts:45:28)
    at   Layer.handle   [as   handle_request]   (/app/node_modules/
express/lib/router/layer.js:95:5)
```

Claude Code определит:

– Строку 45 в user.ts, где вызывается `.map()` на undefined
– Проверит, что функция ожидает массив, но получает undefined
– Найдёт место, где данные должны были быть инициализированы

Пример с multiline/nested ошибками

```
    > Вот логи из продакшена (последние 50 строк):
    [2026-02-16 10:15:23] ERROR: Request failed
    [2026-02-16 10:15:23] Caused by: TimeoutError:
    connection pool is exhausted
    [2026-02-16 10:15:23] Active connections: 20/20
    [2026-02-16 10:15:23] Waiting queue: 47
    [2026-02-16 10:15:24] ERROR: Request failed
```

```
[2026-02-16 10:15:24] ERROR: Request failed
...
```

Claude Code увидит паттерн и предложит решение: увеличить пул, добавить таймаут для соединений, найти запросы-«долгожители».

8.3. Поиск багов в коде

Систематический поиск

```
> В функции processPayment что-то не так – клиенты жалуются,
что деньги списываются дважды. Найди причину.
Файл: src/services/payment.ts
```

Claude Code прочитает файл и может обнаружить:

```
// Проблема: отсутствие идемпотентности
async processPayment(orderId: string, amount: number) {
const order = await this.orderRepo.findById(orderId);
// BUG: Нет проверки, был ли уже обработан платёж!
// Если пользователь нажмёт кнопку дважды или
// произойдёт retry – деньги спишутся повторно.
const charge = await stripe.charges.create({
amount: amount * 100,
currency: 'usd',
source: order.paymentSource,
});
order.status = 'paid';
await this.orderRepo.save(order);
}
```

И предложит исправление:

```
async processPayment(orderId: string, amount: number) {
const order = await this.orderRepo.findById(orderId);
// Проверка идемпотентности
if (order.status === 'paid') {
return { already_paid: true };
}
// Оптимистичная блокировка
const updated = await this.orderRepo.updateStatus(
orderId, 'pending', 'processing'
);
if (!updated) {
throw new ConflictError('Order is being processed');
}
try {
```

```
    const charge = await stripe.charges.create({
    amount: amount * 100,
    currency: 'usd',
    source: order.paymentSource,
    idempotency_key: `order_${orderId}`, // Идемпотентность Stripe
    });
    await   this.orderRepo.updateStatus(orderId,   'processing',
'paid');
    return { charge_id: charge.id };
    } catch (error) {
    await   this.orderRepo.updateStatus(orderId,   'processing',
'pending');
    throw error;
    }
    }
```

Поиск утечек памяти

```
> Приложение постепенно потребляет всё больше памяти.
Через 24 часа работы – OOM kill.
Помоги найти утечку памяти. Проверь:
1. Event listeners, которые не снимаются
2. Кэши без ограничения размера
3. Замыкания, удерживающие большие объекты
4. Глобальные переменные, накапливающие данные
5. Незакрытые соединения с БД/Redis
```

Claude Code проверит указанные паттерны и может найти:

```
// Утечка: массив растёт без ограничений
const requestLog = []; // Глобальный массив!
app.use((req, res, next) => {
requestLog.push({ // Каждый запрос добавляет запись
url: req.url,
timestamp: Date.now(),
headers: req.headers, // Большой объект!
});
next();
});
```

Поиск race conditions

```
> Иногда (не каждый раз!) при создании заказа количество
товара на складе уходит в минус. Подозреваю race condition.
Файлы:
- src/services/orderService.ts
- src/repositories/inventoryRepo.ts
```

Claude Code найдёт:

```
// Проблема: check-then-act без блокировки
async createOrder(productId: string, quantity: number) {
const inventory = await this.inventoryRepo.getStock(productId);
// RACE CONDITION: Между проверкой и списанием другой
// запрос может изменить inventory!
if (inventory.stock >= quantity) {
await this.inventoryRepo.decreaseStock(productId, quantity);
// ...
}
}
```

Решение: использовать атомарную операцию в БД:

```
UPDATE inventory
SET stock = stock - $quantity
WHERE product_id = $productId AND stock >= $quantity
RETURNING stock;
```

8.4. Отладка асинхронного кода

Асинхронный код – рассадник тонких ошибок. Claude Code помогает найти типичные проблемы.

Забытый await

```
> Эндпоинт возвращает пустой массив вместо данных.
Проверь src/controllers/products.ts
```

Claude Code может обнаружить:

```
// BUG: забыт await!
async getProducts(req: Request, res: Response) {
const products = this.productService.findAll(); // Promise!
res.json(products); // Отправляет Promise, не данные
}
```

Необработанные промисы

```
> Найди все промисы без обработки ошибок в каталоге src/
```

Claude Code выполнит поиск по паттернам и найдёт «висящие» промисы, fire-and-forget вызовы без `.catch()` и async-функции без try-catch в контексте Express-обработчиков.

Deadlock в async-коде

```
> Приложение периодически зависает. Подозреваю deadlock
в async-операциях. Помоги найти.
```

Claude Code проанализирует цепочки await-вызовов и может обнаружить:

```
// Deadlock: два сервиса ждут друг друга
// Service A Service B
// lock(user) -> lock(order) ->
// await lock(order) await lock(user) <- DEADLOCK!
```

8.5. Отладка конфигурации

Ошибки конфигурации – одни из самых коварных, потому что код выглядит правильным, а приложение всё равно не работает.

```
> Приложение не подключается к Redis. Локально работает,
в Docker - нет. Ошибка: ECONNREFUSED 127.0.0.1:6379
Файлы:
- docker-compose.yml
- src/config.ts
- .env
```

Claude Code проверит конфигурацию и может обнаружить:

```
// config.ts
const redisUrl = process.env.REDIS_URL || 'redis://
localhost:6379';
// В Docker "localhost" указывает на сам контейнер приложения,
// а не на контейнер Redis!
// Нужно: 'redis://redis:6379' (имя сервиса Docker Compose)
```

Отладка переменных окружения

```
> Приложение использует неправильные настройки в production.
Помоги разобраться:
1. Покажи все .env файлы и их содержимое
2. Покажи, как config загружает переменные
3. Какие значения используются по умолчанию
4. Есть ли конфликты между .env файлами
```

Отладка CORS

```
> Фронтенд на localhost:3000 не может обратиться к API
на localhost:8080. В консоли:
"Access to XMLHttpRequest has been blocked by CORS policy"
Покажи текущую конфигурацию CORS и исправь.
```

8.6. Отладка производительности

```
> API endpoint GET /api/dashboard отвечает за 5+ секунд.
Помоги найти причину и оптимизировать.
1. Проанализируй код endpoint'а
2. Определи, какие операции занимают больше всего времени
3. Есть ли N+1 запросы?
4. Есть ли вычисления, которые можно кэшировать?
5. Можно ли что-то выполнить параллельно?
```

Claude Code проанализирует и может найти:

```
// Проблема: последовательные запросы, которые можно
// выполнять параллельно
async getDashboard(userId: string) {
const user = await this.userRepo.findById(userId);
const orders = await this.orderRepo.findByUser(userId); //
500ms
const stats = await this.statsService.calculate(userId); //
800ms
const notifications = await
this.notifRepo.getUnread(userId); // 300ms
const recommendations = await this.recService.get(userId); //
1200ms
// Итого: ~2800ms последовательно
}
```

Оптимизация:

```
async getDashboard(userId: string) {
const [user, orders, stats, notifications, recommendations] =
await Promise.all([
this.userRepo.findById(userId),
this.orderRepo.findByUser(userId),
this.statsService.calculate(userId),
this.notifRepo.getUnread(userId),
this.recService.get(userId),
]);
// Итого: ~1200ms (время самого медленного запроса)
}
```

8.7. Стратегии отладки

Бинарный поиск ошибки

Когда ошибка возникла после множества изменений:

```
> После последних 10 коммитов приложение стало падать.
Помоги найти коммит, который внёс ошибку.
Используй git bisect или просмотри diff каждого коммита.
```

Claude Code может использовать `git bisect` автоматически:

```
git bisect start
git bisect bad # Текущий коммит – плохой
git bisect good abc123 # Этот коммит – хороший
```

Метод «резинового утёнка» с AI

Иногда достаточно описать проблему подробно:

```
> Я не могу понять, почему тест test_user_creation
проходит локально, но падает в CI. Давай разберёмся
пошагово.
Вот вывод CI:
[вставка логов]
Вот мой локальный вывод:
[вставка логов]
```

Claude Code заметит различия в окружении: версии зависимостей, переменные окружения, конфигурацию БД и т. д.

Воспроизведение ошибки

```
> Клиент сообщает об ошибке, но я не могу её воспроизвести.
Описание: при добавлении товара в корзину с количеством 999
происходит ошибка 500.
Помоги:
1. Написать тест, который воспроизводит эту ситуацию
2. Определить причину
3. Исправить
4. Убедиться, что тест проходит
```

8.8. Реальные кейсы: сложные баги

Кейс 1: Призрачные данные

```
> Проблема: пользователи видят данные других пользователей.
Не всегда – только при определённых условиях.
Это приложение Express.js с аутентификацией через JWT.
Помоги найти причину.
```

Claude Code может обнаружить проблему на уровне кэширования:

```
// Проблема: глобальный кэш без учёта пользователя!
const cache = {};
async function getUserDashboard(req, res) {
const key = 'dashboard'; // Один ключ для всех пользователей!
if (cache[key]) {
return res.json(cache[key]); // Один пользователь видит
// данные другого!
}
const data = await fetchDashboard(req.user.id);
cache[key] = data;
res.json(data);
}
```

Кейс 2: Периодические таймауты

```
> Раз в 2–3 часа приложение начинает отвечать с таймаутами.
Через 5 минут всё восстанавливается.
Инфраструктура: Node.js, PostgreSQL, Redis, Nginx.
Теории:
1. Garbage collection? (Node.js)
2. PostgreSQL autovacuum?
3. Redis persistence (RDB/AOF)?
4. Nginx worker restarts?
Помоги разобраться.
```

Claude Code предложит план диагностики и может найти, что PostgreSQL autovacuum на большой таблице блокирует запросы.

Кейс 3: Невоспроизводимый edge case

```
> Один из 10000 заказов обрабатывается с ошибкой.
Ошибка: "Invalid amount: NaN"
Не могу понять, как amount становится NaN.
```

```
Помоги проследить путь данных от API до расчёта.
```

Claude Code проследит цепочку и может найти:

```
// В одном из 10000 случаев клиент отправляет пустую строку
// вместо числа, и parseInt("") === NaN
const amount = parseInt(req.body.amount);
// Нет проверки isNaN(amount)!
```

8.9. Автоматизация отладки

Скрипт для сбора диагностики

```
> Создай диагностический скрипт (scripts/diagnose.sh),
который собирает:
- Версии Node.js, npm, системы
- Статус зависимостей (npm ls -depth=0)
- Результат тестов
- Состояние БД (подключение, размер, активные запросы)
- Использование памяти и CPU
- Логи последних ошибок
Вывод в формате, удобном для отправки в issue.
```

Регрессионные тесты для багов

```
> Для каждого бага, который мы нашли сегодня, напиши
регрессионный тест:
1. Двойное списание при race condition ->
тест на конкурентные запросы
2. NaN в amount -> тест на невалидный ввод
3. Кэш без привязки к пользователю ->
тест на изоляцию данных
```

Итоги главы

В этой главе мы освоили:

– Философию отладки с AI: цикл описание->гипотеза->проверка->исправление
– Анализ stack traces на Python, Node.js и смешанных
– Поиск типичных багов: дублирование операций, утечки памяти, race conditions
– Отладку асинхронного кода: забытые await, необработанные промисы, дедлоки
– Отладку конфигурации: Docker, переменные окружения, CORS
– Отладку производительности: N+1 запросы, последовательные вызовы, кэширование
– Стратегии: бинарный поиск, «резиновый утёнок», воспроизведение

– Реальные кейсы: призрачные данные, периодические таймауты, edge cases
– Автоматизацию отладки: диагностические скрипты и регрессионные тесты

В следующей главе мы научимся писать тесты – лучшую профилактику от багов.

Глава 9. Написание тестов

9.1. Почему тесты – лучшая инвестиция в качество

Тесты – это страховка для вашего кода. Без тестов каждое изменение становится игрой в русскую рулетку: работает? сломал что-то? а в продакшене? Claude Code превращает написание тестов из утомительной обязанности в быстрый и даже увлекательный процесс.

Есть одна прекрасная ирония в написании тестов с помощью AI: Claude Code знает типичные ошибки и edge cases лучше большинства разработчиков, потому что он обучен на миллиардах строк кода. Он «видел» тысячи вариантов тех же самых функций и знает, что может пойти не так. Это делает его идеальным кандидатом для генерации тестов – именно потому, что его собственный код может содержать ошибки, и тесты являются механизмом их обнаружения.

Виды тестов

Кратко напомним иерархию тестирования:

Unit-тесты проверяют отдельные функции и классы в изоляции. Быстрые, простые, многочисленные. Составляют основу тестовой пирамиды.

Интеграционные тесты проверяют взаимодействие нескольких компонентов: сервис + база данных, API endpoint + middleware + валидация. Медленнее, но надёжнее.

E2E (End-to-End) тесты проверяют всю систему целиком: от пользовательского интерфейса через бэкенд до базы данных. Самые медленные и хрупкие, но дают максимальную уверенность.

Claude Code эффективно генерирует все три вида.

9.2. Unit-тесты

Генерация тестов для существующего кода

Самый частый сценарий – у вас есть код без тестов:

```
> Напиши unit-тесты для файла src/utils/validators.ts.
Покрой все публичные функции. Для каждой функции
протестируй: happy path, edge cases, невалидный ввод.
Фреймворк: Vitest
```

Claude Code прочитает файл, проанализирует функции и сгенерирует тесты:

```
// src/utils/__tests__/validators.test.ts
import { describe, it, expect } from 'vitest';
import {
validateEmail,
validatePhone,
validatePassword,
validateUrl
} from '../validators';
describe('validateEmail', () => {
it('should accept valid email addresses', () => {
expect(validateEmail('user@example.com')).toBe(true);
expect(validateEmail('name.surname@domain.co.uk')).toBe(true);
expect(validateEmail('user+tag@example.com')).toBe(true);
});
it('should reject invalid email addresses', () => {
expect(validateEmail('')).toBe(false);
expect(validateEmail('plaintext')).toBe(false);
expect(validateEmail('@no-local.com')).toBe(false);
expect(validateEmail('user@')).toBe(false);
expect(validateEmail('user@.com')).toBe(false);
expect(validateEmail('user@ example.com')).toBe(false);
});
it('should handle null and undefined', () => {
expect(validateEmail(null as any)).toBe(false);
expect(validateEmail(undefined as any)).toBe(false);
});
it('should handle extremely long emails', () => {
const longLocal = 'a'.repeat(65) + '@example.com';
expect(validateEmail(longLocal)).toBe(false);
});
});
describe('validatePassword', () => {
it('should accept strong passwords', () => {
expect(validatePassword('Str0ng!Pass')).toEqual({      valid:
true });
expect(validatePassword('C0mpl3x#P@ss')).toEqual({     valid:
true });
});
it('should reject short passwords', () => {
const result = validatePassword('Ab1!');
expect(result.valid).toBe(false);
expect(result.errors).toContain('minimum 8 characters');
});
it('should require uppercase letter', () => {
const result = validatePassword('lowercase1!');
expect(result.valid).toBe(false);
expect(result.errors).toContain('uppercase letter required');
```

```
    });
    it('should require number', () => {
    const result = validatePassword('NoNumbers!');
    expect(result.valid).toBe(false);
    expect(result.errors).toContain('number required');
    });
    it('should require special character', () => {
    const result = validatePassword('NoSpecial1');
    expect(result.valid).toBe(false);
    expect(result.errors).toContain('special character
required');
    });
    });
```

Тесты с моками

```
    > Напиши unit-тесты для UserService. Замокай UserRepository
    и EmailService. Проверь:
    - Создание пользователя (happy path)
    - Дублирование email (должен бросить ConflictError)
    - Ошибка при отправке email (пользователь создаётся, но
    ошибка логируется)
```

Claude Code сгенерирует тесты с корректном использованием моков:

```
    import { describe, it, expect, vi, beforeEach } from 'vitest';
    import { UserService } from '../userService';
    import { UserRepository } from '../../repositories/
userRepository';
    import { EmailService } from '../emailService';
    vi.mock('../../repositories/userRepository');
    vi.mock('../emailService');
    describe('UserService', () => {
    let userService: UserService;
    let mockUserRepo: jest.Mocked<UserRepository>;
    let mockEmailService: jest.Mocked<EmailService>;
    beforeEach(() => {
    mockUserRepo = new UserRepository() as
jest.Mocked<UserRepository>;
    mockEmailService = new EmailService() as
jest.Mocked<EmailService>;
    userService = new UserService(mockUserRepo, mockEmailService);
    });
    describe('createUser', () => {
    const validInput = {
    email: 'john@example.com',
    name: 'John Doe',
    password: 'Str0ng!Pass',
```

```
    };
    it('should create user and send welcome email', async () => {
    mockUserRepo.findByEmail.mockResolvedValue(null);
    mockUserRepo.create.mockResolvedValue({
    id: '1',
    …validInput,
    createdAt: new Date(),
    });
    mockEmailService.sendWelcome.mockResolvedValue(undefined);
    const user = await userService.createUser(validInput);
    expect(user.id).toBe('1');
    expect(mockUserRepo.create).toHaveBeenCalledOnce();
    expect(mockEmailService.sendWelcome).toHaveBeenCalledWith(
    'john@example.com',
    'John Doe'
    );
    });
    it('should throw ConflictError if email exists', async () => {
    mockUserRepo.findByEmail.mockResolvedValue({
    id: '1',
    email: validInput.email,
    });
    await expect(userService.createUser(validInput))
    .rejects.toThrow('User with this email already exists');
    expect(mockUserRepo.create).not.toHaveBeenCalled();
    });
    it('should create user even if welcome email fails', async ()
=> {
    mockUserRepo.findByEmail.mockResolvedValue(null);
    mockUserRepo.create.mockResolvedValue({     id:     '1',     …
validInput });
    mockEmailService.sendWelcome.mockRejectedValue(
    new Error('SMTP error')
    );
    const user = await userService.createUser(validInput);
    expect(user.id).toBe('1');
    // Ошибка email не должна бросать исключение
    });
    });
    });
```

Параметризованные тесты

```
    > Напиши параметризованные тесты для функции calculateTax.
    Покрой все комбинации: разные ставки НДС, разные суммы,
    разные страны.

    describe('calculateTax', () => {
```

```
    it.each([
    { country: 'RU', amount: 1000, expected: 200 },
    { country: 'RU', amount: 0, expected: 0 },
    { country: 'US', amount: 1000, expected: 100 },
    { country: 'DE', amount: 1000, expected: 190 },
    { country: 'GB', amount: 1000, expected: 200 },
    { country: 'JP', amount: 1000, expected: 100 },
    ])('should calculate $expected tax for $country on $amount',
    ({ country, amount, expected }) => {
    expect(calculateTax(amount, country)).toBe(expected);
    }
    );
    it.each([
    { amount: -100, country: 'RU' },
    { amount: NaN, country: 'RU' },
    { amount: Infinity, country: 'RU' },
    ])('should throw for invalid amount: $amount',
    ({ amount, country }) => {
    expect(() => calculateTax(amount, country)).toThrow();
    }
    );
    });
```

9.3. Интеграционные тесты

Тестирование API endpoints

```
    > Напиши интеграционные тесты для API.
    Используй supertest + тестовую БД SQLite.
    Эндпоинты:
    - POST /api/products – создание
    - GET /api/products – список с пагинацией
    - GET /api/products/:id – по ID
    - PUT /api/products/:id – обновление
    - DELETE /api/products/:id – удаление
    Для каждого тестируй: success, validation errors,
    not found, unauthorized.
```

Claude Code создаст:

```
    import { describe, it, expect, beforeAll, afterAll,
beforeEach } from 'vitest';
    import request from 'supertest';
    import { app } from '../../src/app';
    import { db } from '../../src/db';
    describe('Products API', () => {
    let authToken: string;
```

```
beforeAll(async () => {
await db.migrate.latest();
// Создаём тестового пользователя
const res = await request(app)
.post('/api/auth/register')
.send({ email: 'test@test.com', password: 'Test1234!' });
authToken = res.body.token;
});
afterAll(async () => {
await db.destroy();
});
beforeEach(async () => {
await db('products').truncate();
});
describe('POST /api/products', () => {
const validProduct = {
name: 'Test Product',
price: 29.99,
category: 'electronics',
};
it('should create a product (201)', async () => {
const res = await request(app)
.post('/api/products')
.set('Authorization', `Bearer ${authToken}`)
.send(validProduct);
expect(res.status).toBe(201);
expect(res.body).toMatchObject({
id: expect.any(String),
name: 'Test Product',
price: 29.99,
});
});
it('should return 422 for invalid data', async () => {
const res = await request(app)
.post('/api/products')
.set('Authorization', `Bearer ${authToken}`)
.send({ name: '' }); // Missing required fields
expect(res.status).toBe(422);
expect(res.body.errors).toBeDefined();
});
it('should return 401 without auth token', async () => {
const res = await request(app)
.post('/api/products')
.send(validProduct);
expect(res.status).toBe(401);
});
});
describe('GET /api/products', () => {
beforeEach(async () => {
```

```
// Seed test data
await db('products').insert([
{ name: 'Product 1', price: 10 },
{ name: 'Product 2', price: 20 },
{ name: 'Product 3', price: 30 },
]);
});
it('should return paginated list', async () => {
const res = await request(app)
.get('/api/products?page=1&per_page=2')
.set('Authorization', `Bearer ${authToken}`);
expect(res.status).toBe(200);
expect(res.body.data).toHaveLength(2);
expect(res.body.pagination.total).toBe(3);
});
});
});
```

Тестирование с базой данных

```
> Напиши интеграционные тесты для OrderService,
которые проверяют реальное взаимодействие с PostgreSQL.
Используй Testcontainers для запуска PostgreSQL в Docker.
```

9.4. E2E-тестирование

Playwright

```
> Напиши E2E тесты с Playwright для flow регистрации:
1. Открыть /register
2. Заполнить форму (имя, email, пароль)
3. Нажать "Зарегистрироваться"
4. Проверить редирект на /dashboard
5. Проверить, что имя отображается в header
Дополнительно:
- Тест на невалидный email
- Тест на существующий email
- Тест на слабый пароль
```

Claude Code создаст:

```
import { test, expect } from '@playwright/test';
test.describe('Registration Flow', () => {
test('should register successfully', async ({ page }) => {
await page.goto('/register');
```

```
    await page.fill('[data-testid="name-input"]', 'John Doe');
    await                page.fill('[data-testid="email-input"]',
'john@example.com');
    await  page.fill('[data-testid="password-input"]',  'Str0ng!
Pass');
    await     page.fill('[data-testid="confirm-password-input"]',
'Str0ng!Pass');
    await page.click('[data-testid="register-button"]');
    // Wait for redirect
    await page.waitForURL('/dashboard');
    // Verify user info displayed
    const userName = page.locator('[data-testid="user-name"]');
    await expect(userName).toHaveText('John Doe');
    });
    test('should show error for invalid email', async ({ page })
=> {
    await page.goto('/register');
    await page.fill('[data-testid="name-input"]', 'John');
    await    page.fill('[data-testid="email-input"]',    'not-an-
email');
    await  page.fill('[data-testid="password-input"]',  'Str0ng!
Pass');
    await page.click('[data-testid="register-button"]');
    const error = page.locator('[data-testid="email-error"]');
    await expect(error).toBeVisible();
    await expect(error).toContainText('email');
    });
    test('should show error for weak password', async ({ page })
=> {
    await page.goto('/register');
    await page.fill('[data-testid="name-input"]', 'John');
    await                page.fill('[data-testid="email-input"]',
'john@test.com');
    await page.fill('[data-testid="password-input"]', '123');
    await page.click('[data-testid="register-button"]');
    const error = page.locator('[data-testid="password-error"]');
    await expect(error).toBeVisible();
    });
    test('should show error for existing email', async ({ page })
=> {
    // First registration
    await page.goto('/register');
    await page.fill('[data-testid="name-input"]', 'First User');
    await                page.fill('[data-testid="email-input"]',
'duplicate@test.com');
    await  page.fill('[data-testid="password-input"]',  'Str0ng!
Pass');
    await     page.fill('[data-testid="confirm-password-input"]',
'Str0ng!Pass');
```

```
    await page.click('[data-testid="register-button"]');
    await page.waitForURL('/dashboard');
    // Logout
    await page.click('[data-testid="logout-button"]');
    // Second registration with same email
    await page.goto('/register');
    await page.fill('[data-testid="name-input"]', 'Second User');
    await                page.fill('[data-testid="email-input"]',
'duplicate@test.com');
    await  page.fill('[data-testid="password-input"]',  'Str0ng!
Pass');
    await     page.fill('[data-testid="confirm-password-input"]',
'Str0ng!Pass');
    await page.click('[data-testid="register-button"]');
    const error = page.locator('[data-testid="form-error"]');
    await expect(error).toContainText('already exists');
    });
    });
```

9.5. TDD с Claude Code

Test-Driven Development приобретает новое измерение с Claude Code.

Цикл TDD: Red -> Green -> Refactor

```
> Напиши тест для функции formatCurrency(amount, currency):
formatCurrency(1234.5, 'RUB') -> '1 234,50 '
formatCurrency(1234.5, 'USD') -> '$1,234.50'
formatCurrency(1234.5, 'EUR') -> '1.234,50'
ТОЛЬКО тест, без реализации.
> Теперь напиши реализацию formatCurrency,
которая проходит все тесты. Запусти тесты.
> Тесты проходят. Теперь рефактори:
- Извлеки конфигурацию валют в отдельный объект
- Добавь поддержку локализации
- Оптимизируй для частых вызовов
Тесты должны продолжать проходить.
```

BDD с Cucumber/Gherkin

```
> Напиши BDD-сценарии для корзины покупок в Gherkin-формате:
Feature: Shopping Cart
Scenario: Add item to cart
Given I have an empty cart
When I add "MacBook Pro" with price 2499.99
```

```
Then my cart should contain 1 item
And cart total should be 2499.99
Реализуй step definitions для Cucumber.js.
```

9.6. Тестирование Python-проектов с pytest

```
> Напиши тесты для Django REST API (pytest + pytest-django).
Приложение: блог-платформа.
Модели: User, Article, Comment, Tag.
Viewsets: ArticleViewSet (CRUD + publish/unpublish).
Покрой:
1. Создание статьи (авторизованный, анонимный)
2. Редактирование (автор, не-автор)
3. Публикация (автор, не-автор)
4. Листинг (фильтрация по тегам, поиск, пагинация)
5. Комментарии (создание, удаление)
Используй fixtures, factory_boy, параметризацию.
```

Claude Code создаст структурированные тесты:

```
import pytest
from rest_framework.test import APIClient
from factory import Factory, Faker, SubFactory
from articles.models import Article, Tag
class UserFactory(Factory):
class Meta:
model = 'auth.User'
username = Faker('user_name')
email = Faker('email')
password = 'testpass123'
class ArticleFactory(Factory):
class Meta:
model = Article
title = Faker('sentence')
body = Faker('text', max_nb_chars=500)
author = SubFactory(UserFactory)
status = 'draft'
@pytest.fixture
def api_client():
return APIClient()
@pytest.fixture
def user(db):
return UserFactory.create()
@pytest.fixture
def auth_client(api_client, user):
api_client.force_authenticate(user=user)
return api_client
class TestArticleCreation:
```

```
    def test_create_article_authenticated(self, auth_client,
user):
    data = {'title': 'Test Article', 'body': 'Content here'}
    response = auth_client.post('/api/articles/', data)
    assert response.status_code == 201
    assert response.data['title'] == 'Test Article'
    assert response.data['author']['id'] == user.id
    assert response.data['status'] == 'draft'
    def test_create_article_anonymous(self, api_client):
    data = {'title': 'Test', 'body': 'Content'}
    response = api_client.post('/api/articles/', data)
    assert response.status_code == 401
    @pytest.mark.parametrize('field,value,error', [
    ('title', '', 'This field may not be blank.'),
    ('title', 'x' * 201, 'at most 200 characters'),
    ('body', '', 'This field may not be blank.'),
    ])
    def test_validation_errors(self, auth_client, field, value,
error):
    data = {'title': 'Valid Title', 'body': 'Valid Body'}
    data[field] = value
    response = auth_client.post('/api/articles/', data)
    assert response.status_code == 400
    assert error in str(response.data[field])
```

9.7. Покрытие кода и метрики качества

Настройка coverage

```
> Настрой измерение покрытия кода:
- vitest с v8 coverage provider
- Минимальное покрытие: 80% (lines, branches, functions)
- Исключить: тесты, конфигурацию, типы, миграции
- Генерировать отчёт: lcov + html
- Добавить скрипт в package.json
- Настроить CI для проверки coverage
```

Увеличение покрытия

```
> Текущее покрытие: 45%. Помоги довести до 80%.
Покажи файлы с наименьшим покрытием и напиши
для них тесты, начиная с наиболее критичных.
```

Mutation testing

```
> Настрой mutation testing с Stryker.
Покажи, насколько наши тесты действительно эффективны
(не просто покрывают строки, а ловят ошибки).
```

9.8. Генерация тестовых данных

```
> Создай фабрики тестовых данных для модели Order:
Order:
id: UUID
customer: Customer
items: OrderItem[]
status: 'pending' | 'confirmed' | 'shipped' | 'delivered'
total: number
address: Address
created_at: Date
Фабрика должна:
- Генерировать реалистичные данные (faker.js)
- Поддерживать overrides
- Создавать связанные объекты (customer, items)
- Иметь специализированные варианты:
orderFactory.pending(), orderFactory.shipped(), etc.
```

9.9. Snapshot тестирование

```
> Настрой snapshot-тесты для React компонентов.
Для каждого компонента в src/components/ui/:
- Создай snapshot для default state
- Snapshot с разными props-вариациями
- Snapshot для loading и error states
Используй inline snapshots где возможно.
```

Итоги главы

В этой главе мы рассмотрели:

– Философию тестирования с AI и иерархию тестов
– Генерацию unit-тестов с моками и параметризацией
– Интеграционные тесты для API с supertest и Testcontainers
– E2E-тестирование с Playwright
– TDD-цикл с Claude Code: Red -> Green -> Refactor
– Тестирование Python-проектов с pytest
– Настройку покрытия кода и mutation testing

– Генерацию тестовых данных и snapshot-тестирование

В следующей главе мы рассмотрим работу с базами данных: от проектирования схем до оптимизации запросов.

Глава 10. Работа с базами данных

10.1. Проектирование схем данных

Проектирование базы данных – одна из тех областей, где Claude Code может сэкономить часы работы. Он знает нормальные формы, типичные паттерны и антипаттерны, а также специфику разных СУБД.

От требований к схеме

```
> Спроектируй базу данных для системы бронирования отелей:
Бизнес-требования:
- Отели с номерами разных категорий
- Гости с историей бронирований
- Бронирования с датами заезда/выезда
- Дополнительные услуги (завтрак, трансфер, SPA)
- Отзывы гостей
- Ценообразование зависит от сезона и категории номера
Требования:
- PostgreSQL 16
- Нормализация до 3NF
- Индексы для типичных запросов
- Soft delete для ключевых таблиц
- Аудит изменений
```

Claude Code создаст продуманную схему:

```
-- Отели
CREATE TABLE hotels (
id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
name VARCHAR(200) NOT NULL,
slug VARCHAR(200) UNIQUE NOT NULL,
description TEXT,
city VARCHAR(100) NOT NULL,
country VARCHAR(100) NOT NULL,
address TEXT NOT NULL,
latitude DECIMAL(10, 8),
longitude DECIMAL(11, 8),
star_rating SMALLINT CHECK (star_rating BETWEEN 1 AND 5),
amenities JSONB DEFAULT '[]',
check_in_time TIME DEFAULT '14:00',
check_out_time TIME DEFAULT '12:00',
is_active BOOLEAN DEFAULT true,
created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
```

```
updated_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);
-- Категории номеров
CREATE TABLE room_categories (
id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
hotel_id UUID NOT NULL REFERENCES hotels(id),
name VARCHAR(100) NOT NULL,
description TEXT,
max_guests SMALLINT NOT NULL DEFAULT 2,
area_sqm DECIMAL(6, 2),
amenities JSONB DEFAULT '[]',
photo_urls TEXT[] DEFAULT '{}',
UNIQUE (hotel_id, name)
);
-- Конкретные номера
CREATE TABLE rooms (
id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
category_id UUID NOT NULL REFERENCES room_categories(id),
room_number VARCHAR(20) NOT NULL,
floor SMALLINT,
is_available BOOLEAN DEFAULT true,
notes TEXT,
UNIQUE (category_id, room_number)
);
-- Сезонное ценообразование
CREATE TABLE pricing_seasons (
id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
hotel_id UUID NOT NULL REFERENCES hotels(id),
name VARCHAR(100) NOT NULL,
start_date DATE NOT NULL,
end_date DATE NOT NULL,
multiplier DECIMAL(4, 2) DEFAULT 1.00,
CHECK (end_date > start_date)
);
CREATE TABLE room_prices (
id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
category_id UUID NOT NULL REFERENCES room_categories(id),
season_id UUID REFERENCES pricing_seasons(id),
price_per_night DECIMAL(10, 2) NOT NULL,
currency VARCHAR(3) DEFAULT 'RUB',
UNIQUE (category_id, season_id)
);
-- Гости
CREATE TABLE guests (
id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
email VARCHAR(255) UNIQUE NOT NULL,
phone VARCHAR(20),
first_name VARCHAR(100) NOT NULL,
last_name VARCHAR(100) NOT NULL,
```

```
    passport_number VARCHAR(50),
    date_of_birth DATE,
    loyalty_tier VARCHAR(20) DEFAULT 'standard'
    CHECK (loyalty_tier IN ('standard', 'silver', 'gold',
'platinum')),
    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    deleted_at TIMESTAMPTZ - soft delete
    );
    -- Бронирования
    CREATE TABLE bookings (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    booking_number VARCHAR(20) UNIQUE NOT NULL,
    guest_id UUID NOT NULL REFERENCES guests(id),
    room_id UUID NOT NULL REFERENCES rooms(id),
    check_in DATE NOT NULL,
    check_out DATE NOT NULL,
    guests_count SMALLINT DEFAULT 1,
    status VARCHAR(20) DEFAULT 'pending'
    CHECK (status IN ('pending', 'confirmed', 'checked_in',
    'checked_out', 'cancelled')),
    total_price DECIMAL(10, 2) NOT NULL,
    currency VARCHAR(3) DEFAULT 'RUB',
    special_requests TEXT,
    cancelled_at TIMESTAMPTZ,
    cancellation_reason TEXT,
    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    CHECK (check_out > check_in)
    );
    -- Индексы
    CREATE INDEX idx_bookings_dates ON bookings(check_in,
check_out);
    CREATE INDEX idx_bookings_guest ON bookings(guest_id);
    CREATE INDEX idx_bookings_room_dates ON bookings(room_id,
check_in, check_out);
    CREATE INDEX idx_bookings_status ON bookings(status);
    CREATE INDEX idx_hotels_city ON hotels(city);
    CREATE INDEX idx_guests_email ON guests(email);
    CREATE INDEX idx_guests_not_deleted ON guests(id) WHERE
deleted_at IS NULL;
```

Prisma Schema

```
    > Перепиши эту схему как Prisma schema для TypeScript-проекта.
    Добавь отношения, валидацию и seed-данные.
```

Миграции

```
> Создай миграции для добавления системы отзывов:
- Таблица reviews (guest_id, hotel_id, booking_id, rating, text)
- Средний рейтинг в таблице hotels (materialized)
- Триггер для автообновления среднего рейтинга
Используй knex migrations.
```

10.2. ORM и Query Builder

Prisma

```
> Напиши Prisma-запросы для дашборда отеля:
1. Список бронирований на сегодня (check-in и check-out)
2. Заполняемость за последний месяц (в процентах)
3. Топ-5 гостей по количеству бронирований
4. Доход по категориям номеров за месяц
5. Номера, доступные для бронирования на указанные даты
```

Claude Code создаст эффективные запросы:

```
// 5. Доступные номера на указанные даты
async function getAvailableRooms(
hotelId: string,
checkIn: Date,
checkOut: Date,
guests: number
) {
const bookedRoomIds = await prisma.booking.findMany({
where: {
room: { category: { hotelId } },
status: { notIn: ['cancelled'] },
OR: [
{ checkIn: { lt: checkOut }, checkOut: { gt: checkIn } },
],
},
select: { roomId: true },
});
const bookedIds = bookedRoomIds.map(b => b.roomId);
return prisma.room.findMany({
where: {
category: {
hotelId,
maxGuests: { gte: guests },
```

```
},
isAvailable: true,
id: { notIn: bookedIds },
},
include: {
category: {
include: {
roomPrices: {
where: {
season: {
startDate: { lte: checkIn },
endDate: { gte: checkIn },
},
},
},
},
},
},
orderBy: { category: { name: 'asc' } },
});
}
```

SQLAlchemy (Python)

```
> Напиши SQLAlchemy модели и запросы для системы
управления задачами. Используй:
- Declarative base с аннотациями типов (SQLAlchemy 2.0)
- Async session
- Отношения с lazy/eager loading
- Составные индексы
```

TypeORM

```
> Реализуй repository pattern с TypeORM для модели Product.
Включи:
- Поиск с фильтрацией, сортировкой и пагинацией
- Soft delete
- Кэширование query results
- Транзакции для сложных операций
- Query builder для гибких запросов
```

10.3. Оптимизация запросов

Анализ и оптимизация

```
> Вот медленный запрос (выполняется 12 секунд):
SELECT u.*,
COUNT(DISTINCT o.id) as order_count,
SUM(oi.price * oi.quantity) as total_spent,
MAX(o.created_at) as last_order_date
FROM users u
LEFT JOIN orders o ON o.user_id = u.id
LEFT JOIN order_items oi ON oi.order_id = o.id
WHERE u.created_at > '2024-01-01'
GROUP BY u.id
HAVING COUNT(DISTINCT o.id) > 5
ORDER BY total_spent DESC
LIMIT 100;
Таблицы: users (2M rows), orders (10M rows),
order_items (50M rows).
Оптимизируй. Покажи EXPLAIN ANALYZE и предложенные индексы.
```

Claude Code предложит:

```
-- 1. Добавить индексы
CREATE INDEX idx_orders_user_created ON orders(user_id,
created_at);
CREATE INDEX idx_order_items_order ON order_items(order_id)
INCLUDE (price, quantity);
CREATE INDEX idx_users_created ON users(created_at);
-- 2. Оптимизированный запрос с CTE
WITH active_users AS (
SELECT id FROM users WHERE created_at > '2024-01-01'
),
user_orders AS (
SELECT
o.user_id,
COUNT(*) as order_count,
MAX(o.created_at) as last_order_date,
array_agg(o.id) as order_ids
FROM orders o
JOIN active_users au ON o.user_id = au.id
GROUP BY o.user_id
HAVING COUNT(*) > 5
),
user_totals AS (
SELECT
uo.user_id,
```

```
    uo.order_count,
    uo.last_order_date,
    SUM(oi.price * oi.quantity) as total_spent
    FROM user_orders uo
    JOIN order_items oi ON oi.order_id = ANY(uo.order_ids)
    GROUP BY uo.user_id, uo.order_count, uo.last_order_date
    )
    SELECT u.*, ut.order_count, ut.total_spent, ut.last_order_date
    FROM users u
    JOIN user_totals ut ON u.id = ut.user_id
    ORDER BY ut.total_spent DESC
    LIMIT 100;
```

N+1 проблема

```
    > Найди и устрани N+1 запросы в файле src/services/
orderService.ts.
    Покажи до и после с количеством SQL-запросов.
```

Партицирование

```
    > Таблица events (500M+ записей) стала тормозить.
    Предложи стратегию партицирования.
    Типичные запросы:
    - По дате (последние 30 дней)
    - По tenant_id + дате
    PostgreSQL 16. Покажи DDL для партицирования.
```

10.4. Миграции

Создание миграций

```
    > Создай миграцию для следующих изменений:
    1. Добавить таблицу subscriptions
    2. Добавить колонку subscription_id в users
    3. Мигрировать данные из users.plan_type в subscriptions
    4. Удалить колонку plan_type из users
    Используй Prisma migrate.
    Убедись, что миграция обратима (down).
```

Миграция данных

```
    > Нужно мигрировать данные при изменении структуры:
```

```
Было: address (TEXT) - полный адрес одной строкой
Стало: street, city, postal_code, country - отдельные поля
1. Создай миграцию с новыми колонками
2. Напиши скрипт для парсинга адресов и заполнения
новых колонок
3. Удали старую колонку
Данных: ~100,000 записей. Некоторые адреса невалидны.
Невалидные - в таблицу failed_migrations для ручной обработки.
```

10.5. Redis и кэширование

```
> Реализуй слой кэширования через Redis:
1. Кэш для отдельных объектов: get/set с TTL
2. Кэш для списков с инвалидацией
3. Cache-aside pattern
4. Rate limiting на базе Redis
5. Distributed locks (Redlock)
Обёртка: RedisCacheService
Сериализация: JSON
Инвалидация: по ключу, по паттерну, по тегам
```

Стратегии инвалидации

```
> Реализуй инвалидацию кэша с использованием тегов:
Пример:
- cache.set('user:1', data, { tags: ['user:1', 'users'] })
- cache.set('user:1:orders', orders, { tags: ['user:1',
'orders'] })
- cache.invalidate('user:1') -> удаляет оба ключа
- cache.invalidate('users') -> удаляет только 'user:1'
```

10.6. MongoDB

```
> Спроектируй MongoDB-схему для CMS:
Коллекции:
- articles: с embedded comments, tags
- users: с встроенными настройками
- media: метаданные файлов
Индексы: текстовый поиск, compound для фильтрации.
Агрегации: статистика по авторам, популярные теги.
Покажи Mongoose-схемы и типичные запросы.
```

10.7. Безопасность баз данных

```
> Проведи аудит безопасности запросов к БД:
1. Найди SQL-инъекции (строковая конкатенация в запросах)
2. Проверь использование параметризованных запросов
3. Проверь права доступа (principle of least privilege)
4. Найди sensitive данные без шифрования
5. Проверь настройки connection string (SSL, таймауты)
```

Итоги главы

В этой главе мы рассмотрели:

– Проектирование схем данных: от требований к SQL и Prisma
– Работу с ORM: Prisma, SQLAlchemy, TypeORM
– Оптимизацию запросов: индексы, N+1, партицирование
– Миграции: создание, миграция данных, обратимость
– Redis и стратегии кэширования
– MongoDB и документоориентированные БД
– Безопасность баз данных

В следующей главе мы перейдём к документированию кода и проекта.

Глава 11. Документирование кода и проекта

11.1. Документация как часть продукта

Документация часто рассматривается как «побочный продукт» разработки – нечто, что делается в последнюю очередь (или никогда). Claude Code радикально меняет эту ситуацию: генерация документации из кода становится настолько простой, что у лени не остаётся оправданий.

Хорошая документация – это не описание того, ЧТО делает код (это и так видно), а объяснение ПОЧЕМУ он это делает и КАК его использовать. Claude Code понимает эту разницу и генерирует осмысленную документацию, а не бесполезные комментарии типа `// increment counter` над строкой `counter++`.

11.2. JSDoc и TSDoc

Автоматическая генерация

```
> Добавь JSDoc-комментарии ко всем экспортируемым функциям
и классам в каталоге src/services/.
Для каждой функции:
- @description – что делает (на русском)
- @param – описание каждого параметра
- @returns – что возвращает
- @throws – какие исключения может бросить
- @example – пример использования
```

Claude Code проанализирует каждую функцию и создаст осмысленные комментарии:

```
/**
* Создаёт новый заказ и инициирует процесс оплаты.
*
* Проверяет наличие товаров на складе, рассчитывает итоговую
* стоимость с учётом скидок и промокодов, создаёт запись
* в базе данных и отправляет запрос на оплату через Stripe.
*
* @param input – Данные для создания заказа
* @param input.customerId – ID покупателя
* @param input.items – Массив товаров с количеством
* @param input.promoCode – Промокод (опционально)
* @param input.deliveryAddress – Адрес доставки
*
* @returns Созданный заказ с ID и статусом оплаты
```

```
*
* @throws {InsufficientStockError} Если товара нет на складе
* @throws {InvalidPromoCodeError} Если промокод недействителен
* @throws {PaymentError} Если оплата не прошла
*
* @example
* ```typescript
* const order = await orderService.createOrder({
* customerId: 'cust_123',
* items: [{ productId: 'prod_456', quantity: 2 }],
* promoCode: 'SAVE10',
* deliveryAddress: { city: 'Москва', street: '…' }
* });
* console.log(order.id); // 'ord_789'
* console.log(order.status); // 'pending_payment'
* ```
*/
async createOrder(input: CreateOrderInput): Promise<Order> {
// ...
}
```

Python Docstrings

```
> Добавь Google-style docstrings ко всем функциям
в модуле app/services/analytics.py.
Формат:
- Описание на русском
- Args: с типами и описанием
- Returns: описание возвращаемого значения
- Raises: возможные исключения
- Examples: doctests

def calculate_retention(
cohort_date: date,
periods: int = 7,
granularity: str = 'day'
) -> RetentionReport:
"""Рассчитывает retention когорты пользователей.
Анализирует возвращаемость пользователей, зарегистрированных
в указанную дату, за указанное количество периодов.
Поддерживает дневную, недельную и месячную гранулярность.
Args:
cohort_date: Дата регистрации когорты.
periods: Количество периодов для анализа (по умолчанию 7).
granularity: Гранулярность: 'day', 'week' или 'month'.
Returns:
RetentionReport с полями:
- cohort_size: размер когорты
```

```
    - retention_rates: список процентов retention по периодам
    - active_users: список количества активных пользователей
    Raises:
    ValueError: Если granularity не в ['day', 'week', 'month'].
    CohortNotFoundError: Если на указанную дату нет пользователей.
    Examples:
    >>> report = calculate_retention(date(2024, 1, 1), periods=3)
    >>> report.cohort_size
    150
    >>> report.retention_rates
    [100.0, 45.3, 32.1]
    """
```

11.3. README-файлы

README для проекта

```
    > Сгенерируй README.md для этого проекта.
    Включи:
    - Название и описание (бейджи: CI, coverage, license)
    - Скриншоты / GIF демо
    - Быстрый старт (3-5 шагов)
    - Требования к системе
    - Установка и настройка
    - Основные команды разработки
    - Структура проекта
    - API документация (кратко + ссылка на Swagger)
    - Деплой
    - Contributing guidelines
    - License
```

Claude Code создаст профессиональный README:

```
    ![CI](https://github.com/username/hotel-booking/workflows/CI/
badge.svg)
    ![Coverage](https://img.shields.io/codecov/c/github/username/
hotel-booking)
    ![License](https://img.shields.io/github/license/username/
hotel-booking)
    Система бронирования отелей с REST API, real-time
    уведомлениями и интеграцией с платёжными системами.
    ## Быстрый старт

    git clone https://github.com/username/hotel-booking.git
    cd hotel-booking

    pnpm install
```

cp .env.example .env

docker compose up -d db redis

pnpm dev

```
API доступен на <http://localhost:3000>
Swagger UI: <http://localhost:3000/api/docs>
...
```

README для библиотеки

```
> Создай README для npm-пакета: библиотека для валидации
российских документов (паспорт, ИНН, СНИЛС, ОГРН).
Формат: как у популярных npm-пакетов (zod, joi).
Включи: установку, API reference, примеры для каждого
типа документа, TypeScript типы.
```

11.4. API-документация

OpenAPI / Swagger

```
> Создай OpenAPI 3.1 спецификацию для нашего REST API.
Базовый URL: /api/v1
Auth: Bearer JWT
Ресурсы: пользователи, проекты, задачи
Для каждого endpoint:
- Summary и description
- Параметры запроса
- Request body schema
- Response schemas (200, 400, 401, 404, 500)
- Примеры
Формат: YAML
```

Claude Code создаст полную спецификацию:

```
openapi: 3.1.0
info:
title: TaskManager API
version: 1.0.0
description: |
API для управления проектами и задачами.
## Аутентификация
Используйте Bearer JWT токен. Получите токен через
```

```
`POST /api/v1/auth/login`.
contact:
email: api@example.com
servers:
- url: https://api.example.com/api/v1
description: Production
- url: http://localhost:3000/api/v1
description: Development
security:
- bearerAuth: []
paths:
/projects:
get:
summary: Список проектов
description: Возвращает список проектов текущего пользователя
tags: [Projects]
parameters:
- name: page
in: query
schema:
type: integer
default: 1
- name: per_page
in: query
schema:
type: integer
default: 20
maximum: 100
- name: status
in: query
schema:
type: string
enum: [active, archived]
responses:
'200':
description: Список проектов
content:
application/json:
schema:
$ref: '#/components/schemas/ProjectList'
example:
data:
- id: "proj_1"
name: "Website Redesign"
status: "active"
tasksCount: 42
pagination:
total: 5
page: 1
```

```
perPage: 20
```

AsyncAPI для WebSocket

```
> Создай AsyncAPI спецификацию для WebSocket-событий:
Events (server -> client):
- task.created: новая задача
- task.updated: обновление задачи
- task.deleted: удаление задачи
- notification: уведомление пользователю
Commands (client -> server):
- subscribe: подписка на канал
- unsubscribe: отписка от канала
```

11.5. Architecture Decision Records (ADR)

```
> Создай ADR для решения об использовании PostgreSQL
вместо MongoDB.
Формат: стандартный ADR (Title, Status, Context, Decision,
Consequences)
```

Claude Code создаст структурированный ADR:

```
## Статус
Принято (2024-01-15)
## Контекст
Нам нужно выбрать СУБД для нашей системы бронирования отелей.
Основные требования:
- Строгая консистентность данных (бронирования, платежи)
- Сложные запросы с JOIN-ами (отчёты, аналитика)
- Полнотекстовый поиск (каталог отелей)
- Геопространственные запросы (поиск ближайших отелей)
- ACID-транзакции (атомарные операции с бронированиями)
Рассматривались: PostgreSQL, MongoDB, MySQL.
## Решение
Используем PostgreSQL 16.
Основания:
1. ACID-транзакции критичны для финансовых операций
2. Встроенный полнотекстовый поиск (FTS)
3. PostGIS для геопространственных запросов
4. JSONB для гибких структур данных (amenities, metadata)
5. Mature экосистема инструментов (Prisma, pg-promise)
6. Богатый набор типов данных (UUID, TSTZRANGE, INTERVAL)
## Последствия
### Положительные
- Строгая типизация на уровне БД
- Мощные возможности индексирования
```

```
- Проверенная надёжность в production
- Хорошая поддержка в Prisma ORM
### Отрицательные
- Горизонтальное масштабирование сложнее, чем в MongoDB
- Менее гибкая схема данных
- Требуются миграции при изменении схемы
### Риски
- При >10M бронирований может потребоваться партицирование
- Митигация: настроить мониторинг производительности
и подготовить план партицирования заранее
```

11.6. Генерация CHANGELOG

```
> Сгенерируй CHANGELOG.md на основе git-истории:
Format: Keep a Changelog
Группировка: Added, Changed, Deprecated, Removed, Fixed,
Security
Период: последние 3 месяца
Язык: русский
Для каждой записи: описание + ссылка на PR/issue.
```

11.7. Диаграммы и визуализации

```
> Создай Mermaid-диаграммы для документации:
1. Entity Relationship Diagram для базы данных
2. Sequence Diagram для процесса оформления заказа
3. Architecture Diagram (C4 model, level 2)
4. State Diagram для статусов заказа
```

Claude Code создаст, например:

```
> Sequence Diagram для оформления заказа

sequenceDiagram
actor User
participant Web as Frontend
participant API as Backend API
participant DB as PostgreSQL
participant Stripe as Stripe API
participant Email as Email Service
User->>Web: Нажимает "Оформить заказ"
Web->>API: POST /api/orders
API->>DB: Проверить наличие товаров
DB->>API: Товары в наличии
API->>DB: Создать заказ (status: pending)
DB->>API: Order created
API->>Stripe: Create PaymentIntent
```

```
    Stripe->>API: client_secret
    API->>Web: { orderId, clientSecret }
    Web->>Stripe: Confirm Payment
    Stripe->>Web: Payment succeeded
    Web->>API: POST /api/orders/{id}/confirm
    API->>DB: Update status: paid
    API->>DB: Уменьшить stock
    API->>Email: Отправить подтверждение
    API->>Web: { status: "confirmed" }
    Web->>User: "Заказ оформлен!"
```

11.8. Inline-документация проектных решений

```
    > Добавь комментарии, объясняющие БИЗНЕС-ЛОГИКУ (не код)
    в файле src/services/pricing.ts.
    Пример хорошего комментария:
    // Скидка 15% для VIP-клиентов - маркетинговое решение от
2024-01-15
    // Согласовано с коммерческим директором (Jira: MARK-234)
    Это НЕ тот тип комментариев:
    // Вычисляем скидку (<- бесполезный)
```

11.9. Автоматизация документирования

Генерация документации из кода

```
    > Настрой автогенерацию API-документации:
    1. TypeDoc для TypeScript -> HTML документация
    2. Настроить GitHub Pages для хостинга
    3. CI: автообновление документации при merge в main
    4. Добавить ссылку в README
```

Проверка актуальности документации

```
    > Создай скрипт, который проверяет:
    1. Все экспортируемые функции имеют JSDoc
    2. JSDoc параметры совпадают с реальными параметрами
    3. README ссылки не битые
    4. Примеры кода в документации компилируются
    Добавь в CI как проверку.
```

Итоги главы

В этой главе мы рассмотрели:

– Генерацию JSDoc/TSDoc и Python Docstrings
– Создание профессиональных README
– API-документацию: OpenAPI, AsyncAPI
– Architecture Decision Records
– Генерацию CHANGELOG
– Mermaid-диаграммы для визуализации
– Inline-документирование бизнес-логики
– Автоматизацию документирования через CI

В следующей главе мы завершим вторую часть книги, рассмотрев работу с Git и совместную разработку.

Глава 12. Git и совместная разработка

12.1. Git-операции через Claude Code

Claude Code – полноценный Git-клиент. Он может выполнять все операции с Git: от создания коммитов до разрешения конфликтов слияния. Это особенно ценно для разработчиков, которые не помнят каждую команду Git наизусть.

Базовые операции

```
> Создай коммит для текущих изменений.
Формат: conventional commits.
Определи тип (feat/fix/refactor/docs) по содержанию diff.
```

Claude Code проанализирует `git diff`, определит характер изменений и создаст осмысленный коммит:

```
git add -A
git commit -m "feat(auth): add JWT refresh token rotation
- Implement automatic token refresh on 401 response
- Add refresh token storage in httpOnly cookie
- Add /api/auth/refresh endpoint
- Add token blacklisting for invalidated refresh tokens
Closes #142"
```

Комплексные git-операции

```
> У меня незакоммиченные изменения в трёх разных фичах.
Разбей их на отдельные коммиты:
1. Изменения в auth/ -> feat(auth): …
2. Исправления в api/ -> fix(api): …
3. Обновление конфигурации -> chore(config): …
```

Claude Code использует `git add -p` или selective staging для каждой группы изменений.

Работа с ветками

```
> Создай feature branch от develop, перенеси туда
последние 3 коммита из main и создай PR.
```

Отмена изменений

```
> Откати последний коммит, но сохрани изменения в working tree.
> Отмени push последнего коммита в remote (безопасно, через revert).
```

12.2. Разрешение конфликтов слияния

Конфликты слияния – одна из наиболее стрессовых ситуаций в разработке. Claude Code превращает их в рутину.

```
> У меня конфликт слияния в 5 файлах при сливании
feature/user-profiles в develop. Разреши конфликты:
- Сохрани функциональность обеих веток
- Если есть противоречия, выбери версию из feature branch
- Объясни каждое решение
```

Claude Code исследует конфликтные маркеры (<<<<<<<, =======, >>>>>>>), проанализирует контекст и примет обоснованные решения:

```
Файл: src/services/userService.ts
Конфликт в функции updateProfile():
- main: добавлен параметр `avatarUrl`
- feature: добавлен параметр `bio`
-> Решение: сохраняю ОБА параметра (дополняющие изменения)
Конфликт в функции deleteUser():
- main: мягкое удаление (soft delete)
- feature: жёсткое удаление (hard delete)
-> Решение: оставляю soft delete из main (безопаснее),
добавляю параметр `force: boolean` для hard delete
```

Сложные конфликты

```
> Конфликт слияния затрагивает одну и ту же функцию,
которая была отрефакторена в обеих ветках по-разному.
В main: функция разбита на 3 подфункции
В feature: функция полностью переписана
Помоги принять решение и разрешить конфликт.
```

12.3. Pull Request workflow

Создание PR

```
> Создай pull request:
- Source: feature/payment-integration
- Target: develop
- Title: по conventional commits
- Description: что сделано, как тестировать, скриншоты
- Labels: feature, needs-review
- Reviewers: @teammate1, @teammate2
```

Claude Code создаст детальное описание PR:

```
## Интеграция с платёжной системой
### Что сделано
- Интеграция с Stripe API (создание и подтверждение платежей)
- Webhook-обработчик для статусов платежей
- Страница оплаты с формой карты (Stripe Elements)
- Retry-логика для неуспешных платежей (3 попытки)
### Как тестировать
1. Запустить `pnpm dev`
2. Перейти на /checkout
3. Использовать тестовую карту: 4242 4242 4242 4242
4. Проверить, что заказ переходит в статус "paid"
5. Для тестирования ошибки: карта 4000 0000 0000 0002
### Чеклист
- [x] Unit-тесты
- [x] Интеграционные тесты
- [x] E2E тест (Playwright)
- [x] Документация API обновлена
- [x] ENV-переменные добавлены в .env.example
- [ ] Протестировано в staging
### Migration
[!] Требуется миграция: `pnpm db:migrate`
Добавлена таблица `payments` и колонка `order.payment_id`
```

Code Review с Claude Code

```
> Проведи code review для PR #42.
Посмотри все изменённые файлы и оставь комментарии:
- [!] Critical – баги, уязвимости
- [~] Warning – потенциальные проблемы
- [i] Suggestion – улучшения
- [+] Good – хорошие практики
```

12.4. Управление git-историей

Интерактивный rebase

```
> Я сделал 15 коммитов в feature branch, но многие из них —
промежуточные (WIP, fix typo, oops). Помоги привести
историю в порядок:
- Объедини связанные коммиты (squash)
- Перефразируй сообщения
- Результат: 3-5 логичных коммитов
```

Git blame и археология

```
> Эта функция работает неправильно. Помоги найти:
1. Когда она была написана (git blame)
2. С каким PR она была влита
3. Какие изменения были с тех пор
4. Кто менял и почему
```

Cherry-pick

```
> Из ветки feature/experiments нужно перенести в main
только коммиты, связанные с оптимизацией кэширования
(3 конкретных коммита). Остальное не переносить.
```

12.5. GitFlow и стратегии ветвления

Настройка GitFlow

```
> Настрой GitFlow для проекта:
Ветки:
- main - продакшен
- develop - разработка
- feature/* - фичи
- release/* - подготовка релиза
- hotfix/* - экстренные исправления
Создай:
1. Защиту веток (branch protection rules)
2. CODEOWNERS файл
3. PR template
```

```
4. Issue templates (bug, feature, task)
```

Trunk-based development

```
> Настрой trunk-based development:
- Короткоживущие feature branches (< 2 дня)
- Feature flags для незавершённых фич
- Автоматический merge при green CI
- Релиз через git tags
```

12.6. Автоматизация Git-хуков

```
> Настрой Git hooks через Husky + lint-staged:
pre-commit:
- ESLint --fix для изменённых файлов
- Prettier для изменённых файлов
- TypeScript type check
commit-msg:
- Проверка формата conventional commits
pre-push:
- Запуск тестов
- Проверка coverage (>= 80%)
```

Claude Code создаст все необходимые конфигурации:

```
// package.json
{
"lint-staged": {
"*.{ts,tsx}": [
"eslint --fix",
"prettier --write"
],
"*.{json,md}": [
"prettier --write"
]
}
}

// .husky/commit-msg
#!/usr/bin/env sh
. "$(dirname -- "$0")/_/husky.sh"
npx --no -- commitlint --edit ${1}
```

12.7. Monorepo и Multi-repo

Monorepo с Turborepo

```
> Настрой monorepo с Turborepo для проекта:
Пакеты:
- apps/web – Next.js фронтенд
- apps/api – Express бэкенд
- packages/ui – Shared UI компоненты
- packages/types – Shared TypeScript типы
- packages/config – Shared конфигурации (ESLint, TS)
Создай:
- turbo.json с pipeline
- Shared tsconfig
- Workspace dependencies
- CI для параллельной сборки
```

Submodules

```
> Объясни, как работают git submodules.
Нужно подключить shared-library из другого репозитория.
Покажи: добавление, обновление, клонирование с submodules.
```

12.8. Решение типичных проблем с Git

«Случайно закоммитил секрет»

```
> Я случайно закоммитил API-ключ в .env файл и запушил.
Как безопасно удалить его из истории?
```

Claude Code покажет использование `git filter-branch` или `BFG Repo Cleaner`, а также напомнит об отзыве скомпрометированного ключа.

«Запутался в ветках»

```
> У меня 20 локальных веток, некоторые уже вмержены.
Покажи статус каждой и удали уже вмерженые ветки.
```

«Нужно отменить merge»

```
> Вмержили в develop ветку с багами. Как безопасно
отменить merge, сохранив возможность смержить
исправленную ветку позже?
```

12.9. Совместная разработка: практические сценарии

Сценарий 1: Onboarding нового разработчика

```
> Я новый разработчик на проекте. Подготовь для меня:
1. Инструкцию по настройке окружения
2. Объяснение архитектуры проекта
3. Список первых задач для знакомства с кодом
4. Кто отвечает за какие модули (посмотри CODEOWNERS)
```

Сценарий 2: Код, написанный Claude Code, в командной разработке

```
> Я хочу предложить команде использовать Claude Code.
Какие правила и соглашения нужно установить, чтобы
AI-сгенерированный код не снижал качество проекта?
```

Claude Code может предложить:

– **AI-authored code review** – код от AI проходит тот же review, что и человеческий
– **Тестовое покрытие** – минимум 80% для AI-кода
– **Chunking** – AI не должен менять больше 200 строк за раз
– **CLAUDE.md** – единый файл для всей команды

Сценарий 3: Координация параллельной работы

```
> Мы с коллегой работаем над связанными фичами:
- Я делаю бэкенд (API endpoints)
- Коллега делает фронтенд (UI)
Помоги определить контракт (API interface),
чтобы мы могли работать параллельно.
```

Итоги главы

В этой главе мы рассмотрели:

– Git-операции через Claude Code: коммиты, ветки, отмена
– Разрешение конфликтов слияния
– Pull Request workflow: создание, review, описание
– Управление историей: rebase, blame, cherry-pick
– Стратегии ветвления: GitFlow и Trunk-based
– Автоматизацию через Git hooks
– Monorepo и submodules
– Решение типичных проблем с Git
– Совместную разработку с Claude Code в команде

С этой главой мы завершаем вторую часть книги – «Практическое использование». Вы научились писать код, рефакторить, отлаживать, тестировать, работать с базами данных, документировать и управлять Git-операциями с помощью Claude Code. В третьей части мы перейдём к продвинутым возможностям: MCP-серверы, автоматизация, CI/CD и работа с инфраструктурой.

Часть III. Продвинутые возможности

Глава 13. MCP: Model Context Protocol

13.1. Что такое MCP

Model Context Protocol (MCP) – это открытый стандарт, разработанный Anthropic для подключения AI-моделей к внешним источникам данных и инструментам. Если Claude Code сам по себе – это мощный инструмент, то с MCP он становится расширяемой платформой, способной взаимодействовать практически с любой системой.

Представьте, что Claude Code – это смартфон. Из коробки он умеет многое: звонить, фотографировать, навигировать. Но по-настоящему мощным его делает App Store – возможность установить тысячи приложений для любых задач. MCP – это «App Store» для Claude Code: каждый MCP-сервер добавляет новые возможности.

Архитектура MCP

MCP следует клиент-серверной архитектуре:

```
+-+ MCP Protocol +-+
| | <-> | MCP Server: |
| Claude Code | | PostgreSQL |
| (MCP Host) | +-+
| | MCP Protocol +-+
| | <-> | MCP Server: |
| | | GitHub |
| | +-+
| | MCP Protocol +-+
| | <-> | MCP Server: |
| | | Jira |
+-+ +-+
```

MCP-сервер предоставляет три типа возможностей:

– **Tools** (Инструменты) – функции, которые Claude Code может вызывать (запрос к БД, создание PR, отправка сообщения)
– **Resources** (Ресурсы) – данные для чтения (файлы, записи БД, документация)
– **Prompts** (Промпты) – шаблоны взаимодействия (готовые сценарии работы)

Протокол коммуникации

MCP использует JSON-RPC 2.0 поверх двух транспортов:

– **stdio** – для локальных серверов (обмен через stdin/stdout)
– **SSE (Server-Sent Events)** – для удалённых серверов (через HTTP)

13.2. Настройка MCP-серверов

Конфигурация

MCP-серверы настраиваются в файле `~/.claude/settings.json` или в проектном `.claude/settings.json`:

```
{
"mcpServers": {
"postgres": {
"command": "npx",
"args": ["-y", "@modelcontextprotocol/server-postgres"],
"env": {
"DATABASE_URL": "postgresql://user:pass@localhost:5432/mydb"
}
},
"github": {
"command": "npx",
"args": ["-y", "@modelcontextprotocol/server-github"],
"env": {
"GITHUB_TOKEN": "ghp_xxx"
}
},
"filesystem": {
"command": "npx",
"args": [
"-y", "@modelcontextprotocol/server-filesystem",
"-allowed-directories", "/home/user/projects"
]
}
}
}
```

Проверка подключения

После настройки можно проверить:

```
> Какие MCP-серверы подключены? Покажи список
доступных инструментов и ресурсов для каждого.
```

Claude Code покажет:

```
Подключённые MCP-серверы:
1. postgres
Tools: query, execute, list_tables, describe_table
Resources: schema://public
2. github
Tools: create_issue, create_pull_request, list_repos,
search_code, get_file_contents
Resources: repo://owner/name
3. filesystem
Tools: read_file, write_file, list_directory
Resources: file:///home/user/projects
```

13.3. Популярные MCP-серверы

PostgreSQL

```
> Используя MCP-сервер postgres, покажи:
1. Все таблицы в базе данных
2. Структуру таблицы orders
3. Последние 10 заказов с именами клиентов
4. Самые популярные товары за месяц
```

Claude Code выполнит SQL-запросы через MCP:

```
-- Через MCP инструмент `query`
SELECT t.table_name,
pg_size_pretty(pg_total_relation_size(t.table_name::regclass))
as size
FROM information_schema.tables t
WHERE t.table_schema = 'public'
ORDER BY pg_total_relation_size(t.table_name::regclass) DESC;
```

GitHub

```
> Используя MCP-сервер github:
1. Покажи открытые issues в репозитории
2. Найди все TODO в коде через code search
3. Создай issue для найденных проблем
4. Создай PR с исправлениями
```

Slack

```
{
"mcpServers": {
"slack": {
"command": "npx",
"args": ["-y", "@anthropic/mcp-server-slack"],
"env": {
"SLACK_BOT_TOKEN": "xoxb-xxx",
"SLACK_TEAM_ID": "T0123456789"
}
}
}
}
```

```
> Найди в Slack-канале #dev-backend обсуждения о проблемах
с производительностью за последнюю неделю.
Создай summary.
```

Sentry

```
> Подключи Sentry MCP-сервер и:
1. Покажи топ-10 ошибок за последние 24 часа
2. Для самой частой ошибки найди корневую причину в коде
3. Предложи исправление
4. Создай PR с фиксом
```

Puppeteer (Browser)

```
{
"mcpServers": {
"puppeteer": {
"command": "npx",
"args": ["-y", "@anthropic/mcp-server-puppeteer"]
}
}
}
```

```
> Открой наш сайт через MCP Puppeteer:
1. Сделай скриншот главной страницы
2. Проверь, что все ссылки работают
3. Заполни форму обратной связи и отправь
4. Проверь, что сообщение появилось в БД
```

13.4. Создание собственного MCP-сервера

Архитектура MCP-сервера

```
> Создай MCP-сервер для нашей внутренней системы
управления задачами (TaskTracker).
Tools:
- list_tasks: получить список задач (фильтрация по статусу)
- create_task: создать задачу
- update_task: обновить статус/assignee
- get_task_stats: статистика (open/in_progress/done)
Resources:
- task://{id}: данные конкретной задачи
- tasks://board/{board_id}: все задачи доски
Prompts:
- daily_standup: шаблон для ежедневного стендапа
- sprint_review: шаблон для обзора спринта
Используй TypeScript + @modelcontextprotocol/sdk.
```

Claude Code создаст MCP-сервер:

```
// src/server.ts
import { Server } from '@modelcontextprotocol/sdk/server/
index.js';
import { StdioServerTransport } from '@modelcontextprotocol/
sdk/server/stdio.js';
import {
CallToolRequestSchema,
ListToolsRequestSchema,
ListResourcesRequestSchema,
ReadResourceRequestSchema,
} from '@modelcontextprotocol/sdk/types.js';
import { TaskTracker } from './taskTracker.js';
const tracker = new TaskTracker(process.env.TRACKER_API_URL!);
const server = new Server(
{ name: 'tasktracker', version: '1.0.0' },
{ capabilities: { tools: {}, resources: {} } }
);
// Регистрация инструментов
server.setRequestHandler(ListToolsRequestSchema, async () =>
({
tools: [
{
name: 'list_tasks',
description: 'Получить список задач с фильтрацией',
inputSchema: {
type: 'object',
```

```
    properties: {
    status: {
    type: 'string',
    enum: ['open', 'in_progress', 'done', 'all'],
    description: 'Фильтр по статусу',
    },
    assignee: {
    type: 'string',
    description: 'Фильтр по исполнителю',
    },
    limit: {
    type: 'number',
    default: 20,
    description: 'Максимальное количество задач',
    },
    },
    },
    },
    {
    name: 'create_task',
    description: 'Создать новую задачу',
    inputSchema: {
    type: 'object',
    properties: {
    title: { type: 'string', description: 'Название задачи' },
    description: { type: 'string', description: 'Описание' },
    assignee: { type: 'string', description: 'Исполнитель' },
    priority: {
    type: 'string',
    enum: ['low', 'medium', 'high', 'critical'],
    },
    },
    required: ['title'],
    },
    },
    ],
    }));
    // Обработка вызовов инструментов
    server.setRequestHandler(CallToolRequestSchema,         async
(request) => {
    const { name, arguments: args } = request.params;
    switch (name) {
    case 'list_tasks': {
    const tasks = await tracker.listTasks(args);
    return {
    content: [{
    type: 'text',
    text: JSON.stringify(tasks, null, 2),
    }],
```

```
};
}
case 'create_task': {
const task = await tracker.createTask(args);
return {
content: [{
type: 'text',
text: `Задача создана: ${task.id} – ${task.title}`,
}],
};
}
default:
throw new Error(`Unknown tool: ${name}`);
}
});
// Запуск
const transport = new StdioServerTransport();
await server.connect(transport);
```

Тестирование MCP-сервера

```
> Напиши тесты для MCP-сервера TaskTracker:
1. Unit-тесты для каждого tool
2. Интеграционный тест (подключение к серверу)
3. Тест на обработку ошибок (невалидные аргументы)
```

Публикация MCP-сервера

```
> Подготовь MCP-сервер к публикации в npm:
1. package.json с правильными полями (bin, main, type)
2. README с инструкцией по настройке
3. Типы TypeScript
4. GitHub Actions для CI и публикации
```

13.5. Паттерны использования MCP

Паттерн «Мост к внутренним системам»

MCP идеально подходит для подключения Claude Code к внутренним инструментам компании, которые не имеют публичного API:

```
> Создай MCP-сервер для нашей внутренней Wiki:
- Поиск статей
- Чтение статьи по ID
```

```
- Создание/обновление статей
Wiki API: REST, авторизация через LDAP токен.
```

Паттерн «Агрегатор данных»

Используйте несколько MCP-серверов одновременно для комплексных задач:

```
> Используя MCP-серверы (postgres + github + sentry):
1. Найди в Sentry самую частую ошибку
2. Найди в GitHub файл, где она происходит
3. Проверь в PostgreSQL, каких пользователей затрагивает
4. Исправь ошибку и создай PR
```

Паттерн «Workflow автоматизация»

```
> Создай MCP-сервер для автоматизации деплоя:
Tools:
- check_ci_status: статус CI для коммита
- deploy_staging: деплой на staging
- run_smoke_tests: запуск smoke-тестов
- deploy_production: деплой на production
- rollback: откат на предыдущую версию
- notify_team: уведомление в Slack
```

13.6. Безопасность MCP

Принципы безопасности

– **Минимальные привилегии** – давайте MCP-серверу только необходимые разрешения
– **Изоляция** – каждый сервер в своём процессе
– **Ротация токенов** – используйте переменные окружения, не хардкодьте
– **Аудит** – логируйте все вызовы инструментов
– **Ограничение операций** – блокируйте деструктивные операции по умолчанию

Пример безопасной конфигурации

```
{
"mcpServers": {
"postgres-readonly": {
"command": "npx",
"args": ["-y", "@modelcontextprotocol/server-postgres"],
"env": {
```

```
    "DATABASE_URL":   "postgresql://readonly_user:pass@localhost/
mydb"
    }
    }
    }
    }
```

Обратите внимание: используется readonly-пользователь БД, что предотвращает модификацию данных через MCP.

Итоги главы

В этой главе мы рассмотрели:

– Архитектуру MCP: клиент-серверная модель, транспорты, типы возможностей
– Настройку популярных MCP-серверов: PostgreSQL, GitHub, Slack, Sentry, Puppeteer
– Создание собственного MCP-сервера на TypeScript
– Паттерны использования: мост, агрегатор, workflow
– Безопасность MCP: принципы и практики

В следующей главе мы рассмотрим автоматизацию рабочих процессов разработки.

Глава 14. Автоматизация рабочих процессов

14.1. Claude Code как двигатель автоматизации

Автоматизация – это суперспособность разработчика. Каждая рутинная задача, которую вы автоматизируете, освобождает время для творческой работы. Claude Code позволяет автоматизировать на новом уровне: вместо написания скриптов вручную вы описываете, что нужно сделать, и получаете готовый скрипт.

Но автоматизация с Claude Code выходит за рамки простых скриптов. Используя режим headless, пользовательские команды и хуки, вы можете создавать полноценные конвейеры, интегрирующие Claude Code в ваш рабочий процесс.

14.2. Пользовательские команды

Проектные команды

Файлы в `.claude/commands/` становятся slash-командами, доступными через `/project::`

```
<!- .claude/commands/new-feature.md ->
Создай новую фичу: $ARGUMENTS
1. Создай feature-branch: feature/$ARGUMENTS
2. Создай файлы по нашей структуре:
- src/modules/$ARGUMENTS/
- src/modules/$ARGUMENTS/module.ts
- src/modules/$ARGUMENTS/service.ts
- src/modules/$ARGUMENTS/controller.ts
- src/modules/$ARGUMENTS/types.ts
- src/modules/$ARGUMENTS/__tests__/
3. Добавь базовый scaffold (boilerplate)
4. Зарегистрируй модуль в src/modules/index.ts
5. Создай начальный коммит: feat($ARGUMENTS): scaffold module
```

Использование:

```
> /project:new-feature payments
```

Персональные команды

Файлы в `~/.claude/commands/` доступны глобально:

```
<!- ~/.claude/commands/morning.md ->
Утренний ритуал разработчика:
1. `git fetch -all` и покажи новые коммиты в develop
2. Проверь, нет ли падающих тестов
3. Покажи открытые PR, ожидающие моего review
4. Покажи мой TODO-список (TODO в коде по @my-username)
5. Резюме: что изменилось за вчера

<!- ~/.claude/commands/deploy-check.md ->
Проверь готовность к деплою:
1. Все тесты проходят?
2. Lint без ошибок?
3. TypeScript без ошибок?
4. Build успешен?
5. Нет незакоммиченных изменений?
6. Ветка синхронизирована с remote?
Выведи [+] или [-] для каждого пункта.
```

Команды для code review

```
<!- .claude/commands/review-pr.md ->
Проведи полный code review текущих изменений: $ARGUMENTS
## Автоматические проверки
1. Запусти `pnpm lint` и покажи результат
2. Запусти `pnpm test` и покажи результат
3. Запусти `pnpm build` и проверь на ошибки
## Code Review
4. Проанализируй diff (`git diff develop…HEAD`)
5. Для каждого изменённого файла:
- Корректность логики
- Обработка ошибок
- Безопасность
- Производительность
- Тестовое покрытие
## Результат
Выведи в формате:
- [!] Critical: [блокирующие проблемы]
- [~] Warning: [рекомендации]
- [i] Note: [замечания]
- [+] Хорошее: [что сделано хорошо]
```

14.3. Hooks: реактивная автоматизация

Hooks позволяют запускать код при определённых событиях Claude Code. Они настраиваются в `.claude/settings.json`:

```
{
```

```
    "hooks": {
    "PreToolUse": [
    {
    "matcher": "Bash",
    "hooks": [
    {
    "type": "command",
    "command": "echo 'About to run: '$(echo $TOOL_INPUT | jq -r
'.command')"
    }
    ]
    }
    ],
    "PostToolUse": [
    {
    "matcher": "WriteFile",
    "hooks": [
    {
    "type": "command",
    "command": "/home/user/scripts/format-on-save.sh $FILE_PATH"
    }
    ]
    }
    ],
    "Notification": [
    {
    "matcher": "",
    "hooks": [
    {
    "type": "command",
    "command": "notify-send 'Claude Code' '$MESSAGE'"
    }
    ]
    }
    ]
    }
    }
```

Практические примеры хуков

Автоформатирование при записи файла:

```
    {
    "hooks": {
    "PostToolUse": [
    {
    "matcher": "WriteFile",
    "hooks": [
```

```
{
"type": "command",
"command": "npx prettier –write $FILE_PATH 2>/dev/null || true"
}
]
}
]
}
}
```

Проверка безопасности перед выполнением команд:

```
{
"hooks": {
"PreToolUse": [
{
"matcher": "Bash",
"hooks": [
{
"type": "command",
"command": "/home/user/scripts/check-safe-command.sh"
}
]
}
]
}
}
```

Уведомление о завершении долгих операций:

```
{
"hooks": {
"Notification": [
{
"matcher": "",
"hooks": [
{
"type": "command",
"command": "terminal-notifier –title 'Claude Code' –message
'$MESSAGE'"
}
]
}
]
}
}
```

14.4. Headless-режим и SDK

Headless-режим

Claude Code может работать без интерактивного терминала – в headless-режиме:

```
claude -p "Сколько TODO в проекте?" -output-format json
cat error.log | claude -p "Проанализируй логи и найди причину ошибки"
claude -p "Создай unit-тесты для src/utils/math.ts" \
--output-format json \
--max-turns 10
```

Автоматизация через скрипты

```
#!/bin/bash
DIFF=$(git diff origin/main…HEAD)
RESULT=$(echo "$DIFF" | claude -p "
Проведи code review этого diff.
Формат: JSON с массивом comments.
Каждый comment: { file, line, severity, message }
" -output-format json)
echo "$RESULT" | jq -r '.comments[]' | while read comment; do
gh pr comment -body "$comment"
done
```

Claude Code SDK

Для интеграции в Node.js-приложения:

```
import { Claude } from '@anthropic-ai/claude-code';
const claude = new Claude({
model: 'claude-sonnet-4-20250514',
});
// Программное выполнение задач
async function analyzeProject(path: string) {
const result = await claude.run({
prompt: `Проанализируй проект в ${path}.
Верни JSON с метриками: loc, files, dependencies, test_coverage`,
workingDirectory: path,
outputFormat: 'json',
maxTurns: 5,
});
```

```
    return JSON.parse(result.output);
    }
```

14.5. CI/CD интеграция

GitHub Actions

```
    name: Claude Code Review
    on:
    pull_request:
    types: [opened, synchronize]
    jobs:
    review:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    with:
    fetch-depth: 0
    - name: Install Claude Code
    run: npm install -g @anthropic-ai/claude-code
    - name: Run Code Review
    env:
    ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
    run: |
    DIFF=$(git diff origin/main…HEAD)
    echo "$DIFF" | claude -p "
    Review this diff. Focus on:
    - Bugs and logic errors
    - Security vulnerabilities
    - Performance issues
    Output as GitHub PR review comments format.
    " > review.md
    - name: Post Review
    uses: actions/github-script@v7
    with:
    script: |
    const   review   =   require('fs').readFileSync('review.md',
'utf8');
    github.rest.pulls.createReview({
    …context.repo,
    pull_number: context.issue.number,
    body: review,
    event: 'COMMENT'
    });
```

Автоматическая генерация CHANGELOG

```
name: Auto Changelog
on:
push:
tags: ['v*']
jobs:
changelog:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v4
with:
fetch-depth: 0
- name: Generate Changelog
env:
ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
run: |
COMMITS=$(git log $(git describe -tags -abbrev=0 HEAD~1)..HEAD
\
--pretty=format:"%s")
echo "$COMMITS" | claude -p "
Сгенерируй CHANGELOG из этих коммитов.
Формат: Keep a Changelog.
Язык: русский.
Группировка: Added, Changed, Fixed, Removed.
" > RELEASE_NOTES.md
```

14.6. Автоматизация повседневных задач

Автоматическое обновление зависимостей

```
> Создай скрипт, который:
1. Проверяет outdated зависимости (npm outdated)
2. Для каждой outdated зависимости:
- Читает CHANGELOG (breaking changes?)
- Обновляет до latest
- Запускает тесты
- Если тесты прошли - коммит
- Если нет - откат и логирование
3. Создаёт PR со всеми обновлениями
```

Автоматическая миграция кода

```
> Создай скрипт для миграции с React Class Components
```

```
на Function Components:
1. Найди все Class Components в src/
2. Для каждого:
- Конвертируй в Function Component
- Запусти тесты компонента
- Если ok – коммит
- Если fail – пропусти и залогируй
3. Отчёт: [мигрировано / не удалось / осталось]
```

Автоматический аудит безопасности

```
#!/bin/bash
npm audit –json > audit.json
npx eslint –plugin security src/ –format json > eslint-
security.json
claude –p "
Проанализируй результаты аудита безопасности:
$(cat audit.json)
$(cat eslint-security.json)
Создай отчёт:
- Критические уязвимости (требуют немедленного исправления)
- Важные (исправить в ближайшем спринте)
- Низкого приоритета (backlog)
Для каждой: описание, затронутые файлы, рекомендация.
" –output-format json > security-report.json
```

14.7. Cron-автоматизация

```
> Создай набор cron-задач для ежедневного обслуживания проекта:
06:00 – Проверить CI/CD pipeline (все ли зелёное)
06:30 – Проверить outdated зависимости
07:00 – Проверить SLA (время ответа API за последние 24 часа)
18:00 – Daily summary (что изменилось за день)
```

Итоги главы

В этой главе мы рассмотрели:

– Пользовательские команды: проектные и персональные
– Hooks: реактивная автоматизация на события Claude Code
– Headless-режим и Claude Code SDK
– CI/CD интеграцию с GitHub Actions
– Автоматизацию повседневных задач: обновление зависимостей, миграция, аудит
– Cron-автоматизацию для ежедневного обслуживания

В следующей главе мы рассмотрим работу с инфраструктурой: Docker, Kubernetes, облачные платформы.

Глава 15. DevOps и инфраструктура

15.1. Docker: контейнеризация приложений

Claude Code превращает Docker из набора загадочных команд в понятный инструмент. Даже если вы никогда не писали Dockerfile, Claude Code создаст его профессионально.

Написание Dockerfile

```
> Создай оптимизированный Dockerfile для Node.js-приложения:
Требования:
- Node 20 Alpine
- Мультистейдж сборка
- Непривилегированный пользователь
- Health check
- Минимальный размер образа
- Кэширование зависимостей
- .dockerignore
```

Claude Code создаст:

```
FROM node:20-alpine AS deps
WORKDIR /app
COPY package.json pnpm-lock.yaml ./
RUN corepack enable pnpm && pnpm install --frozen-lockfile
FROM node:20-alpine AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
RUN corepack enable pnpm && pnpm build
RUN pnpm prune --prod
FROM node:20-alpine AS runner
WORKDIR /app
RUN addgroup --system --gid 1001 nodejs \
&& adduser --system --uid 1001 app
COPY --from=builder --chown=app:nodejs /app/dist ./dist
COPY --from=builder --chown=app:nodejs /app/node_modules ./
node_modules
COPY --from=builder --chown=app:nodejs /app/package.json ./
USER app
EXPOSE 3000
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --
retries=3 \
```

```
    CMD wget -no-verbose -tries=1 -spider http://localhost:3000/
health || exit 1
    CMD ["node", "dist/main.js"]
```

Docker Compose для разработки

```
> Создай Docker Compose для полного dev-окружения:
Сервисы:
- app: наше Node.js приложение (hot reload)
- db: PostgreSQL 16
- redis: Redis 7 (кэш + очереди)
- mailhog: перехват email
- traefik: reverse proxy + TLS
- prometheus + grafana: мониторинг
Сети: frontend, backend, monitoring
Volumes: для persistence данных
Health checks для всех сервисов
```

Оптимизация образов

```
> Наш Docker-образ весит 2.1 ГБ. Оптимизируй его.
Текущий:
- Base: node:20
- Все devDependencies установлены
- Исходный код и тесты в образе
- node_modules полностью скопированы
Цель: < 200 МБ
```

Claude Code предложит:

– Перейти на `node:20-alpine` (-1.5 ГБ)
– Мультистейдж сборку (-dev зависимости)
– `.dockerignore` для исключения тестов, docs, .git
– `pnpm prune -prod` для только production-зависимостей

15.2. Kubernetes

Helm charts

```
> Создай Helm chart для нашего приложения:
Компоненты:
- API сервер (Deployment, 2-5 реплик, HPA)
- Worker для фоновых задач (Deployment, 1-3 реплики)
- PostgreSQL (StatefulSet или Bitnami chart)
```

```
- Redis (Bitnami chart)
- Ingress с TLS (cert-manager)
Values:
- image.repository, image.tag
- replicas для каждого компонента
- resources (limits/requests)
- environment-specific values
Environments: dev, staging, production
```

Kubernetes манифесты

```
> Создай Kubernetes манифесты для blue-green deployment:
- Deployment для "blue" и "green" версий
- Service для переключения трафика
- Скрипт для переключения
- Автоматический rollback при failed health checks
```

Troubleshooting Kubernetes

```
> Pod'ы в состоянии CrashLoopBackOff.
Помоги разобраться:
1. Какие команды kubectl выполнить для диагностики?
2. Какие типичные причины?
3. Как исправить?
```

Claude Code предложит:

```
kubectl describe pod <pod-name>
kubectl logs <pod-name> -previous
kubectl get events -sort-by=.metadata.creationTimestamp
```

15.3. Terraform и IaC

Создание инфраструктуры

```
> Создай Terraform-конфигурацию для AWS:
Ресурсы:
- VPC с public/private subnet (2 AZ)
- ECS Fargate cluster
- RDS PostgreSQL (Multi-AZ)
- ElastiCache Redis
- ALB с HTTPS
- S3 для статики + CloudFront CDN
- Route53 для DNS
```

```
- ACM для TLS-сертификата
Модули:
- networking (VPC, subnets, security groups)
- database (RDS, ElastiCache)
- compute (ECS, ALB)
- storage (S3, CloudFront)
Environments: staging, production
State: S3 + DynamoDB
```

Terraform модули

```
> Создай переиспользуемый Terraform-модуль для ECS-сервиса:
Inputs:
- name, container_image, container_port
- cpu, memory
- desired_count, min_count, max_count
- environment_variables
- health_check_path
Resources:
- ECS Task Definition
- ECS Service
- ALB Target Group
- ALB Listener Rule
- Auto Scaling Policy
- CloudWatch Log Group
- IAM Roles
```

15.4. CI/CD Pipelines

GitHub Actions (полный pipeline)

```
> Создай полный CI/CD pipeline:
Trigger: push to main, PR to main
Jobs:
1. lint: ESLint + Prettier
2. typecheck: TypeScript
3. test:
- Unit tests (Vitest)
- Integration tests (с PostgreSQL service)
- E2E tests (Playwright)
4. security:
- npm audit
- Snyk scan
- SAST (Semgrep)
5. build: Docker build + push to ECR
```

```
6. deploy-staging: (auto на main)
7. smoke-tests: на staging
8. deploy-production: (manual approval)
Matrix: Node 18, 20
Cache: node_modules, Docker layers, Playwright browsers
Artifacts: test reports, coverage, screenshots
```

GitLab CI

```
> Создай .gitlab-ci.yml с аналогичным pipeline:
Stages: lint, test, build, deploy
Включи:
- Параллельные jobs
- Cache между stages
- Review apps для PR
- Canary deployment
- Auto rollback
```

15.5. Мониторинг и observability

Prometheus + Grafana

```
> Настрой мониторинг для Node.js-приложения:
Метрики (prom-client):
- HTTP request duration (histogram)
- HTTP request count (counter) по method/path/status
- Active connections (gauge)
- Event loop lag (histogram)
- Memory usage (gauge)
- Custom: orders_processed_total, payment_errors_total
Создай:
1. Middleware для сбора метрик
2. Endpoint GET /metrics
3. Grafana dashboard (JSON)
4. Alerting rules (PrometheusRule)
```

Structured logging

```
> Настрой structured logging:
Logger: winston (Node.js) / structlog (Python)
Формат: JSON с полями:
- timestamp, level, message
- requestId (correlation)
- userId (если авторизован)
```

```
- duration (для запросов)
- error (stack trace)
Транспорты:
- Console (dev)
- File (production)
- CloudWatch (AWS)
Request-scoped logging через AsyncLocalStorage.
```

Health checks

```
> Реализуй comprehensive health check endpoint:
GET /health -> 200 { status: "ok" }
GET /health/ready -> проверяет:
- PostgreSQL connection
- Redis connection
- Disk space
- Memory usage
GET /health/live -> только проверка, что процесс жив
Используй для Kubernetes readiness/liveness probes.
```

15.6. Nginx и reverse proxy

```
> Создай конфигурацию Nginx:
- Reverse proxy к Node.js (localhost:3000)
- SSL/TLS (Let's Encrypt)
- HTTP/2
- Gzip и Brotli сжатие
- Security headers (HSTS, CSP, X-Frame-Options)
- Rate limiting
- Кэширование статики (1 год)
- WebSocket proxy
- Access/error logs
```

15.7. Скрипты для DevOps

Скрипт деплоя

```
> Создай скрипт деплоя (scripts/deploy.sh):
Этапы:
1. Проверка: тесты, линт, build
2. Бэкап: дамп БД перед деплоем
3. Build: Docker image + push
4. Migrate: выполнить миграции БД
5. Deploy: обновить контейнеры (rolling update)
```

```
6. Verify: smoke tests
7. Notify: сообщение в Slack
Rollback: автоматический при failure на шагах 5-6
Логирование: все шаги в deploy.log
```

Скрипт бэкапа

```
> Создай систему бэкапов:
- Ежедневный бэкап PostgreSQL (pg_dump)
- Сжатие через gzip
- Загрузка в S3 с lifecycle policy
- Ротация: 7 ежедневных, 4 еженедельных, 12 месячных
- Проверка бэкапа: автоматический restore на тестовом сервере
- Alert при неудачном бэкапе
```

Итоги главы

В этой главе мы рассмотрели:

– Docker: Dockerfile, Docker Compose, оптимизация образов
– Kubernetes: Helm charts, манифесты, troubleshooting
– Terraform: IaC для AWS, модули
– CI/CD: GitHub Actions, GitLab CI
– Мониторинг: Prometheus, Grafana, structured logging
– Nginx и reverse proxy
– DevOps-скрипты: деплой, бэкап

В следующей главе мы рассмотрим вопросы безопасности.

Глава 16. Безопасность

16.1. Безопасность при работе с AI-ассистентом

Безопасность в контексте Claude Code имеет два измерения. Первое – защита вашего кода и данных при использовании AI. Второе – создание безопасного кода с помощью AI. Обе темы критически важны, и обе заслуживают глубокого рассмотрения.

Защита данных при работе с Claude Code

Claude Code работает с вашим кодом – а код может содержать секреты:

– API-ключи и токены
– Пароли к базам данных
– Приватные ключи шифрования
– Конфиденциальную бизнес-логику

Claude Code имеет встроенные механизмы защиты:

Секреты не передаются. Claude Code автоматически обнаруживает строки, похожие на секреты (API-ключи, токены), и редактирует их перед отправкой в модель. Если вы вставляете `.env` файл в контекст, значения секретов будут замаскированы.

Разрешения. Каждое действие с побочными эффектами (запись файла, выполнение команды) требует подтверждения. Вы контролируете, что именно делает Claude Code.

Локальная обработка. Файлы читаются и обрабатываются локально. В модель отправляется только то, что необходимо для ответа.

Тем не менее, следуйте правилам гигиены:

```
> Вот мой .env файл:
DATABASE_URL=postgresql://root:super_secret_password@…
STRIPE_SECRET_KEY=sk_live_…
> Приложение не подключается к PostgreSQL.
Конфигурация в .env (пароль и URL скрыты).
Ошибка: "Connection refused at localhost:5432"
```

16.2. Аудит безопасности кода

Комплексный аудит

```
> Проведи аудит безопасности проекта. Проверь:
## Инъекции
- SQL Injection (строковая конкатенация в запросах)
- NoSQL Injection
- Command Injection (exec, spawn с пользовательским вводом)
- LDAP Injection
## XSS (Cross-Site Scripting)
- Reflected XSS
- Stored XSS
- DOM-based XSS
## Аутентификация и авторизация
- Слабое хеширование паролей
- Отсутствие rate limiting на login
- JWT без проверки подписи
- Отсутствие RBAC
## Конфигурация
- Секреты в коде или конфигурационных файлах
- Debug mode в production
- CORS: allow all origins
- Отсутствие security headers
## Зависимости
- Известные уязвимости (CVE)
- Устаревшие пакеты
Формат: таблица [Файл, Строка, Тип, Серьёзность, Рекомендация]
```

Проверка конкретных уязвимостей

SQL Injection:

```
> Найди все места в коде, где используется строковая
конкатенация в SQL-запросах. Замени на параметризованные.
```

Claude Code найдёт:

```
// [-] Уязвимо
const query = `SELECT * FROM users WHERE email = '${email}'`;
// [+] Безопасно
const query = 'SELECT * FROM users WHERE email = $1';
const result = await pool.query(query, [email]);
```

XSS:

```
> Найди все места, где пользовательский ввод выводится
в HTML без экранирования. Добавь sanitization.

// [-] Уязвимо
element.innerHTML = userInput;
// [+] Безопасно
import DOMPurify from 'dompurify';
element.innerHTML = DOMPurify.sanitize(userInput);
```

Command Injection:

```
> Найди вызовы exec/spawn/system с пользовательским вводом.
Замени на безопасные альтернативы.

// [-] Уязвимо
exec(`convert ${filename} output.png`);
// [+] Безопасно
execFile('convert', [filename, 'output.png']);
```

16.3. Аутентификация и авторизация

Реализация аутентификации

```
> Реализуй безопасную систему аутентификации:
- Регистрация: email + password
- Пароль: bcrypt с salt rounds = 12
- JWT: access token (15 min) + refresh token (7 days)
- Refresh token rotation (новый при каждом refresh)
- Token blacklisting при logout
- Rate limiting: 5 попыток по IP + 10 попыток по email
- Account lockout: 30 минут после 10 неудачных попыток
- Secure cookies: httpOnly, secure, sameSite=strict
```

RBAC (Role-Based Access Control)

```
> Реализуй систему ролей и разрешений:
Роли:
- admin: полный доступ
- manager: CRUD для products, read для orders, users
- editor: CRUD для content, read для products
- viewer: read-only для всего
Middleware:
- requireAuth: проверка JWT
- requireRole(role): проверка роли
```

```
- requirePermission(resource, action): гранулярная проверка
Гибкость: разрешения хранятся в БД, не захардкожены.
```

OAuth 2.0

```
> Добавь OAuth 2.0 аутентификацию:
Провайдеры: Google, GitHub, VK
Flow: Authorization Code (с PKCE для SPA)
При первом входе: создание пользователя
При повторном: привязка к существующему аккаунту
Храни access/refresh токены провайдеров в БД (зашифрованные).
```

16.4. Защита API

Rate Limiting

```
> Реализуй многоуровневый rate limiting:
Уровни:
1. Global: 1000 req/min (защита от DDoS)
2. Per IP: 100 req/min
3. Per User: 200 req/min (авторизованные)
4. Per Endpoint:
- POST /api/auth/login: 5 req/min per IP
- POST /api/uploads: 10 req/hour per user
- GET /api/search: 30 req/min per user
Storage: Redis (sliding window algorithm)
Headers: X-RateLimit-Limit, X-RateLimit-Remaining, Retry-After
```

Input Validation

```
> Создай слой валидации для API:
Используй Zod для TypeScript.
Для каждого endpoint:
- Body: schema с type + constraints
- Query params: schema
- Path params: schema
- Headers: проверка Content-Type, Accept
Централизованная обработка ошибок валидации:
422 с описанием каждого невалидного поля.
```

Security Headers

```
> Настрой security headers middleware:
```

```
    - Strict-Transport-Security: max-age=31536000;
includeSubDomains
    - Content-Security-Policy: (строгая политика)
    - X-Content-Type-Options: nosniff
    - X-Frame-Options: DENY
    - X-XSS-Protection: 0 (deprecated, CSP заменяет)
    - Referrer-Policy: strict-origin-when-cross-origin
    - Permissions-Policy: camera=(), microphone=(), geolocation=()
    CSP: настрой под наше приложение (разреши наши домены, CDN,
API).
```

16.5. Шифрование данных

```
    > Реализуй шифрование чувствительных данных:
    Данные для шифрования:
    - PII (Personally Identifiable Information)
    - Платёжные данные
    - OAuth-токены провайдеров
    Подход:
    - AES-256-GCM для данных at rest
    - Ключ хранится в переменных окружения
    - Key rotation: возможность перешифровки данных
    - Прозрачное шифрование: middleware на уровне ORM
    Используй crypto (Node.js stdlib), НЕ сторонние библиотеки.
```

16.6. CSRF и CORS

```
    > Настрой защиту от CSRF:
    - Double Submit Cookie pattern (для SPA)
    - SameSite cookies
    - Origin check
    CORS:
    - Разрешённые origins: наш фронтенд (localhost:3000,
app.example.com)
    - Methods: GET, POST, PUT, DELETE, PATCH
    - Headers: Authorization, Content-Type
    - Credentials: true
    - Max-age: 86400
```

16.7. Безопасность зависимостей

```
    > Настрой автоматическую проверку безопасности зависимостей:
    1. npm audit в CI (блокировать merge при critical)
    2. Snyk или Socket для глубокого анализа
    3. Dependabot/Renovate для автообновления
```

```
4. Lock-файл: запретить install без lock-файла
5. Проверка на typosquatting (подозрительные имена пакетов)
```

16.8. Логирование для безопасности

```
> Реализуй Security Event Logging:
Логируемые события:
- Успешный/неудачный вход
- Смена пароля
- Изменение пермиссий
- Доступ к чувствительным данным
- API rate limit exceeded
- Подозрительная активность
Формат: JSON с timestamp, userId, IP, userAgent, event, details
Хранение: отдельная таблица/коллекция (security_events)
Retention: 90 дней
Alerts: на critical events в реальном времени
```

16.9. Чеклист безопасности

```
> Создай security checklist для нашего проекта:
## Pre-deployment
- [ ] Все секреты в переменных окружения
- [ ] Debug mode выключен
- [ ] HTTPS обязателен
- [ ] Security headers настроены
- [ ] CORS настроен правильно
- [ ] Rate limiting включён
- [ ] Input validation на всех endpoints
- [ ] SQL injection защита
- [ ] XSS защита
- [ ] CSRF защита
- [ ] JWT правильно настроен (short-lived, signed)
- [ ] Passwords: bcrypt/argon2, salt
- [ ] Sensitive data encrypted
- [ ] npm audit: 0 critical/high
- [ ] Dependency lock file up to date
- [ ] Error messages: не раскрывают детали реализации
- [ ] Logging: security events логируются
- [ ] Backup: настроен и протестирован
Добавь как CI-проверку: скрипт, проверяющий каждый пункт.
```

Итоги главы

В этой главе мы рассмотрели:

– Безопасность данных при работе с Claude Code
– Аудит безопасности: инъекции, XSS, конфигурация
– Аутентификацию и авторизацию: JWT, OAuth, RBAC
– Защиту API: rate limiting, валидация, security headers
– Шифрование чувствительных данных
– CSRF и CORS
– Безопасность зависимостей
– Security logging и чеклисты

В следующей главе мы рассмотрим работу на разных языках программирования.

Глава 17. Мультиязычная разработка

17.1. Claude Code – полиглот

Одна из сильнейших сторон Claude Code – свободное владение десятками языков программирования. Это не просто «знание синтаксиса» – Claude Code понимает идиоматику каждого языка, его экосистему, лучшие практики и типичные ошибки.

Для разработчика это открывает несколько возможностей:

– **Кроссплатформенная разработка** – один проект, разные языки
– **Миграция** – перевод кода с одного языка на другой
– **Изучение нового языка** – Claude Code как наставник
– **Полиглотная архитектура** – микросервисы на разных языках

17.2. Python

Python – один из языков, где Claude Code проявляет себя лучше всего. Экосистема Python богата, и Claude Code знает её глубоко.

Научные вычисления

```
> Создай скрипт для анализа данных о продажах:
Данные: CSV с колонками (date, product_id, quantity, price,
category)
Анализ:
1. Общая статистика (pandas describe)
2. Продажи по месяцам (line chart)
3. Топ-10 товаров (bar chart)
4. Распределение по категориям (pie chart)
5. Корреляция цена-количество (scatter plot)
6. Прогноз на следующий месяц (простая линейная регрессия)
Используй: pandas, matplotlib, seaborn, scikit-learn
Вывод: HTML-отчёт с интерактивными графиками (plotly)
```

FastAPI + SQLAlchemy

```
> Создай microservice на Python:
- FastAPI 0.109+
- SQLAlchemy 2.0 (async)
- Pydantic v2 для валидации
- Alembic для миграций
```

```
- pytest + httpx для тестов
- Poetry для зависимостей
CRUD для модели Article:
- Полнотекстовый поиск
- Пагинация (cursor-based)
- Фильтрация по тегам
- Сортировка по дате/популярности
```

Django

```
> Создай Django-приложение для учёта рабочего времени:
Модели:
- Employee (name, department, position)
- TimeEntry (employee, date, hours, project, description)
- Project (name, client, budget_hours)
Views:
- Dashboard с общей статистикой
- Табель за месяц (calendar view)
- Отчёт по проектам
Admin: кастомные list_display, filters, actions.
API: DRF с сериализаторами и viewsets.
Templates: используй Bootstrap 5.
```

Asyncio и конкурентность

```
> Создай асинхронный web scraper:
- aiohttp для HTTP-запросов
- asyncio.Semaphore для ограничения конкурентности
- BeautifulSoup для парсинга HTML
- Retry с exponential backoff
- Прогресс-бар (tqdm или rich)
- Сохранение в SQLite через aiosqlite
Парсить: список URL из файла (до 10000)
Конкурентность: 20 одновременных запросов
Rate limit: 2 запроса в секунду на домен
```

17.3. Go

Go – язык, который ценится за производительность, простоту и надёжность.

HTTP-сервер

```
> Создай HTTP API на Go:
- Chi router
```

```
- pgx для PostgreSQL
- Structured logging (slog)
- Environment config (envconfig)
- Graceful shutdown
- Middleware: logging, recovery, cors, auth
- Health check endpoint
- Swagger documentation (swaggo)
Clean Architecture:
- cmd/api/main.go – точка входа
- internal/handler/ – HTTP handlers
- internal/service/ – бизнес-логика
- internal/repository/ – работа с БД
- internal/model/ – доменные модели
```

CLI-приложение

```
> Создай CLI-утилиту на Go для управления проектами:
Команды:
- init – создать новый проект
- build – собрать проект
- test – запустить тесты
- deploy – задеплоить
- status – показать статус
Используй: cobra для CLI, viper для конфигурации,
bubbletea для TUI (интерактивный интерфейс)
Cross-compilation: Linux, macOS, Windows
Релиз: GoReleaser
```

Конкурентность в Go

```
> Реализуй pipeline обработки данных с goroutines:
Этапы:
1. Reader: читает CSV по строкам (channel)
2. Parser: парсит строки в структуры (N workers)
3. Validator: валидирует данные (N workers)
4. Transformer: обогащает данные (API-вызовы, N workers)
5. Writer: записывает в PostgreSQL (batch insert)
Error handling: errgroup
Cancellation: context.Context
Progress: atomic counter
Обработать 1M записей с конкурентностью 100.
```

17.4. Rust

```
> Создай CLI-инструмент на Rust:
```

```
Функциональность: анализатор зависимостей проекта
(аналог npm list, но для любого языка).
Поддержка:
- package.json (Node.js)
- Cargo.toml (Rust)
- go.mod (Go)
- requirements.txt / pyproject.toml (Python)
Вывод: дерево зависимостей, дубликаты, устаревшие
Используй: clap (CLI), serde (JSON/TOML), reqwest (HTTP),
tokio (async), colored (цветной вывод)
```

17.5. TypeScript/JavaScript (продвинутый)

Full-Stack TypeScript

```
> Создай type-safe full-stack приложение:
Shared types: packages/types/
- Общие TypeScript типы для frontend и backend
- Zod-схемы для runtime-валидации
- Автовывод типов из Zod-схем
Backend: packages/api/
- tRPC router
- Prisma (типы из схемы)
- End-to-end type safety от БД до UI
Frontend: packages/web/
- tRPC client (типизированные запросы)
- React Query (кэширование)
- Automatic refetch on mutation
```

Node.js Streams

```
> Реализуй ETL-пайплайн на Node.js Streams:
Extract: читать CSV (10 ГБ файл, не загружая в память)
Transform:
- Парсинг полей
- Валидация
- Обогащение (lookup в Redis)
- Агрегация (считать суммы по категориям)
Load: Запись в PostgreSQL (batch по 1000 записей)
Backpressure handling, error recovery, progress tracking.
```

17.6. C# / .NET

```
> Создай ASP.NET Core Web API:
```

```
.NET 8, Minimal APIs.
Endpoints: CRUD для Products
- Entity Framework Core + PostgreSQL
- AutoMapper для DTO
- FluentValidation
- Serilog для логирования
- MediatR (CQRS pattern)
Testing: xUnit + Moq + TestContainers
Docker: multi-stage build
CI: GitHub Actions
```

17.7. Транспиляция между языками

Claude Code может переводить код с одного языка на другой, сохраняя семантику:

```
> Переведи этот Python-скрипт на Go:
[Python код]
Требования:
- Идиоматичный Go (не «Python на Go»)
- Обработка ошибок через возврат (не panic)
- Конкурентность через goroutines (вместо asyncio)
- Используй stdlib где возможно
```

Практические примеры транспиляции

```
> Переведи Express.js API на FastAPI:
- Сохрани все endpoints
- Переведи middleware в FastAPI depends
- Замени Joi на Pydantic
- Замени Knex на SQLAlchemy
- Сохрани тесты (перепиши для pytest)

> Переведи React-компонент на Svelte:
- Состояние: useState -> $state
- Эффекты: useEffect -> $effect
- Props: interface -> export let
- Стили: CSS Modules -> scoped <style>
```

17.8. Языково-специфичные паттерны

Claude Code знает идиоматику каждого языка:

```
> Напиши функцию обработки списка пользователей
на идиоматичном [языке]:
- Отфильтровать активных
```

```
- Трансформировать в формат для API
- Отсортировать по имени
- Сгруппировать по роли
```

Python:

```
result = {
role: sorted(users, key=lambda u: u.name)
for role, users in groupby(
sorted(
(transform(u) for u in users if u.is_active),
key=lambda u: u.role
),
key=lambda u: u.role
)
}
```

Go:

```
activeUsers := make([]APIUser, 0)
for _, u := range users {
if u.IsActive {
activeUsers = append(activeUsers, transform(u))
}
}
sort.Slice(activeUsers, func(i, j int) bool {
return activeUsers[i].Name < activeUsers[j].Name
})
grouped := make(map[string][]APIUser)
for _, u := range activeUsers {
grouped[u.Role] = append(grouped[u.Role], u)
}
```

Rust:

```
let grouped: HashMap<String, Vec<ApiUser>> = users
.iter()
.filter(|u| u.is_active)
.map(|u| transform(u))
.sorted_by(|a, b| a.name.cmp(&b.name))
.group_by(|u| u.role.clone())
.into_iter()
.map(|(role, group)| (role, group.collect()))
.collect();
```

Каждая версия идиоматична для своего языка – Claude Code не пытается написать «Python на Go».

Итоги главы

В этой главе мы рассмотрели:

– Claude Code как мультиязычного разработчика
– Python: научные вычисления, FastAPI, Django, asyncio
– Go: HTTP API, CLI, конкурентность
– Rust: CLI-инструменты, системное программирование
– TypeScript: full-stack, streams, advanced patterns
– C# / .NET: ASP.NET Core
– Транспиляцию между языками
– Языково-специфичные идиомы

В следующей главе мы завершим третью часть, рассмотрев работу с различными предметными областями.

Глава 18. Предметные области: от веба до IoT

18.1. Мобильная разработка

Claude Code не ограничивается веб-разработкой. Он эффективно помогает в мобильной разработке, хотя и с некоторыми ограничениями: Claude Code не может запустить эмулятор или увидеть результат на экране телефона. Однако он может генерировать код, настраивать проекты и решать проблемы.

React Native

```
> Создай компонент экрана профиля на React Native:
- Аватар (ImagePicker для выбора/съёмки)
- Поля: имя, email, телефон (editable)
- Кнопка "Сохранить" (React Query mutation)
- Pull-to-refresh
- Скелетон при загрузке
- Анимация при сохранении (Reanimated)
Навигация: React Navigation (Stack)
Стили: StyleSheet (не Tailwind)
```

Flutter

```
> Создай экран со списком задач на Flutter:
- ListView с TaskCard виджетами
- Swipe actions: завершить (left), удалить (right)
- FloatingActionButton для новой задачи
- Фильтр: все / активные / завершённые
- Локальное хранение: Hive
- State management: Riverpod
Виджеты: Material Design 3
Анимации: Hero transition при открытии задачи
```

Swift (iOS)

```
> Создай SwiftUI view для дашборда финансов:
- Баланс с анимированным числом
- Список транзакций (LazyVStack)
- График расходов за неделю (Charts framework)
- Segmented control: день / неделя / месяц
- Core Data для хранения
- MVVM архитектура
```

18.2. Data Science и ML

Анализ данных

```
> Создай notebook-style скрипт для анализа датасета:
Датасет: CSV с e-commerce данными (orders, users, products)
Анализ:
1. EDA (Exploratory Data Analysis):
- Распределения, выбросы, пропуски
- Корреляционная матрица
2. Когортный анализ (retention по месяцам)
3. RFM-сегментация (Recency, Frequency, Monetary)
4. Прогноз продаж (Prophet / ARIMA)
5. Рекомендации на основе анализа
Библиотеки: pandas, numpy, matplotlib, seaborn, plotly
Вывод: HTML-отчёт с интерактивными графиками
```

Machine Learning pipeline

```
> Создай ML-pipeline для предсказания оттока клиентов:
1. Data loading (pandas)
2. Feature engineering:
- Дни с последнего заказа
- Средний чек
- Частота заказов
- Категории товаров (one-hot)
3. Preprocessing:
- Handle missing values
- Scale numerical features
- Encode categorical
4. Model training:
- Train/test split (80/20)
- Cross-validation (5-fold)
- Модели: Logistic Regression, Random Forest, XGBoost
- Hyperparameter tuning (Optuna)
5. Evaluation:
- Classification report
- ROC-AUC curve
- Feature importance
6. Deployment:
- Serialize model (joblib)
- FastAPI endpoint для inference
- Docker container
```

18.3. Системы реального времени

WebSocket-сервер

```
> Создай real-time чат с WebSocket:
Сервер (Node.js + ws):
- Комнаты (rooms)
- Приватные сообщения
- Статус "печатает…" (typing indicator)
- История сообщений (последние 100)
- Reconnection с восстановлением пропущенных сообщений
Клиент (React):
- Список комнат
- Чат с автоскроллом
- Отправка изображений (base64)
- Индикатор онлайн-пользователей
- Push-уведомления (Notification API)
Масштабирование: Redis Pub/Sub для нескольких серверов
```

Event-driven архитектура

```
> Создай event-driven систему обработки заказов:
Events:
- OrderCreated -> validate stock, calculate total
- OrderPaid -> decrease stock, send confirmation
- OrderShipped -> update tracking, notify customer
- OrderDelivered -> request review, update stats
Message broker: Redis Streams (или RabbitMQ)
Handlers: отдельные микросервисы
Saga pattern: для координации между сервисами
Dead Letter Queue: для failed events
Idempotency: through event ID deduplication
```

18.4. Блокчейн и Web3

```
> Создай смарт-контракт на Solidity:
NFT-маркетплейс:
- ERC-721 для NFT
- Листинг: выставить на продажу
- Покупка: перевод ETH -> перевод NFT
- Аукцион: ставки с таймером
- Royalties: автоотчисление автору при перепродаже
Используй: Hardhat, OpenZeppelin, ethers.js
Тесты: Chai + Hardhat test
```

```
Deploy script: для Sepolia testnet
```

18.5. IoT и встраиваемые системы

```
> Создай IoT-систему мониторинга на базе MQTT:
Сенсоры (симуляция на Python):
- Температура
- Влажность
- Давление
Broker: Mosquitto (Docker)
Backend (Node.js):
- Подписка на MQTT-топики
- Сохранение в TimescaleDB
- REST API для получения данных
- WebSocket для real-time графиков
Dashboard (React):
- Графики в реальном времени
- Алерты при выходе за пороговые значения
- История за период
```

18.6. Игровая разработка

```
> Создай 2D-игру "Космический шутер" на HTML5 Canvas:
Механики:
- Корабль управляется мышью/клавиатурой
- Стрельба (пробел)
- Волны врагов (увеличивающаяся сложность)
- Power-ups (щит, усиленная стрельба, замедление)
- Score и High Score (localStorage)
- Particles для эффектов
- Звуковые эффекты (Web Audio API)
Оптимизация:
- RequestAnimationFrame
- Object pooling
- Collision detection (AABB)
- Sprite sheet animation
```

18.7. Обработка естественного языка

```
> Создай сервис анализа текста:
Endpoints:
- POST /api/sentiment – определение тональности
- POST /api/summarize – суммаризация текста
- POST /api/classify – классификация по категориям
- POST /api/extract – извлечение сущностей (NER)
```

```
Варианты реализации:
1. Через Claude API (Anthropic)
2. Через open-source модели (Hugging Face Transformers)
Python + FastAPI, batch processing, Redis caching.
```

18.8. Финтех

```
> Создай систему расчёта кредитного скоринга:
Входные данные:
- Возраст, доход, стаж работы
- Кредитная история (платежи за 12 месяцев)
- Текущие обязательства
- Тип занятости
Модель:
- Взвешенная скоринговая карта
- Каждый параметр -> баллы (0-100)
- Итоговый скор: 300-850
Категории решения:
- 750+: одобрено
- 650-749: одобрено с повышенной ставкой
- 500-649: требует ручной проверки
- <500: отказ
API + Dashboard + PDF-отчёт.
```

Итоги главы

В этой главе мы рассмотрели применение Claude Code в различных предметных областях:

– Мобильная разработка: React Native, Flutter, Swift
– Data Science и ML: анализ данных, ML-pipelines
– Системы реального времени: WebSocket, event-driven
– Блокчейн и Web3: смарт-контракты
– IoT: MQTT, сенсоры, мониторинг
– Игровая разработка: HTML5 Canvas
– NLP: анализ текста, классификация
– Финтех: скоринг, финансовые расчёты

С этой главой мы завершаем третью часть книги – «Продвинутые возможности». В четвёртой части мы перейдём к реальным проектам: пошаговым руководствам по созданию полноценных приложений.

Часть IV. Реальные проекты

Глава 19. Проект: SaaS-приложение с нуля

19.1. Постановка задачи

В этой главе мы создадим полноценное SaaS-приложение для управления проектами – «TaskFlow». Это не учебный пример и не MVP: мы реализуем приложение продакшен-класса с аутентификацией, мультитенантностью, real-time обновлениями и интеграциями.

Цель главы – показать, как Claude Code помогает на каждом этапе: от идеи до развёртывания.

Функциональные требования

Управление проектами:

– Создание проектов с описанием и дедлайнами
– Канбан-доска с drag-and-drop
– Списки задач с приоритетами и тегами
– Вложенные подзадачи (до 3 уровней)
– Назначение исполнителей
– Файловые вложения

Командная работа:

– Приглашение участников по email
– Роли: владелец, админ, участник, наблюдатель
– Комментарии к задачам
– Уведомления в реальном времени
– Activity feed

Аналитика:

– Дашборд с метриками (завершённые/открытые/просроченные)
– Burndown chart
– Velocity по спринтам
– Время в каждом статусе

Технологический стек

```
> Предложи технологический стек для SaaS-приложения
```

```
управления проектами. Обоснуй каждый выбор.
```

Claude Code предлагает:

Компонент | Технология | Обоснование
Frontend | Next.js 14 (App Router) | SSR, RSC, оптимизация из коробки
UI | Tailwind CSS + Radix UI | Гибкость + доступность
State | Zustand + React Query | Простота + серверный state
Backend | tRPC | Type safety от БД до UI
ORM | Prisma | Типизация, миграции, studio
Database | PostgreSQL 16 | ACID, FTS, JSONB
Auth | NextAuth.js v5 | OAuth, JWT, sessions
Real-time | Pusher | Managed WebSocket
Storage | S3 + CloudFront | Файлы + CDN
Hosting | Vercel + Supabase | Serverless + Managed DB
CI/CD | GitHub Actions | Интеграция с GitHub

19.2. Инициализация проекта

```
> Инициализируй Next.js 14 проект для TaskFlow:
- TypeScript (strict mode)
- App Router
- Tailwind CSS
- ESLint + Prettier
- Husky + lint-staged
- Vitest для тестов
Структура:
src/
+- app/ # Next.js страницы и layouts
+- components/ # React компоненты
| +- ui/ # Базовые UI (Button, Input, etc.)
| +- features/ # Бизнес-компоненты
| +- layouts/ # Layouts
+- server/ # Серверная логика
| +- routers/ # tRPC роутеры
| +- services/ # Бизнес-логика
| +- db/ # Prisma
+- lib/ # Утилиты
+- hooks/ # React хуки
+- types/ # TypeScript типы
+- styles/ # Глобальные стили
```

19.3. Модели данных

```
> Создай Prisma schema для TaskFlow:
Модели: Organization, User, Membership, Project, Board,
Column, Task, Comment, Tag, Attachment, Notification,
```

```
ActivityLog
Отношения:
- Organization -> many Users (через Membership)
- Organization -> many Projects
- Project -> many Boards
- Board -> many Columns -> many Tasks
- Task -> many Comments, Tags, Attachments
- User -> many Notifications
Поля:
- Все даты: createdAt, updatedAt
- Soft delete: deletedAt где нужно
- UUID для ID
- Enum для статусов
```

Claude Code создаст полную Prisma-схему на 200+ строк с продуманными отношениями, индексами и ограничениями.

19.4. Аутентификация

```
> Настрой NextAuth.js v5 для TaskFlow:
Провайдеры:
- Email (Magic Link через Resend)
- Google OAuth
- GitHub OAuth
После первого входа:
- Создать Organization (или присоединиться по инвайту)
- Создать Membership
- Редирект на /dashboard
Middleware:
- Защита /dashboard/* - только авторизованные
- /auth/* - только неавторизованные
- / - редирект на /dashboard если авторизован
```

19.5. Канбан-доска

```
> Реализуй канбан-доску:
Frontend:
- @dnd-kit для drag & drop
- Колонки с задачами
- Drag между колонками (меняет статус)
- Drag внутри колонки (меняет порядок)
- Добавление задачи (inline form)
- Клик на задачу -> боковая панель с деталями
- Фильтры: по исполнителю, приоритету, тегу
Backend:
- tRPC mutation для перемещения
- Оптимистичные обновления (React Query)
```

```
- Real-time синхронизация (Pusher)
```

19.6. Real-time обновления

```
> Реализуй real-time синхронизацию через Pusher:
События:
- task:created - новая задача
- task:updated - обновление задачи
- task:moved - перемещение между колонками
- task:deleted - удаление
- comment:created - новый комментарий
- member:joined - новый участник
На клиенте:
- React-хук usePusher
- Автоподписка при входе на доску
- Обновление React Query cache при событии
- Уведомление toast при изменениях от других пользователей
```

19.7. Деплой

```
> Подготовь проект к деплою:
1. Vercel: настрой vercel.json
2. Supabase: подключи PostgreSQL
3. Environment variables: всё в .env.example
4. GitHub Actions: CI pipeline
5. Domain: настрой custom domain
6. Мониторинг: Vercel Analytics + Sentry
```

19.8. Уроки проекта

Создание TaskFlow демонстрирует ключевые принципы работы с Claude Code на реальном проекте:

– **Планирование перед кодированием** – каждый диалог начинается с «расскажи, что собираешься делать»
– **Инкрементальная разработка** – одна фича за раз
– **Type Safety** – TypeScript от БД до UI
– **Тесты на каждом шаге** – не откладывать на потом
– **Review AI-кода** – проверять каждый файл

Итоги главы

В этой главе мы создали SaaS-приложение TaskFlow, пройдя путь от идеи до деплоя. Мы использовали Claude Code на каждом этапе: проектирование, скаффолдинг, реализация,

тестирование, деплой. Каждый шаг был диалогом с AI – не магией, а методичной совместной работой.

Глава 20. Проект: REST API для маркетплейса

20.1. Постановка задачи

В этой главе мы построим бэкенд для маркетплейса – платформы, где продавцы размещают товары, а покупатели их приобретают. Мы сосредоточимся на бэкенде: проектирование API, модели данных, бизнес-логика, платежи и масштабирование.

Бизнес-модель

MarketPlace – это C2C (Customer-to-Customer) платформа:

– **Продавцы** создают магазины и размещают товары
– **Покупатели** ищут, сравнивают и покупают
– **Платформа** обрабатывает платежи и берёт комиссию (5%)
– **Доставка** через интеграцию с логистическими компаниями

Технологический стек

Компонент | Технология
Runtime | Node.js 20
Framework | Express.js
Language | TypeScript 5.4
Database | PostgreSQL 16
ORM | Prisma
Cache | Redis 7
Queue | BullMQ
Auth | JWT + Passport.js
Validation | Zod
Docs | Swagger (OpenAPI 3.1)
Testing | Vitest + Supertest
Container | Docker + Docker Compose

20.2. Архитектура API

```
> Спроектируй REST API для маркетплейса:
Ресурсы:
- Auth: register, login, refresh, forgot-password
- Users: profile, settings
- Stores: CRUD, верификация продавца
- Products: CRUD, поиск, фильтрация, категории
- Cart: add, remove, update quantity, checkout
```

```
    - Orders: create, status, history
    - Reviews: create, read, moderate
    - Payments: process, refund, payout to seller
    - Notifications: list, mark read
    Для каждого ресурса: endpoints, HTTP-методы, request/response,
    коды ответа, пагинация, фильтрация, сортировка.
```

Claude Code спроектирует RESTful API с вложенными ресурсами:

```
    POST /api/v1/auth/register
    POST /api/v1/auth/login
    POST /api/v1/auth/refresh
    POST /api/v1/auth/forgot-password
    GET                                          /api/v1/products?
category=electronics&sort=price_asc&page=1
    GET /api/v1/products/:id
    POST /api/v1/stores/:storeId/products
    PUT /api/v1/stores/:storeId/products/:id
    DELETE /api/v1/stores/:storeId/products/:id
    POST /api/v1/cart/items
    PUT /api/v1/cart/items/:id
    DELETE /api/v1/cart/items/:id
    POST /api/v1/cart/checkout
    GET /api/v1/orders
    GET /api/v1/orders/:id
    POST /api/v1/orders/:id/cancel
    POST /api/v1/products/:id/reviews
    GET /api/v1/products/:id/reviews
```

20.3. Модели данных

```
    > Создай Prisma schema для маркетплейса.
    Ключевые модели:
    - User, Store, Product, Category, ProductImage
    - Cart, CartItem
    - Order, OrderItem
    - Review, Rating
    - Payment, Payout
    - Notification
    Особенности:
    - Полнотекстовый поиск по Products
    - JSONB для product attributes (размер, цвет, материал)
    - Денежные значения: Decimal(10, 2)
    - Enum для статусов (OrderStatus, PaymentStatus)
    - Soft delete для Products и Reviews
```

20.4. Бизнес-логика

Система корзины

```
> Реализуй CartService:
- addItem: проверить наличие, добавить или увеличить количество
- removeItem: удалить из корзины
- updateQuantity: обновить количество (проверить stock)
- getCart: получить корзину с ценами и итогом
- checkout: создать заказ из корзины
Edge cases:
- Товар закончился между добавлением и checkout
- Цена изменилась после добавления в корзину
- Корзина не должна содержать товары из неактивных магазинов
```

Система заказов

```
> Реализуй OrderService:
Flow:
1. Checkout -> создать заказ (status: pending)
2. Payment -> оплата через Stripe (status: paid)
3. Seller confirms -> (status: processing)
4. Shipped -> (status: shipped, tracking number)
5. Delivered -> (status: delivered)
6. Review window -> 14 дней для отзыва и возврата
Автоматические действия:
- При оплате: уведомить продавца
- При отправке: уведомить покупателя + трекинг
- При доставке: запросить отзыв через 3 дня
- Через 30 дней: выплатить продавцу (payout)
Отмена:
- Покупатель: до отправки (полный refund)
- После отправки: через dispute (частичный refund)
```

Система платежей

```
> Реализуй PaymentService (Stripe Connect):
Модель: Marketplace payments
- Покупатель платит через Stripe Checkout
- Stripe удерживает деньги
- После подтверждения доставки -> transfer продавцу
- Комиссия платформы: 5% (application_fee)
Функции:
- processPayment: создать Stripe Checkout session
```

```
- handleWebhook: обработать Stripe webhook events
- createPayout: перевести на счёт продавца
- processRefund: возврат средств покупателю
```

20.5. Поиск и фильтрация

```
> Реализуй поиск товаров:
1. Полнотекстовый поиск (PostgreSQL FTS):
- Поиск по названию и описанию
- Поддержка русского и английского языков
- Ранжирование по релевантности
2. Фильтрация:
- По категории (иерархическая)
- По цене (min/max)
- По рейтингу (>= N)
- По атрибутам (размер, цвет)
- По наличию (in_stock)
- По магазину
- По местоположению (геопоиск)
3. Сортировка:
- По релевантности
- По цене (asc/desc)
- По рейтингу
- По новизне
- По популярности (количество заказов)
4. Пагинация: cursor-based для стабильности
```

20.6. Тестирование API

```
> Напиши интеграционные тесты для API:
Setup:
- Testcontainers: PostgreSQL + Redis
- Seed data: категории, тестовые пользователи и товары
- Auth helper: генерация JWT для разных ролей
Тесты:
1. Product CRUD (создание, получение, обновление, удаление)
2. Поиск и фильтрация (комбинации фильтров)
3. Cart flow (добавление -> checkout -> заказ)
4. Payment webhook handling
5. Authorization (продавец не может менять чужие товары)
6. Rate limiting
7. Validation (невалидные данные)
Coverage target: 85%
```

20.7. Масштабирование

```
> Подготовь API к масштабированию:
1. Кэширование:
- Каталог товаров: Redis, TTL 5 min
- Карточка товара: Redis, TTL 10 min, invalidate on update
- Категории: Redis, TTL 1 hour
2. Фоновые задачи (BullMQ):
- Обработка изображений (resize, optimize)
- Отправка email-уведомлений
- Расчёт рейтингов
- Генерация отчётов
3. Мониторинг:
- Prometheus метрики
- Health check endpoint
- Error tracking (Sentry)
4. Docker:
- Multi-stage build
- Docker Compose с всеми сервисами
- nginx reverse proxy
```

Итоги главы

В этой главе мы построили бэкенд маркетплейса – сложное бизнес-приложение с десятками моделей, интеграцией с платёжной системой и продуманной архитектурой. Claude Code помог на каждом этапе: от проектирования API до написания тестов и подготовки к масштабированию.

Глава 21. Проект: Telegram-бот на Python

21.1. Постановка задачи

Telegram-боты – одна из самых популярных разработок среди русскоязычных программистов. В этой главе мы создадим бота-ассистента для управления личными финансами: учёт расходов, категоризация, бюджеты и аналитика.

Функциональные требования

Учёт транзакций:

– Быстрый ввод расхода: `500 кофе` или `1200 такси`
– Автоматическая категоризация (кофе -> Еда, такси -> Транспорт)
– Ввод доходов: `/income 80000 Зарплата`
– Редактирование и удаление последних записей

Бюджеты:

– Установка месячного бюджета по категориям
– Предупреждение при приближении к лимиту (80%)
– Уведомление при превышении

Аналитика:

– Расходы за день / неделю / месяц
– Диаграмма расходов по категориям
– Тренд расходов (рост/снижение)
– Сравнение с прошлым месяцем

Технологический стек

Компонент | Технология
Language | Python 3.12
Framework | aiogram 3
Database | SQLite (через aiosqlite)
ORM | SQLAlchemy 2.0 (async)
Charts | matplotlib
Deploy | Docker + systemd

21.2. Структура проекта

```
> Создай структуру проекта для Telegram-бота:
finance_bot/
+- bot/
| +- __init__.py
| +- main.py # Точка входа
| +- config.py # Конфигурация
| +- handlers/ # Обработчики сообщений
| | +- __init__.py
| | +- start.py # /start, /help
| | +- expenses.py # Учёт расходов
| | +- income.py # Учёт доходов
| | +- budget.py # Управление бюджетом
| | +- analytics.py # Отчёты и графики
| | +- settings.py # Настройки
| +- keyboards/ # Клавиатуры
| | +- __init__.py
| | +- inline.py
| +- services/ # Бизнес-логика
| | +- __init__.py
| | +- expense_service.py
| | +- budget_service.py
| | +- analytics_service.py
| | +- categorizer.py # Автокатегоризация
| +- models/ # SQLAlchemy модели
| | +- __init__.py
| | +- models.py
| +- db/ # Database
| | +- __init__.py
| | +- database.py
| +- utils/ # Утилиты
| +- __init__.py
| +- charts.py # Генерация графиков
+- tests/
+- alembic/
+- Dockerfile
+- docker-compose.yml
+- pyproject.toml
+- .env.example
```

21.3. Обработчики сообщений

```
> Реализуй обработчик расходов для aiogram 3:
Форматы ввода:
- "500 кофе" -> сумма + описание
- "1200" -> только сумма, запросить категорию
```

```
    - "/expense 500 Еда Обед в кафе" -> сумма + категория + описание
    Автокатегоризация:
    - "кофе", "обед", "ужин", "продукты" -> Еда
    - "такси", "метро", "бензин" -> Транспорт
    - "аренда", "коммуналка" -> Жильё
    - Неизвестное -> показать inline-клавиатуру с категориями
    После записи:
    - Подтверждение с деталями
    - Кнопки: [-] Удалить, Изменить категорию
    - Показать остаток бюджета категории (если установлен)
```

Claude Code реализует:

```
    from aiogram import Router, F
    from aiogram.types import Message, CallbackQuery
    from aiogram.filters import Command
    import re
    router = Router()
    @router.message(F.text.regexp(r'^(\d+[\.,]?\d*)\s*(.*)$'))
    async def handle_expense(message: Message, session:
AsyncSession):
    """Обработка быстрого ввода расхода: '500 кофе'"""
    match = re.match(r'^(\d+[\.,]?\d*)\s*(.*)$', message.text)
    amount = float(match.group(1).replace(',', '.'))
    description = match.group(2).strip()
    category = await categorizer.detect(description)
    if category:
    expense = await expense_service.create(
    user_id=message.from_user.id,
    amount=amount,
    category=category,
    description=description,
    )
    budget_info = await budget_service.check(
    message.from_user.id, category
    )
    budget_text = ""
    if budget_info:
    remaining = budget_info.limit - budget_info.spent
    percent = (budget_info.spent / budget_info.limit) * 100
    if percent >= 100:
    budget_text = f"\n[!] Бюджет на {category} превышен!"
    elif percent >= 80:
    budget_text = f"\n[!] Осталось {remaining:.0f} из бюджета"
    await message.reply(
    f"[+] Записано: {amount:.0f} - {category}\n"
    f" {description}{budget_text}",
    reply_markup=expense_keyboard(expense.id)
    )
```

```
else:
await message.reply(
f" {amount:.0f} - {description}\n"
f"Выберите категорию:",
reply_markup=category_keyboard(amount, description)
)
```

21.4. Аналитика и графики

```
> Реализуй генерацию графиков для аналитики:
/stats - расходы за текущий месяц:
- Pie chart: распределение по категориям
- Bar chart: расходы по дням
- Сравнение с прошлым месяцем (процентное изменение)
/trend - тренд расходов:
- Line chart: расходы по месяцам (последние 6)
- Средняя линия
- Прогноз на текущий месяц
Стиль графиков: тёмная тема (для Telegram)
Формат: PNG, 800x600px
```

21.5. Деплой бота

```
> Подготовь бота к деплою:
1. Dockerfile (Python 3.12-slim)
2. docker-compose.yml
3. systemd unit file (если без Docker)
4. Webhook vs Polling (настрой оба варианта)
5. Мониторинг: health check, error logging
6. Graceful shutdown
7. Автозапуск при перезагрузке
```

Итоги главы

В этой главе мы создали Telegram-бота для управления финансами – практичное приложение, которое решает реальную задачу. Claude Code помог написать обработчики aiogram, бизнес-логику, генерацию графиков и подготовку к деплою.

Глава 22. Проект: CLI-утилита на Go

22.1. Постановка задачи

Командная строка остаётся главным инструментом разработчика. В этой главе мы создадим CLI-утилиту «DevTool» – швейцарский нож для повседневных задач разработки: генерация проектов, управление .env файлами, проверка здоровья сервисов и быстрый деплой.

Функциональность

```
devtool init <template> # Инициализация проекта
devtool env sync # Синхронизация .env файлов
devtool env diff staging prod # Сравнение окружений
devtool health check <url> # Проверка health endpoint
devtool health monitor <url> # Мониторинг в реальном времени
devtool deploy staging # Деплой на staging
devtool deploy production # Деплой на production
devtool db migrate # Миграции БД
devtool db backup # Бэкап БД
devtool logs <service> # Просмотр логов
devtool config # Конфигурация утилиты
```

Технологический стек

Компонент | Технология
Language | Go 1.22
CLI | cobra
Config | viper
TUI | bubbletea + lipgloss
HTTP | net/http (stdlib)
SSH | golang.org/x/crypto/ssh
Build | GoReleaser

22.2. Структура проекта

```
> Создай структуру Go CLI-проекта:
devtool/
+- cmd/
| +- root.go # Корневая команда
| +- init.go # devtool init
| +- env.go # devtool env
| +- health.go # devtool health
```

```
| +- deploy.go # devtool deploy
| +- db.go # devtool db
| +- logs.go # devtool logs
| +- config.go # devtool config
+- internal/
| +- config/ # Конфигурация
| +- env/ # Работа с .env файлами
| +- health/ # Health checks
| +- deploy/ # Деплой
| +- db/ # Миграции и бэкапы
| +- logs/ # Просмотр логов
| +- tui/ # Интерактивный UI
| +- utils/ # Утилиты
+- templates/ # Шаблоны проектов
+- main.go
+- go.mod
+- go.sum
+- Makefile
+- .goreleaser.yml
+- README.md
```

22.3. Команда init

```
> Реализуй команду devtool init:
Шаблоны:
- nextjs: Next.js + TypeScript + Tailwind
- express: Express + TypeScript + Prisma + Docker
- fastapi: FastAPI + SQLAlchemy + Docker
- go-api: Go + Chi + PostgreSQL + Docker
При инициализации:
1. Выбор шаблона (интерактивно или аргументом)
2. Ввод имени проекта
3. Скаффолдинг файлов
4. git init
5. Установка зависимостей
6. Вывод next steps
Интерактивный режим: bubbletea TUI с выбором шаблона.
```

Claude Code реализует:

```
package cmd
import (
"fmt"
"github.com/spf13/cobra"
tea "github.com/charmbracelet/bubbletea"
"github.com/charmbracelet/lipgloss"
)
var initCmd = &cobra.Command{
```

```
    Use: "init [template]",
    Short: "Initialize a new project from template",
    Long: "Create a new project structure from predefined
templates",
    Args: cobra.MaximumNArgs(1),
    RunE: func(cmd *cobra.Command, args []string) error {
    var template string
    if len(args) > 0 {
    template = args[0]
    } else {
    // Интерактивный выбор через TUI
    p := tea.NewProgram(newTemplateSelector())
    m, err := p.Run()
    if err != nil {
    return fmt.Errorf("template selection failed: %w", err)
    }
    template = m.(templateModel).selected
    }
    return scaffold(template)
    },
    }
    func init() {
    rootCmd.AddCommand(initCmd)
    }
```

22.4. Команда health

```
    > Реализуй команду devtool health:
    ## devtool health check <url>
    Проверяет health endpoint и выводит:
    - Статус: [+] Healthy / [-] Unhealthy
    - Response time
    - HTTP status code
    - Body (если JSON - parsed)
    ## devtool health monitor <url>
    Мониторинг в реальном времени (TUI):
    - Опрос каждые 5 секунд
    - График response time (ASCII)
    - Уведомление при downtime
    - История за сессию
    - Ctrl+C для остановки
    Флаги:
    --interval 5s # Интервал проверки
    --timeout 10s # Таймаут запроса
    --header "Auth: Bearer xxx" # Кастомные заголовки
```

22.5. Команда env

```
> Реализуй управление окружениями:
## devtool env sync
Синхронизирует .env.example с .env:
- Находит новые переменные в .env.example
- Запрашивает значения интерактивно
- Добавляет в .env
- Предупреждает об удалённых переменных
## devtool env diff <env1> <env2>
Сравнивает два .env файла:
- Показывает различия (без значений!)
- Переменные только в env1
- Переменные только в env2
- Разные (маскированно)
## devtool env validate
Проверяет .env на:
- Пустые обязательные переменные
- Формат (DATABASE_URL - валидный URL?)
- Дубликаты
```

22.6. Деплой

```
> Реализуй команду devtool deploy:
Конфигурация (.devtool.yml):
deploy:
staging:
host: staging.example.com
user: deploy
path: /opt/app
branch: develop
pre_deploy:
- "docker compose pull"
deploy:
- "docker compose up -d"
post_deploy:
- "docker compose exec app npm run migrate"
health_check: "https://staging.example.com/health"
production:
host: prod.example.com
user: deploy
path: /opt/app
branch: main
requires_confirmation: true
Flow:
1. Проверка: ветка, uncommitted changes, тесты
2. Pre-deploy commands (SSH)
```

```
3. Deploy commands (SSH)
4. Post-deploy commands (SSH)
5. Health check
6. Rollback при failure
7. Уведомление (webhook)
```

22.7. Сборка и дистрибуция

```
> Настрой GoReleaser для cross-platform сборки:
Платформы: linux/amd64, linux/arm64, darwin/amd64,
darwin/arm64, windows/amd64
Дистрибуция:
- GitHub Releases (бинарники + checksums)
- Homebrew tap
- Scoop bucket (Windows)
- Docker image
- AUR пакет (Arch Linux)
CI: GitHub Actions -> автоматический релиз при git tag v*
```

Итоги главы

В этой главе мы создали CLI-утилиту на Go – инструмент, который каждый разработчик может настроить под свои нужды. Claude Code помог реализовать cobra-команды, TUI-интерфейс с bubbletea, SSH-деплой и кросс-платформенную сборку.

Глава 23. Проект: Chrome-расширение

23.1. Постановка задачи

Chrome-расширения – мощный инструмент кастомизации браузера. В этой главе мы создадим расширение «DevBookmarks» – менеджер закладок для разработчиков с хранением сниппетов кода, заметок к страницам и интеграцией с GitHub.

Функциональные требования

– Сохранение закладки с заметкой и тегами
– Сохранение фрагмента кода со страницы (выделение -> правый клик -> сохранить)
– Быстрый поиск по закладкам и сниппетам (popup)
– Группировка по проектам и тегам
– Синхронизация через Chrome Storage Sync
– Экспорт в Markdown / GitHub Gist

Архитектура расширения

```
> Создай Chrome-расширение Manifest V3:
devbookmarks/
+- manifest.json # Конфигурация (MV3)
+- popup/
| +- popup.html # UI попапа
| +- popup.css
| +- popup.js
+- sidepanel/
| +- sidepanel.html # Боковая панель
| +- sidepanel.css
| +- sidepanel.js
+- background/
| +- service-worker.js # Background service worker
+- content/
| +- content.js # Content script
+- options/
| +- options.html # Страница настроек
| +- options.css
| +- options.js
+- icons/
| +- icon16.png
| +- icon48.png
| +- icon128.png
+- lib/
```

```
+- storage.js # Работа с Chrome Storage
+- github.js # GitHub API
```

23.2. Manifest и конфигурация

```
> Создай manifest.json для DevBookmarks:
Permissions:
- storage (chrome.storage.sync)
- activeTab (для content script)
- contextMenus (правый клик)
- sidePanel (боковая панель)
- identity (OAuth для GitHub)
Content scripts: на всех страницах (код выделения)
Background: service worker
Popup: 400x600px

{
"manifest_version": 3,
"name": "DevBookmarks",
"version": "1.0.0",
"description": "Менеджер закладок для разработчиков",
"permissions": [
"storage",
"activeTab",
"contextMenus",
"sidePanel",
"identity"
],
"action": {
"default_popup": "popup/popup.html",
"default_icon": {
"16": "icons/icon16.png",
"48": "icons/icon48.png",
"128": "icons/icon128.png"
}
},
"background": {
"service_worker": "background/service-worker.js"
},
"content_scripts": [{
"matches": ["<all_urls>"],
"js": ["content/content.js"],
"css": ["content/content.css"]
}],
"side_panel": {
"default_path": "sidepanel/sidepanel.html"
},
"options_page": "options/options.html"
```

```
}
```

23.3. Popup – быстрый доступ

```
> Реализуй popup расширения:
Верхняя часть:
- Поле поиска (мгновенная фильтрация)
- Кнопка "Сохранить эту страницу"
Средняя часть:
- Список последних закладок (10 шт)
- Каждая: title, URL, теги, дата
- Клик -> открыть URL
- Иконка удаления
Нижняя часть:
- Кнопка "Все закладки" (открывает side panel)
- Кнопка "Настройки"
Стиль: тёмная тема, компактный
```

23.4. Контекстное меню и сниппеты

```
> Реализуй контекстное меню:
При выделении текста -> правый клик:
"DevBookmarks: Сохранить сниппет"
При нажатии:
1. Определить язык (по URL или содержимому страницы)
2. Сохранить: код, URL страницы, язык, дата
3. Показать уведомление "Сниппет сохранён"
Бонус: если выделенный текст – код из GitHub/StackOverflow,
автоматически определить язык программирования.
```

23.5. Синхронизация и хранение

```
> Реализуй storage layer:
chrome.storage.sync:
- Лимит: 100KB
- Стратегия: хранить метаданные (title, url, tags, date)
chrome.storage.local:
- Лимит: 5MB
- Стратегия: хранить сниппеты кода, заметки
CRUD операции:
- addBookmark(bookmark)
- getBookmarks(filters)
- updateBookmark(id, updates)
- deleteBookmark(id)
- searchBookmarks(query)
```

```
- exportToMarkdown()
- exportToGist(token)
```

23.6. Интеграция с GitHub

```
> Реализуй экспорт в GitHub Gist:
1. OAuth через chrome.identity:
- GitHub OAuth App
- Scopes: gist
2. Создание Gist:
- Все сниппеты -> файлы в Gist
- Группировка по языку
- Description: "DevBookmarks export - {date}"
3. Обновление Gist:
- При повторном экспорте -> обновить существующий
```

23.7. Тестирование расширения

```
> Напиши тесты для расширения:
Unit-тесты (Vitest):
- Storage layer (мок chrome.storage)
- Search/filter функции
- Export в Markdown
Integration тесты:
- Puppeteer + chrome-extension-testing
- Загрузка расширения
- Клик по popup
- Сохранение закладки
- Поиск
```

Итоги главы

В этой главе мы создали Chrome-расширение – тип приложения, который отличается от обычной веб-разработки архитектурой, API и ограничениями. Claude Code помог освоить Manifest V3, Chrome APIs и специфику разработки расширений.

Глава 24. Проект: Миграция легаси-кода

24.1. Постановка задачи

Работа с legacy-кодом – реальность, с которой сталкивается каждый разработчик. В этой главе мы проведём миграцию реального (условно) legacy-приложения: Express.js 4 на JavaScript -> Next.js 14 на TypeScript.

Это не абстрактное упражнение – мы пройдём через все типичные проблемы: отсутствие документации, изиптивный код, устаревшие зависимости, отсутствие тестов, неявные зависимости между модулями.

Исходное приложение

```
legacy-app/
+- server.js # 800 строк, Express + Pug
+- routes/
| +- index.js # 500 строк
| +- users.js # 400 строк
| +- api.js # 600 строк
+- models/
| +- user.js # Mongoose
| +- product.js # Mongoose
+- views/ # Pug шаблоны
+- public/ # jQuery + Bootstrap 3
+- helpers/ # Утилиты (200-300 строк каждый)
+- config.js # Конфигурация (hardcoded)
+- package.json # 30 зависимостей, многие deprecated
+- NO TESTS # Нет ни одного теста
```

Характеристики легаси:

– Node.js 12, npm scripts
– MongoDB через Mongoose (без миграций)
– Pug-шаблоны (server-side rendering)
– jQuery для интерактивности
– Bootstrap 3
– Callback-based async
– Нет TypeScript, нет линтера
– `var` везде, inconsistent code style
– 4 года без обновлений зависимостей

Цель миграции

Аспект | Было | Стало
Runtime | Node 12 | Node 20
Framework | Express 4 | Next.js 14
Language | JavaScript | TypeScript
Database | MongoDB | PostgreSQL
ORM | Mongoose | Prisma
Frontend | Pug + jQuery | React (RSC)
CSS | Bootstrap 3 | Tailwind CSS
Tests | Нет | Vitest + Playwright
CI/CD | Нет | GitHub Actions

24.2. Фаза 1: Исследование

```
> Проанализируй этот legacy-проект:
1. Общая статистика:
- Строк кода, файлов, функций
- Изипортированные модули и зависимости
2. Архитектура:
- Как организован код (layers, modules)
- Основные потоки данных
- Модели данных
3. Проблемы:
- Устаревшие зависимости
- Security vulnerabilities (npm audit)
- Anti-patterns
- Dead code
4. Зависимости между модулями:
- Граф зависимостей
- Circular dependencies
5. Миграционные риски:
- Что будет сложно перенести
- Что потребует переписывания
```

24.3. Фаза 2: Подготовка

Добавление тестов к legacy-коду

```
> Напиши характеристические тесты (characterization tests)
для legacy API:
Для каждого API endpoint:
1. Вызови endpoint с типичными данными
2. Записывай результат (response body, status, headers)
```

```
3. Создай тест, который проверяет этот результат
Цель: зафиксировать текущее поведение,
чтобы при миграции убедиться, что ничего не сломалось.
Используй: supertest + jest
```

Документация модели данных

```
> Проанализируй Mongoose-модели и создай:
1. ER-диаграмму (Mermaid)
2. Описание каждого поля: тип, обязательность, default, validations
3. Неявные отношения (reference by ID в коде, но без ref в схеме)
4. Вложенные документы -> нормализация для PostgreSQL
5. Prisma-схему на основе анализа
```

24.4. Фаза 3: Миграция данных

```
> Создай скрипт миграции данных из MongoDB в PostgreSQL:
Этапы:
1. Читать из MongoDB (коллекция users)
2. Трансформировать:
- ObjectId -> UUID
- Embedded address -> отдельная таблица
- Array of order_ids -> relational join table
- Denormalized fields -> нормализация
3. Записать в PostgreSQL через Prisma
4. Валидация: сравнить количество записей,
spot-check отдельных документов
Обработка ошибок:
- Невалидные данные -> таблица migration_errors
- Дубликаты email -> стратегия merge
- Отсутствующие обязательные поля -> значения по умолчанию
```

24.5. Фаза 4: Поэтапная миграция кода

Strangler Fig Pattern

Мы используем паттерн «Удушающее дерево» (Strangler Fig), который позволяет мигрировать постепенно:

```
> Настрой параллельную работу legacy и нового приложения:
1. Nginx reverse proxy:
- /api/* -> legacy Express (порт 3000)
```

```
- /* -> новый Next.js (порт 3001)
2. По мере миграции endpoints:
- Переносим endpoint в Next.js API Routes
- Меняем правило в Nginx
- Тесты проходят? -> удаляем из Express
3. Когда все endpoints перенесены:
- Удаляем legacy Express
- Убираем proxy
```

Миграция роутов

```
> Перепиши Express роут на Next.js API Route:
Legacy (Express + JavaScript + MongoDB):
[показываем legacy-код]
Target (Next.js + TypeScript + Prisma + Zod):
- Input validation: Zod schema
- Error handling: try/catch -> NextResponse
- Auth: middleware -> NextAuth session
- Database: Mongoose -> Prisma
- Response format: сохранить совместимость (characterization
tests)
```

Миграция фронтенда

```
> Перепиши Pug-шаблон + jQuery в React-компонент:
Legacy:
- Pug template: users/list.pug
- jQuery для динамической таблицы
- Bootstrap 3 классы
Target:
- React Server Component (Next.js)
- Tailwind CSS вместо Bootstrap
- Server-side data fetching (вместо AJAX)
- TypeScript типы для props
```

24.6. Фаза 5: Верификация

```
> Создай набор проверок для верификации миграции:
1. Характеристические тесты (из фазы 2) проходят?
2. Данные мигрированы корректно?
- Количество записей совпадает
- Spot-check 100 записей
- Все связи сохранены
3. Производительность:
- Latency сравнима или лучше
```

```
- Load test: 100 RPS без ошибок
4. Функциональность:
- Все страницы рендерятся
- Формы работают
- Авторизация работает
- Email-ы отправляются
```

24.7. Уроки миграции

– **Тесты перед миграцией** – никогда не мигрируйте без тестов
– **Поэтапно** – Strangler Fig, не Big Bang
– **Данные отдельно** – миграция данных миграция кода
– **Claude Code – археолог** – он отлично понимает legacy-код
– **Не улучшайте во время миграции** – сначала перенесите as-is, потом рефакторьте
– **Rollback plan** – всегда имейте возможность откатить

Итоги главы

В этой главе мы провели полную миграцию legacy-приложения – задачу, которая в реальности может занять месяцы. Claude Code ускорил каждый этап: анализ legacy-кода, написание характеристических тестов, миграция данных, поэтапная перезапись кода, верификация. Ключевой вывод: не пытайтесь всё переписать за один раз – методичный поэтапный подход с тестами на каждом шаге надёжнее.

Часть V. Лучшие практики и будущее

Глава 25. Лучшие практики и паттерны

25.1. Золотые правила работы с Claude Code

За время работы с Claude Code сообщество выработало набор практик, которые радикально повышают эффективность. Эти правила – квинтэссенция всей книги.

Правило 1: Контекст – это всё

Качество ответа Claude Code прямо пропорционально качеству контекста. Плохой контекст -> плохой результат, и никакая «магия» не спасёт.

Три компонента контекста:

– ЧТО вы делаете – описание задачи
– ГДЕ вы это делаете – проект, технологии, структура
– ЗАЧЕМ вы это делаете – бизнес-цель, motivation

```
> Сделай авторизацию
> Добавь JWT-аутентификацию в Express API.
Проект: e-commerce бэкенд (TypeScript, Prisma, PostgreSQL).
Цель: подготовка к запуску MVP через 2 недели.
Уже есть: модель User с email и password_hash.
Нужно: login/register endpoints, middleware для защиты routes,
refresh token rotation.
```

Правило 2: CLAUDE.md – ваш стратегический актив

Инвестируйте время в CLAUDE.md. Каждая минута, потраченная на его улучшение, сэкономит часы в будущем.

```
## Проект
E-commerce API для продажи электроники. MVP, запуск 15 марта.
## Стек
- TypeScript 5.4, Node.js 20, Express 4
- PostgreSQL 16, Prisma ORM
- Redis (кэш + сессии)
- Docker + Docker Compose
## Принципы
- Функциональный стиль: чистые функции, иммутабельность
```

```
    - Обработка ошибок через Result<T, E>, не через throw
    - Все строки - в переменных окружения, не hardcoded
    - Логирование: structured JSON через pino
    - Тесты: unit (vitest), integration (supertest + testcontainers)
    ## Стиль кода
    - Без классов (only functions and interfaces)
    - Без default exports
    - Без enum (use as const objects)
    - camelCase для переменных, PascalCase для типов
    - Максимальная длина функции: 30 строк
    ## Определения «готово»
    - [ ] Код написан
    - [ ] TypeScript: без ошибок
    - [ ] Тесты: unit + integration
    - [ ] Документация: JSDoc для экспортов
    - [ ] Lint: нет предупреждений
```

Правило 3: Маленькие шаги побеждают

Никогда не просите Claude Code сделать всё за раз. Разбивайте на шаги:

```
> Создай полноценный e-commerce с авторизацией, каталогом,
корзиной, оплатой, доставкой, отзывами и админкой.
> Шаг 1: Создай Prisma-схему для User и Product.
> Шаг 2: Реализуй CRUD для Product.
> Шаг 3: Добавь авторизацию (JWT).
> Шаг 4: Реализуй Cart.
> …
```

Правило 4: Всегда проверяйте

Claude Code – не оракул. Он может:

– Использовать устаревший API
– Пропустить edge case
– Написать неоптимальный запрос
– Не знать о специфике вашей инфраструктуры

Ваша задача – проверять. Запускайте тесты, читайте код, задавайте вопросы.

Правило 5: Итеративное улучшение

Первый результат – это черновик. Улучшайте итеративно:

```
> [Начальный запрос]
-> Результат v1
> Добавь обработку ошибок.
-> Результат v2
> Оптимизируй SQL-запрос, он делает N+1.
-> Результат v3
> Добавь кэширование для getProducts.
-> Результат v4
```

25.2. Антипаттерны

Антипаттерн: «Сделай всё»

```
> Создай полное приложение для учёта расходов:
бэкенд, фронтенд, мобильное приложение,
аналитику и интеграцию с банками.
```

Почему это плохо: контекстное окно ограничено. Чем больше задача, тем хуже каждая часть.

Антипаттерн: «Слепое доверие»

```
> Сгенерируй код -> скопировал -> задеплоил
```

Почему это плохо: вы отвечаете за код, который попадает в production.

Антипаттерн: «Промпт-инжиниринг вместо навыков»

```
Полагаться на Claude Code для всего, не развивая собственные навыки.
```

Claude Code – мультипликатор ваших навыков. 0 × 10 = 0. Чем больше вы знаете, тем эффективнее используете инструмент.

Антипаттерн: «Контекстная амнезия»

```
[Большой диалог, 100+ сообщений, без /compact]
```

Почему это плохо: модель теряет ранний контекст. Используйте `/compact` и начинайте новые диалоги для новых задач.

25.3. Паттерны для разных режимов работы

«Архитектор»

```
> Я проектирую систему. НЕ пиши код.
Обсудим архитектуру:
- Компоненты и их взаимодействие
- Выбор технологий (trade-offs)
- Потенциальные проблемы масштабирования
- Data flow
```

«Наставник»

```
> Объясни, как работает Event Loop в Node.js.
Я понимаю callback queue, но не понимаю microtask queue.
Объясни с примерами. Не используй аналогии
с ресторанами и конвейерами – давай по существу.
```

«Pair Programmer»

```
> Я пишу функцию для расчёта скидки. Вот моя текущая версия:
[код]
Проблемы:
1. Не учитывает накопительные скидки
2. Возможно деление на ноль
Предложи улучшения, но сохрани общую структуру.
```

«Ревьюер»

```
> Проведи code review. Фокус:
- Баги и edge cases
- Производительность
- Безопасность
- Не предлагай стилистические правки
```

25.4. Workflow для команды

1. Стандартизация CLAUDE.md

Команда использует единый CLAUDE.md с общими правилами.

2. Пользовательские команды

Команда создаёт shared commands в `.claude/commands/`:

– `review.md` – code review
– `test.md` – генерация тестов
– `doc.md` – документирование
– `refactor.md` – рефакторинг

3. AI Code Review Policy

– AI-сгенерированный код проходит тот же review, что и обычный
– Автор отвечает за весь код, включая AI-сгенерированный
– В PR указывать: «Written with Claude Code» (опционально)

Итоги главы

Лучшие практики работы с Claude Code:

– **Контекст** – давайте максимум релевантной информации
– **CLAUDE.md** – инвестируйте в настройку
– **Маленькие шаги** – декомпозиция задач
– **Проверка** – всегда верифицируйте результат
– **Итерации** – улучшайте постепенно
– **Антипаттерны** – знайте, чего избегать
– **Режимы** – используйте подходящий стиль общения
– **Команда** – стандартизируйте использование

Глава 26. Ограничения и подводные камни

26.1. Честный разговор об ограничениях

Claude Code – мощнейший инструмент, но не волшебная палочка. Понимание ограничений так же важно, как умение пользоваться возможностями. Разработчик, знающий границы AI, эффективнее того, кто верит в его всемогущество.

26.2. Контекстное окно

Проблема

Каждая модель имеет ограниченный размер контекста – объём информации, которую она «помнит» в рамках одного диалога. Для Claude – это ~200,000 токенов. Звучит много, но на практике это быстро заканчивается:

– Один файл на 1000 строк 4000 токенов
– Проект из 50 файлов 200,000 токенов
– Каждое ваше сообщение + ответ модели – ещё токены

Когда контекст заполняется, модель начинает «забывать» ранние инструкции.

Симптомы

– Claude Code перестаёт следовать правилам из начала диалога
– Повторяет ошибки, которые вы уже исправляли
– «Забывает» архитектурные решения, принятые 20 сообщений назад
– Начинает использовать стиль кода, отличный от установленного

Решения

– **`/compact`** – сжатие контекста (удаление промежуточных сообщений)
– **Новые сессии** – для каждой отдельной задачи
– **CLAUDE.md** – правила загружаются в начале каждой сессии
– **Маленькие диалоги** – решайте одну задачу за раз
– **Не копируйте большие файлы** – ссылайтесь, а не цитируйте

26.3. Галлюцинации

Проблема

Claude Code может уверенно утверждать неправильные вещи:

– Несуществующие API-функции
– Неправильные сигнатуры
– Устаревшие пакеты
– Вымышленные параметры конфигурации

Примеры

```
// Claude Code может написать:
import { useServerAction } from 'next/server';
// Такой функции не существует!
// Или:
const result = await prisma.user.findManyAndCount();
// Такого метода нет в Prisma!
```

Как распознать

– Функция, которую вы никогда не видели в документации
– Пакет, который не находится в npm/PyPI
– Синтаксис, который кажется «слишком удобным»
– Параметр конфигурации, о котором вы не слышали

Решения

– **Проверяйте** – запускайте код, не копируйте вслепую
– **Спрашивайте** – «Ты уверен, что эта функция существует?»
– **Документация** – проверяйте по официальной документации
– **TypeScript** – компилятор поймает несуществующие функции
– **Тесты** – запускайте тесты после каждого изменения

26.4. «Знание» технологий

Сильные стороны

Claude Code хорошо знает:

– JavaScript/TypeScript и экосистему Node.js
– Python и популярные фреймворки
– React, Next.js, Vue, Angular
– SQL, PostgreSQL, MongoDB
– Docker, Kubernetes (базовый)
– Git
– Паттерны проектирования
– Алгоритмы и структуры данных

Слабые стороны

Claude Code знает хуже:

– Очень новые технологии (вышли после даты обучения)
– Нишевые фреймворки с малым комьюнити
– Proprietary инструменты (Bitrix, 1С)
– Глубокая настройка инфраструктуры (kernel tuning)
– Специфика конкретных облачных провайдеров (неочевидные фичи)

Дата среза знаний

Claude Code обучен на данных до определённой даты. Это значит:

– Новые версии библиотек после даты среза – неизвестны
– Новые API – неизвестны
– Deprecated features – могут рекомендоваться как актуальные

Решение: указывайте версии:

```
> Используй React 19 (не 18).
Используй новый use() хук вместо useEffect для fetch.
Не используй forwardRef – в React 19 ref передаётся как обычный prop.
```

26.5. Проблемы с большими проектами

Проблема

Большие проекты (100,000+ строк) не помещаются в контекст целиком. Claude Code видит только те файлы, которые вы явно показываете или которые он сам решает прочитать.

Последствия

– Может не заметить зависимость в другом файле
– Может дублировать существующий код
– Может нарушить неочевидные контракты
– Может использовать стиль, отличный от проекта

Решения

– **CLAUDE.md с путями** – укажите ключевые файлы и структуру
– **Попросите читать файлы** – «Прочитай src/utils/auth.ts перед тем, как менять авторизацию»
– **Показывайте интерфейсы** – «Вот interface UserService, реализуй его»
– **Проверяйте imports** – Claude Code может импортировать несуществующие модули

26.6. Безопасность: чего не делать

– **Не вставляйте реальные секреты** – API-ключи, пароли, приватные ключи
– **Не вставляйте конфиденциальный код** – NDA, проприетарные алгоритмы (если ваша компания не разрешает)
– **Не доверяйте безопасности AI-кода без проверки** – всегда проводите аудит
– **Не автоматизируйте деструктивные операции** – DROP TABLE, rm -rf, force push

26.7. Когда Claude Code – не лучший выбор

Задача | Лучший инструмент
Визуальный дизайн (pixel-perfect) | Figma, дизайнер
Отладка сетевых проблем | Wireshark, tcpdump
Профилирование производительности | Chrome DevTools, perf
Сборка native-библиотек | Специализированные инструменты
Принятие бизнес-решений | Product Manager
Финальная проверка безопасности | Профессиональный пентестер

26.8. Этические аспекты

AI-сгенерированный код и авторство

– Вы отвечаете за любой код, который добавляете в проект
– AI-сгенерированный код – это инструмент, как автодополнение в IDE
– Лицензионные вопросы: код Claude Code не копирует из конкретных репозиториев

AI и рынок труда

Claude Code не заменяет разработчиков – он меняет их роль:

– Меньше рутинного кода -> больше архитектуры и дизайна
– Меньше копирования из StackOverflow -> больше понимания
– Меньше боилерплейта -> больше бизнес-логики
– Порог входа в разработку снижается -> но «потолок» растёт

Итоги главы

Ограничения Claude Code:

– Конечное контекстное окно -> маленькие сессии
– Галлюцинации -> проверяйте результат
– Дата среза знаний -> указывайте версии
– Большие проекты -> CLAUDE.md и точные инструкции
– Безопасность -> не вставляйте секреты
– Не панацея -> знайте, когда использовать другие инструменты

Понимание этих ограничений не уменьшает ценность Claude Code – оно делает вас более эффективным пользователем.

Глава 27. Будущее AI-разработки

27.1. Куда мы движемся

Мы стоим в начале фундаментального сдвига в разработке ПО. Claude Code в 2025 году – это как первый iPhone в 2007: уже впечатляет, но через несколько лет мы оглянемся назад и удивимся, как работали с «такими ограниченными инструментами».

Давайте заглянем в будущее – не в фантастическое, а в практическое, основанное на текущих тенденциях.

27.2. Эволюция AI-инструментов

От ассистента к агенту

Сегодня Claude Code – ассистент: вы ставите задачу, он выполняет, вы проверяете. Завтра AI станет агентом – он будет сам планировать, выполнять, тестировать и деплоить. Ваша роль сместится от «пиши код» к «определяй требования и веди ревью».

```
Сегодня:
User: "Добавь пагинацию в API"
AI: [пишет код] -> User: [проверяет] -> User: [деплоит]
Завтра:
User: "Добавь пагинацию в API"
AI: [анализирует проект] -> [пишет код] -> [пишет тесты] ->
[запускает тесты] -> [исправляет ошибки] -> [создаёт PR] ->
[описывает изменения] -> User: [ревью + approve]
```

Мультимодальность

Скоро AI-ассистенты будут:

– **Видеть** ваш экран и понимать UI/UX
– **Слышать** вашу речь (голосовое программирование)
– **Рисовать** – генерировать UI по макету на бумаге
– **Понимать** скриншоты ошибок без текстового описания

```
Будущее:
User: [рисует макет на бумаге, фото]
User: "Реализуй этот экран"
AI: [генерирует React-компоненты по фотографии]
```

Долгая память

Текущее ограничение контекстного окна постепенно исчезнет. AI будет помнить весь проект, все предыдущие диалоги, ваши предпочтения и стиль разработки.

27.3. Как изменится роль разработчика

Новые навыки

Навыки, которые будут цениться больше:

– **Архитектурное мышление** – понимание системы в целом
– **Требования и спецификации** – точно формулировать, что нужно
– **Code Review** – умение оценивать AI-сгенерированный код
– **Тестирование и верификация** – проверять корректность
– **Безопасность** – понимать риски и уязвимости
– **Бизнес-домен** – понимать предметную область

Навыки, которые не устареют

– Алгоритмическое мышление
– Понимание Computer Science (как работает memory, network, OS)
– Отладка (AI помогает, но debug mindset незаменим)
– Системный дизайн
– Коммуникация с людьми

Что изменится

– Написание boilerplate-кода -> автоматизация
– Поиск в StackOverflow -> диалог с AI
– Чтение документации -> AI объясняет
– Ручное тестирование -> AI генерирует тесты
– Code review стиля -> автоматизация (lint + AI)

27.4. AI в архитектуре будущих систем

AI-native приложения

Приложения всё чаще будут строиться с AI в ядре:

```
Традиционное приложение:
User -> UI -> API -> Business Logic -> Database
AI-native приложение:
User -> UI -> AI Orchestrator -> [
Traditional API,
AI Analysis,
AI Generation,
Database,
External APIs
]
```

AI-powered DevOps

```
Будущее DevOps:
- AI мониторит production 24/7
- Обнаруживает аномалии до того, как пользователи заметят
- Предлагает fix -> автодеплоит после approval
- Оптимизирует инфраструктуру (scale up/down)
- Пишет post-mortem после инцидента
```

27.5. Практические советы на будущее

1. Инвестируйте в фундамент

Чем глубже ваши знания CS, тем эффективнее вы используете AI. Алгоритмы, структуры данных, паттерны проектирования, сети, ОС – это фундамент, который AI усиливает, но не заменяет.

2. Развивайте «AI literacy»

Понимайте, КАК работают LLM: промпт-инжиниринг, контекстное окно, temperature, токенизация. Это поможет получать лучшие результаты.

3. Создавайте свои инструменты

MCP-серверы, пользовательские команды, хуки – создавайте инструменты, которые усиливают ваш workflow.

4. Следите за развитием

– Anthropic Blog – обновления Claude

– MCP Specification – развитие стандарта
– GitHub Claude Code – исходники и обсуждения

5. Экспериментируйте

AI-инструменты развиваются стремительно. То, что не работало месяц назад, может работать отлично сегодня. Регулярно пробуйте новые подходы.

27.6. Заключение

Claude Code – это не просто инструмент. Это сдвиг парадигмы в разработке ПО, сравнимый с переходом от ассемблера к высокоуровневым языкам, от текстовых редакторов к IDE, от монолитов к микросервисам.

Мы прошли долгий путь в этой книге:

– **Часть I** – познакомились с Claude Code и научились с ним общаться
– **Часть II** – освоили практические задачи: от написания кода до Git
– **Часть III** – погрузились в продвинутые возможности: MCP, автоматизация, DevOps, безопасность
– **Часть IV** – создали реальные проекты: SaaS, API, бот, CLI, расширение, миграция legacy
– **Часть V** – осмыслили лучшие практики, ограничения и будущее

Главный вывод: Claude Code – мультипликатор ваших навыков. Чем лучше вы как разработчик, тем мощнее этот инструмент в ваших руках. Инвестируйте в свои навыки, используйте AI как катализатор, и вы будете создавать невероятные вещи.

Удачи в разработке!

Приложения

Приложение А. Шпаргалка по командам Claude Code

Основные команды запуска

Команда | Описание
`claude` | Запуск в интерактивном режиме
`claude "запрос"` | Одноразовый запрос
`claude -p "запрос"` | Print mode (headless, без интеракции)
`claude -p "запрос" –output-format json` | JSON-вывод
`claude –resume` | Продолжить последнюю сессию
`claude –model claude-sonnet-4-20250514` | Указать модель
`cat file | claude -p "…"` | Pipe mode

Slash-команды в сессии

Команда | Описание
`/help` | Справка
`/compact` | Сжать контекст
`/clear` | Очистить контекст полностью
`/cost` | Показать стоимость сессии
`/model` | Сменить модель
`/permissions` | Управление разрешениями
`/project:command` | Проектная команда (.claude/commands/)
`/user:command` | Персональная команда (~/.claude/commands/)
`/vim` | Режим Vim-биндингов

Конфигурационные файлы

Файл | Область | Описание
`CLAUDE.md` | Корень проекта | Инструкции для проекта
`.claude/settings.json` | Проект | Настройки, MCP, hooks
`~/.claude/settings.json` | Глобально | Глобальные настройки
`~/.claude/commands/` | Глобально | Персональные команды
`.claude/commands/` | Проект | Проектные команды

Режимы разрешений

Режим | Описание

`–allowedTools` | Список разрешённых инструментов
`–dangerouslySkipPermissions` | Пропустить все подтверждения (CI)

Горячие клавиши

Клавиша | Действие
`Enter` | Отправить сообщение
`Shift+Enter` | Новая строка
`Ctrl+C` | Отмена текущей операции
`Ctrl+D` | Выход из сессии
`Tab` | Автодополнение файлов/команд
`^` | Предыдущее сообщение
`Escape` | Отмена текущего ввода

Форматы вывода

Формат | Описание
`–output-format text` | Текст (по умолчанию)
`–output-format json` | JSON с metadata
`–output-format stream-json` | Потоковый JSON

Переменные окружения

Переменная | Описание
`ANTHROPIC_API_KEY` | API-ключ Anthropic
`CLAUDE_CODE_MAX_TURNS` | Максимум шагов (headless)
`CLAUDE_CODE_USE_BEDROCK` | Использовать AWS Bedrock
`CLAUDE_CODE_USE_VERTEX` | Использовать Google Vertex

Приложение Б. Шаблоны CLAUDE.md

Б.1. Минимальный CLAUDE.md

```
## Проект
[Название] – [краткое описание]
## Стек
– [Язык, версия]
– [Фреймворк]
– [База данных]
## Команды
– Запуск: `npm run dev`
– Тесты: `npm test`
– Сборка: `npm run build`
## Стиль кода
– [Основные правила]
```

Б.2. Backend API (TypeScript)

```
## Проект
REST API для [описание]. Production-сервис.
## Стек
– TypeScript 5.4, Node.js 20 LTS
– Express 4 + express-async-errors
– PostgreSQL 16, Prisma ORM
– Redis 7 (кэш + Bull queues)
– Docker + Docker Compose
## Структура
src/
+– routes/ # Express роутеры
+– services/ # Бизнес-логика
+– repositories/ # Работа с БД
+– middleware/ # Express middleware
+– types/ # TypeScript типы
+– utils/ # Утилиты
+– config/ # Конфигурация
## Принципы
– Функциональный стиль: без классов, чистые функции
– Error handling: Result<T, E>, не через throw
– Validation: Zod schemas на входе каждого endpoint
– Logging: structured JSON через pino
– Все строки конфигурации – из env (process.env)
## Стиль кода
– camelCase для переменных, PascalCase для типов
```

```
- Named exports only (без default export)
- Enum через `as const` объекты
- Функция: макс 30 строк
- Файл: макс 200 строк
## Команды
- `pnpm dev` – запуск dev-сервера
- `pnpm test` – vitest (unit)
- `pnpm test:int` – тесты с DB (testcontainers)
- `pnpm lint` – ESLint
- `pnpm build` – сборка
- `pnpm db:migrate` – миграции Prisma
- `pnpm db:studio` – Prisma Studio
## Тестирование
- Unit: vitest, моки через vi.mock
- Integration: supertest + testcontainers (PG + Redis)
- Coverage: минимум 80%
- Naming: describe('[Module]', () => { it('should …') })
## Git
- Conventional Commits: feat|fix|refactor|docs|test|chore
- Branch: feature/*, bugfix/*, hotfix/*
- PR: всегда через PR в develop
```

Б.3. Frontend (React/Next.js)

```
## Проект
SPA для [описание]. Next.js 14 App Router.
## Стек
- TypeScript 5.4, Next.js 14
- Tailwind CSS + Radix UI
- Zustand (client state) + React Query (server state)
- tRPC для API-вызовов
## Структура
src/
+- app/ # Next.js страницы (App Router)
+- components/
| +- ui/ # Примитивы (Button, Input, Modal)
| +- features/ # Бизнес-компоненты
+- hooks/ # React хуки
+- lib/ # Утилиты
+- stores/ # Zustand stores
+- types/ # TypeScript типы
## Принципы
- Server Components по умолчанию
- 'use client' только когда нужна интерактивность
- Компонент: макс 100 строк
- Composition over inheritance
- Prop drilling до 2 уровней, дальше – context/store
## Стиль
```

```
- Tailwind: утилитарные классы, без @apply
- Цвета: через CSS-переменные (--color-primary)
- Responsive: mobile-first
- a11y: semantic HTML, aria-labels, keyboard nav
## Тестирование
- Vitest + React Testing Library
- Playwright для E2E
```

Б.4. Python Backend

```
## Проект
[Описание]. FastAPI + SQLAlchemy.
## Стек
- Python 3.12
- FastAPI 0.109
- SQLAlchemy 2.0 (async)
- PostgreSQL 16
- Redis (aioredis)
- Poetry для зависимостей
## Структура
app/
+- api/ # Routers
+- models/ # SQLAlchemy models
+- schemas/ # Pydantic schemas
+- services/ # Business logic
+- repositories/ # DB queries
+- core/ # Config, security, deps
+- utils/ # Helpers
## Принципы
- Type hints everywhere
- async/await для I/O
- Dependency injection через FastAPI Depends
- Pydantic v2 для валидации
- Google-style docstrings
## Команды
- `poetry run uvicorn app.main:app --reload`
- `poetry run pytest`
- `poetry run pytest --cov=app`
- `poetry run alembic upgrade head`
- `poetry run ruff check .`
- `poetry run mypy .`
```

Б.5. Monorepo

```
## Проект
Monorepo для [описание].
## Стек
```

```
- Turborepo
- pnpm workspaces
- TypeScript 5.4
## Структура
apps/
+- web/ # Next.js фронтенд
+- api/ # Express бэкенд
+- admin/ # Админ-панель
packages/
+- ui/ # Shared компоненты
+- types/ # Shared типы
+- config/ # Shared ESLint, TSConfig
+- utils/ # Shared утилиты
## Правила
- Shared код - в packages/
- Не импортируй из apps/ в другие apps/
- Новый shared код - обсуди, нужен ли он в packages/
## Команды
- `pnpm dev` - запуск всех apps
- `pnpm build` - сборка всего
- `pnpm test` - тесты всего
- `pnpm -filter web dev` - только web
- `pnpm -filter api test` - тесты только api
```

Приложение В. Словарь терминов

А

AI-ассистент – программа на базе искусственного интеллекта, которая помогает разработчику писать код, отвечать на вопросы и автоматизировать задачи.

API (Application Programming Interface) – набор правил и протоколов для взаимодействия между программными компонентами.

Agentic mode – режим работы AI, в котором он самостоятельно планирует и выполняет последовательность действий для достижения цели.

Б

Boilerplate – шаблонный, повторяющийся код, который необходим для работы программы, но не содержит бизнес-логики.

В

Вложенный контекст (Nested Context) – дополнительные файлы контекста (CLAUDE.md), расположенные в подкаталогах проекта.

Г

Галлюцинация – феномен, при котором AI-модель генерирует уверенно звучащую, но фактически неверную информацию.

К

Контекстное окно (Context Window) – максимальный объём текста (в токенах), который модель может обработать за один раз.

Conventional Commits – спецификация формата сообщений коммитов: `type(scope): description`.

Л

LLM (Large Language Model) – большая языковая модель, нейронная сеть, обученная на больших объёмах текста.

Legacy-код – устаревший код, который всё ещё используется, но написан с использованием старых технологий и практик.

М

MCP (Model Context Protocol) – открытый стандарт для подключения AI-моделей к внешним источникам данных и инструментам.

MCP-сервер – процесс, предоставляющий инструменты, ресурсы или промпты через протокол MCP.

П

Промпт (Prompt) – текстовая инструкция, которую пользователь отправляет AI-модели.

Промпт-инжиниринг (Prompt Engineering) – искусство формулирования запросов к AI для получения оптимальных результатов.

Р

Рефакторинг – изменение внутренней структуры кода без изменения его поведения.

С

Скаффолдинг (Scaffolding) – автоматическая генерация начальной структуры проекта или компонента.

Slash-команда – команда, начинающаяся с /, выполняемая в интерактивной сессии Claude Code.

Т

Токен – единица текста для языковой модели. Один токен 3-4 символа кириллицы или ~4 символа латиницы.

Temperature – параметр, определяющий «креативность» ответов модели. Низкая (0) – детерминированные, высокая (1) – более разнообразные.

Х

Headless-режим – режим работы Claude Code без интерактивного терминала, для автоматизации.

Hooks – механизм реактивной автоматизации в Claude Code, привязывающий действия к событиям.

Ш

Шпаргалка CLAUDE.md – см. Приложение Б.

JSON-RPC 2.0

Протокол удалённого вызова процедур, используемый MCP для коммуникации между клиентом и сервером.

SSE (Server-Sent Events)

Технология односторонней передачи данных от сервера к клиенту через HTTP, используемая в MCP для удалённых серверов.