Правила работы с Cursor и AI-инструментами в Ruby/Rails проектах с примерами промптов

DeepSeek Version

by Thinknetika and Eugene Leontev

Ответ deepseek

Мы — команда из 10 опытных Ruby-разработчиков, ежедневно использующих Cursor (AI-инструмент) в работе.
Наша задача: составить топ-30 самых важных и эффективных правил (Cursor Rules) для проектов на Ruby и Ruby on Rails.

Критерии:

- Правила должны быть краткими, в формате markdown.
- Каждое правило должно быть обосновано (почему попало в топ).
  Процесс:

1. Сначала проведем мозговой штурм по категориям (например: качество кода, безопасность, производительность, работа с БД, тестирование, использование AI и т.д.)
2. Соберем больше 30 идей, затем отфильтруем до топ-30.
3. Каждое правило сформулируем четко и кратко, а затем дадим аргументацию.

Топ-30 правил использования Cursor для Ruby/Rails проектов

🔒 Безопасность

1. **Фильтруй генерируемые гемы через bundle-audit**

   ```
   // @gemfile: Проверь уязвимости в геме 'pdf-generator'
   ```

   ❌ Плохо: "Добавь гем для генерации PDF" без проверки

   Предотвращает использование уязвимых зависимостей

2. **Никогда не доверяй SQL-инъекциям AI**

   ```
   // Сгенерируй безопасный запрос: User.where("email = ?", params[:email])
   ```

   ❌ Плохо: "Найди пользователя по email: #{params[:email]}"

   Исключает риски инъекций через параметризованные запросы

3. **Шифруй секреты с помощью Rails Credentials**

   ```
   // Обнови секрет API в зашифрованных credentials
   ```

   ❌ Плохо: "Добавь API_KEY='123' в .env файл"

   Защищает чувствительные данные от попадания в репозиторий

4. **Генерируй безопасные параметры через .permit()**

   ```
   // Создай strong parameters для User: params.permit(:name, :email)
   ```

   ❌ Плохо: "Прими все параметры для модели"

   Предотвращает массовое присваивание уязвимых атрибутов

🧩 Качество кода

5. **Всегда ревьюируй сгенерированный код**

   ```
   // @app/services/payment.rb: Проверь логику списания средств
   ```

   ❌ Плохо: Слепое копирование 50+ строк без анализа

   Исключает логические ошибки и архитектурные противоречия

6. **DRY-принцип важнее "умного" кода**

   ```
   // Рефакторинг: выдели общий метод для calculate_tax в модуль
   ```

   ❌ Плохо: "Сгенерируй отдельную реализацию для каждого контроллера"

   Уменьшает дублирование и упрощает поддержку

7. **Избегай метапрограммирования**

   `// Создай явные методы вместо define_method`

   ❌ Плохо: "Динамически сгенерируй методы на лету"

   Упрощает отладку и читаемость кода

8. **Дедупликация кода через модули**

   `// Вынеси аутентификацию в Concern @app/controllers/concerns/`

   ❌ Плохо: "Скопируй этот код в 10 контроллеров"

   Централизует общую логику для повторного использования

⚡ Производительность

9. **Проверяй N+1 в сгенерированных запросах**

   ```
   // Оптимизируй: User.includes(:posts).where(active: true)
   ```

   ❌ Плохо: "Выведи всех пользователей с их постами" без .includes

   Исключает проблемы с производительностью на больших данных

10. **Оптимизируй запросы через explain-analyze**

    ```
    // Покажи EXPLAIN ANALYZE для этого SQL: #{query}
    ```

    ❌ Плохо: "Сгенерируй сложный JOIN без анализа"

    Выявляет узкие места в работе с базой данных

11. **Кешируй фрагменты, а не целые страницы**

```
// Добавь кеширование для блока с продуктами: cache @products
```

❌ Плохо: "Закешируй всю страницу каталога"

Оптимизирует использование памяти и инвалидацию кеша

12. **Генерируй background-задачи для Sidekiq**

```
// Создай воркер для отправки email: MailerWorker.perform_async
```

❌ Плохо: "Отправляй письмо прямо в контроллере"

Разгружает основной поток выполнения

🧪 Тестирование

13. **Тестируй миграции БД в режиме dry-run**

    ```
    // Проверь миграцию: rails db:migrate --dry-run
    ```

    ❌ Плохо: "Запусти миграцию сразу на прод"

    Предотвращает конфликты схемы в рабочем окружении

14. **Генерируй RSpec-тесты, а не Minitest**

    ```
    // Напиши request-тест для API users#create
    ```

    ❌ Плохо: "Создай тесты в формате Minitest"

    RSpec лучше подходит для сложных сценариев через DSL

15. **Тестируй edge-cases вручную**

```
// Добавь тест для email с кириллицей: "пользователь@почта.рф"
```

❌ Плохо: "Протестируй только happy path"

Выявляет пограничные случаи, которые пропускает AI

16. **Валидации моделей — только ручной ввод**

```
// Добавь валидацию: validates :phone, format: {with: /regex/}
```

❌ Плохо: "Сгенерируй валидации для модели"

Гарантирует корректность бизнес-правил

📚 Документация и инструменты

17. **Автоматизируй RuboCop-исправления**

    `// Исправь нарушения RuboCop в @app/models/user.rb`

    ❌ Плохо: "Игнорируй стилевые ошибки, главное чтобы работало"

    Поддерживает единый стиль кода автоматически

18. **Генерируй документацию в Yard-формате**

    `// Напиши Yard-доку для метода #calculate_tax`

    ❌ Плохо: "Оставь метод без комментариев"

    Стандартизирует документацию для командной работы

19. **Локализацию генерируй только в YAML-файлы**

    `// Добавь перевод для кнопки в config/locales/en.yml`

    ❌ Плохо: "Вставь текст прямо в шаблон: 'Submit'"

    Отделяет контент от логики для легкой интернационализации

20. **Генерируй .env.example вместо .env**

    `// Создай шаблон конфига с примерными значениями`

    ❌ Плохо: "Добавь реальные секреты в .env"

    Предотвращает случайную публикацию чувствительных данных

🚀 Архитектура и лучшие практики

21. **Генерируй код небольшими частями (1-5 строк)**

    ```
    // Реализуй фильтрацию по статусу в методе search
    ```

    ❌ Плохо: "Напиши весь класс сервиса за один запрос"

    Увеличивает точность и предсказуемость результата

22. **Явно указывай контекст через @-ссылки**

    ```
    // @app/controllers/users_controller.rb:15 — Обнови метод update
    ```

    ❌ Плохо: "Добавь проверку прав" без указания места

    Дает AI точный контекст для генерации релевантного кода

23. **Генерируй GraphQL-схемы, а не REST-эндпоинты**

    `// Создай GraphQL тип для модели Product`

    ❌ Плохо: "Добавь 5 REST-методов для товаров"

    Современный подход с типизацией и гибкими запросами

24. **Строгая типизация для Sorbet/RBS**

    `// Сгенерируй сигнатуры Sorbet для User#authenticate`

    ❌ Плохо: "Оставь методы без аннотаций типов"

    Снижает количество runtime-ошибок

25. **Паттерн Policy Object для авторизации**

    `// Создай класс UserPolicy с методом can_edit?`

    ❌ Плохо: "Вставь проверки прав прямо в контроллер"

    Изолирует бизнес-правила для тестирования

🛠️ Инфраструктура

26. **Контейнеризация через Dockerfile.prod**

    `// Обнови Dockerfile для production-сборки`

    ❌ Плохо: "Используй образ с полным набором dev-инструментов"

    Оптимизирует размер и безопасность образов

27. **Оптимизация ассетов через importmap**

    `// Мигрируй JS с Webpacker на importmap`

    ❌ Плохо: "Добавь jQuery через отдельный script-тег"

    Современный стандарт управления зависимостями

28. **Генерация ERB вместо Haml/Slim**

    `// Создай шаблон index.html.erb для продуктов`

    ❌ Плохо: "Сгенерируй вьюху на Slim без конвертации"

    Стандартный синтаксис меньше подвержен ошибкам AI

29. **Мониторинг: только Datadog/Sentry-интеграции**

    `// Настрой кастомную метрику для Datadog`

    ❌ Плохо: "Реализуй систему логов с нуля"

    Использует проверенные инструменты вместо самописных

30. **Обработка ошибок через rescue_from**

    `// Добавь глобальную обработку 404 ошибок в ApplicationController`

    ❌ Плохо: "Отлавливай исключения только в отдельных методах"

    Централизованный подход для обработки исключений

Ключевые принципы

- **Контекст — король**: Всегда используй @-ссылки для точного позиционирования (`@app/models/user.rb:15`)
- **Инкрементальность**: Генерация по 1-5 строк > монолитные блоки
- **Безопасность по умолчанию**: Параметризованные запросы, bundle-audit, credentials
- **Человеческий надзор**: AI генерирует — разработчик валидирует
- **Rails Way**: Предпочтение конвенциям фреймворка вместо креативных решений AI

> Пример идеального промпта:
>
> `// @app/services/payment_service.rb:32 — Оптимизируй метод process! с обработкой ошибок и проверкой N+1 в связанных моделях`
>
> Такой запрос дает точное местоположение, конкретную задачу и учитывает ключевые аспекты (ошибки, производительность).


THINKNETICA
онлайн-школа для разработчиков