Пособие по промпт-инжинирингу для программистов

Разработчики всё чаще полагаются на ИИ-помощников, чтобы ускорить повседневную работу с кодом. Эти инструменты умеют автозаполнять функции, предлагать исправления ошибок и даже генерировать целые модули или MVP. Тем не менее, как многие из нас убедились, качество вывода ИИ во многом зависит от качества предоставленного запроса. Плохо сформулированный промпт может привести к нерелевантным или общим ответам, в то время как хорошо составленный — дать продуманные, точные и даже креативные решения для кода.

Под катом Эдди Османи, ведущий инженер Google, выделяет **ключевые шаблоны запросов, повторяемые фреймворки и запоминающиеся примеры, которые нашли отклик у разработчиков**.

Автор приводит **параллельные сравнения** хороших и плохих промптов, фактические ответы ИИ, а также комментарии: чтобы понять, почему один запрос успешен, а другой терпит неудачу.

Стартовая шпаргалка:

Техника	Шаблон запроса	Цель
Ролевой промптинг (Role Prompting)	“Ты — senior {язык} разработчик. Сделай ревью этой функции для {цель}”	Симулировать экспертное код-ревью, дебаггинг или рефакторинг
Явная настройка контекста (Explicit Context Setup)	“Здесь есть проблема: {описание}. Код — ниже. Он должен выполнять {ожидаемое поведение}, но вместо этого выполняет {текущее поведение}. Почему?”	Чётко сформулировать проблему, чтобы избежать общих, поверхностных ответов
Примеры ввода/вывода (Input/Output Examples)	“Эта функция должна возвращать {ожидаемый вывод}, когда дано {ввод}. Можешь ли ты написать или поправить код? ”	Направить помощника, показывая намерение на примерах
Итеративное связывание (Iterative Chaining)	“Сперва сгенерируем скелет компонента. Далее, добавим состояние. Затем обработаем вызовы API”	Разбить крупные задачи на этапы, чтобы избежать непонятных или расплывчатых промптов
Отладка с помощью моделирования (Debug with Simulation)	“Пройдись по функции строка за строкой. Каковы значения переменных? Где она может сломаться?”	Заставить помощника имитировать поведение во время выполнения и выявлять скрытые ошибки
Планирование фичи (Feature Blueprinting)	“Я разрабатываю {фича}. Требования: {список}. С использованием: {стек}. Пожалуйста, создай каркас начального компонента и объясни свой выбор”	Начать разработку фичи на основе ИИ-планирования
Руководство по рефакторингу кода (Code Refactor Guidance)	“Сделай рефакторинг этого кода для улучшения {цель}, в том числе {читаемость, производительность, идиоматический стиль}. Используй комментарии, чтобы объяснить изменения”	Заставить ИИ-рефакторинг соответствовать вашим целям, а не произвольным изменениям
Запрос альтернативы (Ask for Alternatives)	“Можешь переписать это, используя функциональный стиль? Как будет выглядеть рекурсивная версия?”	Изучить несколько путей внедрения и расширить свой инструментарий
Метод утёнка (Rubber Ducking)	“Вот что, по-моему, делает эта функция {ваше объяснение}. Я что-то упускаю? Выявляет ли она какие-либо ошибки?”	Позволить ИИ проверить ваше понимание и обнаружить несоответствия
Привязка ограничений (Constraint Anchoring)	Пожалуйста, избегай {например, рекурсии} и придерживайся {например, синтаксиса ES6, отсутствия внешних библиотек}. Оптимизируй {например, память}. Вот функция.	Не допустить выхода ИИ за рамки своих возможностей или введения несовместимых шаблонов

Подсказка ИИ-инструменту для программирования чем-то похожа на общение с очень буквальным, *иногда* знающим сотрудником. Чтобы получить полезные результаты, вам необходимо подготовиться: направить ИИ на то, что и как именно вы хотите сделать.

Вот основополагающие принципы для дальнейших примеров в этой статье:

- **Обеспечьте богатый контекст**. Всегда предполагайте: ИИ знает о вашем проекте не больше, чем вы ему сообщаете. Включите в контекст соответствующие детали, такие как язык программирования, фреймворк и библиотеки, а также конкретную функцию или фрагмент, о котором идёт речь. Если произошла ошибка, предоставьте точное сообщение об ошибке и опишите, что должен делать код. **Конкретика** и **контекст** определяют разницу между расплывчатыми предложениями и точными, действенными решениями. На практике это означает, что запрос может включать краткую настройку, например: «У меня есть функция Node.js с использованием Express и Mongoose, которая должна получать данные пользователя по ID, но она выдаёт ошибку TypeError. Вот код и ошибка...». Чем больше настроек вы дадите, тем меньше ИИ придется угадывать. (*Прим. инженеров Сравни: но стоит учитывать, что у LLM ограниченное окно контекста, обычно 8–128k токенов*)
- **Четко сформулируйте свою цель или вопрос**. Расплывчатые вопросы приводят к расплывчатым ответам. Вместо того, чтобы спрашивать что-то вроде «Почему мой код не работает?», точно определите, какая информация вам нужна. Например: «Эта функция JavaScript возвращает undefined вместо ожидаемого результата. Учитывая приведенный ниже код, можешь ли ты помочь определить, почему, и как это исправить?» с гораздо большей вероятностью даст полезный ответ. Одна из формул запроса для отладки: «*Ожидается, что код выполнит [ожидаемое поведение], но вместо этого он выполняет [текущее поведение], когда ему задан [пример входных данных]. Где баг?*». Точно так же, если вам нужна оптимизация, запросите её *определённый вид* (например: «*Как я могу улучшить производительность этой функции сортировки для 10k элементов?*»). Специфичность направляет фокус внимания ИИ.
- **Разбивайте сложные задачи**. При внедрении новой фичи или решении многоэтапной проблемы не стоит описывать всё в одном огромном запросе. Часто эффективнее разделить работу на более мелкие части и выполнять итерации. Например: «*Сперва сгенерируй скелет компонента React для страницы со списком товаров. Далее мы добавим управление состоянием. Затем мы интегрируем вызов API*». Каждый промпт основывается на предыдущем. Часто не рекомендуется запрашивать целую большую функцию за один раз; вместо этого начните с высокоуровневой цели, а затем итеративно запрашивайте по каждой составляющей. Такой подход не только делает реакции ИИ сфокусированными и управляемыми, но и отражает то, как человек будет постепенно создавать решение.
- **Включите примеры вводов/выводов или ожидаемого поведения**. Если вы можете проиллюстрировать то, что хотите, на примере, сделайте это. Например: *«Учитывая массив [3,1,4], эта функция должна возвращать [1,3,4]*». Предоставление

конкретного примера в промпте помогает ИИ понять ваше намерение и уменьшает двусмысленность. Это похоже на то, как если бы вы дали младшему разработчику быстрый тест-кейс — ИИ уточняет требования. В терминах промпт-инжиниринга это иногда называется «**Few-Shot-промптинг**», когда вы показываете ИИ шаблон, которому нужно следовать. Даже один пример правильного поведения может в значительной степени повлиять на реакцию модели.

- **Используйте роли или персоны**. Мощный метод, популяризированный во многих вирусных примерах промптов, заключается в том, чтобы попросить ИИ «действовать как» определённый персонаж или роль. Это может повлиять на стиль и глубину ответа. Например, «*Выступи в качестве старшего разработчика React и проверь мой код на предмет потенциальных ошибок*» или «*Ты — эксперт по производительности JavaScript. Оптимизируй следующую функцию*». Назначив роль, вы заставляете помощника использовать соответствующий тон — будь то строгий рецензент кода, полезный ментор для младшего разработчика или аналитик по безопасности, ищущий уязвимости. Промпты с использованием этого метода, которыми сообщество поделилось в интернете, оказались успешны, например: «*Действуй как обработчик ошибок JavaScript и отладь для меня эту функцию. Данные неправильно отображаются из вызова API*». Да, нам по-прежнему необходимо предоставить код и подробную информацию о проблеме, но ролевой промптинг может дать более структурированные и экспертные рекомендации.

- **Повторяйте и уточняйте**. Промпт-инжиниринг — это интерактивный процесс, а не разовая «сделка». Разработчикам часто приходится просматривать первый ответ ИИ, а затем задавать последующие вопросы или вносить исправления. Если решение не совсем правильное, вы можете сказать: «Это решение использует рекурсию, но я бы предпочел итеративный подход — можешь ли ты попробовать

ещё раз без рекурсии?» Или: «Отлично, теперь ты можешь улучшить имена переменных и добавить комментарии?» ИИ запоминает контекст в сеансе чата, и вы можете постепенно направлять его к желаемому результату. Ключ в том, чтобы рассматривать ИИ как партнера, которого вы можете тренировать – **прогресс важнее совершенства** с первой попытки.
- **Поддерживайте ясность и согласованность кода**. Этот принцип немного косвенный, но очень важный для инструментов, которые работают с контекстом вашего кода. Пишите чистый, хорошо структурированный код и комментарии ещё до того, как ИИ вступит в игру. Осмысленные имена функций и переменных, единообразное форматирование и строки документации не только делают ваш код более понятным для людей, но и дают ИИ более точные подсказки о том, что вы делаете. Если вы зададите последовательный шаблон или стиль, ИИ продолжит его. Относитесь к ИИ-инструментам как к крайне внимательным младшим разработчикам — они улавливают каждую подсказку из вашего кода и комментариев.

Помня об этих основополагающих принципах, углубимся в конкретные сценарии. Начнём с **отладки**, возможно, самого непосредственного варианта использования: у вас есть код, который ведёт себя некорректно, и вы хотите, чтобы ИИ помог выяснить причину.

(*Прим. инженеров Сравни: в тексте автора отсутствуют предупреждения о некоторых недостатках LLM, таких как галлюцинации, ложные объяснения, искажение логики при генерации кода. Советуем учитывать их вероятность при работе с ИИ-инструментами*!)

Шаблоны промптов для отладки кода

Отладка — естественная задача для ИИ-помощника. Это как резиновый утёнок, который не только вас слушает, но и отвечает, предлагая решения. Однако успех во многом зависит от того, как вы представите проблему ИИ.

Вот как систематически запрашивать помощь в поиске и исправлении ошибок:

1. **Чётко опишите проблему и её симптомы**. Начните запрос с описания того, что идёт не так и что должен делать код. Всегда включайте точное сообщение об ошибке или некорректном поведении. Например, вместо того, чтобы просто написать «Мой код не работает», вы можете сообщить: «*У меня есть функция в JavaScript, которая должна вычислять сумму массива чисел, но она возвращает NaN (не число) вместо фактической суммы. Вот код: [включить код]. Она должна вывести число (сумму) для массива чисел, например [1,2,3], но я получаю NaN. Что может быть причиной этой ошибки?*» В этом промпте указывается язык, предполагаемое поведение, наблюдаемый неправильный вывод и предоставляется контекст кода — вся важная информация. Предоставление структурированного контекста (код + ошибка + ожидаемый результат + то, что вы пробовали) даёт ИИ надёжную отправную точку. Напротив, общий вопрос типа «Почему моя функция не работает?» даёт скудные результаты — модель может предложить только самые общие предположения без контекста.

2. **Используйте пошаговый или построчный подход для сложных ошибок**. В случае более сложных логических ошибок (где нет явного сообщения об ошибке, но вывод неверен) вы можете попросить ИИ пройтись по выполнению кода. Например: «*Пройдись по этой функции построчно и отслеживай значение total на каждом шаге. Сумма накапливается неправильно — где сбой логики?*» Это пример отладочной подсказки «резинового утёнка» — вы, по сути, просите ИИ смоделировать процесс отладки, который мог бы выполнить человек с помощью печати или отладчика. Такие подсказки часто выявляют тонкие проблемы вроде несброса переменных или некорректной условной логики, поскольку ИИ будет описывать состояние на каждом шаге. Если у вас есть подозрения относительно определённой части кода, вы можете уточнить: «*Объясни, что здесь делает вызов фильтра, и не исключает ли он больше элементов, чем следует*». Привлечение ИИ к роли объяснителя может помочь выявить ошибку в процессе объяснения.

3. **По возможности предоставляйте минимально воспроизводимые примеры**. Иногда ваша фактическая кодовая база велика, но ошибку можно продемонстрировать в небольшом фрагменте. Если вы можете извлечь или упростить код, который всё ещё воспроизводит проблему, сделайте это и передайте его ИИ. Это не только поможет ИИ сосредоточиться, но и заставит вас прояснить проблему (часто полезное упражнение само по себе). Например, если вы получаете TypeError в глубоко вложенном вызове функции, попробуйте воспроизвести её с помощью нескольких строк, которыми можете поделиться. Стремитесь изолировать ошибку с помощью минимального кода, сделайте предположение о том, что не так, протестируйте его и выполните итерацию. Вы можете привлечь ИИ к этому, сказав: «*Вот урезанный пример, который всё ещё вызывает ошибку [включаемый фрагмент]. Почему возникает эта ошибка?*» Упрощая, вы устраняете шум и помогаете ИИ точно определить проблему. (Этот метод отражает совет многих ведущих инженеров: если не можете сразу найти ошибку, упростите проблемное пространство. ИИ способен помочь в этом анализе, если вы дадите ему более ёмкий пример.)

4. **Задавайте конкретные и уточняющие вопросы**. После предоставления контекста часто бывает эффективно напрямую спросить, что вам нужно, например: «Что может быть причиной этой проблемы и как я могу её исправить?». Это побуждает ИИ диагностировать проблему и предложить решение. Если первый ответ ИИ неясен или не совсем полезен, не стесняйтесь задавать уточняющие вопросы. Можно сказать: «*Это объяснение имеет смысл. Можешь ли показать мне, как исправить код? Пожалуйста, предоставь исправленный код*». В настройках чата ИИ хранит историю бесед, поэтому может напрямую выводить изменённый код. Если вы используете встроенный инструмент, такой как Copilot в VS Code или Cursor, без чата, можете вместо этого написать комментарий над кодом, например: // BUG: returns NaN, исправить эту функцию и посмотреть, как она автодополнится, — но, как правило, интерактивный чат даёт более подробные объяснения. Другой шаблон для дальнейшего взаимодействия: если ИИ предлагает решение, но вы не понимаете, почему именно его, спросите: «Можешь ли объяснить, почему это изменение решает проблему?» Таким образом, вы извлечете урок на будущее и ещё раз проверите обоснованность рассуждений ИИ.

Теперь проиллюстрируем эти принципы отладочных промптов на конкретном примере: **плохой промпт vs усовершенствованный промпт** и разница в ответах ИИ.

Пример отладки: плохой и улучшенный промпт

Представьте, что у нас есть простая функция Node.js, предназначенная для преобразования списка объектов пользователей в карту поиска по ID пользователя. Однако она выдаёт ошибку. Вот код с ошибкой:

```
// Buggy function: converts array of users to a map by ID
function mapUsersById(users) {
  const userMap = {};
  for (let i = 0; i <= users.length; i++) {
    // Note: using <= will go out of bounds on the last iteration
    const user = users[i];
    userMap[user.id] = user;
  }
  return userMap;
}

// Example usage:
const result = mapUsersById([{ id: 1, name: "Alice" }]);
```

Этот код должен перебирать массив `users` и создавать объект, ключами которого являются идентификаторы пользователей. Однако есть ошибка: в цикле используется `i <= users.length` вместо `i < users.length`. Это приведёт к тому, что на последней итерации значение `users[i]` станет неопределённым, что, вероятно, приведёт к ошибке выполнения (например, попытке доступа к свойству `id`, которое не определено). Давайте посмотрим, как ИИ-помощник может отреагировать на два разных запроса, связанных с этой проблемой.

 Плохой промпт: «Почему моя функция `mapUsersById` не работает?»

Реакция ИИ на неудачный промпт (симуляция):

> *Сложно сказать без дополнительной информации. Вы уверены, что функция вызывается с правильным массивом пользователей? Возможно, проблема в том, что `users` пуст или не является массивом, из-за чего цикл не выполняется должным образом. Также может быть ошибка при установке свойства в `userMap`. Убедитесь, что у каждого пользователя есть свойство `id`. Если `users` не определено или равно `null`, это также может привести к сбою.*

Анализ: Этот ответ довольно расплывчат. В промпте не было никаких подробностей, кроме имени функции, поэтому ИИ догадывается об общих причинах («возможно, массив пуст или не является массивом», «убедитесь, что у каждого пользователя есть идентификатор»). Он

не идентифицировал фактическую ошибку. Это тот тип ответа, который вы получаете, когда вопрос недостаточно определён — модель может только выдавать общие советы по отладке. Разработчику это не очень полезно, потому что не устраняет реальную ошибку. **Отсутствие контекста** и конкретики в запросе приводит к общему ответу (по сути, эквивалентному для ИИ «вы пробовали выключить и снова включить его?»). Это согласуется с мнением сообщества о том, что расплывчатые промпты приводят к нерелевантным или общим ответам.

 Улучшенный промпт: «У меня есть функция JavaScript `mapUsersById`, которая должна преобразовывать массив объектов пользователей в карту (объект), ключом которой является идентификатор пользователя. Однако при запуске возникает ошибка. Например, когда я передаю `[ {id: 1, name: "Alice"} ]`, я получаю TypeError: Невозможно прочитать свойство `'id'` из undefined. Вот код функции:

```
function mapUsersById(users) {
  const userMap = {};
  for (let i = 0; i <= users.length; i++) {
    const user = users[i];
    userMap[user.id] = user;
  }
  return userMap;
}
```

Должен вернуться `{ "1": {id: 1, name: "Alice"} }`. В чём ошибка и как её исправить?

Реакция ИИ на улучшенный промпт (симуляция):

> **Ошибка «Невозможно прочитать свойство 'id' из undefined» предполагает, что в какой-то момент значение `user` становится неопределённым, и затем вы пытаетесь получить доступ к `user.id`. Если посмотреть на ваш цикл, проблема связана с границами цикла. Вы используете `i <= users.length`. Если `users.length` равно `1` (для одного пользователя в массиве), цикл выполняется с `i = 0` и `i = 1`. Когда `i` равно `1`, значение `users[1]` не определено (поскольку единственный индекс в массиве равен 0). Таким образом, значение `user` становится неопределённым на этой итерации, а `user.id` выдает исключение TypeError. Чтобы исправить это, нужно изменить условие цикла на `i < users.length`, чтобы значение `i` изменялось от `0` до `users.length-1`. Например:*

```
for (let i = 0; i < users.length; i++) {
  const user = users[i];
  userMap[user.id] = user;
```

```
}
```

> *Таким образом, для первого (и единственного) элемента `i` будет равно `0`, после чего цикл прекратится. Благодаря этому изменению ваша функция больше не будет обращаться к неопределённому пользователю и должна работать как и ожидалось, возвращая `{"1": {id: 1, name: "Alice"}}` для заданных входных данных.**

Анализ: Этот ответ напрямую указывает на ошибку (используя оператор <= в цикле) и объясняет, почему она приводит к появлению значения user в состоянии undefined. Затем он предоставляет корректное решение с помощью кода (`i < users.length`) и подтверждает ожидаемый результат. Улучшенный промпт оказался успешным, поскольку соответствовал нашим принципам: в нём был указан язык (JavaScript), описано назначение функции, приведено точное сообщение об ошибке и пример входных данных, а также предоставлен фрагмент кода, о котором идёт речь. Обратите внимание, как ИИ использовал сообщение об ошибке в качестве подсказки для определения границ цикла — целевая подсказка позволила ИИ заняться настоящим решением проблемы, эффективно имитируя ход мыслей отладчика-человека: «Откуда могло взяться undefined? Вероятно, из-за индексации цикла». Это наглядная демонстрация преимуществ подробных промптов.

Дополнительные тактики отладки. Помимо выявления очевидных ошибок, вы можете использовать промпт-инжиниринг для более глубокой отладки:

- **Спросите о возможных причинах**. Если вы действительно в тупике, можно немного расширить вопрос: «*Каковы возможные причины ошибки TypeError: невозможно прочитать свойство 'foo' со значением undefined в этом коде?*», приложив код. Модель может перечислить несколько сценариев (например, объект не был инициализирован, состояние гонки, неправильная область действия переменной и т. д.). Это может дать вам новые аспекты для исследования, которые вы не рассматривали. Похоже на мозговой штурм с коллегой.
- **Спросите резинового утёнка** — то есть объясните свой код ИИ. Это может показаться нелогичным (зачем объяснять помощнику?), но сам процесс написания объяснения может прояснить ваше собственное понимание, и затем вы можете попросить ИИ проверить или оценить его. Например: «*Я объясню, что делает эта функция: [ваше объяснение]. Учитывая это, верны ли мои рассуждения и выявляют ли они ошибку?*» ИИ может обнаружить изъян в вашем объяснении, указывающий на реальную ошибку.
- **Попросите ИИ создать тестовые случаи**. Вы можете спросить: «*Можешь ли ты предоставить пару тестовых случаев (входных данных), которые могут привести к поломке этой функции?*» Помощник может предложить пограничные случаи, о которых вы не подумали (пустой массив, очень большие числа, значения NULL и т. д.). Это полезно как для отладки, так и для создания тестов для будущей проверки надёжности.

- **Задайте роль код-ревьюера**. В качестве альтернативы прямому запросу «отладь это» вы можете сказать: «*Выступи в роли код-ревьюера. Вот фрагмент, который работает не так, как ожидалось. Просмотри его и укажи на любые ошибки или плохие практики, которые могут вызывать проблемы: [код]*». Это переводит ИИ в критический режим. Многие разработчики считают, что формулировка запроса для код-ревью позволяет получить очень тщательный анализ, поскольку модель будет комментировать каждую часть кода (и часто, делая это, она обнаруживает ошибку). Фактически, один из советов по разработке промптов — явно попросить ИИ вести себя как дотошный проверяющий. Это может выявить не только имеющуюся ошибку, но и другие проблемы (например, потенциальное отсутствие проверок на null).

Подводя итог, можно сказать, что при отладке с помощью ИИ-помощника **детали и указания** — ваши главные друзья . Опишите ситуацию, симптомы, а затем задавайте конкретные вопросы. Разница между хаотичными промптами в духе «Не работает, помогите!» и точным указанием на отладку мы увидели выше. Далее перейдём к другому важному варианту использования ИИ: рефакторингу и улучшению существующего кода.

Промпты для рефакторинга и оптимизации

Рефакторинг кода — повышение его чистоты, скорости и выразительности без изменения функциональности — это область, где ИИ-помощники могут проявить себя во всей красе. Они прошли обучение на обширных объёмах кода, включая множество примеров хорошо структурированных и оптимизированных решений. Однако для эффективного использования этих знаний **в вашем промпте должно быть чётко указано, что означает «лучше» в вашей ситуации** . Вот как промптить задачи по рефакторингу:

1. **Чётко сформулируйте цели рефакторинга**. Само по себе «Рефакторинг этого кода» — слишком обще. Хотите ли вы улучшить читаемость? Снизить сложность? Оптимизировать производительность? Использовать другую парадигму или библиотеку? ИИ нужна цель. Хороший запрос сформулирует задачу, например: «*Сделай рефакторинг следующей функции для улучшения ее читаемости и удобства поддержки (уменьши повторения, используй более понятные имена переменных)*». Или «*Оптимизируй этот алгоритм для скорости — он слишком медленный на больших входных данных*». Указывая **конкретные цели** , вы помогаете модели решить, какие преобразования применять. Например, если вы скажете ей, что вас заботит производительность, она может использовать более эффективный алгоритм сортировки или кэширования, в то время как сосредоточение на читаемости может привести к тому, что она разобьёт функцию на более мелкие или добавит комментарии. Если у вас несколько целей, перечислите их. Шаблон промпта из руководства Strapi предлагает даже перечислить проблемы: «*Проблемы, которые я хотел бы решить: 1) [проблема с производительностью], 2) [дублирование кода], 3) [устаревшее использование API]*». Таким образом, ИИ точно знает, что нужно исправить. Помните, он не будет знать, *что*

именно вы считаете проблемой в коде — вы должны ему об этом сообщить.

2. **Предоставьте необходимый контекст кода**. При рефакторинге вы обычно включаете в промпт фрагмент кода, который нужно улучшить. Важно указать полную функцию или раздел, который вы хотите рефакторить, а иногда и немного окружающего контекста, если это уместно (например, использование функции или связанный код, который может повлиять на то, как вы проводите рефакторинг). Также укажите язык и фреймворк, потому что «идиоматический» код различается: скажем, между идиоматическим Node.js и идиоматическим Deno или компонентами класса React и функциональными компонентами. Например: «*У меня есть компонент React, написанный как класс. Пожалуйста, рефакторингуй его в функциональный компонент, используя хуки*». Затем ИИ применит типичные шаги (используя useState, useEffect и т. д.). Если вы просто написали «рефакторинг этого компонента React», не уточнив стиль, ИИ может не понять, что вам нужны именно хуки.

- **При необходимости укажите сведения о версии или среде**. Например, «*Это кодовая база Node.js v14*» или «*Мы используем модули ES6*» . Это может повлиять на использование ИИ определённого синтаксиса (например, import/export или require), что является частью корректного рефакторинга. Если вы хотите убедиться, что это не приведёт к несовместимости, укажите свои ограничения.

3. **Поощряйте пояснения вместе с кодом**. Отличный способ извлечь уроки из рефакторинга, проводимого ИИ (и убедиться в его корректности), — попросить объяснить изменения. Например: «*Пожалуйста, предложи рефакторинговую версию кода и объясни внесенные тобой улучшения*». Это даже было встроено в шаблон промпта, на который мы ссылались: «*…предложи рефакторинговый код с пояснениями твоих изменений*». Когда ИИ дает пояснение, вы можете оценить, понял ли он код и достиг ли ваших целей. В пояснении может быть сказано: «*Я объединил два похожих цикла в один, чтобы уменьшить дублирование, и использовал словарь для более быстрого поиска*» и т. д. Если в объяснении что-то звучит не так, это веский повод для тщательного изучения кода. Короче говоря, *используйте способность ИИ к объяснению в качестве меры предосторожности* — как если бы ИИ проводил проверку кода при собственном рефакторинге.

4. **Используйте ролевой промптинг для установки высоких стандартов**. Как упоминалось ранее, просьба к ИИ выступить в роли ревьюера кода или старшего инженера бывает очень эффективной. Для рефакторинга можно сказать: «*Веди себя как опытный эксперт по TypeScript и рефакторингуй этот код в соответствии с лучшими практиками и современными стандартами*». Это часто приводит не только к поверхностным изменениям, но и к более глубоким улучшениям, поскольку ИИ старается соответствовать персоне «эксперта». Популярным примером из руководства по промптам является использование ИИ в роли наставника: «*Веди себя как опытный разработчик Python, наставляющий новичка. Давай объяснения и пиши строки документации. Перепиши код, чтобы оптимизировать его*». Результатом в этом случае стало то, что ИИ использовал более эффективную структуру данных (настроенную на удаление дубликатов) и предоставил однострочное решение для функции, которая изначально использовала цикл. Ролевой промпт помог ему не только провести рефакторинг, но и объяснить, *почему* новый подход лучше (в этом случае

использование множества — хорошо известная оптимизация для уникальности).

Теперь рассмотрим пример рефакторинга, чтобы увидеть, как промпт может повлиять на результат. Мы используем сценарий на JavaScript (Node.js), где у нас есть неидеальный код, и мы хотим его улучшить.

Пример рефакторинга: плохой и улучшенный промпт

Предположим, у нас есть функция, которая делает два обращения к базе данных и выполняет некоторую обработку. Она работает, но выглядит не очень красиво: код дублируется и его сложно читать. Мы хотели бы провести рефакторинг для большей ясности и эффективности. Вот исходный код:

```
// Original function: Fetches two lists and processes them (needs refactoring)
async function getCombinedData(apiClient) {
  // Fetch list of users
  const usersResponse = await apiClient.fetch('/users');
  if (!usersResponse.ok) {
    throw new Error('Failed to fetch users');
  }
  const users = await usersResponse.json();
  // Fetch list of orders
  const ordersResponse = await apiClient.fetch('/orders');
  if (!ordersResponse.ok) {
    throw new Error('Failed to fetch orders');
  }
  const orders = await ordersResponse.json();
  // Combine data (match user to orders)
  const result = [];
  for (let user of users) {
    const userOrders = orders.filter(o => o.userId === user.id);
    result.push({ user, orders: userOrders });
  }
  return result;
}
```

Проблемы в этом коде: мы видим дублирование логики выборки для пользователей и заказов (можно преобразовать в вспомогательный метод или цикл), отсутствие подробной информации об ошибках (только общее сообщение об ошибке), и, возможно, стоит выполнять выборку параллельно для повышения скорости. Рассмотрим два подхода к промптам:

 Промпт (недоопределённый): «Сделай рефакторинг указанной выше функции getCombinedData».

Ответ ИИ на Промпт 1:

> *Конечно. Вот переработанная версия `getCombinedData`:

```
async function getCombinedData(apiClient) {
  // Fetch users and orders in parallel
  const [usersResponse, ordersResponse] = await Promise.all([
    apiClient.fetch('/users'),
    apiClient.fetch('/orders')
  ]);
  if (!usersResponse.ok || !ordersResponse.ok) {
    throw new Error('Failed to fetch data');
  }
  const [users, orders] = await Promise.all([
    usersResponse.json(),
    ordersResponse.json()
  ]);
  // Map orders by userId for efficient lookup
  const ordersByUser = {};
  for (const order of orders) {
    if (!ordersByUser[order.userId]) {
      ordersByUser[order.userId] = [];
    }
    ordersByUser[order.userId].push(order);
  }
  // Combine data
  return users.map(user => ({
    user,
    orders: ordersByUser[user.id] || []
  }));
}
```

> Я извлекал данные о пользователях и заказах параллельно для повышения производительности. Также консолидировал обработку ошибок и

> использовал карту (`ordersByUser`) для группировки заказов по пользователю, что делает этап комбинирования более понятным и потенциально более быстрым.*

Анализ: На самом деле, это довольно хороший рефакторинг, включающий несколько улучшений: параллельная выборка, группировка заказов по пользователю для повышения эффективности, более чистое комбинирование с использованием .map и использование пустого массива по умолчанию. ИИ также объяснил свои действия. Так почему же мы помечаем этот промпт как недостаточно определённый или «плохой»? В данном случае ИИ *сделал* правильный выбор, но со многими предположениями, поскольку промпт был открытым. ИИ предположил, что производительность должна быть улучшена (параллельные запросы были разумным изменением, но что, если параллельные запросы здесь небезопасны?), и изменил обработку ошибок на общее сообщение «Не удалось получить данные», которое при комбинировании ошибок теряло специфичность того, какой вызов завершился ошибкой.

В случае с общим запросом «Рефакторинг этого» риск заключается в том, что ИИ может внести изменения, которые непреднамеренно изменят поведение. В данном случае комбинированная обработка ошибок может быть приемлемой или нет, в зависимости от требований. Помощник также внёс структурное изменение (используя словарь для группировки заказов) — это нормально и, вероятно, к лучшему, но если у нас были какие-то определённые ограничения (например, желание транслировать результаты потоком или поддерживать низкий уровень использования памяти), мы не сообщали ему об этом. Поэтому, хотя результат выглядит хорошо, ИИ повезло, что его собственная интерпретация «рефакторинга» совпала с нашими ожиданиями. Это показывает, что недостаточно определённый запрос иногда даёт хорошие результаты в простых случаях; но в сложном коде он может повести вас в нежелательном направлении.

 Промпт 2 (целеориентированный): Сделай рефакторинг указанной выше функции `getCombinedData`, чтобы устранить дублирование кода и повысить производительность. В частности: (1) Избегай повторения логики выборки для пользователей и заказов — возможно, используй вспомогательный метод или выполняй их одновременно. (2) По возможности выполняй выборку обоих списков параллельно. (3) Сохрани обработку ошибок для каждой выборки (мы хотим знать, какой вызов завершился ошибкой). (4) Улучши комбинирование данных, возможно, используя более эффективную структуру поиска вместо вложенного цикла. Предоставь реорганизованный код с комментариями, объясняющими изменения».

Ответ ИИ на Промпт 2:

> *Конечно! Вот улучшенная версия `getCombinedData` с запрошенным рефакторингом:

```
async function getCombinedData(apiClient) {
  // 1. Fetch users and orders in parallel to improve performance
  const [usersResponse, ordersResponse] = await Promise.all([
    apiClient.fetch('/users'),
    apiClient.fetch('/orders')
  ]);

  // 2. Handle errors separately to know which call failed
  if (!usersResponse.ok) {
    throw new Error('Failed to fetch users');
  }
  if (!ordersResponse.ok) {
    throw new Error('Failed to fetch orders');
  }

  // Parse JSON responses in parallel
  const [users, orders] = await Promise.all([
    usersResponse.json(),
    ordersResponse.json()
  ]);

  // 3. Use a lookup map for orders by userId for efficient matching
  const ordersByUser = orders.reduce((map, order) => {
    if (!map[order.userId]) {
      map[order.userId] = [];
    }
    map[order.userId].push(order);
    return map;
  }, {});

  // 4. Combine data using the lookup map
  return users.map(user => ({
    user,
    orders: ordersByUser[user.id] || []  // empty array if no orders for user
  }));
}
```

Объяснение изменений:

- Параллельная выборка: я использовал `Promise.all` для одновременной выборки пользователей и заказов (пункт 1).
- Сохранена индивидуальная обработка ошибок: я сохранил отдельные проверки для `usersResponse.ok` и `ordersResponse.ok`, чтобы любая ошибка чётко указывала, какая именно выборка не удалась (пункт 2).
- Параллельный парсинг JSON: также используется `Promise.all` для вызовов `response.json()`, что является незначительным улучшением.
- Эффективное сочетание: вместо вложенного цикла я создал карту `ordersByUser` (пункт 3). Она группирует заказы по `userId` за один проход. Затем я сопоставляю пользователей для построения результата, извлекая заказы каждого пользователя из карты за время O(1) (пункт 4). Это более эффективно и лаконично.*

Анализ: Этот ответ напрямую решает все поставленные задачи. Код рефакторингован, чтобы быть чище и быстрее, и поддерживает отдельные сообщения об ошибках, как и требовалось. Объяснение ИИ подтверждает каждый из перечисленных нами пунктов, что показывает: он внимательно следовал инструкциям подсказки. Это отличный результат, потому что мы, как подсказчик, определили, что означает «рефакторинг» в данном контексте. Тем самым направили ИИ к созданию решения, которое соответствует нашим потребностям, с минимальными затратами времени. Если бы ИИ упустил один из пунктов (например, всё же объединил обработку ошибок), мы могли бы легко подсказать ещё раз: «Выглядит хорошо, но, пожалуйста, убедись, что сообщения об ошибках остаются разными для пользователей и заказов». Однако в данном случае в этом не было необходимости, поскольку наш запрос был подробным.

Этот пример демонстрирует ключевой урок: **когда вы знаете, что хотите улучшить, сформулируйте это**. ИИ хорошо выполняет инструкции, но не читает ваши мысли. Обобщённая фраза «сделай это лучше» может сработать для простых задач, но для сложного кода наилучших результатов вы добьётесь, если перечислите, что для вас означает «лучше». Это согласуется с мнением сообщества о том, что чёткие, структурированные промпты дают значительно лучшие результаты.

Дополнительные советы по рефакторингу:

- *Поэтапный рефакторинг*: если код очень большой или у вас длинный список изменений, вы можете выполнять их по одному. Например, сначала попросите ИИ «*выполнить рефакторинг для удобства чтения*» (сосредоточиться на переименовании и разделении функций), а затем «*оптимизировать алгоритм в этой функции*». Это предотвратит перегрузку модели слишком большим количеством инструкций одновременно и позволит поэтапно проверять каждое изменение.
- *Спросите об альтернативных подходах*: Возможно, первый рефакторинг ИИ сработает, но вам интересно взглянуть на него с другой стороны. Вы можете спросить: «*Можно ли провести рефакторинг по-другому, например, используя функциональный стиль программирования (например, методы массивов вместо циклов)?*» или «*Как насчёт использования рекурсии вместо итеративного подхода, просто для сравнения?*». Таким образом вы сможете оценить различные решения.

Это как штурмить с коллегой, обдумывая несколько вариантов рефакторинга.
- *Сочетайте рефакторинг с объяснениями для изучения закономерностей*: мы уже затрагивали этот вопрос, но стоит подчеркнуть: используйте ИИ как инструмент обучения. Если он грамотно рефакторит код, изучите результаты и объяснения. Возможно, вы откроете для себя новый API или метод (например, использование reduce для построения карты), который раньше не использовали.
- *Проверка и тестирование*: После любого рефакторинга, сгенерированного ИИ, всегда запускайте тесты или проверяйте код с примерами входных данных. ИИ может непреднамеренно внести малозаметные ошибки, особенно если в запросе не было указано важное ограничение. В нашем рефакторинге, если исходный код намеренно разделял ошибки выборки для логирования, но мы не упомянули о логировании, комбинированная ошибка может быть менее полезной. Наша задача — выявить это при анализе. ИИ также может помочь, написав тесты — вы можете попросить «*Сгенерировать несколько модульных тестов для рефакторингованной функции*», чтобы убедиться, что она ведёт себя так же, как и раньше, при ожидаемых входных данных.

На этом этапе мы рассмотрели отладку и рефакторинг — улучшение существующего кода. Следующий логичный шаг — использование ИИ для *реализации новых функций* или генерации нового кода. Мы рассмотрим, как эффективно использовать промпты для этого сценария.

Современные сценарии отладки

Проблемы с зависимостями React Hook

 Плохой промпт: «Мой useEffect работает неправильно»

 Улучшенный промпт:

```
I have a React component that fetches user data, but it's causing infinite re-renders. Here's
my code:



const UserProfile = ({ userId }) => {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetchUser(userId).then(setUser).finally(() => setLoading(false));
  }, [userId, setUser, setLoading]); // Problem is here

  return loading ? <div>Loading...</div> : <div>{user?.name}</div>;
};
```

Ожидаемое поведение: необходимо извлекать данные пользователя один раз при изменении `userId`. Фактическое поведение: компонент бесконечно перерисовывается. Ошибка в консоли: «Предупреждение: превышена максимальная глубина обновления».

Что является причиной этого бесконечного цикла и как исправить массив зависимостей?

Почему это работает: предоставляет точный код, сообщение об ошибке, ожидаемое и фактическое поведение, а также фокусируется на конкретном шаблоне React, который часто понимается неправильно.

Современная архитектура состояний

 Плохой промпт: «Создай управление состоянием для моего приложения электронной коммерции Next.js»

 Улучшенный промпт: Я разрабатываю приложение электронной коммерции на Next.js 14 и мне нужно спроектировать архитектуру управления состоянием. Вот требования:

Компоненты:

- Страница со списком товаров (требуется: товары[], фильтры, пагинация)
- Корзина покупок (требуется: товары в корзине, итоги, информация о доставке)
- Авторизация пользователя (требуется: профиль пользователя, статус авторизации, настройки)
- Уведомления в режиме реального времени (требуется: всплывающие сообщения, состояния ошибок)

Технические ограничения:

Next.js 14 с компонентами App Router и Server

- Строгий режим TypeScript
- Извлечение данных на стороне сервера для SEO
- Интерактивность на стороне клиента для действий корзины/пользователя
- Состояние должно сохраняться на протяжении всей навигации.

Следует ли мне использовать:

1. Сторы Zustand для каждого домена (корзина, авторизация, уведомления)
2. React Query/TanStack Query для состояния сервера + Zustand для состояния клиента
3. Один стор Zustand со слайсами

Предоставь рекомендуемую архитектуру с примерами кода, демонстрирующими, как структурировать хранилища и интегрировать их с шаблонами Next.js App Router.

Почему это работает: Реальный сценарий с определенным технологическим стеком, четкими требованиями и запросом архитектурного руководства с подробностями реализации.

Шаблоны промптов для внедрения новых фичей

Одно из самых интересных применений ИИ-помощников по кодированию — помощь в написании нового кода с нуля или интеграции новой фичи в существующую кодовую базу. Задачи могут варьироваться от создания шаблона для компонента React до написания новой конечной точки API в приложении Express. Сложность здесь часто заключается в том, что эти задачи не имеют чёткого определения — есть множество способов реализации фичи.

Принцип промпт-инжиниринга для генерации кода: научить ИИ создавать код, соответствующий вашим потребностям и стилю. Вот стратегии для этого:

1. **Начните с общих инструкций, а затем углубитесь в детали**. Для начала опишите простым языком, что вы хотите реализовать, возможно, разбив это на более мелкие задачи (аналогично совету по разбиению сложных задач, данному ранее). Например, вы хотите добавить **функцию панели поиска** в существующее веб-приложение. Сначала можно запросить: «*Опиши план добавления функции поиска, которая фильтрует список товаров по названию в моём React-приложении. Товары извлекаются из API*».

ИИ может предоставить вам пошаговый план: «1. Добавьте поле ввода для поискового запроса. 2. Добавьте состояние для хранения запроса. 3. Отфильтруйте список товаров на основе запроса. 4. Убедитесь, что он нечувствителен к регистру и т. д.». Получив этот план (который можно уточнить посредством ИИ), вы сможете обрабатывать каждый пункт с помощью конкретных промптов.

Например: «*Хорошо, реализуем шаг 1: создадим компонент* `SearchBar` *с полем ввода, обновляющим состояние* `searchQuery`*». После этого: «Реализуем шаг 3: по заданному* `searchQuery` *и массиву товаров отфильтруем товары (сравнение по имени без учёта регистра)*». Разделяя функцию, вы обеспечиваете конкретику каждого запроса и управляемость ответами. Это также отражает логику итеративной разработки — вы можете тестировать каждый элемент по мере его создания.

2. **Предоставьте релевантный контекст или справочный код**. Если вы добавляете фичу в существующий проект, очень полезно показать ИИ, как в нём реализованы похожие фичи. Например, если у вас уже есть компонент, похожий на тот, который вам нужен, вы можете сказать: «*Вот существующий компонент* `UserList` *(код...). Теперь создай компонент* `ProductList`*, похожий на него, но с панелью поиска*».

ИИ распознает шаблоны (возможно, вы используете определённые библиотеки или

соглашения по стилю) и применит их. Открытие соответствующих файлов или ссылки на них в командной строке создают контекст, который приводит к более точным и последовательным предложениям по коду, специфичным для проекта. Ещё один приём: если в вашем проекте используется определённый стиль кодирования или архитектура (например, Redux для управления состоянием или конкретный CSS-фреймворк), упомяните об этом. «*Мы используем Redux для управления состоянием — интегрируй состояние поиска в хранилище Redux*».

Хорошо обученная модель будет генерировать код, соответствующий шаблонам Redux и т. д. По сути, вы обучаете ИИ **среде вашего проекта** , чтобы он мог адаптировать выводимые данные. Некоторые помощники могут даже использовать весь ваш репозиторий в качестве контекста для работы; если вы используете их, убедитесь, что указали ему аналогичные модули или документацию в вашем репозитории.

- Если вы начинаете что-то новое, но у вас есть предпочтительный подход, можете упомянуть следующее: «*Я бы хотел реализовать это с помощью функционального стиля программирования (без внешнего состояния, с использованием методов массива)*». Или: «*Обязательно следуй шаблону MVC и помести логику в контроллер, а не в представление*». Это те детали, о которых старший инженер мог бы напомнить младшему, и вот вы, старший, рассказываете ИИ .

3. **Используйте комментарии и TODO в качестве встроенных промптов**. При работе непосредственно в IDE с Copilot, один из эффективных рабочих процессов — это написание комментария, описывающего следующий фрагмент кода, который вам нужен, а затем автодополнение со стороны ИИ. Например, в бэкенде Node.js можно написать: // TODO: проверить полезную нагрузку запроса (убедитесь, что указаны имя и адрес электронной почты), а затем начать следующую строку. Copilot часто определяет намерение и генерирует блок кода, выполняющий эту проверку. Это работает, поскольку ваш комментарий фактически представляет собой промпт на естественном языке. Однако будьте готовы отредактировать сгенерированный код, если ИИ неправильно его интерпретирует — как всегда, проверяйте корректность.

4. **Приведите примеры ожидаемых входных/выходных данных или использования**. Аналогично тому, что мы обсуждали ранее, если вы просите ИИ реализовать новую функцию, включите краткий пример её использования или простой тестовый пример. Например: «*Реализуй функцию* `formatPrice(amount)` *в JavaScript, которая принимает число (например, 2,5) и возвращает строку, отформатированную в долларах США (например, 2,50 доллара США). Например,* `formatPrice(2,5)` *должна возвращать '$2,50'*».

С помощью примера вы ограничиваете ИИ для достижения нужного вам результата. Без этого примера ИИ мог бы использовать другое форматирование или валюту. Разница может быть незначительной, но важной. Другой пример в веб-контексте: «*Реализуй миделвару Express, которая регистрирует запросы. Например, GET-запрос к /users должен выводить „GET /users“ на консоль*». Это даёт понять, как должен выглядеть вывод. Включение ожидаемого поведения в промпт служит своего рода проверкой, которую ИИ попытается выполнить.

5. **Если результат не тот, что вы хотели, перепишите промпт, добавив больше подробностей или ограничений**. Часто первая попытка создания новой фичи не удаётся. Возможно, код запускается, но не является идиоматичным, или в нём не хватает какого-либо требования. Вместо того, чтобы расстраиваться, отнеситесь к ИИ как к младшему разработчику, который сделал первый черновик — теперь вам нужно дать обратную связь. Например, «*Решение работает, но я бы предпочёл, чтобы ты использовал встроенный метод фильтрации массива вместо цикла for*». Или «*Можешь ли реорганизовать сгенерированный компонент, чтобы использовать React Hooks для состояния вместо компонента класса? Наша кодовая база полностью состоит из функциональных компонентов*». Вы также можете добавить новые ограничения: «*Кроме того, убедись, что функция выполняется за время O(n) или быстрее, поскольку n может быть большим*». Такое итеративное подсказывание очень полезно. Реальный сценарий: один разработчик попросил LLM сгенерировать код для рисования рожка мороженого с использованием библиотеки JS Canvas, но сталкивался с нерелевантным выводом — пока не уточнил промпт, добавив больше конкретики и контекста. Урок таков: *не сдавайтесь после первой попытки*. Выясните, чего не хватало или что было неверно понято в промпте, и уточните это. В этом суть проектирования промптов: каждое изменение может приблизить модель к вашему замыслу.

Давайте проиллюстрируем промпт по реализации фичи на примере сценария.

Пример реализации фичи: создание компонента React с помощью ИИ

Предположим, у нас есть React-приложение, и мы хотим добавить новый компонент: список товаров с фильтром поиска и раскрывающимся списком сортировки. Это довольно типичная функция веб-приложения. Для краткости сосредоточимся на реализации функции поиска в этом компоненте.

Начальный промпт (запрос на фичу): «Создай функциональный компонент React под названием `ProductList`, который отображает список товаров и включает текстовое поле для фильтрации товаров по названию. Компонент должен: извлекать массив товаров из `/api/products` (предположим, что он возвращает JSON-массив `{id, name, ...}`), сохранять их в состоянии и позволять пользователю вводить данные в поле поиска для фильтрации отображаемых товаров по названию (регистрозависимый поиск). Отображать отфильтрованный список в простом теге <ul> с названиями товаров. Включать базовую обработку ошибок (например, сообщение об ошибке при сбое вызова API) и состояние загрузки во время извлечения».

Это длинный запрос, но он конкретен. Предположим, мы передаём его LLM в чате.

Ответ ИИ (*краткое изложение того, что он может выдать*): ИИ, скорее всего, выведет компонент `ProductList` примерно следующего вида:

```
import React, { useState, useEffect } from 'react';

function ProductList() {
  const [products, setProducts] = useState([]);
  const [search, setSearch] = useState('');
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function fetchProducts() {
      try {
        const res = await fetch('/api/products');
        if (!res.ok) throw new Error('Network response was not ok');
        const data = await res.json();
        setProducts(data);
      } catch (err) {
        console.error('Failed to fetch products:', err);
        setError('Failed to load products.');
      } finally {
        setLoading(false);
      }
    }
    fetchProducts();
  }, []);

  const handleSearchChange = (e) => {
    setSearch(e.target.value);
  };

  // Filter products by search query (case-insensitive)
  const filteredProducts = products.filter(product =>
    product.name.toLowerCase().includes(search.toLowerCase())
  );

  if (loading) return <div>Loading products...</div>;
  if (error) return <div>{error}</div>;

  return (
    <div className="product-list">
      <input
```

```
        type="text"
        placeholder="Search products"
        value={search}
        onChange={handleSearchChange}
      />
      <ul>
        {filteredProducts.map(prod => (
          <li key={prod.id}>{prod.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default ProductList;
```

(Плюс, возможно, некоторые комментарии в коде, поясняющие каждую часть, если мы их запросим.)

Теперь давайте проанализируем этот результат и рассмотрим возможные уточнения:

- Предположим, код выше вернулся, но наш проект использует контекст или глобальное хранилище для извлечения данных, а не извлечения внутри компонента. Мы этого не указали. ИИ выбрал извлечение в `useEffect`. Если мы предпочитаем использовать пользовательский хук или контекст, нам следовало бы указать на это. Таким образом, мы могли бы уточнить наше сообщение: «*На самом деле, в нашем приложении мы используем пользовательский хук `useProducts()`, который уже обрабатывает извлечение. Пожалуйста, реорганизуй компонент, чтобы использовать хук `useProducts` вместо прямого вызова `fetch`*». ИИ затем скорректирует его, возможно, предположив, что `useProducts` возвращает `{ products, loading, error }`, и соответствующим образом упростит компонент.
- Ещё одно уточнение: возможно, мы осознаём, что нам также нужен раскрывающийся список сортировки (о котором мы изначально не упоминали). Расширим диалог: «*Отлично, теперь добавим раскрывающийся список для сортировки товаров по названию (AZ или ZA). В раскрывающемся списке пользователь должен выбирать сортировку по возрастанию или убыванию, а список должен сортироваться соответствующим образом в дополнение к фильтрации*». Поскольку у ИИ есть контекст существующего кода, он может добавить состояние сортировки и скорректировать рендеринг. Мы предоставили чёткое новое требование, и ИИ попытается его выполнить, вероятно, добавив что-то вроде:

```
const [sortOrder, setSortOrder] = useState('asc');
// ... a select input for sortOrder ...
```

```
// and sort the filteredProducts before rendering:
const sortedProducts = [...filteredProducts].sort((a, b) => {
  if (sortOrder === 'asc') return a.name.localeCompare(b.name);
  else return b.name.localeCompare(a.name);
});
```

- (плюс раскрывающийся пользовательский интерфейс).
  Так, функция за функцией, мы имитируем цикл разработки с помощью ИИ. Это гораздо эффективнее, чем пытаться сразу задать весь сложный компонент со всеми функциями. Поскольку снижает количество ошибок и позволяет вносить исправления в процессе разработки, по мере прояснения требований.
- Если ИИ допускает незначительную ошибку (например, забыл сделать фильтр поиска нечувствительным к регистру), мы просто указываем на неё: «*Сделай поиск нечувствительным к регистру*». Он настроит фильтр на использование сравнения со строчными буквами.
- Этот пример показывает, что реализация функций с помощью ИИ — **постепенная разработка и оперативное совершенствование**. В Твиттере можно было бы похвастаться тем, как кто-то создал небольшое приложение, постоянно используя LLM для каждой составляющей — по сути, это и есть подход: разработать, проверить, доработать, расширить. Каждый промпт — своего рода коммит в процессе разработки.

Дополнительные советы по реализации фичей:

- *Позвольте ИИ создать шаблон и сами внесите детали*: иногда полезно, чтобы ИИ сгенерировал приблизительную структуру, а вы её дорабатывали. Например, «*Сгенерируй скелет маршрута Node.js Express для регистрации пользователей с валидацией и обработкой ошибок*». Он может создать общий маршрут с плейсхолдерами. Затем вы можете заполнить правила валидации или вызовы базы данных, специфичные для вашего приложения. ИИ избавит вас от необходимости писать шаблонный код, а сами вы займётесь пользовательской логикой (если она важна).
- *Запросите обработку пограничных случаев*: при создании функции можете предложить ИИ подумать о пограничных случаях: «*Какие пограничные случаи следует учитывать для этой функции (и можно ли их обработать в коде)?*»
  Например, в примере с поиском пограничным случаем может быть «что, если товары ещё не загружены, когда пользователь вводит текст?» (хотя наш код обрабатывает это через состояние загрузки) или «что, если у двух товаров одинаковые названия» (не такая уж большая проблема, но стоит упомянуть о ней). ИИ может упомянуть такие вещи, как обработка пустых результатов, очень большие списки (возможно, потребуется задержка при поиске) и т. д. Это способ обучать ИИ на распространённых ошибках.
- *Разработка на основе документации*: некоторые используют интересный подход: сначала пишут строку документации или пример использования, а затем ИИ

реализует соответствующую функцию. Например:

```
/**
 * Returns the nth Fibonacci number.
 * @param {number} n - The position in Fibonacci sequence (0-indexed).
 * @returns {number} The nth Fibonacci number.
 *
 * Example: fibonacci(5) -> 5  (sequence: 0,1,1,2,3,5,…)
 */
function fibonacci(n) {
  // ... implementation
}
```

- Если вы напишете комментарий и сигнатуру функции, представленные выше, LLM может правильно выполнить реализацию, поскольку комментарий точно описывает, что нужно делать, и даже приводит пример. При таком подходе вы сперва объясняете функцию словами (что, как правило, является хорошей практикой), а затем ИИ использует это в качестве спецификации для написания кода.

Рассмотрев стратегии использования промптов для отладки, рефакторинга и генерации нового кода, коснемся некоторых **распространённых ошибок и антипаттернов при проектировании промптов для программирования**. Понимание этих ошибок поможет вам избежать траты времени на непродуктивные взаимодействия и быстро адаптироваться, когда ИИ не даёт нужного результата.

Распространенные анти-шаблоны промптов и как их избежать

Не все промпты одинаковы. Мы уже видели множество примеров эффективных промптов, но не менее полезно распознавать **антишаблоны** — распространённые ошибки, которые приводят к неэффективным реакциям ИИ.

Вот некоторые распространенные сбои в работе промптов и способы их устранения:

- **Антипаттерн: Неопределённый промпт**. Классическое «Это не работает, пожалуйста, исправь это» или «Напиши что-нибудь, что делает X» без достаточных подробностей. Мы видели похожий пример, когда на вопрос «*Почему моя функция не работает?*» получили бесполезный ответ. Нечёткие промпты заставляют ИИ угадывать контекст и часто приводят к общим советам или нерелевантному коду. Решение простое: **добавьте контекст и конкретику**. Если вы задаёте вопрос, а ответ получаете будто из волшебного шара («*Вы пробовали проверить X?*»), переформулируйте запрос, добавив деталей (сообщения об ошибках, отрывок кода, ожидаемый и фактический результат и т. д.). Хорошей практикой будет прочитать ваш промпт и спросить: «*Может ли этот запрос применяться к десяткам различных сценариев?*» Если да, он слишком расплывчатый. Сделайте промпт настолько конкретным, чтобы он был применим только к вашему сценарию.
- **Антипаттерн: Перегруженный промпт**. Обратная проблема: попросить ИИ сделать слишком много вещей одновременно. Например, «*Сгенерируй полное приложение Node.js с аутентификацией, фронтендом на React и скриптами развертывания*». Или даже в меньшем масштабе: «*Исправь эти 5 ошибок, а также добавьте эти 3 функции за один раз*». ИИ может попытаться это сделать, но вы, скорее всего, получите беспорядочный или неполный результат; также ИИ может проигнорировать некоторые части запроса. Даже если он ответит на все вопросы, ответ будет длинным и его будет сложнее проверить. Решение — разделить задачи. Расставьте приоритеты: двигайтесь пошагово, как мы подчеркивали ранее. Это облегчает обнаружение ошибок и гарантирует, что модель останется сфокусированной. Если вы поймали себя на том, что пишете абзац с несколькими

«и» в инструкциях, рассмотрите возможность разбить его на отдельные промпты или последовательные шаги.

- **Антипаттерн: Пропуск вопроса**. Иногда пользователи предоставляют много информации, но никогда чётко не задают вопрос или не уточняют, что им нужно. Например, выкладывают большой фрагмент кода и просто сообщают: «*Вот мой код* ». Это может сбить ИИ с толку — он не знает, чего вы хотите. Всегда включайте чёткий запрос, например: «*Определи любые ошибки в приведенном выше коде*», « *Объясни, что делает этот код*» или «*Выполни TODO в коде*». Промпт должен иметь цель. Если вы просто предоставите текст без вопроса или инструкции, ИИ может сделать неверные предположения (например, резюмировать код вместо того, чтобы исправить его и т. д.). Убедитесь, что ИИ знает, почему вы показали ему какой-то код. Даже простое дополнение, например: «*Что не так с этим кодом?*» или « *Пожалуйста, продолжай реализовывать эту функцию*», даёт ему направление.
- **Антипаттерн: Размытые критерии успеха**. Это тонкий момент — иногда вы можете попросить об оптимизации или улучшении, но не определить, как выглядит успех. Например, «*Сделай эту функцию быстрее*». Быстрее по какой метрике? Если ИИ не знает ваших ограничений производительности, он может микрооптимизировать что-то неважное или использовать подход, который теоретически быстрее, но практически незначителен. Или «*сделай этот код чище*» — «чище» субъективно. Мы справились с этим, явно указав цели, такие как «уменьшить дублирование» или «улучшить имена переменных» и т. д. Решение: количественно оценить или квалифицировать улучшение . Например, «*оптимизируй эту функцию для работы за линейное время (текущая версия квадратичная)*» или « *проведи рефакторинг, чтобы удалить глобальные переменные и вместо этого использовать класс*». По сути, чётко укажите, какую проблему вы решаете с помощью рефакторинга или фичи. Иначе ИИ может взяться вовсе не за ту проблему, которую вы подразумевали.
- **Антипаттерн: Игнорирование уточнений или выводов ИИ**. Иногда ИИ может ответить уточняющим вопросом или предположением. Например: «*Вы используете компоненты класса React или функциональные компоненты?*» или «*Я предполагаю, что входные данные представляют собой строку — подтвердите, пожалуйста*». Если вы проигнорируете их и просто повторите свой запрос, вы упустите возможность улучшить промпт. ИИ сигнализирует, что ему нужно больше информации. Всегда отвечайте на его вопросы или уточняйте свой промпт, включив эти детали. Кроме того, если вывод ИИ явно неверен (например, он неправильно понял вопрос), не повторяйте тот же промпт дословно. Уделите время, чтобы скорректировать формулировку. Возможно, в вашем промпте была двусмысленная фраза или было упущено что-то важное. Относитесь к этому как к разговору — если бы человек неправильно понял, вы бы объяснили по-другому; сделайте то же самое для ИИ.
- **Антипаттерн: Разный стиль или непоследовательность.** Если вы постоянно меняете способ подачи вопроса или смешиваете разные форматы, модель может запутаться. Например, переключение между первым и третьим лицом в инструкциях или микс псевдокода с реальным кодом может сбить ИИ с толку. Старайтесь придерживаться единого стиля в рамках одного запроса. Если вы приводите примеры, убедитесь, что они чётко обозначены (используйте тройные

обратные кавычки Markdown для кода, кавычки для примеров ввода/вывода и т. д.). Последовательность помогает модели правильно проанализировать ваши намерения. Кроме того, если у вас есть предпочтительный стиль (например, синтаксис ES6 или ES5), постоянно упоминайте его, иначе модель может предложить один вариант в одном запросе и другой в другом.
- **Антипаттерн: Неопределённые ссылки, такие как «код выше»**. Если вы используете чат и пишете «функция выше» или «предыдущий вывод», убедитесь, что ссылка понятна. Если в длинной беседе вы пишите «рефакторинг кода выше», ИИ может потерять нить обсуждения или выбрать не тот фрагмент кода для рефакторинга. Безопаснее либо снова процитировать код, либо конкретно указать функцию, которую вы хотите рефакторить. Окно внимания моделей ограничено, и хотя многие LLM могут ссылаться на предыдущие части диалога, повторное указание явного контекста может помочь избежать путаницы. Это особенно актуально, если с момента отображения кода прошло некоторое время (или было несколько сообщений).

Наконец, вот **тактический подход к переписыванию промптов**, когда что-то идет не так:

- **Определите, что было пропущено или некорректно в ответе ИИ**. Решил ли он другую задачу? Возникла ли ошибка или предложено неподходящее решение? Возможно, вы запросили решение на TypeScript, но он выдал простой JavaScript. Или он написал рекурсивное решение, хотя вы явно хотели итеративное. Найдите несоответствие.
- **Добавьте или подчеркните это требование в новом запросе**. Вы можете сказать: «*Решение должно быть на TypeScript, а не на JavaScript. Пожалуйста, включи аннотации типов*». Или: «*Я уже упоминал, что мне нужно итеративное решение — пожалуйста, избегай рекурсии и используй цикл*». Иногда полезно буквально использовать фразы вроде «Примечание:» или «Важно:» в запросе, чтобы выделить ключевые ограничения (у модели нет эмоций, но она учитывает определённые фразы как важные). Например: «Важно: не используй для этого внешние библиотеки» или «Примечание: код должен выполняться в браузере, поэтому API, специфичные для Node, не требуются» .
- **При необходимости разбейте запрос на более мелкие части**. Если ИИ не справляется со сложным запросом, попробуйте разделить его. Или задайте вопрос, который может прояснить ситуацию: «*Ты понимаешь, что я имею в виду под X?*» Модель может перефразировать то, что, по её мнению, вы имеете в виду, и вы сможете исправить её при необходимости. Это метапромт — обсуждение самого промпта — и иногда он может разрешить недопонимание.
- **Если ветка застряла, попробуйте начать заново**. Иногда после нескольких попыток разговор заходит в тупик. Можно начать новый сеанс (либо на время очистить историю чата) и задать вопрос с нуля, более точно сформулированный с учётом предыдущих неудач. Модель не боится повторений, а свежий контекст поможет устранить любую путаницу, накопившуюся в предыдущих сообщениях.

Зная эти антипаттерны и способы их решения, вы сможете гораздо быстрее корректировать промпты на ходу. Разработка промптов для разработчиков — это, по сути, итеративный процесс, основанный на обратной связи (как и любая задача программирования!). Хорошая новость в том, что теперь в вашем инструментарии есть множество шаблонов и примеров, из которых можно черпать вдохновение.

Заключение

Разработка промптов — это одновременно и искусство, и наука, и, как мы видим, она быстро становится обязательным навыком для разработчиков, работающих с ИИ-помощниками по кодированию. Создавая понятные, контекстно-обогащённые промпты, вы, по сути, обучаете ИИ тому, что вам нужно — так же, как если бы нанимали нового члена команды или объясняли проблему коллеге. В этой статье мы рассмотрели, как систематически подходить к промптам для отладки, рефакторинга и реализации функций:

- Научились передавать ИИ ту же информацию, которую вы даете коллеге, прося его о помощи: что должен делать код, в чем его ошибка, соответствующие фрагменты кода и т. д. — тем самым получая гораздо более адресную помощь.
- Увидели всю мощь итераций с ИИ, будь то пошаговое выполнение логики функции строка за строкой или уточнение решения с помощью нескольких промптов (например, преобразование рекурсивного решения в итеративное, а затем улучшение имён переменных). Терпение и итерации превращают ИИ в настоящего напарника-программиста, а не генератора одноразового кода.
- Использовали ролевой промптинг для повышения уровня ответов, обращаясь с ИИ как с проверяющим код, наставником или экспертом в определённом стеке. Это часто приводит к более точным и подробным результатам, которые не только решают проблему, но и обучают нас в процессе.
- В рамках рефакторинга и оптимизации уделили особое внимание определению того, что такое «хорошо» (быстрее, чище, более идиоматично и т. д.), и ИИ продемонстрировал, что может применять известные передовые практики под руководством (например, распараллеливание вызовов, удаление дублирования, корректную обработку ошибок). Это похоже на доступ к коллективной мудрости бесчисленных рецензентов кода, но чтобы воспользоваться ею, нужно задавать правильные вопросы.
- Также продемонстрировали пошаговое создание новых функций с помощью ИИ, показав, что даже сложные задачи можно разложить на части и решать по одному запросу за раз. ИИ может создавать шаблоны, предлагать варианты реализации и даже выделять пограничные случаи при необходимости, выступая в роли компетентного соразработчика, который всегда готов помочь.
- По ходу дела выявили подводные камни, которых следует избегать: не делать промпты слишком обобщенными и не перегружать их, всегда указывать наши намерения и ограничения, а также быть готовыми к корректировке, если результат работы ИИ не соответствует целевому. Мы привели конкретные примеры неудачных промптов и увидели, как незначительные изменения (например, добавление

сообщения об ошибке или ожидаемого результата) могут значительно улучшить результат.

Внедряя эти методы в свой рабочий процесс, вы, вероятно, обнаружите, что работа с ИИ становится более интуитивно понятной. Вы научитесь понимать, какие формулировки дают наилучшие результаты и как направлять модель, когда она отклоняется от курса. Помните, что ИИ — это продукт его обучающих данных: он видел множество примеров кода и решений задач, но именно вы указываете, какие из этих примеров актуальны сейчас. По сути, **вы задаёте контекст, а ИИ следует ему** .

Стоит также отметить, что проектирование промптов — развивающаяся практика. Сообщество разработчиков постоянно открывает новые приёмы: умный однострочный промпт или структурированный шаблон могут внезапно стать вирусными в социальных сетях, поскольку они открывают возможности, о существовании которых люди даже не подозревали. Следите за этими обсуждениями (в Hacker News, Twitter и т. д.), они могут вдохновить вас на создание собственных методов. Но не бойтесь экспериментировать. Относитесь к ИИ как к гибкому инструменту: если у вас есть идея («что, если попросить его нарисовать ASCII-диаграмму моей архитектуры?»), просто попробуйте. Результаты могут вас удивить, а если ничего не получится, ничего страшного — вы узнали что-то новое об ограничениях или потребностях модели.

И наконец, помните, что промптинг — это **итеративный диалог** . Относитесь к ИИ с той же ясностью, терпением и тщательностью, с которой вы общаетесь с другим инженером. Сделайте это, и вы обнаружите, что ИИ-помощники могут значительно расширить ваши возможности, помогая быстрее отлаживать код, эффективнее проводить рефакторинг и проще внедрять фичи.

Удачных промптов и удачного кодинга!

Revision #2
Created 2026-02-05 07:33:44 UTC by Эд Кукса
Updated 2026-02-05 07:46:16 UTC by Эд Кукса