12+

ВИТАЛИЙ ФАРТУШНОВ

Вайб-кодинг (vibe coding). Паттерны проектирования

Виталий Фартушнов

Вайб-кодинг (vibe coding). Паттерны проектирования

«Издательские решения»

Фартушнов В.

Вайб-кодинг (vibe coding). Паттерны проектирования /
В. Фартушнов — «Издательские решения»,

Вайб-кодинг — это подход к программированию, где человек описывает задачу на обычном языке, а ИИ генерирует код, помогает его править и доводить до рабочего состояния. Идея в том, что разработчик больше формулирует цель и проверяет результат, чем пишет всё вручную.

Содержание

Вайб-кодинг (vibe coding). Паттерны проектирования

Виталий Фартушнов

ISBN 978-5-0070-0909-6
Создано в интеллектуальной издательской системе Ridero

Вайб-кодинг (vibe coding) — описать подробно

Вайб-кодинг — это подход к программированию, где человек описывает задачу на обычном языке, а ИИ генерирует код, помогает его править и доводить до рабочего состояния. Идея в том, что разработчик больше формулирует цель и проверяет результат, чем пишет всё вручную [1] [2].

Как это работает

Обычно процесс выглядит так:

— Вы формулируете, что нужно сделать: например, «сделай форму входа с email, паролем и кнопкой».

— ИИ предлагает код или сразу несколько вариантов реализации.

— Вы уточняете требования, исправляете поведение, просите рефакторинг.

— Вы тестируете результат и при необходимости откатываете изменения через контроль версий [1] [3].

Главная особенность вайб-кодинга — работа идет итерациями, через диалог с моделью, а не через ручное пошаговое написание всего кода [4] [3].

Чем он полезен

Вайб-кодинг особенно хорош для быстрого прототипирования, MVP, лендингов, простых веб-приложений и шаблонных задач. Он ускоряет старт проекта и помогает быстро проверить идею без долгой ручной разработки [1] [5].

Еще он полезен новичкам: можно собирать простые приложения, даже не зная синтаксис языка в деталях, если умеешь четко описывать задачу и проверять результат [2] [6].

Ограничения и риски

У этого подхода есть важные минусы. ИИ может сгенерировать код, который выглядит рабочим, но содержит ошибки, плохую архитектуру или уязвимости, поэтому результат нужно внимательно проверять [3] [7].

Для сложных систем, где важны производительность, безопасность и надежность, традиционная разработка все еще остается необходимой. Вайб-кодинг лучше всего работает как ускоритель, а не как полная замена инженерной дисциплины [3] [8].

Где уместен

Лучше всего вайб-кодинг подходит для:
— Быстрых прототипов.
— Простых внутренних утилит.
— Лендингов и небольших веб-сервисов.
— Экспериментов с UI и бизнес-логикой.
— Рутинных фрагментов кода и шаблонов [5] [8].
Хуже он подходит для:
— Крупных backend-систем.
— Критичных финансовых или медицинских сервисов.
— Нагрузочных и высоконадежных систем.
— Проектов, где нужен строгий контроль качества и безопасности [3] [7].

Как делать лучше

Чтобы вайб-кодинг не превратился в хаос, обычно помогают несколько правил: давать конкретные инструкции, решать одну задачу за раз, просить модель задавать уточняющие вопросы и обязательно сохранять версии кода для отката [1] [9].

Практически это означает, что лучше писать не «сделай мне приложение», а «сделай страницу логина на React с двумя полями, валидацией email и обработкой ошибки входа». Чем точнее запрос, тем качественнее результат [3] [6].

Простая формула

Если совсем коротко, вайб-кодинг — это **«сначала идея и описание, потом генерация и правка кода с помощью ИИ»** [2] [4].

Это особенно удобно, когда нужно быстро получить рабочую версию, но не стоит слепо доверять сгенерированному коду без проверки [1] [7].

Могу дальше показать:

— как вайб-кодить на практике;

— какие промпты работают лучше;

— чем вайб-кодинг отличается от классической разработки.

—

⁂

Как программировать в стиле М Фаулера с использованием вайб-кодинга — описать подробно, дать ссылки на литературу

Да — если соединить вайб-кодинг с подходом Мартина Фаулера, получится не «печатать код по наитию», а **управлять изменениями маленькими безопасными шагами**, где ИИ помогает быстро реализовать следующий шаг, а человек удерживает дизайн, тесты и эволюцию архитектуры [11] [12].

Какой здесь принцип

У Фаулера ключевая идея такая: хороший код — это код, который легко читать, менять и сопровождать; поэтому улучшение дизайна должно идти через небольшие поведенчески сохранные изменения, то есть через refactoring [11] [13].

В TDD-подходе, который он поддерживает, цикл выглядит как Red-Green-Refactor: сначала тест, затем минимальная реализация, затем улучшение структуры кода [12].

Если перенести это в вайб-кодинг, ИИ становится ускорителем цикла, а не заменой инженерного мышления [14] [12].

Рабочая модель

Практически это выглядит так:

— Сформулировать маленькую бизнес-задачу.

— Попросить ИИ написать минимальный код и тесты.

— Запустить тесты и посмотреть, что именно сломано.

— Попросить ИИ сделать один небольшой рефакторинг без изменения поведения.

— Повторять, пока код не станет чище, но все тесты продолжают проходить [12] [11].

Такой стиль особенно хорошо сочетается с идеей Фаулера о «small steps»: большие изменения разбиваются на цепочку маленьких, проверяемых преобразований [15] [11].

Вайб-кодинг здесь полезен тем, что ускоряет написание чернового кода, тестов, адаптеров и промежуточных рефакторингов [14] [13].

Как именно кодить

Лучше всего мыслить не файлами, а **микро-изменениями**.

Например: «выдели метод», «убери дублирование», «замени флаг-аргумент на полиморфизм», «вынеси доменную логику из контроллера», «добавь тест на граничный случай» [13] [16].

Полезный стиль промпта для ИИ:

— «Сначала предложи минимальное изменение, не меняя поведение».

— «Сначала напиши тест, который фиксирует текущее поведение».

— «Теперь рефакторни только этот метод».

— «Не трогай соседние модули».

— «Покажи, какие инварианты ты сохраняешь» [12] [11].

Это очень близко к фаулеровскому мышлению: сначала сделать код безопасным для изменения, затем улучшать его понемногу [17] [18].

Что брать у Фаулера

Если переводить идеи Фаулера в вайб-кодинг, то ядро такое:

— Refactoring как повседневная практика, а не отдельная «уборка» после проекта [11] [13].

— Тесты как страховка для изменений [12] [19].

— Маленькие шаги вместо больших переписываний [15] [11].

— Работа с legacy через seams, то есть через места, где поведение можно менять без прямого вмешательства в сердце системы [17] [18].

— Поиск code smells как сигналов, что пора менять структуру, а не только багфиксить симптомы [16] [13].

Как использовать ИИ

ИИ в этом стиле лучше использовать как помощника по четырем ролям:

— Генератор черновика.

— Партнер по рефакторингу.

— Генератор тестов.

— Аналитик code smells и точек декомпозиции [14] [16].

Плохой сценарий — попросить ИИ «сделай всё приложение».

Хороший сценарий — поручить один шаг: «разбей этот класс», «добавь тесты», «вынеси зависимость», «замени ветвление на стратегию», «покажи, как сделать seam для legacy-кода» [17] [13].

Практический шаблон

Для проекта можно держать такой цикл:

— Определить поведение.

— Добавить или обновить тест.

— Сгенерировать минимальный код.

— Проверить.

— Рефакторить маленькими шагами.

— Повторить [12] [11].

Пример: если нужно вынести логику расчета комиссии, сначала пишешь тесты на текущие кейсы, потом просишь ИИ выделить отдельную функцию или объект, затем постепенно убираешь дублирование и упрощаешь интерфейс [16] [13].

Именно так вайб-кодинг становится не хаотичным «кодогенератором», а инструментом аккуратной эволюции системы [11] [17].

Литература

Вот хорошие источники, с которых стоит начать:

— Martin Fowler, **Refactoring: Improving the Design of Existing Code** [11] [20].

— Martin Fowler, **Catalog of Refactorings** [13].

— Martin Fowler, **Test-Driven Development** [12].

— Martin Fowler, **The Practical Test Pyramid** [19] [21].

— Michael Feathers, **Working Effectively with Legacy Code** [18].

— Martin Fowler, **Legacy Seam** [17].

— Martin Fowler, **Refactoring to Patterns** [15].

Где читать

Официальные страницы и материалы:

— https://martinfowler.com/books/refactoring.html [11]

— https://refactoring.com/catalog/ [13]

— https://martinfowler.com/bliki/TestDrivenDevelopment.html [12]

— https://martinfowler.com/articles/practical-test-pyramid.html [19]

— https://martinfowler.com/bliki/LegacySeam.html [17]

— http://objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf [18]

Если хочешь, я могу следующим сообщением собрать для тебя **готовый пошаговый workflow «vibe coding + Fowler» для реального проекта на Python/TypeScript**.

⁂

Примеры промптов для ИИ в стиле Фаулера

Ниже — набор **практичных промптов в стиле Фаулера**: то есть не «сгенерируй всё целиком», а «сделай маленький безопасный шаг, сохрани поведение, улучши дизайн, опирайся на тесты» [40] [41].

Базовый шаблон

Используй такой каркас почти для любой задачи:

— Контекст: что это за система и где боль.

— Цель: что нужно улучшить.

— Ограничения: не менять поведение, не трогать соседние модули, сохранить API.

— Критерий успеха: тесты проходят, код стал проще, дублирование уменьшилось.

— Формат ответа: план, изменения, риски, тесты [40] [42].

Пример промпта:

У меня есть фрагмент legacy-кода. Найди самый безопасный **следующий маленький шаг** для улучшения дизайна, не меняя поведение. Сначала предложи тесты, затем минимальный рефакторинг, затем укажи, какие запахи кода ты устранил. Не переписывай всё целиком [43] [40].

Для нового кода

Если проект только начинается, Фаулерский стиль означает сразу закладывать простоту и тестируемость.

Помоги спроектировать модуль так, чтобы его было легко рефакторить потом. Предложи минимальную доменную модель, границы ответственности, тесты и точки расширения. Избегай преждевременной абстракции [40] [44].

Сгенерируй минимальную реализацию функции с TDD-подходом: сначала тест, потом код, потом один рефакторинг. Не добавляй лишних слоёв и паттернов без явной необходимости [41] [40].

Для рефакторинга

Это самый «фаулеровский» сценарий.

У меня есть метод с несколькими ответственностями. Разбей его на более маленькие части, сохранив поведение. Сделай изменения маленькими шагами и покажи, какой тест защищает каждый шаг [40] [42].

Найди code smells в этом фрагменте: длинный метод, дублирование, флаг-аргументы, смешение доменной и инфраструктурной логики. Для каждого запаха предложи один небольшой рефакторинг и объясни, почему он безопасен [45] [42].

Предложи план refactoring-by-steps: шаг 1, шаг 2, шаг 3. Каждый шаг должен быть обратимым и проверяемым тестом [46] [40].

Для legacy-кода

Фаулерский стиль особенно хорошо работает с legacy, если просить ИИ искать seams и точки изоляции.

У меня legacy-модуль без хороших тестов. Найди seam, через который можно безопасно добавить тестирование, не переписывая систему. Предложи самый дешёвый способ зафиксировать текущее поведение [43] [47].

Помоги обернуть внешний сервис адаптером, чтобы можно было протестировать бизнес-логику отдельно. Сначала опиши точку изоляции, потом предложи изменения в коде [43] [44].

Для этого старого модуля предложи стратегию постепенной модернизации: что тестировать сначала, что вынести в отдельный слой, что не трогать до появления покрытия [47] [40].

Для работы с ИИ

Здесь важно управлять моделью как напарником по инженерной дисциплине.

Не пиши полный код сразу. Сначала задай мне 3 уточняющих вопроса, потом предложи план маленьких шагов, потом сгенерируй только первый шаг [48] [41].

Когда предлагаешь изменения, обязательно указывай: какой инвариант сохраняется, какой риск появляется, какой тест это проверит [41] [40].

Если видишь несколько вариантов дизайна, сравни их по поддерживаемости, тестируемости и стоимости изменения. Не выбирай самый «умный» вариант без необходимости [40] [42].

Для TDD

Это очень полезный стиль промптинга.

Напиши тесты, которые описывают поведение системы на уровне намерений, а не реализации. Затем предложи минимальную реализацию, достаточную для прохождения тестов [41] [44].

Сначала сформулируй failing test для граничного случая, затем покажи минимальный production-code fix, затем предложи один рефакторинг без изменения поведения [41] [40].

Проверь, есть ли в этом коде тестируемые границы. Если их нет, предложи, как ввести их с минимальными изменениями [44] [43].

Для архитектуры

Фаулер бы одобрил вопросы про границы, слои и простоту изменений.

Помоги разделить систему на слои так, чтобы доменная логика не зависела от UI и внешних API. Предложи минимальные интерфейсы и места для dependency injection [40] [44].

Предложи архитектуру, которая позволит менять одну подсистему без каскадных правок. Какие seams и абстракции нужны прямо сейчас, а какие преждевременны [43] [40].

Хорошие ограничения

Вот фразы, которые сильно улучшают результат:

— «Не меняй поведение».

— «Ограничься одним файлом или одним методом».

— «Не добавляй новые библиотеки».

— «Сначала тест, потом код».

— «Сделай минимальный шаг».

— «Покажи, как откатить изменение».

— «Не оптимизируй заранее».

— «Не переписывай целиком» [41] [40].

Плохие формулировки

Избегай таких запросов:

— «Сделай красиво».

— «Полностью перепиши архитектуру».

— «Добавь лучшие практики».

— «Сделай enterprise-grade» [40] [45].

Проблема в том, что они слишком расплывчаты и толкают модель к избыточному дизайну или большим изменениям без проверки [42] [40].

Мини-шаблон для копипаста

У меня есть фрагмент кода.

Цель: [что улучшить].

Ограничения: не менять поведение, не трогать соседние модули, не добавлять новые зависимости.

Сначала предложи тесты, которые зафиксируют текущее поведение.

Затем предложи только один маленький рефакторинг.

Объясни, какой code smell он убирает и почему изменение безопасно [41] [40] [42].

Литература

Для этой манеры мышления особенно полезны:

— Martin Fowler, **Refactoring: Improving the Design of Existing Code** [40].

— Martin Fowler, **Catalog of Refactorings** [42].

— Martin Fowler, **Test-Driven Development** [41].

— Martin Fowler, **The Practical Test Pyramid** [44].

— Michael Feathers, **Working Effectively with Legacy Code** [47].

— Martin Fowler, **Legacy Seam** [43].

Могу следующим сообщением дать **20 готовых промптов именно под Python, TypeScript или Java/Spring**.

⁂

Как использовать эти промпты для рефакторинга legacy кода

Использовать эти промпты для рефакторинга legacy-кода лучше как **последовательный конвейер**, а не как один большой запрос: сначала понять систему, потом зафиксировать поведение тестами, затем делать маленькие безопасные изменения [58] [59] [60].

Рабочий порядок

— Сначала дай ИИ фрагмент legacy-кода и попроси провести аудит без изменений: найти роли модулей, зависимости, code smells, рискованные места и точки для seam [60] [59].

— Затем попроси написать или предложить тесты, которые фиксируют текущее поведение, включая странные edge-cases, чтобы рефакторинг не сломал неочевидную логику [61] [60].

— После этого используй промпты на **один маленький шаг**: выделение метода, удаление дублирования, инкапсуляция зависимости, упрощение ветвления, разделение ответственности [58] [62].

— В конце попроси модель показать, что изменилось, какие риски остались и как откатить правку через git [60] [63].

Как строить промпт

Хороший промпт для legacy-кода должен содержать четыре блока: контекст, цель, ограничения и формат ответа [60] [58].

Например: «Вот модуль заказа. Он работает, но плохо читается. Не меняй поведение. Сначала предложи тесты на текущее поведение, потом один безопасный рефакторинг, потом объясни, какой smell он убрал» [60] [64].

Чем жестче ограничение на поведение, тем меньше шанс, что ИИ начнет «улучшать» логику и ломать бизнес-кейс [60] [61].

Практический цикл

Удобно работать по циклу:

— Аудит.

— Тесты на текущее поведение.

— Один рефакторинг.

— Прогон тестов.

— Следующий маленький шаг [60] [58].

Такой цикл очень близок к фаулеровскому стилю: изменения делаются маленькими порциями, а безопасность обеспечивает тестовая сетка [58] [61].

Если модуль сильно связанный, сначала попроси ИИ найти seam — место, где можно отделить внешние зависимости от доменной логики [59] [63].

Примеры промптов

1. Аудит legacy без правок

Проанализируй этот код без изменения. Объясни, что он делает, где у него зависимости, какие есть code smells, какие места наиболее рискованные для рефакторинга и где можно поставить seam для тестов [60] [59].

2. Зафиксировать поведение

Сгенерируй набор тестов, которые фиксируют текущее поведение этого модуля, включая граничные случаи и странные edge-cases. Не исправляй код, только опиши и покажи тесты [61] [60].

3. Один шаг рефакторинга

Сделай только один безопасный рефакторинг этого метода. Поведение не должно измениться. Сначала покажи план шага, затем код, затем перечисли, какой smell устранен [58] [62].

4. Работа с зависимостями

Вынеси внешнюю зависимость за интерфейс или адаптер, чтобы доменная логика стала тестируемой отдельно. Не меняй публичное API [59] [63].

5. Стратегия миграции

Предложи пошаговый план модернизации этого legacy-модуля: что тестировать первым, что изолировать, что оставить без изменений до появления покрытия [60] [63].

Что просить у модели

Для legacy особенно полезно просить не только код, но и **обоснование**.

Пусть ИИ отвечает в формате: «что изменил, почему это безопасно, что осталось неизменным, какой тест это подтверждает» [61] [58].

Если модуль большой, проси работать по одному файлу или одному методу за раз, иначе модель почти неизбежно сделает слишком крупную правку [60] [62].

Полезная тактика

Для удобства можно задавать роли:

— «Ты — ревьюер legacy-кода.»

— «Ты — специалист по безопасному рефакторингу.»

— «Ты — помощник по TDD.»

— «Ты — эксперт по seams и адаптерам» [59] [61].

Это помогает модели держать дисциплину: не переписывать лишнее, не предлагать преждевременную архитектуру и не ломать работающий код [58] [60].

Литература и опора

Для этой работы лучше всего опираться на:

— Martin Fowler, **Refactoring: Improving the Design of Existing Code** [58].

— Martin Fowler, **Catalog of Refactorings** [62].

— Martin Fowler, **Test-Driven Development** [61].

— Martin Fowler, **Legacy Seam** [59].

— Michael Feathers, **Working Effectively with Legacy Code** [63].

Если хочешь, я могу следующим сообщением дать **готовый промпт-шаблон для рефакторинга legacy-кода**, который можно просто вставлять в ChatGPT/Claude/Cursor.

⁂

Примеры рефакторинга FastAPI legacy кода с этими промптами

Ниже — несколько **типовых примеров рефакторинга legacy FastAPI-кода «в стиле Фаулера»**: маленькими шагами, с фиксацией поведения тестами, с выделением seams и вынесением бизнес-логики из fat endpoints [74] [75] [76].

Идея для FastAPI

В legacy FastAPI часто встречаются три проблемы: толстые route handlers, смешение HTTP-слоя и бизнес-логики, а также прямое использование БД и внешних сервисов прямо в endpoint-функции [76] [77].

Фаулеровский подход здесь — не переписывать всё сразу, а сначала зафиксировать текущее поведение, затем выделить seam, потом вынести логику в service/repository-слои и сохранить публичное API [74] [75].

Пример 1: fat endpoint
До

```
@router.post (»/orders/ {order_id} /approve»)
async def approve_order (
order_id: int,
db: Session = Depends (get_db),
user: User = Depends (get_current_user),
):
order = db.query(Order).filter(Order.id == order_id).first ()
if not order:
raise HTTPException (status_code=404, detail=«Order not found»)


if order.status!= «pending»:
raise HTTPException (status_code=400, detail=«Invalid status»)


if user.role!= «manager»:
raise HTTPException (status_code=403, detail=«Forbidden»)


order.status = «approved»
order. approved_by = user.id
db.commit ()
db.refresh (order)


send_email(order.customer_email, «approved»)
audit_log («order_approved», order.id, user.id)


return {«id»: order.id, «status»: order.status}
```

В этом коде HTTP-логика, авторизация, бизнес-правила, работа с БД и побочные эффекты смешаны в одном месте, что затрудняет тестирование и изменение поведения [76] [78].

Промпт к ИИ

Проанализируй этот FastAPI endpoint. Не меняй поведение.
После первого шага

```
class OrderService:
def __init__ (self, db: Session):
self. db = db


def approve (self, order_id: int, user: User):
order = self.db.query(Order).filter(Order.id == order_id).first ()
if not order:
raise OrderNotFound ()
if order.status!= «pending»:
raise InvalidOrderStatus ()
if user.role!= «manager»:
raise PermissionDenied ()


order.status = «approved»
order. approved_by = user.id
self.db.commit ()
self.db.refresh (order)
return order


def get_order_service (db: Session = Depends (get_db)) ->
OrderService:
return OrderService (db)


@router.post (»/orders/ {order_id} /approve»)
async def approve_order (
order_id: int,
user: User = Depends (get_current_user),
service: OrderService = Depends (get_order_service),
):
order = service. approve (order_id, user)
send_email(order.customer_email, «approved»)
audit_log («order_approved», order.id, user.id)
return {«id»: order.id, «status»: order.status}
```

Это уже лучше, потому что endpoint начинает работать как контроллер, а бизнес-решение перемещается в service-слой, который легче покрывать тестами отдельно от HTTP [76] [77].

По-Фаулеру это хороший первый шаг: небольшой, локальный и поведенчески безопасный [75] [79].

Пример 2: выделение seam
Проблема

Частая беда legacy FastAPI — прямые вызовы email, Kafka, Redis, внешних API или audit-функций внутри бизнес-метода. Такие зависимости мешают изолированному тестированию и делают код хрупким [74] [80].

Промпт к ИИ

Найди в этом FastAPI service внешние зависимости, мешающие тестированию.

Предложи seam для изоляции email и audit-логирования.

Сделай один маленький шаг: заверни побочные эффекты в интерфейс NotificationPort и AuditPort, не меняя поведения сервиса [74] [81].

После

```
class NotificationPort (Protocol):
def order_approved (self, email: str) -> None:…


class AuditPort (Protocol):
def order_approved (self, order_id: int, user_id: int) -> None:…


class OrderService:
def __init__ (
self,
db: Session,
notifications: NotificationPort,
audit: AuditPort,
):
self. db = db
self.notifications = notifications
self.audit = audit


def approve (self, order_id: int, user: User):
order = self.db.query(Order).filter(Order.id == order_id).first ()
if not order:
raise OrderNotFound ()
if order.status!= «pending»:
raise InvalidOrderStatus ()
if user.role!= «manager»:
raise PermissionDenied ()


order.status = «approved»
order. approved_by = user.id
self.db.commit ()
self.db.refresh (order)


self.notifications.order_approved(order.customer_email)
self.audit.order_approved(order.id, user.id)
return order
```

Теперь появился seam: в тестах можно подставить fake-реализации портов и проверить только бизнес-поведение без реальных внешних эффектов [74] [80].

Это очень фаулеровский ход: не спорить с legacy-кодом в лоб, а сначала открыть место для безопасной подмены зависимостей [74] [75].

Пример 3: слишком много Depends

В реальных FastAPI legacy-проектах endpoint может принимать слишком много зависимостей, а затем просто прокидывать их дальше. Это признак того, что композиция зависимостей находится не в том слое [82] [76].

До

```
@router.get (»/ {order_id}»)
async def get_order (
order_id: int,
session: Session = Depends (get_db),
actor_service: ActorService = Depends (get_actor_service),
event_service: EventService = Depends (get_event_service),
item_service: ItemService = Depends (get_item_service),
comment_service: CommentService = Depends (get_comment_service),
):
service = OrderService (
session=session,
actor_service=actor_service,
event_service=event_service,
item_service=item_service,
comment_service=comment_service,
)
return service.get_order (order_id)
```

Промпт к ИИ

Упрости FastAPI endpoint в legacy-модуле.

Не меняй поведение.

Вынеси сборку сложного OrderService в dependency provider, чтобы endpoint зависел только от одного сервиса [82] [76] [77].

После

```
def get_order_service (
session: Session = Depends (get_db),
actor_service: ActorService = Depends (get_actor_service),
event_service: EventService = Depends (get_event_service),
item_service: ItemService = Depends (get_item_service),
comment_service: CommentService = Depends
(get_comment_service),
) -> OrderService:
return OrderService (
session=session,
actor_service=actor_service,
event_service=event_service,
item_service=item_service,
comment_service=comment_service,
)


@router.get (»/ {order_id}»)
async def get_order (
order_id: int,
service: OrderService = Depends (get_order_service),
):
return service.get_order (order_id)
```

Такой шаг уменьшает связность HTTP-слоя и делает endpoint чище без большого переписывания архитектуры [76] [77].

Это не «идеальная архитектура за раз», а маленькое улучшение формы кода, что хорошо соответствует каталогу рефакторингов Фаулера [79] [75].

Пример 4: постепенное покрытие тестами

Когда legacy FastAPI-код плохо тестируется, не надо сразу пытаться писать идеальные unit-тесты на всё. Сначала имеет смысл зафиксировать текущее поведение на уровне API или service-метода, а уже потом дробить систему дальше [81] [83].

Промпт к ИИ

Напиши characterization tests для этого FastAPI endpoint.

Цель — зафиксировать текущее поведение, включая странные ветки и edge-cases.

Не исправляй баги, только опиши и покрой текущее поведение тестами через TestClient или через сервисный слой, если это дешевле [81] [83].

Это полезно, потому что characterization tests снижают риск «рефакторинга вслепую», особенно если бизнес-логика уже не до конца понятна команде [81] [75].

Готовые промпты

Вот хорошие промпты именно для FastAPI legacy:

— «Проанализируй этот endpoint и выдели HTTP-логику, бизнес-логику, доступ к данным и побочные эффекты. Предложи один минимальный рефакторинг без изменения поведения» [75] [76].

— «Сгенерируй characterization tests для этого FastAPI route, чтобы зафиксировать текущее поведение перед рефакторингом» [81] [83].

— «Найди seam для изоляции внешнего API/почты/очереди, чтобы сервис можно было тестировать без реальных интеграций» [74] [80].

— «Вынеси бизнес-логику из fat endpoint в service-слой, сохрани сигнатуру маршрута и response shape» [76] [77].

— «Упрости dependency graph: endpoint должен зависеть от одного сервиса, а не от пяти отдельных зависимостей» [82] [76].

— «Предложи пошаговый план рефакторинга этого FastAPI-модуля в 3—5 маленьких шагов, каждый с тестом и возможностью отката» [84] [75].

Как применять на практике

Самый рабочий способ такой:

— Вставляешь один endpoint или один service-класс.

— Просишь ИИ сначала объяснить текущую структуру.

— Затем просишь тесты на текущее поведение.

— Потом — один рефакторинг за раз.

— После каждого шага запускаешь тесты и только потом переходишь дальше [75] [83].

Если хочешь, я могу следующим сообщением сделать **полный разбор одного реального FastAPI legacy endpoint: до → промпты → после → тесты**.

⁂

Пример рефакторинга FastAPI на чистую архитектуру в стиле lobeHub

Да: в стиле LobeHub это обычно означает **разнести FastAPI-приложение по слоям API → Application/Use Cases → Domain → Infrastructure**, чтобы endpoint был тонким, бизнес-правила жили вне FastAPI, а зависимости смотрели внутрь, а не наружу [89] [90] [91].

Что значит «в стиле LobeHub»

В описании skill fastapi-clean-architecture на LobeHub акцент сделан на четком разделении слоев, dependency injection, repository pattern и тестируемости, где API-слой отвечает за маршруты и валидацию, Domain — за сущности и правила, а Infrastructure — за БД и внешние адаптеры [89] [90].

Это хорошо сочетается с фаулеровским подходом: не переписывать всё в один заход, а постепенно выделять use cases, порты и адаптеры, сохраняя поведение системы [92] [93].

Типичный legacy FastAPI

Обычно legacy FastAPI-модуль выглядит так: route сам читает данные из SQLAlchemy, сам проверяет права, сам реализует бизнес-логику и сам вызывает внешние сервисы. Такой код трудно тестировать и он жестко связан с FastAPI и ORM [94] [91].

Упрощенный пример «до»:

```
@router.post (»/users/ {user_id} /deactivate»)
def deactivate_user (
user_id: int,
db: Session = Depends (get_db),
current_user: User = Depends (get_current_user),
):
user = db.query(UserModel).filter(UserModel.id == user_id).first ()
if not user:
raise HTTPException (404, «User not found»)
if current_user.role!= «admin»:
raise HTTPException (403, «Forbidden»)
if user.status == «inactive»:
raise HTTPException (400, «Already inactive»)

user.status = «inactive»
db.commit ()
send_email (user. email, «Your account was deactivated»)
return {«id»: user.id, «status»: user.status}
```

Здесь смешаны presentation, application logic, persistence и integration side effects, то есть почти все слои одновременно [94] [91].

Куда двигаться

Для clean architecture в стиле LobeHub целевая структура обычно такая:

Слой	Что хранит
API	FastAPI routers, request/response DTO, auth wiring [89][91]
Application	Use cases, orchestration, транзакционные сценарии [95][96]
Domain	Entities, value objects, business rules, repository interfaces [89][90]
Infrastructure	SQLAlchemy repositories, email adapters, Redis, external APIs [89][91]

Ключевой принцип: бизнес-правила не должны зависеть от FastAPI, а репозитории и адаптеры реализуют абстракции, определенные ближе к домену или application-слою [89] [91].

Пошаговый рефакторинг

Начинать лучше не с «перестроим весь проект», а с одного вертикального среза — одного endpoint и связанных с ним правил [92] [97].

Практически последовательность такая:

— Зафиксировать текущее поведение тестами.

— Вынести бизнес-операцию в use case.

— Вынести доступ к данным в repository interface + implementation.

— Изолировать внешние вызовы через порт.

— Оставить router только как адаптер HTTP ↔ use case [98] [89] [91].

Пример «после»

Domain

```
from dataclasses import dataclass
from typing import Protocol, Optional


@dataclass
class User:
id: int
email: str
status: str


class UserRepository (Protocol):
def get_by_id (self, user_id: int) -> Optional [User]:…
def save (self, user: User) -> None:…


class NotificationPort (Protocol):
def send_deactivated (self, email: str) -> None:…
```

В domain остаются сущность и абстракции, без FastAPI, SQLAlchemy и внешних SDK [89] [90].

Application

```
class UserNotFoundError (Exception): pass
class ForbiddenError (Exception): pass
class AlreadyInactiveError (Exception): pass


class DeactivateUserUseCase:
def __init__ (self, users: UserRepository, notifications:
NotificationPort):
self.users = users
self.notifications = notifications
```

```
def execute (self, actor_role: str, user_id: int) -> User:
user = self.users.get_by_id (user_id)
if not user:
raise UserNotFoundError ()


if actor_role!= «admin»:
raise ForbiddenError ()
```

Конец ознакомительного фрагмента.