ChatGPT

Эффективное использование Codex при разработке full-stack TypeScript приложений

Введение. Интеллектуальные помощники для программирования – такие как OpenAI Codex (основа GitHub Copilot) – стремительно меняют процесс разработки. Особенно заметно их влияние при создании сложных full-stack TypeScript приложений. В этом практическом исследовании мы разберём, как инженер может эффективно вовлечь Codex (через GitHub Copilot, OpenAI API и аналогичные ИИ-ассистенты) в работу над проектом – например, над многопользовательским редактором à la Notion с AI-ассистентом, учитывающим контекст рабочей области (workspace-aware). Мы рассмотрим, как грамотно взаимодействовать с Codex на всех этапах разработки, формулировать для него задачи, описывать архитектуру и компоненты, использовать для рефакторинга и документирования, соблюдать при этом контроль над архитектурой, а также обсудим выбор технологий (backend-стек, база данных) и подходящего AI-инструмента. Стиль изложения – сугубо практический, ориентированный на реальные инженерные задачи.

Взаимодействие с Codex в full-stack TypeScript-проекте

Роль Codex как «парного программиста». GitHub Copilot (работающий на модели Codex) можно представить как виртуального напарника-программиста, помогающего писать код быстрее и с меньшими усилиями [1]. Однако чтобы эта помощь была эффективной, разработчик должен оставаться «ведущим» в паре: **Codex не заменяет опыт и навыки инженера** и не принимает архитектурных решений – за это по-прежнему отвечает человек [2]. Codex отлично справляется с рутинными задачами: автодополнение кода, генерация шаблонных фрагментов, типичных функций, тестов и т.п. [3]. Например, он способен ускорить написание как фронтенд-, так и бэкенд-кода, предложить готовые вызовы API, сгенерировать повторяющиеся конструкции и тесты [4]. Всё это позволяет сконцентрироваться на решении архитектурных и продуктовых проблем, делегируя часть реализации помощнику.

Рабочий процесс с ИИ-ассистентом. Практика показывает, что оптимальный способ использования Codex – это **итеративная пошаговая разработка**. На старте проекта стоит набросать общую архитектуру и структуру (например, определить, что у нас будет frontend на React + TipTap, backend на NestJS с WebSocket для реального времени, БД PostgreSQL и т.д.). Затем работу можно разбить на небольшие, чётко очерченные задачи и решать их последовательно, привлекая Codex на каждом шаге. Обычно цикл выглядит так:

1. **Постановка задачи.** Прежде чем писать код, разработчик формулирует конкретную задачу: например, *“необходимо реализовать REST-эндпойнт* `/login` *для аутентификации пользователя с использованием JWT”* или *“нужно добавить в текстовый редактор кнопку выделения текста жирным шрифтом”*. Важно чётко представлять, **какой результат нужен и какими средствами его лучше достичь**, иначе Codex может увести не туда.
2. **Выбор подходящего инструмента Codex.** Если задача локальная (написать функцию, дополнить текущий файл кодом) – удобнее Copilot автодополнение. Если требуется обсудить архитектуру или сгенерировать большой фрагмент кода с пояснениями – полезен Copilot Chat или внешний чат с GPT. Для интеграции ИИ в собственные инструменты можно использовать OpenAI Codex API. Инструменты разные, и важно

использовать каждый по назначению [5] [6]. Например, Copilot внутри IDE отлично дополняет код по контексту, а Copilot Chat способен ответить на вопросы по вашему коду и сгенерировать более крупные куски, которые затем дорабатываются [7].

3. **Предоставление контекста Codex.** Codex работает статистически и **очень зависит от предоставленного контекста** [8]. Поэтому перед генерацией кода стоит подготовить для него почву: открыть связанные файлы, описать в коде или комментариях, что должно быть сделано, указать сигнатуры функций, типы данных, и т.д. Мы подробно разберём это в следующем разделе.
4. **Генерация и правка кода.** Начинаем реализовывать функцию или компонент – чаще всего с комментария или с объявления функции – и смотрим, что предлагает Codex. При корректном контексте он часто генерирует готовое решение или приближенную заготовку. Код **нужно внимательно проверить** и при необходимости поправить вручную или с помощью повторных запросов к Codex. Ни в коем случае нельзя слепо доверять сгенерированному коду без валидации [9].
5. **Тестирование и отладка.** После интеграции полученного кода в проект запускаем тесты (или пишем их, возможно, тоже с помощью Codex) и проверяем, работает ли всё как задумано. Codex не гарантирует безошибочность, поэтому цикл “сгенерировал – протестировал – исправил” обязателен.
6. **Рефакторинг и документирование.** По мере роста проекта Codex пригодится для улучшения кода и написания документации (об этом также подробнее далее). Он может помочь найти более элегантное решение, объяснить сложный участок кода или подготовить комментарии и документацию по написанному.

Такая итеративная схема позволяет использовать сильные стороны AI-ассистента (скорость, широта знаний, генерация шаблонного кода) и нивелировать его слабости (недопонимание контекста, склонность к ошибкам без контроля). Ключевой принцип: **разработчик остаётся ведущим**, Codex – инструмент, хоть и «умный».

Формулирование запросов для получения точного результата

Эффективность Codex напрямую зависит от того, **насколько понятно вы ставите перед ним задачу**. Принципы здесь похожи на постановку задачи для джуна-разработчика: нужно четко объяснить, что требуется, иначе результат будет неопределённым. GitHub называет это *“prompt engineering”*, и рекомендует несколько правил для хороших промптов [10]:

- **Дробите сложные задачи на части.** Codex (особенно в виде Copilot) не предназначен для разом «написать весь приложение» [11]. Он лучше справляется с небольшими, изолированными шагами. Поэтому вместо запроса *“Сгенерируй мне многопользовательский редактор как в Notion”*, сформулируйте конкретный подзадачу: например, *“Реализуй класс NodeRepository с методами для сохранения и загрузки узлов документа в PostgreSQL”*. Решив одну задачу, перейдите к следующей. Такой подход дает более релевантный код и снижает риск ошибок [11] [12].
- **Будьте специфичны в требованиях.** Чем точнее описано, что вы ожидаете от кода, тем выше шанс получить полезный фрагмент. Избегайте расплывчатых фраз вроде “сделай красиво” или “оптимизируй всё” – вместо этого укажите конкретно: *“функция должна принимать email и пароль, проверять соответствие с хэшом в БД и возвращать JWT при успехе”*. Также полезно указать ограничения: например, *“не используй внешних библиотек кроме уже установленных”* или *“следуй стилю кода, принятому в проекте”*. Copilot лучше откликается на четкие указания и примеры входных/выходных данных [10] [13].

- **Описывайте контекст и пример использования.** Полезный приём – прежде чем просить сгенерировать код, показать Codex *пример*, как этот код будет использоваться. В TypeScript это может быть обозначение интерфейсов, типов или даже написанный заранее вызов функции. Например, чтобы сгенерировать реализацию функции `formatDate()`, можно сначала написать, как она должна работать:

  ```
  // Возвращает дату в формате ДД.ММ.ГГГГ
  formatDate(new Date('2025-11-11')) // ожидается "11.11.2025"
  ```

  После этого Codex с большой вероятностью предложит правильную реализацию, опираясь на пример вывода. Примеры входных и выходных данных существенно улучшают качество ответа [10]. Аналогично, при генерации UI-компонента можно сначала набросать JSX-разметку использования этого компонента или описать в комментарии сценарий работы.
- **Используйте понятные комментарии и названия.** Комментарии на естественном языке – мощный способ направить Codex. Если вы напишете перед функцией комментарий `// Хэширует пароль с помощью bcrypt и возвращает соль и хэш`, Copilot часто сразу предложит код с использованием bcrypt, даже может подключить нужный импорт. **Именование** тоже служит подсказкой: функция с названием `generateDocumentTree()` вероятнее получит правильную реализацию, чем `func1()`. Хорошие, говорящие названия переменных и функций выступают своего рода встроенными подсказками [14] [15]. Codex обучен на массе открытого кода, где имена следуют смыслу, поэтому он часто продолжает логику, исходя из названий.
- **Обеспечьте достаточный объем видимого контекста.** Copilot видит не весь ваш проект, а в основном содержимое текущего файла + ограниченное окно вокруг курсора, возможно связанные файлы, открытые в редакторе [16]. Поэтому прежде чем запрашивать код, **откройте в IDE все связанные файлы** – например, модель данных, интерфейсы API, константы. Если пишете функцию, которая сохраняет данные в БД, откройте файл с описанием схемы или ORM-моделей – тогда Copilot сможет учесть их структуры при генерации запросов. Не лишним будет в начале файла или в комментарии кратко описать, какой модуль вы пишете и какова его роль. Если контекста не хватает или он устарел – Codex может «галлюцинировать» неуместный код [17]. Не стесняйтесь перефразировать и повторять свои запросы: если предложение Copilot не подходит, лучше переписать комментарий более ясно или разбить задачу, и попробовать снова [18] [12].

Пример постановки задачи: допустим, нужно быстро написать на Express middleware для проверки JWT-токена. Плохой вариант запроса: `// проверка JWT`. Хороший вариант:

```
// Middleware: проверяет JWT из заголовка Authorization.
// Если токен валиден, добавляет информацию о пользователе в req.user и
вызывает next().
// Если не валиден – возвращает 401 ошибку.
```

Такое описание практически гарантированно приведёт к развёрнутому коду с проверкой подписи JWT (например, через jsonwebtoken) и нужной логикой, потому что Codex «понимает», что от него требуется, вплоть до деталей вроде кода ошибки.

Пример: хорошо сформулированный prompt. На изображении показано, как простой комментарий (описание желаемой функциональности) вместе с началом сигнатуры функции дали Copilot

достаточный контекст, чтобы предложить осмысленный код (серым цветом) для запуска веб-сервера с нужным маршрутом [19]. Использование понятного комментария `// start a web server with the Echo library` и объявление функции `func startServer()` сразу подсказало ИИ, какой код вписать внутрь. Этот пример иллюстрирует принцип: правильно сформулированное текстовое описание задачи и чётко названная функция позволяют Codex предложить релевантный фрагмент кода, экономя вам время.

Наконец, стоит помнить: **не задавайте Codex того, чего сами не знаете**. Если вы очень смутно представляете решение (например, реализацию сложного алгоритма или работу новой библиотеки), велик риск, что и Codex выдаст что-то неверное, а вы это не распознаете [12]. AI-ассистент лучше всего работает под контролем опытного разработчика: когда вы можете оценить адекватность ответа и скорректировать его. Воспринимайте Codex как ускоритель для ваших идей, а не как оракул, способный заменить понимание задачи.

Описание архитектуры, компонентов и интерфейсов для Codex

Чтобы Codex писал код, соответствующий вашей архитектуре и дизайну системы, **ему нужно явно или неявно объяснить архитектурные границы и соглашения** вашего проекта. В отличие от человека-разработчика, ИИ не читает вашу мысль и не видит диаграммы – он делает выводы только из текстового контекста кода. Поэтому важно встроить архитектурные подсказки прямо в процесс кодирования:

- **Опишите слои и роли через код.** Если у вас принята многоуровневая архитектура (например, контроллеры – сервисы – репозитории), постарайтесь отразить это в коде, который видит Copilot. Например, создайте и откройте пустые классы `UserController`, `UserService` и `UserRepository` с базовым скелетом (методы объявлены, но не реализованы). Тогда, когда вы попросите Codex реализовать метод `createUser` в `UserService`, он уже будет понимать, что контроллер должен вызывать сервис, а сервис – репозиторий, и сгенерирует код соответственно. **Codex улавливает паттерны**, поэтому если в проекте уже есть пара готовых модулей, он по аналогии будет писать новые. Например, видя оформленный по шаблону NestJS-модуль, Copilot может повторить ту же структуру для другого домена.
- **Пропишите сигнатуры функций и типы данных заранее.** TypeScript здесь ваш союзник. Если вы сначала определите интерфейсы и типы, описывающие данные и границы модулей, то генерация реализаций пройдёт гораздо точнее. Допустим, вы описали тип `DocumentTree` и интерфейс репозитория `DocumentRepository` с методами `saveTree(tree: DocumentTree): Promise<void>` и `loadTree(id: string): Promise<DocumentTree>`. Когда затем вы попросите Codex реализовать эти методы, он уже будет знать, что оперировать нужно с `DocumentTree` и, вероятно, сохранение должно идти, скажем, в базу (по контексту импорта, если подключён драйвер БД). **Чем полнее типизация и явнее границы – тем меньше простор для ошибок у Codex.** Он стремится выдать код, который компилируется и соответствует объявлениям.
- **Используйте комментарии-договорённости.** Иногда полезно прямо в комментариях указать архитектурные ограничения. Например:

```
// Контроллер не должен напрямую обращаться к базе данных – только через Service.
```

```
// Следуя архитектуре, этот метод должен лишь вызывать userService и
возвращать DTO.
```

Если вставить такой комментарий перед началом метода контроллера и затем попросить сгенерировать тело метода, Copilot скорее всего подчинится описанию (или хотя бы попытается). Он может сам подтянуть нужный вызов `userService.createUser(...)` вместо, допустим, прямой работы с ORM. Конечно, гарантировать послушание ИИ сложно, но чёткие текстовые указания снижают вероятность получить «неправильный» архитектурно код. Аналогично можно указать желаемые соглашения по именованию: *"// все переменные должны использовать camelCase, функции в стиле verbNoun"*, или по структуре файлов: *"// код ниже расположен в модуле auth, не следует обращаться к сущностям других модулей"*.

- **Поддерживайте единый стиль и шаблоны.** Если ваш проект придерживается определённого паттерна (например, Repository pattern, MVC, Clean Architecture), убедитесь, что первые реализованные вами модули чётко отражают этот паттерн. Тогда Codex, обученный на тысячах проектов, будет «видеть» знакомую структуру и интуитивно ей следовать. Например, NestJS-проекты имеют ярко выраженный стиль (декораторы `@Controller`, `@Injectable`, методы с декораторами запросов и т.д.). Если в вашем коде это уже есть, Copilot, дополняя новый код, тоже расставит нужные декораторы и вызовы, вместо того чтобы придумывать свою структуру [20] [21]. В случае с tRPC – если вы показали пример создания router'а, Codex попытается добавить новые процедуры аналогично.
- **Осознавайте ограничения: многофайровую логику Codex не «видит».** Крупные архитектурные изменения (например, переставить местами слои, переименовать глобально сервисы) Codex сделать не сможет автоматически [8]. Он не анализирует проект целиком и не умеет выполнять рефакторинг во всех файлах сразу. Поэтому архитектурные перестройки выполняются вами вручную или с помощью специализированных инструментов. Зато **локально** внутри одного файла Copilot может ускорить перестройку кода: к примеру, вы решаете вынести часть логики из контроллера в сервис – можно вырезать этот код, вставить в новый файл сервиса и попросить ИИ «отрефакторить метод под новый класс», подправив переменные и зависимости. Если объём небольшой, он справится.

Таким образом, чем больше **структурированной информации** о вашем проекте вы дадите Codex, тем точнее он впишется в ваш дизайн. Архитектуру надо доносить не только документами для людей, но и явными маркерами в коде: типы, имена, комментарии. В пределе, можно даже написать мини-спецификацию: например, сделать файл `ARCHITECTURE.md` с описанием модулей и соглашений и держать его открытым рядом с кодом – Copilot, конечно, не гарантированно будет его читать, но если скопировать оттуда пару строк в комментарий, это уже контекст. Ещё лучше – использовать возможности кастомизации Copilot: недавно появилась функция «Custom instructions», где можно задать ИИ персональные предпочтения (например: "Я пишу на NestJS, придерживаюсь DDD, не использую any-тип, от тебя жду кода в таком же стиле"). Эти инструкции Copilot будет учитывать глобально. Если такой функционал недоступен, те же указания можно кратко прописывать в комментариях при начале новой сессии работы.

Применение Codex для рефакторинга и понимания существующего кода

AI-ассистент полезен не только при генерации нового кода, но и при **чтении, анализе и улучшении** уже написанного. В большом TypeScript-проекте наступает момент, когда часть

модулей уже реализована, и нужно либо оптимизировать их, либо просто понять, как всё работает (например, если код писал не вы, а коллега… или сам Codex неделю назад).

Используйте Copilot для пояснения кода. Иногда удобнее спросить у ИИ, что делает тот или иной фрагмент, чем разбираться самостоятельно, особенно если код сложный или плохо документирован. GitHub Copilot Chat способен отвечать на вопросы о коде на естественном языке [6]. Выделите кусок кода (или просто поместите курсор рядом) и задайте вопрос в чате: *"Объясни, что делает эта функция."* В ответ вы получите развернутое пояснение алгоритма, значений параметров и логики, которую он нашёл в коде. Это можно использовать для быстрого понимания чужого или давно не тронутого участка. Даже обычный Copilot (автодополнение) можно заставить объяснять – например, вставив строчку комментария `// Explanation:` и нажав Tab, он часто допишет разъяснение, основанное на имени функции и её тексте. Однако чат-режим или прямое обращение к OpenAI API (модели GPT-4) даёт более надёжный результат. Конечно, критически важно перепроверять такие объяснения – модель может что-то упустить или, наоборот, нафантазировать несуществующую цель кода. Но в целом это экономит время: **Codex действует как консультант, готовый 24/7 пояснять любой фрагмент вашего проекта**.

Рефакторинг с подсказками AI. Предположим, у вас есть рабочий код, но вы хотите улучшить его качество – сделать читабельнее, быстрее или лучше структурировать. Codex может выступать в роли советчика-рефакторинг-инженера. Несколько практических подходов:

- *Встроенный рефакторинг через комментарии.* Добавьте перед функцией комментарий, описывающий желаемое улучшение. Например:

  ```
  // Оптимизируй эту функцию, избегая лишних проходов по массиву
  function calculateStats(data: number[]) { ... }
  ```

  Затем попросите Copilot сгенерировать новую версию – либо нажав Tab после комментария, либо скопировав код функции и запросив в чат *"Refactor this function to be more efficient"*. Модель попытается переписать код, учитывая ваши указания. Аналогично можно попросить *"разбить функцию на несколько, каждая выполняет одну задачу (Single Responsibility)"* или *"переименовать переменные на более понятные"*. Конечно, не всегда Codex угадает точно ваше видение, но это хороший старт для улучшенной версии. Он, например, может устранить дублирование кода или заменить устаревший API на новый, если распознает такой случай.
- *Поиск уязвимостей и багов.* Хотя ИИ не заменит полноценное тестирование, иногда можно прямо спросить: *"Есть ли ошибки в этой реализации?"* или *"Нет ли тут проблем с конкурентностью/безопасностью?"*. Модель, обученная на сотнях тысяч репозиториев, может узнать типичные ошибки. Скажем, вы скармливаете Codex функцию загрузки файла – он может предупредить, что вы не закрываете файл или не обрабатываете исключения. Такие подсказки стоит воспринимать осторожно, но в комплексе с другими методами (линтеры, код-ревью) они добавляют ценность.
- *Автоматизация простых перестроек.* Инструменты вроде Copilot хорошо справляются с **однотипными правками**. Например, вы решили поменять библиотеку запросов с fetch на Axios или заменить все callback-функции на async/await. Можно показать Codex один пример преобразования и затем поручить аналогичное на других участках: *"Apply the above change to the rest of the code"*. Однако учтите, что Copilot **не умеет** сам проходиться по проекту – вы должны открывать файлы или фрагменты кода. Для многофайлового рефакторинга эффективнее использовать скрипты или утилиты. Но ускорить единичную операцию внутри файла – вполне.

- *Использование Copilot для тестов перед рефакторингом.* Прежде чем кардинально менять код, полезно иметь набор тестов. Codex может быстро набросать unit-тесты по описанию функции (либо по её коду). Вы можете попросить: *“Сгенерируй тесты для функции X, учитывая граничные случаи”*. Получив тесты (обязательно их проверить и поправить!), запускаете – убеждаетесь, что исходная версия проходит. Затем делаете рефакторинг (с помощью ИИ или вручную) и снова запускаете тесты. Это даёт уверенность, что ничего не сломалось. GitHub Copilot особенно силён в генерации тестов и повторяющегося кода [3] – он может за секунды написать десяток тестовых случаев, на которые вручную ушло бы значительно больше времени.

Совет: не забывайте, что Codex не держит контекст ваших правок постоянно. Если вы попросили совета и получили новый код, **сравните его со старым**, подумайте, все ли случаи учтены. Иногда AI уверенно выдаёт оптимизацию, а при ближайшем рассмотрении она упускает важный угол. Всегда лучше совместить несколько методов: собственный анализ, помощь Codex, обзор коллеги (если есть такая возможность). В итоге вы получаете более чистый код и сами учитесь на предложениях ИИ.

Соблюдение архитектурных ограничений и стиля кода

Одно из потенциальных **затруднений при использовании AI-кодогенерации** – чтобы сгенерированный код не превратился в разнородный «стиль Винни-Пуха», не вписывающийся в ваш проект. У каждого проекта есть договорённости: как именуются переменные, как форматируется код, какие паттерны допускаются, а какие запрещены. Ниже перечислим, как сделать так, чтобы Codex придерживался ваших правил.

- **Единый стиль кодирования.** Настройте инструменты форматирования (Prettier, ESLint) в проекте и используйте их постоянно. Copilot, конечно, зачастую и так пишет код в более-менее правильном стиле, но могут быть нюансы (например, кавычки одинарные vs двойные, порядок импортов). Если автоформатирование применяется на сохранение файла, то мелкие расхождения будут поправлены автоматически и Codex «видит» отформатированный код, подстраиваясь под него. Кроме того, ESLint с набором правил (включая naming-conventions) поймает места, где Codex вдруг назвал переменную не по соглашению – вы сразу исправите и дальше AI будет видеть только правильный стиль.
- **Custom Instructions (если доступны).** GitHub Copilot имеет функцию «Ваши настройки» (в Copilot Labs), где можно написать пару фраз о предпочтениях стиля. Например, *“Всегда пиши JSDoc-комментарии к публичным функциям”, “Не используй var, только let/const”*. Эти инструкции не являются жёсткой гарантией, но модель будет больше склоняться им соответствовать. В сочетании с общим контекстом проекта это помогает удерживать единый тон кода. Если же в вашем инструменте таких настроек нет (например, при работе с Codex через OpenAI API), вы можете включать style guide прямо в промпт. Перед тем как запросить большой фрагмент, напишите: *“// Code style: all functions documented, use functional programming where possible”*, и затем запрос. Либо в системном сообщении (для ChatGPT API) заложите ваши правила кодирования.
- **Ограничение использования библиотек.** Архитектурное ограничение часто связано с тем, какие технологии можно или нельзя применять. Codex мог обучиться на разных проектах, где использовались, скажем, `lodash` или Moment.js, или различные UI-библиотеки. Он может без задней мысли предложить их и даже подключить `import`, если посчитает нужным. Если у вас политика “минимум внешних зависимостей”, стоит указать это. В комментарии к файлу: *“// Project constraint: do NOT use external libs if standard library suffices”*. Или если можно, используйте `.copilotignore` (существует такая настройка), куда добавить названия пакетов, которые не должны предлагаться. На практике можно

просто отклонять подобные предложения и писать код самостоятельно – со временем Copilot подстроится. Например, если вы всегда вручную исправляете `forEach` на `for-of`, модель начнёт чаще сразу предлагать `for-of`.

- **Валидация архитектуры через код-ревью и тесты.** ИИ не понимает архитектуру на уровне намерений – он может случайно нарушить инвариант. Например, вы строго разделили домены, а сгенерированный код вдруг лезет напрямую из одного модуля в другой. Здесь спасают **автотесты и ревью**. Пишите тесты не только на функциональность, но и на архитектурные допущения. Например, можно иметь тест, который сканирует проект на отсутствие импортов запрещённых модулей в конкретном слое. Если Codex попробует внедрить неподходящую зависимость, тест или линтер сразу вас оповестят. Тогда вы поправите код и таким образом научите AI, что так делать не надо. Со временем подобных ошибок станет меньше.
- **Постепенное улучшение через рефакторинг.** Даже при всех мерах некоторый сгенерированный код может не идеально соответствовать вашим представлениям об архитектуре. Не страшно – рефакторинг никто не отменял. Возможно, стоит принять первоначальное решение Codex, быстро получить рабочий код, а затем **отрефакторить его вручную** или с помощью того же Codex, приведя в соответствие с архитектурой. Это особенно верно на ранних этапах, когда важнее добиться рабочего прототипа. Главное – не забыть потом выделить время на чистку. Подход “сначала работает, потом красиво” с AI особенно действенен, так как ускоряет первый этап. Но всегда планируйте второй этап! Не оставляйте «как есть», если знаете, что код нарушает ваши же правила.

В итоге, соблюдение архитектуры при использовании Codex сводится к комбинации: **давать ИИ чёткие рамки + применять стандартные инструменты качества к его коду**. Если вы это делаете, то проект останется цельным, даже если его строчки написаны нейросетью. Разработчик задаёт правила игры, а AI старается им следовать – под вашим присмотром.

Ускорение разработки редактора (TipTap/Lexical) и реалтайм-логики с помощью Codex

Рассмотрим теперь прикладной сценарий: вы создаёте **многопользовательский текстовый редактор с AI-помощником**, по функциональности похожий на Notion. Это сложный full-stack проект, сочетающий фронтенд (богатый UI редактора, например на основе библиотеки TipTap или Lexical), бэкенд (реалтайм-сервер для синхронизации и хранения данных) и интеграцию с моделью ИИ (ассистент, работающий на основе содержимого документов). Здесь Codex может значительно ускорить разработку отдельных частей – но важно понимать, как правильно его применять.

Интеграция и кастомизация редактора (frontend). Библиотеки богатого текста, вроде TipTap, Lexical, Slate.js и др., достаточно сложны в использовании, но обычно хорошо документированы. Прежде чем звать Codex, рекомендуется **изучить документацию** выбранной библиотеки и создать минимальный рабочий пример редактора вручную. Поняв основу (например, как инициализировать редактор, как добавить простейший тулбар), можно дальше задействовать Copilot для ускорения типовых задач:

- *Написание плагинов и расширений.* В Notion-подобных редакторах часто нужны кастомные блоки (например, блоки кода, таблицы, упоминания пользователей). Если библиотека поддерживает плагины, вы можете объяснить Codex, какой плагин нужен. Например: *“Create a TipTap extension for a /mention feature that triggers a dropdown of user names”*. Предварительно можно скормить ему API библиотеки (например, открыв файл с базовым

шаблоном Extension). Copilot, натренированный на популярных репозиториях, вероятно предложит код расширения, опираясь на паттерны из документации TipTap или примеров от сообщества. Вам останется его скорректировать под свои нужды. Такой подход экономит время, ведь писать с нуля форматирование сложного блока – долго, а Copilot может «угадать» 70-80% шаблонного кода.

- *Генерация UI-компонентов вокруг редактора.* Редактор – центральная часть, но вокруг него есть UI: панели инструментов, меню, модальные окна. Эти компоненты более типовые (например, React-компонент “ToolbarButton Bold”). Codex справляется с ними отлично, если дать чёткое описание. Можно буквально писать JSX-разметку в комментарии, и Copilot преобразует её в код. Например:

```
{/* Кнопка Bold: иконка "B", по клику вызывает команду редактора
toggleBold */}
```

После этого автодополнение выдаст компонент с `onClick={() => editor.chain().focus().toggleBold().run()}` (если в контексте есть редактор TipTap). Сами по себе эти кусочки просты, но их много – AI существенно ускорит их создание.

- *Документация и пояснения по библиотеке.* Не забывайте, что Codex может рассказать про сам TipTap/Lexical. Можно в Copilot Chat спросить: *“Как в Lexical реализовать коллаборативное редактирование?”* или *“Приведи пример использования TipTap History extension”*. AI, возможно, выдаст пояснение или даже код, взятый (сформулированный) на основе официальных примеров [22] . Однако будьте осторожны: если библиотека очень новая (после 2023 года) или документация изменилась, Copilot может основываться на устаревшей информации. Всегда проверяйте с официальными источниками. Но в целом, для **рутинных операций с UI Codex – большой помощник**: сверстать панель, связать её с состоянием редактора, обработать события – всё это он умеет благодаря контексту и знаниям о распространённых подходах.

Реалтайм-синхронизация (backend). Многопользовательский редактор требует, чтобы изменения одного пользователя почти мгновенно отображались у других. Это обычно достигается через WebSocket-соединения и алгоритмы слияния изменений (Operational Transform или CRDT). Как тут поможет Codex?

- *Реализация WebSocket-сервера.* Если вы используете NestJS, у него есть встроенная поддержка Gateway (Socket.IO). В Express можно взять `ws` или `Socket.IO` сервер. Codex способен сгенерировать каркас такого сервера: обработку подключений, рассылку сообщений. Например, попросите: *“Напиши WebSocket-сервер на Node.js, который рассылает обновления документа всем подключённым клиентам”*. Он может выдать шаблон: создать `WebSocket.Server`, слушать события `connection`, внутри на `message` – бродкаст всем остальным. Это сэкономит вам время на вспоминание API методов. Далее вы конкретизируете логику (например, какие именно сообщения рассылаются – Codex мог придумать формат, вы поправите под свой).
- *Алгоритмы слияния изменений.* Честно говоря, надеяться, что Codex напишет вам с нуля алгоритм OT или CRDT – слишком оптимистично. Такие вещи лучше брать готовыми (существуют Y.js, Automerge и др.). Но Codex поможет **интегрировать готовую библиотеку**. Допустим, вы решили использовать Y.js для расшаренного состояния. Вы читаете документацию Y.js и затем просите Copilot: *“Импортируй Y.js в проект NestJS и сделай сервис DocumentCollab, который хранит Y.Doc для каждого документа и обновляет его при получении патча”*. Модель, возможно, предложит класс с Map<docId, Y.Doc>, методами applyUpdate и getState. Она может и ошибиться в деталях Y.js API, но общую

структуру задаст. Это лучше, чем писать с нуля. По сути, **Codex ускорит написание «обвязки» вокруг сторонних решений** для реалтайма.

- *Код на стороне клиента для реалтайма.* Не забудьте, что и на клиенте нужно подключиться по WebSocket и применять приходящие патчи к редактору. Copilot и тут поможет: он может дописать код, который на событие `message` от сервера применяет update к состоянию редактора TipTap. Если вы правильно назвали вещи (например, message типа `UPDATE_DOC` с payload), то продолжая писать обработчик, AI доделает его.
- *Ошибки и восстановление соединения.* Та часть, о которой часто забывают – обработка отключений, конфликтов. Codex не разрабатывает архитектуру, поэтому сам не предложит "а что, если клиент отвалился?". Но если вы напомните в комментарии (*"// TODO: обработать переподключение клиента и отправить ему текущее состояние документа"*), Copilot вполне может сгенерировать код, который при новом подключении отправит через socket текущее состояние документа (например, полный текст или синхронизирующий патч). Опять же, проверить и отладить это – ваша задача, но черновой вариант AI предоставит.

Интеграция AI-ассистента (RAG). В проекте по условию есть умный ассистент, который должен помогать пользователю, анализируя содержимое документов. Обычно подобное реализуется через **retrieval-augmented generation (RAG)** – когда перед обращением к модели ИИ (например, GPT-4) мы извлекаем релевантные данные пользователя (куски документов, заметок) и вставляем их в запрос, чтобы модель дала контекстуально верный ответ [23]. Как Codex может облегчить жизнь в этой части?

- *Работа с векторными базами данных.* В RAG часто используют векторные представления текста. Есть библиотеки (🤔 Pinecone, Weaviate, или просто pgvector в Postgres) для хранения эмбеддингов. Вы можете поручить Codex рутину по взаимодействию с ними. Например: *"Реализуй функцию searchDocuments(query: string): Document[] с использованием pgvector в PostgreSQL"*. Если к этому моменту у вас подключен модуль pgvector и есть SQL-схема, Copilot может написать SQL-запрос: `SELECT * FROM documents ORDER BY embedding <-> query_embedding LIMIT 5` (в синтаксисе pgvector) – он мог «подглядеть» что-то подобное в обучающих данных. Конечно, вам нужно было заранее получить эмбеддинг запроса – возможно, Codex даже предложит вычислить его через вызов OpenAI Embeddings API (если этот код где-то был в репозиториях). То есть модель может помочь состыковать куски: взять текст, получить эмбеддинг, найти близкие документы, передать их в prompt ассистенту. **Фактически, Codex способен написать "склейку" между вашим хранилищем знаний и вызовом модели.**
- *Обращение к внешним API (OpenAI, и др.).* Интеграция API тоже хлеб для AI-ассистента. Чтобы подключить ассистента, нужно дернуть OpenAI API (или иной сервис) с определённым промптом. Codex наверняка знает, как выглядит вызов к `openai.ChatCompletion.create()` или `openai.createCompletion()` в Node.js (через официальную библиотеку openai). Стоит вам написать комментарий вроде *"// TODO: вызвать OpenAI ChatGPT с сообщением, вернуть ответ"*, Copilot может тут же вставить код типа:

```
const openai = new OpenAIApi(config);
const response = await openai.createChatCompletion({ model: "gpt-4",
messages });
return response.data.choices[0].message?.content;
```

Экономия нескольких минут на поиске в документации – AI сразу даёт используемый шаблон. Но обратите внимание на **безопасность и ошибки**: Codex может упустить обработку ошибок сети или учесть, что ответ может не прийти. Поэтому добавляем сами или просим: *"// обработать ошибки и ограничение скорости"* – возможно, он допишет try/catch с задержкой на повтор.

- *Связка ассистента с редактором.* На фронтенде ассистент, вероятно, появится в виде какой-то панели или чата. Сгенерировать UI под это – задача тривиальная для Copilot, если описать что нужно (например: *"Компонент AssistantSidebar, показывающий ответы AI и имеющий форму ввода вопроса"*). Он создаст React-компонент с состоянием для ответов и вызовами API. API вызовы, кстати, тоже может написать – например, HTTP запрос на ваш бэкенд `/ask`, который вы реализовали выше. **Codex помогает связать фронт и бэк**, предлагая код и там, и там, следуя вашим названиям эндпойнтов и функций.

Таким образом, при разработке подобного **AI-редактора** Codex ускоряет рутинные части: UI-компоненты, обработчики событий, шаблонный код сетевого взаимодействия, интеграцию с API. На сложных частях (алгоритмы синхронизации, тонкая бизнес-логика) он может предложить идею, но полагаться лишь на него рискованно – лучше использовать проверенные решения и звать AI для «обёрток» вокруг них. В итоге, время разработки сокращается, но качество и корректность системы требуют ваших собственных знаний и тестирования.

Отдельно отметим: всегда держите под контролем **качество генерируемого кода**. Реалтайм-системы и AI-интеграции сложно отлаживать, и мелкая ошибка (неправильный идентификатор сообщения, небольшая утечка памяти) может привести к серьёзным проблемам. Codex иногда генерирует *правдоподобный, но неверный* код, поэтому тестируйте каждую такую часть в изоляции (например, имитацией нескольких подключений, проверкой, что ассистент не выходит за лимиты токенов, и т.д.).

Реализация ключевых функций проекта с помощью Codex

В многопользовательском приложении редактора с AI есть ряд базовых функциональных модулей: **аутентификация пользователей, загрузка файлов, хранение и структура документов, публикация контента, система прав доступа и интеграция RAG (поиск по знаниям)**. Рассмотрим, как можно грамотно применять Codex при разработке каждой из этих составляющих, и на что обратить внимание.

1. Аутентификация и авторизация. Начало любого веб-приложения – это система регистрации, логина и управления пользователями. Здесь Codex может облегчить:

- *Имплементацию стандартных подходов.* Скажем, вы решили использовать JWT для аутентификации. Вместо того чтобы вспоминать все детали, можно попросить AI: *"Напиши middleware для Express, который проверяет JWT в заголовке Authorization"*. Как обсуждалось, он выдаст готовый код проверки токена и передачи пользователя в `req` [13]. Однако **осторожно**: убедитесь, что он использует ваш способ хранения секретов (например, берет секрет из env, а не хардкод) и правильные коды ошибок. Или, например, на NestJS можно попросить: *"создай Guard, проверяющий роли пользователя по декоратору @Roles"* – Codex знаком с шаблонами Nest и, вероятно, сгенерирует guard, похожий на примеры из док. Вам останется проверить логику. **Проверка безопасности – ваша работа**: AI может не помнить актуальные рекомендации (например, алгоритм подписи JWT, длину соли для паролей). Поэтому после генерации кода аутентификации **пересмотрите его с точки зрения безопасности**: нет ли уязвимостей (SQL-инъекции, уязвимых хешей, утечек

информации). Желательно подключить линтеры безопасности или сканеры (типа Snyk), они иногда могут выявить проблемы в AI-коде [24].

- *OAuth и внешние провайдеры.* Если нужен вход через Google/GitHub и проч., Codex также поможет сэкономить время. Он знает про Passport.js и подобные библиотеки. Дайте указание: *"Реализуй OAuth2 авторизацию через Google в NestJS приложении"*. Вероятно, он выдаст конфигурацию Strategy, контроллер для редиректа и callback – по шаблонам. Всё равно нужно свериться с официальной документацией (AI мог устареть, особенно если API изменилось после 2021-2022), но каркас, возможно, уже будет на месте.
- *Авторизация (права доступа).* Предположим, у вас есть роли пользователей или нужно проверять право на доступ к документу. Можно попросить Copilot написать декоратор или функцию проверки: *"проверь, что текущий пользователь является создателем документа или имеет роль admin, иначе выдай 403"*. Он впишет этот чек в нужные места. Но **позаботьтесь о тестах**: напишите несколько unit-тестов на функции авторизации (можно тоже сгенерировать их), чтобы убедиться: AI-код правильно различает случаи "владелец", "не владелец", "admin". Это критичная логика, и её правильность – залог безопасности приложения.

2. Загрузка и хранение файлов. Редактор может позволять прикреплять изображения, видео и т.д. Это типовая задача, Codex тут весьма кстати:

- *Endpoint для загрузки.* Можно описать: *"Создай Express-роут POST /upload, который принимает файл изображения (form-data) и сохраняет на диск (или в облако)"*. Copilot, скорее всего, выдаст код с использованием multer (популярный middleware для загрузки в Express). Это удобно – он подключит multer, настроит папку и вернёт URL сохранённого файла. Но **внимание**: убедитесь, что нет пробелов в безопасности (например, проверка типа файлов, размера). AI мог не добавить ограничений – лучше дописать: *"// TODO: добавить проверку типа файла (только PNG/JPEG)"*.
- *Взаимодействие с облачными хранилищами.* Если файлы должны храниться, например, в S3, Codex может помочь с интеграцией SDK. Запрос: *"Напиши функцию uploadToS3(file: Buffer, name: string)"* вероятно даст шаблон с использованием `AWS.S3` клиента, взятый из официальных примеров. Экономия: не лезть в документацию, а сразу получить рабочий код отправки объекта. Конечно, нужно подставить ваши креденшалы/конфиг.
- *Обработка результатов на фронтенде.* После загрузки обычно нужно показать превью файла или список. Copilot поможет и здесь: быстро сгенерирует React-компонент для списка файлов, или функцию, которая берёт base64 строку и делает preview.
- *Рефакторинг/uploads в фоновом режиме.* Если загрузки становятся сложными (например, генерация миниатюр), AI-ассистент может подготовить каркас фонового сервиса (например, на NestJS модуль с Bull (Redis Queue) для обработки файлов). Он напишет шаблон кода, останется подстроить под нужды.

3. Структура и хранение документов. Суть редактора – хранить иерархические документы, возможно структурированные в виде блоков (как в Notion). Встает вопрос: как это моделировать в базе и реализовать CRUD операций. Codex может помочь:

- *Проектирование схемы данных.* Можно описать AI предполагаемую модель: *"У нас есть таблица Documents(id, title, content, parent_id) для дерева документов, где content хранится в JSON. Предложи SQL схему."*. Он, вероятно, выведет SQL с self-referencing foreign key для parent_id, индексами и т.д. Даже если вы сами знаете, это проверить полезно – вдруг что-то упущено. Или, если NoSQL, спросить: *"Как представить дерево документов в MongoDB?"* – ответ может натолкнуть на мысль, например, хранить путь узла в поле. Однако это больше теоретизирование – для практики лучше самому решить, а Codex – для реализации.

- *CRUD операции.* Как только решено, как хранить, рутинно написать сервис для создания/чтения/обновления документов можно поручить Codex. Например: *“Напиши класс DocumentService с методами createDocument, getDocument(id), updateDocument, deleteDocument. Используй TypeORM (или Prisma) для доступа к PostgreSQL.”*. Copilot, зная синтаксис ORM, сгенерирует методы – с вызовами `repository.save`, `findOne` и т.д., и даже обработкой ошибок. Это ускоряет работу, хотя всё равно надо глазами проверить каждую строку (нет ли там, к примеру, забытого await или неправильного условия).
- *Обновление структуры (перемещение узлов).* Более сложная задача – например, переместить подраздел из одного раздела в другой. Алгоритмически это не тривиально (нужно либо обновить parent_id, либо менять всю ветку). Объясните Codex: *“Реализуй функцию moveDocument(docId, newParentId) для дерева документов в SQL. Учти, что документ нельзя переместить внутрь своего потомка.”*. AI может попытаться написать проверку на циклы (через запрос всех родителей, например) и обновить parent. Это нетривиально, он может и ошибиться, но как минимум даст основу, а вы доведёте.
- *Версионирование и история.* Если нужны версии документов, Codex может подсказать схему (таблица Revisions) и сгенерировать код сохранения версии при каждом обновлении. Так как он обучен на реальных проектах, он мог видеть похожие решения и повторит их.

4. Публикация и шаринг контента. Предположим, в редакторе можно публиковать документы (делать их доступными по публичной ссылке) или делиться с конкретными людьми:

- *Генерация публичных ссылок.* Попросите: *“Сгенерируй уникальный токен и сохрани его для документа, чтобы по нему можно было получить документ без логина”*. Copilot напишет функцию, которая генерирует случайную строку (скорее всего, Crypto или UUID) [25], сохраняет в поле `public_token`. Возможно, сразу выдаст эндпойнт `/public/:token` для просмотра. Опять же, детали (срок действия ссылки, права) надо додумать самому, но шаблон – уже результат.
- *Механизм приглашений пользователей.* Например, invite по email с определённой ролью. Codex поможет сформировать письмо, если надо, или endpoint для акцепта приглашения. Но надо помнить о безопасности: ссылки-приглашения, лучше одноразовые. AI может не предусмотреть повторное использование – вам следует добавить проверку.
- *Режимы доступа.* Если у документа три режима (приватный, по ссылке, по конкретным пользователям), то на стороне UI Copilot тоже может ускорить: выпадающий список, функция изменения статуса, и на бэкенде – проверка доступа в middleware на основании этого статуса. Просто ясно опишите это, и он реализует базовую логику. Затем прогоним тестами.

5. Права доступа и разрешения. В продолжение темы – более сложные ACL (access control lists). Например, у документа могут быть редакторы, читатели, комментаторы:

- *Модели ACL.* Codex подскажет схему таблиц: DocumentPermissions с полями user_id, doc_id, role. Это стандартно и он знает.
- *Функции проверки.* Как упоминалось, AI напишет guard или middleware, который проверит: *“есть ли у user право 'edit' на данный doc”*. Это делается запросом или проверкой в кэше, Copilot код напишет.
- *UI подсказки.* На фронте можно попросить компонент ShareDialog, который использует API списка пользователей, чекбоксы прав – он сгенерирует JSX разметку и, возможно, базовую логику (но тут много деталей, например автокомплит пользователей – AI может не осилить без контекста, но может хотя бы форму).
- *Опять же тесты!* Правила доступа – то, что должно быть безупречно. После генерации таких модулей обязательно протестируйте все комбинации (можно поручить Codex сгенерировать тест-кейсы: *“Напиши тесты на функцию canEditDoc(user, doc) с сценариями:*

автор документа, приглашённый редактор, просто залогиненный, не залогиненный." Он опишет, вы допишете проверку).

6. RAG-интеграция (AI-помощник). Чуть ранее мы затрагивали, как Codex помогает написать связующий код для retrieval. Добавим практические моменты:

- *Потоковые ответы (streaming) от AI.* Часто, чтобы UX был лучше, ответ ассистента присылается частями (stream). Реализовать это через OpenAI SDK можно, но синтаксис неочевиден. Copilot знает пример с `{ stream: true }` и обработкой data по chunk'ам. Спросите его: *"Как получить стриминг-ответ от OpenAI и пересылать клиенту в реальном времени через WebSocket?"* – он может даже сгенерировать соединение этих двух частей. Это пример нестандартной задачи, которую AI всё же может облегчить: либо ответом с пояснением, либо сразу кодом. Опять же, **внимательно протестируйте**: стриминг – тонкая вещь, Codex мог где-то опечататься.
- *Ограничение запросов и кеширование.* Если ассистент использует дорогостоящий API, разумно кешировать результаты или ограничивать частоту. Эти задачи повторяемые – например, "кеш по ключу запрос+документ" или "не более 5 запросов в минуту с IP". Вместо ручного написания – попросите AI: *"Добавь в этот эндпойнт кэш Redis на 5 минут"* или *"реализуй rate limiter (5 per minute) для API /ask"*. Он, обученный на типовых решениях (библиотеки express-rate-limit или @nestjs/throttler), быстро выдаст интеграцию. Ваша задача – удостовериться, что это вписывается в ваш стек (например, он может предложить express middleware, а у вас Nest – надо адаптировать).
- *Подготовка данных для обучения AI.* Если планируется тонкая донастройка модели или анализ usage-логов, Codex может генерировать SQL-агрегации или скрипты. Например, *"Собери все вопросы пользователей и ответы ассистента за месяц в CSV"* – он может написать SQL join по таблице запросов и ответов, даже добавить заголовки. Вещь простая, но экономит время.

Как видно, практически для каждого аспекта функциональности **AI-ассистент предлагает шаблонное решение**. Это особенно ценно, когда вы используете распространённые технологии: Express, Passport, JWT, Socket.IO, PostgreSQL, OpenAI API – всё это есть в тренировочных данных Codex, и он выдает узнаваемые блоки кода. Но чем специфичнее ваш код (например, нестандартная бизнес-логика), тем больше нужна ваша собственная реализация с минимальной подсказкой от AI.

И помните главный принцип: **Codex ускоряет черновое кодирование, но проверка и доводка – за вами**. В критичных местах (безопасность, согласованность данных) делайте code review, подключайте статический анализ. Опыт показывает, что AI иногда совершает глупые ошибки, которые опытный разработчик сразу заметит. Например, забывает await, не обрабатывает промис-ошибку, или сравнивает строки с ошибкой. Поэтому доверяй, но проверяй: пусть Codex пишет 80% кода, а вы уделите усилия на те 20%, которые делают код надёжным и production-ready.

Проверка и улучшение кода, генерируемого Codex

Высокое качество проекта – обязательная цель, независимо от того, писал код человек или нейросеть. При использовании AI-ассистента особое внимание надо уделять **проверке, тестированию и улучшению** полученного кода, чтобы ни баги, ни уязвимости не прокрались незамеченными.

Всегда проверяйте предложения Copilot вручную. Это золото правило: **AI может ошибаться**, и довольно уверенно. Никогда не принимайте сгенерированный код “на веру”, не разобравшись. GitHub прямо рекомендует разработчикам верифицировать каждый фрагмент – не только на предмет работы, но и на предмет читабельности и поддержки в будущем [26]. Хорошая практика – после автодополнения **прочитать код вслух себе**: “так, эта функция берёт X, вызывает Y, возвращает Z; всё ли логично, не забыл ли про крайние случаи?”. Если что-то смущает – лучше исправить сразу или переспросить AI, чем тянуть технический долг. Особенно важна такая перепроверка в местах, где вы сами не эксперт. Например, Codex нагенерировал вам сложный RegEx – вполне возможно, он не работает на каких-то входных данных. Протестируйте его на разных примерах.

Пишите и гоняйте тесты. Тестирование – наш верный союзник в контроле качества AI-кода. Старайтесь покрывать функционал юнит-тестами или интеграционными тестами. Кстати, сам Copilot **умеет писать тесты**, причём довольно добротно, если функции хорошо описаны [3]. Вы можете после написания модуля попросить: *“сгенерируй тесты для методов класса DocumentService”*. Он создаст ряд случаев, имитирующих разные сценарии. Хотя их тоже надо просмотреть и поправить, это даст базу. Прогоняйте тесты при каждом крупном добавлении AI-кода. Если что-то падает – разберитесь, баг ли в логике или в самом тесте (AI-тест мог неправильно ожидать поведение). Автотесты, помимо проверки корректности, еще и служат примером использования кода – вы увидите, насколько удобно ваш API спроектирован. **Human-in-the-loop** принцип здесь критичен: AI написал – человек протестировал [24]. Такая двойная проверка улавливает большую часть ошибок до того, как они попали в прод.

Используйте анализаторы кода и линтеры. Прогоняйте сгенерированный код через ESLint/TSLint с строгими правилами. Линтер может выявить, например, неиспользуемые переменные, странные конструкции, несоответствие типам. Также статические анализаторы (SonarQube, CodeQL) способны найти уязвимости: неэкранированные вводы, опасные вызовы, известные небезопасные функции [24]. Например, если Codex предложил прямой SQL без параметризации, анализатор пометит возможную SQL-инъекцию. Воспринимайте такие предупреждения серьёзно и исправляйте сразу. AI иногда выдаёт код, попадающий под известные security-правила, о чём были случаи в сообществе [27]. В идеале, настроить CI/CD, где каждый коммит (а Codex код тоже коммитится) прогоняется через линтеры и тесты – чтобы никакой “полуфабрикат” не проскочил дальше dev-ветки.

Берите второе мнение (от людей). Если работаете в команде – не стесняйтесь отправлять AI-сгенерированный код на ревью коллегам. Возможно, стиль слегка отличается или решение избыточно – лучше получить отзыв. Если вы соло-разработчик, можно периодически пересматривать свой код “со стороны” спустя пару дней. Часто свежим взглядом видно, что тут Codex перемудрил, а здесь не учёл важный момент. В сообществе появилось даже понятие “AI code smell” – когда код вроде бы рабочий, но какой-то странноватый или не идиоматичный для данного языка. Убирайте такие места. Иногда лучше переписать их вручную, чем пытаться править по кусочку. Однако можно и спросить сам AI: *“Этот код можно сделать более понятным?”* – и он попытается отрефакторить. Совмещение нескольких подходов делает результат чище.

Ограничивайте «галлюцинации». В режиме диалога, если замечаете, что Codex начал уверенно выдавать чепуху (например, ссылаться на несуществующую функцию или API), не пытайтесь продолжать диалог на этой неверной основе. Лучше очистить контекст или переформулировать вопрос. Copilot inline-режиме реже галлюцинирует, т.к. привязан к вашему коду, но и там бывает (например, предлагает вызов метода, который вы не реализовали и не собирались). Всегда проверяйте: откуда взялся тот или иной вызов? Нет ли магических

несуществующих зависимостей? Если есть – убирайте и диктуйте явнее. В итоге проект не обрастёт “мертвым” кодом.

Доводите до продакшн-качества. После того как функциональность сделана и предварительно проверена, подумайте, какие **нефункциональные аспекты** нужно улучшить: логирование, обработку ошибок, производительность. AI обычно пишет минимально достаточный код: например, если произошла ошибка – просто кидает. Вам же, возможно, нужно залогировать и вернуть дружелюбное сообщение. Такие места лучше пройти и доделать вручную. Производительность: Codex написал корректный, но O(N^2) алгоритм – замените на более эффективный, если нужно. Зачастую при использовании AI остаются упущенными тонкие моменты, которые важны в продакшне (очистка ресурсов, сделки транзакций, тайм-ауты). **Перед релизом устраивайте небольшой аудит кода.** Это хорошая привычка и без AI, но с AI – тем более: просмотрите критические пути, убедитесь, что всё учтено.

Учтите ограничения знаний Codex. У Copilot есть *обучающая выборка* с определённым «срезом времени». Он мог не знать о новинках после примерно 2022 года [28]. Если вы используете свежайший фреймворк или новую версию, AI может предлагать методы из старой версии, или не знать про важные изменения. Например, если вышел NestJS 10 с иной конфигурацией модулей, Copilot может продолжать писать по старому образцу. Будьте бдительны и в таких местах лучше ориентироваться на официальную документацию. С другой стороны, наработки десятилетий (SQL, HTTP, алгоритмы) – у него хорошо усвоены. Так что всё новое проверяем особо тщательно.

Подводя итог: **Codex сам код не тестирует** и не доказывает его корректность, это работа разработчика [29]. При правильной организации (автотесты + ревью + анализаторы) можно достаточно уверенно отфильтровать ошибки и получить качественный код. Многие команды отмечают, что при использовании Copilot число багов не растёт, а иногда даже снижается – благодаря тому, что AI предлагает сразу неплохие реализации, проверенные практикой [30]. Но это верно только, если выполнять роль “контролёра качества” и не отключать критическое мышление.

Долгосрочный процесс: Codex как помощник, а не замена мышления

Одно дело – написать проект с помощью Codex, другое – затем жить с этим кодом, поддерживать и развивать его. Важный вопрос: **как выстроить длительное сотрудничество с AI-инструментом, чтобы он помогал, но не притуплял навыки разработчика и не приводил к потере понимания кода?**

Не прекращайте думать самостоятельно. Это кажется очевидным, но поддаваясь удобству Copilot, легко начать доверять ему архитектурные решения или выбор технологий. Всегда останавливайтесь и продумывайте ключевые моменты самостоятельно. Например, планируя новую фичу, сначала грубо прикиньте, как **вы** бы её реализовали: какие модули тронуть, какие новые классы нужны, где возможны проблемы. Лишь затем приступайте к кодингу с Copilot. Иначе есть риск превратиться в оператора, который слепо расширяет проект подсказками ИИ, постепенно теряя целостное видение. Codex – это как автоматическая коробка передач: да, вам не нужно вручную переключать скорости, но вы всё равно ведёте машину и следите за дорогой. Не отпускайте руль разработки из своих рук.

Регулярно изучайте сгенерированный код, чтобы им овладеть. По мере того, как проект растёт, убедитесь, что **вы понимаете каждый его уголок**, даже если его писал AI. Тратьте время

на чтение кода, написанного с помощью Codex, рефакторите его так, словно он написан джуном, которому нужен ментор. Поправляйте названия, добавляйте комментарии, упрощайте сложные конструкции. Этот процесс обеспечит, что **вы становитесь хозяином кода**, а не Codex. К тому же, так вы учитесь: Codex мог применить какой-то шаблон, которого вы раньше не встречали – разберитесь, почему так и как это работает. Если что-то кажется лишним или не оптимальным – переделайте. В конечном счёте, через несколько циклов таких улучшений, кодовая база станет вполне "вашей", а не загадочным творением ИИ.

Не дайте архитектуре дрейфовать. По мере добавления новых функций следите, чтобы архитектурные принципы, заложенные вначале, соблюдались. Codex может ненароком ввести "технический долг" – например, для простоты кинуть SQL в контроллере, минуя сервис. Если вовремя не заметить, таких обходных путей может накопиться. Поэтому полезно периодически проводить **рефакторинг-сессии без участия Codex**: посмотреть на структуру проекта свежим взглядом, убедиться, что слои четкие, зависимости направлены правильно. Если видите отклонения – исправьте и задумайтесь, почему они возникли. Может, вы неучли какой-то сценарий в архитектуре, и Codex восполнил пробел. Тогда доработайте архитектуру, документируйте её, чтобы в будущем ИИ тоже следовал обновлённым правилам.

Ограничивайте использование AI там, где нужен творческий подход. Разработка – это не только шаблоны, но и изобретение новых подходов, оптимизация. Codex хорош в известных решениях, но **новатором** он не станет. Поэтому ключевые архитектурные инновации, оптимизации алгоритмов, выбор нестандартных инструментов – это зона, где лучше полагаться на свой опыт и исследование, а не на подсказки модели. Используйте AI, чтобы проверить идеи (например: *"можно ли с помощью Redis Sorted Set реализовать мою задачу?"* – он опишет, как, но решение принимаете вы). В длинной перспективе, это сохранит и разовьёт ваши навыки как архитектора, а Codex останется ценным *исполнителем* ваших идей.

Учитесь у Codex, но и задавайте свою планку. Наблюдая за предложениями Copilot, можно многому научиться – новым API, лаконичным трюкам языка, паттернам. Не стесняйтесь признаваться: *"О, я не знал, что так можно!"* и брать на вооружение. Но также критически оценивайте: *"А нет ли способа лучше?"*. Иногда AI предлагает окольный путь, а вы знаете более прямой. Не принимайте всё как истину – сохраняйте планку качества и оптимальности, которой придерживались бы без AI. Если вы всегда пишете функциональные компоненты, а Copilot внезапно насовал классовых – смело переделывайте на свой лад. В долгой перспективе, проект должен выглядеть единообразно, будто его писал один профессионал, а не сотня разных людей (чьи стили смешались в модели).

Контролируйте зависимость от инструмента. Представьте, что завтра у вас отняли Copilot – сможете ли вы работать с этим проектом дальше? Ответ должен быть: *"Да, конечно, ведь я всё в нём понимаю."* Если же вы чувствуете, что стали слишком зависимы (например, вам сложно написать код без подсказки, даже если понимаете что нужно) – потренируйтесь время от времени **кодить самостоятельно**. Можно, например, выключать Copilot на час-другой, реализуя несложные модули вручную, чтобы не утратить навык. Это как с калькулятором: важно уметь считать и без него, хотя с ним быстрее.

Следите за обновлением самого Codex. AI-инструменты развиваются. Возможно, выйдут новые версии Copilot, обновятся модели (например, Codex заменится на более мощный аналог). Нужно иногда читать новости, смотреть, что нового: может, появились настройки, позволяющие лучше задавать стиль, или расширилась контекстное окно (значит, можно давать больше файлов на вход). Используйте новое, если оно поможет. Например, Copilot X обещает интегрировать

обсуждение PR и пояснения к диффам – это тоже упростит поддержание проекта в будущем. Быть в курсе – значит, получать максимум пользы долгосрочно.

В общем, долговременная работа с Codex – это баланс: **максимальная выгода от ускорения, но минимум уступок в качестве мысли и дизайна**. Правильно найденный баланс сделает вас супертрофейным разработчиком: вы и проектно мыслите, и рутину делаете молниеносно с помощником. А проект будет успешным и поддерживаемым, потому что под капотом – продуманные решения, покрытые тестами и ясно задокументированные, кем? Вами, при поддержке Codex.

Генерация документации и знаний о проекте с помощью Codex

Хорошая документация – признак зрелого проекта. И здесь AI-ассистент способен внести огромный вклад, сэкономив часы на писание комментариев, README, описаний архитектуры и т.д.

Автодокументирование кода. Copilot умеет автоматически генерировать комментарии и docstring'и к функциям, если начать их писать. Попробуйте: над функцией вставьте шаблон JSDoc:

```
/**
 *
 */
```

и нажмите автодополнение. Модель заполнит описание того, что делает функция, на основе её имени и кода. Часто описание бывает точным и хорошо сформулированным [31] . Возможно, вам останется чуть подправить стилистику. Таким образом, вы можете быстро обеспечить все публичные методы проекта понятными комментариями. Это не только удобно для вас и команды, но и для AI: с комментариями он дальше будет ещё лучше понимать контекст. Аналогично с типами: можно попросить AI добавить комментарии к сложной структуре данных или к классу, объяснив его предназначение.

Генерация README и архитектурных обзоров. Когда проект подходит к более-менее рабочему состоянию, надо бы написать README.md с описанием, как запустить, какие технологии используются, как устроена архитектура. Codex в состоянии сгенерировать начерк такого текста, если вы дадите ему основные пункты. Например, можно написать план:

```
# Проект X

## Установка

## Стек технологий

## Архитектура

## Использование
```

и попросить Copilot заполнить. Он, глядя на код, уже догадывается, что за стек (например, увидит package.json с NestJS/React, увидит, что Postgres используется) – и укажет их [32]. Может даже описать архитектуру: “Frontend built with React and TipTap for rich text editing, backend on NestJS handling real-time collaboration via WebSocket, and a PostgreSQL database with a documents table” – конечно, по-английски, но вы переведёте или сразу попросите на русском. Такой автосгенерированный README обязательно прочитайте и исправьте факты (вдруг что-то не так угадал), но это даёт быстрый скелет документации, когда времени мало.

Диаграммы и схемы. Более продвинутый трюк – попросить AI сгенерировать диаграмму (например, в формате Mermaid) по описанию архитектуры. Например: *“Нарисуй Mermaid-диаграмму: [Frontend] -> [Backend API] -> [Database]; [Backend] -> [OpenAI API]”*. Copilot может написать код диаграммы. Можете вставить её в README для наглядности. В итоге, у вас архитектура задокументирована схемой практически “по щелчку”.

Объяснение архитектурных решений. Интересный подход – использовать GPT-модели для **документирования обоснований** (Architectural Decision Records, ADR). Вы можете в диалоге описать: *“Мы выбрали NestJS вместо Express, потому что…”* – и попросить продолжить красиво. Если в исходных данных были статьи (вспомним, у нас в поиске была статья, сравнивающая Nest и Express), модель может выдать сформулированные преимущества: структурированность, DI, масштабируемость [20], а для Express – простота, гибкость [33]. Вы, конечно, проверите и адаптируете под свой проект. Так можно быстро составить раздел “Почему выбран такой стек” в документации.

Документация API и моделей. Если требуется сделать справочник API (например, OpenAPI/Swagger), некоторые части можно сгенерировать Copilot’ом. Например, вы пишете YAML для Swagger – на основании названий контроллеров и DTO он будет автодополнять пути, параметры, модели. С TypeScript-телами можно и программно через рефлексию генерировать (Nest умеет), но если надо вручную – AI поможет, чтобы не писать однообразное.

Комментарии в коде для обучения новых участников. В проекте “редактор с AI” много нетривиальных мест (реалтайм синхронизация, RAG и т.д.). Хорошо бы рядом с кодом оставить поясняющие комментарии: *“// Это реализовано через CRDT (Y.js) – мы храним состояние в памяти и обновляем всех клиентов при изменении”*. Такие вещи Copilot тоже может подсказать формулировку, вы только поправите. Чем лучше документирован код, тем легче его поддерживать.

Составление FAQ или руководства пользователя. Допустим, вы хотите написать для пользователей (или для команды) небольшой FAQ: “что делать, если ассистент не отвечает” или “как подключить новый модуль”. Можно частично поручить это AI: он сгенерирует предположительные вопросы и ответы, особенно если дать ему контекст. Пример: *“Составь 5 часто задаваемых вопросов по использованию этого редактора с ответами.”* – он попытается (возможно, используя типичные проблемы). Опять же, вам редактировать, но время экономится.

Важно: **не забывайте сверяться с реальностью** в этих AI-сгенерированных текстах. Модель может написать, что “наш редактор поддерживает коллаборацию в оффлайне” – хотя вы такого не делали. Она может перепутать названия. Поэтому проверка фактов – критична. Но стиль и общая структуру текста она часто создаёт прилично, избавляя от синдрома чистого листа.

В результате, используя Codex и GPT-модели, вы можете значительно **облегчить создание документации**: от комментариев в коде до высокоуровневых описаний. Это повышает

прозрачность проекта. Новые разработчики, подключаясь, быстрее разберутся, да и самому автору через время проще вспомнить логику по хорошим комментариям. А потрачено на это было меньше усилий, чем писалось бы всё вручную. В сочетании с генерируемыми тестами и явными типами, проект получается самодокументирующимся во многом – что и является одной из целей современного разработки.

Постепенный переход к самостоятельному владению кодовой базой

В начале разработки с Copilot, вы, возможно, полагаетесь на него довольно сильно – он пишет значительные куски, помогает во всём. Но по мере созревания проекта важно, чтобы вы как разработчик **полностью владели кодовой базой**, не ощущая “чёрных ящиков” и снизив зависимость от AI в критичных местах. Как этого достичь?

1. Углубляйтесь в сложные участки кода. Найдите время подробно пройтись по самым сложным частям, даже если они работают. Если их генерировал AI, убедитесь, что понимаете каждую строку. Сделайте, если нужно, ревью “себе самому”: например, открыть Pull Request с кодом и написать комментарии (или спросить ChatGPT “почему здесь так?”). Это выявит потенциальные скрытые проблемы и укрепит ваше понимание. Иначе можно остаться в ситуации, когда код работает, но как – помнит только AI. Избегайте этого.

2. Рефакторинг без AI. После того как функциональность реализована, попробуйте отрефакторить некоторые критические модули **без подсказок** – исключительно собственным мозгом. Это проверка: можете ли вы улучшить код, придуманный AI, на основе собственного опыта? Если да – отлично, вы не утратили навык. Если нет – возможно, стоит почитать материалы по данному вопросу или потренироваться. Например, AI написал алгоритм, вы считаете его некрасивым – сядьте и перепишите лучше. Потом сравните: стало ли понятнее, короче, быстрее? Такой активный пересмотр кода помогает “присвоить” его себе.

3. Обучение по ходу. В процессе использования Codex вы наверняка встретите новые библиотеки, паттерны. Не ограничивайтесь тем, что “AI же за меня знает, как их применять”. Берите и **читайте первоисточник**: официальную доку, статьи. Используйте AI-решения как отправную точку, но затем углубляйтесь. Например, Copilot помог с Y.js – почитайте документацию Y.js, вдруг там есть тонкости (с вариантами конфликта) которые AI не учёл. В итоге, вы уже не зависите от AI в этом вопросе: вы сами знаете, как оно устроено. Постепенно по всем основным технологиям вашего стека вы станете достаточно компетентны, чтобы и без AI имплементировать фичи (просто с AI быстрее).

4. Контролируемое уменьшение помощи. Попробуйте иногда решать небольшие задачи без Copilot. Например: “сейчас попробую сам быстро набросать эту функцию, а Copilot пусть только дополняет мелочи”. Или даже отключите его на час, как эксперимент, и поработайте old school. Если вы ловите себя на том, что без AI стало тяжко вспоминать синтаксис или API – это сигнал подтянуть знания. В идеале, AI-ассистент не должен превращаться в “костыль”, без которого вы забыли язык программирования. Он лишь ускоряет. Можно сравнить с автопилотом в самолёте: пилот должен уметь и вручную посадить самолёт при необходимости. Так же и вы – ключевые части приложения вы должны быть готовы при необходимости написать или починить вручную, если завтра, гипотетически, AI-сервис будет недоступен.

5. Знание всей “истории” проекта. Когда код нагенерирован, есть риск, что часть решений неочевидна. Поэтому ведите запись (да хотя бы в голове, а лучше в ADR или README), почему и

как что делалось. AI не оставит за вас комментарий “этот workaround нужен из-за бага в библиотеке X”, если только вы сами не добавите. Фиксируйте такие моменты. Потом, через полгода, вы или другой человек, глядя на код, не будет ломать голову “ну почему тут именно так решено?”. Помните, AI не несёт ответственности за поддержку – это на вас. Так что код должен иметь объяснения, понятные людям.

6. Постоянно тренируйте архитектурное мышление. Даже сделав проект на Codex, на следующем этапе задумайтесь: *“А как бы я мог его улучшить кардинально? Может, разбить на микросервисы? Или вынести вычислительно тяжёлую часть в отдельный сервис?”*. Такие идеи – инициатива только от человека. AI их не предложит, если вы не спросите. И даже если спросите, вам решать, надо ли. Поэтому инициируйте архитектурные улучшения сами. Это помогает не застаиваться и не становиться просто “наблюдателем за AI”. Вы рулевой, и видите наперёд. Скажем, вы понимаете: окей, сейчас один сервер, но будет расти нагрузка – надо будет масштабировать. Продумайте сразу, а AI пусть поможет позже реализовать (например, интеграцию с Redis cache или перенос файлов в CDN). Но эти задачи ставите вы.

В конечном счёте, цель – **сделать себя независимым от AI, используя AI по максимуму**. Это звучит двусмысленно, но логика такая: пока проект делается – вы используете Codex, потому что это разумно и эффективно. Но после того как всё готово, вы должны быть в состоянии уволить Codex и всё равно поддерживать и развивать систему. Если это так – вы достигли идеального баланса. Если чувствуете, что без AI “как без рук” – значит, в процессе что-то пошло не так, и стоит подтянуть свою техническую базу.

Хорошая практика – по завершении каждого крупного модуля задавать себе вопрос: *“Что я узнал, что могу теперь сам воспроизвести без подсказок?”*. Например: *“Теперь я понял, как работает CRDT, и смогу реализовать простой CRDT без AI”*. Если ответ положительный – супер, вы выросли вместе с проектом. Если нет – возможно, полагание на AI было чрезмерным в этой части, стоит компенсировать, почитав/попрактиковавшись.

Напоследок: **воспринимайте Codex как ускоритель вашего обучения и производства кода, а не как костыль**. Тогда с каждым проектом вы становитесь всё более самостоятельным и опытным разработчиком, просто вооружённым лучшими инструментами. Именно этого мы и хотим добиться – а не зависимости от подсказчика.

Выбор технологий: backend-стек, база данных, вариант Codex

Наш проект-редактор мог бы быть реализован на разных технологиях. От выбора стека зависит и взаимодействие с AI-ассистентом. Рассмотрим плюсы и минусы популярных вариантов, а также различных форм самого Codex, чтобы помочь вам выбрать оптимальное сочетание под свои нужды.

Express vs NestJS vs tRPC (Node.js backend). Все три подхода совместимы с TypeScript и могут быть использованы с Codex, но отличаются философией:

- **Express.js:** минималистичный фреймворк. *Плюсы:* Простота и гибкость. Вы практически не ограничены в структуре – можете выстроить всё как хочется. Огромное сообщество, примеров масса. Codex очень хорошо знаком с Express-синтаксисом и типовыми middlewares – он без проблем сгенерирует роуты, обработчики, взаимодействие с `req`/`res`, подключение JWT, multer и т.д. Это значит, что AI-помощь будет на высоком уровне.

Минусы: Отсутствие встроенной структуры – в большом проекте легко скатиться в хаос, если не поддерживать архитектуру вручную. Типизация не навязывается – придется тщательно определять типы самому, иначе теряется смысл TS. В команде Express-проекты сложнее поддерживать, так как каждый может писать по-своему. В контексте AI это риск: Codex может в разное время генерировать разный стиль, если вы чётко не задали каркас. Express хорош для прототипа или небольшого сервиса, но для нашего сложного приложения может потребовать много дисциплины, чтобы он оставался поддерживаемым.

- **NestJS:** современный фреймворк, вдохновлённый Angular/Spring. *Плюсы:* Чёткая архитектура из коробки (модули, контроллеры, сервисы, DI). Отличная поддержка TypeScript, декораторы, готовые решения для валидации, авторизации, WebSocket gateway и проч. В команде снижает "bus factor": новый член команды легко поймет структуру, т.к. она стандартна. Codex знаком с NestJS подходом – он знает про `@Controller`, `@Get`, `@Injectable` и т.п., может генерировать boilerplate для модулей. Особенно выигрывает генерация повторяющихся частей: сервисы, dtos, guards – AI их делает быстро, следуя примеру. *Минусы:* Большой объем шаблонного кода (который, правда, и пишется Codex'ом быстро). Некоторая избыточность для маленьких задач. Более высокий порог входа, если вы не знакомы с Nest – нужно понять его концепции. Codex может помочь в обучении, но без понимания DI можно нагенерировать что-то неправильно связывающееся. То есть Nest требует, чтобы **разработчик знал фреймворк**, тогда AI просто ускоряет его работу. Если не знать – лучше сначала изучить. В нашем случае, NestJS подходит, т.к. проект большой и комплексный. AI будет полезен, чтобы гасить рутину (создание десятков файлов по шаблону), а строгая структура поможет поддерживать порядок [20].
- **tRPC:** это не совсем фреймворк, а библиотека для создания end-to-end типобезопасных API. *Плюсы:* Невероятная скорость разработки фронт+бэк одновременно – вы описываете router на бэке, а на фронте сразу вызываете как обычную функцию, и все типы данных строго сохраняются. Нет необходимости писать вручную клиентские API вызовы или Swagger – меньше кода вообще. Codex, если он обучен на tRPC примерах, может помогать с написанием routers, процедур, особенно шаблонных (CRUD). TRPC хорошо интегрируется с React (через React Query), что AI тоже знает. *Минусы:* tRPC предполагает монолитный репозиторий (или по крайней мере общее описание) – если у вас будут сторонние клиенты (не на TypeScript, или мобильные), им будет сложнее. Как отмечают сами авторы, tRPC не лучший выбор, если API планируется публичным или для третьих лиц [34] – он больше для внутреннего сопряжения фронта и бэка [35]. Ещё нюанс: тестирование отдельных частей сложнее, т.к. логика тесно связана с описанием процедур. Codex здесь менее проверен – tRPC появился не так давно, возможно, он знает ранние версии. Возможны подсказки, уже устаревшие (нужно смотреть changelog). Если вы хотите суперсовременный стек и вас устраивает его ограничения, tRPC – классный выбор, и AI поможет, но будьте готовы вкладывать больше контроля (вдруг Copilot что-то не так понял). В контексте редактора, если он чисто веб и вы контролируете фронт и бэк, tRPC может ускорить dev, но если в перспективе сделать открытое API или мобильный клиент – Nest/Express с REST более универсальны.

Итого: Для максимально надёжного результата с AI, многие выбирают NestJS – он навязывает правильную архитектуру и дружелюбен к кодогенерации (много шаблонного кода) [20]. Express подойдёт, если у команды достаточно дисциплины и проект относительно прост, либо нужен минимальный оверхед – AI с ним справится, но структурирование ложится на людей. TRPC – для проектов, где full-stack TS (и, возможно, Next.js на фронте) – можно достичь высокой скорости разработки, однако требует определённой квалификации и осторожности с AI (следить за актуальностью рекомендаций, тестировать типы).

PostgreSQL vs SQLite vs MongoDB (база данных). База – сердце системы, и взаимодействие с ней тоже можно ускорять AI-ассистентом. Выбор БД влияет на характер работы:

- **PostgreSQL (реляционная БД):** *Плюсы:* Богатая функциональность, строгие схемы, транзакции, сложные запросы. Отлично подходит для данных с отношениями (у нас, например, пользователи, документы, разрешения – связи, которые удобно выразить SQL). PostgreSQL – open-source стандарт де-факто, Codex наверняка знает синтаксис SQL и типичные запросы [36] . Генерация кодовой логики с Postgres (через ORM или query builder) будет надёжной: AI напишет и миграции, и JOIN запрос, и оптимизации типа индексов, если попросить. *Минусы:* Некоторая сложность в настройке и эксплуатации, требовательность к ресурсам по сравнению с SQLite. Если модель данных часто меняется, схема требует миграций. AI может не учесть специфические Postgres моменты (например, что `SERIAL` устарел, лучше `GENERATED AS IDENTITY`). Но в целом, **для серьёзного проекта выбор PostgreSQL** обоснован: данные консистентны, можем использовать JSONB для хранения контента документов (иерархию можно там держать). Codex помогает писать сложные SQL без ошибок синтаксиса, но сложность запросов и оптимизацию всё равно надо проверять (он не всегда знает, как много данных у вас, индексы и т.д.).
- **SQLite (встраиваемая SQL-БД):** *Плюсы:* Очень простая – один файл, никаких серверов. Идеально для начальной стадии или прототипа. Можно быстро запускать, миграции проще (или вообще не нужны – просто файл поменял). Codex тоже знаком с SQLite, он пишет практически тот же SQL (за вычетом парочки особенностей). Отличный выбор для локальной разработки, тестов – можно даже AI попросить писать код так, чтобы в dev использовался SQLite, а в prod – Postgres (ORM это позволяет). SQLite быстр на небольших данных и при малой конкуренции, Codex может генерировать даже raw SQL для него, это не проблема – синтаксис почти ANSI. *Минусы:* Не предназначена для высоконагруженных многопользовательских сценариев (не очень справляется с множеством одновременных записей [37]). Отсутствуют некоторые фичи Postgres (сложные запросы, JSONB, расширения). То есть для прототипа можно, для production с десятками тысяч пользователей – скорее нет. С AI-стороны, всё ок: он знает её ограничения (например, нет полноценной поддержки параллельных транзакций), хотя вам нужно понимать, когда перейти с SQLite на что-то мощнее. Мы могли бы начать проект на SQLite, а по мере роста мигрировать на Postgres – Codex даже поможет с миграцией (можно попросить: *“Напиши скрипт миграции с SQLite на Postgres”*, он подготовит выгрузку/загрузку).
- **MongoDB (NoSQL документо-ориентированная БД):** *Плюсы:* Гибкая схема – документы JSON могут свободно меняться. Хорошо подходит для хранения структур типа дерева документов, поскольку можно хранить всю иерархию в одном JSON-документе (либо коллекции вложенных). Проста в начале – не надо определять схему жёстко. Масштабируется горизонтально относительно легко. Codex знаком с Mongo, Mongoose (ODM для Node) – он умеет генерировать схемы Mongoose, запросы find/update, агрегации. Для хранения содержимого блоков Notion-подобного редактора Mongo может быть привлекательна, так как контент – полу-структурированный JSON. *Минусы:* Отсутствие транзакционности на уровне, привычном в SQL (хотя мульти-док транзакции появились, но сложнее) [38] . Возможны несогласованные данные, если не продумать структуру. Сложнее делать сложные выборки (SQL JOIN по сути заменяется ручными операциями). Для AI: он может предложить неправильные Mongoose-запросы, или забыть `await` – надо тщательно тестировать. Модель данных гибкая, но это палка о двух концах: Codex может сохранять то один формат JSON, то другой, если нет строгости. В больших проектах Mongo требует такой же дисциплины, как Express: надо чётко договориться, какой формат документа, и проверять его. В нашем случае, MongoDB можно было бы использовать для хранения, например, **истории правок** или **логов**, а основную структуру (пользователи,

права) держать в SQL для целостности. Codex справится и с миксом (подключение двух БД), но это повышает сложность.

Итого: Для **масштабируемого многопользовательского приложения** PostgreSQL выглядит наиболее надёжно – строгие гарантии, сложные запросы, которые могут понадобиться (например, отчет "сколько документов у каждого пользователя"). Codex хорошо помогает с SQL, а типы (через Prisma/TypeORM) сохраняют безошибочность. SQLite – хорошо на старте или для офлайн-версии (скажем, локальный Electron-редактор). MongoDB – подходит, если вы предпочитаете гибкость и будете в основном работать с документами как с JSON. В частности, если ваша логика – в основном хранить и извлекать большой кусок документа (JSON) и редактирование происходит на клиенте, Mongo вполне. Codex же поможет с любой из них, вопрос в ваших требованиях к данным (схемы vs гибкость, консистентность vs скорость разработки). Кстати, никто не мешает гибридному подходу: AI может помочь связать, например, Postgres для одних данных и ElasticSearch (или даже SQLite) для других, если так решите.

Варианты AI-ассистентов: GitHub Copilot vs OpenAI Codex API vs другие. На момент 2025 года у разработчиков есть несколько способов использовать модель Codex/GPT для кодирования:

- **GitHub Copilot (в IDE):** Самый удобный вариант для большинства – прямо внутри VS Code / JetBrains. *Плюсы:* Мгновенные подсказки по мере набора кода, контекст – текущий файл и окружающие вкладки, минимальные усилия для интеграции. Обучен именно на продолжение кода, поэтому очень хорош в синтаксическом дополнении, учёте контекста файла [39]. Работает оффлайн (требует интернета для модели, но вы не покидаете IDE). Фиксированная подписка – не нужно думать о токенах и запросах [40]. *Минусы:* Контекстное окно ограничено – если проект очень большой, он не увидит всё, нужно самому открывать нужные части [16]. Иногда даёт несколько вариантов, нужно руками выбирать. Не умеет диалог долговременный (без Copilot Chat). Не интегрируется в произвольные сценарии (например, сгенерировать код вне IDE). В целом, Copilot – оптимален для 90% задач описанных выше. Он ускоряет именно "в потоке" разработки и не требует переключения контекста [41].
- **OpenAI API (Codex или ChatGPT models):** Это вариант программно вызывать модели. *Плюсы:* Гибкость – можно написать скрипт, который берет ваш код, отправляет на модель с определённым запросом и получает изменённый код (например, автономный рефакторинг). Можно накормить модель большим контекстом (до 16k или 32k токенов на GPT-4) – т.е. она увидит целый файл или несколько. Можно делать нестандартные запросы: генерация документации, анализ репозитория, даже управлять моделью как частью CI (например, проверка кода). *Минусы:* Нужно самому писать такие интеграции, нет "магии" автодополнения – вы формулируете запросы явно. Это медленнее по итерации, т.к. вы ждёте ответа от API. Также учет стоимости – API платное по токенам, если проект крупный, можно потратить существенную сумму, если часто обращаться. В общем, API хорош для разовых задач (сгенерировать большой кусок или проанализировать код) [42] [43]. Но для повседневного "набора кода" Copilot удобнее. Некоторые используют связку: Copilot – при написании кода, а ChatGPT – при обсуждении архитектуры или получении объяснений, т.е. как чат-консультант.
- **Copilot Chat (в IDE) и аналоги:** Новые инструменты, как Copilot Chat, combine удобство IDE и диалоговые возможности. *Плюсы:* Можно выделить код в редакторе и спросить "что это делает?" или "оптимизируй", и сразу получить ответ/правку. Это сочетает лучшее от API-чатов с знанием вашего контекста (у Copilot Chat доступ к вашему открытому коду). *Минусы:* Пока относительно новое, может быть сыровато. Кроме того, без привычки можно начинать злоупотреблять чат-режимом даже там, где проще написать самому. Но в общем,

для нашего сценария – отличная вещь: вы можете общаться с AI зная, что он “видит” проект (в рамках открытых файлов) и может давать совет по месту.

- **Альтернативные AI-кодеры:** Это Tabnine, Amazon CodeWhisperer, Code Llama (от Meta) и др. *Плюсы:* Иногда дешевле или с большим упором на приватность (CodeWhisperer, например, не посылает код наружу, если настроить, но обучен похуже на публичных данных). Code Llama можно запустить локально, что важно для закрытых проектов. *Минусы:* По качеству генерации пока уступают Copilot. Copilot/Codex получили огромное преимущество за счет обучения на GitHub и поддержки Microsoft. Например, CodeWhisperer от AWS хорош в AWS-коде, но в общем JS/TS может быть слабее. Если ваш проект – абсолютно закрытый код и нельзя рисковать отправлять его на внешние сервера, тогда либо локальные модели, либо ограничение Copilot (у них есть режимы без сбора данных). Но это компромисс: локальные пока слабее.

Рекомендация по инструменту: GitHub Copilot (IDE) + периодически ChatGPT (веб или Copilot Chat) – это, пожалуй, оптимум. Copilot берёт на себя рутину письма, ChatGPT – отвечает на концептуальные вопросы и генерирует большие тексты (документацию, обзоры). Открытый Codex API может не понадобиться напрямую, разве что для каких-то custom-скриптов (скажем, вы хотите автоматически генерировать на основе шаблона много кода).

Стоимость: Copilot – ~10$ в месяц фиксировано, OpenAI API – pay-as-you-go. Для большого проекта постоянное использование API может выйти дороже. Зато API можно масштабировать (встроить в свои инструменты). Выбор зависит от того, как вы работаете: если постоянно в IDE – Copilot выгоднее. Если хотите автоматизировать документацию или сделать своего бота, интегрированного в репо – тогда API.

Вывод по стеку: Для проекта “многопользовательский AI-редактор” мы бы выбрали: **NestJS + PostgreSQL** как основу – это даёт надёжность и структурность, что облегчит и самому Codex работу (он будет следовать устоявшимся паттернам) [44]. Если акцент на быстром прототипе – можно начать с Express или tRPC, SQLite, потом при необходимости мигрировать. AI поможет на каждом этапе, но миграцию надо будет контролировать (можно попросить AI переписать код под NestJS, но лучше руками с его подсказками).

Помните, что стек – инструмент для вас и Codex. **Выбирайте тот, в котором вы чувствуете себя комфортно,** потому что AI восполнит пробелы, но основную работу всё равно orchestrate вам. Если вы отлично знаете MongoDB и не любите SQL – Mongo может быть предпочтительней. Codex и с ним работает [45] [46]. Если вы цените строгий фреймворк – Nest. Если любите minimalism – Express. Главное, учитывайте масштаб: для серьёзной нагрузки лучше типизация и структуры (Nest/Postgres), для легковесного стартапа – можно быстрее начать с Express/SQLite.

Заключение. Мы рассмотрели практически все аспекты разработки full-stack TypeScript приложения с помощью OpenAI Codex/GitHub Copilot – от эффективного промптинга и поддержания архитектуры до внедрения конкретных фич и выбора стека. Подытоживая, **Codex – мощный ускоритель, способный кратно повысить продуктивность** разработки [47] [30], но использовать его нужно осознанно. В сложных проектах, подобных «редактору с AI-помощником», человеческий фактор (дизайн системы, контроль качества, творческое решение проблем) остаётся ключевым. AI берёт на себя рутину и частично роль консультанта, но не заменяет инженера [2]. Правильно организовав процесс, вы можете двигаться очень быстро (быстрее конкурентов, возможно) и при этом сохранять высокое качество и контроль. Надеемся, что практики и советы из этого исследования помогут вам уверенно интегрировать Codex в ваш

workflow и построить отличный продукт, сочетая лучшее из опыта человека и возможностей искусственного интеллекта.

Источники: Использованы материалы GitHub Docs [3] [26], экспертные статьи и FAQ по Copilot [11] [8], а также сравнительные обзоры технологий (NestJS, tRPC) [20] [21] и базы данных [25] [46] для полноты и достоверности советов.

---

[1] [2] [3] [5] [6] [7] [10] [16] [18] [26] [31] Best practices for using GitHub Copilot - GitHub Docs
https://docs.github.com/en/copilot/get-started/best-practices

[4] [9] [24] [27] [30] [47] Generative AI in Code: How Codex, AlphaCode, and GitHub Copilot Are Transforming Development -Reckonsys
https://www.reckonsys.com/blogs/generative-ai-in-code-how-codex-alphacode-and-github-copilot-are-transforming-development/

[8] [11] [12] [13] [17] [22] [28] [29] GitHub Copilot: Dos and Don'ts
https://www.dataart.team/articles/github-copilot-dos-and-donts

[14] [15] 10 Advanced Github Copilot tips & tricks | Copilot best practices
https://www.coderabbit.ai/blog/github-copilot-best-practices-10-tips-and-tricks-that-actually-help

[19] AI-powered code generation: A deep dive into GitHub Copilot | Thoughtworks United States
https://www.thoughtworks.com/en-us/insights/blog/generative-ai/ai-powered-code-generation-deep-dive-into-github-copilot

[20] [21] [33] [34] [35] [44] Picking a Node.js Backend Framework in 2023
https://janejeon.dev/picking-a-node-js-backend-framework-in-2023/

[23] Retrieval Augmented Generation (RAG) and Semantic Search for GPTs | OpenAI Help Center
https://help.openai.com/en/articles/8868588-retrieval-augmented-generation-rag-and-semantic-search-for-gpts

[25] [32] [36] [37] [38] [45] [46] Pros and Cons: MySQL, PostgreSQL, SQLite, MongoDB
https://skynix.co/resources/pros-and-cons-mysql-postgresql-sqlite-mongodb

[39] [40] [41] [42] [43] OpenAI Codex vs GitHub Copilot: Comparing AI Code Assistant
https://www.zignuts.com/blog/openai-codex-vs-github-copilot-comparison