CODEX

Codex CLI: понятная дорожная карта по установке, настройке и работе с агентом OpenAI

Пошаговое руководство для человека, который хочет научиться работать с Codex в терминале: от первых команд до настройки безопасной агентной среды.

О документе

Этот документ — практическое руководство по Codex CLI. Его задача — не просто перечислить команды или пересказать документацию, а дать рабочую ментальную модель: как устроен Codex, где лежат его настройки, как он принимает решения, какие инструкции читает, какие права имеет и как правильно управлять его поведением.

Codex CLI нужно понимать не как обычный чат с моделью, а как локальную агентную систему. Он запускается в терминале, работает в контексте выбранной директории, может читать проект, менять файлы, запускать команды, проверять результат и продолжать работу итерациями.

Главная ценность этого руководства — связать все ключевые части Codex в одну понятную карту:

- установка и первый запуск;
- Авторизация;
- Работа в терминале;
- Структура Codex;
- Глобальная и проектная конфигурация;
- `config.toml` Настройка агента;
- `AGENTS.md` Проектные инструкции;
- Skills Загружаемые скиллы;
- Разрешенный рабочий контур и права;
- Настройка и выбор моделей;
- Автономно длительные задачи (рекомендации);
- MCP, Субагенты, Хуки;
- готовые шаблоны для практического использования.

Для кого это руководство

Для новичков

Если вы только начинаете работать с терминалом, этот документ поможет понять базовые вещи простым языком:

- что такое CLI;
- куда вводить команды;
- где лежат настройки Codex;
- почему важна папка, из которой вы запускаете Codex;
- что значит “агент может менять файлы”;
- как не включить опасные режимы случайно.

Для разработчиков

Если вы уже работаете с Git, проектами, зависимостями и терминалом, руководство поможет использовать Codex как инженерный инструмент:

- настроить `config.toml`;
- создать проектный `AGENTS.md`;
- описать правила разработки;
- настроить безопасный режим работы;
- запускать Codex для code review, bugfix, refactoring и генерации тестов;
- диагностировать проблемы через `/status`, `/debug-config` и логи.

Для тимлидов и архитекторов

Если вы хотите внедрить Codex в командную работу, документ поможет спроектировать устойчивую агентную среду:

- разделить глобальные и проектные настройки;
- описать правила проекта через `AGENTS.md`;
- создать Skills для повторяемых workflow;
- ограничить права через sandbox и approvals;
- подключить внешние инструменты через MCP;
- организовать long-horizon задачи через внешнюю память проекта.

Для инженеров AI-агентов

Если вы проектируете собственные агентные системы, Codex можно рассматривать как пример практической агентной архитектуры:

- модель;
- агентный цикл;
- инструментальный слой;

- конфигурационные слои;
- постоянные инструкции;
- навыки;
- внешние инструменты;
- управление контекстом;
- безопасность;
- диагностика.

---

1. Самое важное перед стартом: терминал

Что такое терминал

Терминал — это окно, куда человек вводит текстовые команды для компьютера.

Обычно мы управляем компьютером мышкой: открываем папки, нажимаем кнопки, запускаем программы. В терминале всё делается словами-командами.

Пример команды:

```
pwd
```

Она означает: “покажи, в какой папке я сейчас нахожусь”.

Другой пример:

```
ls
```

Она означает: “покажи файлы и папки здесь”.

Ещё пример:

```
cd my-project
```

Она означает: “перейди в папку `my-project` или любую другую папку `мои_семейные_фото`”.

Если вы не знаете, где находится терминал на вашем компьютере, можно воспользоваться отдельным ИИ-помощником:
👉 https://operatorvlm.ru 👈
Он поможет подобрать команды под вашу операционную систему.

---

2. Что такое Codex

Codex CLI — это ИИ помощник для работы с кодом, который запускается в терминале.

Он может:

- читать файлы проекта;
- понимать структуру кода;
- искать ошибки;
- предлагать изменения;
- изменять файлы;
- запускать команды проверки;
- исправлять ошибки после проверки;
- работать по правилам проекта;
- использовать заранее заданные навыки.

Обычный чат с ИИ работает так:

```
Вы задали вопрос → ИИ написал ответ.
```

Codex работает иначе:

```
Вы поставили задачу → Codex изучил проект → внёс изменения → проверил результат →
сообщил итог.
```

Поэтому Codex лучше воспринимать не как “чат”, а как “агента”.

Что значит “агент”

Агент — это система, которая может не только отвечать, но и действовать.

В случае Codex действиями могут быть:

- открыть файл;
- прочитать код;
- изменить файл;
- выполнить команду;
- посмотреть ошибку;
- исправить ошибку;
- повторить проверку.

Если прям кратко - Codex - это ваш личный программист, которые работает вместо вас на вашем компьютере

3. Базовые слова, без которых Codex будет казаться сложным

Перед установкой нам нужно договориться о словах. Это важно, потому что большая часть страха возникает не из-за сложности инструмента, а из-за незнакомых терминов.

3.1 Команда

Команда — это текстовая инструкция компьютеру. Пишется в терминале

Пример:

```
codex --version
```

Смысл:

> “Покажи версию установленного Codex”.

3.2 Папка

Папка — место, где лежат файлы и другие папки.

В терминале папки называют директориями. Это одно и то же.

Здесь будем чаще писать “папка”, а если встречается слово “директория”, считайте его техническим синонимом.

3.3 Путь

Путь — это адрес файла или папки в файловой системе компьютера.

Пример:

```
~/.codex/config.toml
```

Это путь к файлу настроек Codex.

Разберём его по частям:

```
~              домашняя папка (Не пугайтесь черточки – позже будет понятнее)
.codex         скрытая папка Codex (Точка стоит в начале имени – это нормально.)
config.toml    файл настроек
/              Это разделитель между папками
```

3.4 Домашняя папка

Домашняя папка — личная папка пользователя на компьютере.

В терминале она часто обозначается символом:

```
~
```

Например:

```
~/.codex
```

означает:

> папка `.codex` внутри домашней папки пользователя.

3.5 Скрытая папка

Если имя папки начинается с точки, например:

```
.codex
```

то такая папка обычно скрыта в обычном файловом менеджере (то есть глазами ее нет, но в компьютере она есть).

Но терминал её видит.

Команда, которая показывает и обычные, и скрытые файлы:

```
ls -la
```

3.6 Проект

Проект — это папка с кодом, файлами приложения, настройками, документацией и, часто, Git-историей.

Пример:

```
my-app/
├── package.json
├── src/
├── README.md
└── .git/
```

3.7 Корень проекта

Корень проекта — главная папка проекта. То есть если вы перешли внутрь необходимой папки - это корень вашего проекта)

Например:

```
my-app/
```

Если внутри неё лежат `src/`, `README.md`, `.git/` (*не важно какие папки это для примера*) то именно `my-app/` чаще всего является корнем проекта.

Codex пытается понять корень проекта автоматически. Обычно он ориентируется на существование папки `.git`.

3.8 Файл настроек

Файл настроек Codex'а обычно называется:

```
config.toml
```

Главный пользовательский файл настроек обычно лежит здесь:

```
~/.codex/config.toml
```

Это файл внутри которого в структурированном виде расписаны параметры работы агента:

- Выбор модели
- Сила рассуждений
- Уровень доступа
- и прочее (разберем позже)

3.9 Инструкция - промпт

Промпт — это правило для Codex'а, как ему работать.

Разовая инструкция (промпт) — это то, что вы пишете в сообщении:

```
Исправь ошибку авторизации и запусти тесты.
```

Постоянная инструкция — это файл:

```
AGENTS.md
```

В нём можно записать правила проекта, которые Codex должен учитывать каждый раз. (Разберем позже)

3.10 Песочница

Песочница — это ограниченная область, в которой Codex может работать. (Специальные ограничения агента, чтобы он не отправил друзьям ваши фоточки без разрешения)

Технический термин:

```
sandbox
```

В этом документе будем чаще писать “песочница”, а рядом иногда указывать техническое имя `sandbox`. Она пригодится нам чуть позже для того чтобы ограничить возможности агента (ДА такое нужно!)

3.11 Скилл

Скилл — это заранее подготовленная инструкция для повторяемой работы. Специально загружаемый модуль внутрь агента, который расширяет возможности агента специализированными знаниями.

Технический термин:

```
Skill
```

Например, можно создать навык:

```
code-review-workflow
```

и описать в нём, как Codex должен делать проверку кода.

3.12 Подтверждение

Подтверждение — это момент, когда Codex перед действием спрашивает разрешение пользователя.Технический термин:

```
approval
```

Например, если Codex хочет выполнить чувствительную команду, он может сначала спросить:

> Разрешить выполнение?

За это отвечает настройка:

```
approval_policy = "on-request"
```

НА ЭТОМ БАЗОВЫЕ ТЕРМИНЫ ЗАКОНЧИЛИСЬ, МОЖНО ВЫДОХНУТЬ. Не нужно их сейчас учить - далее будет понятнее)

---

4. Главная карта Codex

Чтобы не запутаться, держите в голове простую схему того как работает наш агент:
(Функционально)

```
Вы пишете задачу
        ↓
Codex читает инструкции
        ↓
Codex смотрит настройки
        ↓
Codex проверяет свои права
        ↓
Codex работает с файлами и командами
        ↓
Codex показывает результат
```

Теперь та же схема с файлами (какие файлы за что отвечают):
(Навигационно)

```
Ваш промпт (одноразовая инструкция)
    ↓
AGENTS.md                постоянные инструкции (Разберем далее...)
    ↓
config.toml              настройки поведения (Разберем далее...)
    ↓
sandbox / approvals      ограничения и подтверждения (Разберем далее...)
    ↓
папка проекта
(ваша выбранная папка)   файлы, с которыми работает Codex
```

---

5. Установка Codex

Что нужно до установки

Перед установкой нужно понять три вещи:

1. Какая у вас операционная система.
2. Есть ли у вас терминал.

3. Установлены ли `Node.js` и `npm`.

Codex устанавливается и запускается через терминал. Поэтому сначала важно не сам Codex настраивать, а убедиться, что компьютер готов выполнять команды.

Операционная система

Возможные варианты:

- macOS;
- Linux;
- Windows;
- Windows через WSL.

WSL — это Linux-среда внутри Windows. Если вы не знаете, что это такое, не страшно. Просто считайте, что это отдельный способ работать с Linux-командами на Windows.

Для Windows чаще всего удобнее использовать именно WSL, потому что Codex — терминальный инструмент, а многие команды разработки лучше работают в Linux-среде.

5.1 Как открыть терминал

Если вы уже умеете открывать терминал — переходите дальше.

Если нет:

macOS

Нажмите:

```
Cmd + Space
```

Введите:

```
Terminal
```

Нажмите `Enter`.

Linux

Обычно работает сочетание:

```
Ctrl + Alt + T
```

Windows

Откройте PowerShell:

```
Win → PowerShell → Enter
```

Если используете WSL, откройте установленную Linux-среду, например `Ubuntu`.

Если вы не знаете, как открыть терминал на вашем компьютере, перейдите в чат с ИИ-помощником и задайте вопрос:

```
Как открыть терминал на моей операционной системе?
```

5.2 Проверка Node.js и npm

Для установки Codex через `npm` нужны две программы:

- `Node.js`;
- `npm`.

`Node.js` — это среда, которая нужна для работы многих инструментов разработки.

`npm` — это установщик пакетов. Через него можно установить Codex.

Откройте терминал и выполните:

```
node -v
npm -v
```

Если вы видите версии, например:

```
v22.0.0
10.0.0
```

значит `Node.js` и `npm` установлены.

Если появляется ошибка вроде:

```
command not found: node
```

или:

```
npm is not recognized
```

значит `Node.js` и `npm` не установлены.

В этом случае сначала установите **Node.js LTS** с официального сайта Node.js.
После установки закройте терминал, откройте его заново и снова проверьте:

```
node -v
npm -v
```

5.3 Установка Codex через npm

Если `node -v` и `npm -v` работают, установите Codex командой:

```
npm i -g @openai/codex
```

Расшифровка:

```
npm             программа для установки пакетов Node.js
i               install, то есть “установить”
-g              глобально, то есть для всей системы пользователя
@openai/codex   пакет Codex CLI
```

После установки проверьте:

```
codex --version
```

Если версия выводится, Codex установлен.

5.4 Установка Codex на macOS через Homebrew

Если у вас macOS и установлен Homebrew, можно установить Codex так:

```
brew install --cask codex
```

Проверка:

```
codex --version
```

Если команда выводит версию, Codex установлен.

Если Homebrew не установлен

Проверьте:

```
brew --version
```

Если терминал пишет, что `brew` не найден, Homebrew нужно установить.

Команда установки Homebrew:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

После установки закройте терминал, откройте его снова и проверьте:

```
brew --version
```

Затем установите Codex:

```
brew install --cask codex
```

5.5 Установка на Windows

На Windows есть два варианта.

Вариант 1. Через WSL

Рекомендуемый вариант.

Схема такая:

```
Windows → WSL → Node.js/npm → Codex
```

То есть Codex устанавливается не в обычный Windows-терминал, а внутрь Linux-среды WSL.

После открытия WSL проверьте:

```
node -v
npm -v
```

Если Node.js и npm установлены, выполните:

```
npm i -g @openai/codex
```

Проверка:

```
codex --version
```

Вариант 2. Напрямую в Windows

Иногда Codex можно установить через обычный PowerShell:

```
npm i -g @openai/codex
```

Но если вы новичок, лучше использовать WSL. Так меньше проблем с путями, командами и средой разработки.

5.6 Первый запуск после установки

После установки выполните:

```
codex
```

При первом запуске Codex предложит авторизацию.

Обычно есть два варианта:

- войти через ChatGPT-аккаунт;
- использовать API-ключ.

Если вы не знаете, что выбрать, начните с входа через ChatGPT-аккаунт.

---

6. Первый запуск

6.1 Создадим безопасную тренировочную папку

Чтобы не бояться за настоящий проект, начнём с отдельной пустой папки. В ней Codex сможет работать без риска для ваших рабочих файлов.

В терминале выполните:

```
mkdir codex-training
cd codex-training
```

Что делают эти команды:

```
mkdir codex-training    создаёт папку с именем codex-training
cd codex-training       переходит внутрь этой папки
```

Теперь вы находитесь внутри тренировочной папки. Это безопасное место для первого запуска.

Проверить, где вы сейчас находитесь, можно командой:

```
pwd
```

Она покажет путь к текущей папке.

6.2 Почему важно, из какой папки запущен Codex

Это один из главных принципов работы с Codex.

Codex не работает “где-то вообще”. Он всегда запускается из конкретной папки.

Например:

```
cd codex-training
codex
```

Это значит:

1. вы перешли в папку `codex-training`;
2. запустили Codex именно оттуда;
3. Codex будет воспринимать эту папку как рабочую область.

Рабочая область — это место, где Codex смотрит файлы, создаёт новые файлы и выполняет задачи.

Если запустить Codex из другой папки, он будет видеть другой набор файлов.

Например:

```
cd ~/Desktop
codex
```

и:

```
cd ~/projects/my-app
codex
```

это два разных запуска. В первом случае Codex работает на рабочем столе, во втором — внутри проекта `my-app`.

Поэтому первый вопрос при любой непонятной ситуации:

> Из какой папки был запущен Codex?

Команда для проверки:

```
pwd
```

6.3 Запускаем Codex

Когда вы находитесь внутри тренировочной папки, выполните:

```
codex
```

После этого откроется интерфейс Codex в терминале.

Если Codex запускается впервые, он может попросить войти в аккаунт или выбрать способ авторизации. Если вход уже выполнен, вы сразу попадёте в рабочий экран Codex.

6.4 Как пользоваться Codex в терминале

После запуска Codex вы работаете с ним прямо в терминале.

Основная логика простая:

```
вы пишете задачу → нажимаете Enter → Codex начинает думать и действовать
```

Отправить сообщение

Обычно сообщение отправляется клавишей:

```
Enter
```

Например, вы пишете:

```
Изучи эту папку и объясни, какие файлы здесь есть. Ничего не меняй.
```

Затем нажимаете `Enter`.

Остановить выполнение

Если Codex начал выполнять задачу, а вы хотите остановить процесс, обычно используется:

```
Ctrl + C
```

Важно для macOS:

> В терминале используется именно `Ctrl + C`, а не `Cmd + C`.

На macOS `Cmd + C` чаще отвечает за копирование, а не за остановку команды.

Выйти из Codex

Чаще всего можно использовать:

```
Ctrl + C
```

Если Codex попросит подтверждение выхода, подтвердите его.

Также внутри Codex можно попробовать команду:

```
/quit
```

или:

```
/exit
```

Если конкретная команда не поддерживается вашей версией, используйте `Ctrl + C`.

Посмотреть доступные команды

Внутри Codex можно ввести:

```
/
```

Обычно после этого появляется список доступных команд.

Полезные команды:

```
/status       показать текущий режим и состояние
/permissions  посмотреть или изменить разрешения
/model        выбрать модель
/diff         посмотреть изменения
/review       проверить изменения
/help         справка, если доступна
/keymap       клавиши управления, если доступна
```

Если вы не уверены, какие клавиши работают именно в вашей версии Codex, используйте:

```
/keymap
```

Эта команда нужна, чтобы посмотреть актуальные сочетания клавиш внутри Codex.

Вернуться к прошлым сообщениям

В терминальных программах часто используется клавиша `esc`:

```
Esc    предыдущее сообщение или команда
 ->    следующее сообщение или команда
```

Но поведение может зависеть от версии Codex и вашего терминала. Если стрелки не работают так, как вы ожидали, это не ошибка с вашей стороны.

Очистить ошибочно набранный текст

Если вы начали писать сообщение и хотите быстро стереть строку, часто работает:

```
Ctrl + U
```

Это очищает текущую строку ввода в терминале.

6.5 Первая безопасная задача

Внутри Codex напишите:

```
Изучи эту папку и объясни, какие файлы здесь есть. Ничего не меняй.
```

Это безопасная первая задача, потому что вы прямо запретили изменять файлы.

Codex должен посмотреть текущую папку и объяснить, что в ней находится.

Если папка пустая, он так и скажет. Это нормально.

6.6 Первая задача с созданием файла

Теперь можно попросить Codex сделать маленькое безопасное действие:

```
Создай файл README.md и напиши в нём короткое описание этой тренировочной папки.
```

После этого Codex может создать файл `README.md`.

Проверить это можно в терминале командой:

```
ls
```

Если файл появился, вы успешно запустили Codex и дали ему первую практическую задачу.

6.7 Что вы только что сделали

Вы:

1. открыли терминал;
2. создали отдельную безопасную папку;
3. перешли внутрь неё;
4. запустили Codex;
5. дали ему первую задачу;
6. убедились, что он работает внутри выбранной папки.

Поздравляю! Вы успешно запустили личного программиста. Теперь он работает внутри вашей папки и может создавать внутри нее разные скрипты: сайты, игры и прочее.

---

7. Структура Codex

В этом разделе описаны важные файлы и папки, которые влияют на стабильную работу Codex.

Codex — это не только команда:

```
codex
```

После установки у него появляются свои файлы настроек, авторизации, истории, логов и инструкций. Часть из них лежит в главной папке Codex, а часть можно создать внутри конкретного проекта.

Главная мысль этого раздела:

> Codex работает лучше и предсказуемее, когда его файлы разложены по правильным местам.

7.1 Главная папка Codex

Обычно Codex хранит свои пользовательские служебные файлы здесь:

```
~/.codex
```

Символ `~` означает домашнюю папку пользователя.

То есть путь:

```
~/.codex
```

можно понимать так:

```
домашняя папка пользователя → папка .codex
```

Эта папка может содержать:

```
~/.codex/
├── config.toml
├── auth.json
├── history.jsonl
├── log/
├── AGENTS.md
├── AGENTS.override.md
├── hooks.json
└── rules/
```

Не обязательно, что все эти файлы появятся сразу после установки. Некоторые создаются только после авторизации, настройки истории, логов, инструкций или дополнительных возможностей.

Проверить содержимое папки можно командой:

```
ls -la ~/.codex
```

Если папки ещё нет, её можно создать:

```
mkdir -p ~/.codex
```

7.2 `config.toml`

`config.toml` — это главный файл настроек Codex.

Обычно он находится здесь:

```
~/.codex/config.toml
```

В этом файле можно указать:

- какую модель использовать;
- когда Codex должен спрашивать разрешение;
- может ли Codex менять файлы;
- какой режим безопасности использовать;
- есть ли доступ к сети для команд;
- сохранять ли историю;
- какие профили использовать;
- какие внешние инструменты подключать.

Пример простого файла настроек:

```
approval_policy = "on-request"
sandbox_mode = "workspace-write"

[history]
persistence = "save-all"
```

Что это значит:

```
approval_policy = "on-request"
Codex будет спрашивать разрешение в чувствительных случаях.

sandbox_mode = "workspace-write"
```

```
Codex сможет менять файлы в рабочей папке проекта.

history.persistence = "save-all"
История сессий будет сохраняться.
```

Если файла `config.toml` нет, его можно создать вручную:

```
nano ~/.codex/config.toml
```

Подробнее разберем в отдельной главе --> 13. Подробно про настройку агента

7.3 `auth.json`

`auth.json` — это файл, связанный с авторизацией Codex.

Он может появиться после входа в аккаунт или настройки доступа.

Путь:

```
~/.codex/auth.json
```

Обычно этот файл не нужно открывать и редактировать руками.

Важно:

> Не публикуйте `auth.json`, не отправляйте его другим людям и не добавляйте его в Git.

Причина простая: файл может быть связан с доступом к вашему аккаунту или ключам авторизации.

7.4 `history.jsonl`

`history.jsonl` — это файл истории сессий Codex.

Путь:

```
~/.codex/history.jsonl
```

Если история включена, Codex может сохранять туда данные о прошлых сессиях.

Обычно история включена по умолчанию или включается через настройки.

Пример настройки:

```
[history]
persistence = "save-all"
```

Если вы не хотите сохранять историю, можно указать:

```
[history]
persistence = "none"
```

Важно понимать:

> История помогает продолжать работу и разбирать прошлые действия, но на общих или чужих компьютерах её лучше отключать.

7.5 `log/`

`log/` — это папка с логами Codex.

Путь:

```
~/.codex/log/
```

Логи — это технические записи о работе программы.

Они нужны, когда что-то работает непонятно. Например:

- Codex не подхватил инструкции;
- Codex использует не те настройки;
- Codex странно завершает работу;
- нужно понять, какие файлы и правила были загружены;
- нужно диагностировать проблему с запуском.

Обычно пользователю не нужно читать логи каждый день. Но если что-то сломалось, папка `log/` становится важной.

7.6 `AGENTS.md`

`AGENTS.md` в папке `~/.codex` — это глобальная инструкция для Codex.

Путь:

```
~/.codex/AGENTS.md
```

Глобальная инструкция может применяться в разных проектах.

Например, туда можно записать личные правила:

```
Всегда сначала объясняй план перед крупными изменениями.
Не выполняй опасные команды без разрешения.
После изменения файлов кратко перечисляй, что было изменено.
```

Важно:

> Глобальный `AGENTS.md` — это не инструкция конкретного проекта, а ваши личные общие правила для Codex.

Для правил конкретного проекта лучше использовать `AGENTS.md` внутри папки проекта.

Подробнее разберем в главе --> 10. Подробнее про `AGENTS.md`

7.7 `AGENTS.override.md`

`AGENTS.override.md` — это файл переопределения инструкций.

Путь:

```
~/.codex/AGENTS.override.md
```

Если рядом есть и `AGENTS.md`, и `AGENTS.override.md`, файл `AGENTS.override.md` может иметь приоритет на этом уровне.

Простыми словами:

```
AGENTS.md           обычная инструкция
AGENTS.override.md  инструкция, которая заменяет обычную на этом уровне
```

Новичку этот файл обычно не нужен. Его используют, когда нужно временно или явно заменить набор инструкций.

7.8 `hooks.json` и `rules/`

Эти элементы относятся к продвинутой настройке Codex.

```
~/.codex/hooks.json
~/.codex/rules/
```

`hooks.json`

`hooks.json` — это файл для автоматических действий на определённых этапах работы Codex.

Например, в продвинутых сценариях можно настраивать проверки, которые запускаются до или после некоторых действий.

`rules/`

`rules/` — это папка для правил выполнения команд.

Например, можно точнее управлять тем, какие команды Codex может выполнять автоматически, какие требуют подтверждения, а какие запрещены.

На первом этапе эти файлы можно не трогать.

Главное сейчас знать:

> `hooks.json` и `rules/` нужны для продвинутого контроля поведения Codex. Для базовой работы достаточно понимать `config.toml`, `AGENTS.md`, историю и логи.

---

8. Какие файлы нужно создать для правильной работы

После установки Codex уже может работать. Но если вы хотите, чтобы он работал стабильнее и лучше понимал ваш проект, полезно создать несколько дополнительных файлов внутри проекта.

Типовая структура:

```
my-project/
├── AGENTS.md
├── .codex/
│   └── config.toml
└── .agents/
    └── skills/
```

Здесь:

```
my-project/        ваша папка проекта
AGENTS.md          правила проекта для Codex
```

```
.codex/config.toml настройки Codex только для этого проекта
.agents/skills/    навыки для повторяемых задач
```

Не обязательно создавать всё сразу.

Минимально полезный файл для проекта:

```
AGENTS.md
```

Остальное можно добавлять позже, когда появится необходимость.

8.1 `AGENTS.md`

`AGENTS.md` — это файл с правилами для Codex внутри конкретного проекта.

Он помогает Codex понять:

- что это за проект;
- какие папки важны;
- как запускать проверки;
- какие команды использовать;
- какие файлы нельзя менять;
- какие правила разработки соблюдать;
- что считать завершённой задачей.

Сейчас рассмотрим упрощённо. Более детально `AGENTS.md` будет разобран в отдельной главе.

Пример содержимого файла:

````
# AGENTS.md

## Описание проекта

Это backend-сервис на Python.

## Команды проверки

После изменения backend-кода запускай:

```bash
pytest
```

## Ограничения
````

```
Не добавляй новые зависимости без отдельного объяснения.

Не меняй публичный API без согласования.

## Критерии готовности

Задача считается завершённой, когда:

- изменения внесены;
- тесты проходят;
- в ответе перечислены изменённые файлы;
- кратко объяснено, что было исправлено.
```

Этот файл лучше положить в корень проекта:

```
my-project/AGENTS.md
```

Корень проекта — это главная папка проекта.

Например:

```
my-project/
├── AGENTS.md
├── src/
├── tests/
└── README.md
```

8.2 `.codex/config.toml`

`.codex/config.toml` — это настройки Codex только для конкретного проекта.

Путь:

```
my-project/.codex/config.toml
```

Этот файл нужен, если для одного проекта нужны особые настройки.

Например:

- в одном проекте Codex может только читать файлы;
- в другом проекте ему можно изменять код;
- в третьем проекте нужно включить более строгие подтверждения;

- в четвёртом проекте нужны особые профили или внешние инструменты.

Пример проектной настройки:

```
approval_policy = "on-request"
sandbox_mode = "workspace-write"
```

Важно:

> Проектный `.codex/config.toml` не заменяет полностью глобальные настройки, а участвует в системе слоёв.

Также важно:

> Проектные настройки могут не применяться, если проект не считается доверенным.

Доверенный проект — это проект, которому вы разрешаете использовать локальные настройки Codex.

Если вы работаете с чужим или непроверенным проектом, не стоит сразу делать его доверенным.

8.3 `.agents/skills/`

`.agents/skills/` — это папка для скиллов Codex.

Путь:

```
my-project/.agents/skills/
```

скилл — это отдельная инструкция для повторяемой работы.

Например, если вы часто просите Codex делать проверку кода, можно создать навык:

```
code-review-workflow
```

Тогда структура будет такой:

```
my-project/.agents/skills/code-review-workflow/
└── SKILL.md
```

`SKILL.md` — главный файл навыка. В нём описывается:

- когда использовать навык;

- какие шаги выполнять;
- какие проверки запускать;
- какой результат вернуть пользователю;
- какие ограничения соблюдать.

Пример пути:

```
my-project/.agents/skills/code-review-workflow/SKILL.md
```

Сейчас достаточно понимать общую идею:

> `AGENTS.md` задаёт правила проекта, а Skill задаёт отдельный повторяемый рабочий сценарий.

Например:

```
AGENTS.md
Правила всего проекта.

Skill
Инструкция для конкретной задачи: ревью кода, исправление бага, написание тестов.
```

8.4 Что создавать сначала

Если вы только начинаете, не нужно создавать всё сразу.

Рекомендуемый порядок:

```
1. Сначала установить Codex.
2. Потом создать AGENTS.md в проекте.
3. Только после этого добавлять .codex/config.toml для проекта.
4. Skills создавать позже, когда появятся повторяемые задачи.
```

Минимальный набор для спокойной работы:

```
my-project/AGENTS.md
```

Расширенный набор:

```
~/.codex/config.toml
my-project/AGENTS.md
```

```
my-project/.codex/config.toml
my-project/.agents/skills/
```

---

9. Подробнее про `AGENTS.md`

9.1 Что это такое

`AGENTS.md` — это файл с постоянными правилами для Codex.

Он нужен, чтобы не повторять одни и те же инструкции в каждом запросе.

Например, вместо того чтобы каждый раз писать:

```
После изменения backend запусти pytest.
Не добавляй зависимости без согласования.
Не меняй публичный API.
Перед крупным изменением сначала покажи план.
```

можно один раз записать эти правила в `AGENTS.md`.

После этого Codex сможет учитывать их при работе в проекте.

Простыми словами:

> `AGENTS.md` — это памятка для Codex о том, как правильно работать именно с вашим проектом.

9.2 Зачем нужен `AGENTS.md`

Без `AGENTS.md` Codex видит только текущий запрос и файлы проекта. Он может сам догадаться о структуре, но не всегда будет понимать ваши внутренние правила.

Например, Codex может не знать:

- какие команды проверки нужно запускать;
- какие папки самые важные;
- какие файлы нельзя менять;
- какие зависимости запрещено добавлять;
- какой стиль кода принят в проекте;
- что считается завершённой задачей;
- нужно ли сначала показывать план;

- можно ли менять архитектуру;
- можно ли трогать публичный API.

`AGENTS.md` решает эту проблему.

Он превращает Codex из “просто помощника по коду” в помощника, который знает правила вашего проекта.

9.3 Чем `AGENTS.md` отличается от обычного запроса

Обычный запрос действует один раз.

Пример обычного запроса:

```
Исправь ошибку в авторизации.
```

Codex выполнит эту задачу, но из самого запроса не узнает всех правил проекта.

`AGENTS.md` действует как постоянная памятка.

Пример содержимого `AGENTS.md`:

```
После изменения кода всегда запускай тесты.
Перед крупным изменением покажи план.
Не меняй публичный API без согласования.
Не добавляй новые зависимости без объяснения.
```

Разница такая:

```
Обычный запрос
Конкретная задача прямо сейчас.

AGENTS.md
Постоянные правила, которые нужно учитывать при разных задачах.
```

Хорошая связка выглядит так:

```
AGENTS.md говорит Codex, как работать.
Ваш запрос говорит Codex, что сделать сейчас.
```

9.4 Где может лежать `AGENTS.md`

`AGENTS.md` может лежать на разных уровнях.

Глобальный уровень

```
~/.codex/AGENTS.md
```

Это личные правила пользователя для Codex.

Например:

```
Всегда отвечай кратко и по делу.
Перед опасными действиями объясняй риск.
После изменения файлов перечисляй, что было изменено.
```

Такой файл подходит для ваших общих предпочтений, которые полезны во всех проектах.

Уровень проекта

```
my-project/AGENTS.md
```

Это правила конкретного проекта.

Например:

```
Проект написан на Python.
После изменения backend-кода запускай pytest.
Не меняй публичный API без согласования.
```

Это самый частый и полезный вариант.

Уровень подпапки проекта

```
my-project/backend/AGENTS.md
```

Такой файл нужен, если в части проекта действуют отдельные правила.

Например:

```
my-project/
├── AGENTS.md
```

```
├── backend/
│   └── AGENTS.md
└── frontend/
    └── AGENTS.md
```

В корневом `AGENTS.md` можно описать общие правила всего проекта.

В `backend/AGENTS.md` — правила backend-части.

В `frontend/AGENTS.md` — правила frontend-части.

Главное правило:

> Чем ближе `AGENTS.md` к папке, из которой запущен Codex, тем более конкретные правила он может задавать.

9.5 `AGENTS.override.md`

Кроме обычного файла `AGENTS.md`, может использоваться файл:

```
AGENTS.override.md
```

Он нужен для переопределения инструкций на том же уровне.

Простая логика:

```
AGENTS.md
Обычные правила.

AGENTS.override.md
Правила, которые заменяют обычные правила на этом уровне.
```

Новичку этот файл обычно не нужен.

На старте достаточно использовать обычный:

```
AGENTS.md
```

9.6 Что писать в `AGENTS.md`

В `AGENTS.md` нужно писать то, что помогает Codex работать стабильнее.

Хорошее содержимое:

- описание проекта;
- структура проекта;
- команды запуска;
- команды проверки;
- правила разработки;
- ограничения;
- что нельзя менять;
- как работать с зависимостями;
- как оформлять результат;
- критерии готовности задачи.

Не нужно превращать `AGENTS.md` в огромную энциклопедию.

Хороший `AGENTS.md` должен быть:

```
понятным;
коротким;
практичным;
с конкретными командами;
с конкретными запретами;
с понятными критериями результата.
```

9.7 Что не стоит писать в `AGENTS.md`

Не стоит добавлять туда:

- очень длинные справочники;
- полную документацию библиотек;
- большие таблицы;
- временный статус задачи;
- огромные рассуждения;
- информацию, которая быстро устареет;
- секреты, пароли, токены, API-ключи.

Особенно важно:

> Никогда не храните API-ключи, пароли и секретные токены в `AGENTS.md`.

Если Codex нужен дополнительный справочный материал, лучше положить его в отдельный файл, например:

```
docs/architecture.md
docs/api-rules.md
docs/testing.md
```

А в `AGENTS.md` написать ссылку:

```
Перед изменением API смотри правила в `docs/api-rules.md`.
```

9.8 Как заполнять `AGENTS.md`

Ниже базовый шаблон, который можно использовать почти в любом проекте.

```
# AGENTS.md

## Описание проекта

Кратко опишите, что это за проект.

Важно, чтобы Codex понял:

- какую задачу решает проект;
- на каком языке или фреймворке он написан;
- какую часть системы он представляет;
- какие сценарии работы самые важные.

Пример:

Это backend-сервис на Python для обработки пользовательских заказов.
Основная логика находится в `app/`, тесты находятся в `tests/`.

## Структура проекта

Опишите основные папки и файлы.

Не нужно перечислять абсолютно каждый файл. Важно объяснить, какие части проекта
за что отвечают.

Пример:

- `app/` — основной код приложения.
- `app/auth/` — авторизация и работа с пользователями.
- `app/api/` — HTTP-обработчики.
- `app/services/` — бизнес-логика.
- `tests/` — тесты.
- `docs/` — документация проекта.
```

- `README.md` — основное описание проекта.

Если для разных задач важны разные папки, укажите это явно.

Пример:

- При задачах по авторизации сначала смотри `app/auth/` и `tests/auth/`.
- При задачах по API сначала смотри `app/api/`.
- При изменении бизнес-логики проверяй `app/services/`.

Команды запуска и проверки

Укажите команды, которые Codex должен использовать для проверки работы.

Пример:

```bash
pytest
```

Если есть разные проверки, перечислите их отдельно:

```bash
pytest
ruff check .
mypy .
```

Если команда долгая или тяжёлая, тоже напишите это.

Пример:

Полный набор тестов может выполняться долго. Для маленьких изменений сначала запускай тесты только по изменённой области.

Ограничения

Здесь напишите, чего Codex не должен делать.

Пример:

- Не добавляй новые зависимости без отдельного объяснения.
- Не меняй публичный API без согласования.
- Не удаляй существующие тесты.
- Не переписывай большие части проекта без необходимости.
- Не меняй формат базы данных без отдельного плана миграции.

Дополнительные данные

```
Здесь можно указать важные документы или правила, которые Codex должен учитывать.

Пример:

- Архитектурные решения описаны в `docs/architecture.md`.
- Правила API описаны в `docs/api.md`.
- План текущей задачи находится в `Plan.md`.
- Текущий статус работы находится в `Documentation.md`.

Если проект большой, этот раздел особенно полезен.

## Правила работы

Опишите, как Codex должен вести работу.

Пример:

- Перед крупными изменениями сначала предложи план.
- Если задача неясна, сначала задай уточняющий вопрос.
- Делай минимальные изменения, достаточные для решения задачи.
- После изменения кода запускай подходящие проверки.
- Если проверка упала, сначала объясни причину, затем исправь.
- В конце кратко перечисли изменённые файлы.

## Критерии готовности

Опишите, когда задача считается завершённой.

Пример:

Задача считается завершённой, когда:

- изменения внесены;
- код соответствует правилам проекта;
- тесты или проверки проходят;
- итог объяснён пользователю;
- перечислены изменённые файлы;
- указано, какие проверки были запущены.
```

9.9 Простой пример `AGENTS.md` для Python-проекта

```
# AGENTS.md

## Описание проекта

Это backend-сервис на Python.
```

Основной код находится в `app/`.

Тесты находятся в `tests/`.

Структура проекта

- `app/` — основной код приложения.
- `app/api/` — обработчики API.
- `app/auth/` — авторизация.
- `app/services/` — бизнес-логика.
- `tests/` — тесты.
- `README.md` — описание проекта.

Команды проверки

После изменения backend-кода запускай:

```bash
pytest
```

Если изменялись только тесты авторизации, можно сначала запускать:

```bash
pytest tests/auth
```

Ограничения

- Не добавляй новые зависимости без отдельного объяснения.
- Не меняй публичный API без согласования.
- Не удаляй тесты, чтобы “починить” проверку.
- Не переписывай архитектуру без отдельного плана.

Правила работы

- Перед крупными изменениями сначала предложи план.
- Делай минимальные изменения.
- После изменений запускай подходящие тесты.
- Если тесты упали, исправь причину.
- В конце объясни, что изменилось.

Критерии готовности

Задача считается завершённой, когда:

- изменения внесены;
- тесты проходят;

```
- пользователь получил краткое объяснение результата;
- перечислены изменённые файлы.
```

9.10 Простой пример `AGENTS.md` для frontend-проекта

```
# AGENTS.md

## Описание проекта

Это frontend-проект.

Основной код интерфейса находится в `src/`.

Компоненты находятся в `src/components/`.

## Структура проекта

- `src/` — основной код приложения.
- `src/components/` — переиспользуемые компоненты.
- `src/pages/` — страницы приложения.
- `src/styles/` — стили.
- `public/` — статические файлы.
- `package.json` — зависимости и команды проекта.

## Команды проверки

После изменения кода запускай:

```bash
npm run lint
```

Если менялась сборка или структура приложения, запускай:

```bash
npm run build
```

Ограничения

- Не добавляй новые библиотеки без объяснения.
- Не меняй дизайн глобально без запроса.
- Не удаляй существующие компоненты без необходимости.
- Не меняй публичные свойства компонентов без согласования.

Правила работы
```

```
- Перед крупными изменениями сначала предложи план.
- При изменении компонента объясняй, в каком файле он находится.
- Если нужно изменить код, указывай, куда вставляется изменение.
- После изменений запускай подходящую проверку.
- В конце перечисляй изменённые файлы.

## Критерии готовности

Задача считается завершённой, когда:

- нужное поведение реализовано;
- проект проходит проверку;
- изменения объяснены простым языком;
- перечислены изменённые файлы.
```

9.11 Минимальный `AGENTS.md`, если вы не знаете проект

Если вы пока не знаете, что писать подробно, начните с минимального варианта.

```
# AGENTS.md

## Описание проекта

Проект нужно сначала изучить перед изменениями.

## Правила работы

- Перед изменением файлов сначала кратко объясни план.
- Не добавляй новые зависимости без объяснения.
- Не выполняй опасные команды без подтверждения.
- Делай минимальные изменения.
- После изменений перечисли изменённые файлы.

## Критерии готовности

Задача считается завершённой, когда:

- изменения соответствуют запросу;
- результат объяснён;
- перечислены изменённые файлы;
- указано, какие проверки были выполнены.
```

Такой файл уже лучше, чем отсутствие правил.

9.12 Как проверить, что `AGENTS.md` работает

Самый простой способ — запустить Codex из папки проекта и спросить:

```
Какие инструкции проекта ты видишь? Кратко перескажи правила из AGENTS.md.
```

Если Codex пересказывает ваши правила, файл подхватился.

Если нет, проверьте:

1. `AGENTS.md` лежит в правильной папке;
2. Codex запущен из проекта или его подпапки;
3. имя файла написано правильно: `AGENTS.md`;
4. рядом нет `AGENTS.override.md`, который переопределяет правила;
5. проектные инструкции не слишком большие;
6. проект не запущен из неожиданной директории.

Команда, чтобы проверить текущую папку:

```
pwd
```

10. Песочница и подтверждения

Этот раздел нужен, чтобы понять, **что Codex может делать сам**, а **где он должен спросить разрешение**.

В Codex есть две важные настройки:

```
sandbox_mode
```

и:

```
approval_policy
```

Простыми словами:

```
sandbox_mode
ограничивает технические возможности Codex.

approval_policy
определяет, когда Codex должен спрашивать разрешение.
```

10.1 Что такое песочница

Песочница — это ограниченная рабочая зона для Codex.

Техническое название:

```
sandbox
```

Песочница отвечает за то, куда Codex может иметь доступ и что он может делать с файлами.

Главная идея:

> Песочница нужна, чтобы Codex не имел лишних прав.

Например, можно разрешить Codex:

```
только читать файлы;
читать и менять файлы проекта;
получить полный доступ к системе.
```

За это отвечает настройка:

```
sandbox_mode = "workspace-write"
```

10.2 Основные режимы песочницы

У `sandbox_mode` есть три основных режима:

```
read-only
workspace-write
danger-full-access
```

10.3 `read-only`

Режим:

```
sandbox_mode = "read-only"
```

означает:

> Codex может читать файлы, но не должен их изменять.

Этот режим подходит для:

```
первого знакомства с проектом;
анализа кода;
ревью;
поиска важных файлов;
объяснения структуры проекта;
составления плана.
```

Пример задачи для этого режима:

```
Изучи проект и объясни его структуру.
Ничего не меняй.
```

Это самый спокойный режим для начала.

10.4 `workspace-write`

Режим:

```
sandbox_mode = "workspace-write"
```

означает:

> Codex может менять файлы внутри рабочей области проекта.

Это основной режим для разработки.

Он подходит, когда нужно:

```
исправить баг;
создать файл;
обновить README;
написать тесты;
изменить код проекта.
```

Пример задачи:

```
Исправь ошибку в авторизации.
Не меняй публичный API.
После изменения запусти тесты.
```

Для обычной работы чаще всего используют именно:

```
sandbox_mode = "workspace-write"
```

10.5 `danger-full-access`

Режим:

```
sandbox_mode = "danger-full-access"
```

означает:

> Codex получает очень широкий доступ к системе.

Этот режим опасен для новичков.

Его не стоит включать просто потому, что “так удобнее”.

Почему:

```
Codex может получить доступ шире папки проекта;
ошибочная команда может затронуть лишние файлы;
сложнее контролировать последствия;
выше риск случайных изменений.
```

Используйте этот режим только если точно понимаете, зачем он нужен.

Для обучения и обычной разработки он не требуется.

10.6 Что такое подтверждения

Подтверждение — это момент, когда Codex спрашивает пользователя перед действием.

Техническое название:

```
approval
```

За это отвечает настройка:

```
approval_policy = "on-request"
```

Простыми словами:

> Подтверждения нужны, чтобы Codex не выполнял чувствительные действия без вашего разрешения.

10.7 Основные режимы подтверждений

Чаще всего встречаются такие значения:

```
on-request
never
untrusted
```

10.8 `approval_policy = "on-request"`

Режим:

```toml
approval_policy = "on-request"
```

означает:

> Codex будет спрашивать разрешение в чувствительных случаях.

Это хороший режим по умолчанию.

Он подходит для:

```
обычной разработки;
первых запусков;
работы с важным проектом;
ситуаций, где нужен баланс между удобством и контролем.
```

Рекомендуемый стартовый вариант:

```toml
approval_policy = "on-request"
sandbox_mode = "workspace-write"
```

10.9 `approval_policy = "never"`

Режим:

```
approval_policy = "never"
```

означает:

> Codex будет стараться работать без дополнительных подтверждений.

Это более автономный режим.

Он может быть удобен для:

```
тренировочной папки;
автоматических задач;
контролируемой среды;
однотипных безопасных операций.
```

Но для новичка это не лучший первый режим.

Важно:

> Не начинайте работу с новым проектом через `approval_policy = "never"`.

Сначала лучше понять проект, права и поведение Codex.

10.10 `approval_policy = "untrusted"`

Режим:

```
approval_policy = "untrusted"
```

означает более осторожное поведение.

Он подходит, если проект:

```
чужой;
непроверенный;
скачан из интернета;
может содержать неизвестные скрипты;
не должен получать много доверия.
```

Если вы открыли незнакомый репозиторий, лучше начинать осторожно:

```
approval_policy = "untrusted"
sandbox_mode = "read-only"
```

11. Подробнее про скиллы `Skill`

11.1 Что такое скилл

Скилл — это отдельная папка с инструкциями для повторяемой задачи.

Техническое название:

```
Skill
```

В этом документе можно понимать слово `Skill` как “навык” или “рабочий сценарий”.

Простыми словами:

> Скилл — это заранее подготовленная инструкция, которая помогает Codex выполнять определённый тип задач лучше и стабильнее.

Например, Codex сам по себе может помогать с кодом. Но если добавить скилл для проверки кода, он будет лучше понимать, **как именно** нужно делать проверку:

- на что обращать внимание;
- в каком порядке читать файлы;
- какие ошибки искать;
- как оформлять результат;
- какие проверки запускать;
- чего избегать.

То есть скилл не просто “добавляет информацию”. Он задаёт Codex готовую методику работы.

Например, можно создать:

- скилл для проверки кода;
- скилл для исправления багов;
- скилл для написания тестов;
- скилл для обновления документации;
- скилл для подготовки релиза;
- скилл для анализа интерфейса;
- скилл для проектирования архитектуры;
- скилл для работы с конкретным фреймворком.

Главная идея:

> Обычный Codex — это универсальный помощник.
> Codex со скиллом — это помощник, которому дали специализированную инструкцию под конкретную работу.

Например, условно:

```
Codex без скилла
“Помоги проверить код”.

Codex со скиллом code-review-workflow
“Проверь код по заранее заданному процессу: найди риски, ошибки, проблемы
архитектуры, пропущенные тесты и верни результат в нужном формате”.
```

11.2 Чем скилл отличается от `AGENTS.md`

Важно не путать `AGENTS.md` и скиллы.

Они оба помогают Codex работать лучше, но используются для разных задач.

```
AGENTS.md
Правила проекта.

Skill
Отдельный повторяемый рабочий сценарий.
```

Пример:

```
AGENTS.md
“Это backend-проект на Python. После изменений запускай pytest. Не меняй
публичный API.”

Skill
“Когда исправляешь баг, сначала найди причину, потом предложи план, потом внеси
минимальное изменение, потом запусти тесты.”
```

То есть:

- `AGENTS.md` объясняет Codex, **где он работает и какие правила проекта соблюдать**;
- `Skill` объясняет Codex, **как выполнять конкретный тип задачи**.

Хорошая связка выглядит так:

```
AGENTS.md
Описывает проект.

Skill
Описывает рабочий процесс.

Запрос пользователя
Говорит, что нужно сделать прямо сейчас.
```

11.3 Когда нужен скилл

Скилл стоит создавать, если вы часто повторяете одну и ту же задачу.

Например, вы регулярно просите Codex:

```
Проверь этот код.
Исправь баг.
Напиши тесты.
Обнови документацию.
Подготовь релиз.
Проверь архитектуру.
```

Если задача повторяется, её лучше оформить в скилл.

Тогда не придётся каждый раз подробно объяснять порядок работы. Codex сможет брать готовую инструкцию из папки с навыком.

Скилл особенно полезен, когда важно соблюдать одинаковый процесс.

Например:

```
При исправлении бага всегда:
1. Найти причину.
2. Объяснить её.
3. Предложить план.
4. Внести минимальное изменение.
5. Запустить тесты.
6. Перечислить изменённые файлы.
```

Без скилла это пришлось бы писать вручную в каждом запросе.

11.4 Где лежат скиллы

Скиллы могут лежать на разных уровнях.

Личные скиллы

Личные скиллы хранятся здесь:

```
~/.agents/skills/
```

Это скиллы, которые можно использовать в разных проектах.

Например:

```
~/.agents/skills/code-review-workflow/
~/.agents/skills/bugfix-workflow/
~/.agents/skills/test-generation-workflow/
```

Такие скиллы подходят для ваших личных рабочих привычек.
Проверить доступные навыки и способ их загрузки можно внутри Codex через `/skills`, если команда доступна в вашей версии.

Скиллы проекта

Скиллы конкретного проекта хранятся внутри проекта:

```
my-project/.agents/skills/
```

Пример:

```
my-project/.agents/skills/bugfix-workflow/
```

Такие скиллы полезны, если процесс связан именно с этим проектом.

Например:

```
my-project/.agents/skills/api-review-workflow/
my-project/.agents/skills/release-checklist/
my-project/.agents/skills/database-migration-check/
```

11.5 Минимальная структура скилла

Минимальный скилл — это папка и файл `SKILL.md`.

```
bugfix-workflow/
└── SKILL.md
```

Где:

```
bugfix-workflow/
папка навыка.

SKILL.md
главная инструкция навыка.
```

Название папки должно быть понятным.

Хорошие названия:

```
bugfix-workflow
code-review-workflow
test-generation-workflow
release-checklist
documentation-update
```

Плохие названия:

```
skill1
new-skill
mything
test
```

Название должно сразу объяснять, для чего нужен навык.

11.6 Расширенная структура скилла

Иногда одного `SKILL.md` достаточно. Но для более сложных навыков можно добавить дополнительные папки.

```
bugfix-workflow/
├── SKILL.md
├── scripts/
├── references/
└── assets/
```

`SKILL.md`

Главная инструкция.

В ней написано:

- когда использовать скилл;
- какой порядок действий;
- какие ограничения соблюдать;
- какой результат вернуть.

`scripts/`

Папка для скриптов.

Скрипт — это файл с командами или кодом, который можно запускать.

Например:

```
scripts/run-checks.sh
scripts/collect-logs.py
```

`references/`

Папка для справочных материалов.

Например:

```
references/testing-rules.md
references/review-checklist.md
references/api-guidelines.md
```

Это удобно, если информации много и не хочется перегружать основной `SKILL.md`.

`assets/`

Папка для дополнительных файлов.

Например:

- шаблоны;
- примеры;
- картинки;
- таблицы;
- заготовки документов.

11.7 Как устроен файл `SKILL.md`

Файл `SKILL.md` состоит из двух частей:

1. служебное описание в начале файла;
2. основная инструкция.

В начале файла находится блок:

```
---
name: bugfix-workflow
description: Используй этот навык, когда нужно найти и исправить ошибку в коде
проекта.
---
```

Этот блок помогает Codex понять:

- как называется скилл;
- когда его нужно использовать.

После этого идёт обычный текст инструкции.

Пример:

```
# Назначение

Этот навык помогает Codex исправлять ошибки аккуратно и проверяемо.

# Порядок работы

1. Найди описание ошибки.
2. Найди связанные файлы.
3. Объясни вероятную причину.
4. Предложи план исправления.
5. Внеси минимальные изменения.
6. Запусти тесты.
7. Если тесты упали, исправь причину.
8. В конце перечисли изменённые файлы и результат проверки.
```

11.8 Поле `name`

Поле `name` — это имя скилла.

Пример:

```
name: bugfix-workflow
```

Лучше использовать короткое и понятное имя.

Рекомендации:

```
используйте маленькие английские буквы;
разделяйте слова дефисом;
не используйте пробелы;
не делайте название слишком длинным.
```

Хорошо:

```
bugfix-workflow
code-review-workflow
test-generation
```

Плохо:

```
Bug Fix Workflow
мой навык
skill for checking and fixing all bugs in project
```

11.9 Поле `description`

Поле `description` — это описание, по которому Codex понимает, когда использовать скилл.

Пример:

```
description: Используй этот навык, когда нужно найти и исправить ошибку в коде
проекта.
```

Описание должно быть конкретным.

Плохо:

```
description: Помогает с кодом.
```

Почему плохо:

> Непонятно, когда именно использовать этот скилл.

Лучше:

```
description: Используй этот навык для поиска причины бага, аккуратного
исправления ошибки и проверки результата тестами.
```

Хорошее описание отвечает на вопрос:

> В какой ситуации этот скилл должен включаться?

11.10 Простой пример `SKILL.md`

```
---
name: bugfix-workflow
description: Используй этот навык, когда нужно найти и исправить ошибку в коде
проекта.
---

# Назначение

Этот навык помогает Codex исправлять ошибки аккуратно и проверяемо.

# Когда использовать

Используй этот навык, когда пользователь просит:

- найти баг;
- исправить ошибку;
- разобраться, почему тесты падают;
- починить неправильное поведение функции;
- исправить проблему после изменения кода.

# Порядок работы

1. Сначала воспроизведи ошибку или найди её описание.
2. Найди связанные файлы.
3. Объясни вероятную причину.
4. Предложи короткий план исправления.
5. Внеси минимальные изменения.
6. Запусти подходящие тесты.
7. Если тесты упали, прочитай ошибку и исправь причину.
8. В конце перечисли изменённые файлы и результат проверки.

# Ограничения

- Не добавляй новые зависимости без необходимости.
- Не переписывай большие части проекта без причины.
```

```
- Не меняй публичное поведение, если задача этого не требует.
- Не удаляй тесты только ради того, чтобы проверка прошла.

# Формат результата

В конце работы верни:

- что было причиной ошибки;
- что было изменено;
- какие файлы изменены;
- какие проверки запущены;
- какой результат проверки.
```

11.11 Как создать скилл в проекте

Допустим, у вас есть проект:

```
my-project/
```

Перейдите в папку проекта:

```
cd my-project
```

Создайте папку для скилла:

```
mkdir -p .agents/skills/bugfix-workflow
```

Создайте файл:

```
nano .agents/skills/bugfix-workflow/SKILL.md
```

Вставьте туда инструкцию скилла.

Структура получится такой:

```
my-project/
└── .agents/
    └── skills/
        └── bugfix-workflow/
            └── SKILL.md
```

11.12 Как создать личный скилл

Если вы хотите использовать скилл во многих проектах, создайте его в личной папке:

```
mkdir -p ~/.agents/skills/bugfix-workflow
```

Затем создайте файл:

```
nano ~/.agents/skills/bugfix-workflow/SKILL.md
```

Такой скилл будет не привязан к одному проекту.

11.13 Как понять, что скилл нужен

Используйте простое правило:

```
Если вы объясняете Codex один и тот же порядок действий больше трёх раз — пора
сделать скилл.
```

Например, если вы часто пишете:

```
Сначала найди причину ошибки.
Потом предложи план.
Потом внеси минимальные изменения.
Потом запусти тесты.
В конце перечисли файлы.
```

лучше оформить это в `bugfix-workflow`.

11.14 Примеры полезных скиллов

`code-review-workflow`

Для проверки кода.

Что может содержать:

- искать ошибки логики;
- проверять безопасность;
- искать пропущенные тесты;

- проверять читаемость;
- возвращать список замечаний по важности.

`bugfix-workflow`

Для исправления багов.

Что может содержать:

- найти причину;
- исправить минимально;
- проверить тестами;
- объяснить результат.

`test-generation-workflow`

Для написания тестов.

Что может содержать:

- изучить поведение функции;
- определить важные сценарии;
- написать тесты;
- запустить проверку;
- не менять production-код без необходимости.

`documentation-update`

Для обновления документации.

Что может содержать:

- найти устаревший текст;
- обновить описание;
- сохранить стиль документа;
- перечислить изменённые разделы.

`release-checklist`

Для подготовки релиза.

Что может содержать:

- проверить тесты;
- проверить changelog;
- проверить версию;
- проверить документацию;

- подготовить итоговый список изменений.

---

12. Как правильно давать задачи Codex

Codex работает лучше, когда задача сформулирована конкретно.

Плохой запрос:

```
Сделай нормально.
```

Почему плохо:

- непонятна цель;
- непонятно, что можно менять;
- непонятно, что нельзя трогать;
- непонятно, как проверить результат;
- Codex может изменить лишнее.

Хороший запрос даёт Codex рамки.

12.1 Формула хорошей задачи

```
Цель
Контекст
Ограничения
Файлы или области проекта
Проверка
Критерий готовности
```

Не обязательно всегда заполнять всё. Но чем сложнее задача, тем полезнее эта структура.

12.2 Что писать в каждом блоке

Цель

Что нужно сделать.

```
Исправить ошибку refresh token в backend.
```

Контекст

Что важно знать о проблеме.

```
Ошибка возникает после истечения access token.
Сейчас приложение возвращает 401.
```

Ограничения

Что нельзя делать.

```
Не менять публичный API.
Не добавлять новые зависимости.
Не переписывать авторизацию полностью.
```

Файлы

Где искать проблему.

```
Сначала проверь app/auth и tests/auth.
```

Если вы не знаете файлы, можно написать:

```
Сначала найди связанные файлы и объясни план. Пока ничего не меняй.
```

Проверка

Как проверить результат.

```
Запусти pytest tests/auth.
```

Если не знаете команду проверки:

```
Сначала найди, какие команды проверки есть в проекте.
```

Критерий готовности

Когда задача считается выполненной.

```
Готово, когда тесты проходят, ошибка исправлена, а diff минимальный.
```

12.3 Пример хорошей задачи

```
Цель:
Исправить ошибку refresh token в backend.

Контекст:
Ошибка возникает после истечения access token.
Сейчас приложение возвращает 401.

Ограничения:
Не менять публичный API.
Не добавлять новые зависимости.
Не переписывать авторизацию полностью.

Файлы:
Сначала проверь папки app/auth и tests/auth.

Проверка:
Запусти pytest tests/auth.

Готово, когда:
Тесты проходят, ошибка исправлена, а итоговый diff минимальный.
В конце перечисли изменённые файлы.
```

12.4 Если вы не знаете, где проблема

Используйте безопасный запрос:

```
Я хочу исправить ошибку, но не знаю, где она находится.
Сначала изучи проект, найди вероятные файлы и предложи план.
Пока ничего не меняй.
```

12.5 Короткий шаблон

```
Цель:
[...]

Контекст:
[...]

Ограничения:
[...]
```

```
Файлы:
[...]

Проверка:
[...]

Готово, когда:
[...]
```

Главное правило:

> Не просите Codex “сделать нормально”. Объясните, что сделать, что не трогать и как проверить результат.

13. Подробнее про настройку агента

В этом разделе разберём, как настраивать поведение Codex.

Главный файл настроек называется:

```
config.toml
```

Через него можно управлять тем:

- какую модель использовать;
- насколько глубоко агент должен рассуждать;
- когда он должен спрашивать разрешение;
- может ли он менять файлы;
- есть ли доступ к сети;
- сохраняется ли история;
- какие режимы работы использовать.

Простыми словами:

> `config.toml` — это панель управления Codex.

13.1 Где лежит файл настроек

Основной пользовательский файл настроек находится здесь:

```
~/.codex/config.toml
```

Он действует как личная настройка Codex для всех проектов.

Если файла нет, его можно создать:

```
mkdir -p ~/.codex
nano ~/.codex/config.toml
```

На macOS и Linux символ `~` означает домашнюю папку пользователя.

13.2 Глобальная и проектная настройка

У Codex может быть несколько уровней настроек.

Глобальная настройка

```
~/.codex/config.toml
```

Это настройки пользователя для всех проектов.

Например:

```
approval_policy = "on-request"
sandbox_mode = "workspace-write"
```

Проектная настройка

```
my-project/.codex/config.toml
```

Это настройки только для конкретного проекта.

Например, в одном проекте можно разрешить Codex менять файлы, а в другом оставить только чтение.

```
sandbox_mode = "read-only"
```

Главная идея:

```
~/.codex/config.toml
Общие настройки для всех проектов.
```

```
my-project/.codex/config.toml
Настройки только для одного проекта.
```

13.3 Что важнее: глобальная или проектная настройка

Если одно и то же правило указано в разных местах, Codex использует более конкретную настройку.

Упрощённый порядок такой:

```
1. Параметры при запуске команды
2. Выбранный профиль
3. Настройки проекта
4. Личные настройки пользователя
5. Настройки по умолчанию
```

Пример:

В глобальном файле написано:

```toml
sandbox_mode = "workspace-write"
```

А в проекте написано:

```toml
sandbox_mode = "read-only"
```

Тогда для этого проекта Codex будет работать в режиме:

```toml
sandbox_mode = "read-only"
```

Потому что проектная настройка конкретнее.

13.4 Минимальный безопасный `config.toml`

Для начала можно использовать такой вариант:

```toml
approval_policy = "on-request"
sandbox_mode = "workspace-write"
web_search = "cached"
```

```toml
[sandbox_workspace_write]
network_access = false

[history]
persistence = "save-all"
```

Что это значит:

```
approval_policy = "on-request"
Codex будет спрашивать разрешение в чувствительных случаях.

sandbox_mode = "workspace-write"
Codex сможет менять файлы внутри рабочей папки проекта.

web_search = "cached"
Codex сможет использовать более безопасный режим поиска информации.

network_access = false
Команды, которые запускает Codex, не получают свободный доступ к сети.

history.persistence = "save-all"
История работы сохраняется.
```

Это хороший стартовый режим для обычной работы.

13.5 Настройка модели

Параметр `model` задаёт модель, которую использует Codex.

Пример:

```toml
model = "gpt-5.5"
```

Если вы не уверены, какую модель указывать, можно оставить значение по умолчанию или использовать модель, рекомендованную в текущей версии Codex.

13.6 Настройка глубины рассуждения

Параметр:

```toml
model_reasoning_effort = "medium"
```

отвечает за то, насколько тщательно Codex должен думать над задачей.

Возможные режимы:

```
minimal  простые задачи
low      лёгкие задачи
medium   обычная работа
high     сложные задачи
xhigh    очень сложные задачи, если поддерживается
```

Для большинства случаев подходит:

```
model_reasoning_effort = "medium"
```

Для сложной архитектурной задачи можно использовать:

```
model_reasoning_effort = "xhigh"
```

Не стоит всегда ставить максимальный режим. Чем выше режим рассуждения, тем дольше и дороже может быть работа.

13.7 Настройка разрешений

Параметр:

```
approval_policy = "on-request"
```

определяет, когда Codex должен спрашивать разрешение.

Основные варианты:

```
on-request
Codex спрашивает разрешение в чувствительных случаях.

never
Codex старается работать без дополнительных подтверждений.

untrusted
Более осторожный режим для неизвестных проектов.
```

Для обычной работы лучше использовать:

```toml
approval_policy = "on-request"
```

Не рекомендуется начинать с:

```toml
approval_policy = "never"
```

Потому что в этом режиме Codex может действовать слишком самостоятельно.

13.8 Настройка доступа к файлам

Параметр:

```toml
sandbox_mode = "workspace-write"
```

отвечает за то, может ли Codex читать и менять файлы.

Основные варианты:

```
read-only
Codex может читать файлы, но не должен их менять.

workspace-write
Codex может менять файлы в рабочей папке проекта.

danger-full-access
Codex получает очень широкий доступ. Опасный режим.
```

Для знакомства можно использовать:

```toml
sandbox_mode = "read-only"
```

Для обычной разработки:

```toml
sandbox_mode = "workspace-write"
```

Опасный режим:

```toml
sandbox_mode = "danger-full-access"
```

нельзя использовать без понимания последствий.

Подробно песочница будет разобрана в следующем разделе.

13.9 Настройка доступа к сети

Внутри режима `workspace-write` можно отдельно указать, есть ли у команд доступ к сети.

```
[sandbox_workspace_write]
network_access = false
```

Это значит:

> Команды, которые запускает Codex, не получают свободный доступ к интернету.

Например, если Codex попробует выполнить:

```
npm install
```

или:

```
pip install package-name
```

таким командам может понадобиться сеть.

Если вы хотите разрешить сетевой доступ для команд, можно указать:

```
[sandbox_workspace_write]
network_access = true
```

Но для безопасного старта лучше оставить:

```
network_access = false
```

Важно:

```
web_search
Отвечает за поиск информации самим Codex.

network_access
Отвечает за доступ к сети для команд терминала.
```

Это разные настройки.

13.10 Настройка истории

История настраивается так:

```
[history]
persistence = "save-all"
```

Это значит, что история сессий сохраняется.

Если вы не хотите сохранять историю:

```
[history]
persistence = "none"
```

Для личного компьютера удобно использовать:

```
[history]
persistence = "save-all"
```

Для чужого, общего или временного компьютера лучше использовать:

```
[history]
persistence = "none"
```

13.11 Профили

Профиль — это готовый набор настроек под конкретный режим работы.

Например, можно создать три режима:

```
[profiles.safe]
approval_policy = "on-request"
sandbox_mode = "read-only"

[profiles.dev]
approval_policy = "on-request"
sandbox_mode = "workspace-write"

[profiles.auto]
approval_policy = "never"
sandbox_mode = "workspace-write"
```

Теперь можно запускать Codex с нужным профилем:

```
codex --profile safe
```

или:

```
codex --profile dev
```

Смысл простой:

```
safe
Осторожный режим: только анализ.

dev
Обычная разработка: можно менять файлы проекта.

auto
Более автономный режим. Использовать осторожно.
```

Профили удобны, потому что не нужно каждый раз вручную менять `config.toml`.

13.12 Пример удобного `config.toml`

Для обычной работы можно начать с такого файла:

```toml
model_reasoning_effort = "medium"
approval_policy = "on-request"
sandbox_mode = "workspace-write"
web_search = "cached"

[sandbox_workspace_write]
network_access = false

[history]
persistence = "save-all"

[profiles.safe]
approval_policy = "on-request"
sandbox_mode = "read-only"

[profiles.dev]
approval_policy = "on-request"
sandbox_mode = "workspace-write"
```

```toml
[profiles.deep]
model_reasoning_effort = "high"
approval_policy = "on-request"
sandbox_mode = "workspace-write"
```

Как использовать:

```
codex --profile safe
```

для безопасного анализа.

```
codex --profile dev
```

для обычной разработки.

```
codex --profile deep
```

для сложной задачи, где нужно больше рассуждения.

13.13 Как понять, какие настройки активны

Внутри Codex можно использовать команду:

```
/status
```

Она показывает текущее состояние Codex.

Также полезна команда:

```
/debug-config
```

Она помогает понять, какие настройки применились.

Если Codex ведёт себя не так, как вы ожидали, проверьте:

1. из какой папки вы его запустили;
2. какой `config.toml` используется;
3. выбран ли профиль;
4. есть ли проектный `.codex/config.toml`;
5. не переопределяет ли проектная настройка глобальную;
6. какой режим песочницы активен;

7. какая политика подтверждений активна.

13.14 Что настраивать в первую очередь

МОЖНО вообще ничего не трогать, в таком случае codex будет настроен по умолчанию!

Не нужно сразу трогать все параметры.

Для старта достаточно настроить:

```
approval_policy
sandbox_mode
history
```

То есть:

```toml
approval_policy = "on-request"
sandbox_mode = "workspace-write"

[history]
persistence = "save-all"
```

После этого можно постепенно добавлять:

```
model_reasoning_effort
web_search
network_access
profiles
model
model_provider
```

13.15 Короткая памятка

```
~/.codex/config.toml
Главный личный файл настроек Codex.

my-project/.codex/config.toml
Настройки Codex только для конкретного проекта.

approval_policy
Когда Codex должен спрашивать разрешение.
```

```
sandbox_mode
Что Codex может делать с файлами.

network_access
Могут ли команды Codex выходить в интернет.

web_search
Может ли Codex искать информацию.

history
Сохранять ли историю.

profile
Готовый режим настроек под конкретную задачу.
```

Главное правило:

> Сначала настройте безопасность и права Codex, а уже потом переходите к продвинутым возможностям.

14. Подробно про MCP

14.1 Что такое MCP

`MCP` — это способ подключить Codex к внешнему источнику данных или инструменту.

Простыми словами:

> MCP нужен, когда Codex должен знать или делать что-то за пределами текущей папки проекта.

Без MCP Codex работает в основном с:

```
файлами проекта;
терминалом;
AGENTS.md;
config.toml;
skills.
```

С MCP он может дополнительно обращаться к:

```
документации;
GitHub;
базе знаний;
системе задач;
внутреннему сервису;
```

```
браузерному инструменту;
другому внешнему источнику.
```

OpenAI, например, предоставляет OpenAI Docs MCP — публичный MCP-сервер для поиска и чтения developer-документации OpenAI. Он работает только с документацией и не вызывает OpenAI API от имени пользователя. (OpenAI Платформа)

14.2 Когда MCP действительно нужен

MCP стоит подключать, если Codex не хватает данных из проекта.

Примеры:

```
Нужно сверяться с актуальной документацией.
Нужно читать GitHub issues или pull requests.
Нужно обращаться к внутренней базе знаний.
Нужно работать с внешней системой задач.
Нужно использовать специальный инструмент компании.
```

Если задача решается только по локальному коду, MCP обычно не нужен.

Примеры задач без MCP:

```
Объясни файл.
Исправь баг.
Напиши тест.
Обнови README.
Проверь текущие изменения.
```

Главное правило:

> MCP подключают не “на всякий случай”, а под конкретный внешний источник.

14.3 Что такое MCP-сервер

`MCP-сервер` — это сервис или программа, которая даёт Codex дополнительные инструменты.

Схема:

```
Codex
  ↓
MCP-сервер
```

```
  ↓
документация / GitHub / база знаний / инструмент
```

MCP-сервер может быть:

```
удалённым    работает по URL;
локальным    запускается как программа на вашем компьютере.
```

14.4 Типы MCP-подключений

1. Публичный MCP без входа

Такой сервер можно подключить сразу.

Пример: OpenAI Docs MCP.

```
codex mcp add openaiDeveloperDocs --url https://developers.openai.com/mcp
```

Проверка:

```
codex mcp list
```

OpenAI указывает этот способ как быстрый вариант подключения Docs MCP к Codex CLI. (OpenAI Платформа)

---

2. MCP с токеном

Некоторым серверам нужен токен доступа.

Токен нельзя вставлять прямо в публичные файлы проекта. Правильнее хранить его в переменной окружения.

Схема:

```
создаём переменную с токеном;
подключаем MCP;
указываем Codex, где лежит токен.
```

Пример создания переменной в терминале:

```
export MY_MCP_TOKEN="ваш_токен"
```

Затем:

```
codex mcp add myService --url https://example.com/mcp --bearer-token-env-var
MY_MCP_TOKEN
```

Смысл:

```
MY_MCP_TOKEN
это имя переменной окружения.

Codex берёт токен из этой переменной,
а не хранит сам токен в config.toml.
```

Это правильнее, чем записывать секрет прямо в файл.

---

3. MCP с OAuth

`OAuth` — это вход через браузер.

Простыми словами:

> Codex открывает страницу входа, вы разрешаете доступ, после этого Codex сохраняет данные авторизации.

Обычно используется команда:

```
codex mcp login имя_сервера
```

Если серверу нужны конкретные права:

```
codex mcp login имя_сервера --scopes read,write
```

`scopes` — это список разрешений.

Например:

```
read     разрешение на чтение
write    разрешение на запись
```

OAuth нужен, когда доступ должен быть привязан к конкретному пользователю: например, к его аккаунту в сервисе, задачам, репозиториям или внутренней системе. OpenAI в материалах по MCP указывает OAuth как рекомендуемый способ защиты пользовательских данных для удалённых MCP-серверов. (OpenAI Платформа)

---

4. Локальный MCP-сервер

Локальный MCP-сервер — это программа, которую Codex запускает на вашем компьютере.

Такой сервер может запускаться через:

```
npx
node
python
docker
```

Общий вид команды:

```
codex mcp add имя_сервера -- команда_запуска
```

Примерная логика:

```
удалённый MCP
Codex подключается к URL.

локальный MCP
Codex запускает программу на вашем компьютере.
```

Локальные MCP чаще используют для внутренних инструментов, тестовых серверов, работы с файлами, базами данных или специализированными скриптами.

14.5 Основные команды MCP

```
codex mcp add имя_сервера ...
```

Добавить MCP-сервер.

```
codex mcp list
```

Показать подключённые MCP-серверы.

```
codex mcp get имя_сервера
```

Посмотреть подробности одного подключения.

```
codex mcp remove имя_сервера
```

Удалить MCP-сервер.

```
codex mcp login имя_сервера
```

Войти через OAuth, если сервер это поддерживает.

```
codex mcp logout имя_сервера
```

Выйти из OAuth-подключения.

Если команда не работает в вашей версии Codex, сначала проверьте доступные команды:

```
codex mcp --help
```

14.6 Где хранится MCP

При добавлении через CLI Codex сохраняет MCP-подключение в своей конфигурации.

Обычно это:

```
~/.codex/config.toml
```

OpenAI также показывает, что OpenAI Docs MCP можно добавить напрямую в `~/.codex/config.toml`, но для обучения проще использовать команду `codex mcp add`, потому что так меньше риска ошибиться в структуре файла. (OpenAI Платформа)

14.7 Как правильно называть MCP-сервер

Имя должно быть коротким и понятным.

Хорошо:

```
openaiDocs
github
companyDocs
jira
browserTools
```

Плохо:

```
server1
test
mything
docs123
```

Почему это важно:

> Когда MCP-серверов несколько, Codex легче выбрать нужный, если имя понятно.

OpenAI также советует использовать короткие и описательные имена MCP-серверов. (OpenAI Платформа)

14.8 Как пользоваться MCP после подключения

После подключения MCP лучше прямо сказать Codex, что его нужно использовать.

Пример:

```
Используй OpenAI Docs MCP и найди актуальную информацию про config.toml в Codex.
```

Или строже:

```
Не отвечай по памяти. Сначала проверь информацию через MCP openaiDeveloperDocs.
```

Если MCP нужно использовать постоянно, добавьте правило в `AGENTS.md`:

```
Если задача связана с OpenAI, Codex или документацией OpenAI, используй MCP-
сервер openaiDeveloperDocs.
```

OpenAI рекомендует добавлять такое правило в `AGENTS.md`, чтобы агент надёжнее использовал Docs MCP без отдельного напоминания. (OpenAI Платформа)

14.9 Как правильно работать с MCP

Правильный порядок:

```
1. Понять, какой внешний источник нужен.
2. Найти MCP-сервер для этого источника.
3. Подключить его через codex mcp add.
4. Если нужна авторизация — выполнить login или указать токен.
5. Проверить подключение через codex mcp list.
6. В запросе явно попросить Codex использовать MCP.
7. Если источник нужен часто — закрепить правило в AGENTS.md.
```

14.10 Что даёт MCP на практике

MCP делает Codex полезнее в трёх случаях.

1. Меньше устаревших ответов

Если подключить документацию, Codex может сверяться с источником, а не отвечать только по памяти.

Пример:

```
Перед ответом проверь актуальную документацию через MCP.
```

2. Больше контекста

Codex может учитывать не только код проекта, но и внешние данные.

Например:

```
Посмотри GitHub issue и найди связанные файлы в проекте.
```

3. Больше действий

Если MCP-сервер даёт инструменты, Codex может не только читать, но и выполнять действия через этот инструмент.

Например:

```
получить список задач;
найти нужный документ;
```

```
прочитать данные из сервиса;
создать объект во внешней системе, если это разрешено.
```

---

15. Субагенты

15.1 Что такое субагент

Субагент — это дополнительный помощник внутри Codex, которому можно поручить отдельную часть большой задачи.

Простыми словами:

> Субагент — это отдельная роль для части работы.

Обычный режим:

```
Один Codex → одна задача
```

Режим с субагентами:

```
Главный Codex
├── субагент для backend
├── субагент для frontend
├── субагент для тестов
└── субагент для документации
```

Главный Codex ставит задачу, а субагенты помогают изучить разные части проекта. Это полезно для больших задач, ревью, анализа архитектуры и долгой автономной работы. OpenAI отдельно описывает Codex-модели как модели для агентного программирования и длинных coding-задач, поэтому субагентов лучше понимать как часть более широкой агентной работы, а не как обычный чат-режим. (OpenAI Разработчики)

15.2 Зачем нужны субагенты

Субагенты нужны, когда задача слишком большая для одного направления внимания.

Они помогают:

```
разделить большую задачу на части;
проверить проект с разных сторон;
не смешивать разные роли;
```

```
получить несколько независимых выводов;
собрать общий итог из отдельных проверок.
```

Пример:

```
Задача:
Проверить проект перед релизом.

Субагенты:
- backend_reviewer проверяет backend;
- frontend_reviewer проверяет frontend;
- test_reviewer проверяет тесты;
- docs_reviewer проверяет документацию.
```

15.3 Когда использовать субагентов

Используйте субагентов для больших задач.

Подходящие случаи:

```
ревью большого проекта;
анализ незнакомой кодовой базы;
поиск причины сложного бага;
подготовка проекта к релизу;
проверка backend, frontend и тестов отдельно;
сравнение нескольких архитектурных вариантов;
анализ безопасности отдельным направлением.
```

Не используйте субагентов для мелких задач:

```
исправить одну опечатку;
объяснить один файл;
добавить один тест;
переименовать переменную;
обновить маленький README.
```

Главное правило:

> Если задачу спокойно может выполнить один Codex, субагенты не нужны.

15.4 Как включить субагентов

В некоторых версиях Codex субагенты могут быть доступны сразу, а в некоторых — через экспериментальные возможности.

Сначала проверьте внутри Codex:

```
/
```

Посмотрите, есть ли команды или разделы, связанные с агентами:

```
/agents
/agent
/experimental
```

Если требуется включение через файл настроек, оно обычно делается в:

```
~/.codex/config.toml
```

Пример:

```
[features]
multi_agent = true
```

После изменения файла Codex нужно перезапустить.

Порядок:

```
1. Открыть ~/.codex/config.toml.
2. Включить multi_agent, если функция требует этого.
3. Сохранить файл.
4. Закрыть Codex.
5. Запустить Codex заново.
6. Проверить доступные команды через /.
```

15.5 Встроенные и свои субагенты

Условно есть два типа субагентов:

```
встроенные;
пользовательские.
```

Встроенные

Встроенные роли уже есть внутри Codex.

Типовая логика:

```
default
универсальный агент.

explorer
агент для изучения проекта и поиска информации.

worker
агент для выполнения работы и внесения изменений.
```

Названия и доступность встроенных ролей могут зависеть от версии Codex. Поэтому проверяйте их через интерфейс Codex, список команд или документацию вашей версии.

Пользовательские

Пользовательские субагенты — это роли, которые вы описываете сами.

Например:

```
backend_reviewer
frontend_reviewer
test_reviewer
security_checker
docs_researcher
```

15.6 Где хранятся пользовательские субагенты

Свой субагент описывается отдельным `.toml`-файлом.

Один файл — один субагент.

Личные субагенты

```
~/.codex/agents/
```

Они доступны в разных проектах пользователя.

Пример:

```
~/.codex/agents/reviewer.toml
~/.codex/agents/security_checker.toml
```

```
~/.codex/agents/docs_researcher.toml
```

Проектные субагенты

```
my-project/.codex/agents/
```

Они относятся к конкретному проекту.

Пример:

```
my-project/
└── .codex/
    └── agents/
        ├── backend_reviewer.toml
        ├── frontend_reviewer.toml
        └── test_reviewer.toml
```

15.7 Как Codex загружает субагентов

Codex смотрит папки с агентами:

```
~/.codex/agents/
my-project/.codex/agents/
```

и читает `.toml`-файлы внутри них.

Каждый `.toml`-файл считается описанием отдельной роли.

Важно:

> Лучше перезапустить Codex после создания нового файла субагента.

Так Codex точно увидит новый файл.

15.8 Формат TOML-файла субагента

Минимальный субагент содержит три главных поля:

```toml
name = "reviewer"
description = "Проверяет код на ошибки, риски и пропущенные тесты."
```

```
developer_instructions = """
Ты ревьюер кода.

Проверяй:
- ошибки логики;
- проблемы безопасности;
- пропущенные тесты;
- слишком широкие изменения.

Не меняй файлы.
Верни список замечаний по важности.
"""
```

Главные поля:

```
name
имя субагента.

description
краткое описание, когда его использовать.

developer_instructions
главные инструкции поведения.
```

15.9 Поле `name`

`name` — это имя субагента.

Пример:

```
name = "backend_reviewer"
```

Лучше, чтобы имя файла и `name` были похожи.

Хорошо:

```
Файл: backend_reviewer.toml
name = "backend_reviewer"
```

Плохо:

```
Файл: agent1.toml
name = "backend_reviewer"
```

Так сложнее понять, какой файл за что отвечает.

Хорошие имена:

```
backend_reviewer
frontend_reviewer
test_reviewer
security_checker
docs_researcher
release_checker
```

Плохие имена:

```
agent1
helper
test
myagent
```

15.10 Поле `description`

`description` объясняет, когда использовать субагента.

Плохо:

```
description = "Помогает с кодом."
```

Слишком общее описание.

Лучше:

```
description = "Проверяет backend-код на ошибки логики, безопасность и пропущенные
тесты."
```

Описание должно отвечать на вопрос:

> Для какой задачи нужен этот субагент?

15.11 Поле `developer_instructions`

`developer_instructions` — это главная инструкция субагента.

Туда нужно писать:

```
кто он;
что проверяет;
какие файлы или области учитывать;
что запрещено;
какой результат вернуть.
```

Пример:

```
developer_instructions = """
Ты backend-reviewer.

Твоя область:
- backend-код;
- API;
- авторизация;
- бизнес-логика;
- работа с базой данных;
- backend-тесты.

Проверяй:
- ошибки логики;
- проблемы безопасности;
- неправильную обработку ошибок;
- риск сломать публичный API;
- отсутствие тестов.

Не меняй файлы.
Верни краткий список проблем по важности.
Для каждой проблемы укажи файл и причину.
"""
```

Плохо:

```
developer_instructions = """
Ты хороший программист. Делай всё правильно.
"""
```

Слишком расплывчато.

15.12 Необязательные поля субагента

Кроме обязательных полей можно указать дополнительные настройки.

Частые поля:

```
model
какую модель использовать.

model_reasoning_effort
насколько глубоко рассуждать.

sandbox_mode
какие права доступа дать.

approval_policy
когда спрашивать разрешение.

web_search
режим веб-поиска.

mcp_servers
какие MCP-серверы доступны.

nickname_candidates
варианты коротких отображаемых имён.
```

Если эти поля не указаны, субагент обычно работает с настройками основной сессии.

15.13 Пример субагента только для анализа

Файл:

```
my-project/.codex/agents/reviewer.toml
```

Содержимое:

```
name = "reviewer"
description = "Проверяет код на ошибки, безопасность и пропущенные тесты."

model_reasoning_effort = "medium"
sandbox_mode = "read-only"

developer_instructions = """
Ты ревьюер кода.

Проверяй:
- ошибки логики;
- проблемы безопасности;
```

```
- регрессии поведения;
- пропущенные тесты;
- слишком широкие изменения.

Не меняй файлы.
Не исправляй код.
Верни список замечаний по важности.
Для каждого замечания укажи файл и причину.
"""
```

Это хороший первый субагент, потому что он только читает и не меняет файлы.

15.14 Пример backend-субагента

Файл:

```
my-project/.codex/agents/backend_reviewer.toml
```

Содержимое:

```
name = "backend_reviewer"
description = "Анализирует backend-код: API, авторизацию, бизнес-логику, базу
данных и тесты."

model_reasoning_effort = "high"
sandbox_mode = "read-only"

developer_instructions = """
Ты backend-reviewer.

Твоя область:
- backend-код;
- API;
- авторизация;
- бизнес-логика;
- работа с базой данных;
- backend-тесты.

Проверяй:
- ошибки логики;
- проблемы безопасности;
- неправильную обработку ошибок;
- риск сломать публичный API;
- пропущенные тесты.
```

```
Не меняй файлы.
Верни краткий список проблем по важности.
Если критичных проблем нет, напиши это явно.
"""
```

15.15 Пример frontend-субагента

Файл:

```
my-project/.codex/agents/frontend_reviewer.toml
```

Содержимое:

```
name = "frontend_reviewer"
description = "Анализирует frontend-код: компоненты, страницы, состояние
интерфейса и пользовательские сценарии."

model_reasoning_effort = "medium"
sandbox_mode = "read-only"

developer_instructions = """
Ты frontend-reviewer.

Твоя область:
- компоненты интерфейса;
- страницы;
- состояние приложения;
- пользовательские сценарии;
- обработка ошибок на клиенте.

Проверяй:
- сломанную логику интерфейса;
- проблемы состояния;
- риск сломать существующие компоненты;
- отсутствие проверок для важных сценариев;
- неочевидные UX-проблемы.

Не меняй файлы.
Верни конкретные замечания с указанием файлов.
"""
```

15.16 Пример субагента с MCP

Если субагенту нужен внешний источник, можно указать MCP-сервер.

Пример:

```
name = "docs_researcher"
description = "Проверяет информацию по документации через подключённый MCP."

model_reasoning_effort = "medium"
sandbox_mode = "read-only"

mcp_servers = ["openaiDeveloperDocs"]

developer_instructions = """
Ты docs-researcher.

Твоя задача — проверять информацию по документации.

Если задача связана с OpenAI, Codex или API, используй MCP-сервер
openaiDeveloperDocs.

Не меняй файлы.
Верни краткий ответ и укажи, что именно было проверено.
"""
```

Важно:

> MCP-сервер сначала должен быть подключён в Codex.

15.17 Как создать субагента пошагово

Допустим, нужно создать ревьюера проекта.

Шаг 1. Перейти в проект

```
cd my-project
```

Шаг 2. Создать папку

```
mkdir -p .codex/agents
```

Шаг 3. Создать файл

```
nano .codex/agents/reviewer.toml
```

Шаг 4. Вставить описание

```
name = "reviewer"
description = "Проверяет код на ошибки, безопасность и пропущенные тесты."

sandbox_mode = "read-only"

developer_instructions = """
Ты ревьюер кода.

Не меняй файлы.
Проверяй корректность, безопасность и наличие тестов.
Верни список конкретных замечаний по важности.
"""
```

Шаг 5. Сохранить и перезапустить Codex

После этого структура будет такой:

```
my-project/
└── .codex/
    └── agents/
        └── reviewer.toml
```

Если Codex уже открыт, закройте и запустите его заново.

15.18 Как вызвать субагента

Лучше вызывать субагента явно.

Пример:

```
Используй субагента reviewer для проверки текущих изменений.
Ничего не меняй.
Верни список замечаний по важности.
```

Или:

```
Используй backend_reviewer и frontend_reviewer.

backend_reviewer проверяет backend.
frontend_reviewer проверяет frontend.
```

```
Ничего не меняйте.
После этого собери общий список рисков.
```

Если не уверены, что Codex видит субагентов:

```
Проверь папку .codex/agents и перечисли доступных субагентов.
```

15.19 Как правильно ставить задачу субагентам

Плохо:

```
Используй субагентов и сделай всё.
```

Лучше:

```
Используй трёх субагентов:

1. backend_reviewer — проверь backend.
2. frontend_reviewer — проверь frontend.
3. test_reviewer — проверь тесты.

Ничего не меняйте в файлах.

Каждый субагент должен вернуть:
- что проверил;
- какие проблемы нашёл;
- какие файлы важны;
- что рекомендует сделать.

После этого собери общий итог.
```

15.20 Настройки `[agents]` в `config.toml`

Общие настройки субагентов можно задавать в:

```
~/.codex/config.toml
```

или в проекте:

```
my-project/.codex/config.toml
```

Пример:

```toml
[agents]
max_threads = 6
max_depth = 1
```

Что значит:

```
max_threads
максимальное количество агентных потоков.

max_depth
глубина вложенности агентов.
```

Для большинства пользователей лучше не увеличивать эти значения без причины.

Почему:

```
больше агентов = больше расход ресурсов;
больше агентов = сложнее собрать общий итог;
больше агентов = выше риск путаницы.
```

15.21 Субагенты и права доступа

Субагенты не отменяют настройки безопасности.

Если основной Codex запущен в режиме:

```toml
sandbox_mode = "workspace-write"
```

то агенты могут работать в рамках этих разрешений.

Если нужен безопасный анализ, лучше задавать самим субагентам:

```toml
sandbox_mode = "read-only"
```

И в запросе писать:

```
Ничего не меняйте в файлах.
Только анализ.
```

Перед большой задачей полезно проверить:

```
/status
```

15.22 Субагенты и стоимость

Субагенты могут расходовать больше ресурсов.

Причина простая:

```
один агент = одна линия работы;
несколько субагентов = несколько линий работы.
```

Для первой большой задачи достаточно:

```
2–3 субагента
```

Не нужно создавать десять ролей, если задача делится на две части.

15.23 Как проектировать своих субагентов

Хороший субагент должен быть узким.

Плохо:

```
super_agent
делает всё: backend, frontend, тесты, документацию и релиз.
```

Хорошо:

```
backend_reviewer
проверяет backend.

test_reviewer
проверяет тесты.

security_checker
```

```
ищет риски безопасности.

docs_researcher
проверяет документацию.
```

Главное правило:

> Один субагент — одна понятная роль.

15.24 Шаблон своего субагента

```toml
name = "agent_name"
description = "Кратко опишите, когда использовать этого субагента."

model_reasoning_effort = "medium"
sandbox_mode = "read-only"

developer_instructions = """
Ты [роль субагента].

Твоя задача:
- [что проверять];
- [на что обращать внимание];
- [какой результат вернуть].

Ограничения:
- [что нельзя делать];
- [какие файлы не менять];
- [когда нужно остановиться].

Формат ответа:
- что проверено;
- что найдено;
- какие файлы важны;
- что рекомендуется сделать.
"""
```

---

16. Хуки

16.1 Что такое хук

Хук — это автоматическое действие, которое Codex запускает в определённый момент своей работы.

Техническое название:

```
hook
```

Простыми словами:

> Хук — это правило: “когда произошло какое-то событие — запусти мой скрипт”.

Примеры:

```
Codex начал сессию → записать это в лог.
Пользователь отправил задачу → проверить текст задачи.
Codex хочет выполнить shell-команду → проверить, безопасна ли она.
Codex завершает работу → запустить финальную проверку.
```

Хуки нужны не для обычного общения с Codex, а для контроля и автоматизации.

16.2 Зачем нужны хуки

Хуки полезны, если вы хотите, чтобы часть правил выполнялась автоматически.

Например:

```
вести журнал действий Codex;
запрещать опасные shell-команды;
проверять текст пользовательского запроса;
автоматически запускать проверку перед завершением;
уведомлять внешнюю систему о начале или конце сессии;
сохранять техническую информацию для диагностики.
```

`AGENTS.md` — это инструкция для Codex.

Хук — это уже автоматический скрипт.

Разница:

```
AGENTS.md
“Codex, пожалуйста, делай так”.

Хук
“Когда происходит событие, автоматически запусти команду”.
```

16.3 Когда хуки нужны

Хуки стоит использовать, когда правило должно выполняться не по памяти, а автоматически.

Хорошие случаи:

```
нужно логировать все сессии;
нужно запрещать опасные команды;
нужно проверять запросы перед запуском;
нужно отправлять уведомление;
нужно запускать обязательную проверку;
нужно собирать аудит действий Codex.
```

Хуки не нужны для первого знакомства с Codex.

Для старта обычно достаточно:

```
config.toml;
AGENTS.md;
skills;
понятных запросов.
```

16.4 Важный статус хуков

Хуки в Codex относятся к продвинутым и активно развивающимся возможностям.

В некоторых версиях их нужно отдельно включать через feature flag:

```
codex_hooks
```

В обсуждениях и issue репозитория OpenAI Codex также видно, что поведение хуков может отличаться между версиями: например, tool-хуки могут срабатывать не для всех типов инструментов, а часть поведения ещё уточняется. Поэтому перед использованием хуков в реальном проекте нужно проверить их именно в вашей версии Codex. (GitHub)

16.5 Как включить хуки

Сначала проверьте, есть ли feature flags:

```
codex features list
```

Если в списке есть:

```
codex_hooks
```

включите:

```
codex features enable codex_hooks
```

После этого перезапустите Codex.

Если включаете вручную через настройки, это обычно делается в:

```
~/.codex/config.toml
```

Пример:

```
[features]
codex_hooks = true
```

После изменения `config.toml` Codex нужно закрыть и открыть заново.

16.6 Где лежит файл хуков

Глобальный файл хуков:

```
~/.codex/hooks.json
```

Проектный файл хуков:

```
my-project/.codex/hooks.json
```

Глобальные хуки относятся к вашей личной среде.

Проектные хуки относятся к конкретному проекту.

Практический вариант для начала:

```
~/.codex/hooks.json
```

Если хук нужен только в одном проекте:

```
my-project/.codex/hooks.json
```

В issue репозитория OpenAI Codex отдельно отмечалось, что runtime фактически исполняет `hooks.json` из config-layer путей, например `~/.codex/hooks.json`; plugin-local `hooks.json` может не запускаться в текущей реализации. (GitHub)

16.7 Какие события бывают у хуков

Событие — это момент, когда Codex может запустить хук.

Часто встречающиеся события:

```
SessionStart
сессия Codex началась.

UserPromptSubmit
пользователь отправил задачу.

PreToolUse
перед использованием инструмента.

PostToolUse
после использования инструмента.

Stop
Codex завершает ход или сессию.
```

Самые понятные для начала:

```
SessionStart
записать старт сессии.

UserPromptSubmit
проверить пользовательский запрос.

Stop
записать завершение.

PreToolUse
проверить команду перед выполнением.

PostToolUse
записать результат после выполнения.
```

Важно: в текущих версиях Codex `PreToolUse` и `PostToolUse` могут работать не для всех инструментов. По открытым issue видно, что они надёжнее покрывают shell/Bash-команды, а для некоторых файловых правок через `apply_patch` поведение может отличаться. (GitHub)

16.8 Базовая структура `hooks.json`

Файл хуков пишется в формате JSON.

Минимальная идея:

```
{
  "hooks": {
    "SessionStart": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "echo session-start >> /tmp/codex-hooks.log"
          }
        ]
      }
    ]
  }
}
```

Что здесь происходит:

```
SessionStart
событие начала сессии.

type = command
запустить команду.

command
какую команду выполнить.
```

После запуска Codex в файл:

```
/tmp/codex-hooks.log
```

должна добавиться строка:

```
session-start
```

16.9 Первый безопасный хук

Для первого теста используйте хук, который только пишет лог.

Создайте файл:

```
nano ~/.codex/hooks.json
```

Вставьте:

```
{
  "hooks": {
    "SessionStart": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "echo '[SessionStart]' >> /tmp/codex-hooks.log"
          }
        ]
      }
    ],
    "Stop": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "echo '[Stop]' >> /tmp/codex-hooks.log"
          }
        ]
      }
    ]
  }
}
```

Затем запустите Codex:

```
codex
```

Выйдите из Codex и проверьте лог:

```
cat /tmp/codex-hooks.log
```

Если вы видите строки:

```
[SessionStart]
[Stop]
```

значит хуки работают.

16.10 Хук на пользовательский запрос

Событие:

```
UserPromptSubmit
```

срабатывает, когда пользователь отправляет задачу.

Пример:

```
{
  "hooks": {
    "UserPromptSubmit": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "echo '[UserPromptSubmit]' >> /tmp/codex-hooks.log"
          }
        ]
      }
    ]
  }
}
```

Такой хук полезен, если нужно фиксировать факт отправки задачи.

Более продвинутый вариант — передавать входной JSON в скрипт и проверять текст запроса. Но начинать лучше с простого логирования.

16.11 Хук перед выполнением команды

Событие:

```
PreToolUse
```

используется перед выполнением инструмента.

Практический смысл:

> Codex собирается выполнить команду — хук может проверить её до запуска.

Например, это нужно, чтобы остановить опасные команды вроде:

```
rm -rf /
```

или подозрительные действия с секретами.

Упрощённый пример логирования:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "echo '[PreToolUse]' >> /tmp/codex-hooks.log"
          }
        ]
      }
    ]
  }
}
```

Для реальной блокировки обычно пишут отдельный скрипт, который читает входные данные, проверяет команду и возвращает решение.

16.12 Хук после выполнения команды

Событие:

```
PostToolUse
```

срабатывает после использования инструмента.

Практический смысл:

```
команда уже выполнена;
можно записать результат;
```

```
можно сохранить лог;
можно проверить stdout/stderr;
можно отправить событие во внешнюю систему.
```

Простой пример:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "echo '[PostToolUse]' >> /tmp/codex-hooks.log"
          }
        ]
      }
    ]
  }
}
```

16.13 Команда-хук и скрипт-хук

Хук может запускать простую команду:

```
{
  "type": "command",
  "command": "echo hello >> /tmp/codex-hooks.log"
}
```

Но для серьёзной логики лучше запускать отдельный скрипт.

Например:

```
{
  "type": "command",
  "command": "python3 ~/.codex/hooks/check_command.py"
}
```

Плюс такого подхода:

```
логика лежит в отдельном файле;
её проще тестировать;
```

```
её проще менять;
hooks.json остаётся коротким.
```

16.14 Пример структуры для хуков

Удобная структура:

```
~/.codex/
├── config.toml
├── hooks.json
└── hooks/
    ├── log_session.py
    ├── check_prompt.py
    └── check_command.py
```

Тогда в `hooks.json` можно ссылаться на скрипты:

```
{
  "hooks": {
    "SessionStart": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "python3 ~/.codex/hooks/log_session.py"
          }
        ]
      }
    ]
  }
}
```

16.15 `timeout`

У хука можно задавать ограничение по времени.

Пример:

```
{
  "type": "command",
  "command": "python3 ~/.codex/hooks/check_command.py",
```

```
  "timeout": 10
}
```

Смысл:

```
timeout = 10
если скрипт выполняется слишком долго, Codex не будет ждать бесконечно.
```

Для хуков лучше делать короткие и быстрые скрипты.

16.16 `statusMessage`

Можно добавить сообщение статуса:

```
{
  "type": "command",
  "command": "python3 ~/.codex/hooks/check_command.py",
  "timeout": 10,
  "statusMessage": "Проверяю команду перед запуском..."
}
```

Это нужно, чтобы было понятно, что делает хук.

16.17 Матчер

Матчер — это условие, при котором хук должен сработать.

Техническое поле:

```
matcher
```

Простыми словами:

> Матчер помогает запускать хук не всегда, а только для подходящего события или инструмента.

Примерная идея:

```
{
  "hooks": {
    "PreToolUse": [
```

```
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "python3 ~/.codex/hooks/check_bash.py"
          }
        ]
      }
    ]
  }
}
```

Такой хук нужен, если вы хотите проверять только shell/Bash-команды.

Точный синтаксис matcher может зависеть от версии Codex, поэтому сначала проверьте самый простой вариант без matcher, а потом усложняйте.

16.18 Как хук получает данные

Командный хук получает данные от Codex через стандартный ввод.

Простыми словами:

> Codex запускает ваш скрипт и передаёт ему JSON с информацией о событии.

Скрипт может:

```
прочитать JSON;
посмотреть, какое событие произошло;
посмотреть команду или контекст;
записать лог;
разрешить действие;
запретить действие;
вернуть сообщение.
```

Для первого уровня не обязательно разбирать весь JSON. Достаточно понимать:

```
Codex передаёт событие скрипту.
Скрипт принимает решение или пишет лог.
```

16.19 Как тестировать хуки правильно

Порядок:

```
1. Включить codex_hooks.
2. Создать простой hooks.json.
3. Использовать только echo в лог.
4. Запустить Codex.
5. Проверить файл лога.
6. Только после этого подключать Python/Node-скрипты.
```

Не начинайте сразу с блокировки команд.

Сначала убедитесь, что сам механизм работает.

16.20 Когда использовать глобальные и проектные хуки

Глобальные хуки:

```
~/.codex/hooks.json
```

Используйте для личных правил.

Например:

```
логировать все сессии;
отправлять уведомления;
запрещать опасные команды везде.
```

Проектные хуки:

```
my-project/.codex/hooks.json
```

Используйте для правил конкретного проекта.

Например:

```
перед завершением запускать тесты проекта;
проверять формат файлов;
запрещать изменение определённых директорий;
записывать аудит только этого репозитория.
```

16.21 Что не стоит делать через хуки

Не стоит сразу использовать хуки для сложной логики.

Плохо:

```
хук сам исправляет код;
хук запускает долгую сборку на 20 минут;
хук меняет много файлов;
хук делает сетевые запросы без необходимости;
хук содержит всю бизнес-логику проекта.
```

Хорошо:

```
хук быстро проверяет;
хук логирует;
хук блокирует явно опасное действие;
хук запускает короткий скрипт;
хук помогает контролировать Codex.
```

Главное правило:

> Хук должен быть коротким, понятным и предсказуемым.

17. Продвинутый режим: рекомендации для автономных длительных задач

17.1 Что такое автономная длительная задача

Автономная длительная задача — это задача, которую Codex выполняет не за один короткий ответ, а через несколько этапов.

Примеры:

```
разобрать большой проект;
сделать крупный рефакторинг;
написать набор тестов;
подготовить релиз;
перенести проект на новую архитектуру;
добавить большую функциональность;
исправить сложный баг в нескольких модулях.
```

В таких задачах Codex должен:

```
изучить проект;
составить план;
```

```
разбить работу на этапы;
внести изменения;
запустить проверки;
исправить ошибки;
обновить документацию;
сообщить итог.
```

Главный риск:

> На длинной задаче агент может потерять цель, забыть ограничения или начать делать лишнее.

Поэтому для таких задач нужна внешняя структура.

17.2 Главный принцип длительной работы

Для короткой задачи достаточно хорошего запроса.

Для длинной задачи одного запроса мало.

Нужны отдельные файлы, которые будут хранить цель, план и статус работы.

Базовая структура:

```
my-project/
├── AGENTS.md
├── Prompt.md
├── Plan.md
├── Implement.md
├── Documentation.md
├── .agents/
│   └── skills/
│       └── ...
└── .codex/
    ├── agents/
    │   ├── backend_reviewer.toml
    │   ├── frontend_reviewer.toml
    │   └── test_reviewer.toml
    ├── hooks.json
    └── hooks/
        ├── log_session.py
        ├── check_prompt.py
        └── check_command.py
```

Простыми словами:

```
Prompt.md
Что нужно сделать.

Plan.md
По каким этапам делать.

Implement.md
Как именно выполнять работу.

Documentation.md
Что уже сделано и какие решения приняты.
```

Эти файлы работают как внешняя память проекта.

17.3 Почему нужна внешняя память

В длинной задаче много информации:

```
исходные требования;
ограничения;
решения;
команды проверки;
ошибки;
промежуточные результаты;
изменённые файлы;
следующие шаги.
```

Если всё это держать только в чате, контекст может размыться.

Файлы решают проблему:

```
Codex может перечитать их;
человек может проверить их;
следующая сессия может продолжить работу;
меньше риск забыть цель;
легче контролировать прогресс.
```

Главное правило:

> Всё важное для длинной задачи должно быть записано в файлы проекта, а не только в переписку.

17.4 `Prompt.md`

`Prompt.md` — это исходная постановка задачи.

Он отвечает на вопрос:

> Что мы вообще делаем?

В нём стоит описать:

```
цель проекта;
исходный контекст;
что должно получиться;
что не должно измениться;
ограничения;
важные файлы;
ожидаемый результат.
```

Пример структуры:

```
# Prompt.md

## Цель

Описать, что нужно сделать.

## Контекст

Почему задача нужна и где она проявляется.

## Ограничения

Что нельзя менять.

## Ожидаемый результат

Что должно получиться в конце.

## Важные файлы

Какие файлы и папки нужно учитывать.
```

`Prompt.md` должен быть стабильным. Его не нужно переписывать после каждого шага.

17.5 `Plan.md`

`Plan.md` — это план выполнения.

Он отвечает на вопрос:

> Как мы будем делать задачу по этапам?

Хороший план состоит из маленьких этапов.

Пример:

```
# Plan.md

## Этап 1. Изучить проект

Цель:
Понять структуру и найти связанные файлы.

Критерий готовности:
Составлена краткая карта проекта.

Проверка:
Ничего не менять в файлах.

## Этап 2. Найти причину ошибки

Цель:
Определить, где возникает проблема.

Критерий готовности:
Есть объяснение причины и список файлов для изменения.

## Этап 3. Внести минимальное исправление

Цель:
Исправить ошибку без лишнего рефакторинга.

Критерий готовности:
Изменения внесены, diff минимальный.

Проверка:
Запустить подходящие тесты.
```

План должен быть не красивым, а рабочим.

17.6 Что должно быть в каждом этапе

Каждый этап лучше оформлять одинаково:

```
Название этапа
Цель
Что сделать
Что нельзя делать
Критерий готовности
Команда проверки
```

Пример:

````
## Этап 2. Добавить тесты

Цель:
Покрыть сценарий истечения access token.

Что сделать:
Добавить тесты в `tests/auth`.

Что нельзя делать:
Не менять production-код на этом этапе.

Критерий готовности:
Тесты добавлены и падают на текущей ошибке.

Проверка:
```bash
pytest tests/auth
```
````

Так Codex проще работать последовательно.

17.7 `Implement.md`

`Implement.md` — это инструкция по стилю выполнения.

Он отвечает на вопрос:

> Как именно Codex должен работать над задачей?

Туда можно записать правила:

```
работать маленькими шагами;
перед изменениями объяснять план;
```

```
не переходить к следующему этапу, пока проверки не прошли;
не добавлять зависимости без причины;
обновлять Documentation.md после этапа;
останавливаться при неясности.
```

Пример:

```
# Implement.md

## Правила выполнения

- Работай по этапам из `Plan.md`.
- Перед началом этапа перечитай `Prompt.md`.
- Не переходи к следующему этапу, пока текущий не завершён.
- После изменения кода запускай проверки из этапа.
- Если проверка упала, сначала исправь ошибку.
- После завершения этапа обнови `Documentation.md`.
- Не добавляй зависимости без отдельного объяснения.
```

17.8 `Documentation.md`

`Documentation.md` — это журнал работы.

Он отвечает на вопрос:

> Что уже сделано?

В нём нужно фиксировать:

```
завершённые этапы;
изменённые файлы;
принятые решения;
найденные проблемы;
результаты проверок;
что осталось сделать.
```

Пример:

```
# Documentation.md

## Статус

Текущий этап: Этап 2. Найти причину ошибки.

## Сделано
```

````
- Изучена структура `app/auth`.
- Найден связанный тестовый файл `tests/auth/test_refresh.py`.
- Обнаружено, что refresh token не проверяется после истечения access token.

## Решения

- Исправление делать в `app/auth/refresh.py`.
- Публичный API не менять.

## Проверки

```bash
pytest tests/auth
```

Результат: падает один тест, ошибка подтверждена.

## Следующие шаги

- Внести минимальное исправление.

- Повторно запустить `pytest tests/auth`.
````

`Documentation.md` особенно полезен, если работа продолжается на следующий день или в новой сессии.

17.9 Как связать эти файлы с `AGENTS.md`

Чтобы Codex не забывал использовать эти файлы, добавьте правило в `AGENTS.md`.

Пример:

```
## Правила для длительных задач

Если в проекте есть `Prompt.md`, `Plan.md`, `Implement.md` и `Documentation.md`,
используй их как рабочую память задачи.

Перед началом работы:
- прочитай `Prompt.md`;
- прочитай `Plan.md`;
- прочитай `Implement.md`;
- проверь текущий статус в `Documentation.md`.
```

```
Во время работы:
- выполняй только текущий этап из `Plan.md`;
- не переходи дальше, пока критерий готовности не выполнен;
- запускай проверки, указанные в этапе;
- после завершения этапа обновляй `Documentation.md`.

В конце ответа:
- укажи, какой этап выполнен;
- какие файлы изменены;
- какие проверки запущены;
- что делать следующим шагом.
```

Так Codex получает устойчивое правило работы.

17.10 Как запускать длительную задачу

Плохой запрос:

```
Сделай весь проект нормально.
```

Лучше:

```
Это длительная задача.

Сначала прочитай:
- Prompt.md
- Plan.md
- Implement.md
- Documentation.md

После этого:
1. Определи текущий этап.
2. Выполни только этот этап.
3. Запусти проверки из Plan.md.
4. Обнови Documentation.md.
5. В конце кратко скажи, что сделано и какой следующий шаг.

Не переходи к следующему этапу без завершения текущего.
```

17.11 Режимы безопасности для длительных задач

Для длительных задач не стоит сразу включать максимальную автономность.

Безопасный старт:

```toml
approval_policy = "on-request"
sandbox_mode = "workspace-write"

[sandbox_workspace_write]
network_access = false
```

Для анализа без изменений:

```toml
approval_policy = "on-request"
sandbox_mode = "read-only"
```

Автономный режим можно использовать только в контролируемой среде:

```toml
approval_policy = "never"
sandbox_mode = "workspace-write"

[sandbox_workspace_write]
network_access = false
```

Важно:

> `approval_policy = "never"` не должен быть первым режимом для новой большой задачи.

Сначала лучше пройти один-два этапа с подтверждениями и только потом решать, можно ли давать больше автономности.

17.12 Когда использовать `read-only`

Режим `read-only` полезен в начале большой задачи.

Он означает:

> Codex может читать файлы, но не должен их менять.

Используйте его для этапов:

```
изучить проект;
найти связанные файлы;
составить план;
проверить архитектуру;
сделать ревью;
найти риски.
```

Пример задачи:

```
Работай в режиме анализа.
Изучи проект, составь карту важных файлов и предложи Plan.md.
Ничего не меняй в коде.
```

17.13 Когда использовать `workspace-write`

Режим `workspace-write` нужен, когда Codex должен менять файлы проекта.

Используйте его после того, как:

```
цель понятна;
план согласован;
ограничения записаны;
команды проверки известны;
есть критерии готовности.
```

Пример задачи:

```
Выполни Этап 3 из Plan.md.
Разрешено менять только файлы, указанные в этапе.
После изменений запусти проверку и обнови Documentation.md.
```

17.14 Когда использовать субагентов

Субагенты полезны на этапах анализа.

Например:

```
один субагент изучает backend;
второй изучает frontend;
третий смотрит тесты;
четвёртый проверяет документацию.
```

Хороший запрос:

```
Используй субагентов только для анализа.

Один пусть изучит backend.
Второй — frontend.
```

```
Третий — тесты.

Ничего не меняйте.
В конце собери общий вывод и предложи обновление Plan.md.
```

Для первого использования не просите субагентов сразу менять код.

17.15 Когда использовать MCP

MCP полезен, если для длительной задачи нужна внешняя информация.

Например:

```
актуальная документация;
GitHub issue;
внутренняя база знаний;
система задач;
спецификация API;
документация OpenAI.
```

Пример:

```
Перед изменением конфигурации проверь актуальную документацию через MCP
openaiDeveloperDocs.
После этого обнови Plan.md, если нужно.
```

17.16 Когда использовать хуки

Хуки полезны, если нужно автоматически контролировать длительную работу.

Например:

```
логировать начало и конец сессии;
запрещать опасные команды;
запускать короткую проверку;
собирать аудит;
проверять, что Documentation.md обновлён.
```

Не начинайте с хуков. Сначала наладьте ручной процесс:

```
Prompt.md
Plan.md
Implement.md
Documentation.md
AGENTS.md
```

Хуки стоит добавлять, когда процесс уже понятен и хочется его автоматизировать.

17.17 Как не дать Codex уйти в сторону

Для длинной задачи важны ограничения.

Записывайте их явно:

```
не менять публичный API;
не добавлять зависимости;
не переписывать архитектуру без отдельного этапа;
не трогать файлы вне указанной области;
не переходить к следующему этапу без проверки;
не удалять тесты ради прохождения проверки.
```

Эти ограничения лучше продублировать в:

```
Prompt.md
Plan.md
Implement.md
AGENTS.md
```

Да, это повторение. Но для длинной задачи оно полезно.

17.18 Как правильно завершать этап

После каждого этапа Codex должен вернуть короткий итог.

Формат:

```
Этап:
что выполнялось.

Сделано:
список изменений.
```

```
Файлы:
что изменено.

Проверки:
какие команды запущены и результат.

Проблемы:
что осталось неясным или сломанным.

Следующий шаг:
какой этап делать дальше.
```

Пример:

```
Этап:
Этап 3. Внести минимальное исправление.

Сделано:
Исправлена проверка refresh token.

Файлы:
app/auth/refresh.py
tests/auth/test_refresh.py

Проверки:
pytest tests/auth — passed.

Проблемы:
Не обнаружены.

Следующий шаг:
Обновить документацию по auth flow.
```

17.19 Когда останавливать Codex

Не нужно заставлять Codex “делать до победного” в любой ситуации.

Остановить работу нужно, если:

```
требования противоречат друг другу;
проверки падают по неизвестной причине;
нужно изменить архитектуру;
нужно добавить зависимость;
нужно менять публичный API;
```

```
Codex не уверен в причине ошибки;
задача вышла за пределы текущего этапа.
```

В `Implement.md` можно записать:

```
Если для продолжения нужно изменить архитектуру, добавить зависимость или
поменять публичный API — остановись и запроси подтверждение.
```
