ChatGPT

OpenAI Codex CLI – Comprehensive Guide

Overview

OpenAI **Codex CLI** is an open-source command-line tool that serves as a local AI coding assistant [1]. It leverages OpenAI's latest reasoning models to help developers write, modify, and understand code directly from the terminal. Unlike cloud-based coding agents, Codex CLI runs entirely on your machine – reading and editing files locally – so your code stays private unless you choose to share it [2]. This tool effectively brings ChatGPT-level coding assistance into your workflow, enabling "chat-driven" development in the terminal.

Key Features:

- **Zero-Setup Installation:** Codex CLI is easy to install with a single command (`npm install -g @openai/codex` or via Homebrew) and minimal configuration [3]. Once installed, you just need to provide an OpenAI API key (or sign in with ChatGPT) to start using it.
- **Terminal-Native Interface:** All interactions occur in your terminal via a text-based UI, so you can integrate AI assistance into your development workflow without context-switching [4]. You can ask Codex to perform tasks in natural language, and it will read files, propose code changes, run tests, etc., all within the terminal.
- **Multimodal Inputs:** In addition to text prompts, Codex CLI can accept other inputs like screenshots or diagrams, interpreting visual information to generate or edit code accordingly [5]. *(For example, the CLI demo shows generating an app from a UI screenshot.)*
- **Flexible Approval Modes:** You control how autonomous the agent is. Codex CLI offers three modes – Suggest, Auto Edit, and Full Auto – that govern what actions the AI can take on its own [6] [7]. In Suggest mode (default), it only **suggests** changes; in Auto Edit, it can apply file edits automatically; in Full Auto, it can modify code and execute commands in a sandbox without user approval.
- **Local Execution & Sandbox:** Codex can run code and shell commands locally. In Full Auto mode, it uses a sandbox (with no network access and directory scoping) to execute commands safely [8] [9]. On macOS, it leverages the built-in sandbox (`sandbox-exec`), and on Linux it can run inside a Docker-based container for isolation [10] [11]. This allows the AI to run tests, linters, or the application code and use the results to inform its next steps, all under your control.
- **No Cloud Code Uploads:** Because the CLI operates on your local environment, your repository's code is not sent to OpenAI's servers. Only high-level context (like your prompt, diffs, or summary info) is sent for generating a response [12]. This addresses security concerns; you get AI assistance while keeping proprietary code on your machine.
- **Open-Source and Extensible:** The Codex CLI is open-sourced on GitHub [13] under the Apache-2.0 license, inviting developers to inspect, contribute, or fork it. OpenAI even launched a $1M fund to support open-source projects using Codex CLI [14], underscoring its commitment to community-driven development.
- **Cross-Platform Support:** Officially, Codex CLI supports macOS and Linux. Windows usage is possible via WSL2 (Windows Subsystem for Linux) [15] [16]. The only prerequisites are a recent Node.js runtime (v22+), npm (v10+), and Git for full functionality [17] [18].

- **Model Flexibility:** By default Codex CLI uses OpenAI's fast reasoning model **o4-mini** (which is optimized for coding tasks), but you can specify any model available in the OpenAI API (e.g. GPT-4.1) using a flag or configuration [19] [20]. It even supports other providers and open-source models via configuration (more on that in Configuration below).

In summary, Codex CLI acts like a smart pair programmer integrated into your terminal – it can analyze your codebase, make code changes, run and verify code, and iteratively help you build and fix software, all through natural language conversations.

CLI Commands and Help Reference

The Codex CLI provides a few primary commands and options. Running `codex --help` in your terminal will display the full usage information (including subcommands, flags, and environment settings). Below is an overview of each available command and flag, along with explanations:

- `codex` (no args): Launches the interactive Text User Interface (TUI) for a chat session [21]. This is the default mode where you enter a conversational loop with the Codex agent. You'll see a prompt like `>_` waiting for your instructions. In this mode, Codex will scan your project (the current directory) and possibly load context (like any `AGENTS.md` files – discussed later) before responding. Use this when you want a back-and-forth conversational coding session.
- `codex "Your request here"`: You can also invoke the interactive TUI with an **initial prompt** provided on the command line [21]. For example: `codex "Explain this repo to me."` will start Codex and immediately ask it to explain the codebase. This saves a step by feeding the first query directly. After the CLI prints the answer, you remain in the interactive session to continue the conversation.
- `codex exec "Your request here"`: Runs Codex in a **non-interactive automation mode**, executing a single task and then exiting [21]. This is useful for scripting or CI pipelines. For instance: `codex exec --full-auto "update the CHANGELOG for the next release"` will instruct Codex to attempt that task, applying changes automatically (because `--full-auto` was used) and then terminate [22]. The output (proposed diffs, command logs, etc.) is printed to the console. This mode is great for one-shot operations or integrating Codex into automated workflows (see CI usage example below).
- `codex login`: Starts an authentication flow to link Codex CLI with your OpenAI ChatGPT account. This command opens a browser to authenticate via ChatGPT (requires a **Plus, Pro, or Team** plan) and then stores an auth token locally [23] [24]. Using `codex login` lets ChatGPT subscribers use Codex with the models included in their plan (including the latest GPT-5 model) **without needing API keys or incurring extra usage fees** [23]. After logging in once, subsequent uses of Codex CLI will use your ChatGPT-linked credentials (stored in `~/.codex/auth.json`). *(Note: If you previously used an API key, you may need to delete the old auth file and update Codex CLI to ≥v0.20.0 before using the new login flow [25].)*
- `codex logout`: Logs out of the ChatGPT-linked account (removing the saved credentials). After this, you can switch to a different account or use an API key. *(This command may not have appeared in early documentation, but is available in recent versions to manage auth.)*
- `codex --upgrade`: Self-updates the Codex CLI to the latest release [26]. This is equivalent to re-running the npm install. Use this periodically to get improvements and bug fixes (the CLI is evolving quickly).

- `codex --version`: Shows the current version of the Codex CLI you have installed (e.g. `0.21.0`). This is helpful to verify if you're up-to-date or to include in bug reports.
- `codex mcp` (advanced): Launches Codex CLI as a **Model Context Protocol (MCP)** server [27]. MCP is a standardized way for tools to expose "AI agent" functionality. Running `codex mcp` allows Codex to act as a backend for MCP-compatible clients (like the Model Context Protocol Inspector). In this mode, Codex registers a single tool (`codex`) that can accept various inputs and configuration overrides via the MCP interface [27]. This is an experimental feature for advanced integrations and likely not needed for everyday use.

Common Flags:

- `--suggest`, `--auto-edit`, `--full-auto`: These flags set the **approval mode** at startup (overriding the default Suggest mode). Use `--auto-edit` to allow automatic file edits, or `--full-auto` to allow fully autonomous changes (within the sandbox) [28]. You can also explicitly specify `--suggest` to force suggestion-only mode. If none of these flags are provided, Codex starts in Suggest mode by default.
- `-m <model>`, `--model <model>`: Specifies which model to use for the AI assistant [21]. For example, `codex -m gpt-4.1` would use the GPT-4.1 model. By default it uses `o4-mini` (OpenAI's Codex-1 model optimized for coding) [20]. You can set a permanent default in the config file as well. Make sure your account has access to the model you request.
- `-a`, `--ask-for-approval`: Forces Codex to ask for user confirmation **before** executing any file edits or shell commands, even in modes that would normally auto-apply changes. This can serve as a safety check if you want to review each action. In Suggest mode this is implicit (since it never auto-executes), but in Auto Edit mode, using `-a` will make it behave more cautiously (prompting you for each edit).
- `-q`, `--quiet`: Runs Codex in "quiet" mode. This suppresses some of the verbose streaming output and chain-of-thought, showing only the final answers or results. Quiet mode is useful if you want cleaner output (e.g., logging just the final diff in a script). *Note:* if you use non-OpenAI providers, you still need an OpenAI API key set due to a known quirk in quiet mode [29] [30].
- `--config <KEY>=<VALUE>`: Overrides configuration values for this run (instead of using the config file defaults). For example, `codex --config model=gpt-4.1 --config reasoningEffort=high` would override those settings just for this invocation. This is helpful for quick experiments with config options without editing the file [31].
- `--no-ansi` / `--ansi`: Toggles ANSI color output in the terminal. If you're piping output to a file or on a CI system that doesn't handle colors, you might use `--no-ansi` to disable colored diffs and prompts.

(To see all flags, run `codex --help`. The CLI's help text will show usage examples and a list of global options. For subcommands like `exec` or `login`, you can also run `codex <subcommand> --help`.)

Interactive Session Commands: When you are inside the interactive Codex TUI (after running `codex` without `exec`), there are special **slash commands** you can use to control the session or get information:

- `/init` – Generate a starter `AGENTS.md` file in the current project [32] [33]. This command is very useful when you start a new project with Codex: it creates a template **AGENTS.md** with sections for you to fill in (project description, guidelines, commands, etc.). It basically helps you set "house rules" for Codex in your project.

- `/status` – Show the current session status, including which model is in use, the mode (Suggest/Auto/etc.), how many tokens have been used, and other configuration info [32]. It's a quick way to verify your settings and usage.
- `/diff` – Display the current git diff of your project (files changed in this Codex session) [32]. It even shows untracked file changes. This lets you review what modifications Codex has proposed or applied so far. After Codex makes suggestions or edits, you can type `/diff` to see the patch before deciding to accept or refine it.
- `/prompts` – Show some example prompts and usage tips [32]. This can give you ideas on how to ask Codex for various tasks.
- `/mode` – Toggle or set the approval mode within the session (e.g., `/mode auto-edit` to switch to Auto Edit on the fly) [34]. Codex will confirm the mode change. You can use this to dynamically adjust how autonomous it should be, without restarting.
- `/help` – (If available) Lists the in-session commands and possibly a short description of each. This is a quick reference if you forget the available "slash" commands. *(In newer versions, simply typing `/` might also show an auto-complete or list of commands.)*
- `Ctrl+C` – (Keyboard interrupt) If Codex gets stuck or is taking too long on a step, pressing Ctrl+C will cancel the current generation or command execution [35]. You can then either retry or ask Codex to continue. This is essentially a panic button to regain control. Use it if the agent seems unresponsive or you want to stop its current thought process.

Approval Modes Explained

Codex CLI allows you to choose how much autonomy to grant the AI when it comes to modifying your code or running commands. These **Approval Modes** are central to using the CLI effectively and safely. You can set the mode at launch (via flags) or change it during a session (via `/mode`). Here's a detailed look at each mode:

- **Suggest (Read-Only) – *Default Mode*:** In **Suggest** mode, the agent can **read** your files and propose changes or command executions, but it **cannot apply or run anything without your explicit approval** [36]. Whenever Codex suggests editing a file, it will show you a diff and ask `[y/n]?` before making the change. Similarly, if it suggests running a shell command (like `npm test`), it will ask for confirmation. Use this mode for safe exploration, code reviews, and learning a new codebase [37]. It ensures nothing happens unless you agree, so it's the least risky mode.
- **Auto Edit (Read & Write):** In **Auto Edit** mode, Codex can both read and **write** to files automatically, without asking for permission for each edit [38]. This means if you prompt it to "refactor function X," it can directly apply the refactor to the file(s). However, **Auto Edit still asks for approval before executing any shell commands or potentially destructive actions** [38]. This mode is useful for tasks like refactoring, applying repetitive changes, or other code modifications where you want to speed things up but still keep an eye on side effects. You might choose Auto Edit when you trust the agent to make straightforward code changes but you're not ready to let it run arbitrary commands without oversight [39].
- **Full Auto (Read, Write & Execute):** In **Full Auto** mode, Codex operates fully autonomously within a sandboxed environment [8]. It can read and write files and also execute commands on its own, all **without asking for user approval at each step**. This is the most powerful (and risky) mode. Codex will attempt to complete your task end-to-end: it might edit code, run tests or build commands, re-edit code based on results, etc., cycling until the task is done or a time limit is reached. All this happens in a restricted sandbox: the agent's process is jailed to your current directory (plus a temp

and config dir) and **outbound network access is disabled** by default [10] [11]. Use Full Auto for longer-running tasks like debugging complex issues, fixing a broken build, or prototyping a feature while you literally grab a coffee [40]. *Always review the results afterward*, since the agent might make broad changes.

- **Auto-Approval Override:** Regardless of mode, you can force Codex to behave more cautiously by using the `--ask-for-approval` flag (`-a`) when launching, which will require confirmation for each action [41]. Conversely, if you start in Suggest mode but want to auto-approve a specific action, you can usually just type "yes" to all suggestions, or switch mode to Auto Edit temporarily.
- **Safety Tip:** Codex CLI will warn you if you try to use Auto Edit or Full Auto in a directory that is not under version control (i.e., not a Git repo) [42]. This is a reminder to initialize Git or create a checkpoint before letting the AI make many changes. It's highly recommended to keep your project in Git and commit changes before and after using Full Auto, so you can track exactly what happened and revert if needed.

When to Use Each Mode:
- *Suggest* – Use this by default, especially when exploring unfamiliar code or if you want to vet every change. It's ideal for Q&A about the codebase, generating examples, or any scenario where safety is a priority.
- *Auto Edit* – Use when performing a series of mechanical refactors or adding boilerplate code. It speeds up the workflow by skipping confirmation for file edits, but still gives you a chance to stop something dangerous (since commands aren't auto-run).
- *Full Auto* – Use sparingly, for complex tasks that involve multiple iterations or executing the code. It shines in scenarios like: "run the tests and fix whatever fails" or "build this feature completely." Be prepared to monitor output and possibly intervene if it goes off-track. Always review the final diff (`/diff`) or output.

You can always downgrade autonomy if needed. For example, if you start a session in Full Auto and realize you want more control, use `/mode suggest` to drop to manual approvals.

Using `AGENTS.md` (and `AGENTS.local.md`) for Persistent Guidance

A powerful feature of Codex CLI is the ability to give the AI **contextual guidance about your project** through special markdown files. These files act like a persistent "instruction manual" for the AI, so you don't have to repeat the same context in every prompt. The two relevant files are `AGENTS.md` and `AGENTS.local.md`. Let's discuss their purpose and differences:

`AGENTS.md` – Project and Global Instructions

Purpose: `AGENTS.md` is a Markdown file where you provide high-level context, constraints, and instructions for your coding assistant [43]. You can think of it as analogous to a README, but addressed to the AI agent. It might include an overview of the project architecture, coding style guidelines, important commands to run, dependency setup, testing instructions, and any conventions or rules the AI should follow.

Usage: Simply create an `AGENTS.md` file and add whatever information might help Codex understand your project and your preferences. Here's how Codex CLI uses `AGENTS.md`:

- It **automatically looks for `AGENTS.md` in certain locations** whenever it runs, and merges all found instructions into its context [44]. The precedence (from highest priority to lowest) is:

- **Local (current directory)** – If you have an `AGENTS.md` in the folder you launch Codex from, it will read that first (consider these feature-specific or sub-folder rules) [44].
- **Repo Root** – If you are in a git repository and there's an `AGENTS.md` at the root of the repo, Codex will also read that (these are project-wide shared notes) [44].
- **Global (`~/.codex/AGENTS.md`)** – Lastly, Codex checks your home directory `~/.codex/AGENTS.md` for personal global guidance that applies to all projects [44]. This is optional; you can use it to set your general preferences (like always prefer certain libraries, or your typical code style).

The contents of all applicable files are **merged together (top-down)** into one set of instructions. Local folder instructions override or add to repo instructions, which override global ones [44]. This lets you have layered guidance: e.g., global defaults, plus project-specific rules, plus any special cases in a subfolder.

What to include in AGENTS.md: Anything that helps the AI. For example: project description and scope, architecture diagrams (even textually described), coding styleguide or lint rules, testing approach (and how to run tests), definitions of domain-specific terms, API keys or URLs (if safe to include), and command instructions (like "to run the app, use `npm start`; to run tests, use `pytest`"). You can also include **lists of tasks or TODOs**, which Codex might use as a checklist. A good AGENTS.md might be structured into sections, such as **Project Overview**, **Design Principles**, **Commands (always run)**, **Conventions**, **Environment Variables**, etc., similar to the example below:

```
# AGENTS.md

## Project
MyApp: A Node.js web service for XYZ. Uses Express and MongoDB.

## Design Guidelines
- Follow MVC pattern. Separate business logic from controllers.
- Use async/await for promises; no callbacks.

## Commands (always run)
- Install: `npm install`
- Test: `npm test`
- Lint: `npm run lint`
- Start: `npm start`

## Conventions
- 4 spaces indent.
- All new features must include unit tests.
- API endpoints should follow REST conventions, etc.

## Env (configure, do not hardcode)
`DB_URI`, `API_KEY`, `PORT`
```

The above is just illustrative. When Codex runs, it will ingest these notes. This often leads to better outcomes: the AI will know how to run tests or what style to follow without being told explicitly each time.

As the OpenAI blog noted, *"Like human developers, Codex agents perform best when provided with configured dev environments, reliable testing setups, and clear documentation."* [43] [45]

Creating AGENTS.md quickly: You can use the interactive command `/init` at the start of a Codex session to auto-generate a skeleton AGENTS.md [32]. This will open an editor (in the CLI) with some headings pre-filled (Project, Design, Commands, etc.) for you to fill in. It's a handy way to get started.

`AGENTS.local.md` – Personal/Private Instructions

Purpose: `AGENTS.local.md` is not an official Codex CLI feature per se, but rather a *convention* some developers use to maintain **private or machine-specific instructions** separate from the main `AGENTS.md`. The idea is to have a file for any guidance you *don't want to commit to the repository* – for example, personal preferences, experimental rules, or sensitive info like internal URLs or API tokens. By keeping these in `AGENTS.local.md`, you can exclude it from version control and still benefit from the instructions locally.

Difference from AGENTS.md: The key difference is in **intended audience and version control**. `AGENTS.md` (in the repo) is meant to be shared with all collaborators and committed to git, so it should contain team-approved guidelines. `AGENTS.local.md` is meant just for you (the individual developer) or for local temporary notes, and typically is **git-ignored**. In fact, tools like *DotAgent* (a third-party AI rules manager) automatically suggest ignoring any `*.local.md` files in the repo [46] [47]. This prevents private instructions from leaking into the repo.

Usage: If you decide to use `AGENTS.local.md`, you'll need to incorporate it manually, since Codex CLI by default does not automatically read a file by that name (it only looks for `AGENTS.md`). You have a couple of options: - **Manual concatenation:** You can copy or include the contents of `AGENTS.local.md` into your prompts or just remember those rules yourself when prompting. (Not ideal.) - **Leverage global AGENTS.md:** A better approach is to use the global `~/.codex/AGENTS.md` for any personal rules you want applied everywhere, rather than per-project local file. Essentially, `AGENTS.local.md` can be simulated by putting content in your global file that is conditional or relevant to that project only. Of course, this global file *is* loaded by Codex CLI. - **Use DotAgent or a Pre-processing Script:** Advanced users might use a script to merge `AGENTS.md` and `AGENTS.local.md` before launching Codex, or a tool like DotAgent which understands various formats and can export a combined file. DotAgent, for example, treats any file named `AGENTS.local.md` as a *private rule file* that should not be exported or shared [48]. You could imagine extending Codex CLI's functionality to load `.local.md` files, but as of now, that's not built-in.

Typical contents: In `AGENTS.local.md`, you might put things like: notes about experimental branches, client-specific hacks, or personal API keys/placeholders. For instance, if you're working with an unreleased library, you might note usage instructions here that you don't want in the official docs. Or perhaps you prefer a different code commenting style than the team – you could remind Codex of that in your local file.

Summary: *Use* `AGENTS.md` *to share consistent project guidelines with Codex (and your team). Use* `AGENTS.local.md` *as a local-only supplement if needed – but remember that by default Codex CLI doesn't load it, so you will maintain it purely for your own reference or via custom tooling.* In most cases, a well-maintained `AGENTS.md` (global and/or repo) is sufficient.

Configuration, Environment Variables, and Usage Patterns

Codex CLI is configurable to suit your preferences and environment. It reads configuration from both environment variables and a config file, and it supports multiple AI providers.

Config File (`~/.codex/config.toml` or `.yaml`)

By default, Codex loads its config from a TOML file located at `~/.codex/config.toml` [49]. (It will create a stub on first run, if not present.) The config uses a simple key = value format. Newer versions of Codex CLI also accept JSON or YAML config if the file is named accordingly (e.g., `config.json` or `config.yaml` in `~/.codex`). Some users have reported using JSON or YAML successfully [50] [51] – the CLI likely auto-detects format by extension.

Here are some **common configuration settings** you might find in `config.toml`:

- `model`: Default model to use for completions (e.g. `"o4-mini"` or `"gpt-4.1"`). Instead of specifying `-m` each time, you can set it here [20].
- `model_provider`: Which API provider to use. Default is `"openai"`, but you can configure others (see *Multiple Providers* below).
- `provider` (older key): Some older docs use `"provider"` to denote the same as `model_provider` – for example, `"provider": "gemini"` to use Google's Gemini API [52]. In TOML you'd use `model_provider = "gemini"`.
- `api_key`: You generally don't put API keys in the config file (environment vars are safer), but if you wanted, you could include an OpenAI API key or others here.
- `reasoningEffort`: Sets how "hard" the model should think (e.g., `"medium"` which is default, or `"high"`). A higher effort might use more chain-of-thought or take longer to respond. This concept comes from OpenAI's Codex-1 configuration for how much reasoning to apply.
- `history.maxSize` / `history.saveHistory`: Controls conversation history length and whether to save it. For example, you might keep the last N turns to maintain context. The default max tokens for history might be ~10000 [53], and `saveHistory` can be true/false if you want to store past sessions.
- `disable_response_storage`: If you are using an OpenAI organization with **Zero Data Retention (ZDR)**, set `disable_response_storage = true` [54] [55]. This ensures Codex doesn't send an ID of a previous message with requests (which ZDR disallows). If you get errors about "Previous response cannot be used... due to Zero Data Retention," this setting is needed [56].
- **Approval mode defaults**: You can fine-tune things like what happens on errors in Full Auto. E.g., a setting `fullAutoErrorMode = "ask-user"` could instruct Codex to pause if a command fails, instead of continuing [57].
- **Notifications**: There is an option `notify = true` which enables desktop notifications on certain events [57]. This might, for instance, pop up a system notification when Codex finishes a long task or needs your input.
- **Profiles**: The config supports defining multiple profiles under a `[profiles]` section [58] [59]. Each profile can specify a model, provider, and other settings. For example, you could define:

```
[profiles.gpt4]
model = "gpt-4.1"
```

```
model_provider = "openai"

[profiles.mistral]
model = "mistral"
model_provider = "ollama"
```

Then you can launch Codex with `--profile gpt4` or `--profile mistral` to quickly switch configurations [59] [60]. Profiles are great for swapping between a local model and OpenAI, or between different model sizes.

- **MCP Servers**: If using the Model Context Protocol integration (e.g., Snyk's security scanner), you can add an `[mcp_servers]` section in config. For instance, to add the Snyk security MCP, you'd include:

```
[mcp_servers.snyk-security]
command = "npx"
args = ["-y", "snyk@latest", "mcp", "-t", "stdio"]
```

as per Snyk's documentation [61] [62]. This will allow Codex to spawn and use that external tool when needed (the AI can invoke it as part of its chain-of-thought).

- **Open-Source Model Support**: Under `model_providers` in config, you can define custom endpoints. For example, to use **OpenRouter** (an OpenAI-compatible proxy with other models) [63]:

```
[model_providers.openrouter]
name = "OpenRouter"
base_url = "https://openrouter.ai/api/v1"
env_key = "OPENROUTER_API_KEY"
```

This tells Codex how to call OpenRouter and which environment variable holds the API key [63]. You might also define providers for Azure OpenAI service, Ollama (for local models), etc., each with their base URL, any required headers, and any special `wire_api` settings (e.g., `wire_api = "responses"` for Azure which uses the "Responses" API format) [64] [65].

Remember to **reload Codex** after changing the config file for changes to take effect. Also, `codex --help` lists available config keys (without the `--` prefix) in case you need the exact names [66].

Environment Variables

Several environment variables are recognized by Codex CLI or its integrations:

- `OPENAI_API_KEY`: *Your OpenAI API key.* Set this if you plan to use the CLI in API mode (not logged in via ChatGPT) [67]. Export it in your shell (`export OPENAI_API_KEY="sk-..."`) or store in a `.env` file. **Plus/Pro users:** If you use `codex login`, you don't need this for OpenAI usage, but it doesn't hurt to have it as a fallback.
- `OPENROUTER_API_KEY`, `GEMINI_API_KEY`, **etc.:** If using alternative providers, set their keys accordingly [68] [69]. For example, if `model_provider = "openrouter"`, ensure

`OPENROUTER_API_KEY` is in your env. If using Google's Gemini via their PaLM API (through an OpenAI-compatible endpoint), set `GEMINI_API_KEY` as shown in config and environment.

- `CODING_ENV` **or others:** Codex doesn't specifically require these, but some users set custom env flags to indicate environment. Notably, Codex will automatically load a `.env` file in your project root if present [70]. This is done via the Node `dotenv/config` mechanism, meaning any variables in `.env` get loaded into the environment when Codex runs. This is super useful – for instance, you can put `OPENAI_API_KEY=...` in a local `.env` so you don't have to export it globally, or include environment-specific variables that your code uses (so that when Codex runs your code, those env vars are set).
- `RUST_LOG`: Codex CLI is written in Rust (for the core and TUI), and it uses the `env_logger` system. Setting `RUST_LOG` enables verbose logging. For example, `export RUST_LOG=codex_core=debug,codex_tui=info` will log debug info from the core and info from the TUI [71]. By default, interactive mode sets `RUST_LOG=codex_core=info,codex_tui=info` and writes logs to `~/.codex/log/codex-tui.log` [71]. You can `tail -F ~/.codex/log/codex-tui.log` in another terminal to watch what the AI is doing behind the scenes. In non-interactive `codex exec`, the default log level is `error` (only showing errors) with output printed to console [72]. Adjusting `RUST_LOG` can help with debugging issues or performance.
- `NO_COLOR`: If this variable is set, the CLI may disable colored output (a common convention in CLI apps). Use it if you're getting odd characters in outputs or don't want ANSI colors.
- **Platform-specific**: On Windows, if using WSL, ensure your display or browser is accessible for `codex login` (you might need to copy a URL to a Windows browser). On Mac/Linux, if you want the login to auto-open a browser, having `BROWSER` env set can guide which browser to use.

Typical Usage Patterns

Interactive Development Workflow: A common pattern is: 1. Navigate to your project directory (at the root of your repo). 2. Run `codex` (maybe in Suggest mode initially). 3. Codex will likely scan your repo (this can take a minute or two for large projects on first query) and maybe present a greeting or tips. 4. Ask Codex a high-level question or give it a task. e.g. "Explain the purpose of each module in this repository," or "Add a new API endpoint for user profile update." 5. Review Codex's suggestions. If in Suggest mode, it will provide a plan or diff and await approval. 6. Approve changes (`y`) or reject (`n`) or edit the suggestions. You can also provide feedback like, "No, don't remove that function, just modify it to do X." 7. Iterate – continue the conversation, possibly switching to Auto Edit for a batch of trivial edits, then maybe to Full Auto to run tests. 8. Use `/diff` frequently to see what's been changed. Use your git tools as well (nothing prevents you from running `git diff` in another window). 9. End the session with `Ctrl+C` or by typing `/exit` (if that's supported, or simply closing the terminal). Save and commit your changes in Git.

One-Shot Scripted Tasks: Using `codex exec`, you can automate tasks: - Example: In a CI pipeline (GitHub Actions, etc.), you might have a step:

```
- name: Update changelog via Codex
  run: |
    npm install -g @openai/codex
    export OPENAI_API_KEY=${{ secrets.OPENAI_KEY }}
    codex exec --full-auto "update CHANGELOG for next release"
```

This would have Codex analyze your repo's commits or state and try to update a CHANGELOG file accordingly [22] . In this mode, since it's Full Auto, it will make the edits and exit. You could then `git add` and commit the changes as needed. - Note: In CI, sandboxing on Linux is not automatic (you might need to run inside Docker or use `run_in_container.sh` as recommended) [11] . Also, avoid extremely long-running tasks in CI; prefer targeted ones like updating docs, fixing formatting, etc.

Multi-Model Setup: Some power users configure multiple providers. For example, you might use OpenAI's GPT-4 for high-stakes tasks, but have a local LLM (via Ollama or OpenRouter with a model like Mistral) for quick iterations to save costs. Using profiles, you could do:

```
codex --profile gpt4 "Refactor the payment processing module for clarity"
# Later...
codex --profile local "Run tests and fix any failures"
```

In the above, the first command might use GPT-4 (more accurate but uses API credit), and the second uses a local model (faster/cheaper) to actually run tests and fix issues. *Note:* Local models support is via providers that mimic OpenAI's API. They might not be as capable, so results can vary.

Team Collaboration: Since Codex CLI works locally, each team member can use it with their own API key or login. It won't automatically share anything unless you commit changes. If you want to share an Codex CLI session's outcome, you could commit the changes it made or copy relevant parts of the conversation. (As of now, there isn't a cloud session sharing for the CLI, unlike ChatGPT which is cloud-based.)

Combining with Version Control: Embrace git when using Codex: - Always ensure you have a clean working state or a new branch before using Full Auto mode. This way you can easily review a final diff or revert if needed. - After Codex makes changes, run your test suite or build to verify. Codex tries to do this itself (especially in Full Auto), but you should double-check. - Commit changes incrementally. For example, after a successful refactor by Codex, commit those before moving to the next task. This isolates the AI's contributions for later inspection.

Dealing with Long Outputs: If Codex's answer scrolls past your terminal (e.g., a very long diff), remember you can scroll in your terminal or pipe output to less. In interactive mode, it pages through output but you might still need to scroll. There's also a possibility to ask Codex to summarize if output is too large (e.g., "summarize the diff").

Troubleshooting Tips: - If Codex seems to misunderstand your code, ensure your `AGENTS.md` is up to date with correct info. - If it times out or stalls, hit `Ctrl+C` and possibly simplify your request. - For any recurring issues or bugs, check the GitHub issues – many users report similar problems and the maintainers often provide workarounds or fixes in newer releases. - You can run Codex with `RUST_LOG` set to see if it's doing something in the background (like waiting on a process). This is advanced, but can reveal if, say, it's stuck running a test indefinitely.

Practical Examples and Workflows

To illustrate how Codex CLI can boost productivity, here are a few practical use cases with example commands and workflows:

- **Codebase Comprehension / Documentation:** If you join a new project, you can quickly get up to speed by asking Codex questions about the code. For example:
- *"Explain this repository to me."* – Codex will read through and produce an overview of the project structure and purpose (it might list major modules, their roles, and how they relate). This uses Suggest mode by default, so no changes are made – it's just reading and summarizing [73].
- *"Document the function* `processOrder` *in* `orders.js`*."* – Codex can insert a docstring or comment above the function, describing its parameters and behavior. In Suggest mode it would show the diff for the added comments, which you can approve to apply.

- *"Generate an ARCHITECTURE.md for this project."* – Codex can gather high-level understanding and propose a new markdown file with architecture notes. It will likely open an editor UI with the content for you to save.

- **Feature Implementation (with Tests):** Suppose you want to add a new feature:

- *Prompt:* "Add a new API endpoint `/users/{id}/activate` that sets a user's status to active. Include a unit test for this."
- **Workflow:** In Suggest mode, Codex might: create a new route in your controller, add logic to set the status, and suggest a test file or test case. It will show each file diff. You approve the changes. Then it might suggest running tests (it often knows to run `npm test` or similar from your `AGENTS.md`). Approve that – Codex runs your tests. If a test fails, Codex will notice and propose a fix (because it can read the test output). In Full Auto mode, it would do all of this in one go autonomously: code -> run tests -> adjust -> repeat until tests pass [74] [75].

- This iterative testing workflow is one of Codex's strengths (it was trained to iterate until tests pass). You essentially describe the goal, and Codex handles code + test until green.

- **Refactoring at Scale:** Want to rename a function or change a pattern across code:

- *Example:* "Refactor the Dashboard component to use React Hooks instead of class components."
- **Workflow:** Codex will find `Dashboard.jsx`, transform the class to a functional component with hooks (like `useState`, `useEffect`), update any references, and present the diff [76] [77]. If multiple files are affected, it will show diffs for each. In Auto Edit mode, it would apply them directly. Then it might suggest running the app or tests to verify nothing broke.

- Bulk changes like renaming a variable project-wide can also be done: "Rename all `.jpeg` files to `.jpg` and update references." Codex can use git moves for each file and adjust import paths accordingly [78]. It will list out each operation. This is where the sandbox is nice: it could execute the actual `git mv` commands for you in Full Auto.

- **Bug Fixing:** Provide the error or bug description:

- *Prompt:* "Fix the null pointer exception that occurs when saving a new Order with no customer."
- Codex will search the code for where an Order is saved, identify potential null references, propose a fix (like adding a null check or default object), and show the diff. It might also run the relevant test suite if configured. As a user, you just described the bug; Codex did the legwork of pinpointing and patching it. Always run the app/tests after and ensure the fix truly solves the issue.

- **Running External Tools via Agents:** If you configured an MCP server or have CLI tools, you can ask Codex to use them. For example, with Snyk integration (as per Snyk's guide):

- *Prompt:* "Scan this directory for security vulnerabilities and fix any issues found."
- Codex will invoke the Snyk MCP tool (if set up in config), get the results, and then possibly start fixing insecure code based on Snyk's report [79] [80]. It can loop: scan -> fix -> rescan until clean [81] [80]. This is an advanced workflow demonstrating Codex CLI's extensibility (and agentic behavior to use other tools).

- **Limitations & Considerations:** While Codex CLI is powerful, be aware of its limitations:

- *Cost:* If using the OpenAI API, each prompt/response (especially with large code context) costs tokens. Monitor usage. If you have Plus/Pro, leverage the login method to use your included quota (OpenAI gave Plus users some Codex credits – e.g., $5 for Plus, $50 for Pro [82] [83]).
- *Performance:* On very large repositories (hundreds of MBs of code), the initial scan or prompt might be slow, and it might not be able to load everything into context. In such cases, consider focusing it (e.g., "focus on the `src/` directory"). The model has a high token limit (Codex-1 can handle up to 192k tokens context in theory [84], which is huge), but practical use might slice the context.
- *Accuracy:* The AI might occasionally misunderstand code or make changes that don't actually solve the problem. Always review and test. It's a helper, not a fully autonomous programmer.
- *Windows quirks:* If on Windows via WSL2, you may run into issues with the browser login (since `localhost:1455` on WSL is not accessible to Windows browser by default). A workaround is to do `codex login` from a local machine and copy the `~/.codex/auth.json` to WSL, or use the suggested method of port forwarding `localhost:1455` to your local machine [85] [86]. Also, Full Auto sandbox might not function in WSL (the Linux container approach might need Docker anyway).

In day-to-day use, many developers find that using Codex CLI for repetitive coding tasks, writing boilerplate, or exploring unfamiliar code saves a lot of time. It's like having a junior developer or pair programmer who works at the speed of a machine. By providing it good guidance (through `AGENTS.md` and clear prompts) and keeping an eye on it via version control, you can integrate it as a reliable part of your development workflow.

Further Resources and Documentation

To dive deeper or get help, check out the following resources:

- **Official GitHub Repository (openai/codex):** This is the source code and primary documentation for Codex CLI [13]. The README contains extensive info (much of which is summarized above), plus links to the latest release downloads. You can also browse the code to understand how it works under the

hood. The issues and discussions on the repo are a good place to see known bugs, upcoming features, and usage tips from the community.

- **OpenAI Help Center – Codex CLI**: OpenAI's help articles on Codex CLI, such as *"Getting Started"* [87] and *"Codex CLI and Sign in with ChatGPT"*. These cover installation, authentication, and FAQs in a concise format. For example, the FAQ clarifies supported platforms, model availability, etc.
- **OpenAI Developer Community Forums:** There is an active section on Codex CLI where developers ask questions and share solutions. If you encounter an issue, it's worth searching there (or on Stack Overflow) – e.g., how to handle .gitignore (there were feature requests to have Codex respect gitignore [88]), or how to use Codex with Vim/Emacs (some have built editor integrations on top of the CLI).
- **OpenAI Blog – *Introducing Codex* (May 16, 2025):** This blog post [89] announced Codex (both the CLI and the cloud version) and provides background on the agent's capabilities, safety mechanisms, and vision. It's more high-level, but worth reading to understand the philosophy of the tool.
- **Community Blog Posts & Tutorials:**
- *"Describe It, Codex Builds It — Quick Start with Codex CLI"* by Rob Śliwa (Medium) – a detailed walkthrough of building a project from scratch with Codex CLI [90] [91].
- *Kevin Leary's blog post* (May 2025) on working with Codex CLI [92] [93] – shares some undocumented tips (like using `.env` files, and ignoring certain files).
- *Analytics Vidhya tutorial* on Codex CLI [94] [95] – covers basics and has screenshots of the process.
- *Phil Schmid's blog post – "OpenAI Codex CLI, how does it work?"* – explaining the internal workflow of the agent.
- *Hacker News discussions* – There was a HN thread when Codex CLI was released [96], which can be insightful to read (developers discuss pros/cons, comparisons to other tools).
- **Related Projects:** The ecosystem of "coding agents" is expanding. If you're interested, check out:
- **Cursor** (by OpenAI) – an editor with built-in agent (shares some underlying tech like MCP).
- **Cline** – an open-source CLI similar to Codex.
- **Amazon CodeWhisperer's agent mode** or **GitHub Copilot CLI** (if available) – analogous tools with different spins.
- **DotAgent** – utility for managing AI agent rules across projects (it generalizes things like AGENTS.md, Copilot instructions, etc., into one format) [48] [97].
- **OpenAI Codex CLI Changelog:** The CLI is evolving rapidly. Keep an eye on the release notes (in the GitHub releases or CHANGELOG.md [98]). For example, features like the ChatGPT login were added in mid-2025, so if you used Codex earlier you'll find the new versions quite different. The changelog will highlight new commands, bug fixes, and known issues.

Finally, remember that Codex CLI is an **experimental tool** (as noted by OpenAI) [99]. Expect some bumps, and don't hesitate to provide feedback or contribute to the project. Using AI in development is still new for many – this guide should equip you to get the most out of Codex CLI as your AI pair-programmer in the terminal. Happy coding!

Sources: The information above was gathered from the official OpenAI Codex CLI documentation and help center [1] [21], the OpenAI blog introduction of Codex [43], the Codex CLI GitHub repository README and config docs [44] [63], as well as community-written articles and guides for practical insights [91] [93]. Relevant excerpts from these sources are cited in-line in the format 【source†lines】 for reference. For full details, please refer to the linked sources and the official Codex CLI repository.

HTML Version of the Guide

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>OpenAI Codex CLI – Comprehensive Guide</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <!-- Link to external CSS for styling (assuming styles.css is available) -->
  <link rel="stylesheet" href="styles.css">
  <style>
    /* Basic minimal styling in case external CSS isn't present */
    body { font-family: sans-serif; line-height: 1.6; margin: 1em 2em;
background: #fafafa; color: #333; }
    h1, h2, h3, h4 { color: #2c3e50; }
    h1 { font-size: 2em; margin-top: 0.5em; }
    h2 { font-size: 1.5em; margin-top: 1.5em; border-bottom: 2px solid #e0e0e0;
padding-bottom: 0.3em; }
    h3 { font-size: 1.25em; margin-top: 1.2em; }
    pre, code { background: #e8e8e8; padding: 0.2em 0.4em; font-family:
monospace; }
    pre { padding: 0.8em; overflow-x: auto; }
    a { color: #1a0dab; text-decoration: none; }
    a:hover { text-decoration: underline; }
    ul { margin: 0.5em 0 0.5em 1.5em; }
    ol { margin: 0.5em 0 0.5em 1.5em; }
    nav { background: #34495e; padding: 0.5em 1em; border-radius: 4px; }
    nav ul { list-style: none; margin: 0; padding: 0; display: flex; flex-wrap:
wrap; }
    nav li { margin: 0.2em 1em 0.2em 0; }
    nav a { color: #ecf0f1; }
    nav a:hover { text-decoration: underline; }
    blockquote { border-left: 4px solid #ccc; padding: 0.5em 1em; color: #555;
background: #f9f9f9; margin: 0.8em 0; }
    table { border-collapse: collapse; }
    th, td { border: 1px solid #ccc; padding: 0.4em 0.6em; }
    th { background: #f0f0f0; }
    /* Ensure citations appear distinctly */
    .citation { color: #555; font-size: 0.9em; }
  </style>
</head>
<body>

<h1>OpenAI Codex CLI – Comprehensive Guide</h1>

<nav>
```

```
  <ul>
    <li><a href="#overview">Overview</a></li>
    <li><a href="#cli-commands">CLI Commands &amp;amp; Help</a></li>
    <li><a href="#approval-modes">Approval Modes</a></li>
    <li><a href="#agents-md">AGENTS.md vs AGENTS.local.md</a></li>
    <li><a href="#configuration">Configuration &amp;amp; Env Vars</a></li>
    <li><a href="#examples">Examples &amp;amp; Workflows</a></li>
    <li><a href="#resources">Further Resources</a></li>
    <li><a href="#top">^ Top</a></li>
  </ul>
</nav>

<a name="overview"></a>
<h2 id="overview">Overview</h2>
<p><strong>OpenAI Codex CLI</strong> is an open-source command-line tool that acts as a local AI coding assistant. It brings the power of OpenAI’s latest code-capable models directly to your terminal [1]. Codex CLI can read, modify, and even execute code on your machine to help you build features faster, fix bugs, and understand unfamiliar code – all through natural language commands. Because it runs locally, your source code never leaves your environment (unless you explicitly share it), addressing privacy concerns for proprietary code [2].</p>
<p>This CLI integrates into your workflow as a chat-like interface in the terminal. You prompt it in English, and it can perform tasks such as creating files, editing functions, running tests, and more, guided by your instructions. It essentially enables “pair programming” with an AI agent, without switching to a browser or editor plugin.</p>
<p><em>Key features include:</em></p>
<ul>
  <li><strong>Zero setup:</strong> Install with <code>npm install -g @openai/codex</code> (or <code>brew install codex</code>) and you’re ready to go [3]. Authenticate once (via API key or ChatGPT login) and the CLI requires no complex config to start working.</li>
  <li><strong>Terminal-native UI:</strong> The tool runs entirely in your terminal. It provides a text-based chat interface for conversation with the AI, so you don’t have to leave your command-line environment [4].</li>
  <li><strong>Multi-modal input:</strong> You can feed not only text but also images (like screenshots or sketched diagrams) to Codex CLI, and it will interpret them to inform its code generation [5]. For instance, you could paste a screenshot of an interface and ask Codex to generate HTML/CSS for it.</li>
  <li><strong>Flexible autonomy (Approval Modes):</strong> You control how much freedom the AI has to make changes. In Suggest mode it only suggests actions, in Auto-Edit it makes changes to files automatically, and in Full-Auto it can even run commands by itself in a safe sandbox [7] [8]. This allows usage ranging from cautious (review every suggestion) to hands-off (let it complete tasks on its own).</li>
  <li><strong>Local execution &amp;amp; sandboxing:</strong> The AI can execute shell commands and code locally to verify its changes. In Full Auto mode, these
```

```
commands run in a sandboxed environment (no network access, limited to your
project directory) [8] [10], ensuring that autonomous actions don't harm your
system or leak data.</li>
  <li><strong>OpenAI-powered, multi-model support:</strong> Codex CLI uses
OpenAI models (by default <code>o4-mini</code>, a Codex-optimized model). You
can specify any available model like GPT-4.1 [20], and as of mid-2025, ChatGPT
Plus users can even use the GPT-5 model through the CLI at no extra cost by
logging in [23]. Moreover, you can configure it to use other model providers
(OpenRouter, Azure, local LLMs via API) by editing the config.</li>
  <li><strong>Privacy and security:</strong> By default, none of your code files
are sent to the cloud. Only prompts and high-level context (and optional diffs
or summaries) are sent to the OpenAI API [12]. Full Auto commands are run in
isolation, and Codex CLI will warn you if you attempt to use autonomous modes
without version control in place [42], emphasizing safe usage.</li>
  <li><strong>Open source:</strong> The CLI is open-sourced on GitHub (<a
href="https://github.com/openai/codex" target="_blank" rel="noopener">openai/
codex</a> [13]). Developers can inspect the code, contribute improvements, or file
issues. OpenAI has also introduced a fund to encourage community-built tooling
around Codex CLI [14].</li>
</ul>
<p>In short, Codex CLI is like having a ChatGPT-powered junior developer
operating directly in your terminal. It can speed up routine coding tasks, help
with on-demand code explanations, and even handle whole refactoring or bug-
fixing sessions – under your supervision.</p>

<a name="cli-commands"></a>
<h2 id="cli-commands">CLI Commands and Help Reference</h2>
<p>Once installed, the primary way to use Codex CLI is by invoking the
<code>codex</code> command. The CLI supports a few subcommands and a variety of
flags to tailor its behavior. Here's a breakdown of the available commands and
options (as of the latest version):</p>
<ul>
  <li><code>codex</code> – Launches the interactive terminal UI (Text User
Interface). This enters a chat session where you and the AI can exchange
messages. If you run <code>codex</code> with no arguments, it will start in the
default Suggest mode and await your prompt [21]. Use this when you want a
conversational session to discuss or modify code.</li>
  <li><code>codex "Your prompt"</code> – You can provide an initial prompt in
quotes when launching the CLI [21]. For example: <code>codex "Explain the purpose
of utils.py"</code>. Codex will start, read your project, and directly answer
that prompt, then remain open for follow-ups. This saves a step, effectively
combining launch+question in one command.</li>
  <li><code>codex exec "task"</code> – Runs Codex in non-interactive mode to
execute a single task and then exit [21]. This is useful for automation or CI/CD
pipelines. Codex will carry out the instruction (applying edits or running
commands as needed) and print the results/diff. For instance: <code>codex exec
--auto-edit "format all Python files according to PEP8"</code> would attempt to
auto-format your codebase. The <code>exec</code> subcommand is essentially a
```

```
one-shot run. An example in CI might be updating a changelog as shown below:</p>
  <blockquote><code>codex exec --full-auto "update CHANGELOG for next release"</
code> <span class="citation">[100]</span></blockquote>
  <p>In that example, Codex would try to autonomously edit the
<code>CHANGELOG.md</code> based on commit history or context and then exit,
which you could then commit as part of your CI routine.</p>
  </li>
  <li><code>codex login</code> – Initiates the authentication flow to link your
ChatGPT account to Codex CLI [23]. This opens a browser window for OpenAI login
and, upon success, stores your credentials in <code>~/.codex/auth.json</code>.
After logging in, Codex CLI can use the models available to your ChatGPT plan
(like GPT-4 or GPT-5) without requiring API key usage. This is recommended for
Plus/Pro users, as OpenAI provides complimentary Codex usage credits for
subscribers (e.g., Plus users got $5 credit as of mid-2025) [82]. If you prefer
not to use the web login, you can still use an API key (see below). Note: You
only need to login once; subsequent uses will refresh tokens automatically. Use
<code>codex logout</code> if you need to disconnect.</li>
  <li><code>codex --help</code> – Shows the detailed help text, including usage
syntax and all available options. It's a good idea to run this at least once.
For example, due to a packaging quirk, it might show the usage with a binary
name (like <code>codex-x86_64-unknown-linux-musl</code>), but effectively that
refers to the <code>codex</code> command [101]. The help will list subcommands
like <code>exec</code>, <code>login</code>, etc., and global flags.</li>
  <li><code>codex --version</code> – Prints the version of the Codex CLI. Useful
for checking if you have the latest version or including when reporting issues
(e.g., <code>0.21.0</code>).</li>
  <li><code>codex --upgrade</code> – Self-upgrades the CLI by fetching the
latest version from npm [26]. This is equivalent to running the npm install
command again. Run this periodically to get new features and fixes.</li>
  <li><code>codex mcp</code> – (Advanced) Runs the CLI in Model Context Protocol
server mode [27]. This is not commonly used unless you are integrating Codex with
other tools via the MCP standard. It basically exposes Codex's functionality as
a service that an MCP client can interact with. Unless you know you need this,
you can ignore this command.</li>
</ul>
<p>Beyond these commands, a number of <strong>flags</strong> can modify
behavior:</p>
<ul>
  <li><code>--suggest</code>, <code>--auto-edit</code>, <code>--full-auto</code>
– Sets the approval mode for the session [28]. By default, <em>Suggest</em> mode
is used, so specifying <code>--suggest</code> is usually not necessary. Use
<code>--auto-edit</code> to start in Auto-Edit mode (Codex will apply file
changes automatically) or <code>--full-auto</code> for Full-Auto mode (Codex
will also execute commands autonomously in sandbox). See the next section for
details on modes.</li>
  <li><code>-m &amp;lt;model&amp;gt;</code> or <code>--model &amp;lt;model&amp;gt;</code> –
Choose a specific model for this run [102]. E.g. <code>-m gpt-4</code> to use
GPT-4, if available to you. Otherwise, the default model (o4-mini or what's
```

configured in your settings) will be used. This flag overrides the config file's model just for the current session.
  <code>-a</code> or <code>--ask-for-approval</code> – Force asking approval for every action [102]. Even in Auto-Edit or Full-Auto, this will make Codex pause for confirmation before applying a change or running a command. Essentially it's a safety toggle to ensure nothing happens without your "yes". If you start in Suggest mode, this flag has no additional effect (since it already asks by default).
  <code>-q</code> or <code>--quiet</code> – Quiet mode, which reduces the verbosity of output. In quiet mode, Codex might suppress intermediate reasoning or streaming text and just output final results. This can be helpful for scripting scenarios where you only want the final answer or diff. *Note:* There was an issue where quiet mode still expected an OpenAI API key even when using a different provider, so ensure you have <code>OPENAI_API_KEY</code> set if you use <code>-q</code> with third-party providers [29].
  <code>--config key=value</code> – Override a config option for this run [31]. For example, <code>codex --config maxTokens=5000</code> (hypothetically) to set a parameter not normally exposed. Most users won't need this, but it's there for advanced tuning. The <code>key</code> names correspond to those in the config file (without spaces, case-sensitive).

Interactive session "slash" commands: When you're inside the Codex TUI (after running <code>codex</code>), you can type special commands starting with <code>/</code> to control the session or get info. These include:

  <code>/init</code> – Create an <code>AGENTS.md</code> file in the current directory, pre-populated with headings and tips [32]. Use this when you start a project with Codex to quickly set up the context file for better results.
  <code>/status</code> – Show current status, like which model is in use, what mode you're in, how many tokens used so far, etc [32]. This can help track usage and settings in the middle of a session.
  <code>/diff</code> – Display the git diff of changes Codex has made or proposed [32]. Even uncommitted/untracked changes are shown. This is extremely useful to review changes incrementally. After a series of AI edits, <code>/diff</code> lets you scroll through the patch.
  <code>/prompts</code> – Show some example prompts and usage hints [32]. If you're not sure what to ask, this can provide inspiration or format examples (the CLI might list things like "/init - to initialize an agent file" etc. as we see in the welcome message).
  <code>/mode</code> – Change the approval mode on the fly [34]. For instance, typing <code>/mode full-auto</code> in the middle of a session will elevate Codex to Full Auto (it will usually confirm the switch in the output). Similarly, <code>/mode suggest</code> can downgrade to manual suggestions.
  <code>/help</code> – (If available) Lists the slash commands and maybe a short description of each. Not all versions have this, but it's common for such tools to include a help command in-session.
  <code>Ctrl+C</code> – Keyboard interrupt. This will cancel the current

```
generation or command. Use this if Codex seems stuck or if it’s taking too long
on a step and you want to regain control [35]. After pressing Ctrl+C, you can
usually type a new prompt to continue.</li>
</ul>
<p>These interactive commands are your control panel during a chat session with
Codex. They are especially important in longer sessions to monitor and manage
what the AI is doing.</p>
<p>Now that we have covered how to invoke and control Codex CLI, let’s explore
the different <a href="#approval-modes">Approval Modes</a> in detail, as they
determine how those commands above actually behave in practice.</p>

<a name="approval-modes"></a>
<h2 id="approval-modes">Approval Modes</h2>
<p>Codex CLI supports three distinct approval modes that govern the agent’s
permissions. Understanding these modes is critical for safe and effective use.
Here’s a closer look at each:</p>
<table>
  <thead>
    <tr>
      <th>Mode</th>
      <th>What the Agent Can Do</th>
      <th>When to Use</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td><strong>Suggest</strong> (default)</td>
      <td>Read files; propose code changes and terminal commands, but <em>always
asks you for approval</em> before actually editing a file or running a command
[7].</td>
      <td>Safe exploration, code reviews, initial understanding of a codebase.
Use this when you want to vet every action, or when trying out Codex for the
first time in a new project [37].</td>
    </tr>
    <tr>
      <td><strong>Auto Edit</strong></td>
      <td>Read and <em>write</em> to files automatically (applies edits without
confirmation). However, still prompts before executing any shell commands [38].</
td>
      <td>Refactoring or repetitive edits where confirming each small change is
tedious, but you still want to manually approve any command executions (which
could have broader effects) [39]. It’s a middle ground – file changes are
streamlined, but potentially dangerous operations are gated.</td>
    </tr>
    <tr>
      <td><strong>Full Auto</strong></td>
      <td>Read, write, and execute commands autonomously, within a sandboxed,
network-disabled environment [8]. No approvals are required once this mode is
```

```
active; the agent decides on its own.</td>
      <td>Long or complex tasks like debugging a failing test suite, fixing a broken build, or implementing a multi-step feature, especially when you trust the agent to iterate. It’s useful when you want to delegate a task entirely and possibly do something else (grab a coffee) [40]. Always review the outcome later.</td>
    </tr>
  </tbody>
</table>
<p>In **Suggest** mode, you are effectively in charge; Codex won’t change anything unless you explicitly okay it. This mode is very safe. For example, if you ask Codex to “optimize this function,” it will show a diff of the proposed optimization. You can then accept it (<code>y</code>) or reject (<code>n</code>). If it suggests running tests, you have to type “yes” before it actually runs them. This is great for learning and for cautious usage, ensuring you have full control.</p>
<p>In **Auto Edit** mode, Codex will directly apply changes to files as it suggests them. You’ll see the changes in the terminal output (usually prefaced by something like “✏ Applying patch to X file...”), but it won’t wait for you to confirm each one. However, if Codex wants to run a shell command (like building your project or executing tests), it will still ask you first. This mode significantly speeds up workflows like “make these 10 small changes across the codebase,” because you don’t have to hit “y” 10 times. But it still errs on the side of safety for commands, since running commands can have side effects beyond just modifying text in a file.</p>
<p>In **Full Auto** mode, Codex doesn’t ask for permission for anything – it will edit files, create new files, delete files, run compilation, run tests, etc. on its own, as needed to accomplish the goal. To prevent chaos, Full Auto always runs in a restricted environment:
<ul>
  <li>On **macOS**, it uses Apple’s sandbox (Seatbelt) via <code>sandbox-exec</code>, which restricts file system access to basically your project directory and a temp dir [10]. It also blocks all network access by default [103] (even if the code tries to call out).</li>
  <li>On **Linux**, Codex CLI (when Full Auto is enabled) can run inside a lightweight Docker container. The container has your current directory mounted (so it can read/write your project) and uses an <code>iptables</code> firewall to block egress (except the connection to OpenAI API) [11]. This container approach is optional (for now you have to run the provided <code>run_in_container.sh</code> to set it up).</li>
</ul>
So, the sandbox means even in Full Auto, the AI can’t accidentally modify files outside your project or exfiltrate data via network. It’s an important safety feature.</p>
<p>When launching Full Auto, Codex CLI will prompt a confirmation, especially if you’re not in a git repo: “Directory not under version control – are you sure?” [42]. This is because without version control, you have no easy way to see
```

```
what it changed or roll back. It’s highly recommended to only use Full Auto on projects tracked by Git (or similar). Commit your work, then let Codex Full Auto run, then you can use <code>git diff</code> or <code>/diff</code> to inspect changes.</p>
<p><strong>Tip:</strong> If you want Codex to behave mostly autonomously but you worry about a specific operation, you can start in Full Auto and then, if you see it heading in a risky direction, press <code>Ctrl+C</code> to stop and then switch to Suggest mode for the next steps. Alternatively, you can start in Suggest mode and after seeing it make good suggestions, type <code>/mode auto-edit</code> or <code>/mode full-auto</code> to speed things up.</p>
<p>In practice, many users start in Suggest for initial prompts (“What does this code do?”) then move to Auto Edit for mechanical changes (“Apply these fixes”). Full Auto is often used for letting the agent run tests and fix things—something that may involve multiple iterations. For example, in Full Auto you might say “Upgrade this project to Python 3.11 and ensure all tests pass.” Codex will:
<ol>
  <li>Edit some files (like update syntax, or dependencies).</li>
  <li>Run <code>pytest</code> (without asking).</li>
  <li>See failures, modify more files to fix, maybe run <code>pytest</code> again.</li>
  <li>Repeat until tests all succeed or it’s stuck.</li>
</ol>
All that would happen without your intervention. At the end, you get control back with a message like “All tasks completed.” Then you can review the changes.</p>
<p>No matter the mode, **you** have the final say. Even in Full Auto, you can always review the git diff or revert commits. The modes just determine how often the AI pauses to ask permission. Using them appropriately can make Codex CLI both safe and highly efficient.</p>
<p style="text-align:right;"><a href="#top">Back to top</a></p>

<a name="agents-md"></a>
<h2 id="agents-md">Using AGENTS.md and AGENTS.local.md</h2>
<p>Codex CLI can be guided by special markdown files named <code>AGENTS.md</code> (and optionally <code>AGENTS.local.md</code>). These files are not code; they are notes/instructions for the AI agent. They persist across sessions and help Codex understand your project’s context and conventions without you having to repeat information in every prompt.</p>
<h3>AGENTS.md – The Project Guidebook</h3>
<p><strong>What is AGENTS.md?</strong> It’s a plain text Markdown file (akin to a README) where you write down important information about your project for the AI to consider [43]. This can include:</p>
<ul>
  <li><em>Project overview:</em> e.g. “This is a web app for X, using framework Y, etc.”</li>
  <li><em>Technical stack:</em> languages, frameworks, versions (e.g. “Node.js 18 with Express, MongoDB backend”).</li>
  <li><em>Architecture and conventions:</em> how the project is structured
```

(“uses MVC pattern, services in <code>src/services</code>, UI in <code>src/components</code>”) and coding guidelines (“follow PEP8 style”, “use 2 spaces indent”).
  Key commands: how to run, test, build the project (“to run: <code>npm start</code>”, “to run tests: <code>npm test</code>”). Codex can use these when appropriate, especially in Auto or Full modes.
  Requirements or TODOs: high-level goals or pending tasks. You could list user stories or bug IDs you intend to work on, so Codex knows the context.

Codex CLI automatically searches for AGENTS.md in three places and merges them [44] :

  Global: <code>~/.codex/AGENTS.md</code> – your personal global instructions. This might include your general preferences or common directives you want for all projects (for example, “always write clear comments” or “favor functional programming style”).
  Repo root: <code>AGENTS.md</code> in the root of your project repository. This is the shared project-specific guidance – anyone using Codex on this repo should ideally have the same file (committed to version control). It’s like documentation specifically for AI assistance.
  Current folder: <code>AGENTS.md</code> in the current working directory (if different from repo root). This allows more granular or feature-specific notes. For instance, if you have a submodule or a specific directory for a microservice, you could put an AGENTS.md there with details just for that part. Codex will read that in addition to the repo root and global files [44] .

The content from all applicable files is merged, with the more local files taking precedence (i.e., deeper folder = higher priority) [44] . In practice, it likely concatenates them in that order (global, then repo, then local dir), so if there are conflicting statements, the later one wins. You can use this layering to your advantage: global could say “use American English in comments” and a repo-level could override “in this project, use British English” as a trivial example.
How Codex uses AGENTS.md: When Codex starts on a project, it will open and read these files internally. The information influences how it responds to your prompts. For example, if <code>AGENTS.md</code> says “All database changes must go through the repository layer, not directly in controllers,” Codex will try to adhere to that rule when generating code. Or if it lists how to run tests, Codex will know the exact command to use (saving it from guessing, which might be wrong). Essentially, it gives Codex additional context beyond just the code it sees.
Creating and editing AGENTS.md: If your project doesn’t have one yet, you can create it manually or use the <code>/init</code> command in Codex which will interactively help you start one [32] . That typically opens a text editor (nano or vim) with a template:
<pre><code># AGENTS.md

```
## Project
_(&lt;Describe the project&gt;)_

## Design
_(&lt;Architecture guidelines&gt;)_

## Commands
_(&lt;Important commands: build, test, run&gt;)_

## Conventions
_(&lt;Code style, naming conventions, etc.&gt;)_

## Env
_(&lt;Key environment variables&gt;)_

## TODO
_(&lt;Planned tasks or known issues&gt;)_
</code></pre>
You fill this out as needed. It’s a good idea to store this in git, so others on your team can also benefit (and so the AI instructions are transparent and versioned).</p>
<p>Notably, the OpenAI blog mentions: *“Codex can be guided by AGENTS.md files... akin to README.md... to inform Codex how to navigate your codebase, which commands to run for testing, and how best to adhere to your project's standard practices.”* [43] This captures the essence – it’s guidance for the agent, which ultimately helps you by making its output more relevant and accurate.</p>
<p>Whenever the project changes (new commands, new sections), update AGENTS.md. It doesn’t require a restart of Codex if you’re in the middle of a session – you can use <code>/init</code> to reload it, or restart Codex to pick up changes. If you have critical new info (like you just added a new dependency that requires a special build step), definitely put it in there so Codex knows.</p>
<h3>AGENTS.local.md – Personal/Local Instructions</h3>
<p><code>AGENTS.local.md</code> is not an official required file, but rather an emerging convention. The idea is to have a separate file for instructions that are <em>not meant to be shared or committed</em> – for your eyes (or AI) only. For example, if you have some experimental notes, or preferences that your team hasn’t agreed on, you might use AGENTS.local.md to keep them.</p>
<p>In some workflows, developers create an <code>AGENTS.local.md</code> and add it to <code>.gitignore</code>. In fact, the DotAgent project (a tool to manage AI instruction files) explicitly suggests ignoring <code>AGENTS.local.md</code> and treats any <code>*.local.md</code> file as private [48] [46] . DotAgent even updates .gitignore with a pattern to exclude <code>AGENTS.local.md</code> automatically [104] [47] .</p>
<p><strong>How to use AGENTS.local.md:</strong> Currently, Codex CLI does not automatically read an AGENTS.local.md file. It only looks for AGENTS.md as described. However, you have a couple of options to use it:
```

```
<ul>
  <li><em>Manual inclusion:</em> You could copy its contents into AGENTS.md when running Codex, or temporarily rename it. This is clunky but straightforward.</li>
  <li><em>Merge via script:</em> Write a small shell script or command alias that, before launching Codex, merges AGENTS.md + AGENTS.local.md into a temp file, and maybe points Codex to use that (for example by using <code>--config agentFile=...temp</code> if such an option existed, or simply by concatenating them into one AGENTS.md and removing after).</li>
  <li><em>Rely on global AGENTS.md:</em> If the content in AGENTS.local.md is truly personal and you want it always, consider putting it in your <code>~/.codex/AGENTS.md</code> instead. That global file is by definition your personal instruction set. You could delineate sections in it per project if needed (like “If project = X, do this…” though Codex may not parse that logic clearly).</li>
</ul>
The key is that AGENTS.local.md is for <strong>private rules</strong> – things like:
<ul>
  <li>Your own preferences (e.g., “I like code comments in a certain style”).</li>
  <li>Client-specific information or credentials that shouldn’t be in the repo. Perhaps the project is open source, but you have local API keys or experimental APIs you use – you might note those in the local file.</li>
  <li>Temporary notes – say you’re in the middle of a big refactor, you might list “Here’s the plan for refactoring module X” in local, since that plan might change or isn’t needed in official docs.</li>
</ul>
In short, AGENTS.local.md is a supplement for your own use. If you share the project with others, they won’t see those instructions (unless they also create their own local file).</p>
<p>Because it’s not automatically loaded, many users actually skip using a separate file and just incorporate everything into AGENTS.md and use comments or separators to mark private stuff. For instance, you could put an HTML comment in AGENTS.md like <code>&lt;!-- DevNote: ... --&gt;</code> that your teammates know to ignore. But the separate file approach has the benefit of clearly not committing it.</p>
<p><strong>Convention in other tools:</strong> This idea of “.local.md” files isn’t unique to Codex. For example, GitHub Copilot has <code>copilot.json</code> and they’ve discussed having <code>copilot.local.json</code> for personal settings, etc. DotAgent (as mentioned) formalizes this across multiple AI assistant tools [97]. So using AGENTS.local.md is future-proofing in a way. If Codex CLI later adds native support for it, it may automatically merge it or something.</p>
<p><strong>Bottom line:</strong> Use AGENTS.md as the main way to give Codex context. Only use AGENTS.local.md if you have a specific need to keep some AI instructions private. And remember, if you do use it, you’ll need to handle its loading/merging yourself (at least until official support potentially arrives).
```

```
Always keep any truly sensitive info (like actual passwords or keys) out of any AI context if possible, as a good security practice, even if running locally.</p>
<p style="text-align:right;"><a href="#top">Back to top</a></p>

<a name="configuration"></a>
<h2 id="configuration">Configuration, Environment Variables, and Usage Patterns</h2>
<p>Codex CLI is highly configurable to fit your development environment and preferences. We'll break this section into:</p>
<ul>
  <li>Setting up the <strong>configuration file</strong> and what options you can tweak.</li>
  <li>Important <strong>environment variables</strong> that influence Codex CLI.</li>
  <li>Typical <strong>usage patterns</strong> and best practices for integrating Codex into your workflow.</li>
</ul>
<h3>Configuration File (~/.codex/config.toml)</h3>
<p>After installing Codex CLI, it will create a directory <code>~/.codex/</code> in your home folder. Inside, you may find <code>config.toml</code> (or it might be generated on first run). This file contains default settings. You can edit it to change how Codex operates by default. The config format is TOML (key = value), but as noted earlier, JSON or YAML formats are also accepted in newer versions (you can use <code>config.yaml</code> if you prefer YAML syntax, for example) [51].</p>
<p>Key settings you can configure:</p>
<ul>
  <li><strong>model:</strong> Which model to use for completions. Example: <code>model = "gpt-4.1"</code> or <code>"o4-mini"</code> [20]. If you have access to GPT-4 (or beyond) and want it always, set it here. Otherwise, you might leave it as default and occasionally override via <code>-m</code> flag when needed.</li>
  <li><strong>model_provider:</strong> By default this is <code>"openai"</code>. But you can define others in the <code>[model_providers]</code> section and set this to use them. For example, to use OpenRouter, you would add:
    <pre><code>[model_providers.openrouter]
name = "OpenRouter"
base_url = "https://openrouter.ai/api/v1"
env_key = "OPENROUTER_API_KEY"</code></pre>
    and then set <code>model_provider = "openrouter"</code> in the main config [63]. Now Codex will send API calls to OpenRouter's endpoint instead of OpenAI's, allowing you to use models hosted there (like Anthropic's Claude, etc.), given you have an API key for it.</li>
  <li><strong>providers:</strong> *(If using JSON config as in older docs)* – It might be under a <code>"providers"</code> object if JSON. For TOML, it's <code>[model_providers]</code> as above. The key point is you can configure multiple and switch.</li>
```

reasoningEffort: Could be <code>"low"</code>, <code>"medium"</code>, <code>"high"</code> – this setting (used by OpenAI's Codex-1 model) indicates how much reasoning the model should apply. Higher usually means more thorough but possibly slower or costlier. If you find Codex's answers too superficial, consider setting a higher effort. This might not affect non-Codex models like GPT-4.
history.maxSize: How many tokens of conversation history to retain. If you set a high number, Codex will keep more of the past dialogue in context (useful for long sessions, but uses more token budget). history.saveHistory: if true, Codex may save your session transcripts somewhere (likely in ~/.codex/history). This could be handy to review later or if you want continuity between sessions, but note that saved history with sensitive info is something to manage carefully.
disable_response_storage: If you're under a Zero Data Retention policy and get errors about it [56], set this to true. This tells Codex not to rely on cached response IDs when calling the API, which is required under ZDR (where the API won't allow linking to past conversation turns) [105].
fullAutoErrorMode: This determines what Codex does if a command it ran in Full Auto fails (exits with error). The value could be <code>"ask-user"</code> (pause and ask you how to proceed) or <code>"ignore-and-continue"</code> (it will note the error and attempt to proceed anyway). Setting it to ask-user can prevent Codex from getting stuck in a loop on something like "server failed to start" – you get to intervene [57].
notify: If set to true, Codex might send desktop notifications for certain events (like when it's waiting for your approval, or after finishing a long task). This is a minor UX thing but can be useful if you switch windows while it's working [57].
profiles: The config supports grouping sets of configs under profiles. For example:
<pre><code>[profiles.o3]
model = "o3"
model_provider = "azure"
azure_version = "2023-05-15"</code></pre>
You might have multiple profiles for different scenarios (fast model vs accurate model, or different cloud providers) [58]. Then launch with <code>codex --profile o3</code> to use that profile [59]. Each profile inherits base settings but can override some. This feature is for power users who frequently swap contexts or models.
mcp_servers: If you plan to integrate external tools via the Model Context Protocol, you'll configure them here. As shown earlier, Snyk's security scanner can be added [61], or any other MCP-compatible service (perhaps you create one for custom analysis). Each entry names the server and the command to start it (plus args and env). Codex will then know it can call that tool. For example, after configuring Snyk's MCP, Codex can respond to a prompt "scan for vulnerabilities" by invoking that tool automatically [79].

<p>The official configuration documentation (in the repo's docs) lists all available options. Many of them you may never need to change. But it's good to

know you have this level of control.
One tip: If you want to quickly see what config options exist, run <code>codex --help</code> and look at the section that might list environment variables or config keys (some CLI help do list them). Alternatively, open the config file – the default one often includes comments or examples.
Remember that after editing the config file, you should restart the Codex CLI for changes to take effect (since it reads config at startup). If something seems off, double-check syntax. TOML is strict with things like quoting strings and not using tabs (should use spaces) etc.
<h3>Environment Variables</h3>
Various environment variables interplay with Codex CLI. Here are the key ones:

  OPENAI_API_KEY: Your OpenAI API key (if using API mode). If you didn't do <code>codex login</code> (which is the ChatGPT method), then Codex will expect an API key to be present [67]. Without it, it cannot call the OpenAI API and will refuse to work. Set this in your shell (e.g., in <code>~/.bashrc</code> or <code>~/.zshrc</code>) or in a local <code>.env</code> file. [106] [107] Codex CLI loads <code>.env</code> automatically thanks to <code>dotenv</code> [70], meaning you can just have a file with <code>OPENAI_API_KEY=sk-...</code> in your project and it will pick it up.
  OPENROUTER_API_KEY, GEMINI_API_KEY, etc.: Similarly, if you configure an alternate provider that needs a key, it likely has an env var. From our earlier example, OpenRouter requires <code>OPENROUTER_API_KEY</code> [63]. If using Azure OpenAI, you might need <code>AZURE_API_KEY</code> and some additional config for endpoint and deployment names (Azure uses a different authentication scheme and endpoints). Google's Gemini (via PaLM API) might use an API key or service account – if it's OpenAI-compatible via OpenRouter or similar, you'd still set an env key specified in config.
  GITHUB_TOKEN (optional): If you plan to use Codex CLI's PR helper functionality (it has some integration to open a GitHub PR with your changes, which might exist given it mentions a "built-in PR helper" requiring Git [108]), having a GitHub token could allow it to create gists or PRs via command. This isn't clearly documented, but just in case: certain OpenAI tools allow linking a GitHub account for PR creation.
  RUST_LOG: As mentioned, for debug logging. E.g. <code>export RUST_LOG=codex_core=debug</code> to see detailed logs in the console. By default, interactive mode logs to <code>~/.codex/log/codex-tui.log</code> with info level [71]. You can watch that file in a separate terminal with <code>tail -F</code> to monitor what the agent is doing behind the scenes (like reading a file, or the full conversation it's sending to the API, etc.). This is advanced usage for debugging or curiosity.
  NO_COLOR: If set (to any value), many CLI apps disable colored output. Codex might respect this too. If you find ANSI codes in your output and you don't want them, either use <code>--no-ansi</code> (if provided) or set NO_COLOR=1.
  http_proxy / https_proxy: If you are behind a corporate proxy, setting these environment variables ensures the CLI can reach the OpenAI

API. Codex CLI uses Node/Rust HTTP libraries which typically respect these standard env vars for proxies.
  CODEX_something: There may be some undocumented env variables for experimental features. For instance, some CLIs use flags via env for things like “enable beta features”. If you browse the code or release notes and see something like that, you’d set it similarly (e.g., CODEX_SUPER_MODE=true as a hypothetical example). This is not from docs, just a note that env vars sometimes hide toggles.

One more environment consideration: **Operating System Path/Dependencies**. Codex will run shell commands as needed. Ensure that any command it might call is available. For example, if you prompt “format code with clang-format,” but you don’t have clang-format installed, Codex can’t magically do it. So, having common build tools, linters, etc. installed is beneficial. If a needed tool is missing, Codex might even try to install it (it could propose running <code>apt-get</code> or <code>npm install</code>), which you’d have to approve. That’s fine, just be aware of what’s on your system.
.env files: It’s worth reiterating that Codex CLI loads a .env file if present in the working directory [70]. So you can keep API keys or other env vars there instead of cluttering your global shell. This .env will also apply to commands Codex runs (so if your app needs certain env vars set to run, putting them in .env means when Codex runs <code>npm start</code> or tests, those env vars are in place). This is hugely helpful for things like database URLs for testing, etc. Make sure not to commit your .env if it contains secrets (add to .gitignore).
<h3>Usage Patterns and Tips</h3>
Now that your Codex CLI is configured and you know how to control it, how do you best incorporate it into daily development? Here are some patterns:

  Start simple, then automate: When you begin using Codex on a project, you might stick to Suggest mode until you trust the agent. For example, ask it to describe the project or do a small change. As you see its behavior is reasonable, you can get bolder – maybe let it auto-apply trivial changes, or eventually try Full Auto on a contained task like running and fixing tests. This phased approach builds trust and understanding.
  One task at a time: Codex works best when given a clear, single objective. Instead of asking: “Build me a new feature with A, refactor module B, and fix bug C,” split those into separate sessions or prompts. Tackle them one by one. You can do multiple in one continuous session, but sequentially. E.g., “Now that feature A is done (and committed), let's refactor module B.” This avoids confusing the AI or stretching context too thin.
  Keep conversations focused: If you stray off-topic in a session (asking about an unrelated part of the code mid-session), Codex might carry a lot of context from earlier discussion that is no longer relevant. It can handle context well, but if you switch to a very different task, consider resetting (closing and reopening Codex) or using the <code>/clear</code> command if one exists (not sure if Codex CLI has a /clear context). This ensures it doesn’t get biased by previous prompts irrelevant to the new task.

Version control is your safety net: Always have git (or another VCS) initialized. Commit before starting a major Codex session. This way, you can compare “before vs after” easily. If Codex makes a mess, you can revert to before the session. If it does well, you can commit the changes with a clear message (perhaps “AI-assisted changes: did X and Y”). Each Codex session or task could be one commit. This makes auditing later easier.
Review diffs and test results: Don’t blindly accept everything. Codex is good but not perfect. Always read through the diffs it proposes (it might occasionally introduce a subtle bug even as it fixes another). Run your test suite after it makes changes, even if Codex itself ran them (double-check in a fresh environment if possible). Essentially, treat Codex’s output like you would a human junior developer’s code – useful, but needing code review.
Leverage approvals strategically: In Suggest mode, you might get a large diff and only want to accept part of it. Currently, Codex CLI’s approval is usually all-or-nothing for a given suggestion. But you can say “No” and then guide it: e.g., “That change to X looks good, but don’t change Y.” Codex will likely redo the diff focusing only on X. This interactive refinement is powerful. Use natural language to tell Codex how to tweak its suggestion, rather than manually editing the file yourself – let the AI do the mechanical part while you specify the intent.
Use Full Auto for grunt work: Some tasks are just painful to do manually but straightforward, e.g., renaming a function across dozens of files and updating references, or reformatting all code. Full Auto shines here – you can literally ask Codex CLI in full auto to “Enforce our coding style on the entire repository” and it might run your linter/formatter or make changes accordingly (provided it knows the style or you have a linter config). Another example: “Upgrade dependency X to latest version and fix any compatibility issues.” It can try to bump a version, run <code>npm install</code>, see if build/tests fail, modify code, etc. This could save hours. Just be sure to watch it (or review after) since automated upgrades can have side effects.
Combining with other tools: Codex CLI can act as an orchestrator of other CLI tools. If you have a Makefile or npm scripts, Codex will often use them (because it can read those files). That means you can simply instruct “Generate the API documentation” and if your project has a script for it, Codex might call it. Or with an MCP integration like Snyk, you say “ensure the code has no security issues,” it will run Snyk. The benefit is you interacting with one interface (Codex) and it delegates to many. This centralizes your workflow remarkably.
Stay aware of costs and rate limits: Each API call costs tokens, and there may be rate limits. Codex CLI streams output which is great for responsiveness. But if you have a very large codebase, the initial context it sends can be large. Monitor your usage on OpenAI’s dashboard if using API keys. If you hit limits, you may need to reduce how much context it sends (e.g., perhaps remove very large files or binary files from the directory to avoid Codex reading them – currently it doesn’t respect .gitignore unless you apply a patch/setting [88], though PRs exist to add that). One quick hack: temporarily rename node_modules or other huge folders while running Codex if it seems to

```
waste time reading those. The community is actively improving this aspect (like adding an <code>.agentignore</code> file to exclude certain paths) [88] .</li>
</ul>
<p>Finally, **remember the human override**: You as the developer have the final responsibility for the code. Codex CLI can handle a lot, but if something is critical (security-sensitive logic, complex algorithm), it's often best to double-check or even do it manually if the AI isn't instilling confidence. Use Codex to handle the boilerplate and mundane, and free your time to focus on the high-level design and tricky parts that require careful thought.</p>
<p>With practice, you'll learn which tasks Codex excels at and which to avoid. Over time, you might find you trust it with more (especially as the models improve). It's like mentoring an intern that quickly becomes a competent assistant – but still needs oversight for the hard stuff. Embrace its help, but stay in control.</p>
<p style="text-align:right;"><a href="#top">Back to top</a></p>

<a name="examples"></a>
<h2 id="examples">Practical Examples &amp;amp; Workflows</h2>
<p>Let's walk through a few concrete scenarios showing how you might use Codex CLI in real situations:</p>
<h3>1. Initial Codebase Exploration</h3>
<p><em>Scenario:</em> You just cloned a repository and want to understand it.</p>
<p><em>Actions:</em> Navigate into the repo and run <code>codex</code>. When prompted, ask something like "Summarize what this repository does." Codex will read through the files and likely generate an overview: e.g., "This is a Flask web application for a task tracker. It has modules A, B, C... etc." [73] You can follow up with questions like "What are the key classes or functions I should know about?" Codex might describe important classes and their relationships.</p>
<p>Use <code>/prompts</code> to see examples; one might be "Explain the code in file X", which you can do for specific files. Or <code>/diff</code> (though initially, no diff since nothing changed). Essentially, Codex acts like a documentation generator and tutor here.</p>
<p><em>Outcome:</em> In a few minutes, you have a decent mental model of the project, without reading all files manually. This helps you onboard faster. (Of course, verify its summary with actual code to ensure accuracy.)</p>
<h3>2. Bug Fix with Full Auto</h3>
<p><em>Scenario:</em> There's a failing unit test in your project. The test output says that for input X the function Y returns the wrong value, causing an assertion failure.</p>
<p><em>Actions:</em> You open Codex CLI. You might literally copy-paste the failing assertion or error message into Codex and say "Fix this bug." Codex will search where that error could come from. It finds function Y, sees what it's doing, and proposes a code change to fix the logic. It presents the diff to you in Suggest mode. If it looks reasonable, you accept it. You then type "Run tests" or Codex itself suggests running them. It runs <code>npm test</code> or <code>pytest</code> (depending on your project setup), sees the results. If all tests pass, it prints success. If not, it will show failures and possibly start
```

addressing the next failure.
You could alternatively do this in one go with Full Auto: <code>codex exec --full-auto "run the tests and fix any failures"</code>. Codex will loop running tests and fixing code until tests are green (or it gets stuck). It provides a log of what it's doing [74].
Outcome: The bug is fixed quickly. Codex not only applied a fix but also verified it via tests. You save time in diagnosing and iterating on the fix.
<h3>3. Implementing a New Feature</h3>
Scenario: You need to add a new feature (say, a new API endpoint in a web app) with certain requirements. You have a spec or general idea of what to do.
Actions: Write a summary of the feature in the prompt: e.g., "Add a new endpoint GET /users/{id}/settings that returns the settings for a user. Include authentication and a new unit test for this endpoint." Hit enter in Codex CLI. In Suggest mode, Codex will likely:

  Create or modify the route configuration to add the new endpoint.
  Implement the handler function (maybe in a controller or route module).
  Add a new test case in the test suite for this endpoint.

It will show you diffs for each file it changes or creates. You approve them if they meet the spec. It might then suggest, "Shall I run tests?" – you approve. It runs them. If something fails (maybe you forgot to update a URL in documentation), it may fix that too. If it passes, great.
During this, you can have a dialogue: "Actually, ensure that this endpoint is only accessible to admin users." Codex will adjust the code to add an auth check (because your AGENTS.md might have how auth is handled, it knows how to do that). You are effectively pair-programming: you state requirements, Codex writes code, you refine, etc.
Outcome: The new feature is implemented along with tests in, say, 10 minutes, whereas manually it might have taken 30. The code still might need some polishing (maybe style fixes), but you can either let Codex format it or do a quick pass yourself.
<h3>4. Large Scale Refactoring</h3>
Scenario: You want to rename a core class or function across the codebase, or change an API usage (e.g., migrate from one library to another equivalent).
Actions: Example prompt: "Rename the class <code>OldManager</code> to <code>NewManager</code> and update all references. Ensure everything compiles." Codex will find <code>OldManager</code> in your code, rename the class and constructor, update all places it's referenced. It might be dozens of changes. It shows the diffs possibly file by file. In Auto Edit mode, it would just apply them all. You double-check with <code>/diff</code> (which will be big). Then you tell Codex "run the build" or "run tests" to be sure. If it missed something, the compiler/test will catch it, and Codex will fix it.
This is similar to what a good IDE refactor tool can do, but here you didn't

```
have to click through anything – just one command. Another example: “We’re replacing usage of <code>LibraryX</code> with <code>LibraryY</code>. Update the import statements and equivalent function calls.” Codex might need some guidance if APIs differ, but you can do it iteratively. Perhaps first ask it to change the import, then handle errors as they arise.</p>
<p><em>Outcome:</em> A potentially error-prone, multi-file refactor is done quickly and with less mental load. And you had the safety net of tests/compilation to verify the refactor didn’t break things.</p>
<h3>5. Documentation and Cleanup</h3>
<p><em>Scenario:</em> You want to improve documentation or code comments across your project.</p>
<p><em>Actions:</em> Using Codex, you could do:
<ul>
  <li>“Generate docstrings for all public functions in <code>utils.py</code>.” – Codex will parse that file and add docstrings in a consistent style to each function that lacks one. It shows the diff; you approve.</li>
  <li>“Create a Markdown API documentation file listing all endpoints and their purpose.” – If it’s a web API, Codex can gather info (from route definitions, perhaps reading your controllers) and produce an <code>API.md</code> file. You review and tweak if necessary. It might not be perfect, but it’s a great start.</li>
  <li>“Find and remove any commented-out code.” – Codex can scan files for blocks of code that are commented and remove them (or better, present a diff removing them for you to confirm). This is a nice cleanup task near release time.</li>
</ul>
In doing these, use Suggest mode first to ensure it’s doing what you intend (documentation quality can vary, you might need to instruct it on format). Once confident, you could let it auto-apply for many files.</p>
<p><em>Outcome:</em> Improved documentation and cleaner code, done in a fraction of the time it would take manually combing through files. And because Codex can understand the code, the docstrings it writes are often accurate (though always give them a read to ensure no hallucinations).</p>
<p>These examples show the versatility of Codex CLI – from understanding code to modifying it in non-trivial ways. The common theme is that you, the developer, provide guidance and validation, while Codex handles the brute-force editing and some of the thinking. You save time and reduce manual mistakes (like missing one of ten references to a function during rename, which Codex would not miss unless it lacked context).</p>
<p>One thing to note: the effectiveness of Codex CLI can depend on how well your project is structured and how good your instructions are. Clear, concise prompts yield better results. Also, if your codebase has a lot of context (like relevant info in README or comments), Codex picks up on that. It’s worth investing in your AGENTS.md and keeping the project’s structure clean – not just for Codex, but it helps any contributor (AI or human).</p>
<p style="text-align:right;"><a href="#top">Back to top</a></p>

<a name="resources"></a>
```

<h2 id="resources">Further Reading & Resources</h2>
For more information, help, or involvement with OpenAI Codex CLI, check out these resources:

  OpenAI Codex CLI GitHub Repository – The official repo contains the source code, README documentation, a <code>docs/</code> folder with more detailed docs (like <code>config.md</code>), and a CHANGELOG. It’s the best place to see the latest updates and known issues. If you encounter a bug or have a feature request, you can open an issue there [109]. Also, browsing the <code>Discussions</code> tab can provide insight into how others are using the tool or clever tricks.
  OpenAI Help Center – Codex CLI Getting Started – An official quickstart article [87] with some FAQ. It covers the basic installation and a short FAQ (which models are used, does it upload code, how to update, etc.). It’s brief but useful for common questions when first installing.
  OpenAI Developer Community (Codex section) – A forum where you can ask questions and share experiences. If you run into a problem, it’s likely someone else has too. For example, issues with Windows login or .gitignore not being respected have been discussed there [110] [88]. The OpenAI staff and community members often provide solutions or workarounds.
  OpenAI Blog: Introducing Codex – This blog post [111] announces Codex (the broader initiative) and specifically mentions the CLI and how it works in tandem with the ChatGPT sidebar version. It gives a perspective on why Codex was built and some design philosophy (like the importance of citations and sandboxing to build trust in AI agents). It’s a good read for understanding the vision behind the tool and its future direction.
  Tutorials & Blog Posts:

      "Describe It, Codex Builds It — Quick Start with Codex CLI" by Rob Śliwa (Medium) – A detailed walkthrough of using Codex CLI to build a feature from scratch [90]. Rob shares his experience and even an example AGENTS.md content for a project. Great for seeing an end-to-end usage in practice, with commentary.
      "Working with OpenAI’s Codex CLI" by Kevin Leary – Focuses on setup, configuration, and some quirks (like using .env, global context, ignoring files) [93] [88]. It’s useful especially for the config and environment preparation aspects.
      "OpenAI Codex CLI: Build Faster Code Right From Your Terminal" (Blott.studio) – An introductory article summarizing features and installation with some imagery [112] [17]. Good for sharing with colleagues who need a high-level overview before diving in.
      "Understanding OpenAI Codex CLI Commands" (machinelearningmastery.com) – As the name suggests, a tutorial focusing on usage of commands and perhaps some internal logic. This could help reinforce

```
your knowledge of the CLI capabilities.</li>
    </ul>
  </li>
  <li><strong>Related Tools:</strong> Codex CLI is one of several AI coding
assistants:
    <ul>
      <li><a href="https://platform.openai.com/docs/codex" target="_blank"
rel="noopener">OpenAI Codex (API)</a> – Not to be confused, but the earlier
Codex model (2021) and its API documentation might still be floating around.
Note that the old Codex model is deprecated [113]; the CLI uses newer models. The
concept is similar but the CLI adds the agentic behavior.</li>
      <li>GitHub Copilot CLI – A small tool from GitHub for shell autocompletion
and small tasks. Not nearly as comprehensive as Codex CLI, but interesting to
compare if you've used it.</li>
      <li><a href="https://github.com/johnlindquist/dotagent" target="_blank"
rel="noopener">DotAgent</a> – The tool we mentioned for managing AI agent
instructions. If you find yourself using not just Codex but other AI tools,
DotAgent helps keep your AGENTS.md, etc., in sync or convert between formats [97].
For example, if you wanted to also support Cursor or Copilot, you could maintain
one set of rules and export to each format.</li>
      <li>Cursor (by OpenAI) – An IDE (VSCode-like) that also uses the Codex
model and supports MCP. If you prefer a GUI editor with similar AI capabilities,
give Cursor a try. Codex CLI and Cursor can actually work together via MCP,
theoretically.</li>
      <li>Local LLM dev helpers: There are emerging open-source projects to
replicate Codex CLI with local models (for those who want completely offline
solutions). One example is "aider" or "GPT-Engineer". They're not as polished as
Codex CLI yet, but you might watch that space if interested in offline usage.</
li>
    </ul>
  </li>
</ul>
<p>By exploring these resources, you can deepen your understanding, troubleshoot
issues, and even contribute to the improvement of Codex CLI. Since the tool is
evolving, staying updated via the GitHub repo (watching releases or discussions)
is useful. Who knows – you might discover new features as they come out (like
when the ChatGPT login was introduced, which was a major change). The community
around these AI coding tools is growing, and sharing experiences helps everyone
learn how to best use them.</p>
<p>Happy coding with Codex CLI!</p>

<!-- Citations (for reference in rendered view, though in a static HTML page
these might just appear as text) -->
<div class="citation">
<p>Sources cited in this guide are indicated in the text with the format
【source†lines】. For example, [1] refers to lines 15-23 of source [1]. Below is
a list of sources:</p>
```

```
<ol>
  <li id="cite-1"><a href="https://help.openai.com/en/articles/11096431-openai-codex-cli-getting-started" target="_blank" rel="noopener">OpenAI Help Center: "OpenAI Codex CLI – Getting Started"</a></li>
  <li id="cite-19"><a href="https://github.com/openai/codex#cli-reference" target="_blank" rel="noopener">OpenAI Codex CLI GitHub README – CLI reference section</a></li>
  <li id="cite-7"><a href="https://openai.com/index/introducing-codex" target="_blank" rel="noopener">OpenAI Blog: "Introducing Codex"</a>, May 16, 2025</li>
  <li id="cite-4"><a href="https://github.com/openai/codex/blob/main/README.md" target="_blank" rel="noopener">OpenAI Codex CLI GitHub README – various sections (Memory, MCP, etc.)</a></li>
  <li id="cite-28"><a href="https://github.com/openai/codex/blob/main/README.md#codex-also-allows-you-to-use-other-providers" target="_blank" rel="noopener">OpenAI Codex CLI GitHub README – Other Providers and Profiles section</a></li>
  <li id="cite-14"><a href="https://medium.com/@robjsliwa_71070/describe-it-codex-builds-it-quick-start-with-codex-cli-8493956b9480" target="_blank" rel="noopener">Rob Śliwa on Medium: "Describe It, Codex Builds It — Quick Start with Codex CLI"</a>, Aug 2025</li>
  <li id="cite-22"><a href="https://www.kevinleary.net/blog/openai-codex-cli/" target="_blank" rel="noopener">KevinLeary.net Blog: "Working with OpenAI's Codex CLI"</a>, May 7, 2025</li>
  <li id="cite-33"><a href="https://www.blott.studio/blog/post/openai-codex-cli-build-faster-code-right-from-your-terminal" target="_blank" rel="noopener">Blott Studio: "OpenAI Codex CLI: Build Faster Code Right From Your Terminal"</a>, Apr 17, 2025</li>
  <li id="cite-31"><a href="https://www.analyticsvidhya.com/blog/2025/05/codex-cli/" target="_blank" rel="noopener">Analytics Vidhya: "How to Install and Use OpenAI Codex CLI Locally?"</a>, May 2025</li>
  <li id="cite-17"><a href="https://docs.snyk.io/integrations/developer-guardrails-for-agentic-workflows/quickstart-guides-for-mcp/codex-cli-guide" target="_blank" rel="noopener">Snyk User Docs: "Codex CLI guide"</a>, 2023</li>
  <li id="cite-27"><a href="https://github.com/openai/codex/issues" target="_blank" rel="noopener">OpenAI Codex CLI GitHub Issues – (search results for OPENROUTER_API_KEY and quiet mode)</a></li>
  <li id="cite-29"><a href="https://github.com/openai/codex/issues/2231" target="_blank" rel="noopener">GitHub Issue #2231: "Help output shows incorrect binary name in Usage section"</a>, Aug 12, 2025</li>
  <!-- Additional sources can be listed similarly -->
</ol>
</div>

</body>
</html>
```

1 2 3 5 6 7 8 12 13 15 19 26 28 34 35 36 37 38 39 40 42 67 73 87 109 OpenAI Codex CLI – Getting Started | OpenAI Help Center
https://help.openai.com/en/articles/11096431-openai-codex-cli-getting-started

4 16 74 75 76 77 78 106 107 OpenAI Codex CLI - Command Line AI Coding Assistant
https://agentsmd.net/codex-cli/

9 10 11 82 83 94 95 99 103 How to Install and Use OpenAI Codex CLI Locally?
https://www.analyticsvidhya.com/blog/2025/05/codex-cli/

14 17 18 51 57 70 108 112 OpenAI Codex CLI: Build Faster Code Right From Your Terminal | Blott Studio
https://www.blott.studio/blog/post/openai-codex-cli-build-faster-code-right-from-your-terminal

20 21 22 23 24 25 27 31 41 44 49 54 55 56 58 59 60 63 64 65 71 72 85 86 98 100 102 105 113 GitHub - openai/codex: Lightweight coding agent that runs in your terminal
https://github.com/openai/codex

29 30 Bug: Quiet mode requires OPENAI_API_KEY even when using other ...
https://github.com/openai/codex/issues/621

32 33 90 91 Describe It, Codex Builds It — Quick Start with Codex CLI | by Rob Śliwa | Aug, 2025 | Medium
https://medium.com/@robjsliwa_71070/describe-it-codex-builds-it-quick-start-with-codex-cli-8493956b9480

43 45 84 89 111 Introducing Codex | OpenAI
https://openai.com/index/introducing-codex/

46 47 48 97 104 GitHub - johnlindquist/dotagent: Universal AI agent configuration parser and converter
https://github.com/johnlindquist/dotagent

50 52 53 68 69 88 92 93 Kevinleary.net: Working with OpenAI's Codex CLI
https://www.kevinleary.net/blog/openai-codex-cli/

61 62 79 80 81 Codex CLI guide | Snyk User Docs
https://docs.snyk.io/integrations/developer-guardrails-for-agentic-workflows/quickstart-guides-for-mcp/codex-cli-guide

66 Run Codex
https://docs.codex.storage/ko/learn/run

96 OpenAI Codex CLI: Lightweight coding agent that runs in your terminal
https://news.ycombinator.com/item?id=43708025

101 Help output shows incorrect binary name in Usage section · Issue #2231 · openai/codex · GitHub
https://github.com/openai/codex/issues/2231

110 We (Codex) shipped a pretty large CLI update today and have many ...
https://news.ycombinator.com/item?id=44833858