ChatGPT

Mastering ChatGPT Plus + Codex + VSCode Extension for Agentic Python Development on Windows 10

Executive Summary

ChatGPT Plus, combined with OpenAI's Codex VS Code extension, enables a powerful *"agentic"* coding assistant directly in your IDE. This setup allows developers on Windows 10 to leverage GPT-based coding features (including GPT-4/5 models) *without* a separate OpenAI API subscription. The Codex agent can read and modify your local code, execute commands (like running tests), and even delegate larger tasks to cloud environments – all through the VS Code interface using your ChatGPT Plus account [1] [2]. In this guide, we provide a deep, engineering-focused analysis of what is possible using **only ChatGPT Plus and the official OpenAI VSCode Codex extension** under Windows 10 (with a Conda Python environment). We'll detail how to maximize Codex's capabilities within these constraints, and when additional tools (CLI utilities, API keys, or Model Context Protocol integrations) might become necessary.

Key points summarized:

- **ChatGPT Plus Integration:** The VSCode Codex extension works out-of-the-box with a ChatGPT Plus login (Plus, Pro, Business, etc.), meaning you get substantial coding assistance without separate API keys [1]. This includes GPT-5 Codex (an optimized model for coding) as well as GPT-4/5 general models for different tasks [3].
- **Agentic Capabilities:** In "Agent" mode, Codex can autonomously read multiple files, perform multi-file edits/refactors, and execute shell commands within your project directory [4]. It can follow a *plan→approve→execute* workflow: for complex tasks you can have Codex draft a plan, then iteratively approve steps as it implements and tests changes.
- **Conda on Windows:** We address how Codex interacts with Conda-managed Python on Windows. By default, Windows support is *experimental* [5], but we provide best practices to ensure Codex uses your Conda environments (e.g. activating envs in the terminal or using `conda run` commands). This avoids needing WSL or system Python, aligning with a Conda-only setup.
- **Project-Wide Context:** Codex excels when given structured context. We show how to feed it design docs, GitHub issues, TODO comments, and architectural notes to drive development. We also explain the **AGENTS.md** convention – a special project instruction file that Codex will always read for context [6] [7]. By using such artifacts, you can guide the AI with specifications and coding standards unique to your project.
- **Limits and Extensibility:** We outline what tasks are fully achievable with the Plus + VSCode setup (e.g. local code edits, test runs, AI-generated commit messages) and which tasks might require augmenting your toolkit. Advanced automation – like continuous integration auto-fixes or extensive GitHub integration – may benefit from the OpenAI Codex CLI and/or MCP (Model Context Protocol) servers. These allow scripting Codex and connecting it to external services (documentation, browsers, GitHub APIs, etc.) when needed. We clarify when an API key or self-hosted tools become

necessary (for example, headless CI usage or exceeding Plus plan limits), versus when the built-in Plus allowance is sufficient.

In summary, with ChatGPT Plus and the Codex VSCode extension on Windows 10, you can achieve a rich AI-assisted development workflow: the AI can *plan, write, modify, and test code* in dialogue with you [4], grounded in your local project context. This guide will walk through detailed capabilities, setup steps, example workflows, and best practices to fully harness this setup in a Conda-based Python development environment.

Capabilities Matrix: ChatGPT Plus vs API vs CLI vs MCP

The following matrix outlines which features are available with a basic ChatGPT Plus + Codex extension setup, and which require additional tools or keys. This will clarify what you can do **without** an OpenAI API subscription, and what extra capabilities become possible with the CLI or MCP integrations.

Feature / Capability	ChatGPT Plus + VSCode	OpenAI API Key	OpenAI Codex CLI	MCP Integration
IDE chat & code completions	**Yes** – included with Plus (GPT-4/5 Codex models) [8] [1]. No extra key needed.	**Yes** – via API calls (e.g. to GPT-4) if not using extension.	N/A (CLI can use either Plus auth or API key).	N/A
Read local files and repository	**Yes** – Agent can open/read files in workspace [4]. Context mainly from opened or referenced files [2].	**Yes** – but manual; API can retrieve files if you program it.	**Yes** – CLI can load files via `open` or context commands [2].	**Yes** – certain MCPs (e.g. a code search MCP) could enhance this, but not required for local files.
Multi-file editing & refactoring	**Yes** – Agent mode auto-edits multiple files (e.g. apply refactor across project) [9].	**Partial** – possible via your own orchestrated API calls or tools like Copilot CLI.	**Yes** – CLI can similarly apply multi-file changes via agent.	**No** – MCP not needed for editing local files (the core agent handles it).
Executing code/tests locally	**Yes** – Agent can run shell and Python commands in project dir (with approval as needed) [4].	**No** – raw API won't run code on your machine without additional tool.	**Yes** – CLI can run commands locally through its agent.	**No** – (MCP not required; the agent's built-in shell tool handles this).

Feature / Capability	ChatGPT Plus + VSCode	OpenAI API Key	OpenAI Codex CLI	MCP Integration
Autonomous plan & tool use	**Yes** – "Agent" mode can autonomously propose a plan and execute tools (within limits) [10] [11]. Plus account covers this.	**No** – not via API alone (you'd have to code an agent loop).	**Yes** – CLI has same autonomous mode and even a TUI for approvals.	**No** – (MCP extends toolset but planning is model-internal).
Context from entire codebase	**Partial** – Codex uses opened/ referenced files (not *all* files by default). Large codebases need targeted prompts or cloud delegation.	**Yes** – you could feed more files via API (token limits permitting).	**Partial** – similar to extension, must open or reference files for context.	**Yes** – e.g. a documentation MCP or custom tool can provide additional context (like searching the codebase).
Respects .gitignore / excludes	**Limited** – By default, Codex may consider all files in workspace; no built-in `.codexignore` support yet [12]. You must avoid opening sensitive files or use AGENTS.md to instruct ignoring.	N/A	**Limited** – same limitation in CLI (no ignore by default) [13].	**N/A** – (MCP isn't about ignoring local files).
Internet access (web browsing)	**Yes, with approval** – Codex can use internet if explicitly allowed (Full Access mode) [14]. By default Plus/Agent blocks network unless permitted.	**Yes** – via API if you use OpenAI's browsing tools or plugins (requires paying API or a Plus plugin).	**Yes** – CLI agent can be configured to allow internet or connect MCP for web.	**Yes** – e.g. a "browser" MCP tool or enabling `internet_access` setting [15] [16].

Feature / Capability	ChatGPT Plus + VSCode	OpenAI API Key	OpenAI Codex CLI	MCP Integration
Cloud execution environment	**Yes** – Included with Plus: you can delegate tasks to Codex Cloud (hosted sandbox) for heavy jobs [17]. No API key needed (uses your Plus credits).	**N/A** (distinct from API usage).	**N/A** – CLI triggers cloud via your account similarly if configured.	**N/A** – (Cloud is an OpenAI service, not an MCP third-party tool).
Using GitHub issues/PRs as input	**Yes (manual)** – e.g. copy-paste issue text or open as file for context. No direct integration without token.	**N/A** (not applicable without coding it yourself).	**Yes (manual)** – you can paste or use CLI with a token to fetch.	**Yes (automated)** – with a **GitHub MCP server** and a token, Codex can fetch and even act on issues/PRs [18].
Automating PRs, commits, CI fixes	**Partial** – Codex can draft commit messages or code changes, but doesn't push to git by itself. You still commit manually (or instruct it to output a patch).	**No** – not without building an automation around the API.	**Partial** – CLI can output patches; GitHub Action integration exists [19] but requires setup and possibly API.	**Yes** – GitHub MCP can let Codex create PRs or comments programmatically [18].
Model customization (fine-tuning)	**No** – Not with ChatGPT Plus alone (not applicable to Codex usage).	**Yes** – if using API, you could fine-tune base models (though not GPT-4/5).	**No** – CLI uses the same models; no fine-tuning interface.	**No** – MCP is about tools, not model weights.
Usage Limits	**Yes** – Plus has generous rate limits (e.g. ~30-150 requests / 5 hours) [20] [21], but heavy use could hit them.	**Yes** – pay-per-use with your own limits (you control via API rate & cost).	**Yes** – CLI obeys whichever auth you use (Plus limits or your API quotas).	**N/A** – MCP calls might have their own limits (e.g. API rate for a doc service) but no OpenAI cost impact.

Notes: *ChatGPT Plus with the VSCode Codex extension covers the vast majority of capabilities needed for interactive development (file editing, running code, using context, etc.) without any extra API costs [1]. The OpenAI Codex CLI is essentially the "engine" under the hood – the VSCode extension is built on it [22] – and can be*

used for non-IDE scenarios (like scripting or CI). Model Context Protocol (MCP) integrations are optional add-ons to grant the AI access to external knowledge or actions (for example, fetching library docs or controlling a browser) [23] [24] *. They don't require separate OpenAI payments, but some MCP servers might need third-party tokens or tools (e.g. a GitHub PAT or Docker). In the sections below, we explore each capability in depth, especially as it relates to a Conda-based Python workflow on Windows 10.*

Feature-by-Feature Analysis: Codex in VSCode (No API Key Needed)

In this section, we break down the core features of the Codex VSCode extension running with a ChatGPT Plus account, and explain how each works in our Windows 10 + Conda environment context:

1. IDE Integration and Context Awareness

The Codex extension lives inside VS Code, providing a chat sidebar where you converse with the AI agent. Because it's in your IDE, it has direct awareness of your workspace. **Opened files and selected code are automatically provided as context** to Codex, which means you can ask questions or give instructions without copy-pasting large chunks of code [2]. For example, if you highlight a function and ask "Can you refactor this?," the agent already "knows" the selected code. You can also explicitly reference files by name using the `@filename` syntax to pull them into context [2]. This tight integration allows Codex to understand your project's state more effectively than the standalone ChatGPT web UI.

However, Codex **does not automatically read the entire repository** by default (which would be impossible for large projects given token limits). It focuses on: - *Active context:* the files you have open or pieces you specifically show it. - *On-demand file access:* If you mention a file (using `@file.py`), it will retrieve that file's content. Similarly, if a stack trace or error mentions a file/line, it can open those. - *Project documentation:* Codex will look for special context files like `AGENTS.md` (explained later) or possibly README files in a project. These serve as persistent context or instructions it should always consider [6] [7].

Ignoring files: Currently, there is no built-in mechanism to exclude certain files or directories from Codex's purview (like a `.codexignore`). In fact, users have requested such a feature [12]. Codex doesn't strictly honor `.gitignore` either – if you open or reference an ignored file, it will see it. So, be mindful not to accidentally expose secrets or irrelevant large files to the AI. A best practice is to put any sensitive credentials in env variables (not in files), and consider using `AGENTS.md` to warn Codex about directories or files it should ignore (though this isn't foolproof, it gives a hint to the model).

Context size and limitations: The Plus plan's Codex models (GPT-4/GPT-5 variants) support fairly large contexts (on the order of thousands of tokens, with GPT-5 likely supporting even more). But they are not unlimited. If you ask Codex to consider too many files at once (say, "analyze these 20 modules entirely"), it may hit context limits or give a shallow analysis. In practice, it's better to **guide the agent with specific parts of the code** relevant to the task. The official guidance suggests breaking big problems into smaller ones [25] – treat Codex like a teammate who can only keep a few files in mind at a time. For truly large-scale understanding (e.g. whole-codebase refactoring), you might use the *cloud delegation* feature (discussed later) which can handle more data by offloading to an OpenAI server built for that.

2. Reading and Editing Multiple Files

One of Codex's strongest capabilities in VSCode is automated code editing across files. In **Agent mode**, it can open, create, and modify files in your project without you manually doing so [4] [26]. For example, if you say "Add a new module `utils` with a function `foo()` and use it in `main.py`," Codex can generate the `utils.py` file and edit `main.py` accordingly, all in one go. It shows you a preview of these changes in the chat, and often you'll see a diff or the new file content before you accept it.

How it works: - **File reads:** Codex may proactively read files it thinks it needs. You'll notice in its responses things like "Reading `models.py`... Done." This happens when the task likely involves those files. It sticks to files under your project directory (it won't read outside without permission) [4]. - **Proposed edits:** The extension will present changes for your approval. It might say "I will update `models.py` to add the new function." In default Agent mode, these intraproject edits are applied automatically unless you cancel. (For external or risky changes, you get an explicit prompt to approve – see *Approval Workflow* below.) - **Refactoring support:** You can ask for large-scale refactors, and Codex will handle updating all occurrences. For instance, "Rename class `OldName` to `NewName` across the project" could lead Codex to open every file where `OldName` appears and change it. It's quite effective at this kind of mechanical change. For more semantic refactors (like abstracting a function), it generally tries a plan: e.g. create new file, move code, adjust imports. It's advisable to review the diffs it provides to ensure nothing is missed.

Limitations: If your project has >100 files or very complex interdependencies, the agent might not catch everything in one shot – it's still constrained by what it can load into memory. For huge changes, it may be better to tackle them in steps (possibly by module or feature). Additionally, because Windows support is *experimental* [5], there have been occasional reports of the extension becoming unresponsive on very large repos (especially if it tries to index lots of files). If you notice slowness, consider closing some folders in VSCode, or using the "cloud" mode for heavy lifting to reduce strain on the local client. The docs note that working in WSL is optimal for performance on large codebases [27], but since we're avoiding WSL, just be mindful of local performance.

3. Running Code, Tests, and Commands Locally

Unlike the ChatGPT web interface, the Codex agent can execute commands on your behalf in the context of your project. This is a game-changer for development workflows: Codex can run your test suite, start your application, or use build tools, then act on the results. Here's how it operates on Windows with a Conda environment:

- **Shell commands:** Codex's Agent mode includes an integrated shell tool. On Windows, by default it will use the system shell (often PowerShell or CMD). It runs commands *inside a sandboxed environment* for safety [28] [29]. By default, it's restricted to your workspace folder and has no network access (unless approved) [4] [14]. So if Codex needs to run `pytest`, it will essentially do so in a pseudo-terminal and capture the output back to the chat. You'll see the command and its output in the conversation, as if you ran it manually.
- **Approvals:** For running **within** the project (like `python manage.py test`), Codex usually proceeds without asking (Agent default permits workspace commands [4]). If it tries something with broader implications (like installing a package or accessing outside directories), the extension will prompt you. For example, "Codex wants to execute `pip install flask`. Allow?" – you can then

approve or deny. This mechanism ensures the AI doesn't do anything too destructive without your consent [30] [14].

- **Conda environment usage:** In our scenario, Python and dependencies are in a Conda env, not the system PATH. Codex *may not automatically know about Conda*, so it might attempt `python` or `pytest` and end up using the wrong interpreter if the env isn't activated in the shell. To address this:
- **Pre-activate environment:** The simplest approach is to open a VSCode integrated terminal (PowerShell or CMD) where your Conda env is activated (for instance, using a `.bat` script or the Anaconda Prompt). Codex, when running commands, can either use that existing terminal or spawn its own. Currently, it tends to spawn its own processes, which won't automatically have the Conda env. So another trick is...
- **Instruct Codex about Conda:** You can explicitly tell Codex, "Before running Python commands, activate the `myenv` environment." However, directly running `activate myenv` might not persist between commands (since each command could be a new shell). A more reliable way is to ask Codex to prepend `conda run -n myenv` to commands, or use the full path to the env's python. For example: *"Use the Python interpreter from the* `myenv` *Conda environment located at* `C:\Miniconda3\envs\myenv`*. For any command (like running tests), ensure you run it in that environment."* Codex may then do `conda run -n myenv pytest` when executing tests. There is an open request for better Conda support in Codex; some users have noted the agent can forget the env if not reminded [31]. Placing environment instructions in `AGENTS.md` (e.g. "Always use the `myenv` environment for Python") can help reinforce this.
- **VSCode Python extension**: Optionally, if you also have the MS Python extension, make sure your workspace is set to use the Conda interpreter. This doesn't directly configure Codex, but any tasks or debugging would use it. Codex's shell might not automatically hook into that, but at least if it runs `python`, the OS might find the one last used by VSCode (especially on Windows if you launched VSCode from an Anaconda Prompt, it inherits that env).
- **Running tests:** You can literally ask, "Run the test suite" or "Run `pytest -q`." Codex will execute it and return the output. If tests fail, it can parse error messages and even offer to fix the failing tests or the code causing them. For example, after running tests, it might respond: *"2 tests failed. Error: AssertionError in test_api.py line 42 ... I will update the function to handle this case."* It then presents a code change. This tight loop of *run → see failure → fix code → re-run* can continue until tests pass. This is essentially AI-driven TDD.
- **Running the app or scripts:** Similarly, you can have Codex run your application (e.g. `streamlit run app.py` or `python main.py`). This is useful if you want it to observe behavior or output. Note that if the app needs to stay running (like a server), Codex will either run it and possibly show some output, but it won't know when to stop it on its own. Typically, you'd use this for short-running commands. Long-running processes may need you to intervene (you can stop them if needed).
- **Build and lint commands:** If your project has build steps or linters (say `npm run build` or `flake8`), Codex can run those too. It's good to include such steps in your instructions ("verify code style by running `flake8`"). The prompting guide notes that giving Codex verification steps leads to higher-quality outcomes [32] – for instance, telling it to run a linter after making changes will make it fix formatting issues automatically.
- **Limitations on Windows:** The Codex Windows sandbox tries to mimic Unix-like behaviors but some differences remain. For example, Codex might attempt a Linux command (like `ls` or `grep`) out of habit. On Windows without a Unix shell, those will fail. Microsoft's sandbox intercepts some common calls and might stub them, but not all. If you see Codex trying a Linux command, you can correct it in prompt ("Use Windows commands or PowerShell for directory listing" or install tools like Git Bash/

MSYS2 and let Codex know they're available). Having MSYS2 or GNU tools in PATH can satisfy Codex if it expects them, but you should explicitly mention if those are present. Otherwise, you might occasionally have to guide it to use `dir` or PowerShell equivalents. Over time, the model has likely learned to adapt when on Windows (especially if it sees a `C:\` path, etc.), but be aware of this nuance.

4. Agent Modes: Chat vs Agent vs Full Access (Approval Workflow)

The Codex extension offers **multiple operating modes** that determine how "autonomously" the AI acts: - **Chat Mode:** *No automatic actions*. In this mode, Codex will only talk – it won't modify files or run commands at all [33]. Think of it as the standard Q&A or code generation mode, similar to ChatGPT web. Use Chat mode when you want to brainstorm, plan, or have the AI explain code, without any risk of it changing your project. It's the safest mode (read-only). This is useful for design discussions or high-level planning: you might switch to Chat mode and ask "How should I approach adding feature X? Let's plan it." Codex will then produce a step-by-step plan *without executing anything* (since it's in Chat mode). - **Agent Mode (Default):** *Limited autonomy with approvals*. In Agent mode, Codex can read and write files in your workspace and execute local commands automatically [4]. It will do so whenever it thinks it needs to, **but with some guardrails**: if it attempts to access files outside your project folder or make network requests, it will pause and ask for your approval [34]. This is the recommended daily mode for development. For example, if you ask it to "Update the dependency to the latest version," it might try to run `pip install -U package` – since that reaches out to the internet (PyPI), you'd get a prompt to approve that network access. Within your project, it might edit 5 files to adjust import statements and just do it (then show you the changes) because that's within the allowed scope. - **Agent (Full Access) Mode:** *Maximum autonomy*. In this mode, Codex will not ask permission for anything – it's free to modify any file it can see and access the network freely [14]. This is powerful but risky. It's only suggested in highly controlled environments (or ephemeral ones). For instance, in a throwaway sandbox or a CI pipeline, you might use Full Access so it can, say, auto-update dependencies or fetch documentation online without prompting you at each step. On a sensitive codebase, you would typically **not** use Full mode during interactive development, because an unchecked AI could theoretically send code externally or make unintended changes. In our Windows 10 scenario, using Full Access might also require tweaking the sandbox settings (the extension's config has options to disable the network sandbox) [35], but generally it's a toggle in the UI ("Agent (Full)").

Approval workflow: With the default Agent mode, expect to occasionally grant or deny actions. The extension UI will pop up a small notification (e.g. "Codex requests permission to run `curl` … [Allow] [Deny]"). These moments are your chance to stay in control. Best practice: **keep an eye on what commands it's asking to run.** They should usually make sense (like running your tests, installing a missing library, etc.). If something looks off (e.g. accessing an unrelated URL), you can deny and ask it for an explanation or re-think the approach.

Plan-first prompting: If you prefer more control, you can explicitly instruct Codex to *only plan and not execute* at first. For example, in Agent mode you might say: "Outline the steps you will take to refactor module X, but do not make any changes yet." Codex will then behave like it's in Chat mode temporarily – listing steps or a pseudo-code plan. You can iterate on that plan with it, and once satisfied, you either instruct "Okay, proceed with implementation" or manually switch it to agent and give the go-ahead for each step. This lets you see its reasoning. In fact, GPT-5 Codex is designed for "autonomous task planning and execution" [36], so it often internally makes a plan. By asking it to externalize that plan, you gain insight and

can adjust instructions before it writes any code. This technique is great for complex or critical changes: you essentially do an AI code review of the plan *before* any code is written.

5. Handling Large Tasks: Cloud Delegation

If you hit a task that is too large or slow for your local machine (or the VSCode extension's context), ChatGPT Plus subscribers have the option to use **Codex Cloud** tasks. This feature lets you "offload bigger jobs to Codex in the cloud" [37] – effectively spinning up an OpenAI-managed server that has your code and can run the agent with more resources. Key points: - You can initiate a cloud run by selecting "Run in the cloud" for a task in the extension [38]. For instance, after a long conversation, you might have an option to delegate the next steps to cloud. - The first time, you'll need to set up a cloud environment. This is done through your ChatGPT account (the extension will direct you to a setup page) [38]. You can typically choose parameters like Python version, or even provide an image (for example, a Docker base or a GitHub repo link). - Once running, the cloud agent remembers the conversation context you had locally [39]. It will attempt the task and you can track progress in VSCode as if it were doing it locally (you'll see file changes or outputs). - A common use is to let the cloud agent handle long-running processes or very large refactors, then have it *return the diff or results* to you. You can review the cloud-generated changes and apply them to your local environment when ready [40]. - For Windows users, cloud tasks can circumvent local issues (like lack of certain tools or performance constraints). The cloud environment is Linux-based by default (since that's what OpenAI's sandbox supports best [29] [41]). That means if, for example, Codex was struggling with Windows path issues locally, running the task in cloud (Linux) might avoid those. Of course, you'd then need to test locally, but code changes themselves are cross-platform. - **No extra cost**: Cloud usage is part of the Plus plan entitlements (with some limits). It consumes your "GPT-4/5 usage credits" similarly to local usage. Essentially, you're using the same model but allowing it to use OpenAI's compute for executing code or handling bigger contexts. Be mindful that extremely large tasks might count as multiple operations.

In summary, use cloud delegation for **heavy-duty tasks**: e.g. "Regenerate documentation for the entire codebase," or "Refactor the code to use asynchronous I/O throughout" – things that involve many files or long reasoning. For everyday quick fixes, the local agent mode is faster and sufficient.

6. Reliability and Limits in Large Projects

A question often arises: *Can Codex handle really large projects (100+ files, multiple modules)?* The experience so far: - **Understanding project architecture:** If the project is well-structured (clear modules, good naming, a central README or AGENTS.md describing it), Codex can navigate it surprisingly well. It won't "read" every file at once, but it can locate functions or classes by name. For example, "Find where `UserProfile` is defined" – it might use an internal search or simply the fact that file names often match class names (it could guess `user_profile.py`). Sometimes it will just ask you or state it needs guidance if uncertain. - **Performance on large codebase:** On Windows local, performance might degrade if Codex tries to scan many files. Unlike Copilot, which pre-indexes the whole repo, Codex tends to load files on demand. As a developer, you can help it by providing entry points: "Open `core/models.py` and check the class definitions" – guiding it to relevant areas. The VSCode extension itself does some smart context management (it might have a limit on how many open file contents it sends at once). - **Token limits:** You might encounter messages like "*Context limit reached, cannot open more files*" if you attempt to include too much at once. In that case, break your query down. For instance, instead of "Explain everything about this project," ask "Explain the purpose of module A," then "Now how does module B interact with A," and so on. - **Usage rate limits:** As a Plus user, you have a cap on how many requests you can make in a period (e.g.,

initially around 50 requests/hour for GPT-4, which may equate to 30-150 every 5 hours as observed [42]). Each action (question, file edit, command run) counts. On a very large project, you might burn through these faster, especially if Codex is running tests repeatedly or reading many files. The extension will notify you if you hit a limit ("You've reached the usage limit, please wait..."). If this becomes a frequent problem, it might warrant considering an upgrade (Pro/Business plans raise the limits significantly [42]) or using an API key on a pay-as-you-go plan as a supplement. However, for most single-developer use, Plus's limits are generous as noted [42] [43]. - **Stability:** Given that Windows support is flagged *experimental*, you might hit occasional glitches (the extension freezing, or failing to apply an edit). Ensuring you have the latest version is important – it auto-updates, but you can check VSCode extensions for updates regularly [44]. Also install any Windows dependencies noted (like the Visual C++ Redistributable and Build Tools) to avoid crashes [45]. If things go awry mid-session, you can often just close and reopen VSCode, or toggle the extension off and on.

In practice, many users have successfully used Codex on large projects, but they do so by **strategically controlling context and scope**. We'll cover project layout and prompting strategies later to help with this.

Python Workflow in a Conda-Only Windows 10 Environment

Now let's focus on making Codex work smoothly with a **Conda-managed Python** setup on Windows 10. The key challenge here is that typical AI coding tools assume a straightforward environment (often system Python or a venv). We have no system Python and want Codex to use Conda envs exclusively.

1. Environment Setup and Activation

Conda activation in VSCode: First, ensure that your Conda environment is accessible in VSCode. If you open a normal VSCode terminal, it won't automatically activate an env unless configured. There are a couple of ways to manage this: - Use the Anaconda Prompt or a custom task: You can create a VSCode *terminal profile* that runs something like `cmd.exe /K "C:\Miniconda3\Scripts\activate.bat myenv"`. This will open a terminal with the env activated. Keep this terminal open. - When you instruct Codex to run something, by default it might spawn its own process rather than using your open terminal. However, as a workaround, you can copy-paste test commands into that activated terminal and then show Codex the output. Or, explicitly tell Codex "Run the tests using the already active terminal" – it might then use a different strategy (like expecting you to run it). This is a bit clunky; an alternative is: - **Use `conda run`:** As mentioned, prefixing commands with `conda run -n myenv` forces them to run under that env's context. It's a one-shot execution and doesn't require activation state. Instruct Codex early on that your environment is called "myenv" and to use `conda run` for any Python-related command. Once given, it often remembers to do so (though sometimes a gentle reminder is needed if it forgets). This prevents the common issue of Codex running `pip` or `python` against the wrong interpreter.

Dependency detection: Codex can figure out a lot about your environment by reading your files: - If you have a `environment.yml` or `requirements.txt`, it will likely notice those. It might even open them to see what libraries are listed. It doesn't automatically install them (unless you ask it to), but it uses them to understand available packages. - If you encounter an import error, Codex may suggest "It looks like `xyz` isn't installed. Should I install it?" – because it saw the failure when running. With our no-API setup, if you approve, it will run `conda install` or `pip install`. Since we prefer conda, you might instruct it to always try `conda install` first. Keep in mind, conda installations can't happen if the environment is not

activated in that shell. `conda run` can install packages too (`conda run -n myenv conda install package -y`), albeit it's a bit meta. In many cases, using `pip` inside the env is fine as well, especially if it's just adding a library for Codex to proceed. (If internet is blocked by sandbox, you'll need to allow it for these install commands.) - Codex does not inherently know your interpreter path, but if you've configured the Python extension, that path might be in your `.vscode/settings.json`. Codex can read that too if needed. However, it usually doesn't need the exact path – as long as the correct `python` is called via activation or `conda run`, that's sufficient.

Virtual environments per workspace: If you use multiple Conda envs for different projects, each VSCode workspace will likely target its own env. Codex doesn't persist environment info globally (except maybe in global instructions if you set them). You should clarify in each new project conversation which env to use. For example: "For this project, the interpreter is Python 3.10 in a Conda env with Flask and SQLAlchemy installed." This helps the AI avoid suggesting things incompatible with your environment (like using Python 3.11 features if you're on 3.8, etc.). It's essentially giving it the *constraints of your environment* up front.

2. Directing Codex in Python Tasks

Running tests: After setting up environment concerns, asking Codex to run tests is straightforward: e.g., "Run `pytest` to ensure everything passes." On Windows, if you see an error like `'pytest' is not recognized`, that indicates the env was not active (pytest not on PATH). That's your cue to enforce the `conda run` approach or manually install/activate. Once resolved, Codex will show test results. It's quite adept at reading Python tracebacks and will often zero in on the line and cause of failure, then jump to editing the code to fix it.

Inspecting modules: You can have Codex open and explain parts of your code. For instance: "Open `database/models.py` and explain the class hierarchy." Codex will display the file (or relevant excerpt) and provide an explanation in plain English, which is great for understanding unfamiliar code. This doesn't change anything, it's just using its reading capability to summarize or answer questions about the code. It's like an always-available second pair of eyes that can parse code and docs for you. If you ask something it can't determine from code alone (e.g. "What version of Django is this project using?"), it might search your `pyproject.toml` or requirements to find it, or even run `pip show django` if needed.

Refactoring across packages: Python projects often have multiple interdependent modules. For example, you want to move a function from `utils.py` to `helpers.py` and update all imports. Codex can handle this: you'd describe the change ("move function X from utils to helpers, and update all references in the project"). It will: - Create or modify `helpers.py` to include the function. - Remove it from `utils.py`. - Find all `import` statements or usages of `X` in other files and change them to import from `helpers`. - Possibly run tests to ensure everything still works (if you instructed it or if it's cautious). Because it can multi-edit, this happens in one conversation turn usually, rather than you doing find/replace manually.

Another scenario: say you have a package `models` and you want to split it into `models.user`, `models.product`, etc. You can instruct Codex with the new structure. It might generate multiple new files for each submodule and redistribute code accordingly. This is essentially *architecture generation*, which we cover in the next section.

Project structure best practices for AI: To help Codex (and yourself), organize your Python project clearly: - Use a consistent naming scheme so files are easily identified by purpose (the AI often infers content from the name, e.g. `views.py` likely contains web views). - Keep one class or logical unit per file where possible. A 5,000-line single file is harder for Codex to work with than the same code split into 10 files; it can load smaller files individually as needed. - If you have data models, consider using Python dataclasses or clearly defined schemas – Codex will then use those as source of truth. It reads type hints and docstrings to understand function and class contracts. - Provide a **README or AGENTS.md** with project setup and guidelines. For instance, mention the Python version, important design patterns, and any deviation from standard practices. In our Conda context, note the environment name and how to run things (so the AI doesn't assume something incorrect). - Document your functions and classes. Codex will sometimes use the docstring as part of its reasoning. If a function says "# TODO: optimize this" in a comment, Codex picks up on that and might attempt to address the TODO if asked to improve code quality.

Developer guides for agents: The `AGENTS.md` file is a powerful way to inject persistent instructions (like a developer guide specifically for the AI). For example, in a Django project, your `AGENTS.md` might include: "This is a web app using Django 4. Views are in views.py, models in models.py. Follow PEP8 style. Always run `black` (formatter) after making changes. Our test command is `pytest`." Codex reads this every session [6] [7], so it will automatically follow those rules (e.g., it might actually run the formatter if instructed, or at least produce formatted code). In a Windows Conda project, *definitely mention environment specifics here.* E.g., "Python environment: Conda env named `myenv`. Use `conda` for installs. Do not use Linux-only commands." By front-loading these into AGENTS.md (or even just the first prompt), you save yourself from repeatedly correcting the AI.

3. Example: Typical Python Agentic Workflow

To illustrate a typical flow, let's combine these pieces: - You open VSCode in your project (say a Flask web app). You've installed the Codex extension, signed in, and you're in Agent mode (default). - In AGENTS.md (project root) you've written some instructions: "Flask app project. Python 3.9. Conda env `flaskenv` contains all dependencies. Start app with `flask run`. Database via SQLAlchemy. Ensure to update `requirements.txt` if new packages installed." - You start a conversation: *"Hi Codex, let's work on this Flask app."* Codex might greet and stand by for tasks. - **Step 1:** *"Plan out adding a new feature: user profile pictures. Don't make changes yet."* (You're using plan-first approach in Chat mode or by explicitly instructing no action). Codex responds with a numbered plan, e.g., "1. Add a field `profile_image` to User model. 2. Create an upload endpoint. 3. Save files to static directory. 4. Update user profile template to show image. 5. Write a test for image upload." You review and perhaps discuss the plan. - **Step 2:** You approve the plan and say, *"Okay, implement steps 1 and 2."* You switch to Agent mode (if not already). Codex proceeds: it opens `models.py`, adds the `profile_image` field, possibly runs `alembic` for a migration (if it infers you use migrations – it might ask if uncertain). It then creates a new route in `routes.py` or `views.py` for file upload. It might realize a package like `werkzeug` is needed for secure filenames – if not installed, it asks to install it (`pip install werkzeug` or `conda install`). You allow it. - It writes the code, shows you diffs. You review the changes right in VSCode diff view. If something is off, you correct by telling Codex, or manually editing and then telling Codex to incorporate your edit. - **Step 3:** *"Now, run the app's tests."* Codex executes `pytest`. Suppose one test fails because an expected field is missing in JSON response. Codex identifies that and updates the relevant serializer or route to include the new field. It says "Updated the response to include profile_image. Re-running tests." Tests pass now. - **Step 4:** *"Great. Please run the app to do a quick manual check."* Codex runs `flask run` (if allowed). The app starts (this will occupy the shell). Codex may not give a response while server is running. You open a browser manually to verify the upload

works. Once satisfied, you stop the server and tell Codex. - **Step 5:** *"Everything works. Commit the changes."* While Codex cannot directly `git commit` (unless you give it permission and credentials via an MCP maybe), you can ask it to **draft a commit message or PR description**. For example: *"Generate a commit message summarizing these changes."* Codex might produce: "feat: add profile picture upload feature – Added `profile_image` field to User model, endpoint `/upload_photo`, updated templates and tests [24]." You copy this and use it in your Git commit. (If you had the GitHub MCP configured, Codex could even push a branch or create a PR on GitHub automatically, but that's beyond the no-API basic setup.)

This example shows how the agent moves through planning, coding, testing, and documenting, all within VSCode, using the Conda environment for execution. With practice, you can accomplish complex tasks in a fraction of the time, while still overseeing the critical decisions.

Agentic Coding Workflows in VSCode: Step-by-Step Guides

Now, let's break down specific advanced workflows you can employ with ChatGPT Plus and Codex. These are **explicit, replicable procedures** demonstrating the "agentic" development style – where the AI participates in multiple stages of coding.

1. Plan-Driven Development Workflow

Use Case: You have a vague feature request or bug report. You want Codex to first **propose a plan** before writing any code, so you can ensure its approach aligns with your expectations.

How to do it: 1. **Switch to Chat Mode** (or instruct "plan only"). Since Chat Mode won't execute or edit files [33], it's ideal for planning. You can do this via the dropdown under the chat input (select "Chat") [33]. 2. **Describe the task and request a plan:** For example, you say: *"We need to implement a new caching layer for our API calls. Outline a step-by-step plan without coding yet."* Emphasize "no coding yet" to ensure it stays in planning mode. 3. **Receive and review the plan:** Codex will enumerate steps. E.g.: *"Step 1: Identify functions to cache. Step 2: Choose a caching library (like functools.lru_cache). Step 3: Implement cache decorator. Step 4: Write tests for caching behavior,"* etc. It might also mention design decisions. 4. **Refine the plan (optional):** You might discuss with Codex: *"Actually, use an in-memory cache dictionary instead of lru_cache, because X reason."* Codex will adjust the plan. 5. **Approve and execute step-by-step:** Once the plan looks good, you have two approaches: - *Manual step approval:* Tell Codex "Proceed with Step 1." It will switch (or you switch it) to Agent mode and implement step 1 (e.g., add caching to one function). It shows the changes. Then you say "Looks good, do Step 2," and so on. This is slow but very controlled. - *Full plan execution:* If you're confident, you can say in Agent mode: *"Go ahead and implement all the steps."* Codex will then try to carry out the entire plan in one session. You'll see it moving through each step (often it will comment in the chat like "**Plan Step 1:** …done; **Step 2:** …done"). If at any point it hits an issue (like something doesn't compile or a test fails), it may adjust on the fly or ask for guidance. 6. **Verify after execution:** Once done, test or review the changes. This workflow's strength is that you had oversight on the strategy *before* any code changed, reducing surprises. It's essentially forcing the AI to show its "pseudo-code" or thought process. This maps well to how senior engineers work – outline first, implement second – and Codex can follow this discipline when asked.

Tip: If Codex ever skips planning and jumps to coding (sometimes it's overeager), gently pull it back: *"Hold on. Please give a plan first."* It will correct itself. As models get more agentic, they're trained to respect such user preferences.

2. Multi-File Architecture Creation Workflow

Use Case: You want to create a new module or even an entire small project from scratch (or a significant extension to an existing project). This involves multiple files that need to work together – for example, adding a new package with several submodules, each responsible for something.

How to do it: 1. **Explain the architecture in natural language:** Start with a high-level prompt describing what you want. E.g.: *"Create a new Python package* `analytics` *with the following: a module* `stats.py` *defining statistical functions (mean, median, etc.), a module* `report.py` *that generates a report using those stats functions, and an* `__init__.py` *that exposes the main classes. Ensure proper imports so that* `from analytics import report` *works."* 2. **Let Codex propose file structure:** Codex may respond by listing the files it will create and their content outline [46]. If it doesn't, you can prompt: *"List the files you will create and their purpose first."* This is similar to plan mode, but specifically about files. 3. **Generate files one by one or in batch:** In Agent mode, Codex can now proceed to create these files. It might do it sequentially in one answer: "Creating `analytics/stats.py`..." followed by code, then "Creating `analytics/report.py`..." and so on. VSCode will show these as new unsaved files in the editor (or it might directly save them, depending on settings). 4. **Review each file's content:** Check that naming conventions are consistent (Codex is generally good at this if you describe them upfront). For instance, if you asked for `mean()` and `median()` in stats, verify they exist and are imported in `report.py`. If something is off, you can correct it or tell Codex to fix, e.g., *"In* `report.py`*, import the mean function from stats."* It will adjust. 5. **Add package boilerplate:** Codex will likely create an `__init__.py` as requested, possibly to import certain functions or classes for convenient access. Ensure this aligns with your intention (maybe you want `analytics.calculate_mean` to be usable directly, etc.). If you forgot to mention something like including a version or metadata, you can still add it. 6. **Documentation and types:** If you want, ask Codex to add docstrings or type hints to the new modules. For example, *"Add docstrings to all functions in stats.py and ensure they have type hints."* It will edit accordingly. Because this is a fresh creation, it's a good chance to get those in from the start. 7. **Integration with existing code:** If this new package is to be used by existing code, next instruct Codex to update those references. E.g., *"Use the new analytics package in* `main.py` *to produce a stats report for sample data."* It will then open `main.py` and show how to use your new package (this also serves as a quick test that the new code works). 8. **Testing the new modules:** You can have Codex write some basic tests for the new package too: *"Create* `analytics/test_stats.py` *with unit tests for mean and median."* It will generate tests. Then run them via Codex to ensure everything passes. This ensures the multi-file system it created is functioning as a whole.

This workflow demonstrates Codex's ability to **maintain consistency across multiple new files** in one go. It will carry over naming conventions, share utility functions, and manage imports such that the new code is integrated. All of this without having to run each file creation as a separate ChatGPT prompt – the extension context keeps track of the project structure as it evolves. As a user, you still define the architecture (you tell it what modules to have), but Codex handles the boilerplate and most of the implementation.

3. Code Execution and Testing Loop (Local)

Use Case: You have an existing test suite or an application that you want to run to catch issues, and then let Codex fix them. This is akin to an automated debug/fix cycle.

How to do it: 1. **Ask Codex to run the code or tests:** e.g., *"Run the full test suite."* As covered, Codex will execute the tests in the sandbox shell [4]. You'll get output. If all tests pass (great!), you can even ask "any

improvements needed?" and it might do code cleanup. But let's assume some tests failed or an error occurred. 2. **Observe and identify failures:** Codex will typically summarize the failing tests or errors after the output. It might say: "Test X failed: expected 200 response, got 500. Likely cause: the new caching layer not handling null." If it doesn't summarize well, you can ask *"What went wrong? Analyze the test failure."* 3. **Let Codex propose a fix:** Often it will immediately suggest a fix after running tests (the chain-of-thought is: run -> see error -> decide fix -> present diff). If not, prompt it: *"Fix the error causing the test failure."* In Agent mode, it will edit the relevant files to address the issue. For example, adjust the caching logic to handle null values. 4. **Re-run tests:** After the fix, Codex may automatically re-run tests to verify (it tends to do this if it was the one running them originally, unless you prevent it). If it doesn't, you prompt *"Run tests again."* This loop can continue until tests are green. 5. **Interactive debugging:** If running the app rather than tests, say the app crashed with a stack trace, Codex will capture that output and can help pinpoint the problem. You can do *"The app crashed, please debug this."* It will examine the trace, open the file and line where the error occurred, and likely fix the bug. This is hugely helpful for runtime errors that aren't immediately obvious. 6. **Edge cases & additional tests:** Codex might even add a new test for a bug it fixed (especially if you say "add a regression test to ensure this doesn't happen again"). It can write that test, then run it to prove the bug is solved. 7. **Repeat as needed:** This cycle can be done anytime – it's particularly powerful after a lot of changes have been made (to verify nothing broke). It's like having an AI QA assistant on standby.

Note on Windows/Conda: If tests involve any external binaries or services, ensure those are accessible. For instance, if tests call a local database or need `sqlite3` library, make sure those are installed in the env. Codex will handle pure Python issues, but environment issues (like missing executables) might need you to intervene (the AI might not be able to install, say, Redis server by itself – that's outside Python's realm, but it might at least tell you "Redis is not running" if a test fails on that).

4. Structured Coding from External Artifacts

Use Case: You have design documents, TODO comments, GitHub issues, or other artifacts that describe what needs to be coded. Instead of manually translating those into code, you want to feed them to Codex and have it do the heavy lifting.

There are several sub-scenarios here:

A. Using Design Docs & Diagrams: Suppose you have a Markdown or text design spec (could be a local file or a wiki page). - Copy the relevant parts of the design text and paste it into the Codex chat (if it's large, do it in pieces or ensure it's within token limit). You might say: *"Here's the design for the new feature:\n[paste design spec]\nPlease implement according to this."* - Codex will read that spec from the conversation and follow it. It might refer back to the spec explicitly: "The design says we need a class X with method Y, I will implement that in module Z." - If the design includes a diagram (like an image), you can't directly feed that (Codex can't "see" images unless converted to text somehow). You'd need to describe the diagram or the important points from it in text form. - The result is the AI effectively uses the spec as a blueprint. This reduces miscommunication – it will align code with what the spec says (function names, algorithms, etc., as long as they were described).

B. TODO / FIXME Comments: Many codebases have inline TODOs. Codex can find and address them systematically: - Ask: *"List all TODO comments in the repository."* Surprisingly, Codex might actually attempt to grep your project for "TODO" if allowed. It could also open files it suspects (it might open every file one by one searching for TODO lines – potentially expensive). If it's a moderate codebase, it can succeed. If not, you

might manually gather them (or do a quick search in VSCode yourself and paste the results). - Once you have the list of TODOs (either provided by Codex or by you), decide which ones to tackle. You can even number them and go one by one: *"Address TODO 1: Optimize the sorting algorithm in function* `sort_data` *(from utils.py)."* - Codex will then open `utils.py`, find that TODO comment, remove it, and implement the intended solution (maybe using a better algorithm or a note you left in the comment). It's very satisfying to watch your TODO list burn down this way! - After each fix, run tests or ensure it didn't break anything. If a TODO was left because something was tricky, verify the AI's solution carefully.

C. Using GitHub Issues as specs: Let's say an issue says "Feature request: add pagination to the user list endpoint." You can copy the issue description (maybe it has acceptance criteria, etc.) right into the chat. Prompt: *"Implement the following GitHub issue:\n>>> [issue text pasted] \n<<<\nFollow all acceptance criteria."* - The `>>>` and `<<<` here are just a way to indicate to the AI that this is quoted content (not a requirement, but can help separate it). Codex will then treat that like a spec. - It will implement the feature addressing each point from the issue. If the issue is a bug report with steps to reproduce, Codex will attempt to reproduce (maybe by writing a test representing those steps) and then fix the bug. - **No API needed:** This approach doesn't require Codex to directly access GitHub. You're manually giving it the info. It's effectively the same as you reading the issue and telling Codex what needs doing – except you can just supply the raw text, and Codex might even summarize or clarify it for you.

D. Using PR diffs for refactoring: If you have a git diff or a PR that someone opened (or that you plan to open), Codex can help interpret or polish it. - For **code review assistance**, copy the diff (unified diff format, or just a before-and-after code snippet) into the chat and ask: *"Review this diff and point out any potential issues or improvements."* Codex will analyze the changes and could catch bugs or suggest better approaches. This is great for double-checking critical patches. - For **writing PR descriptions**, after making a bunch of changes with Codex, you can ask it to *"Summarize the changes in a bullet list for a pull request description."* Because it knows what it just did, it will list something like "– Refactored X module for clarity, – Fixed bug in Y, – Added caching to Z improving performance by ~20%." You can then paste that into your PR body. - For **commit messages**, similar to above, but instruct the style (e.g., conventional commits format). Codex can output: "fix(auth): handle token expiration properly (closes #123)" if it knows the context.

E. Combining multiple artifacts: You might have a design doc, plus a related GitHub issue, plus some TODOs in code – all related to one bigger feature. Don't be afraid to feed *all* relevant context to Codex, one after the other: 1. Paste the design overview. 2. Paste the specific issue that tracks it. 3. Say: *"Using the above requirements, implement the feature. Also, there are TODOs in the code related to this – address those too."* Codex will integrate all these inputs. It's generally very good at synthesizing information and following the most concrete instructions (if the design doc and issue overlap, it will merge the ideas).

By leveraging external structured content, you ensure Codex's output is aligned with stakeholder requirements and not just guesswork. It effectively acts as an executor of the specification.

Using GitHub as Structured Context (without API Access)

We touched on using GitHub artifacts by copy-pasting. Here we'll explicitly address how to incorporate GitHub Issues, PRs, and other GitHub data *when you don't have direct API integration* (i.e., no special permissions given to Codex).

Manual ingestion of GitHub data: Without an API key or the GitHub MCP, Codex cannot on its own retrieve issues or PRs from GitHub. So, it's up to you to provide that info: - **Issues:** Open the issue in your browser, copy the text (description and maybe relevant comments). Paste into VSCode and then into Codex chat. If the text is long, you can summarize or instruct Codex to summarize it once you paste. - **Pull Request diffs:** On GitHub, you can get a unified diff (`.patch` or `.diff` URL) or simply copy changes from the "Files changed" tab. Paste that into Codex. If the diff is huge, consider summarizing sections (or ask Codex to summarize the diff for you first, which it can). - **Repository Wiki/Docs:** Similarly, if an architectural diagram or documentation is on GitHub (maybe in Markdown in the repo or a wiki), copy the relevant parts in.

Constructing specification prompts: The idea is to prepend any code-writing instruction with the context. For example:

```
**Spec:**
- The API should support pagination (issue #123).
- Page size is 20 by default, configurable via query param `?page_size=`.
- The response should include `next_page` URL if more results.

**Task:** Implement pagination in `users_list` API endpoint according to the
above spec.
```

By clearly delineating the spec (which came from an issue or design) and the task, you ensure Codex has all the info. It will likely even include parts of the spec in its answer to confirm it's meeting them.

AI-generated PRs and patches: Interestingly, Codex can generate a patch file for you if you ask: *"Generate a unified diff for the above changes."* It knows the format (diff headers, etc.). So if you prefer to review changes before applying, you could have it output a diff that you then `git apply` manually. However, since the extension can directly edit files, this is usually not necessary – you see the changes in your workspace already. But it's a cool option if, say, you want to send a patch to someone or apply it later.

Commit message and changelog generation: After a set of changes: - *"Draft a commit message:"* Codex will use its knowledge of what changed (from conversation memory) to create a commit message. Always double-check that it correctly captures all changes. Sometimes it might miss something that wasn't discussed even though code changed. - *"Update the CHANGELOG.md:"* If you have a changelog file with a format, Codex can append a new entry. Provide context like version or date if needed. E.g., "Under Unreleased, add an entry: 'Added: User list pagination support'." It usually can follow your changelog format if it's simple bullet lists or headings.

Automating issue chores: While Codex can't directly manipulate GitHub issues without integration, it can assist in: - **Issue triage:** Copy a list of new issue titles into Codex (or do one by one). Ask *"Categorize these issues by bug/feature/support, and suggest a priority."* It will read the titles/descriptions and give you a categorization. This can help you decide what to do first. - **Reproduction steps code:** For a bug issue with given steps, you can say *"Write a test that reproduces issue #456 based on the description."* It will create a test function that follows those steps. Running that test will fail (demonstrating the bug), then you can proceed to fix with Codex. - **Code review assistance:** If you're reviewing someone's PR, you can copy their diff and have Codex highlight potential issues or even suggest improvements. Then you could use those in your code review comments (though be careful to verify, as you are responsible for the review ultimately, not the

AI). - **Continuous integration fixes:** If a CI pipeline reports a failure (maybe a linter or a test failing on remote), you can feed the log to Codex. For instance, paste the relevant part of CI log and ask *"Fix the coding style issues flagged by flake8 in the log above."* Codex will fix style (it recognizes common linter outputs). This is similar to local test fixing but using CI output.

In essence, even without direct API hooks, Codex serves as a *bridge* between human language specs (issues, docs) and code. By copying those artifacts in, you preserve the "single source of truth" – the AI isn't hallucinating feature requirements; it's following the same written criteria you are. This leads to more reliable outcomes.

(For those interested, with a GitHub MCP server configured, Codex could fetch issues or post comments by itself [18]. But that requires setting up a personal token and is beyond the "Plus only" scenario. It's good to know it's possible, but not necessary for benefiting from issue-driven development.)

CLI Tools and Optional Ecosystem Integration

Thus far, we've assumed you're using the VSCode extension exclusively. Now we'll discuss what additional tools exist (like the OpenAI CLI and related utilities), and when you might need them in this Windows + Conda context.

1. OpenAI Codex CLI

What is it? The Codex CLI is a command-line application (Node.js-based) provided by OpenAI that offers a similar AI coding assistant experience in the terminal [8] [22]. It's essentially the engine that the VSCode extension uses behind the scenes (the extension is built around the open-source CLI) [22]. You can run it in any terminal or even integrate it with other editors that aren't VSCode.

Installation: On Windows, you can install it via Node's package manager:

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

(as per official docs and guides [47] [48]). You may need to ensure Node.js is installed and perhaps the Microsoft Build Tools (for any native addon). If the extension is working, the CLI likely will too, since they share dependencies.

Using it with Plus account: The CLI can authenticate via the same ChatGPT account. On first run (`codex` command), it should prompt you to log in (likely opening a browser to auth with OpenAI, similar to how the VSCode extension did) [49]. Alternatively, you can configure it with an API key if you had one, but in our scenario we'd use the ChatGPT auth. Once logged in, it uses your Plus credits just like the extension does – no additional charge.

Capabilities of CLI: - It opens an interactive Terminal User Interface (TUI) where you can chat with Codex. You type prompts, it replies. It can do the same actions: read files, write files, run shell commands. - In the TUI, there are commands (like slash commands) to do things. For example, you might use `/open filename` to load a file into context, or it might automatically mention that it opened a file when reading it.

- You still need to be in a directory (you launch `codex` in your project folder). It then treats that as the workspace. - **Agent modes:** The CLI also supports the modes – by default it might run in Agent mode. Usually it asks you on startup what mode or model to use. You can configure a default in `~/.codex/config.toml` (the same config the extension uses) for approval policy, etc. - **MCP config:** The CLI reads the same `config.toml` for MCP servers configuration [50] [51]. So any MCP you set up (like GitHub or Context7) via the extension or CLI is shared.

When to use CLI instead of VSCode extension: - **Headless/Automation:** If you want to script something or use Codex in a CI pipeline, the CLI is the way. For instance, you might have a CI job that runs `codex --someFlags "fix all ESLint errors"` on a pull request branch, which then commits the fixes. (OpenAI also provides a GitHub Action for such use [19]). - **Other Editors or IDEs:** If sometimes you use a different editor (say Vim or PyCharm) but still want Codex help, you could run the CLI alongside your editor. It won't integrate into the editor UI, but you can copy suggestions from the terminal. - **Stability or performance:** The VSCode extension is convenient but it's another layer that could crash or slow down (particularly on Windows experimental support). If you find the extension is not stable in your setup, the CLI might be more stable to use, albeit less visual. You can always edit files manually while the CLI is giving instructions. - **Debugging Codex issues:** Because the CLI is open source, you could run it with verbose logs or see error messages that the extension might hide. For instance, if Codex crashes or doesn't respond in VSCode, the CLI might show that an error occurred with more detail. - **No GUI environment:** In cases where you SSH into a Windows server or something without a GUI and still want to use Codex, CLI is the only option.

Comparison to VSCode extension: - The features are largely **parity**. Anything the extension does, the CLI can do, since the extension is just a wrapper around it [22]. The difference is UI: VSCode shows diffs and inline edits, the CLI will show you textual diffs or just say "file X edited". - In VSCode, approval prompts are buttons; in CLI, you likely type `y/n` or a command to approve when it asks. - VSCode automatically uses open file context; in CLI you may need to explicitly open files or rely on Codex to guess and open them. - For learning purposes, using the CLI can actually expose more of the underlying process (like you see exactly what commands it runs, etc.). It's educational to try it at least once, to demystify the extension.

Windows 10 + MSYS2 considerations: - The CLI will try to run shell commands. By default on Windows, I believe it uses PowerShell as the shell (the config might have an option, or it just uses the Node `child_process` to spawn commands). It will be under the same sandbox rules as extension. - If you have MSYS2 (or Git Bash) and prefer Codex use a Unix-like shell, you might configure `shell_path` in config.toml or use an MCP for shell if that exists (though likely not needed). Otherwise, to Codex CLI, MSYS2 presence just means commands like `ls` may be available. It could naturally detect if Bash is available. - Running Docker-based MCP servers on Windows likely requires Docker Desktop (which in turn might need WSL2). If you avoid WSL, ensure Docker is set to use Hyper-V backend. - CLI can definitely be run in a normal Windows Terminal or PowerShell. Just be mindful that if it tries to execute Linux-specific stuff, you might have to correct it (same as extension).

2. Tasks Requiring API keys vs Included in Plus

No API key needed for core usage: As shown, everything from editing code, running commands, to even cloud tasks is included in the ChatGPT Plus integration [1]. You do **not** need a separate billing account with API credits for these. However, there are scenarios where an API key (OpenAI's developer API) might be needed or beneficial: - **Extending beyond Plus limits:** If you run into the request limits of Plus frequently, one way to get more is to use an API key and call the models directly. The extension and CLI allow

configuration of an API key usage (so instead of consuming your Plus quota, it charges your API account). For heavy users, this might be worthwhile – essentially treating Codex like a paid API service with possibly higher rate limits. - **Batch or backend usage:** Suppose you want to embed Codex into an automated script (outside of CI, say a custom tool). Using the API (with the Codex CLI or SDK) might integrate better. The ChatGPT Plus account is user-centric and interactive; the API is programmatic. - **Certain features and models:** OpenAI might expose some model or feature via API that's not in the Plus UI. For instance, if a specialized smaller model for quick tasks is available only via API. In our context, GPT-5-Codex is available in the extension by default [3], so that's fine. But if, say, you wanted to fine-tune a model or use the embeddings API for some reason, that requires an API key (Plus doesn't include fine-tuning access by itself). - **MCP servers needing API auth:** This is subtle: some MCP servers might require you to provide *their* API keys. For example, a "Web browsing MCP" might need a token for a web API service (like BrightData proxy or others). That's separate from OpenAI – but still, an API of another service. - **OpenAI Tools/Plugins:** If you wanted to use ChatGPT plugins or the code interpreter outside the ChatGPT UI, you'd likely need API endpoints. But the VSCode extension largely covers those capabilities through Codex Cloud (code interpreter equivalent) and internet access (browsing) in agent mode.

OpenAI CLI (not Codex-specific): There's also an **OpenAI Python CLI/SDK** (for interacting with the API). Don't confuse this with Codex CLI. If you did install the OpenAI Python library to do things like `openai.ChatCompletion.create()`, that strictly requires an API key and uses your pay-as-you-go credits. That's an entirely different workflow (you writing Python scripts to use the model). Our focus is on the Codex developer experience, so you don't need that unless you plan on building your own integration.

Advantages of having CLI tools (Codex CLI or OpenAI API CLI): - For the Codex CLI, as described: flexibility, use outside VSCode, scripting capabilities. - The CLI also provides a way to manage configuration (like the `codex mcp` commands to add servers) more conveniently than editing a file [52]. - It can also run as a **daemon or MCP server itself** if you want to integrate Codex into another system (advanced use; the docs mention `codex mcp-server` can expose Codex as a tool for other agents) [53]. - The **OpenAI API CLI** (like `openai api fine_tunes.create` etc.) is more for data tasks – not relevant to interactive coding, but if you ever needed to fine-tune or moderate content, that's something only the API can do.

CLI vs Extension: workflow differences: - Using CLI might slow down coding slightly because you'll copy changes from terminal to editor manually if you're not in an IDE that can apply them. The VSCode extension is more streamlined for editing. - But CLI might encourage a more linear, thought-out approach (since you're essentially doing all via text interface). - Some devs use the CLI inside Vim or Emacs, leveraging key bindings to send prompts – similar idea to VSCode extension but more DIY.

In summary: You do not *require* any CLI or API key for what we've covered – ChatGPT Plus covers it. The CLI is a useful optional addition, especially if you hit the limits of the extension or want to automate tasks. It's good to know it exists and one can switch to it or run it side-by-side with VSCode (they won't conflict; you could even have two sessions, though note they'd share the usage quota).

3. MSYS2 and Other Windows Tools

The user context mentioned MSYS2 from a Ruby installer as a possible tool. This isn't directly part of Codex or OpenAI's tools, but rather a Windows environment detail: - MSYS2 provides a Unix-like shell and utilities on Windows. If present, Codex might be able to use additional commands like `gcc`, `make`, or even Bash scripts that wouldn't otherwise be available in plain Windows. - If your project involves compiling C

extensions or using Makefiles (common in some Python projects for C libs), having MSYS2 could be necessary. Codex would then have to be instructed to use those appropriately (for example, to compile a C library, it might call `gcc` which MSYS2 supplies). - Ruby's MSYS2 might indicate you have some DevKit installed. Codex might attempt gem commands if it sees a Gemfile, etc. As long as those tools are in PATH, Codex can use them.

CLI Tools Summary: The extended ecosystem (CLI, config files, MCP, etc.) is there to enhance and automate your workflow, but for a developer primarily working in VSCode on Windows 10 with ChatGPT Plus, you can achieve a lot without venturing beyond the built-in extension features. Only consider the CLI or direct API when you either need to script AI actions (like auto-fixing code in CI), or you need to integrate with external systems in a way the extension cannot.

The next section on MCP servers will explore one of these advanced integrations more deeply.

Model Context Protocol (MCP) Servers: What, Why, and How

In the OpenAI Codex ecosystem, **Model Context Protocol (MCP)** servers are a way to give the AI access to external tools and data beyond your local environment [23]. Think of MCP as a plugin system: each MCP server exposes certain "tools" the model can call, like searching documentation, interacting with a browser, querying an API, etc.

Here's a comprehensive but practical rundown:

1. What is MCP and How Does It Differ from Built-in Tools?

By default, Codex has a limited set of built-in tools: - It can read/write files in the workspace. - It can run shell commands on your machine. - It can manage an internal "plan" tool (to reason steps out internally). - It can call out to the internet only in a very generic way (essentially through the shell, e.g., calling `curl` or an internal browser if allowed).

MCP extends this by letting you hook up **third-party or custom tools** in a standardized way [23]. An MCP *server* is simply a program (could be local or remote) that implements a certain protocol (MCP) to communicate with Codex. Codex (the client) can ask the MCP server to perform actions or fetch information, and then return the results to Codex for use in the conversation.

Examples of what MCP can do (supported features): - **Documentation lookup:** A server like Context7 can serve as a documentation tool. Codex can query it like, "Find the docs for Python's `sorted` function." The server returns the relevant doc text, and Codex can incorporate that into its answer [24]. - **Browser automation:** A Chrome DevTools MCP can allow Codex to open a webpage, click elements, or take screenshots [54]. For a developer, this could be used for end-to-end testing or scraping. - **Design tool integration:** The Figma MCP lets Codex query your design files (e.g., fetch colors or measurements from a UI design) [55]. - **Logs/Monitoring:** A Sentry MCP could allow Codex to search your error logs by issue ID or timeframe [56]. - **GitHub control:** The GitHub MCP, as mentioned, can allow Codex to create issues, comment, or modify PRs via the GitHub API [18]. Essentially, Codex could act on your GitHub account given the right permissions.

These are things outside the scope of local file edits and shell commands, hence require an MCP integration.

How it works internally: The model (Codex) will "decide" to use a tool when appropriate. For instance, if you ask, "What does the official NumPy documentation say about the `reshape` function?" Codex might recognize that it doesn't have that in context and that an MCP tool named `context7` is available. It will then formulate a request to that tool (MCP protocols define actions like `open`, `query`, etc.). The server executes (maybe by hitting ReadTheDocs API or local doc cache) and returns a response, which Codex then presents to you or uses to answer your question. This all happens behind the scenes in one seamless interaction, from your perspective.

Difference from built-in tools: Without MCP, Codex's knowledge is basically frozen to its training data (plus your code and whatever it can Google via command line). With MCP, it can get fresh, targeted info. Also, built-in tools (like shell) have a broad but blunt capability – e.g., Codex can try `pip install` or running a program, but it might not know the best way to fetch, say, "latest AWS SDK docs." With a dedicated MCP, it has a direct hook for that.

2. Setting Up and Using MCP in Our Context

We should clarify which MCP servers might be *useful for Python development on Windows 10*. Some likely candidates: - **Context7 (Dev docs)** – extremely useful if you're working with libraries and need quick reference. Instead of you googling, Codex can fetch the docs. For instance, if you forgot how to use Pandas pivot_table, Codex could call Context7 to get the Pandas docs for pivot_table and then apply it in code. - **GitHub MCP** – if you want Codex to manage PRs or issues (maybe automatically open a PR after making changes, or updating an issue status). - **Chrome/Playwright MCP** – if your development involves a web app and you want Codex to simulate a browser to test something. It could, for example, run an end-to-end test clicking through your app. This is advanced and requires those tools installed (Playwright MCP might need Node and Playwright). - **None** – It's worth noting: you don't *need* any MCP for standard usage. Many developers use Codex effectively without adding any. So don't feel obligated to set these up unless you have a clear use case.

Setting up an MCP (general steps): 1. **Install or have the tool available:** Some MCPs are just web endpoints (like Figma's or a remote API). Others you run locally. For example, Context7 can be launched via `npx @upstash/context7-mcp` [57] – meaning you need Node.js, and it will pull that package when invoked. 2. **Configure in `~/.codex/config.toml`:** You'll need to add an entry for the MCP server. The docs show how [58] [59]. For a simple STDIO server (one that runs locally as a child process), you put the command and args. E.g.:

```
[mcp_servers.context7]
command = "npx"
args = ["-y", "@upstash/context7-mcp"]
```

This tells Codex how to start the context7 server. If the server needs environment vars (like an API key for docs), you add those under `env`. For a web-based MCP (like Figma or GitHub which run as web services), you provide the URL and token in config [60] [61]. 3. **Enable any features needed:** The example config sets `rmcp_client = true` for using the Rust MCP client for HTTP/OAuth [62]. On Windows, enabling that

might be needed for some servers. It's a one-time flag. 4. **Restart Codex extension/CLI:** After editing config, you usually restart the extension or CLI so it picks up the new config. In VSCode, there's a convenient UI: you click the Codex settings gear and choose "MCP settings > Open config.toml" [50]. After saving changes, reload VSCode or toggle the extension. 5. **Using the MCP tools:** Now, Codex will know these tools by name. There's usually no additional step to "start" them; Codex will launch them on-demand. You can test if it's working by a direct prompt: *"Use context7 to find documentation for* `itertools.chain`*."* If configured right, Codex will respond with something like: *"According to the documentation, itertools.chain ..."* and possibly cite that it used the tool. The extension might show a brief message like "Launching context7 MCP...". Alternatively, in the CLI TUI, using the `/mcp` command shows connected servers [63], confirming it's running.

Windows-specific notes for MCP: - Some MCP servers (especially STDIO ones launched via a command) may have issues on Windows if they rely on Unix things. Many are Node-based which generally works cross-platform. Context7 and the GitHub MCP (which was run via Docker in the Medium article [64]) need some attention. The Medium guide had to tweak `startup_timeout_ms` and sandbox settings [35] because Windows sandbox by default blocks network. So, if you want Codex to use the internet (which these MCPs often do), you need to allow sandbox network access. Setting `sandbox_workspace_write.network_access = true` in config (as they did) grants the Codex process network access by default [35], so it won't prompt for every tool usage. - Docker-based MCP (GitHub): If you don't use WSL, ensure Docker is installed and working on Windows. The config in the article uses Docker to run the MCP server container [64]. That should work as long as Docker engine is running. The container will need to reach the internet (for GitHub API) – hence the network access and token. If Docker doesn't work in your environment (maybe you opted out of WSL and Hyper-V), you might skip the GitHub MCP or use an alternate approach (some have a direct node or python version). - Stability: Some users reported context7 MCP timeout on Windows [65]. If that happens, try increasing timeouts or running context7 outside Codex (like run the npx command manually to see if it works). Possibly the network sandbox or environment variables hindered it. Ensure the config's `rmcp_client` is enabled for HTTP servers – that can improve compatibility [66].

3. Benefits of MCP for Python Dev Workflows

Let's highlight the tangible benefits: - **Faster access to info:** Instead of leaving VSCode to search docs, you can ask Codex directly. E.g., "How do I use the `dataclass` field metadata attribute?" Without MCP, Codex might rely on its training (which could be outdated or incomplete). With an up-to-date docs MCP, it can fetch the latest Python docs or library docs, ensuring accuracy. It basically integrates StackOverflow/documentation lookup into your AI assistant. - **Extended actions:** With GitHub MCP, after Codex writes code and tests pass, you could literally say: *"Open a pull request with message 'Implemented feature X'."* Codex might draft a PR description (from earlier context) and then use the GitHub MCP to actually create the PR on your repository. This is like having a junior dev who not only writes the code but also files the PR for review. (Of course, you'd want to review it yourself too!) - **Environment management:** While not a classic MCP, one could conceive an MCP to interact with Conda or other environment tools. For instance, a custom MCP that lists Conda env packages or switches envs. Currently, not an out-of-box feature, but if needed, one could script it. - **Documentation generation:** Another angle – Codex could use tools to generate diagrams or docs from code. For example, if integrated with a graphviz MCP or plantUML MCP, you could ask "generate a UML diagram of the class structure" and it might output one via those tools. - **No extra OpenAI cost:** Using MCP doesn't cost extra tokens beyond what it takes for the model to describe the request and read the response. The heavy lifting is on the third-party service. Your Plus plan covers the model's part. Just

remember, if a third-party API (like Figma or GitHub) has its own cost or rate limit, that still applies (GitHub API is free within limits, Figma might have limits, etc., usually it's fine for moderate use). - **Security considerations:** Very important. By enabling MCP, you are effectively granting the AI a key to something. For example, giving it a GitHub PAT (Personal Access Token) means it could, in theory, do anything your token allows on your GitHub account. This is where the trust and mode settings come in: - You likely want to keep Agent mode (not full) so that if Codex attempts something big like deleting a repo via MCP (worst case), you'd get an approval prompt. The config `approval_policy = "never"` the Medium article used for GitHub MCP [35] was to avoid prompting for known benign actions, but use caution with that. Maybe set it to never only for specific tools but still monitor. - Only give it tokens with limited scope. E.g., a token that only can create issues on one repo, not full org admin rights. - Keep an eye on logs; the Codex CLI TUI can show what tool calls were made (like "Tools used: openai/github/create_pr" etc.). Transparency is key. - The Windows sandbox doesn't cover what the AI does via MCP because that can be remote (e.g., posting a comment on GitHub isn't something the local sandbox controls). So the responsibility is on the dev to configure safely. - **Stability issues:** Running multiple MCP servers and Codex plus everything can be a lot. If you find Codex crashing or freezing, consider disabling some MCPs or launching them only when needed. You can toggle `enabled = false` in config for a server to temporarily disable it [67].

Are MCP servers needed in our scenario? Not strictly. They are **nice-to-have enhancements**. This guide is mainly about what you can do *purely* with Plus and extension. Without MCP, you might occasionally copy a web doc yourself or handle the GitHub tasks manually. MCP just automates those. If you're comfortable, they can significantly streamline work (imagine never Alt-tabbing to browser for docs). If not comfortable, you can skip them and still do 90% of what we discussed.

MCP vs Tools vs Plugins: For completeness: - *MCP vs ChatGPT Plugins:* ChatGPT web has plugins like browsing or code interpreter. In VSCode Codex, analogous functionality is achieved via MCP and the Codex cloud. MCP is essentially the plugin system for Codex in IDE/CLI context. - *MCP vs built-in dev tools:* If you already have say a VSCode plugin that shows documentation on hover, you might not need context7 MCP. But the difference is Codex can *use* that info intelligently to solve tasks, not just display it to you.

4. Running Codex itself as MCP

This is advanced and mostly for those building larger AI systems, but to mention: - The docs show you can run Codex as an MCP server itself [53], meaning other AI agents could use Codex's capabilities via a standardized interface. Not directly relevant unless you are integrating Codex with another AI orchestrator. - It demonstrates how flexible the system is – you could chain AI assistants together (one calls another via MCP).

In closing, **MCP servers can greatly expand Codex's usefulness** for a Python developer by providing up-to-date knowledge and automating external tasks, all within the familiar chat-driven workflow. They require some setup and caution, especially on Windows, but can be worth the effort for power users.

Having covered all these angles, we'll now move to final sections providing concrete setup instructions, workflows, and troubleshooting guidance tailored to our scenario.

Detailed Setup Instructions

This section will serve as a step-by-step guide to get everything running for **ChatGPT Plus + VSCode Codex on Windows 10 with Conda**, as well as optional CLI/MCP components. It ensures you have a solid, reproducible environment to follow the workflows discussed.

1. VSCode Extension Installation & Configuration

Step 1: Install VSCode and the OpenAI Codex extension. - Make sure you have Visual Studio Code (latest stable) installed on Windows 10. - In VSCode, go to the Extensions panel (Ctrl+Shift+X), search for "**Codex – OpenAI's coding agent**" [68]. The publisher should be OpenAI. Click Install. - Alternatively, from a browser, visit the Visual Studio Marketplace link and click "Install", which should prompt VSCode to open and install it. - After installation, **restart VSCode** (this ensures the extension fully loads) [5].

Step 2: Sign in with ChatGPT Plus account. - Open the Codex panel in VSCode: you should see an OpenAI icon in the Activity bar (left side) – click it. It might also open automatically once after install. - You'll be prompted to **"Sign in with your ChatGPT account."** Click that. This will open a browser to authenticate (login with your OpenAI credentials, the same you use for ChatGPT web) [49]. - Authorize the VSCode extension when asked (it will say the extension wants certain permissions – this is normal). - Once done, the browser will show something like "Authentication successful, you can close this." Back in VSCode, the Codex panel should now be active and likely greet you or be ready for input.

Step 3: Verify extension settings. - By default, it will use GPT-5 Codex model if available (or GPT-4 Codex if that's current for Plus) [3]. You can see a model selector at the bottom of the Codex chat panel. It should show something like "GPT-5-Codex" as recommended. Keep it on the recommended model for best results. - The reasoning effort is adjustable (low, medium, high) [69]. Medium is a good start. High will think longer (for complex tasks) but also use more tokens. You can leave it at default for now. - Check the **Approval mode** selector (also under the chat input, as a dropdown or toggle). Ensure it's **Agent (Default)** for normal use [4]. We will switch to Chat for planning when needed, but default keeps it in Agent so it can act. - In Codex settings (click the gear icon in top-right of the Codex panel): - Review keyboard shortcuts if you like (you can set hotkeys to open Codex quickly, etc.) [70]. - Under **MCP settings**, you can see the config file link. We won't edit it yet unless setting up MCP now. But it's good to know where it is (usually at `%USERPROFILE%\.codex\config.toml` on Windows). - If using Windows without WSL, consider enabling the Windows sandbox network if you plan to use internet tools. This involves editing config.toml: set `sandbox_workspace_write.network_access = true` under a profile or globally [35]. This prevents being blocked when Codex tries any outbound network call (like pip install or contacting an MCP). Without it, you'd have to approve each time or it might timeout.

Step 4: (Optional) Use WSL for better compatibility. - Since we assume *no WSL*, skip this if sticking to pure Windows. But for completeness: The official advice is to open projects in a WSL2 Ubuntu for best results [5] [71]. If you ever run into too many Windows-specific issues, setting up WSL and using the extension in a WSL VSCode window is a backup plan. It's not required, but keep it in mind.

2. Conda Environment Configuration for Codex

Now ensure Codex will use your Conda Python environment:

Step 1: Install Miniconda/Anaconda if not already. - If you haven't, install Miniconda3 on Windows. During install, you can optionally check "Add to PATH" but it's usually not recommended to permanently add (to avoid conflicts). Codex doesn't strictly need it on PATH if we use absolute paths or conda commands. - Create your environment, e.g., `conda create -n myenv python=3.10` and install necessary packages (`conda install -n myenv flask pytest ...` whatever your project needs). - *No system Python:* If you have one, it's okay, just we won't use it.

Step 2: Decide activation strategy. - Easiest: from an Anaconda Prompt, launch VSCode for your project. This will ensure any integrated terminal in VSCode starts with the environment activated. For example, open "Anaconda Prompt (myenv)" then in it run `code .` to open VSCode in that environment. - Alternatively, configure VSCode's terminal profiles: In `settings.json`, you can create a profile like:

```
"terminal.integrated.profiles.windows": {
    "Conda MyEnv": {
        "path": "C:\\Windows\\System32\\cmd.exe",
        "args": ["/K", "C:\\Miniconda3\\Scripts\\activate.bat myenv"]
    }
}
```

Then in VSCode terminal dropdown, choose "Conda MyEnv". This will open a terminal with `myenv` activated. - The advantage is, if Codex uses the *same terminal session*, it will already be in the env. However, Codex often spawns processes directly, not through the open terminal.

Step 3: Inform Codex of environment. - In your first message to Codex (or better, in AGENTS.md file in repo), specify: "**Python interpreter:** use the Conda environment `myenv` located at XYZ path. Always run Python or pip within this environment." This sets the context. - If you have a `.venv` folder or something, Codex might detect it, but with Conda, it's less auto-detectable. Being explicit helps.

Step 4: Test a simple command via Codex. - For example, ask Codex: *"What is the Python version?"* In Agent mode, it should run `python --version`. If it returns the correct version (3.x from your env) that means it likely picked up your env's Python. If it says "'python' not found" or a wrong version, then we know it's not using the env. - If not correct, you can direct it: *"Run* `conda run -n myenv python --version`*."* This should show correct version. Then you might continue to use `conda run -n myenv` prefix in future commands (maybe Codex will infer to do that for subsequent runs in this session).

Step 5: Lock in interpreter for VSCode's Python extension (optional). - If you also have the MS Python extension, click the Python version in the status bar and select `myenv` interpreter. This writes to `.vscode/settings.json` something like `python.defaultInterpreterPath`. While Codex doesn't directly read that setting to choose an interpreter, it's documentation for you and others. You could mention to Codex: "The interpreter path is set to ... in settings.json" – it might then use that knowledge.

3. Local Toolchain Integration

Beyond Python, ensure your system has the typical tools Codex might call: - **Build tools:** If compiling is needed (e.g., you use a library that needs Visual C++), install the **Build Tools for Visual Studio 2022** (as

recommended in docs [45]) and the latest **VC++ Redistributable** [45]. This prevents mysterious errors when Codex tries to pip install something that compiles native code. - **Git:** Codex might use Git if instructed (like `git diff` or `git init`). Install Git for Windows and ensure it's in PATH. Then it can run those commands. It won't do anything unless you tell it, but it's good to have. - **MSYS2 (optional):** If you have MSYS2 from Ruby or separate, and it's in PATH, then Codex could leverage Unix commands. Up to you. If you see Codex struggling with a command not available, you can either install a Windows equivalent or quickly instruct it differently. For instance, if it tries `grep`, you can install GnuWin32 grep or just correct the command to use PowerShell's `Select-String`. - **Node.js:** Needed if you plan to use certain MCPs or CLI. Already required for the extension/CLI themselves (the extension actually bundles Node runtime or uses VSCode's, so you may not see it). But for MCP servers via npx or the CLI tool, having Node 16+ is recommended. The Windows guide suggests using nvm in WSL for Node [72], but on Windows you can just install the latest Node LTS. - **Docker:** If using the GitHub MCP or other container-based tools, install Docker Desktop and ensure it's running. You might need to enable Hyper-V or WSL2 backend. - **VSCode settings for Codex (optional):** There might be some extension settings accessible via JSON: - E.g., `openai.codex.approvalMode` if you want to change default mode. - `openai.codex.maxFiles` or others if available, but not sure if exposed. - Check VSCode's `settings.json` after using Codex to see if any settings appeared.

At this point, you should have: - Codex extension logged in and ready. - Your project folder open in VSCode. - Conda env prepared and ideally active for any shell usage. - All necessary compilers and tools installed to avoid environment errors.

4. Optional: Setting up Codex CLI

If you want to use the CLI or have it as a backup:

Step 1: `npm install -g @openai/codex` – open a Command Prompt or PowerShell (with Node.js in PATH) and run this. It will fetch the CLI package. - If errors (like Python not found for building something or MSBuild needed), ensure build tools are installed as above. Also, sometimes Node-gyp requires setting up some environment; hopefully the CLI has prebuilt binaries to minimize this.

Step 2: Once installed, run `codex` in a project directory. - It should either ask to log in (open browser, similar to extension flow) or if your extension already authenticated, the CLI might pick up a cached token. If not, log in when prompted. - You'll see a TUI (text UI) interface. Try a simple prompt like "hello" to make sure it responds. Then exit (usually Ctrl+C or type a command like `/exit` if provided).

Step 3: Use CLI config if needed: - The CLI uses the same `~/.codex/config.toml`. You can edit that for things like default model or enabling rmcp. - For example, if you want CLI by default to always use full access (not advisable generally), you could set `approval_policy = "never"` under a profile. - Or if you want it to default to chat vs agent, etc., those can be set too.

Step 4: Update PATH (optional): - The `codex` command should be in your npm global bin (likely `%AppData%\npm` for Node on Windows). If `codex` isn't recognized, add that to your PATH. But typically the installer does it. - Then you can run `codex` from any directory easily.

Now you have the CLI ready for scenarios when VSCode is not available or for integration tasks.

5. Optional: MCP Servers Setup

Only pursue this if you want those advanced integrations: - We'll illustrate one: **Context7** (documentation MCP) because it's straightforward and doesn't need API keys. - And brief mention of **GitHub** (requires token + Docker).

Setting up Context7 MCP: 1. Ensure Node.js is installed (since it uses npx). 2. Open `~/.codex/config.toml` (via VSCode Codex settings or manually). 3. Under `[mcp_servers]`, add:

```
[mcp_servers.context7]
command = "npx"
args = ["-y", "@upstash/context7-mcp"]
```

Optionally, if you want to limit it to certain docs or add API keys for premium doc sources, refer to Context7 docs – but by default it covers a wide range of docs without config. 4. Also in `config.toml`, to be safe, enable the new MCP client:

```
[features]
rmcp_client = true
```

(If a `[features]` section exists, add under it; otherwise create one at bottom.) 5. Save and restart VSCode (or if in CLI, restart CLI). 6. Test: In Codex chat, try: *"Find documentation for Python's* `sorted` *function."* The first time, it may take a few seconds to launch the server (it downloads some data via npx). Then Codex should output an explanation possibly quoting the official docs. If it fails or times out, check the OUTPUT panel for logs or run the CLI with `/mcp` to see if context7 is listed. 7. If it doesn't work, try running `npx -y @upstash/context7-mcp` in a separate terminal to see if that launches (it should print something like "MCP server ready on port ..."). If it fails, there might be an issue with Node or network.

Setting up GitHub MCP: 1. Ensure Docker is installed and running. 2. Get a GitHub Personal Access Token: go to GitHub > Settings > Developer settings > Tokens (classic). Generate one with scope to create issues/pull requests (the Medium guide likely used repo scope) [73] . 3. In `config.toml`, add:

```
[mcp_servers.github]
command = "docker"
args = ["run", "-i", "--rm", "-e", "GITHUB_PERSONAL_ACCESS_TOKEN", "ghcr.io/
github/github-mcp-server"]
env = { GITHUB_PERSONAL_ACCESS_TOKEN = "<your_token_here>" }
startup_timeout_sec = 20
```

And also ensure `network_access = true` in sandbox as mentioned [35] . Possibly set `approval_policy = "never"` for this server or run Codex in full access when using it, to avoid it asking every time it tries to hit GitHub. 4. Save and restart. 5. Test: *"Create a new issue in my repo user/testrepo with title 'Test Issue'."* Codex should either ask which repo if not clear, or attempt to use the MCP to create it. It will require that your local git or config knows which repo or you explicitly tell it (the MCP might allow specifying

owner/repo in prompt). 6. Alternatively, test a read: *"List open issues in user/testrepo."* If configured, Codex will call the MCP and list them. 7. If it doesn't work, check Docker: maybe it needed WSL2. Also ensure your token is valid. 8. Use carefully – maybe try in a throwaway repo first to see what it does.

Other MCPs: Setting up others like Figma would involve providing OAuth tokens etc. Only do these if you need them. The pattern is similar: config entry with either `command` or `url` and tokens.

6. Verification

At the end of setup, do a quick full integration test: - Ask Codex a coding question that requires reading a file: e.g., "What does function `foo` in `utils.py` do?" – expecting it to open that file and summarize. - Ask it to run a snippet: e.g., "Run `print('hello world')`" – expecting it to either just print or actually execute (it might just simulate output for such a trivial command, as it doesn't need to spawn a process for a pure Python print; but if you say run a file, it should). - Try a multi-step simple task: "Create a file `demo.py` that prints the Python version, run it, and show me the output." This touches file creation and execution.

If all those work without errors (and using the correct Python), you are good to proceed with real tasks.

Now, let's proceed to some example workflows, prompting patterns, and troubleshooting tips, now that the environment is ready.

Annotated Example Workflows

We will walk through a few realistic scenarios to illustrate how to effectively use Codex in our setup. Each example will show the conversation flow (with user and Codex interactions) along with explanations of what's happening.

Workflow 1: Multi-step Agentic Refactor

Scenario: You have a legacy module `data_processing.py` with a few functions doing similar tasks. You want to refactor it into cleaner, separate functions and add a new one, using Codex's help.

Initial state: `data_processing.py` exists, containing spaghetti code. Tests cover it, but code is messy.

Steps:

1. **Discuss and Plan (Chat Mode):**

2. **User (you):** Switch Codex to *Chat* mode (to avoid it jumping into code edits immediately). Ask: *"Our `data_processing.py` module is too monolithic. I want to refactor it. Currently, it has functions `load_data`, `clean_data`, `process_data` all in one file. Plan how to refactor: maybe split into multiple files or at least reorganize, and identify improvements."*

3. **Codex (Chat mode):** It responds with an analysis: *"It looks like `load_data` and `clean_data` deal with raw data and could form a `data_ingest` module, whereas `process_data` is more analysis –*

could be separate. Plan: 1) Create `ingestion.py` *for load/clean functions, 2) Keep* `process_data` *in* `processing.py`*, 3) Update import references, 4) Ensure tests still pass, 5) Possibly improve error handling in clean_data (noticed a TODO there).”* [74] [75]

4. (This plan is hypothetical, but Codex often comes up with reasonable suggestions.)
5. **User:** *“Sounds good. Anything else?”*
6. **Codex:** *“Maybe also add docstrings to each function if missing, to improve maintainability.”*

7. You approve this plan or tweak it as needed.

8. **Execute Refactor (Agent Mode):**

9. Switch to *Agent* mode now to let Codex act.

10. **User:** *“Proceed with the refactor as discussed: create* `ingestion.py`*, move load_data & clean_data there with docstrings, create* `processing.py` *for process_data, update references in any other modules accordingly.”*
11. **Codex (Agent mode):**
    - It will likely say: *“Creating* `ingestion.py` *…”* and show the content with `load_data`, `clean_data` moved (and possibly improved, like adding docstrings it promised).
    - *“Updating* `data_processing.py` *…”* – maybe it will gut this file or turn it into an import wrapper, or remove it if you implied splitting fully.
    - *“Creating* `processing.py` *…”* with `process_data` function.
    - *“Updating imports in other files…”* If `main.py` or tests import `from data_processing import load_data`, it will find those and adjust to `from ingestion import load_data`, etc. (It might say which files it edited.)
    - All these edits appear as diffs in VSCode for you to review.
12. **User:** Review the changes. You notice maybe in `ingestion.py`, it forgot to import a library that was implicitly imported in old `data_processing.py`. For example, `pandas as pd` was at top of old file, but in splitting, `processing.py` might need that too but Codex didn’t add it.
13. **User:** *“Add the missing imports: ensure both new modules have the needed imports (like pandas). Also, add a docstring for* `process_data` *too.”*
14. **Codex:** It updates those files accordingly (in Agent mode, it directly applies diff).

15. Now the refactor code is ready in your workspace.

16. **Run Tests:**

17. **User:** *“Run all tests.”*

18. **Codex:** Executes `pytest`. Suppose it outputs: 2 failed tests.
    - One failure: a test for `load_data` fails because it was importing from old module name.
    - Another failure: in `process_data`, it had a reference to a function that moved to ingestion, but Codex didn’t update that internal call.
19. Codex will likely spot these: *“Failure: ImportError, no module data_processing – fix: update test to import from ingestion. Failure: NameError in process_data – fix: add* `from ingestion import clean_data` *in processing.py or adjust call.”*
20. **User:** *“Fix the issues causing test failures.”*

21. **Codex:**
    - Opens the test file, changes `import data_processing` to `import ingestion`.
    - Opens `processing.py`, adds `from ingestion import clean_data` at top (or changes the call `clean_data()` to `ingestion.clean_data()` depending on how it was using it).
22. **Codex:** Re-runs tests (likely automatically after fix). Now tests pass.
23. **User:** *"Great. Show me a summary of changes."*
24. **Codex:** Summarizes: *"Refactored code by splitting* `data_processing.py` *into* `ingestion.py` *and* `processing.py`*. Updated imports and added docstrings. All tests are passing now."*
25. At this point, you have successfully orchestrated a multi-file refactor using Codex.

Notes: You oversaw the plan, reviewed each change (with Codex providing rationale in the chat for each file edit), and Codex even ran tests to ensure nothing broke, fixing issues it found [34]. This workflow saved you from doing the mechanical parts of moving code and updating imports, letting you focus on whether the refactor logically made sense.

Workflow 2: Generate Architecture from Design Doc

Scenario: You have a new feature to implement. A product spec (design doc) describes the feature in detail, including data structures and components needed. We'll have Codex create the skeleton of this feature's codebase from that document.

Initial state: No code for this feature exists yet. You have a design text (could be in a file or an external doc).

Steps:

1. **Provide the design spec to Codex:**
2. **User:** Copy relevant parts of the design doc (or if it's in a local markdown file `FEATURE_X.md`, open it and copy text). Then say: *"Here is the design for Feature X:"* and paste it in.
3. This might be large, but since we're in a deep research context, assume it's within manageable size or we paste sections sequentially.
4. **Codex:** Possibly it summarizes or says "Got it." If it doesn't respond fully (maybe waiting for question), you follow up with specific ask.
5. **Ask for code structure outline:**
6. **User:** *"According to that design, what modules and classes should we create?"*
7. **Codex:** *"Based on the spec, we should have:* `models.py` *with classes A, B;* `service.py` *with logic to do Y;* `api.py` *to expose endpoints (if applicable); maybe a* `utils.py` *for Z."* It basically drafts an outline.
8. **Approve and request generation:**

9. **User:** *"Great. Please create those modules with the basic class and function definitions (methods empty or with pass), including docstrings from the design specs. Also include any TODO comments for parts that need further work as per spec."*
10. (This prompt tells Codex to scaffold code without implementing details fully, which can be useful if you want just the architecture first.)

11. **Codex (Agent mode):**

    - Creates `models.py`: defines classes A, B with attributes from spec, docstrings describing them (perhaps pulling descriptions from spec).
    - Creates `service.py`: functions or classes implementing core logic (maybe stubs with TODO where actual logic like "algorithm to compute X" goes).
    - Creates `api.py`: maybe a FastAPI or Flask blueprint if the design said we need an API interface, with endpoints definitions (again possibly with `pass` or simple return).
    - Possibly other files the design mentioned (configuration, etc.).
    - It will show each file's content in the chat. The files also appear in your workspace.

12. **Review structure:**

13. **User:** Check the files. The names and content should reflect the spec. If something is off (like it misunderstood the design or missed an object), point it out.
14. E.g., "The design mentions a `Calculator` class but I don't see it."

15. **Codex:** *"Oh yes, add Calculator in models.py or a new file."* It fixes accordingly.

16. **Confirm and proceed to detail implementation (if desired):**

17. At this point, you have a blueprint. You might fill in some critical logic yourself or continue with Codex. Let's say you want Codex to implement one of the methods fully that the design describes in detail.
18. **User:** *"Implement the method* `calculate_score` *in* `service.py` *according to the algorithm described (you can find it in the spec above)."*

19. **Codex:** It scrolls up to find that algorithm description from the spec we gave (the text is in conversation memory) and implements the method accordingly. If the spec was clear, it likely gets it right or close. It might even add references or comments citing the spec.

20. **Testing the scaffold (optional):**

21. You could ask Codex to generate tests for these new modules based on spec. *"Create a pytest test file* `test_service.py` *with some basic tests for service functions, based on expected behaviors from the spec."* Codex will do that.
22. Run tests – they might mostly pass if functions are stubs (maybe all you can test is that they raise NotImplemented or return placeholder values). This step ensures the scaffold is at least syntactically correct and hooked up (if any import was wrong, test would fail to import).

This workflow shows using Codex as a **code generator given a high-level design**. It took what could be hours of writing boilerplate and did it in seconds, while you ensure it aligns with the spec. You basically communicated the design once (by copy-paste) rather than retyping class names and comments.

Workflow 3: Implement Code from TODO Comments

Scenario: Over time, your code accumulated TODO comments. Now you want to address them. Codex can be tasked to resolve these systematically.

Initial state: Various .py files contain lines like `# TODO: optimize this` or `# FIXME: handle edge case for X`.

Steps:

1. **Identify all TODOs:**
2. **User:** *"Scan the repository for 'TODO' or 'FIXME' comments and list them along with the file and line number."*
3. **Codex (Agent mode likely):** It might attempt something like `grep -R "TODO" .` under the hood. On Windows, grep might not exist; it could use PowerShell Select-String. If configured with MSYS grep, it'll work. Let's assume it finds some.
4. **Codex:** Responds with a list, e.g.:
    - `utils.py:45 – "# TODO: improve performance of sorting"`
    - `database.py:10 – "# FIXME: handle null values"`
    - `main.py:88 – "# TODO: remove this hack after implementing feature X"`
5. If Codex doesn't give such a list (maybe it says it can't directly search), you might manually search and feed it the list. But Codex often can do a basic search.

6. **Go through each TODO:**

7. **User:** *"Let's address each TODO. Start with utils.py line 45: It's about sorting performance. The code currently uses bubble sort, which is slow. Replace it with Python's built-in sorted or a better algorithm."*
8. **Codex (Agent):** Opens `utils.py`, finds that function, replaces bubble sort with something like `data.sort()` or `sorted(data)` (which uses Timsort, much faster). Removes the TODO comment or marks it done. Shows the diff.
9. It might even add a comment like "# Optimized: replaced bubble sort with built-in sort".
10. **User:** Approve that change (looks correct).
11. **User:** *"Now database.py line 10, null handling."*
12. **Codex:** Opens `database.py`, sees a function maybe `process_record(record)` with a TODO about nulls. Implements a check like `if record is None: return ...` or appropriate handling per context. Removes the TODO.
13. **User:** Check if that matches expectation (or spec if any). Looks good, approve.
14. **User:** *"main.py line 88 – remove hack and implement feature X properly."*
15. This one is trickier because it references an unimplemented feature X. Suppose by now feature X is implemented (maybe by you or in design). You might need to tell Codex what to do instead of hack.
16. **User:** *"Feature X is now implemented in module new_feature.py. In main.py line 88, call the real function from new_feature instead of the hack."*

17. **Codex:** Edits main.py, removes hack (maybe a hardcoded value or workaround) and calls `new_feature.do_real_thing()` or whatever. Adjusts imports if needed.
18. Removes the TODO comment.

19. **User:** Verify the change. Yes, that integrates the new feature properly.

20. **Test after TODO fixes:**

21. Run tests again to ensure nothing broke. If any test was specifically expecting the old behavior (maybe the hack was making a test pass), it might fail now. Codex can then update the test or adapt behavior accordingly.
22. For instance, if main.py hack was returning a static value that tests expected, but now it calls a function that returns dynamic value, you might update the test to not expect the static output.
23. Codex can help: "Update tests to expect new behavior of feature X in main." It will find those tests and change expected values or logic.

After this, all those TODO comments are resolved. This workflow demonstrates Codex as a maintenance tool – it can navigate to the exact points in code that need changes (driven by comments), apply targeted improvements, and run verifications, all while you supervise context like algorithms to use or linking to other modules.

Workflow 4: Run Local Tests and Fix Failures

We've implicitly done this in earlier workflows, but let's illustrate a dedicated bug-fixing session using failing tests as the guide.

Scenario: You run your test suite and a few tests fail. You want to use Codex to analyze and fix the issues.

Initial state: Some tests failing, you have their error trace or messages.

Steps:

1. **Run tests via Codex:**
2. **User:** *"Run* `pytest`*."*
3. **Codex:** Executes tests, collects failures. Suppose it shows:
    - `test_api.py::test_get_user_data FAILED – AssertionError: expected 'age' field in response`
    - `test_calc.py::test_divide_by_zero FAILED – Exception not raised`
4. Codex likely prints the stack trace or at least the reason given by pytest (maybe "KeyError: 'age' in function get_user_data" and "No Exception raised for divide by zero").

5. It may start analyzing right away.

6. **Analyze failures and propose fixes:**

7. **Codex (after test run):** It might say:
    - For test_get_user_data: *"The test expected an 'age' field in the API response, but it's missing. Likely the API endpoint omits that field. Fix: include age in the response JSON."*
    - For test_divide_by_zero: *"The test expects a ZeroDivisionError when dividing by zero, but current code returns inf. Fix: explicitly raise an error or handle the case to pass the test."*
8. If Codex doesn't automatically analyze, you ask: *"Why did those tests fail?"* It will then analyze similarly.
9. **Apply fixes:**
10. **User:** *"Implement the fixes for both failures."*
11. **Codex (Agent):**
    - Opens `api.py` or wherever `get_user_data` is implemented. Adds the missing `'age': user.age` in the return dictionary, for example.
    - Opens `calc.py` or wherever divide is implemented. Perhaps currently it returns `float('inf')` on division by zero. It changes it to raise a `ZeroDivisionError` or handle differently to satisfy test (depending on spec).
    - Shows these changes.
12. **User:** Approve if they align with expected behavior. (If, say, the design was to return inf instead of error, maybe the test is wrong. But assume the test is the requirement here.)
13. **Re-run tests:**
14. **Codex:** It might automatically re-run pytest after making changes (it often does to confirm).
15. Now hopefully all tests pass: it prints summary "2 passed, ...".
16. **Codex:** *"All tests passed."*
17. **Double-check and refine:**
18. If you have additional quality concerns (maybe raising ZeroDivisionError is fine for test, but you'd rather handle it gracefully), you can instruct a refinement. But from strictly getting tests green perspective, we're done.
19. **User:** *"Great, tests are green. Provide a brief summary of what was fixed."*
20. **Codex:** *"Added missing 'age' field to API output in get_user_data, and adjusted divide function to raise exception on divide-by-zero as expected by tests."*

This showcases Codex as an automated debugger and fixer. It took the feedback loop of "fail test -> find cause -> fix code -> rerun" and handled it quickly [34]. This is extremely useful for regression fixing; the AI can parse traces often faster than a human by jumping directly to offending lines.

Workflow 5: Iterative Plan/Approval Coding

This is more of a pattern than a scenario, but let's illustrate an interactive session where you approve steps one by one.

Scenario: Building a new feature incrementally with constant oversight.

Steps:

1. **Start with high-level instruction:**
2. **User:** *"We need to add a caching layer to function* `expensive_computation(x)`*. Let's do this step by step. Codex, propose the first step."*
3. (Agent or Chat mode? Could do Chat to be safe.)
4. **Codex (Chat):** *"Step 1: Identify where to store the cache (maybe a global dict or lru_cache). Step 2: Modify expensive_computation to check cache. Step 3: Add logic to populate cache after computing. Step 4: Add tests for caching."*
5. **User:** *"Good. Implement step 1."* (We mean: create a cache storage.)
6. Switch to Agent mode if not already for implementation.
7. **Codex (Agent):** It adds, say, `cache = {}` at top of module or uses `functools.lru_cache`. Suppose we intended a simple dict.
8. Shows code where the cache dict is defined.
9. **User:** *"OK, I approve step 1."*
10. **User:** *"Proceed to step 2: modify the function to use the cache."*
11. **Codex:** Edits `expensive_computation`:

    ```
    def expensive_computation(x):
        if x in cache:
            return cache[x]
        # original computation...
        result = ...
        # (we will cache it later in step 3)
        return result
    ```

    Leaves a comment or note for adding caching.
12. **User:** Review, looks right.
13. **User:** *"Step 2 done, now step 3: populate the cache after computing."*
14. **Codex:** Edits function:

    ```
        result = ...
        cache[x] = result  # store in cache
        return result
    ```

15. **User:** Great.
16. **User:** *"Step 4: write tests."*
17. **Codex:** Creates maybe a test in `test_cache.py` ensuring calling twice returns same result without recomputation (it might simulate by counting calls).
18. **User:** Check the test logic. Perhaps adjust if needed.
19. **User:** *"Run tests."*
20. **Codex:** Runs tests, hopefully they pass (if we had earlier tests, ensure none broke).
21. If pass, feature done.

Through this iterative approach, you controlled each part of the process. Codex basically acted like a pair programmer that waits for your go-ahead at each step, ensuring transparency. This is useful for delicate

changes where you want to verify intermediate states (maybe you ran tests after step 2 to ensure you didn't break original functionality before caching, etc.).

These workflows illustrate how to engage with Codex effectively: - Communicate your goals clearly (with context like design docs or tests). - Choose the appropriate mode (Chat for discussion, Agent for doing). - Intervene to correct or guide when needed. - Use Codex to verify by running tests or analysis after changes. - Approve changes or ask for modifications iteratively.

Now, having seen these examples, we'll move on to discussing prompting patterns and templates you can use to streamline such interactions.

Prompting Patterns and Templates

Writing effective prompts for Codex is key to getting the results you want. Here are some patterns, templates, and tips tailored to this agentic coding context:

1. Plan-First Prompts

When you want Codex to think before doing, or to reveal its approach:

- **Template:** *"Before writing any code, provide a step-by-step plan to accomplish X."*
- **Example:** "Our goal is to add logging to all functions in `utils.py`. **Before coding, outline how you will do this** (which functions to modify, what logging library or messages to use)."
  This yields a plan you can vet. [76]

- **Follow-up:** If the plan is too vague, ask for detail: "Can you break step 3 into more detail?" or "What alternative approaches did you consider for caching?"
  This forces the AI to elaborate its reasoning. [74]

- **Rationale prompting:** You can explicitly say *"Explain your reasoning as you go."* In Agent mode it might do something like:

  ```
  # Plan:
  1. ...
  2. ...
  # Now implementing...
  ```

  or in Chat mode just list reasoning. Use this when you want transparency, though it can be verbose.

2. File-Specific Prompts

To make Codex focus on certain files or code:

- **Template:** *"In* `filename.py`*, [do X]."* or *"Open* `filename.py` *and explain/fix/improve ..."*
- **Example:** "**In** `database.py`, refactor the `connect()` function to use a context manager."
  This ensures Codex works on that file. [2]
- **Using** `@filename` **references:** As documented, you can say: *"Use* `@models.py` *and* `@views.py` *as references to add a new field."* [2]. This tags those files into context explicitly, useful if it didn't automatically consider them.
- **Line-specific guidance:** "In `utils.py`, at line 50, there's a bug. Fix it." If you know line numbers (from an error), this directs Codex precisely. It will likely search around that line. Or: "In `module.py`, there's a TODO on line 120 about error handling. Resolve that."
  This ties in with the TODO workflow above.

3. Style and Convention Prompts

Codex is trainable via prompt on how to format things:

- **Template:** *"Follow [style/convention] for [task]."*
- **Examples:**
- "Generate a commit message in Conventional Commits format for the above changes."
  It will produce something like `feat(module): ...`.
- "When writing docstrings, use Google style."
  It will then format docstrings accordingly.
- "Format the code with black (PEP8) style if not already."
  It might auto-format or at least not introduce styling issues.
- **Custom instructions:** If you have an `AGENTS.md` with style rules (e.g., "All functions must have type hints and raise ValueError on bad input."), Codex will internalize that and do it by default [6] [7]. But you can also mention them in prompt for immediate effect: "Make sure to add type hints to new functions."

4. Testing-Driven Prompts

Leverage tests in prompts:

- **Template:** *"Write tests for X before implementing"* or *"Given this failing test, fix the code to make it pass."*
- **Example:** "Here's a test (paste test code). **Implement the functionality so this test passes**."
  Codex will write code fulfilling the test's expectations. This is great for TDD: you supply the test, it supplies the code.

- **Failure context:** "The following test is failing with error Y (paste error). **Fix the code**."
  We saw this usage; Codex identifies cause from stack trace.

- **Test generation:** "Write a pytest test for function `foo` that covers edge cases."
  It will generate a new test function. You can specify edge cases or leave it open – Codex might invent some based on its understanding (which can be impressively thorough).

5. Documentation and Explanation Prompts

To get Codex to explain code or decisions:

- **Template:** *"Explain what the following code does"* or *"Add comments explaining the logic in function X."*
- **Example:** "**Explain** the algorithm in `sort_data()` in simple terms."
  It will produce a short explanation, which can be used for documentation or just your understanding.

- **Architecture summary:** "Summarize the architecture of this project." If you have `AGENTS.md` or multiple files open, Codex might give a coherent summary. If too broad, ask specific: "What are the responsibilities of each module in this project?" *Good for knowledge transfer or generating documentation.*

- **Decision reasoning:** After a fix or suggestion, you can ask "Why did you choose to do it that way?" *Codex will articulate its reasoning – helpful if you want to double-check logic.* Usually, it might cite performance or clarity, sometimes even referencing known best practices [4] .

6. Multi-step Execution Prompts

Encourage Codex to do multiple actions in one go (when you trust it or for automation):

- **Template:** *"Do X, then Y, then Z. If any step fails, stop and report."*
- **Example:** "**Open** `data.csv`, **generate** Python code to parse it into a JSON file, **save** the code to `parser.py`, **run** `parser.py` on `data.csv`, and **show** the first 5 lines of output."
  This prompt asks for a lot – Codex will likely clarify steps, then perform them. It might need full access mode for reading an arbitrary file or you provide file content.

This pattern is powerful for automation but requires Full Access ideally to avoid multiple approvals (reading a file and running code might prompt separate). Use caution – maybe test such multi-step with a harmless task to see how Codex sequences actions.

- **Conditional logic in prompt:** "If tests fail, revert changes." Codex can't truly revert Git easily (unless you let it run git commands), but it might remove code it just wrote by itself. It's not guaranteed. But you can instruct it to behave conditionally: it will attempt to follow that logic.

7. Correction and Refinement Prompts

Sometimes Codex's first output isn't right. Guide it with follow-ups:

- **Direct corrections:** “No, use a dictionary comprehension instead of a loop.” or “The output should be sorted, please adjust.”
  Codex will incorporate that into the code. It’s usually very responsive to such specific edits.

- **Ask for alternatives:** “Show me an alternative implementation using recursion.” or “What’s a simpler way to do this?”
  It might provide another approach or a simpler code block. Useful to compare solutions.

- **Partial acceptance:** “Keep the changes to `models.py`, but revert the changes in `controller.py`.”
  Codex can selectively undo or modify. It may require it to recall what it did (in conversation memory it has old code and new code). This is tricky, sometimes easier to just do git stash yourself. But you can try – it might open `controller.py` and restore from before (if it still has the old content in context).

8. Safety and Stop Prompts

If Codex goes off track or tries something dangerous:

- **“Stop” command:** Just type "stop" or “That’s not right, do nothing further.” It likely will halt. Or simply don't approve the action – in Agent mode, if it’s waiting for approval to run a command, you can just cancel it (there’s usually a cancel button).

- **Reset context:** If conversation got confusing, you can use *“Let's start over on this topic.”* Or actually click the "clear conversation" in the extension. Then provide a fresh prompt with necessary context. The extension doesn’t allow multiple chats concurrently yet, so you either have to steer it or reset.

- **Asserting constraints:** “Don’t modify any file other than X.” or “Do not use internet.” You can instruct these if you worry it might (though in default mode it won’t use internet without asking anyway [34]). But if you had Full Access on and want to limit, a prompt reminder helps.

Finally, remember you can always break a task into smaller prompts if output is too big or if the model starts to lose coherence. E.g., rather than “write 10 functions”, do 2-3 at a time.

These patterns should give you a flexible toolkit for interacting with Codex effectively. Next, we’ll address how to structure projects to play nicely with AI assistance.

Recommended Project Layouts for Agentic Coding

The way you structure your project can significantly influence Codex’s effectiveness. Here are best practices to organize code and resources to maximize AI assistance:

1. Use Clear and Modular Structure

Codex (especially GPT-4/5 based) does well with well-named, single-purpose modules: - **One module, one purpose:** Aim for each file to handle a distinct part of the system (e.g., `database.py` for DB access, `auth.py` for authentication functions, etc.). Codex can infer context from filenames and is less likely to confuse functionalities. - **Package by feature:** Consider grouping related modules in a package. E.g., `user/models.py`, `user/views.py` for user feature. That way, when you instruct Codex about user-related tasks, it likely knows to work in that package. It also can use relative imports properly. - **Avoid monster files:** If you have a 5k-line single file with many classes, Codex might struggle to open or understand it fully due to context limits. If possible, break it into smaller files. If not, at least sections with clear comments (so Codex can open just the relevant part if needed).

2. Naming Conventions and Consistency

- **Consistent naming:** Use descriptive names for functions and variables. Codex will utilize these clues. If every data access function is prefixed with `fetch_`, it will likely follow that when adding a new one (or at least not introduce a wildly different naming).
- **Follow common patterns:** If using frameworks, stick to typical names (e.g., in Django, a model class in models.py named `MyModel`, view functions in views.py). The AI has knowledge of common patterns and will adhere to them, making its suggestions more accurate.
- **Capitalization and casing:** Keep it standard (snake_case for functions, PascalCase for classes in Python). The model usually does this by default, but if you had a non-standard style, mention it in AGENTS.md so Codex is aware.

3. Document Key Design in AGENTS.md or README

- **AGENTS.md at root:** As discussed, Codex will read `AGENTS.md` in your project root and possibly in subdirectories [6] [7]. Use it to convey:
- Overall architecture (what are the main components/modules).
- Key decisions or workarounds ("We use our own JSON serializer instead of stdlib for X reason").
- Coding guidelines ("All functions must handle None inputs gracefully", "Use list comprehensions where possible", etc.).
- Project-specific terms or abbreviations.
- **Project README:** If you already have a detailed README or developer guide, Codex might consider it if it finds it. You can also treat it as context by referencing it in prompt: *"Refer to the project README for how to configure the environment."*
- **Inline comments for logic:** Complex algorithms should have comments explaining them. If you ask Codex to modify a complex function, it will read those comments and try to maintain the logic, which is good. If there are no comments and only code, it might risk altering behavior wrongly. So for sections that are delicate or not obvious, comment them – it helps both humans and AI.

4. Provide Example Usage

Codex learns from examples. If your repo has usage examples (like a `examples/` directory or tests), the AI can mimic those patterns: - E.g., if every test uses a certain fixture, Codex will use it in new tests too. - If you have a sample script showing how modules interact, Codex will follow that in its suggestions.

5. Keep Dependencies and Env Info in the Repo

Given our scenario with Conda: - Include an `environment.yml` or `requirements.txt` in the repo. Codex will look at those to know what libraries are available. - If the env has something non-obvious, document it. E.g., if you installed `numpy` as `np` alias or such, mention usage. But normally, standard libs are fine. - If certain tools (like `curl`, `ffmpeg`) are needed by the project, mention them in README or AGENTS.md. Otherwise Codex might try to call them and not find them, causing confusion.

6. Maintain a Logical Layered Architecture

If you structure in layers (say controllers -> services -> models -> database), Codex will generally respect that when adding new code. - It won't, for example, directly query the DB from the controller if it sees the pattern that controllers call service functions which call models. Unless you instruct otherwise, it tries to follow established layering to be consistent. - If the project is inconsistent, the AI might be uncertain. So try to clean up major inconsistencies (like one feature bypasses service layer when others don't). If it's there for a reason, comment it.

7. Use Types and Dataclasses

- **Type hints:** Codex loves when code has type hints because it clarifies what's expected. If your project is fully type-hinted, Codex will almost always include correct type hints in new code. If not, sometimes it omits them. Decide on one approach and stick to it.
- If not using types, that's okay, but then be clear in other ways (docstrings).
- If using, try to have them in function signatures and class attributes (or use `dataclasses` for brevity).
- **Dataclasses:** If you define data structures as dataclasses (or Pydantic models etc.), Codex will likely instantiate or extend them properly. For example, if `User` is a dataclass with fields, and you ask to add a field, Codex can easily add it and update the `__init__`. If it's just a dictionary, it might do it too but with dataclass it knows exactly where to add.
- **Enum and Constant usage:** If you have Enums or constants, Codex will use them if it knows about them. E.g., `Status.ACTIVE`. If not, it might introduce raw strings. So having central constants can guide it to use those (like define `DEFAULT_TIMEOUT = 30` at top of module; Codex will see that and use `DEFAULT_TIMEOUT` instead of literal 30 if writing related code).

8. Test Suite as Specification

We've emphasized tests, but in terms of layout: - Keep your tests well-organized (mirroring the source structure ideally). If Codex sees a pattern that for each module there's a test_module.py, it will follow by creating appropriate new test files for new modules. - Name tests clearly (function names including what they test). Codex often writes tests with names similar to existing ones, or references them. E.g., if you have a test `test_add_user_success`, Codex might name a new one `test_add_user_failure` for the failure scenario if it sees that convention. - If using a framework like pytest fixtures, have them in conftest.py or so. Codex can detect their usage and use them.

9. Keep Dependencies Up-to-date for Cloud Tools

For Codex Cloud tasks or MCP: - If you plan to run code in Codex Cloud, ensure your environment file covers all dependencies so the cloud container can install them easily [38] . - Possibly have a `requirements.txt` for pip even if using conda, because the cloud environment might use pip if not configured with your environment file.

10. Version Control and Commit Strategy

While not project layout per se, in an agentic workflow: - Commit frequently (especially after a chunk of Codex changes that you've tested). This way, if Codex's next suggestion goes awry, you can revert easily. - You might even have Codex generate commit messages or PRs as we showed. - Use branch for AI-driven changes, then review diffs. Codex can assist in reviewing its own diff by explaining it line by line if you want double-check. - If a change is large, consider splitting it and committing in logical parts (Codex can help separate concerns, but you might need to orchestrate that).

By setting up your project with these principles, you essentially make it *AI-friendly*. The structure becomes additional implicit instructions for Codex, and it will generally produce more accurate and maintainable code that fits in.

Finally, let's discuss troubleshooting – how to handle things when they go wrong or when Codex isn't behaving as expected, especially on our Windows + Conda setup.

Troubleshooting Guide

Even with a great tool like Codex, you may encounter issues. Here's a list of common problems and how to solve them in this specific environment:

1. Codex Extension Unresponsive or Crashing (Windows issues)

Symptom: The Codex sidebar is blank or stuck, or you get errors like "Extension host terminated unexpectedly."

- **Ensure dependencies:** As the Windows guide notes, you might need the MSVC build tools and redistributable installed [45] . Without those, some native code the extension uses might fail. Install those and restart VSCode.
- **Sandbox conflicts:** The experimental sandbox on Windows might be causing trouble (especially if your project has unusual file permissions or paths). If Codex logs (in Output panel, choose "Codex" log) mention sandbox errors, try running VSCode as Administrator (just to see if it's a permission issue). Or, as a last resort, use WSL.
- **Large repository performance:** If VSCode feels slow or Codex lags on large repos, consider:
- Close some directories from VSCode workspace that you don't need AI to see (e.g., huge `node_modules` or data files). Codex might be scanning them.
- Increase memory for WSL (if you were using it) [77] , though in pure Windows, memory might not be the limit, but CPU usage might be (the model running remote, but managing context can be heavy).
- **Update extension:** Check if there's an update (marketplace page or VSCode should hint). New releases often fix crashes.

- **Log an issue:** If reproducible, you can report to OpenAI's Codex GitHub. But for immediate work, consider switching to CLI for that session as a workaround.

2. Codex Not Using Conda Env / Wrong Python

Symptom: Running `python` yields "command not found" or uses wrong version (e.g., 3.8 instead of 3.10 which is in env), or pip installs go to wrong place.

- **Diagnosis:** This means the shell Codex uses isn't activated to your env. Possibly the extension spawns a non-interactive shell by default, not loading conda.
- **Fix 1:** Always use `conda run -n yourenv` in the command you ask Codex to run. E.g., "Run tests" becomes "Run `conda run -n myenv pytest`." It's a bit verbose but guarantees correct env.
- **Fix 2:** Configure VSCode tasks: You can define a task in `.vscode/tasks.json` that runs tests in the env. Then you can ask Codex to run that task (not sure if Codex can directly invoke tasks, but you can press Ctrl+Shift+B to run build tasks).
- **Fix 3:** Put environment activation in AGENTS.md or initial instructions, e.g., "Before running any shell command, execute `conda activate myenv`."
- Note: Activating conda is not trivial in a non-interactive shell (since it's a shell function). So this might not work unless Codex opens an interactive session.
- **Observation:** If Codex forgets env mid-session, just remind: "Use the conda environment." It tends to remember for that session after a couple of nudges.
- **Worst-case fallback:** If you only have one Python (like if you did add Conda's python to PATH), then it might just use that by default. Some users opt to do `conda activate base` and then put base Python in PATH so any `python` refers to base. That's not ideal if base isn't the env you want, but you could do it for quick fix.
- **Check config.toml:** There might be an option to specify the interpreter or shell for Codex. The openai/codex repo might have an issue/feature about conda integration. If a new release has something like `python_interpreter_path` in config, use it. As of writing, not sure if present.

3. Failing to Run Commands (Windows)

Symptom: Codex tries to run `ls` or `grep` and fails (since on Windows those aren't built-in), or it attempts a Linux command like `sudo apt-get` when setting up environment (maybe in cloud tasks).

- **Solution:** Guide Codex to use Windows alternatives:
- For listing files, use `dir` or PowerShell `Get-ChildItem`. You can say, "Use Windows commands; for example, use `dir` instead of `ls`." It will adapt.
- For searching text, suggest PowerShell's `Select-String` or just reading the file in Python.
- **Install Unix tools:** If you have Git Bash, you can instruct Codex to use it by starting commands with `bash -c "ls"`. But that complicates things. Better to avoid needing that.
- **Cloud tasks environment:** If delegating to cloud, it's Linux there, so these issues vanish in cloud runs. But locally, must adapt.
- **MCP server not found (like context7)**: This is another command issue – if `npx` not found, make sure Node is on PATH or use full path. You can also install context7 globally with `npm i -g @upstash/context7-mcp` and then in config use `command = "context7-mcp"` directly.

- If context7 times out on Windows, consider enabling `rmcp_client` (we did) or running context7 outside Codex and connecting via an HTTP MCP config (point to localhost:port that context7 runs on). That requires manual steps but can circumvent some spawn issues.

4. Hallucinations or Incorrect Code

Symptom: Codex writes code that doesn't exist or uses functions from a library that aren't actually available in your version.

- **Examples:** Using `pandas.DataFrame.to_json()` when your pandas version doesn't have that (just hypothetical).
- This is more of an AI general issue: double-check any library usage that you're not sure exists. The plus side: with internet MCP, it might actually look up docs and be accurate. Without, it might rely on training data which could be outdated.
- **Fix:** If Codex suggests something and you suspect it's wrong, ask it explicitly:
- "Is `to_json()` actually a method of DataFrame? Confirm from documentation."
- Without MCP, it might still guess, but often GPT-4 has decent knowledge. If uncertain, manually verify.
- **Unit tests as safeguard:** Write tests for new functionality to catch these. Codex may assume a function returns something, but test will reveal if not.
- **Keep libraries updated**: If you're on a very old version, maybe mention it. "We use Django 2.2" – or else Codex might use features from Django 3.x. It's aware of version differences if told. Possibly include that in AGENTS.md: "Note: using pandas 0.24 (older), so some new APIs aren't available."
- **Hallucinated files or paths:** It might sometimes refer to a file that doesn't exist if confused. Just correct it: "That file doesn't exist, maybe you meant X?" Usually resolves confusion.

5. Overuse of API or Tools Without Keys

Symptom: Codex might respond with, "I attempted to fetch data from the internet but cannot without approval," or it keeps asking to use a tool you haven't configured (like trying GitHub MCP when you didn't set it up).

- **Solution:** Remind or instruct boundaries. "Do not use the internet for this, use local data only," or "We don't have GitHub access here, stick to local solution."
- If it persistently wants external info, perhaps your prompt implicitly suggests it (like asking for "latest version" of something). Reframe question to give it any needed data or say "Assume offline environment."
- If an MCP is configured but you temporarily don't want to use it, you can disable in config or just instruct "Don't use the documentation tool for now, answer from knowledge." Usually it will comply.

6. Hitting Usage Limits

Symptom: You get a message like "You have reached the usage limit for 5 hours" or Codex starts refusing requests temporarily.

- This means you've hit the Plus plan cap (e.g., too many large requests). Options:
- Wait for some time (the window to reset).

- If you have GPT-3.5 Codex available (maybe GPT-5 vs GPT-5-codex difference?), try switching models to a lower tier which might have higher limit. But quality may drop.
- Reduce tokens per request: Instead of asking it to open 5 files, do one at a time (makes each interaction smaller).
- If absolutely needed, use an API key with pay-as-you-go in extension settings as a fallback for that heavy session (bearing cost).
- On the bright side, the limits e.g., 30-150 per 5h [42], are usually enough for a normal coding session. If you find hitting often, either your tasks are extremely large or maybe something is looping (like Codex re-running tests too many times).
- Remember, each code run or file read counts as well. If Codex is stuck in a loop (rare, but maybe running tests repeatedly), break the loop and do it manually to save calls.

7. Mis-Understanding Commands

Symptom: You said "run X" but Codex printed code or vice versa.

- Codex might not always know if you want it to execute or just show code.
- For example, “generate migration script” – it might either produce the script or actually attempt to run `makemigrations`.
- If it does the opposite, clarify: “No, I want you to actually run the command, not just show the code” or “I wanted the code, not to execute it”.
- Typically:
- “Write code to do X” yields code.
- “Run X” yields execution.
- Imperative verbs like “open, run, edit” tend to trigger agent actions.
- If it’s in chat mode it’ll never execute, so ensure mode is correct.
- In ambiguous cases, just correct the course with a clearer prompt.

8. Undoing Changes

Symptom: Codex made a change that you realize is wrong. How to revert?

- If using git: easiest is `git diff` to see changes and `git checkout` to undo the specific ones. You can do that outside Codex, or instruct Codex to revert: “Revert the changes made to `foo.py`.” Codex might try to recall original content. If conversation still has the original (like it was open earlier), it could restore it exactly [78].
- If not using git, VSCode has undo but that might be tricky after multiple files changed. In Codex chat, you could copy from earlier messages where old code was shown and instruct to replace new code with old code snippet (paste it).
- To avoid needing this, commit before trying something experimental or use a separate branch.

9. Getting Stuck or Repeating

Symptom: Codex keeps giving the same answer or says “As I mentioned above…” in a loop without progressing.

- Possibly the conversation reached a confusion point. Try rephrasing the request or breaking it into a smaller chunk.

- Or maybe it's waiting for your input on a plan step. Ensure you explicitly tell it to continue.
- If it's truly stuck (like outputs the same error message analysis 3 times), stop and try a fresh angle: clear some messages (or use "Forget the above attempt, let's try a different approach").
- Each new session resets the model's state, which can fix strange loops.

10. Security Prompts

Symptom: Codex refuses to do something citing policy (could happen if it misinterprets a request as something sensitive).

- E.g., if your code deals with encryption or uses a secret, the AI might erroneously trigger a policy.
- If it happens, rephrase the prompt to be very technical and neutral. E.g., instead of "crack password function" say "optimize the password hash checking function" so it doesn't sound malicious.
- Usually coding tasks are fine, but just in case a word triggers it, phrase differently.

This troubleshooting section should help navigate most hiccups.

At last, let's outline **actionable next steps** – basically what a developer should do after reading this guide to put it into practice in their own environment.

Actionable Next Steps (for Windows 10 + Conda Workflows)

To conclude, here's a checklist of concrete steps and best practices you can apply going forward to maximize your productivity with ChatGPT Plus and Codex:

1. **Set Up Environment & Tools** – *Ensure your system is ready:*
2. Install the Codex VSCode extension and authenticate with your Plus account [1].
3. Install necessary Windows tools (Conda, VS Build Tools, Node.js for CLI, Docker if using MCP, etc.).
4. Configure VSCode to use your Conda environment for terminals. Test a simple Codex run to confirm it uses the right Python.
5. (Optional) Install the Codex CLI as a backup or for CI integration [48]. Test it on a sample project to get familiar with the interface.
6. **Integrate Codex into Daily Workflow** – *Gradually introduce Codex to your development routine:*
7. Start with non-critical tasks to build trust. For example, let Codex refactor a small module or write a minor feature, and review its output.
8. Use it for documentation: ask it to explain tricky pieces of your codebase to verify its understanding aligns with yours.
9. Try agentic execution on something simple (like automating a repetitive change across files) to get used to approval prompts and how Codex interacts with your filesystem.
10. **Maintain a Strong Project Structure** – *Apply the layout recommendations:*

11. Create or update `AGENTS.md` in your repo with essential context and guidelines [6]. This will instantly improve Codex's relevance to your project.
12. Clean up any extremely large or monolithic files if possible, or at least section them with clear comments.

13. Add missing type hints or tests for critical code so Codex has references. For legacy code without tests, even a couple of sanity tests written now can guide future Codex changes.

14. **Write AI-Friendly Prompts** – *Keep the prompting patterns handy:*

15. Develop a habit of specifying exactly what you want. For instance, instead of "fix this", say "fix this by doing X and Y, ensure test Z passes after."
16. Use the plan-first approach for big tasks: it prevents wasted effort on unwanted implementations.
17. When in doubt, ask Codex questions – it's not only a coder but also a knowledgeable assistant. If you're not sure how to approach something, ask it for ideas or pros/cons.

18. Leverage it for code reviews: after you write something, ask "Can you review this function for any issues?" – it might spot edge cases or improvements.

19. **Monitor and Adjust** – *Continuously improve the collaboration:*

20. Pay attention to when Codex falters – was the prompt unclear? Did it not have necessary context? Adapt accordingly (e.g., provide more context next time or update AGENTS.md with clarifications).
21. If Codex does something inefficient (like running a loop it could vectorize), point it out and have it optimize. This teaches you what it might miss and teaches it your preferences.
22. Keep your ChatGPT Plus subscription updated, as OpenAI often improves model capabilities (GPT-5 Codex now, maybe GPT-6 later) [3]. Newer models might handle even more context (maybe entire codebase) which will change how you prompt (could be more high-level).

23. Stay aware of new features: for example, if they release better Windows support or new MCP servers, incorporate those as needed (if relevant to your workflow, like a new MCP for Azure or something if you use that).

24. **Scale Up Usage Carefully** – *Use Codex for larger and more critical tasks with confidence built:*

25. As you gain trust in its outputs, let it handle more complex refactors or even generate whole components. Always review, but you'll likely find fewer corrections needed over time as you and the AI "learn" each other's patterns.
26. Consider using the Codex Cloud for heavy tasks (e.g., running integration tests with external services or analyzing huge code diffs) [37]. If you find your local machine struggling or context not enough, a cloud run might do it.

27. Integrate Codex into CI if it fits (for example, an Autofix action for lint errors [19]). This requires an API key and careful setup, but could save time on trivial fixes.

28. **Feedback Loop** – *Contribute back insights:*

29. If you encounter reproducible issues or have feature requests (like better Conda handling), report them on OpenAI's GitHub [31] or forums. This helps improve the tool for you and others.
30. Share effective prompt techniques or workflow automations with teammates if you're in a team setting – a common approach ensures consistency and everyone benefits from optimal usage.

By following these steps and continuously refining how you interact with Codex, you'll transform your Windows 10 + Conda development environment into a highly efficient, AI-assisted workflow. Codex will handle the boilerplate and mechanical tasks, while you focus on creative design and critical decisions – a true partnership between developer and AI agent.

<small>*(End of Guide)*</small> [8] [10]

---

1 3 5 10 11 19 22 44 46 69 70 Codex IDE extension
https://developers.openai.com/codex/ide

2 Codex IDE extension
https://developers.openai.com/codex/ide/

4 9 14 20 21 26 30 33 34 36 42 43 49 76 Using GPT-5 Codex in VS Code to Build Agentic Workflows - WellWells
https://wellstsai.com/en/post/gpt-5-codex/

6 7 Custom instructions with AGENTS.md
https://developers.openai.com/codex/guides/agents-md

8 68 Codex – OpenAI's coding agent - Visual Studio Marketplace
https://marketplace.visualstudio.com/items?itemName=openai.chatgpt

12 [Feature Request] .codexignore file · Issue #205 · openai/codex
https://github.com/openai/codex/issues/205

13 Working with OpenAI's Codex CLI - Kevinleary.net
https://www.kevinleary.net/blog/openai-codex-cli/

15 16 27 28 29 41 45 47 71 72 77 Windows
https://developers.openai.com/codex/windows

17 37 38 39 40 Codex IDE → Cloud tasks
https://developers.openai.com/codex/ide/cloud-tasks

18 23 24 50 51 52 53 54 55 56 57 58 59 60 61 62 63 66 67 Model Context Protocol
https://developers.openai.com/codex/mcp

25 32 74 75 Prompting guide
https://developers.openai.com/codex/prompting

31 Conda envs · Issue #1068 · openai/codex · GitHub
https://github.com/openai/codex/issues/1068

35 48 64 73 Installing OpenAI Codex on Windows in VSCode and adding githubs MCP server | by Ericsharifi | Oct, 2025 | Medium
https://medium.com/@ericsharifi04/installing-openai-codex-on-windows-in-vscode-and-adding-githubs-mcp-server-d44b7370330d

65 Codex CLI on Windows 11: MCP server (Context7) fails ... - GitHub
https://github.com/openai/codex/issues/2555

78 A way to exclude sensitive files · Issue #2847 · openai/codex - GitHub
https://github.com/openai/codex/issues/2847