OpenAI Codex Handbook

Flavio Copes
flaviocopes.com

November 2026

Contents

What Codex does in your project . . . 3
What you need . . . 4
Start with the desktop app . . . 5
Build a small project to learn the loop . . . 8
Recover from an unwanted change . . . 12
Understand the app interface . . . 15
Organize projects and chats . . . 19
Give Codex useful project instructions . . . 19
Decide what context belongs where . . . 20
Write a good prompt . . . 21
Pick a model and reasoning level . . . 22
Settings worth checking . . . 22
Understand permissions and the sandbox . . . 27
Use Plan mode before a difficult change . . . 28
Make the change . . . 30
Use Goal mode for long work . . . 30
Review every change . . . 32
Commit and push with Git . . . 35
Browse the project files . . . 37
Use the integrated terminal . . . 39
Test web apps in the built-in browser . . . 40
Give Codex images and screenshots . . . 42
Use Computer Use for desktop apps . . . 42
Connect external tools with MCP . . . 44
Reuse workflows with skills . . . 46
Install plugins . . . 49
Use hooks for mechanical rules . . . 52
Local work, worktrees, and handoff . . . 53
Split work across subagents . . . 54
Create scheduled tasks . . . 54
Use Codex from the terminal . . . 56
Other ways to use Codex . . . 62
A real debugging workflow . . . 63
What I learned building a full app with Codex . . . 64
What I don't like about Codex . . . 64
When I pick Codex over Cursor or Claude Code . . . 64
My Codex workflow . . . 65
How I would start on an existing project . . . 65

Preface

This is the offline handbook edition of OpenAI Codex Handbook, written by Flavio Copes.

You can read the online version with interactive diagrams and videos at https://flaviocopes.com/codex/, and find more programming tutorials, courses, and handbooks at https://flaviocopes.com.

> **Video walkthrough:** Watch the embedded video in the online edition at https://flaviocopes.com/codex/.

Codex is OpenAI's coding agent. It reads your project, edits files, runs commands, and can inspect your Git history, project instructions, and any external tools you connect. In this tutorial we'll use it to fix a reading-time calculator, test the result, and review the changes before committing them.

I spend most of my time in the desktop app, where the chat, terminal, browser, and diff all live in one window, so that's where we'll start. If you prefer the terminal or work over SSH, you can follow the same exercise in the CLI. We'll cover the CLI and the shared configuration later in the guide.

Some controls move around as OpenAI updates the app, so check the official Codex documentation if something in your window looks different from the screenshots here.

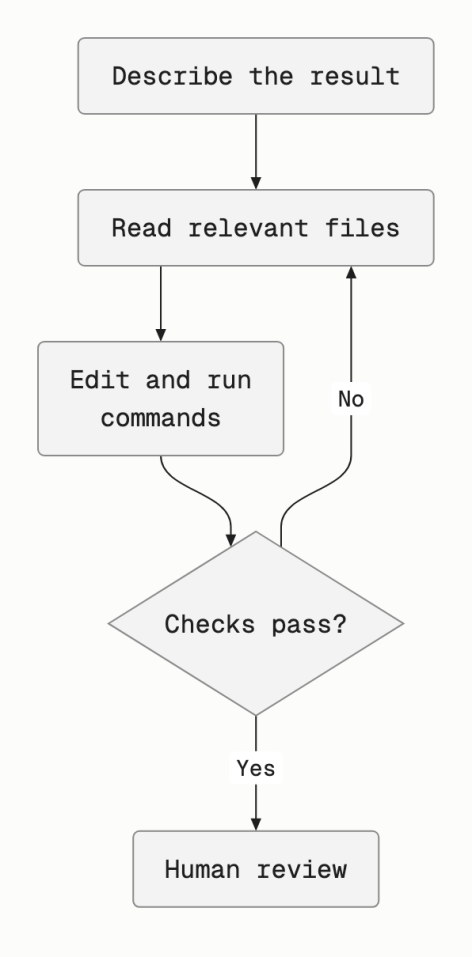
If you already have Codex installed, jump straight to the reading-time exercise. It has the starting files, a change to request, a reference solution, and a recovery exercise. For terminal setup, head to the CLI section.

What Codex does in your project

Codex works directly in your project. You can ask it to find a bug, edit the code, and run the tests instead of copying snippets back and forth between a chat and your editor.

A task can span several files, and in the desktop app Codex can also open a web app in the browser to check its own work.

The agent works in a loop, reading the result of each tool call before deciding what to do next:



I treat it like a developer who just joined the project. I explain what I want, point it at the useful context, and say how I'll decide the work is done.

If this workflow is new to you, my AI fundamentals guide explains agents, tools, context, permissions, and verification from the beginning.

What you need

You need a project folder, a ChatGPT account or an OpenAI API key, and Git installed. The exercise also needs a recent version of Node.js. Git gives us a history of changes and a way to return to a known version.

Codex is included in every ChatGPT plan, even the Free one. The individual plans are Free ($0), Go ($8 a month), Plus ($20), and Pro ($100, $200, or $500 a month), and Business, Enterprise, and Edu plans include it as well.

Free and Go include Codex in the desktop app with GPT-6 Luna, though access is still rolling out. Plus adds the CLI, the IDE extension, web and iOS access, cloud features like code review, and the GPT-6.2 Sol and GPT-6.2 Astra models.

Among the Pro tiers, OpenAI doubled the included Work and Codex usage allowances on Pro 100 and Pro 500, and canceled a planned reduction on Pro 200. Pro 100 doesn't include Ultrafast mode, Pro 200 includes Ultrafast for GPT-6.2 Sol in Codex, and Pro 500 includes Ultrafast across flagship models including Astra. El-

igible Pro subscribers can also claim matching monthly OpenAI API credits ($100, $200, or $500 depending on the tier).

Your plan gives you a usage allowance that Codex shares with ChatGPT Work, so a long agent session and your Work tasks draw from the same pool. OpenAI also announced combining standard ChatGPT usage (outside simple Instant Chat) into that shared pool for Plus, Pro, and Business plans starting in December 2026. How far it stretches depends on the model, the size of the task, and whether it runs locally or in the cloud. The usage page in the app shows where you stand and when the limits reset. Check it every week or two until you know your pace.

You can also run Codex with an OpenAI API key. You then pay standard API rates instead of drawing from a ChatGPT plan, though cloud features that need a ChatGPT account, such as automatic GitHub code review, don't work with a key. The authentication docs list the sign-in methods.

We'll use a small JavaScript project that runs locally, with no database or deployment service to configure.

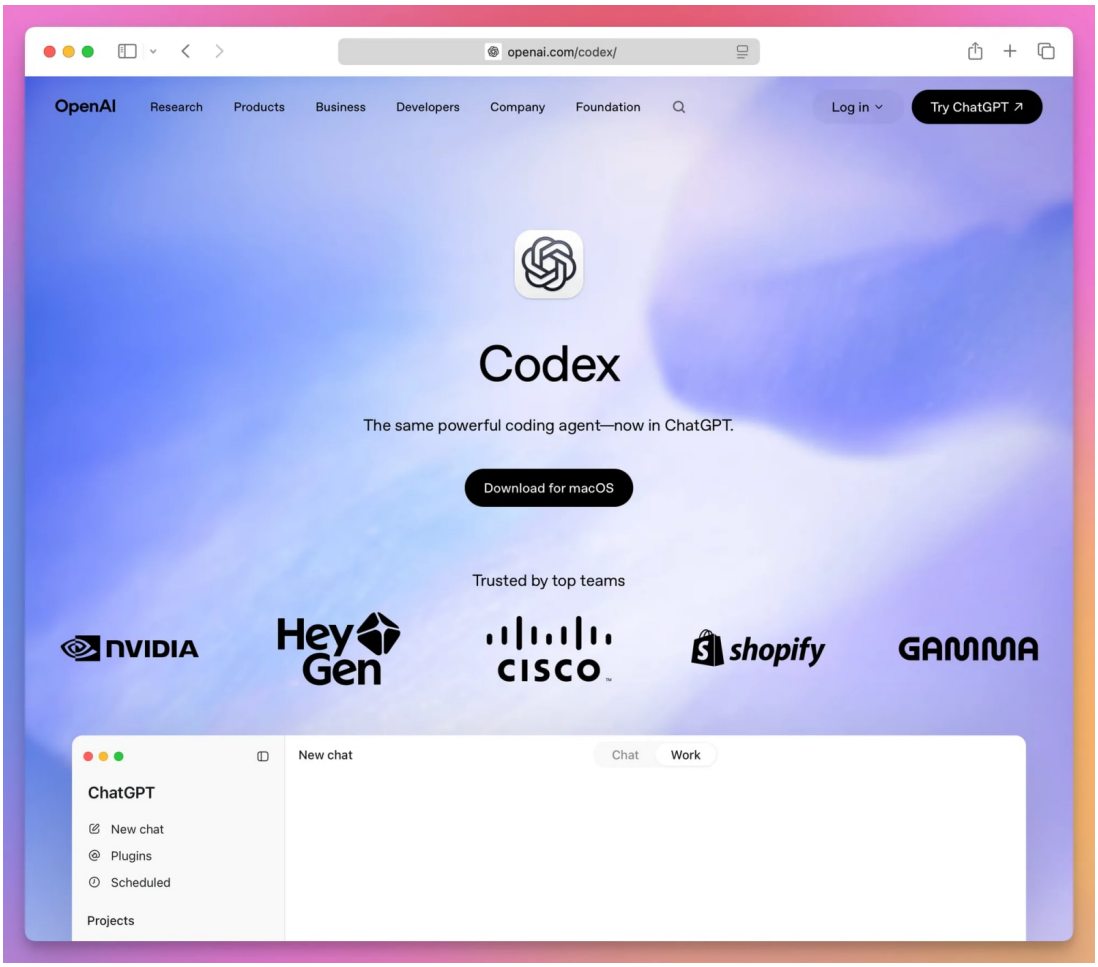
Interface	Best for	What to check
Desktop app	A local app with a browser flow to test	Correct project and permissions
CLI	A terminal session or repeatable script	Current directory and sign-in method
IDE extension	A task starting from an open file	Workspace and attached selection
Cloud	Work delegated to a configured remote environment	Setup commands and required services

Start with the desktop app

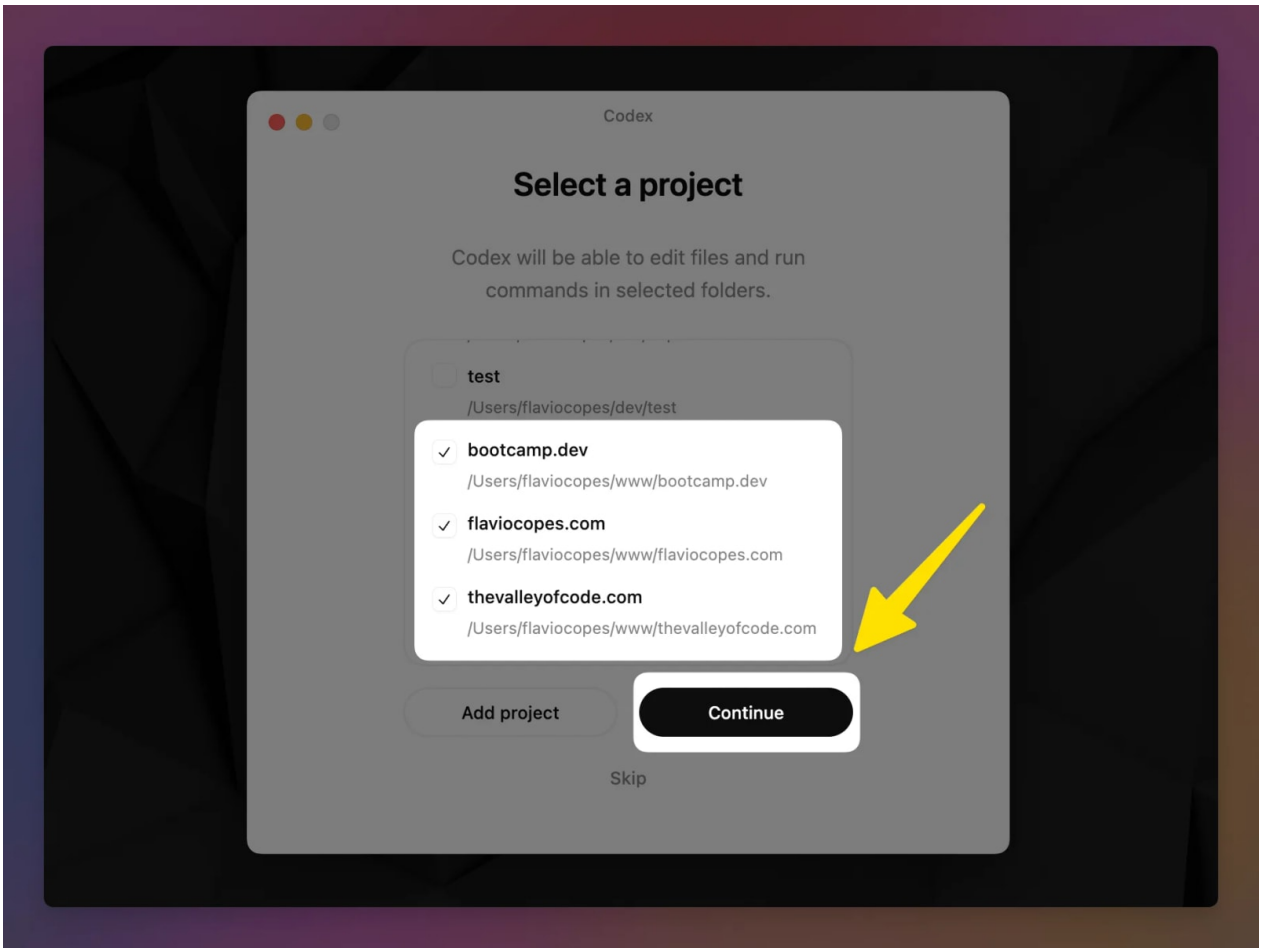
Codex lives inside the ChatGPT desktop app, available on macOS, Windows, and Linux. Switch to Codex in the app switcher (the other option, ChatGPT, holds the Chat and Work views). I'll call it the Codex desktop app throughout this guide.

You can add local projects and run several chats at once. You can also inspect diffs, open files, use a terminal, preview web apps, install plugins, and create scheduled tasks without leaving the app.

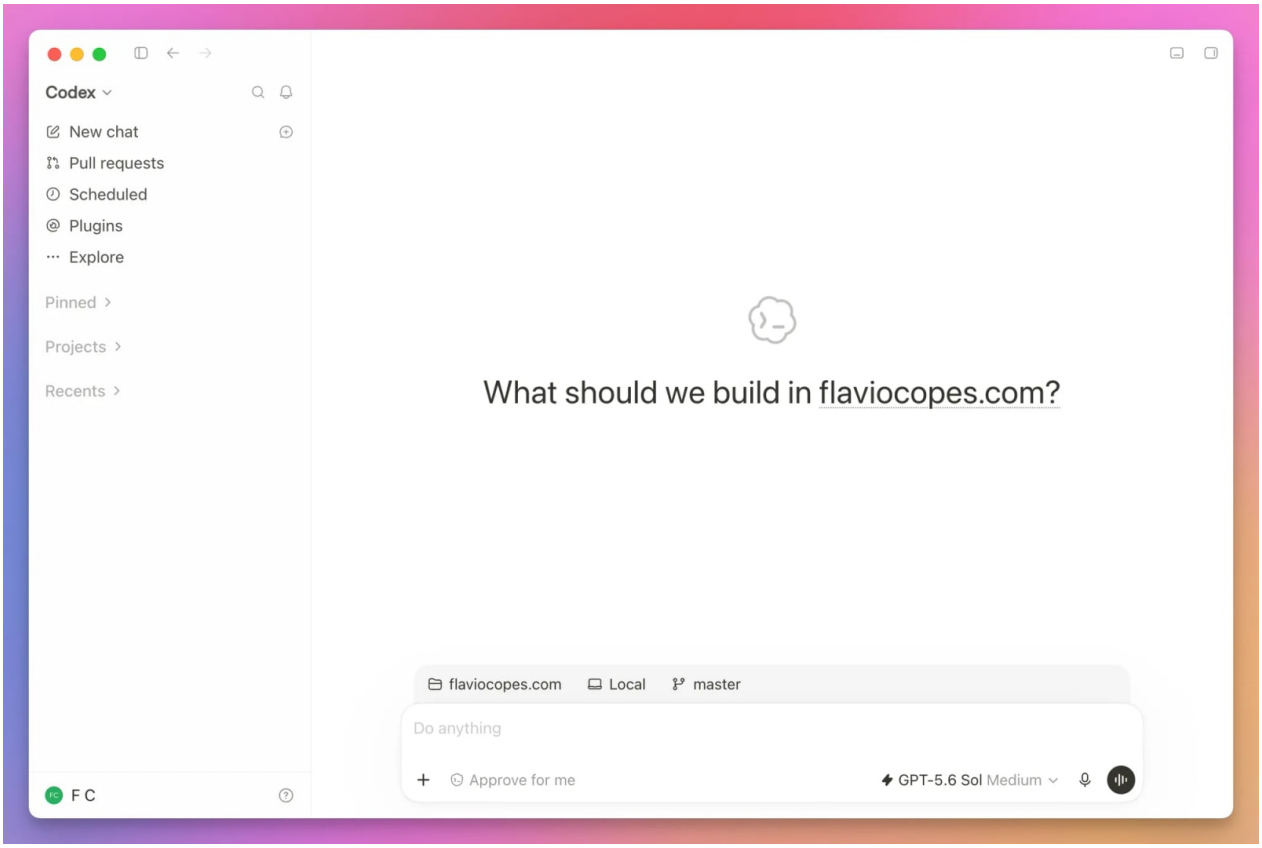
Download it from openai.com/codex, sign in, and switch to Codex.



Then add the folder you want to work on. Codex lists the Git repositories you used recently, or you can pick any folder from disk.



Once the app opens, you can start a chat or select one of your projects from the sidebar.



Before asking Codex to change anything, try a read-only question:

```
Explain what this project does, ELI5.
```

Check its answer before asking for edits. It should identify the project correctly and point to the relevant files.

I like the app because I can stay in one place from the first prompt to the final review. I can also leave a long task running, move to another chat, and come back when Codex needs me.

Build a small project to learn the loop

Our reading-time calculator takes a string of text and returns the number of minutes needed to read it. Its source and tests fit in a few files, so we can read every change Codex makes.

We'll start with working code and ask Codex to add one missing behavior.

Create a folder and the directories for the source and tests:

```
mkdir reading-time
cd reading-time
mkdir src test
```

Create `package.json` in that folder:

```
{
  "name": "reading-time",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "test": "node --test"
  }
}
```

We'll use Node's built-in test runner, so there are no dependencies to install.

Create `src/reading-time.js`:

```
export function readingTime(text, wordsPerMinute = 200) {
  const words = text.split(/\s+/).filter(Boolean).length
  return Math.max(1, Math.ceil(words / wordsPerMinute))
}
```

The function splits text at whitespace and rounds up the reading time. It returns at least one minute, even for empty text. We'll keep that behavior while changing how it handles code blocks.

Now create `test/reading-time.test.js`:

```
import { test } from 'node:test'
import assert from 'node:assert/strict'
import { readingTime } from '../src/reading-time.js'

test('a short post takes one minute', () => {
  assert.equal(readingTime('Hello world'), 1)
})

test('four hundred words take two minutes', () => {
  assert.equal(readingTime('word '.repeat(400)), 2)
})

test('empty text keeps the one minute minimum', () => {
  assert.equal(readingTime(''), 1)
})
```

Run the tests yourself before introducing the agent:

```
npm test
```

Fix any failures before continuing. You want a passing baseline before asking Codex to change the function.

Store the starting version in Git:

```
git init
git add package.json src test
git commit -m "Add reading time calculator"
```

If Git asks for an author name and email on your first commit, configure them using the Git setup lesson and retry the commit. Once that checkpoint is in place, open this folder in Codex: add it as a project in the desktop app, or run `codex` from this directory in the terminal.

The calculator currently counts fenced code blocks as prose. We want to exclude those blocks from the estimate.

Start with an explanation:

```
Explain how the reading-time calculator works and which tests cover it.
Don't edit files. Point out what happens to Markdown fenced code blocks.
```

Read the answer. Codex should identify the function and the test file. If it starts talking about a different project, fix the project selection before continuing.

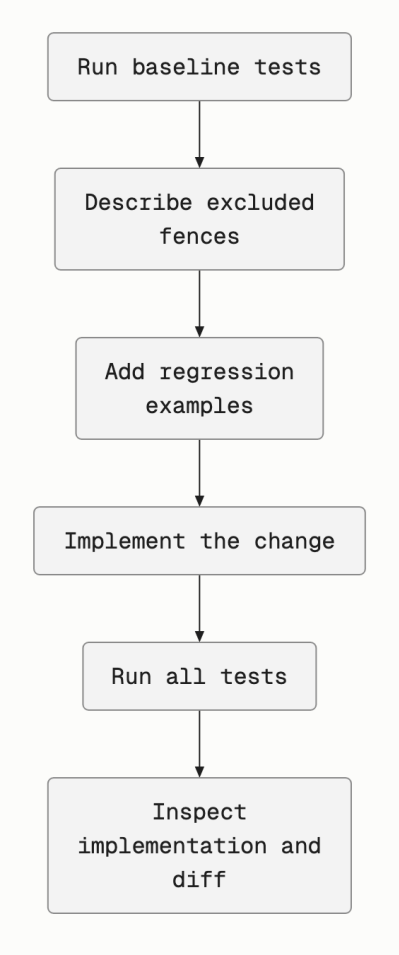
Then give it the change:

```
Exclude Markdown fenced code blocks from the reading-time calculation.
Handle backtick and tilde fences, including an unclosed fence at the end.
Preserve the one-minute minimum and the wordsPerMinute parameter.
Inline code should still count as text.
```

```
Add focused tests and run npm test. Use no dependencies.
Keep the function's public API unchanged.
```

This request names the fence formats to support, the behavior to keep, and how Codex should check the change.

The model may choose a different implementation each time, so judge the behavior rather than expecting one particular regular expression.



Check the test the agent added

A test with ten prose words and ten code words returns one minute with either implementation. It passes even when the bug is still there.

Use enough code to cross the reading-time boundary. For example, this case should stay at one minute even though the fenced section contains 800 words:

```
test('fenced code does not inflate reading time', () => {
  const markdown = ['A short introduction.', '```js', 'word '.repeat(800), '```'].join('\n')
  assert.equal(readingTime(markdown), 1)
})
```

Ask Codex to verify that this test fails against the original implementation and passes with the fix. That confirms the test catches the bug.

Now look at what it considers a fence. Can it handle indentation? Does it close a tilde block with backticks by mistake? What happens when an opening fence has a language label?

Check which Markdown edge cases your version handles. If you need full Markdown compatibility in a production feature, choose a real parser or document the subset you support.

Refine the same task

Suppose the agent's first version removes inline code too. Continue in the same chat:

```
The prompt said inline code should count as text. The implementation removes it.
Add a regression test that fails with this version, preserve inline code,
and rerun the tests. Keep the fenced-code behavior.
```

That points Codex back to the inline-code requirement and asks for a test, so it has a concrete behavior to fix.

When you're satisfied, inspect `git diff` or open the app's review panel. You should see changes to the calculator and its tests. A new framework, a build pipeline, or an unrelated dependency would need an explanation.

Once the behavior matches the prompt and you've reviewed the tests and diff, commit the change.

A reference result to compare with

Try the agent exercise before reading this implementation. If you get stuck, replace `src/reading-time.js` with this complete reference and compare it with the diff Codex produced:

```
export function readingTime(text, wordsPerMinute = 200) {
  const prose = []
  let fence = null

  for (const line of text.split(/\r?\n/)) {
    if (fence) {
      const closing = new RegExp(`^ {0,3}${fence.character}{${fence.length},}\\s*$`)
      if (closing.test(line)) fence = null
      continue
    }

    const opening = line.match(/^ {0,3}(`{3,}|~{3,})(.*)$/)
    if (opening && !(opening[1][0] === '`' && opening[2].includes('`'))) {
      fence = { character: opening[1][0], length: opening[1].length }
      continue
    }

    prose.push(line)
  }

  const words = prose.join('\n').split(/\s+/).filter(Boolean).length
  return Math.max(1, Math.ceil(words / wordsPerMinute))
}
```

This recognizes top-level backtick and tilde fences with up to three leading spaces. A closing fence must use the same character and be at least as long as the opening one, an unclosed fence excludes the remaining lines, and inline code stays in the prose.

It's a small parser built for this exercise, so it skips nested Markdown containers like fenced blocks inside blockquotes. Keep that limit in mind before reusing it in a publishing system.

Keep the original tests, then append these cases to `test/reading-time.test.js`:

```
test('tilde fences do not inflate reading time', () => {
  const markdown = ['Intro.', '~~~js', 'word '.repeat(800), '~~~'].join('\n')
  assert.equal(readingTime(markdown), 1)
})

test('an unclosed fence excludes the remaining code', () => {
  const markdown = ['Intro.', '```js', 'word '.repeat(800)].join('\n')
  assert.equal(readingTime(markdown), 1)
})

test('a different fence character does not close the block', () => {
  const markdown = ['Intro.', '```js', '~~~', 'word '.repeat(800), '```'].join('\n')
  assert.equal(readingTime(markdown), 1)
})

test('inline code still counts as text', () => {
  assert.equal(readingTime('`word` '.repeat(400)), 2)
})

test('the custom reading speed still applies', () => {
  assert.equal(readingTime('word '.repeat(400), 100), 4)
})
```

Run `npm test` after adding these and the backtick regression test above. With Node.js, this reference passes all nine tests: the three starting tests, the backtick regression, and these five. If you run into other Markdown edge cases in your own content, add a test for each one before changing the parser.

Recover from an unwanted change

Before starting another task, commit the reading-time behavior you reviewed above. Run `git status --short` to make sure the working tree is clean, so you have a known point to return to.

For this exercise, make an unwanted edit yourself: change the minimum in `src/reading-time.js` from `1` to `0`. Run `npm test`, and the empty-text test should fail.

Stop any running Codex task before undoing its changes. Otherwise it can write another edit while you're reviewing the first one.

Open the app's diff panel, or inspect the two exercise files in the terminal:

```
git status --short
git diff -- src/reading-time.js test/reading-time.test.js
git diff --cached -- src/reading-time.js test/reading-time.test.js
```

The first diff shows unstaged changes, and the second shows staged changes. In this exercise, only the minimum should have changed, and nothing should be staged.

If those files contain only the unwanted work from this attempt, restore their working copies from the last commit:

```
git restore --source=HEAD --worktree -- src/reading-time.js test/reading-time.test.js
npm test
git status --short
```

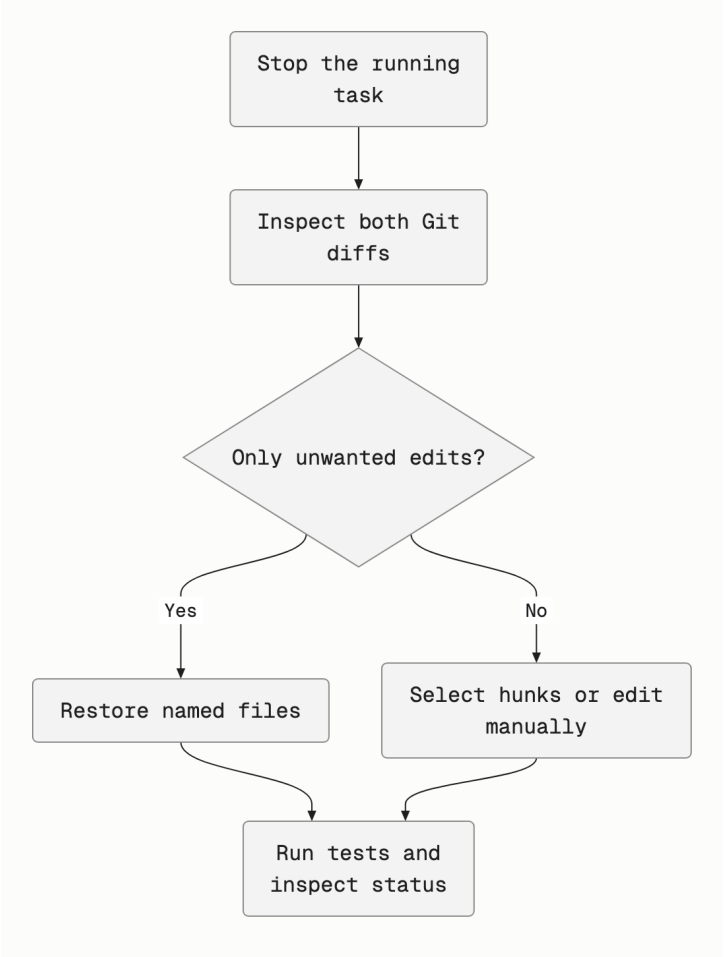
This replaces the working copies of those two tracked files, and your tests should pass again. It leaves other paths and untracked files alone, and it doesn't unstage changes.

Don't use that whole-file command when a file also contains edits you want to keep. Choose the unwanted hunks interactively instead:

```
git restore --patch --source=HEAD --worktree -- src/reading-time.js
```

Read each patch before answering. Split a hunk when Git offers that option, or quit and edit the file manually if the changes are mixed together. Check the staged diff separately, since a working-tree restore doesn't touch anything you've already staged.

The app also has revert controls in its review panel. Read the selected file or hunk before confirming. Keep in mind that starting a new chat won't undo file edits, and a Git restore won't undo changes an MCP tool made in another service.



Carry useful findings into a fresh chat

If the approach was wrong but the investigation turned up useful details, save those findings before starting over. Ask Codex to write down the failing input, the relevant files, and the checks still needed, then review that file too.

For example, save this as `RECOVERY.md` in the project:

```
# Reading-time task

Keep the one-minute minimum, including for empty text.
Exclude backtick and tilde fenced code blocks. Inline code still counts.
The source is src/reading-time.js; tests live in test/reading-time.test.js.
Run npm test. Check that regression tests fail against the original behavior.
```

Start a new chat in the same project and attach the file:

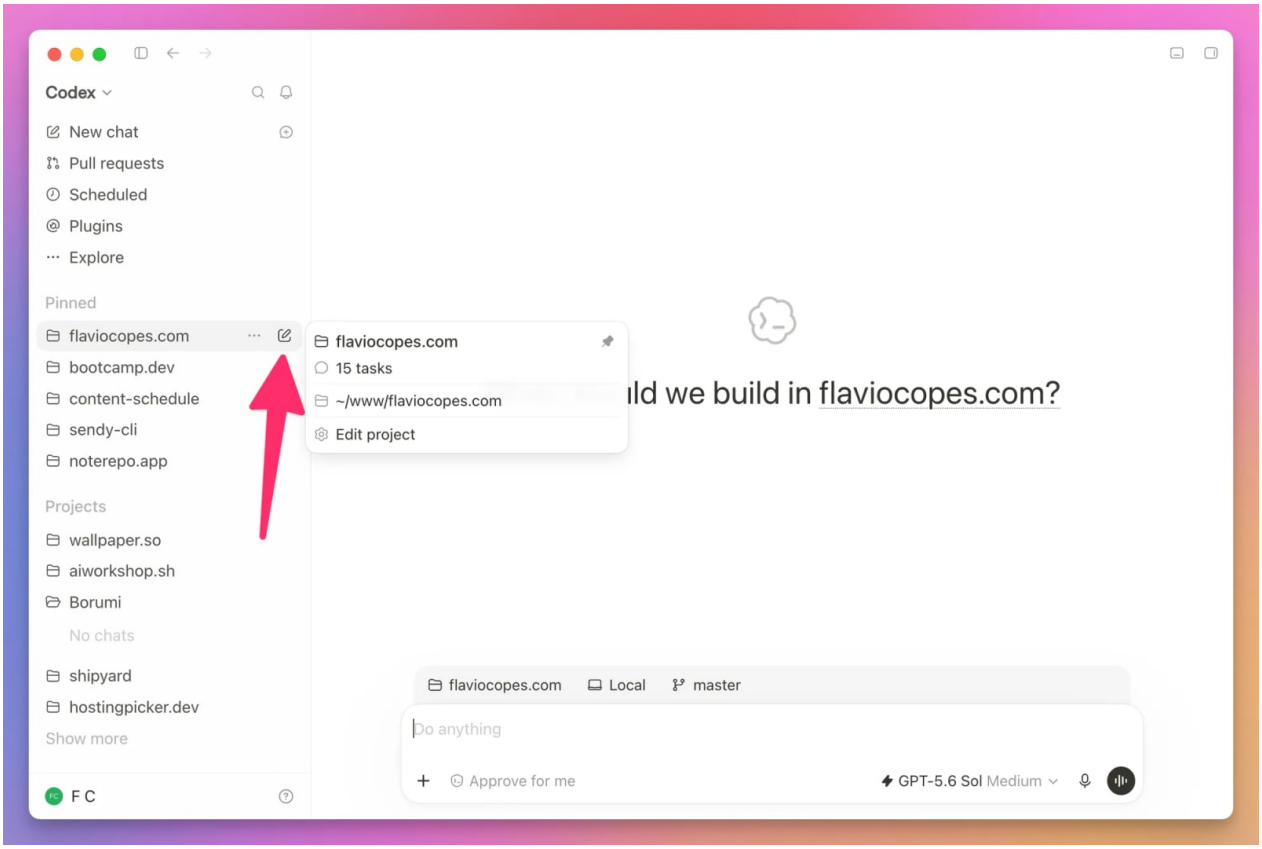
```
Read RECOVERY.md and inspect the current Git diff.
Explain the next small change before editing anything.
Preserve changes that were already present when this chat started.
```

If a change is already committed, use a reviewed revert commit instead of pretending it's an uncommitted edit. My Git course covers that workflow and how to inspect the history first.

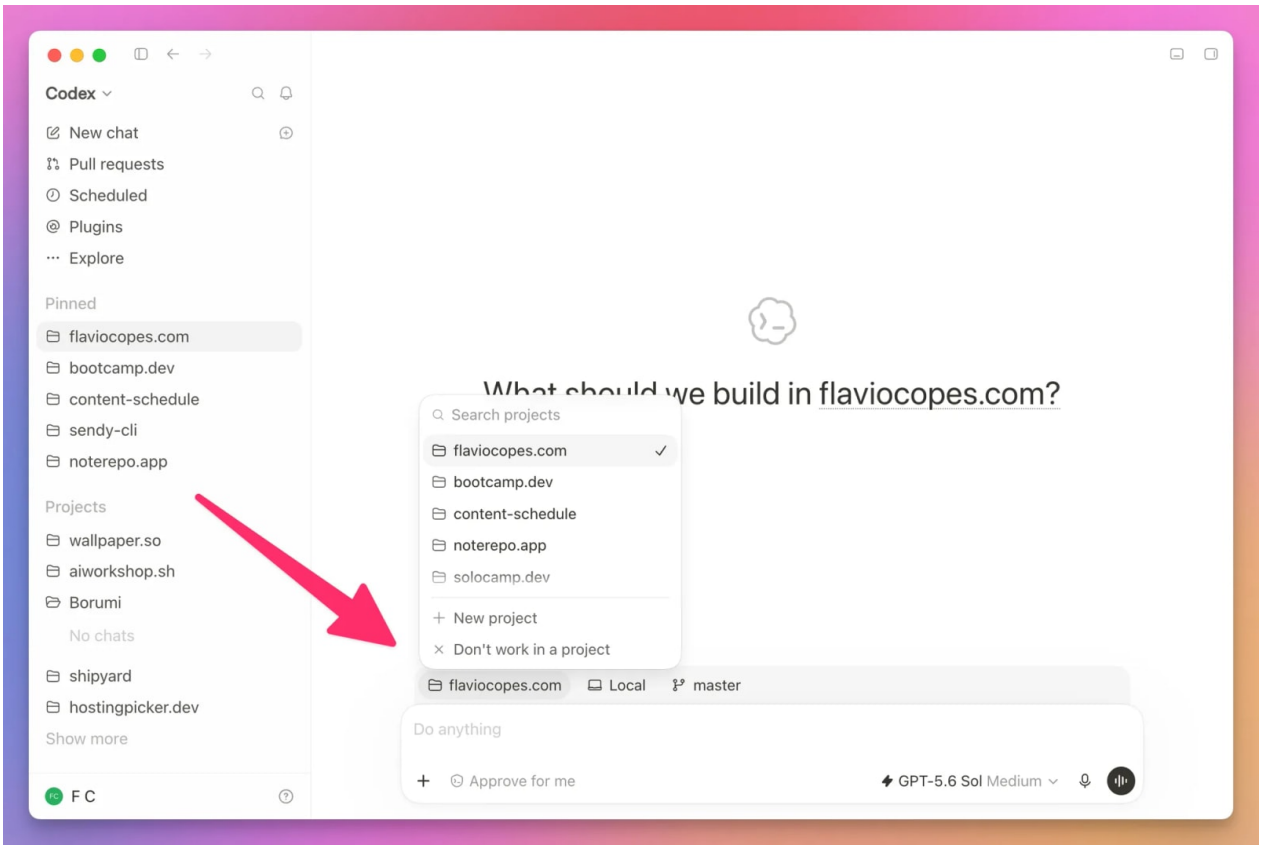
Understand the app interface

Your projects and chats are in the sidebar. You'll also find plugins, scheduled tasks, and settings there.

Hover over a project and click the new chat button.



Click the project name above the prompt to switch projects, add another one, or start a chat without a project.

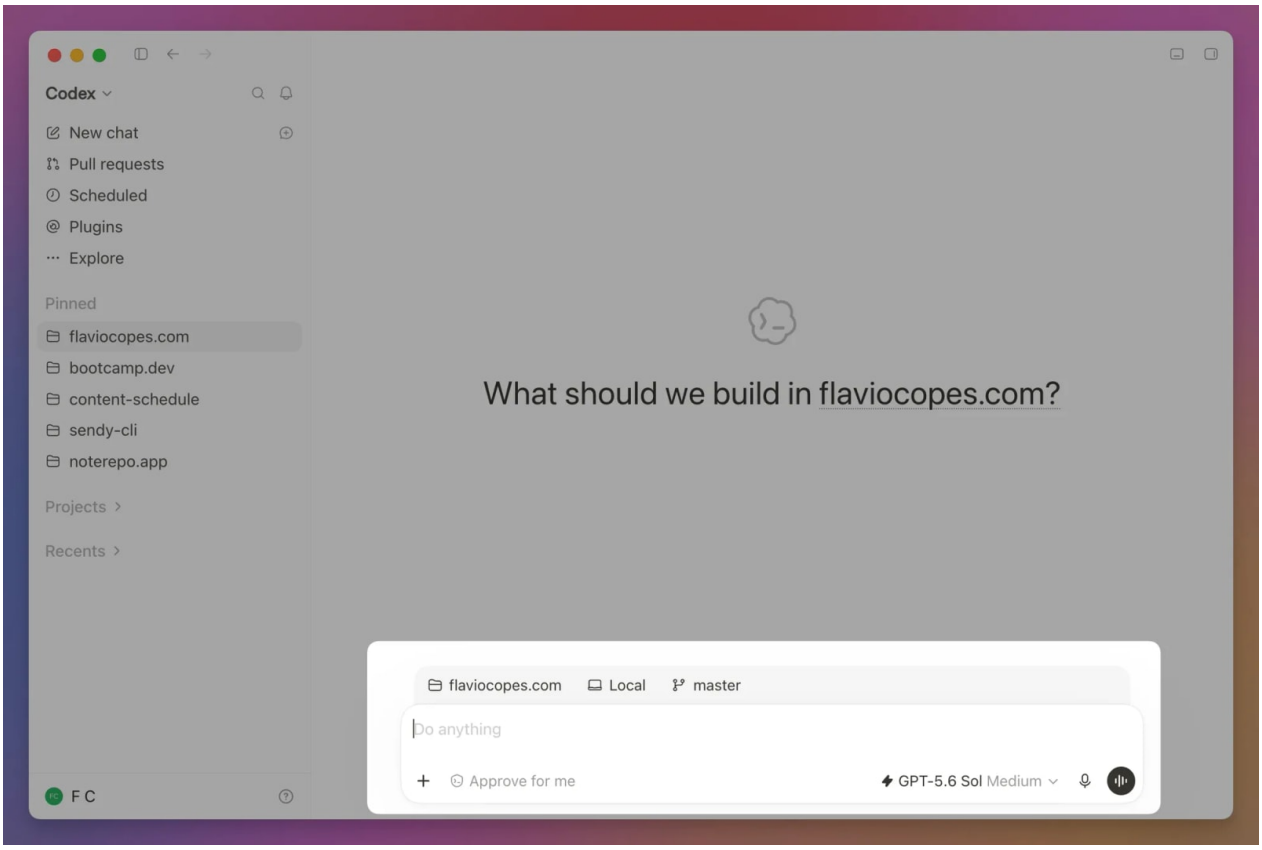


The conversation is in the center. Codex uses it to explain what it's doing, ask questions, show command output, and report back when it's done.

On the right you can open the project files, a terminal, the built-in browser, or the Git diff. I open and close these panels depending on what I'm doing.

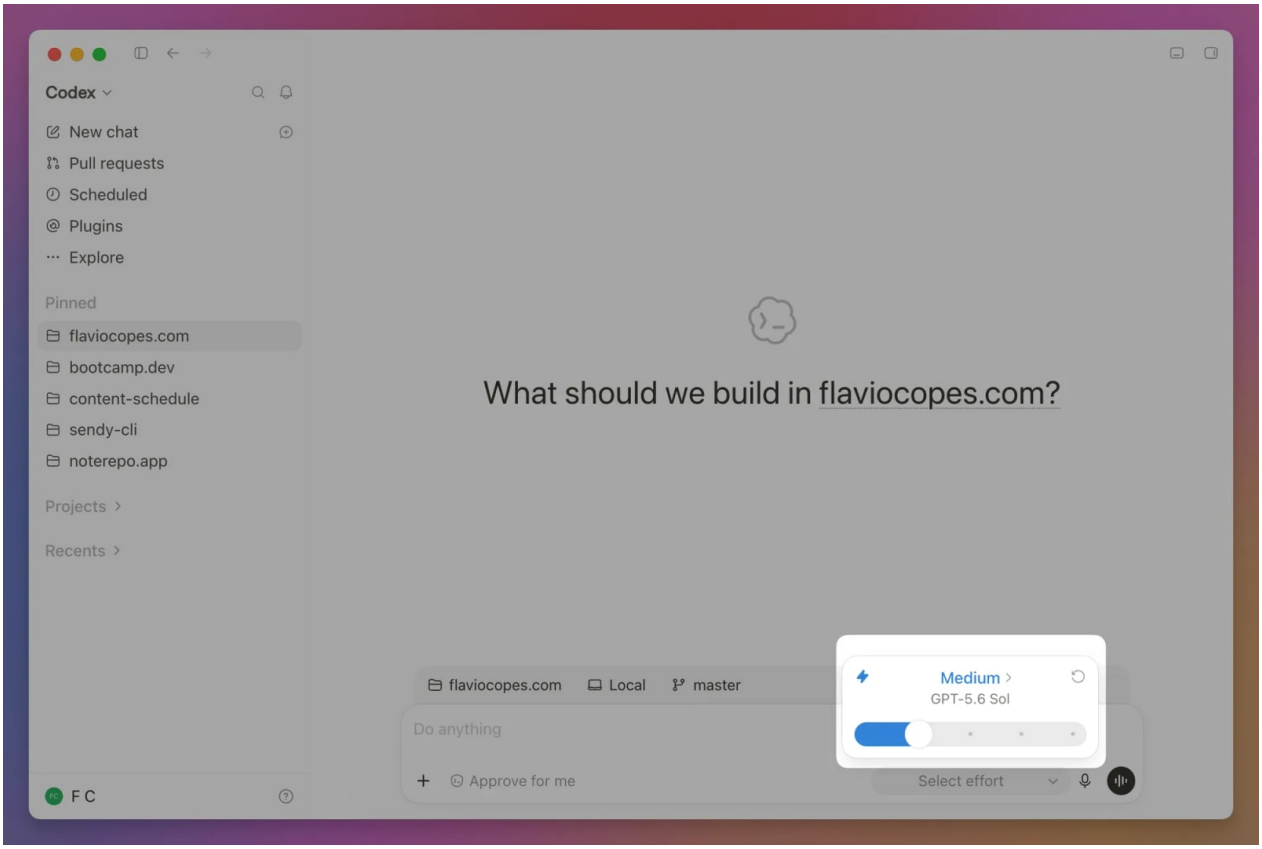
At the bottom is the prompt panel. Besides writing your request, you can use it to:

- attach a file or image
- choose a model and reasoning level
- choose the project, environment, and branch
- change what Codex can do without asking

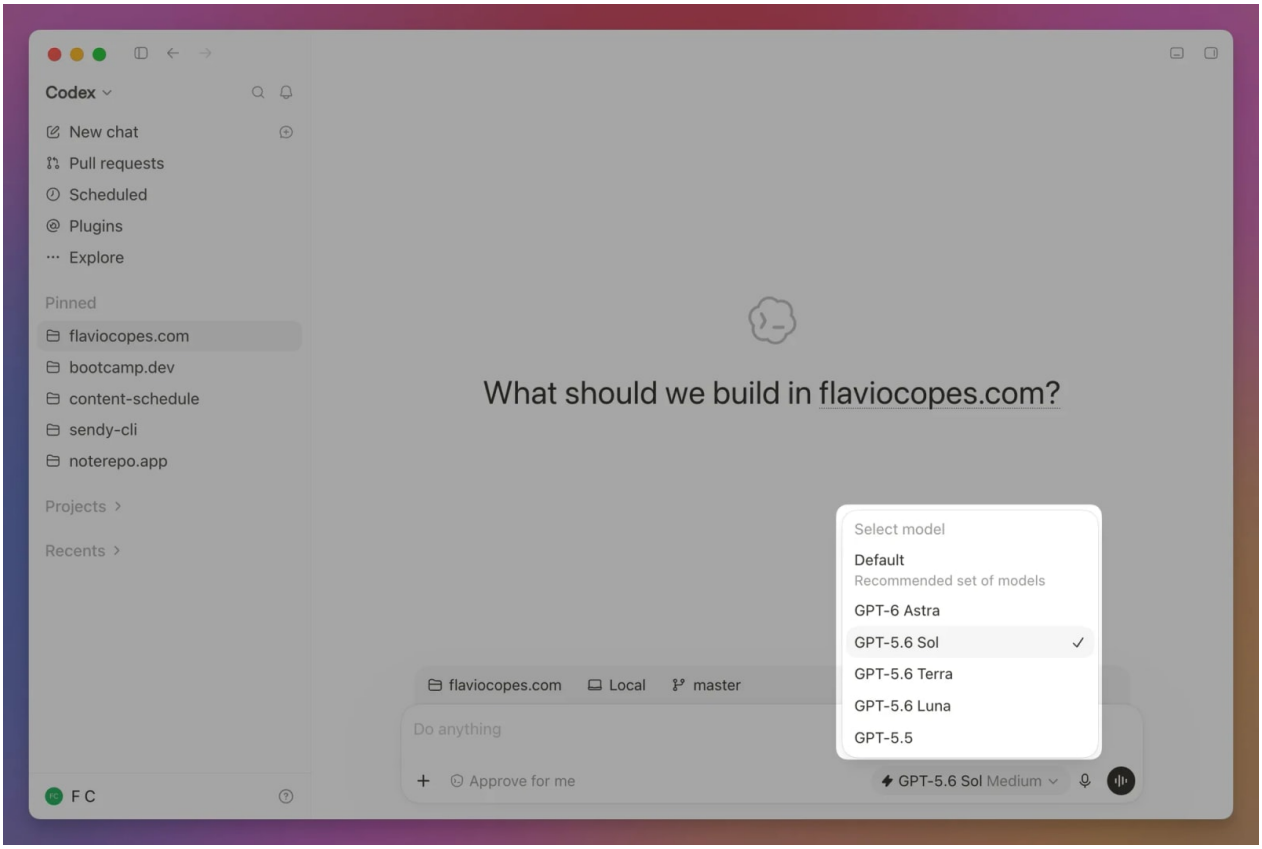


In a Git repository, you can also choose whether the chat works in your local checkout or in a separate worktree. We'll get to that choice later.

Click the reasoning level to decide how much time Codex should spend thinking about the task.

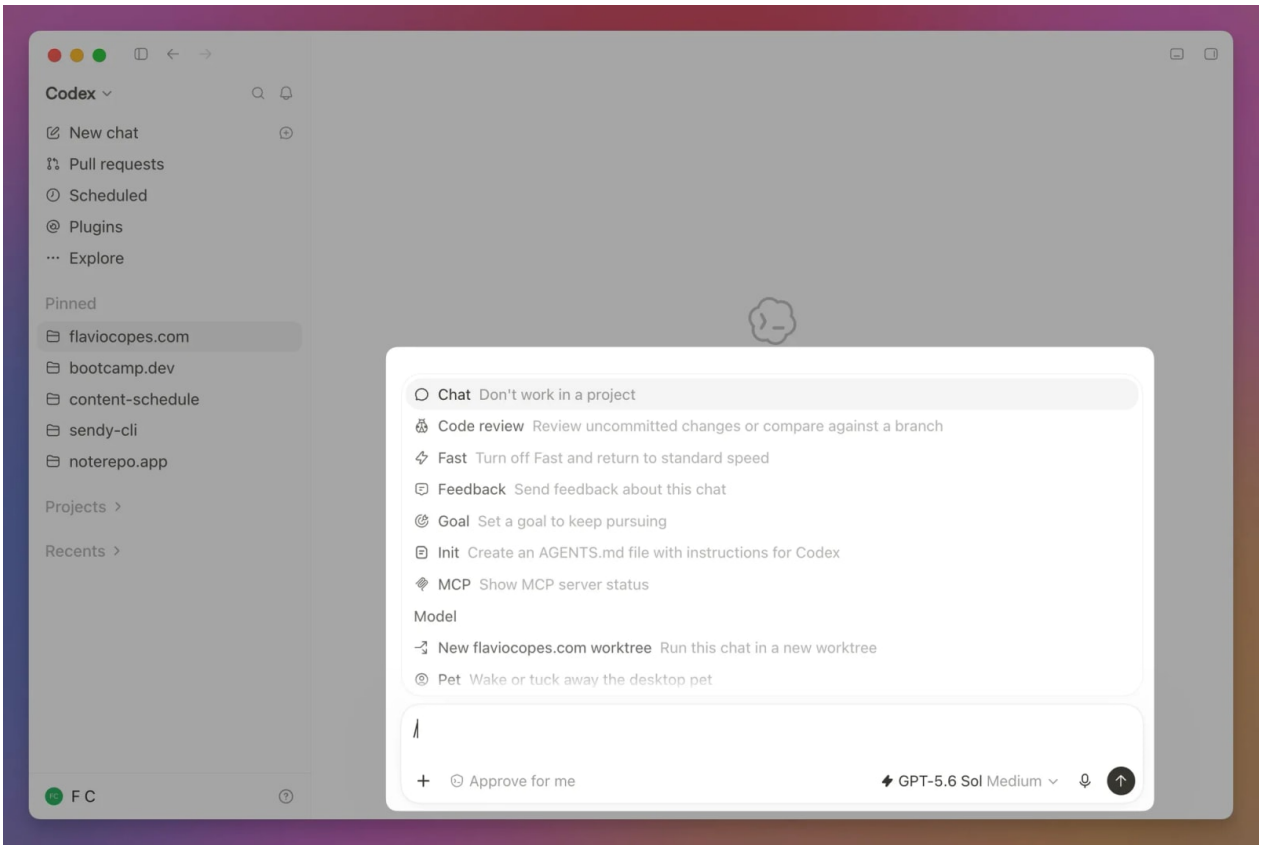


Click the model name when you want to use a different model.



Newer versions list GPT-6.2 Sol and GPT-6.2 Astra in this menu. The screenshot above was taken before they launched.

Type `/` to open the commands available in the current chat.



Some controls depend on the project and your app version, so your screen might look slightly different from these screenshots.

Organize projects and chats

A **project** connects Codex to a folder. Each **chat** holds one conversation and its work.

I keep an authentication fix in one chat, including the follow-up tests and corrections. An unrelated dependency update gets its own chat.

Rename chats you want to find later. I pin the important ones and archive the ones I'm done with. Archiving removes a chat from the active list without deleting its transcript.

Give Codex useful project instructions

Codex automatically reads files named `AGENTS.md` before it starts working.

Use `AGENTS.md` to explain how Codex should work on the project: which commands to run, which conventions to follow, and what to check before finishing.

You can ask Codex to create a starting file after it inspects the project. Then edit it, because the generated file is only a first draft.

A useful `AGENTS.md` might look like this:

```
# Project instructions
```

```
## Commands

- Run `npm run dev` to start the site.
- Run `npm test` after changing JavaScript files.
- Run `npm run build` before finishing.

## Code style

- Use plain JavaScript, not TypeScript.
- Use 2-space indentation.
- Do not add a dependency without asking first.

## Definition of done

- The requested behavior works.
- Existing tests still pass.
- New behavior has a test when practical.
```

Keep the file short and specific. Instead of asking for "good code", list the commands, directories, constraints, and recurring mistakes Codex needs to know about.

You can put one `AGENTS.md` in the repository root and more specific files inside subdirectories. Instructions closer to the current working directory take precedence. The official discovery rules explain the global file, root-to-working-directory chain, and overrides.

I wrote a separate guide about how AGENTS.md works, including global instructions, nested rules, overrides, and maintenance.

Decide what context belongs where

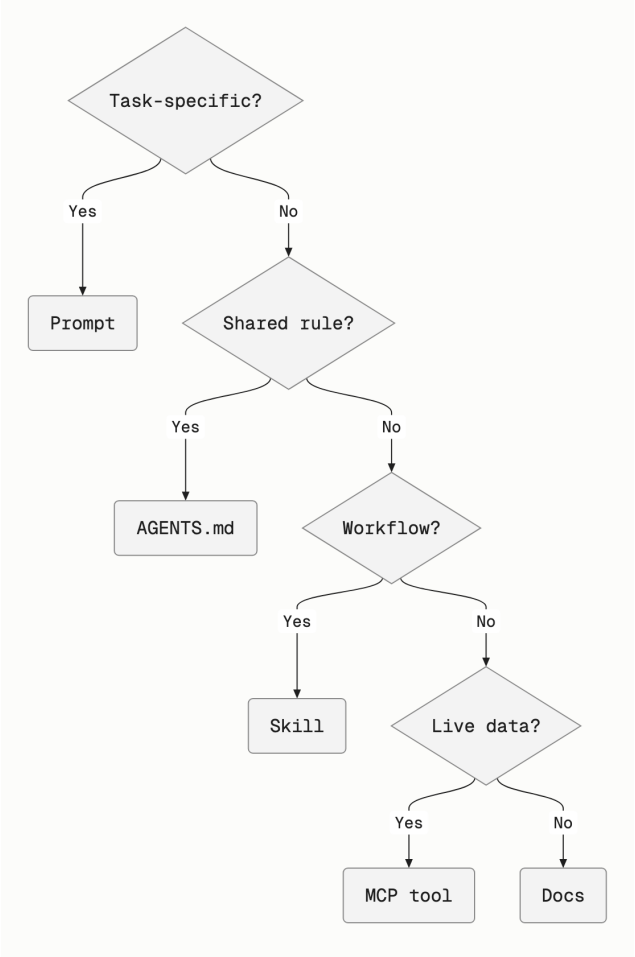
It's tempting to paste the whole project into your first prompt, but Codex can search files on its own. Start with the outcome you want and the context it can't figure out just by reading the code.

In the reading-time calculator, the one-minute minimum is right there in the code, but the reason for keeping it is a product decision. Saying that explicitly stops Codex from "fixing" empty text to return zero while it's working on code fences.

Information	Where to put it	Example
Requirement for this change	Current prompt	Inline code still counts
Project-wide working rule	`AGENTS.md`	Run `npm test` after a JavaScript change
Repeatable procedure	A skill	Review a release before tagging it
Current external data	Plugin or MCP tool	Read the failed CI job

Information	Where to put it	Example
Decision needed in a future chat	Repository documentation	Why the estimate has a one-minute minimum

Save any decisions you'll need in future chats right in the repository. A new chat won't necessarily have the earlier conversation where you agreed on that one-minute minimum.



Write a good prompt

Describe the behavior you want to change. Codex can read the project, but it needs you to explain what the result should be.

Include the details Codex can't guess:

- the result you want
- the behavior that must stay the same
- a useful file, error, screenshot, or Git commit when you have one
- a check it can run when the work is done

This prompt is too vague:

```
Improve the settings page.
```

Give Codex the missing details:

```
On the settings page, the Save button stays enabled after a successful save.
Disable it until the form changes again.

The form is in src/components/SettingsForm.tsx.
Keep the existing API and visual design.
Add a test for the saved state, then run the settings form tests.
```

The file path gives Codex a starting point. The rest explains what to change and what to preserve.

Examples are more useful than adjectives. Instead of asking for a "robust" URL function, include a few inputs and the exact outputs you expect.

Point to real evidence when you have it. Paste the full error instead of summarizing it, attach a screenshot for visual bugs, and name the file if you already know where the code lives.

Let Codex inspect the project before you dictate the implementation. Focus on the result and the constraints, especially when you're not sure how the existing code works.

If you don't know which files matter, don't guess. Ask Codex to find them:

```
Find the code responsible for saving the settings form. Explain the current flow,
then propose the smallest change that keeps Save disabled after a successful save.
Do not implement it yet.
```

Pick a model and reasoning level

The composer lets you choose a model and its reasoning level.

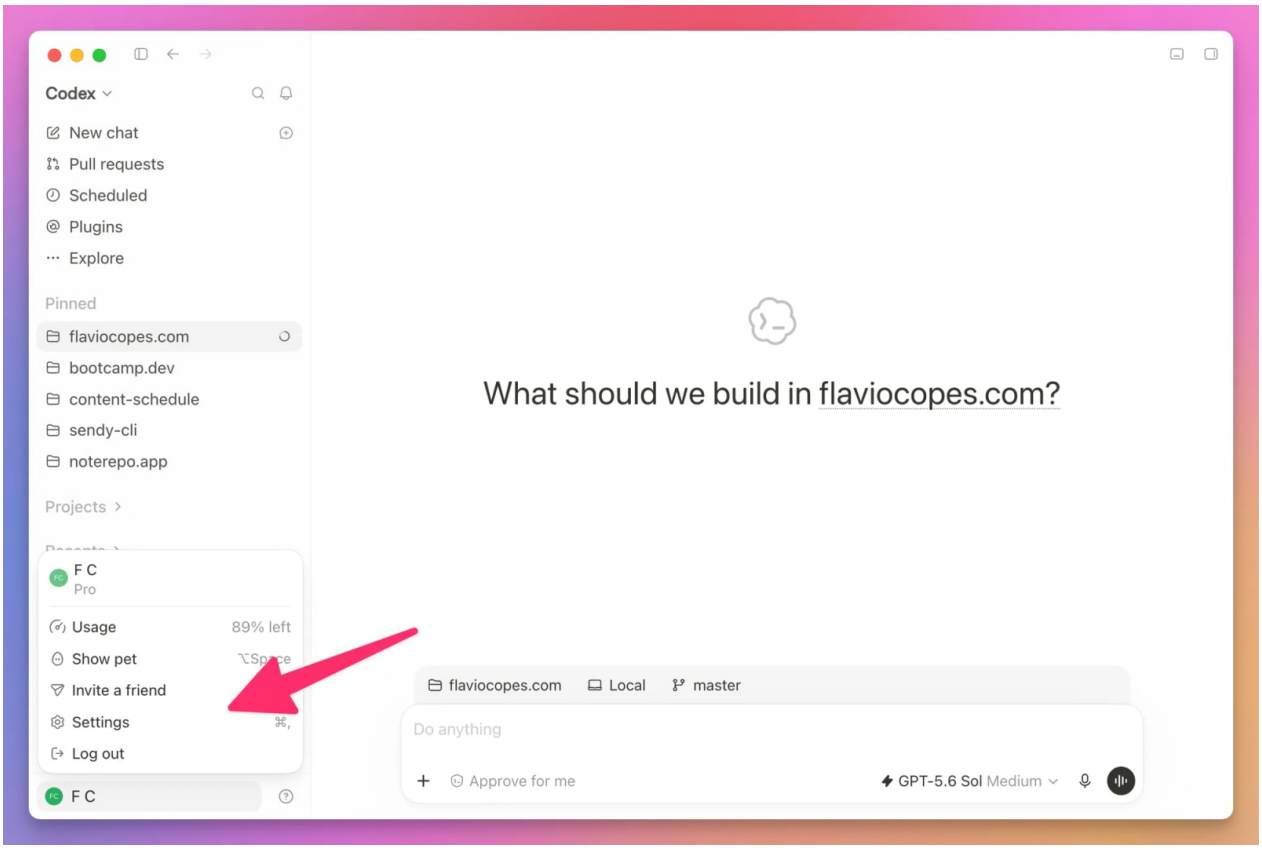
Model names change, so there isn't one model I can always recommend. Start with the default and switch when you have a reason. On paid plans, GPT-6.2 Sol is the everyday default for coding and review, while **GPT-6.2 Astra** is there for harder tasks that cross many files or involve architectural decisions. Astra uses more of your allowance, so test Sol first.

Use lower reasoning for small, straightforward changes, and save higher reasoning for debugging, migrations, architectural work, or tasks where several constraints interact.

A typo doesn't need the same reasoning as a database migration, and higher reasoning takes longer and uses more tokens.

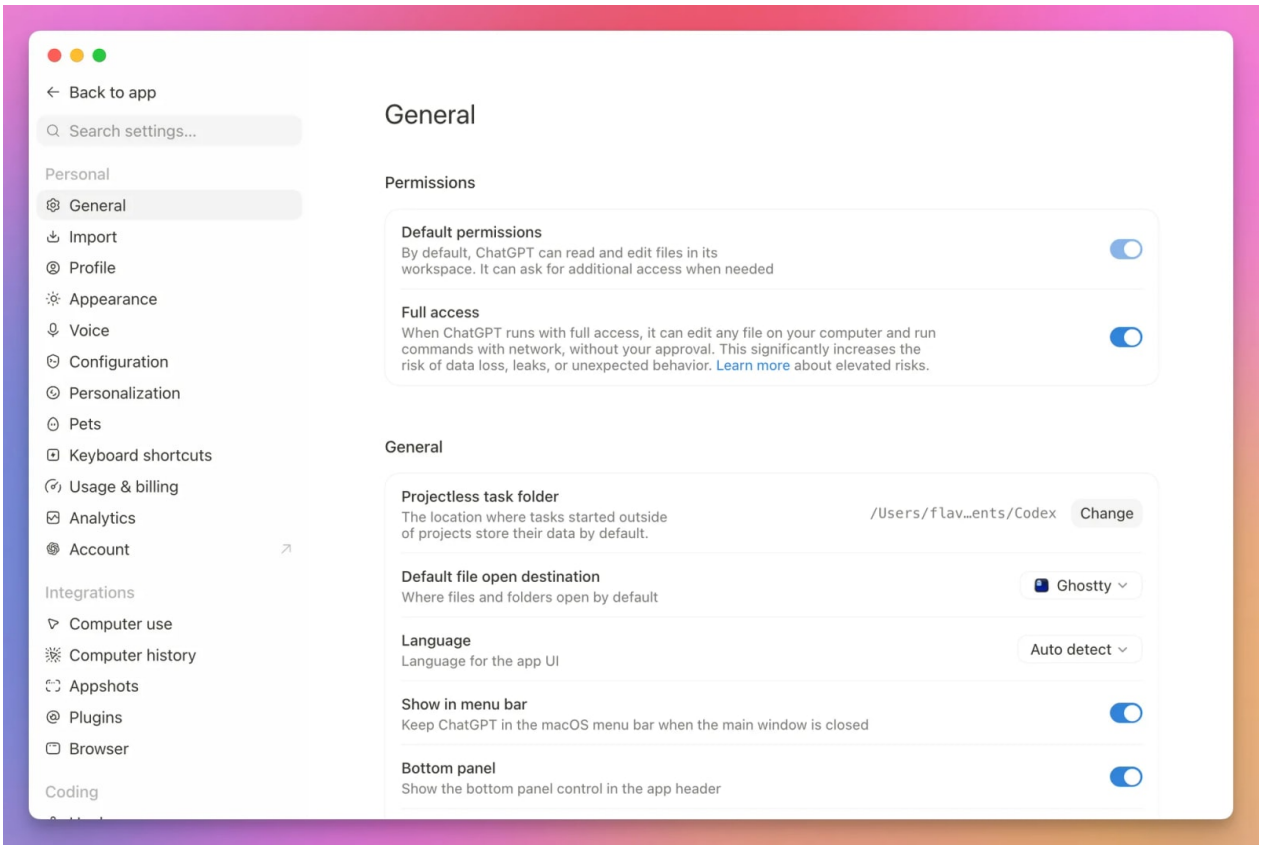
Settings worth checking

Open Settings from the sidebar or the application menu.

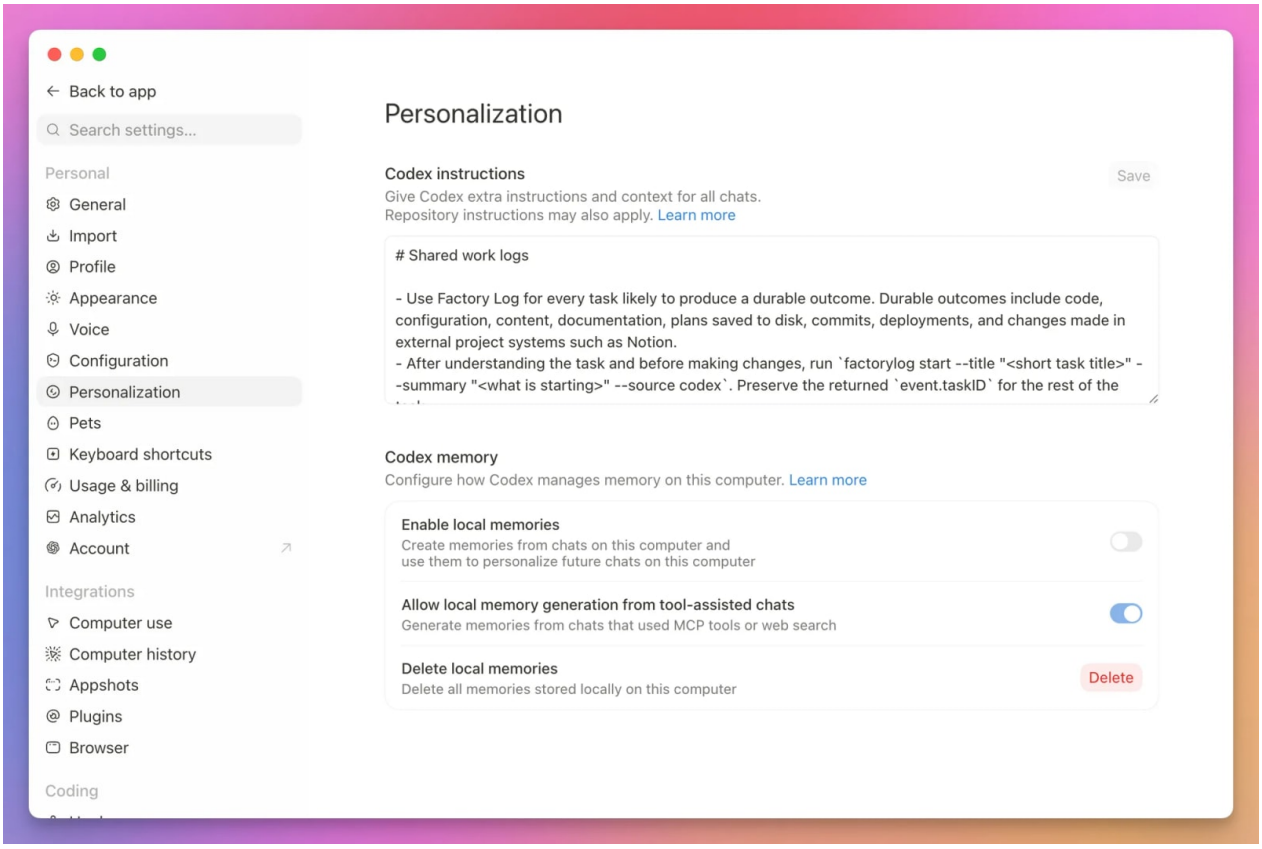


Before your first coding task, check the permission defaults and project settings.

In **General**, check the default permissions. Codex normally suggests workspace write for a Git repository, while an untrusted folder or one without version control can start read-only. Take a quick look at the selected mode before you send a task.

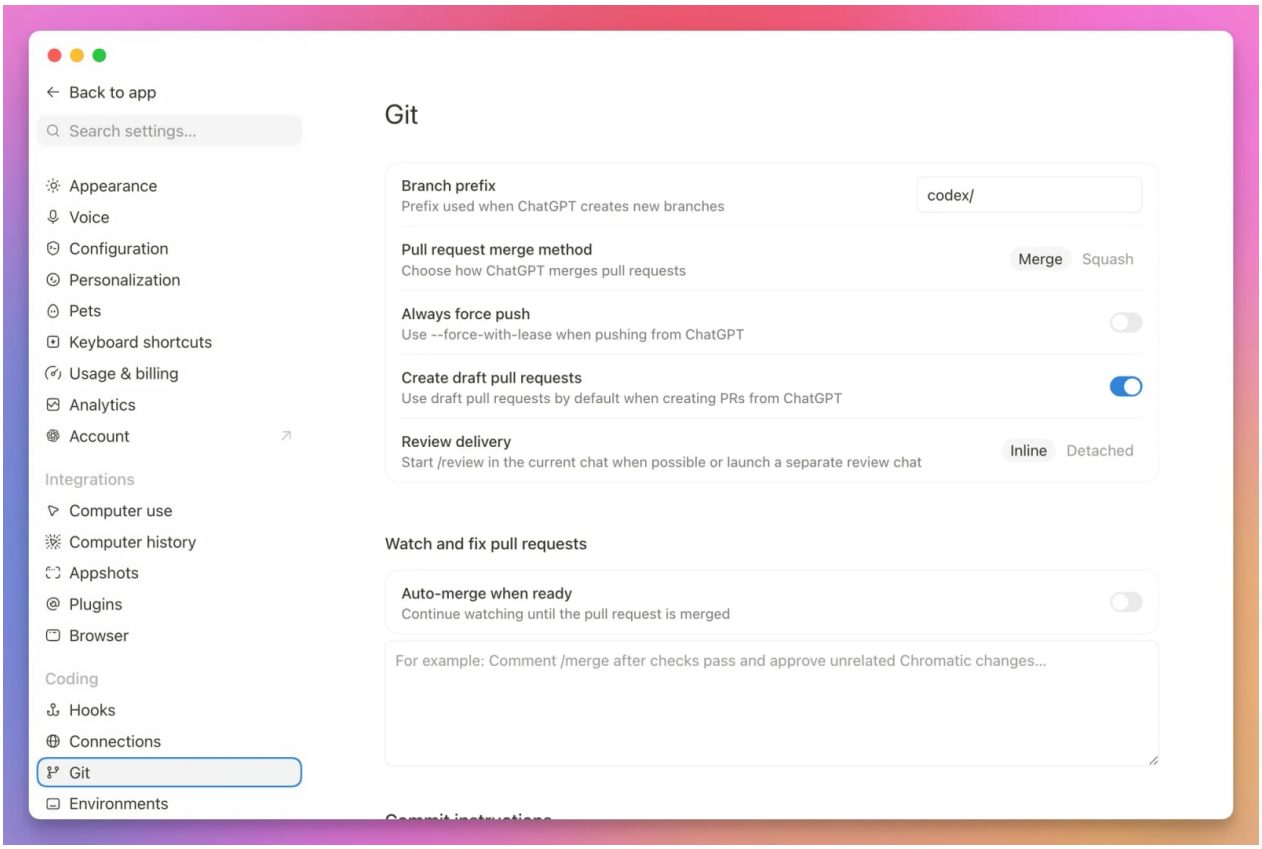


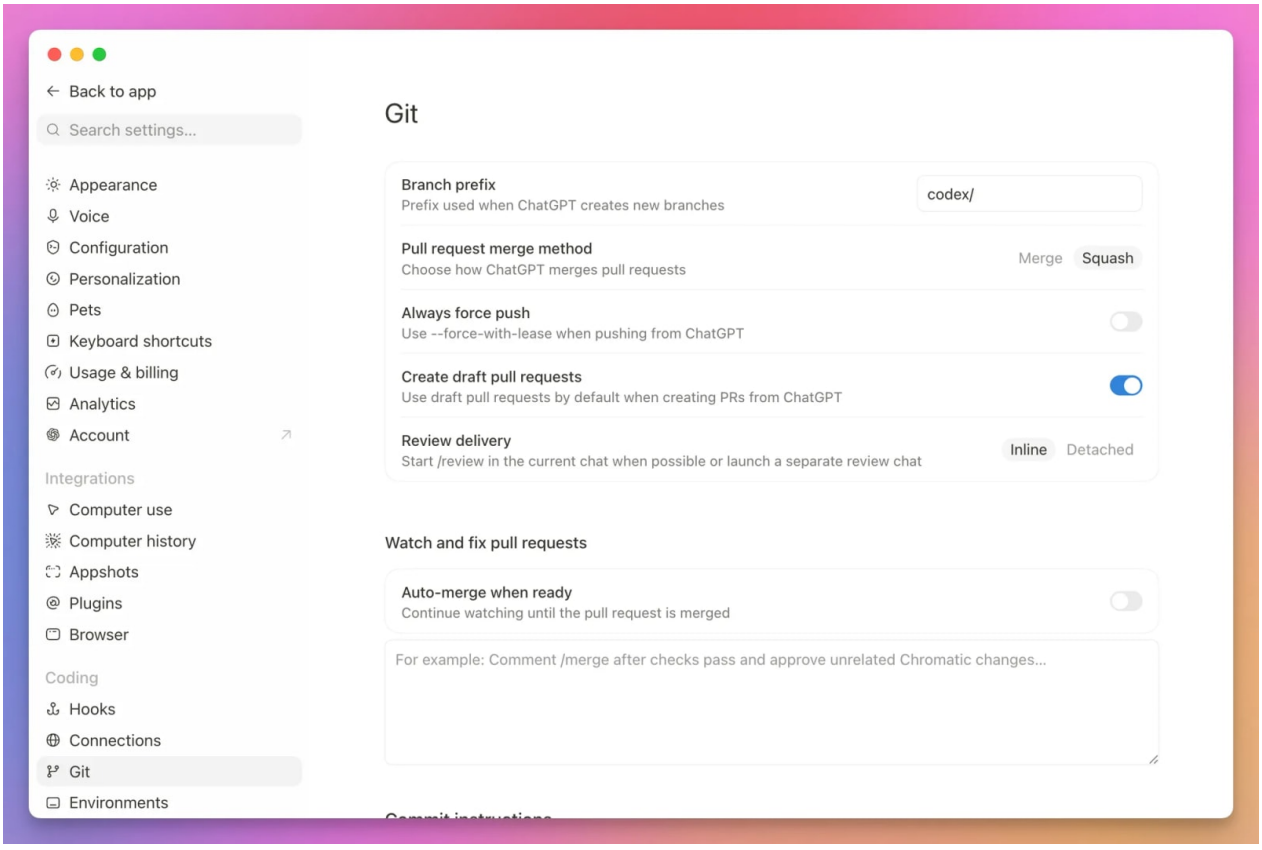
Next is **Personalization**, where you put your global instructions. These tell Codex how to communicate and work with you across every project.



In **Git**, you configure branch names, commits, and how changes come back into

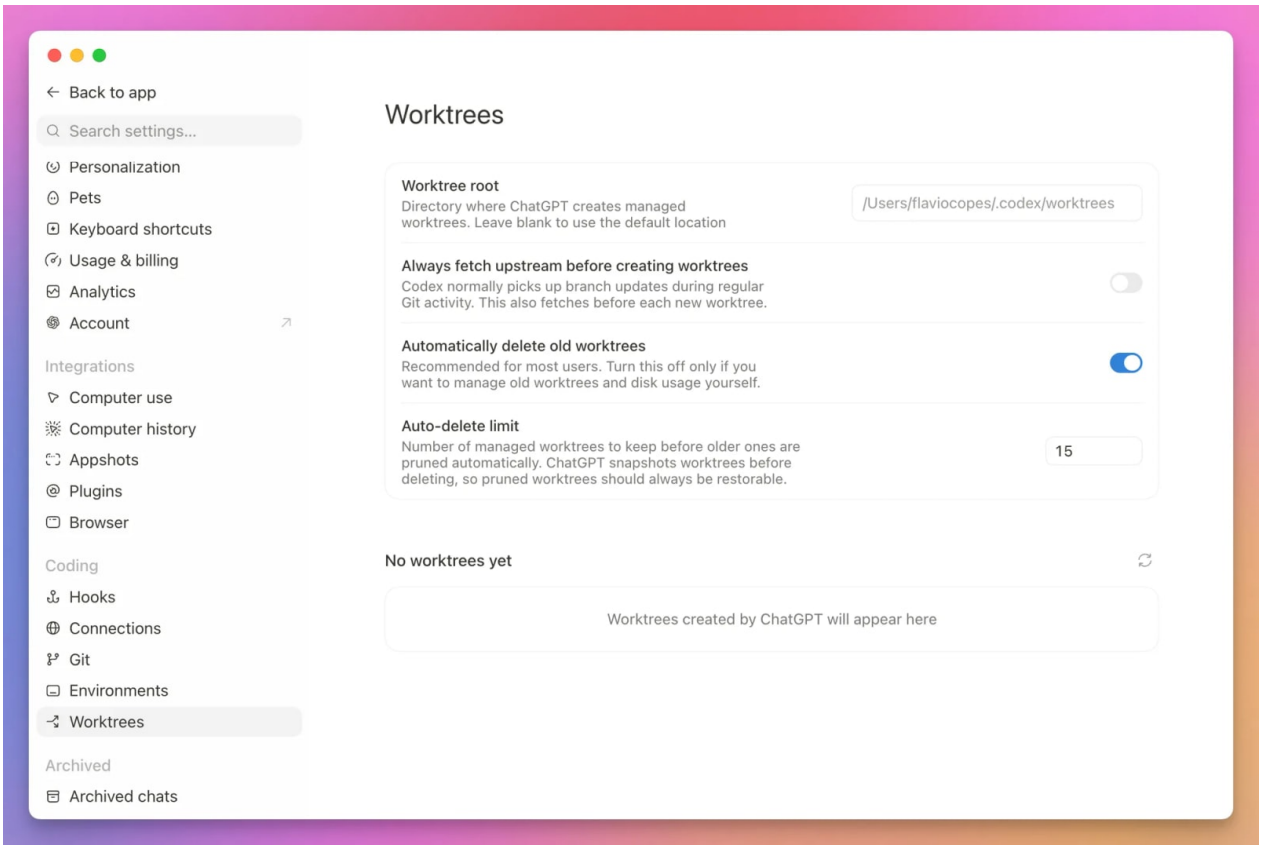
your checkout. The main choice here is whether pull requests use merge commits or squash commits. Match whatever your team already uses.





Under **Worktrees**, you can set where Codex creates its managed worktrees (`$CODEX_HOME/worktrees` by default) and how many it keeps around. Check the

cleanup settings before relying on a worktree for long-running work, and pin or preserve anything you still need.



You'll also find these settings:

- **Appearance**: theme, colors, and fonts.
- **Keyboard shortcuts**: change the keys used around the app.
- **Environments**: setup scripts and quick actions for a project. A worktree can run them to install its dependencies.
- **Plugins**: plugins, apps, MCP servers, and skills. We'll come back to it.
- **Browser**: the built-in browser and its browsing data.
- **Computer use**: the desktop apps Codex is allowed to use.
- **Archived chats**: restore a chat you archived.
- **Usage and billing**: your plan, credits, and current limits.

All the Codex interfaces share the same configuration layers. Your personal defaults live in `~/.codex/config.toml`, and a project can keep shared settings in `.codex/config.toml`.

Make a small project configuration

For the reading-time project, we only need local edits, approval when more access is needed, and cached web search. Create the configuration directory from the project root:

```
mkdir -p .codex
```

Then save this as `.codex/config.toml`:

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

That's the whole file for this exercise. If you already have a `config.toml`, update those values and keep the rest instead of replacing your existing file.

`workspace-write` keeps local command writes inside the workspace and other allowed roots, and `on-request` lets Codex ask for approval when an action needs more access. The web-search setting only controls the search tool, so it won't give arbitrary shell commands network access.

Codex only loads a project config after you trust the project, so check `.codex/` before trusting a repository someone else sent you. Your organization can also enforce requirements that this file can't override.

Settings in a trusted project override your personal defaults, and command-line overrides can take precedence for a single run. You can check configuration precedence for the full order.

Start a new CLI session, then inspect what took effect:

```
/debug-config
/status
```

`/debug-config` shows the active configuration layers and any enforced requirements, while `/status` shows the settings for the current session. If a value doesn't match what's in your file, check those layers before tweaking anything else.

A note in `AGENTS.md` asking Codex not to deploy is just guidance on how you want it to work. Permissions and the sandbox are what block actions from running, so keep both aligned with the task.

Understand permissions and the sandbox

For local commands, the **sandbox** controls filesystem and network access, and the **approval policy** decides when Codex has to ask first. Approval requests can go to you or to an automatic reviewer you've configured. Browser access, Computer Use, and MCP tools each have their own permission controls.

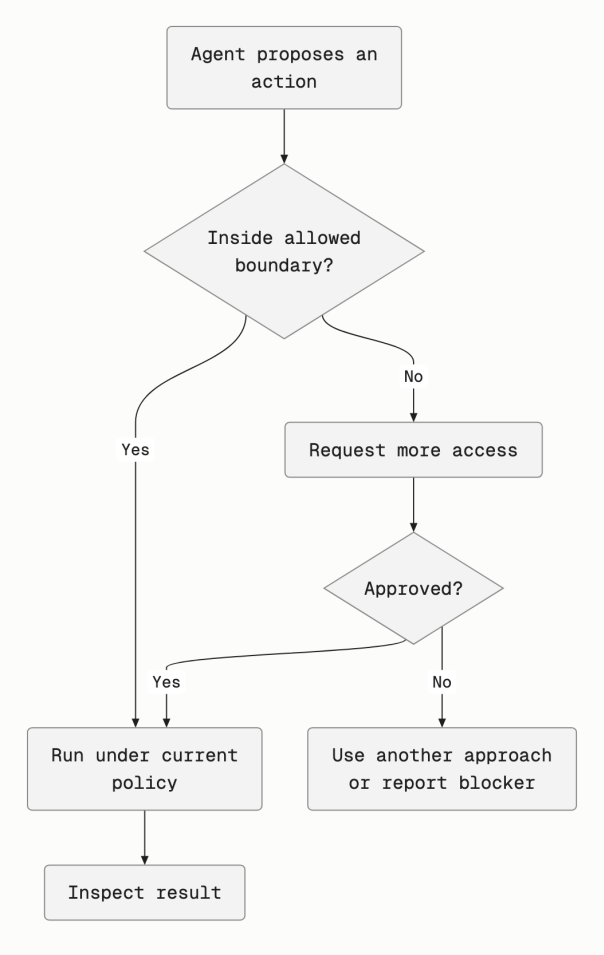
Workspace write lets Codex edit your project while blocking writes everywhere else. Even inside the project, sensitive paths like `.git` stay protected and can still require approval.

Installing a package might need network access, and Computer Use might ask to control an app. Always check what's being requested, especially if a command can affect production.

Read the command before approving it. Be especially careful with commands that

delete files, rewrite Git history, rotate secrets, change a production database, or spend money.

Goals and scheduled tasks keep the same permissions. Running longer doesn't give Codex extra access. Here's how the sandbox and approval rules fit together:



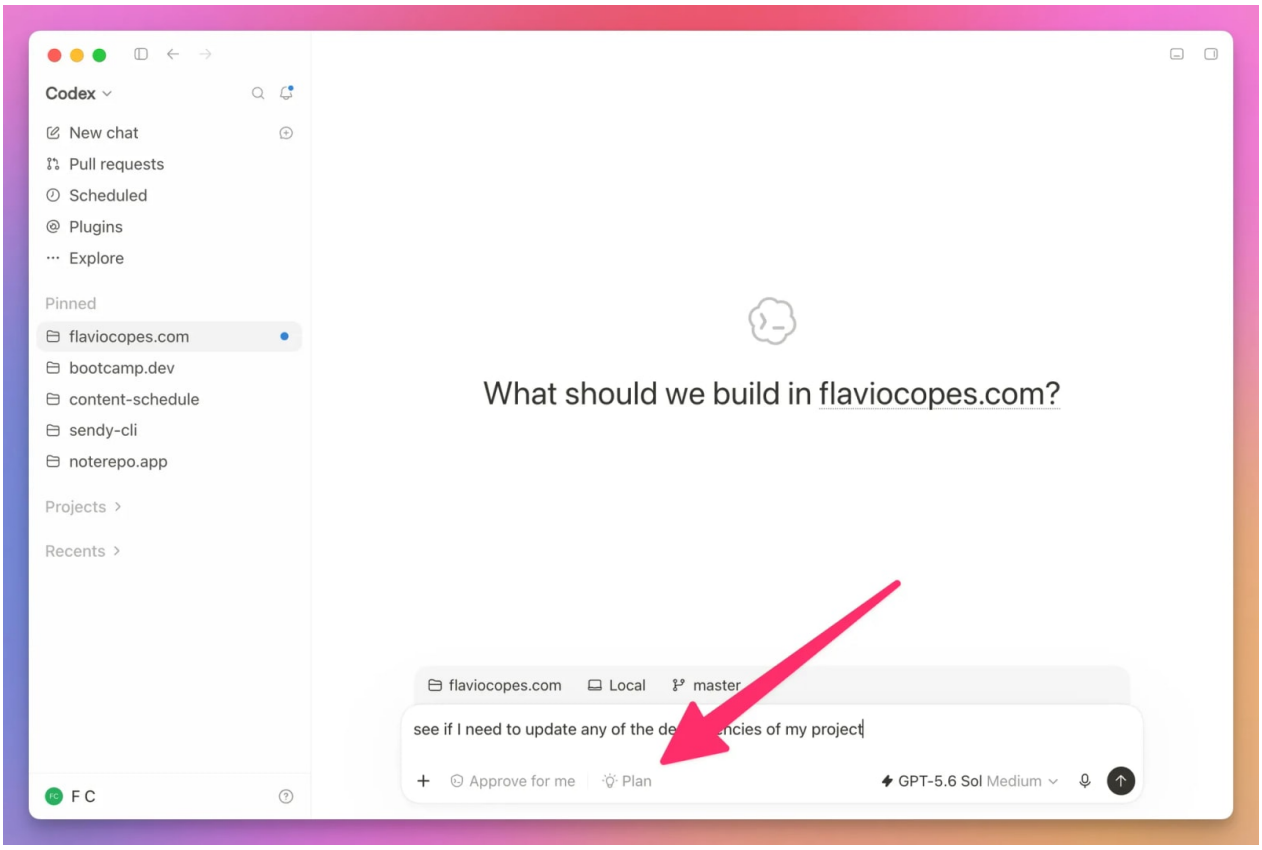
Use Plan mode before a difficult change

Plan mode lets Codex inspect the project and propose an approach before editing files.

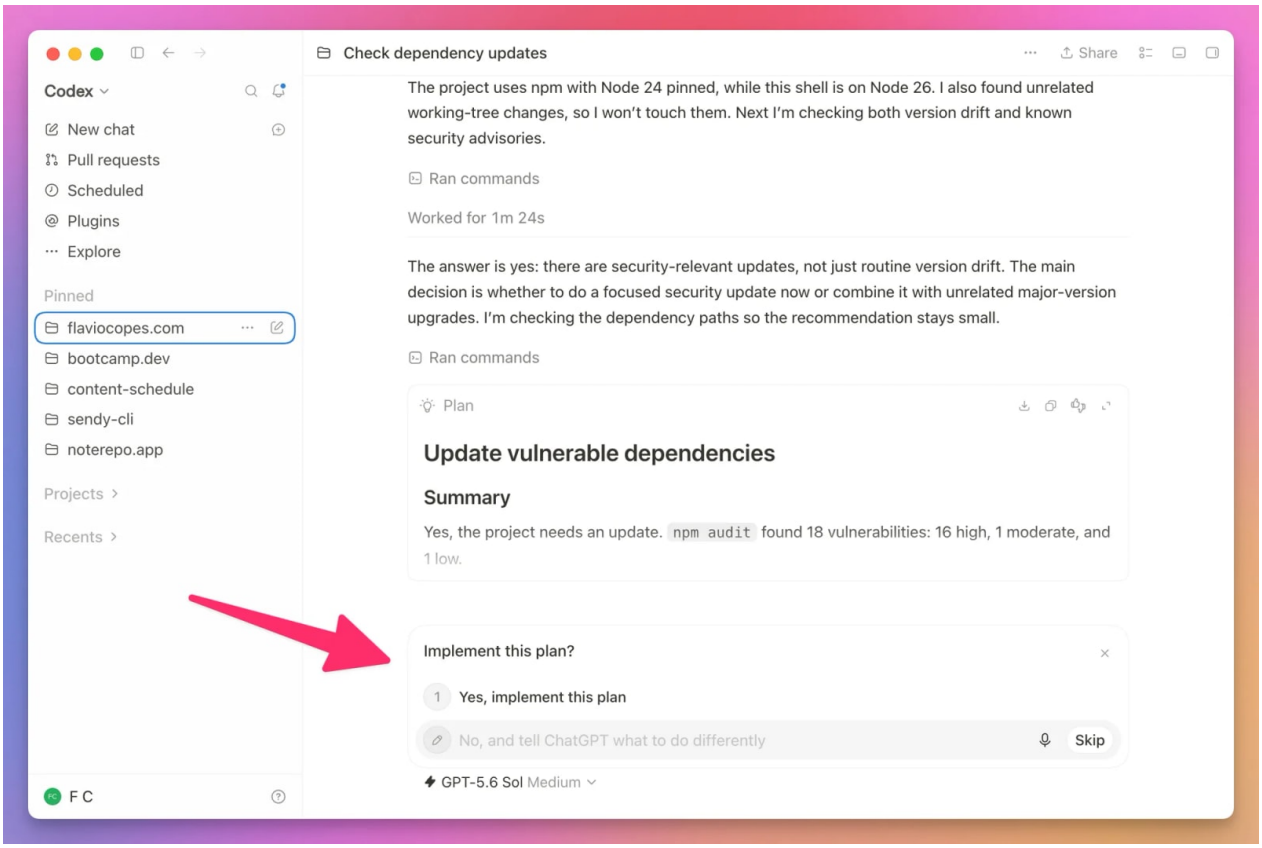
You can select **Plan** in the composer or type:

```
/plan
```

Here I selected Plan before asking Codex to check whether a project needed dependency updates:



When the plan is ready, Codex asks whether it should implement it.



Read through it and fix any wrong assumptions before approving implementation.

Ask for a plan when:

- you're not sure about the implementation
- the change crosses several parts of the project
- the order matters
- a wrong assumption would be expensive

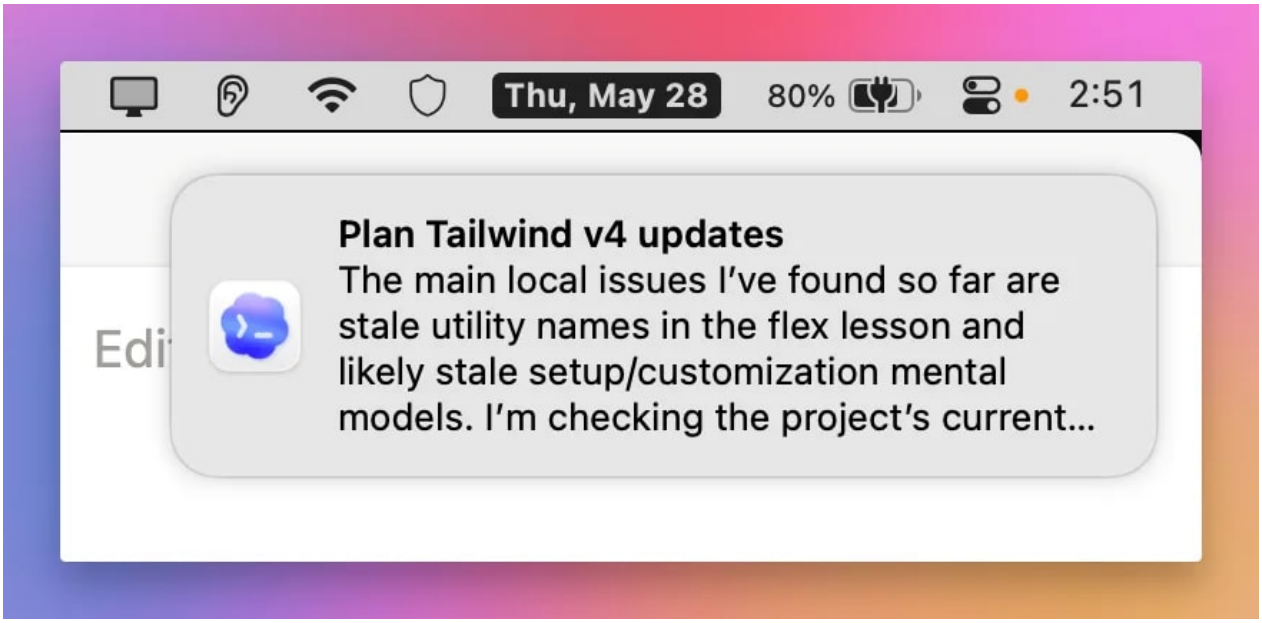
I skip Plan mode for tiny edits. For a larger change, I want to catch a wrong decision while it's still a plan.

Make the change

Once the plan looks right, tell Codex to implement it. You can say something like:

```
Implement the plan. Run the relevant tests and the production build.
Do not change files outside the areas listed in the plan.
```

Codex then reads the files, applies its patches, and runs the commands from the plan. You can switch to another chat or keep working on your own stuff, and the app will send a notification when Codex needs a decision or finishes.



You can also steer a running task. Send another message when you notice a missing constraint:

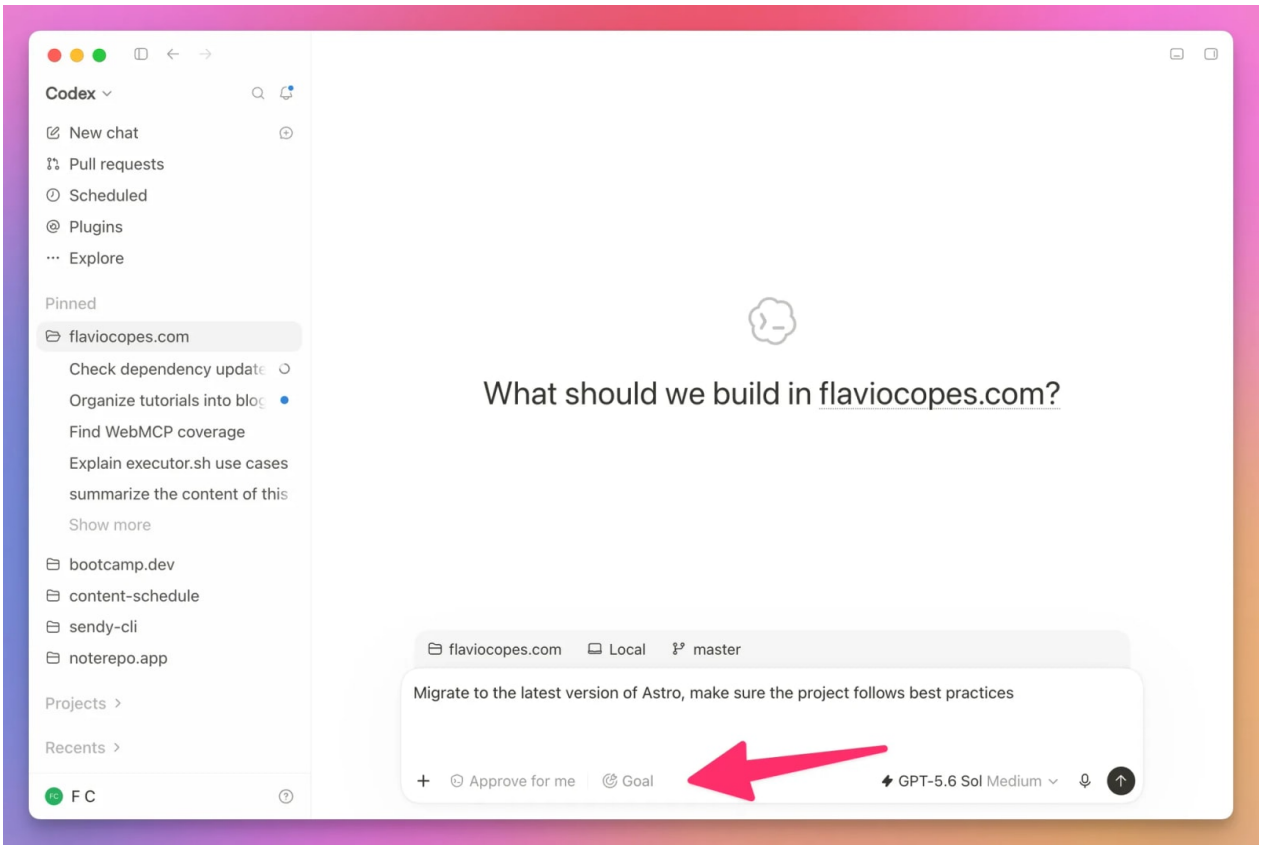
```
Keep the existing URL structure. Other sites link to these routes.
```

I do this often. I'd rather add the constraint right away than wait for the wrong change to finish.

Use Goal mode for long work

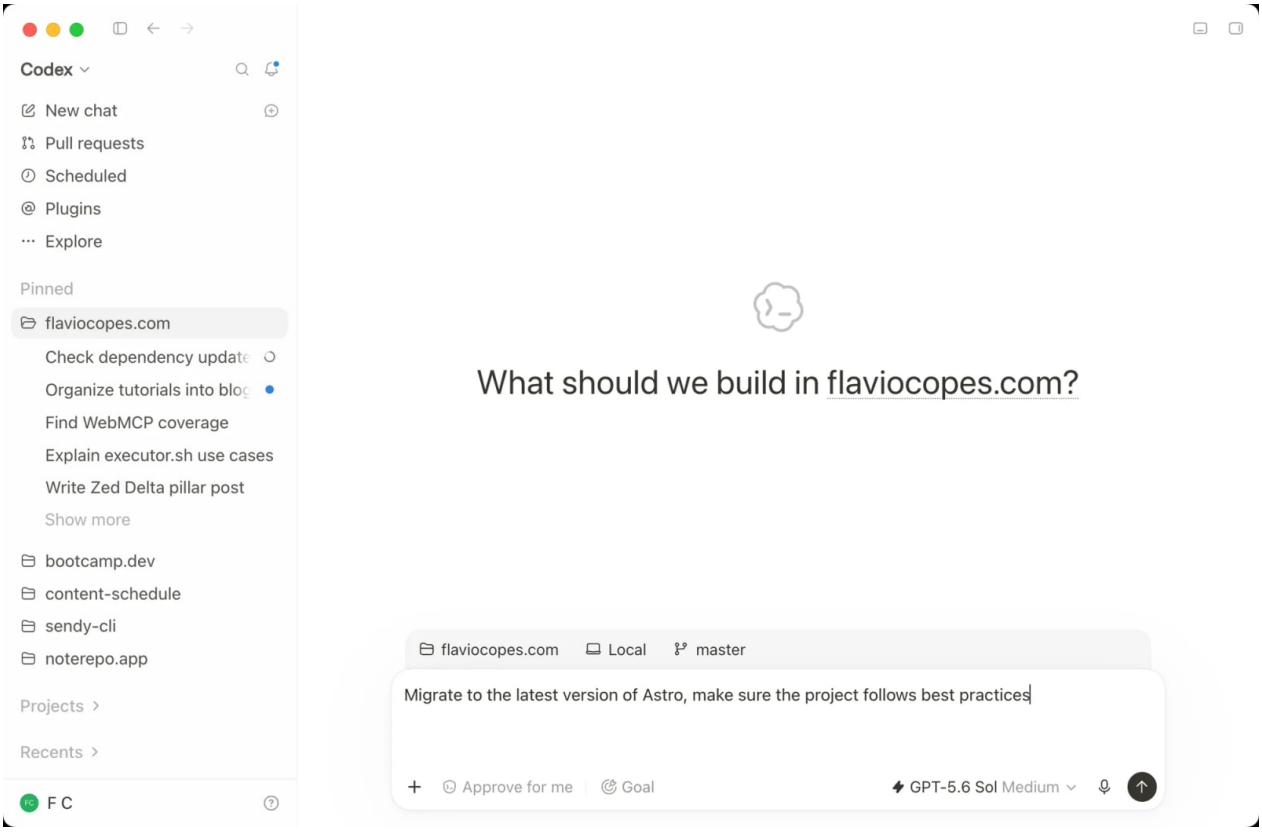
Plan mode maps out how to tackle one task, while Goal mode keeps Codex working toward a bigger result that can take much longer.

Click **Goal** in the composer to switch modes:



Then describe the result you want Codex to keep pursuing:

```
Migrate to the latest version of Astro and make sure the project follows best practices.
```



Tell Codex what counts as finished, what it must preserve, and which checks to run.

You can pause, resume, edit, or clear a goal. It keeps the same sandbox and permission boundaries as a normal task.

I tested Goal mode by asking Codex to recreate PocketBase with two different stacks. It ran for over 24 hours and produced two impressive prototypes, but both still needed a lot of manual work.

My preferred workflow is still one feature at a time, followed by a review. I use a long-running goal when the destination is precise enough for Codex to know when it's done.

Set a limit for an experiment

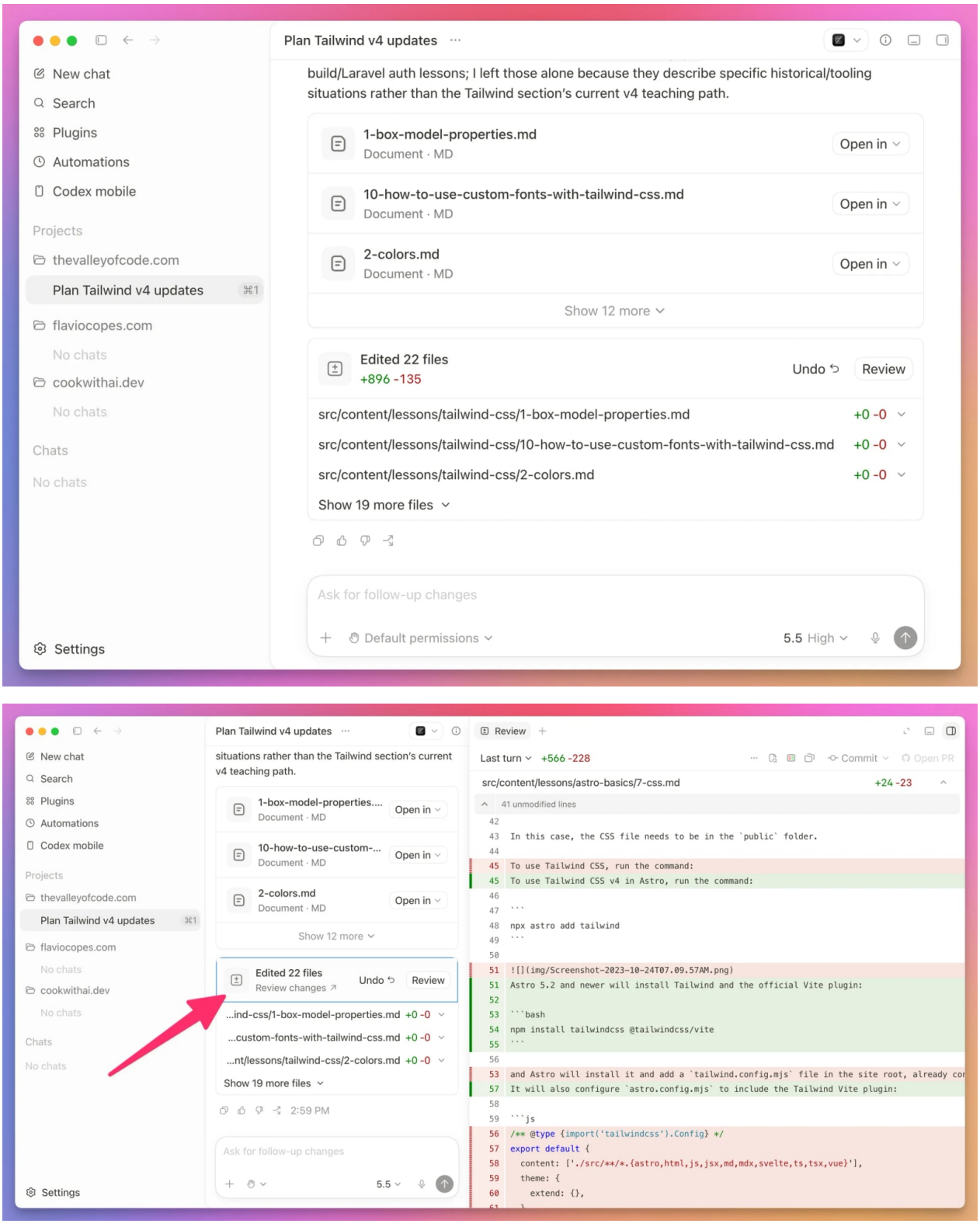
A long goal should have a small enough scope that you can decide whether it worked. For our calculator, you could ask:

```
Set a goal to finish fenced-code support in the reading-time project.
Preserve the one-minute minimum and the public function signature.
Finish when npm test passes, including a regression test that fails
against the original implementation. Then stop and summarize the diff for me.
Use a token budget of 200000. Do not commit or publish.
```

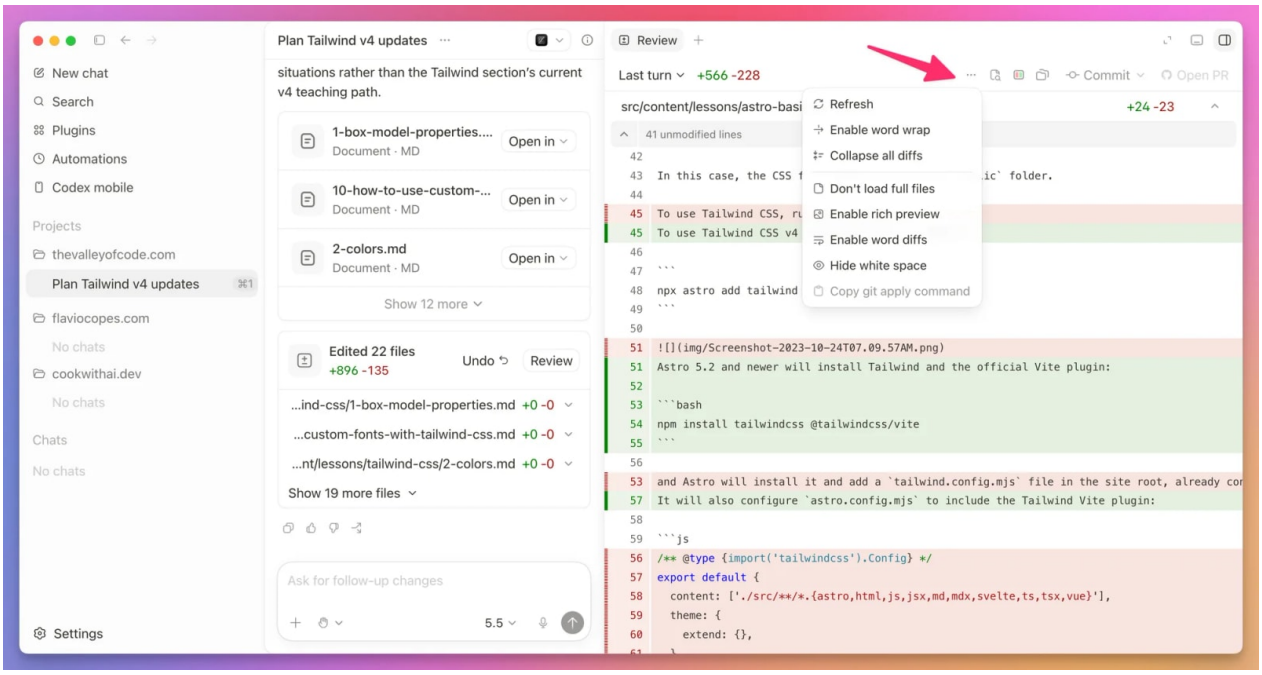
A budget only enforces a real limit if Codex saves it on the goal itself. If you don't see a budget listed on the goal, the number in your prompt is just an instruction. Keep an eye on `/usage` too, because the token budget doesn't cap what external tools or services cost.

Review every change

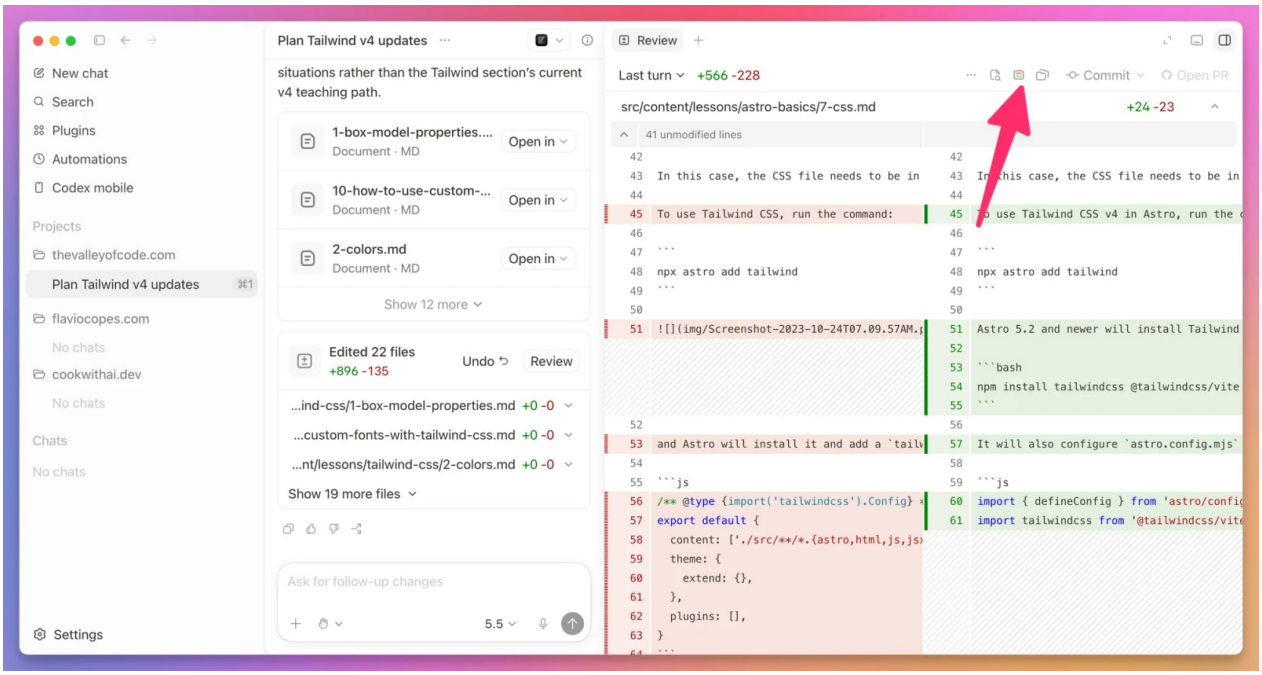
When Codex finishes, open the review panel. The green and red counters show how many lines changed. Click them to see the diff:



The review menu lets you switch how the changes are displayed.



I like the split view when I want to compare the old and new versions side by side.

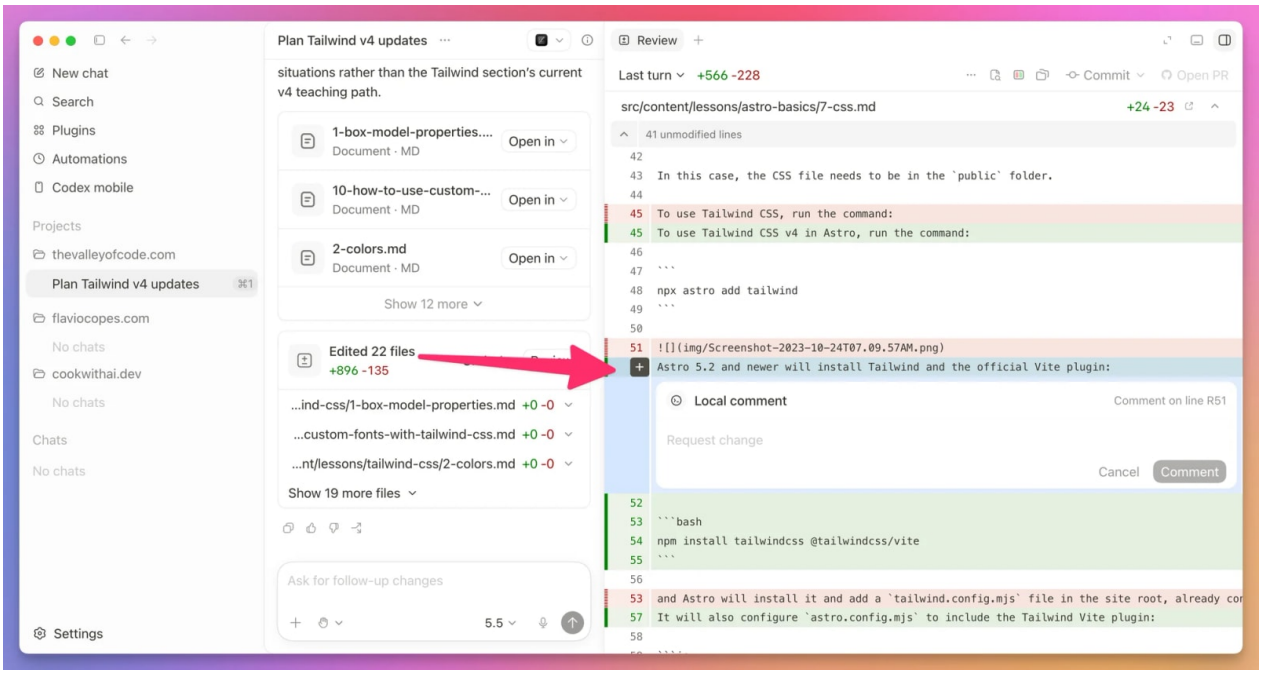


On Windows, Codex app v26.1120 fixed branch reviews so changed files and line counts stay accurate even in repositories with many files and long paths.

Read the diff before you commit. Check that:

- the change matches the request
- unrelated files were left alone
- errors and edge cases are handled
- tests cover the important behavior
- no secret or generated file slipped into Git

You can leave a comment on a specific line and ask Codex to address it:



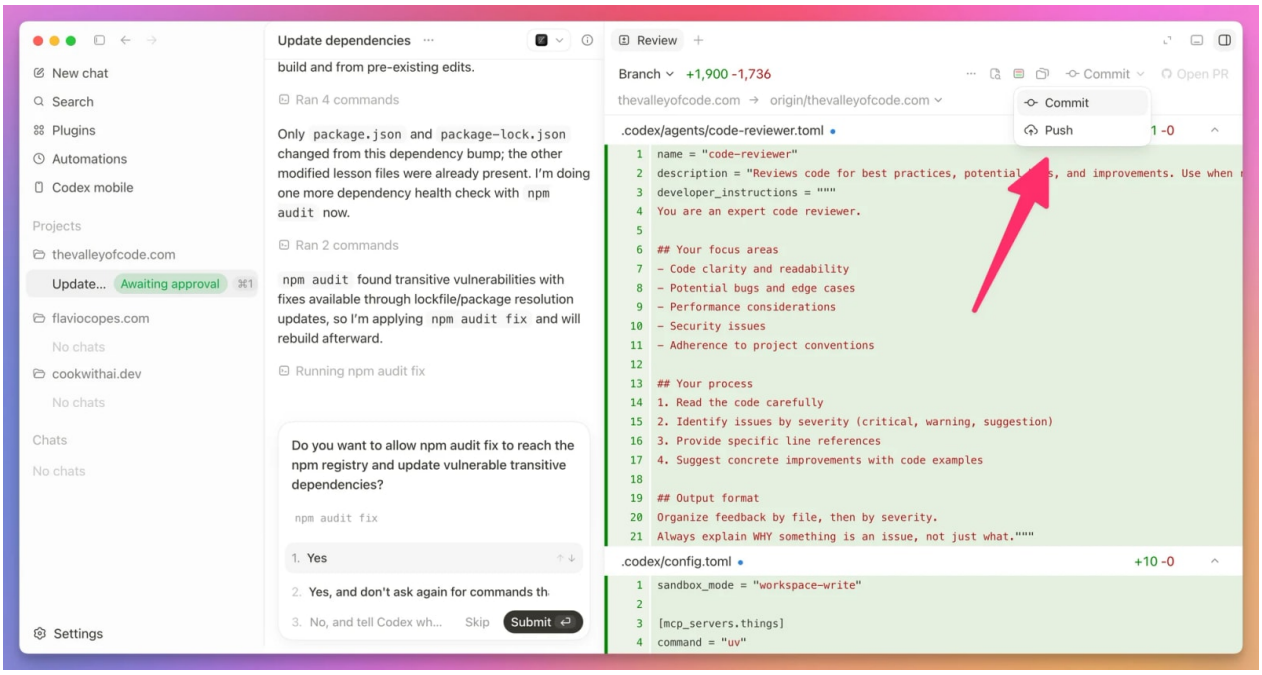
I often ask for one final check after the tests pass:

```
Review the full diff for regressions, unnecessary complexity, and anything
that does not match the request. Fix real issues, then run the checks again.
```

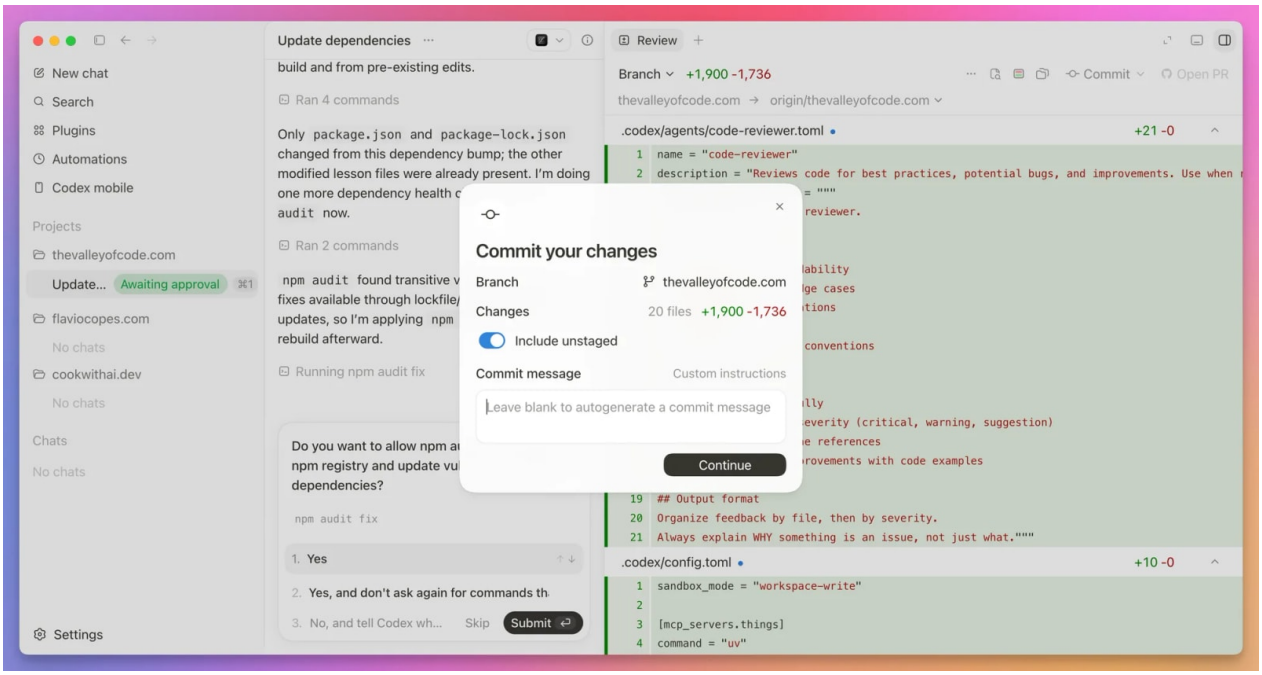
Codex has a built-in version of this. Type `/review` in the composer and choose whether to review against a base branch or the uncommitted changes. A separate reviewer reads that diff and lists the most important issues first, without changing your working tree. In **Settings > General > Code review** you can make it run in a detached chat, so the review doesn't fill up the chat that did the work.

Commit and push with Git

The desktop review panel can stage, revert, commit, push, and open a pull request.

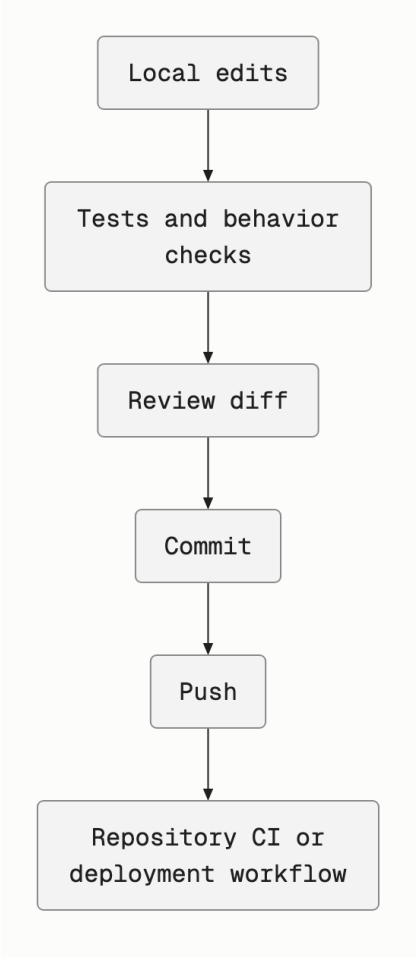


You can write the commit message yourself or let Codex propose one.



For a personal project, I often work directly in my normal checkout and commit after reviewing the result. On a team, use a branch and open a pull request so CI and other people can review the change.

Before committing, I run the checks and read the diff myself:

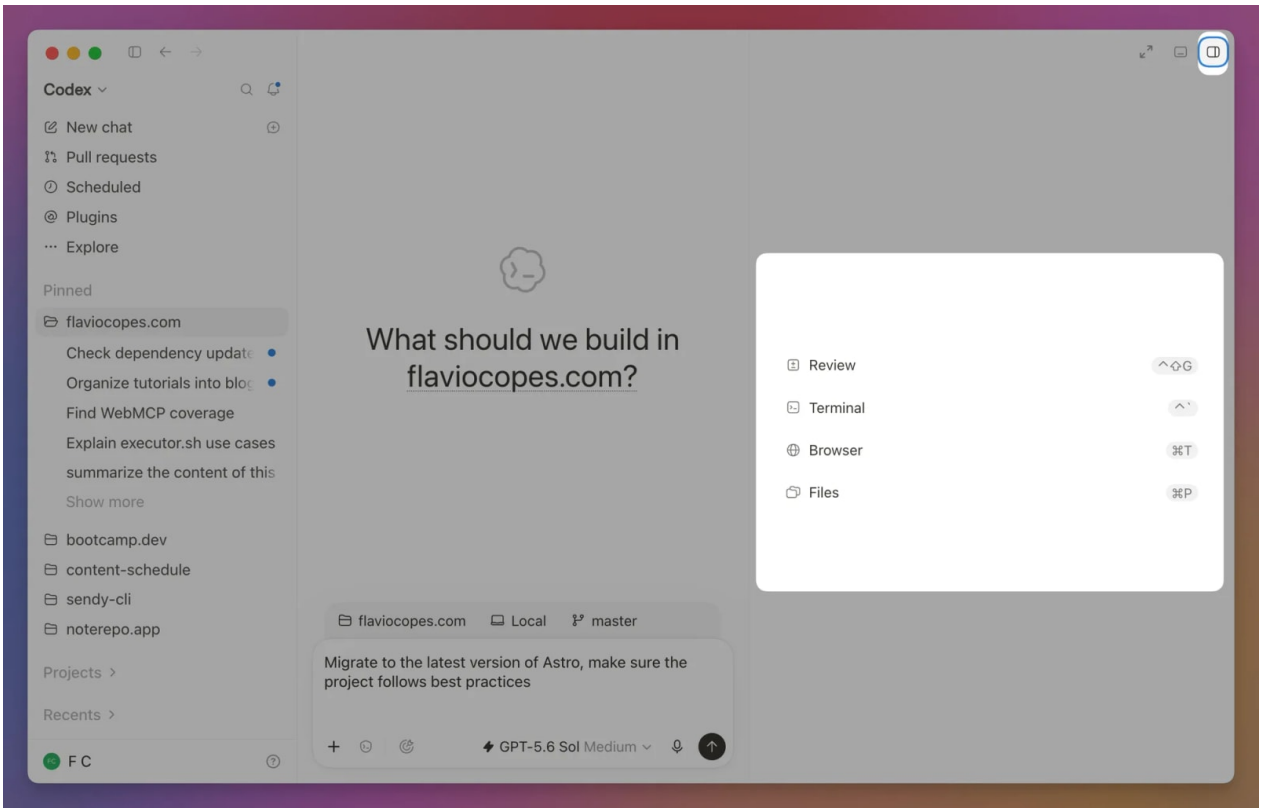


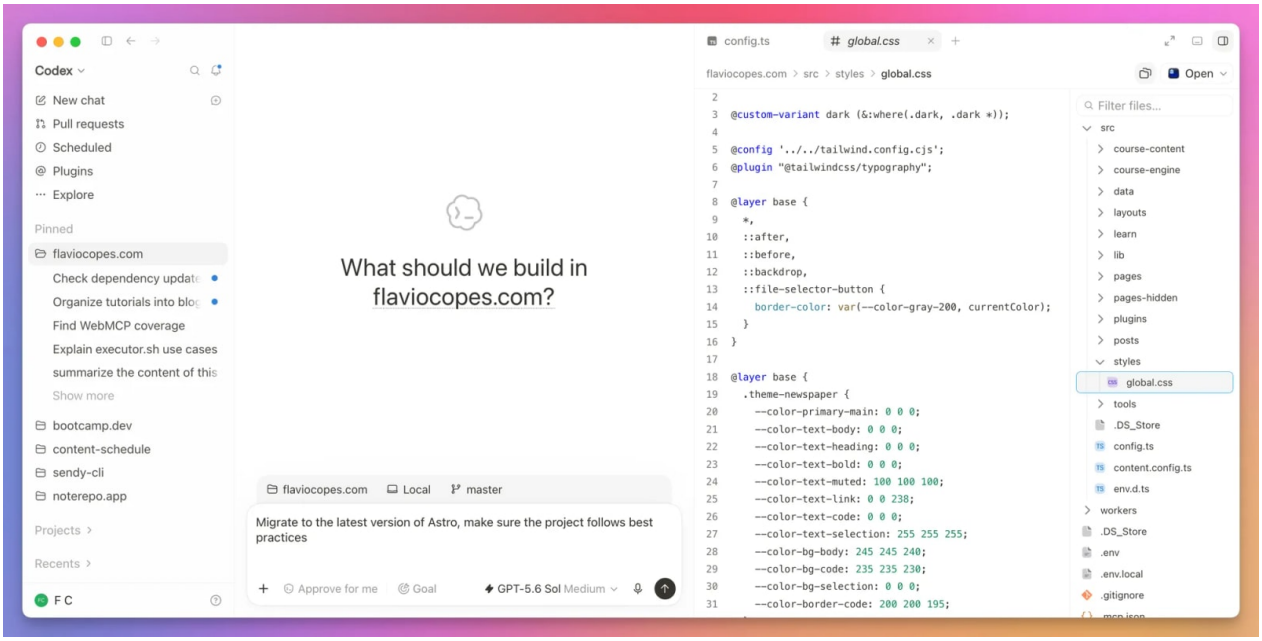
Keep in mind that a push can trigger a deployment if the repository is configured that way. Ask Codex to check the project workflow before treating a push like a harmless backup step. Our reading-time exercise has no deployment workflow, while a connected production app might.

Browse the project files

The desktop app has a file panel for quickly opening files without switching to an editor.

Open the panel menu and choose **Files**.





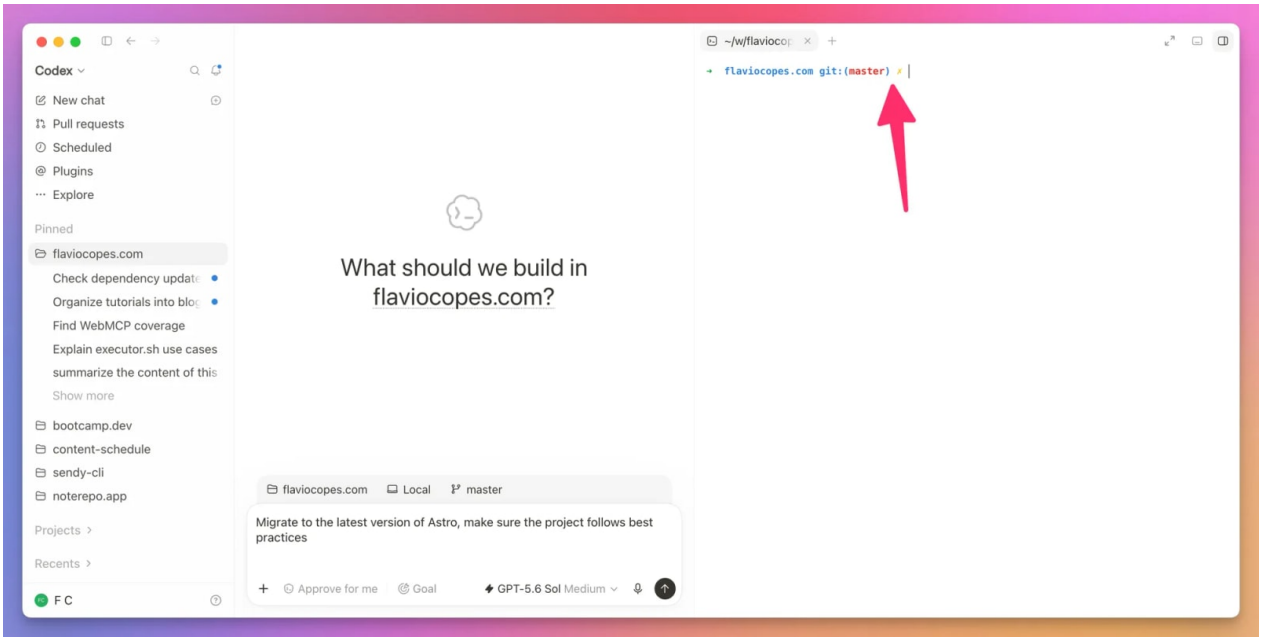
In Codex app v26.1120 and newer, you can open CSV files directly as a table preview and click **View source** to edit the raw file in the same tab. For Markdown files, the panel remembers whether you last used source or preview view when you switch between files.

I use this while reviewing an explanation or comparing a changed file with its neighbors. For a long editing session, I still prefer a full editor.

You can also attach a file from the prompt panel. Attach one when you already know it matters, or let Codex search the repository when you're not sure where to start.

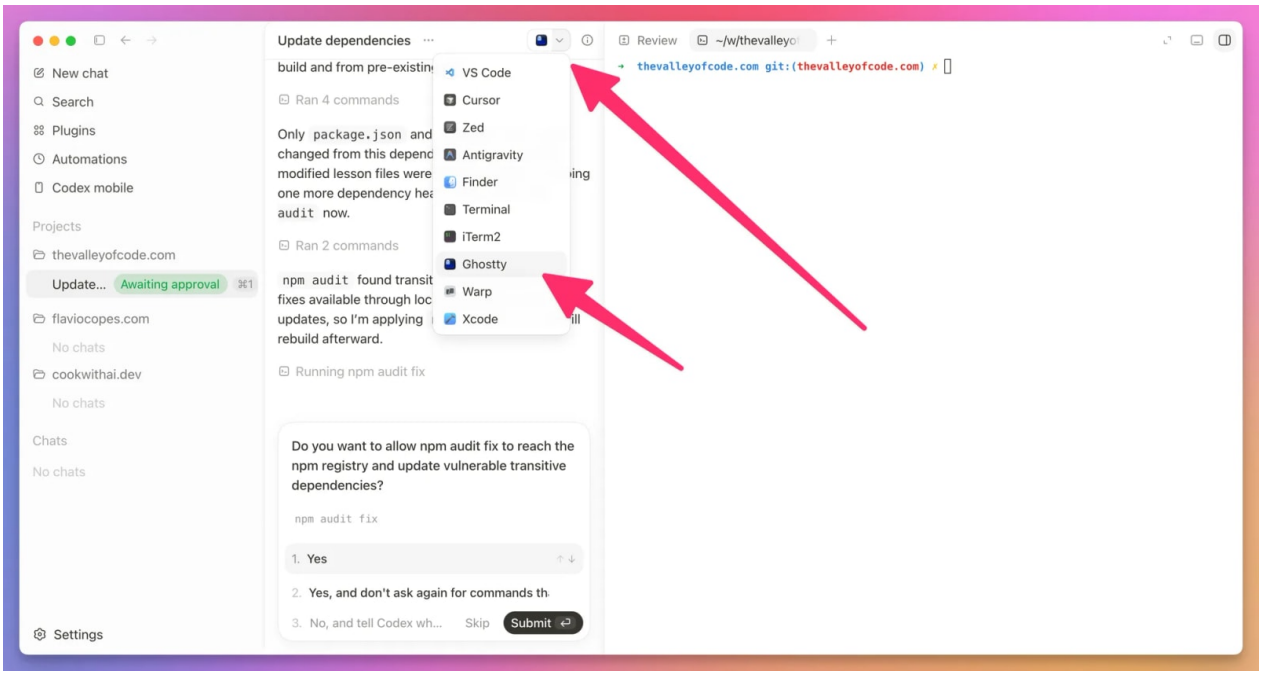
Use the integrated terminal

Every desktop chat has a terminal scoped to its project or worktree. Open the panel menu and choose **Terminal**.



Use it to start the development server, run Git commands, inspect logs, or try the result yourself.

When I want a separate terminal, I use the Open button at the top of the chat instead. It opens the project, or the worktree, in your editor or terminal app. Click the arrow next to it to pick one: VS Code, Cursor, Zed, Ghostty, iTerm2, Warp, and a few more. Mine defaults to Ghostty.



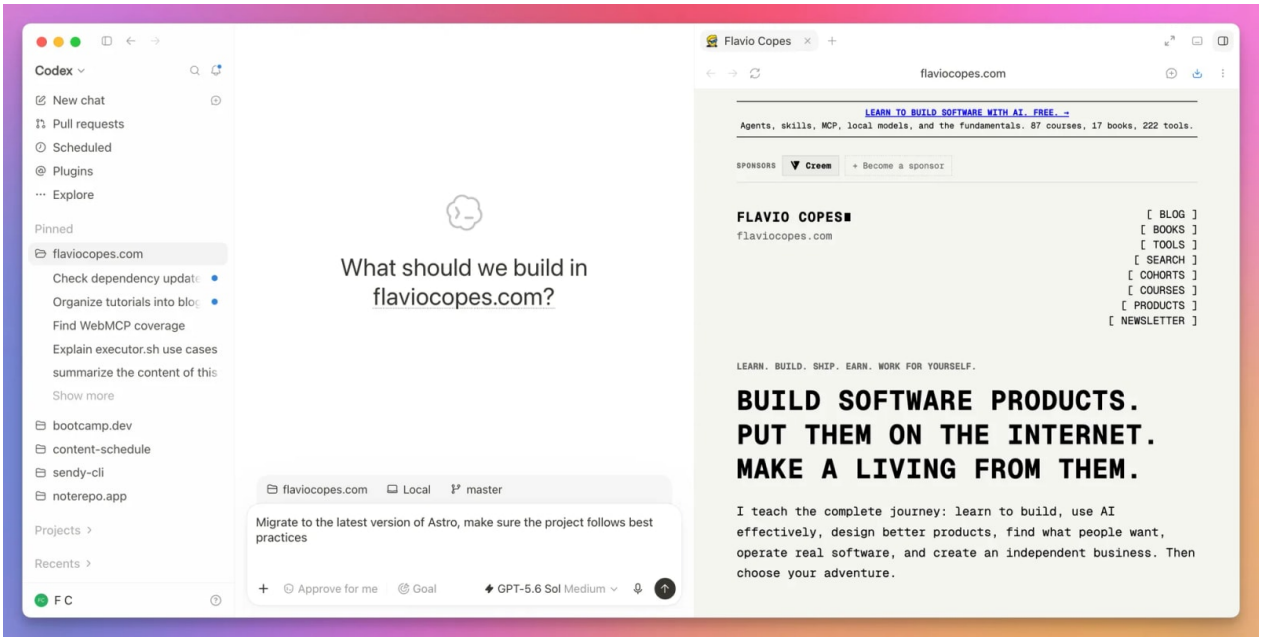
Codex can read the current terminal output. If a build fails, you can ask:

```
Read the terminal output, find the first real error, and fix it.
```

For a web app with a `dev` script, start `npm run dev`, leave it open, and let the agent use the browser against the local URL. Our reading-time exercise only has a test script, so it doesn't need a development server.

Test web apps in the built-in browser

The desktop app includes a browser. You can open it from the panel menu or mention `@Browser` in your prompt.

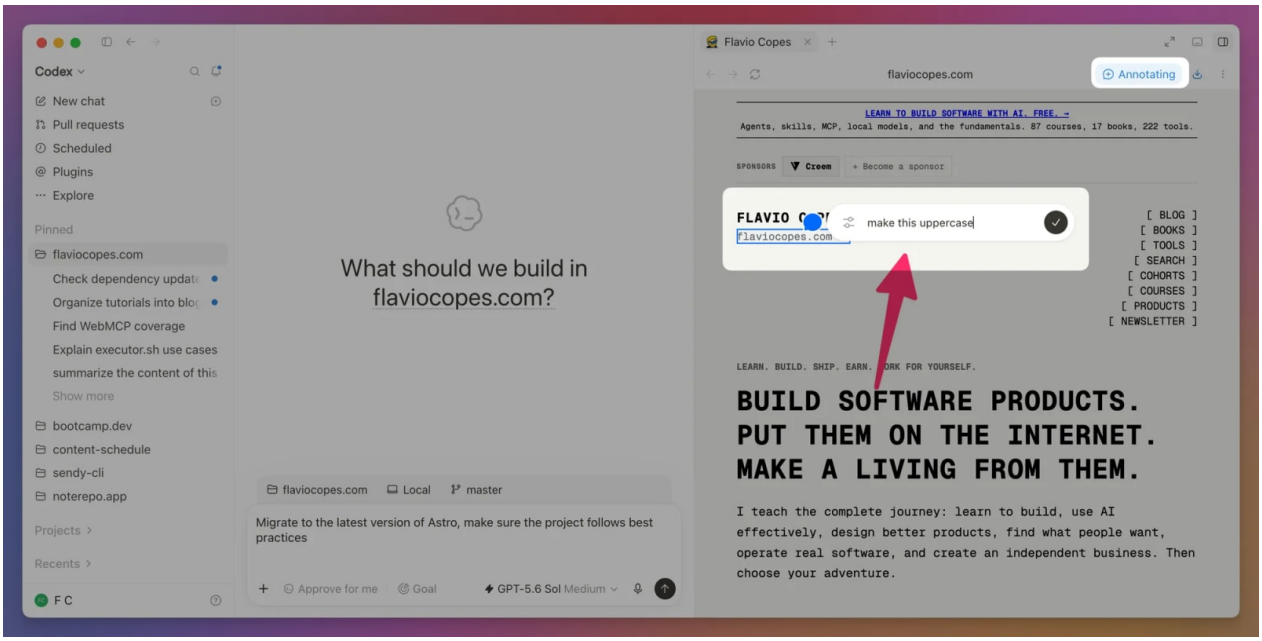


The built-in browser uses its own profile, so your open Chrome or Safari tabs, cookies, and logged-in sessions aren't carried over. When a task needs your real browser and its logins, the browser extension connects Codex to Chrome, Edge, Brave, Opera, or Vivaldi. Set it up under **Settings > Computer use**, then mention `@Chrome` in your prompt.

In a separate web project with a settings page, ask Codex to start the server and open the route:

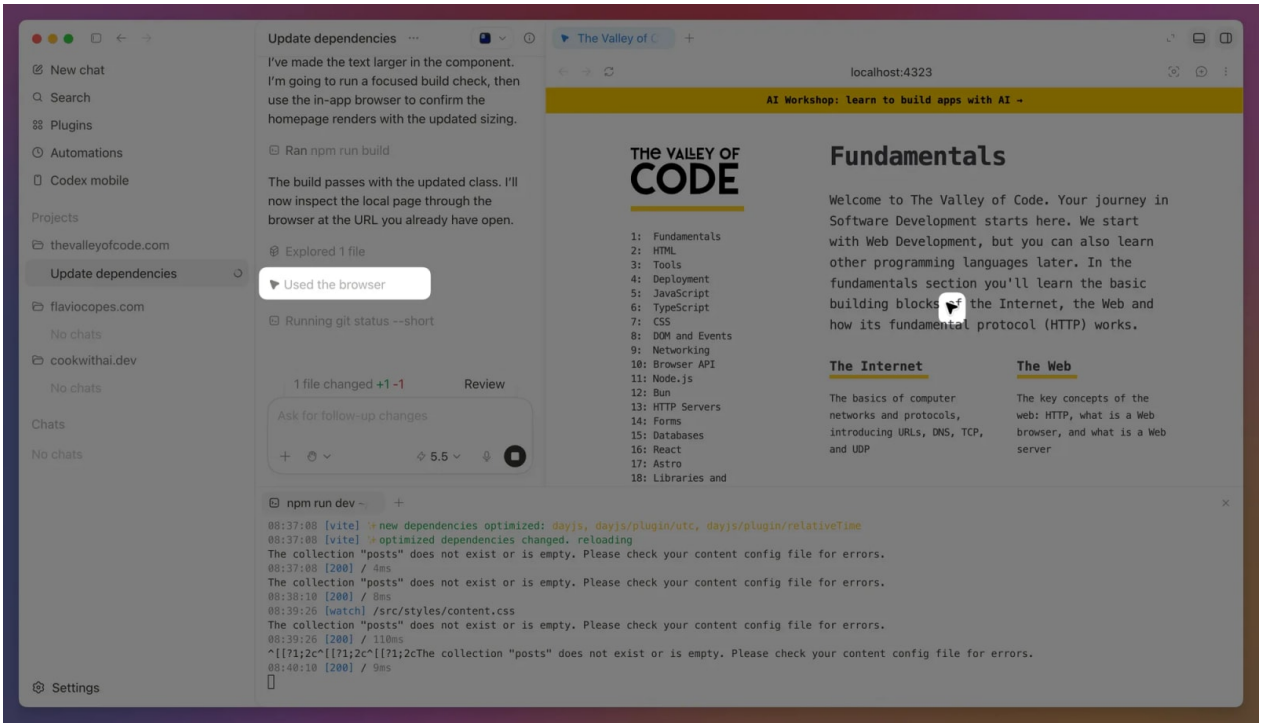
```
Start the development server, open http://localhost:4321/settings in @Browser,
and verify the form at desktop and mobile widths.
```

You can turn on annotation mode, click an element, and describe the exact change you want:



This is one of my favorite parts of the app. Instead of describing "the text block on the right under the second heading", I click it and write "make this font larger".

That selection goes straight to Codex with your prompt. Once it makes the edit, ask it to check the same element again on the running page.



If you turn on browser Developer mode, Codex can also inspect the DOM, console, network requests, and performance data.

I run one more browser check after the last edit. In the prompt I name the page, the state I want to test, and anything that shouldn't change.

Give Codex images and screenshots

Codex can inspect screenshots, mockups, diagrams, and error messages.

In the desktop app, attach an image and tell Codex what to look for:

```
Compare this screenshot with the current checkout page. Fix spacing and
typography only. Do not change the form behavior. Verify with @Browser.
```

Be specific about what matters in the image and what result you want.

Codex can also generate and edit images. Ask in plain language, or invoke the image generation skill with `$imagegen` if it's installed.

GPT-Image-2.6

On 4 November 2026, OpenAI brought GPT-Image-2.6 to Codex and the API with two models, Image-2.6 Flare and Sunburst. Both generate images 50% faster than before, and at medium or high quality Sunburst keeps details more consistent from one edit to the next.

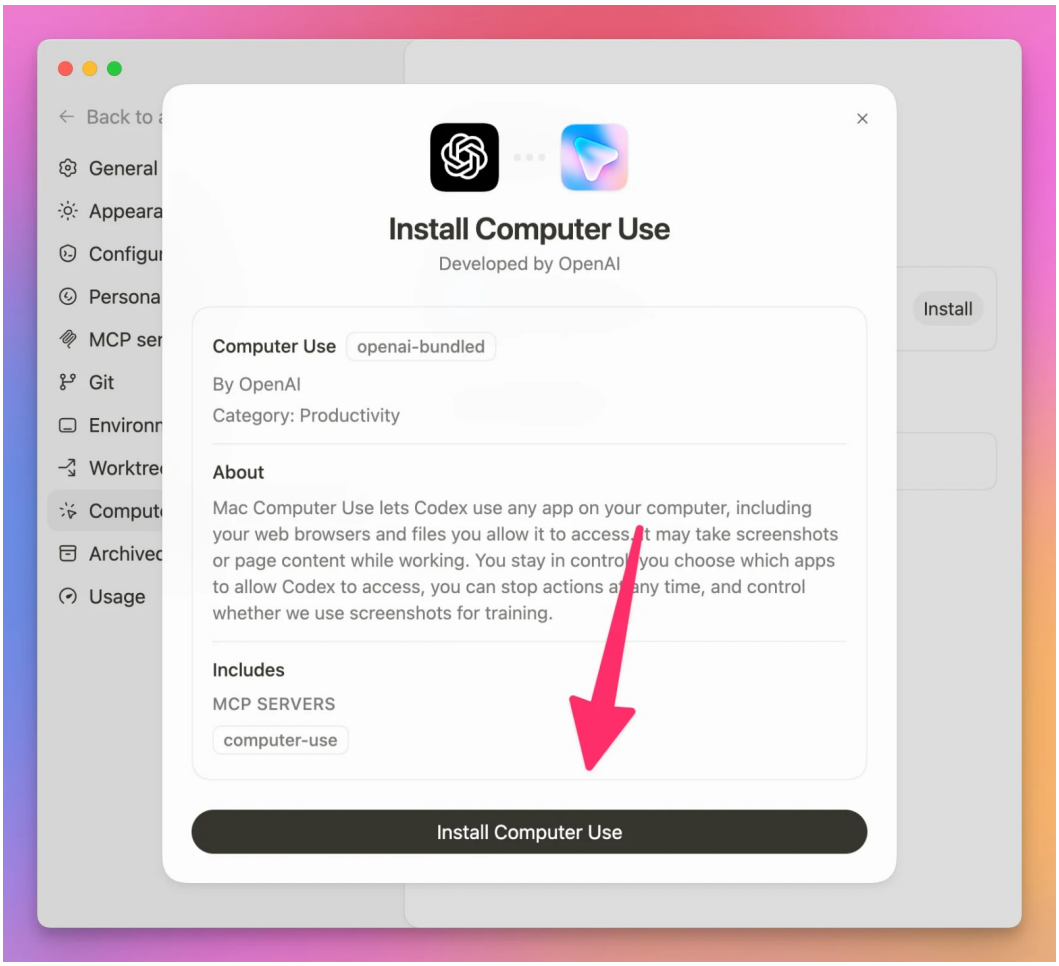
Image edits in Codex also happen right in the overlay now instead of opening a side panel, as shown in OpenAI's follow-up.

Use Computer Use for desktop apps

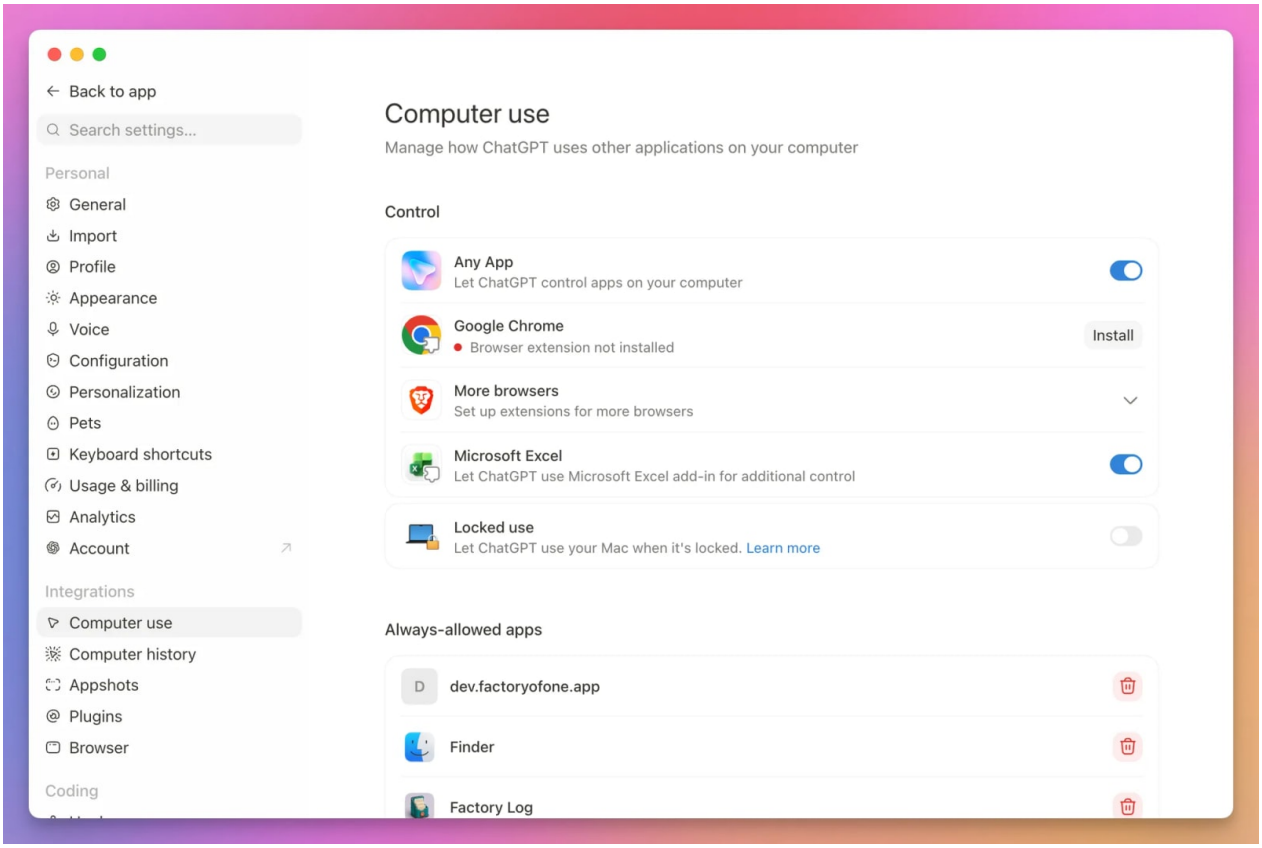
While the built-in browser handles web apps, Computer Use lets Codex work with desktop apps outside the browser. Cursor has its own version too; I wrote about how to use Cursor Computer Use.

In supported regions, Computer Use can operate macOS and Windows apps to reproduce a desktop-only bug, change settings, or work across several applications.

Computer Use ships as a plugin. Open **Plugins > Computer Use**, install it, and turn on its server and skill toggles. On macOS, grant Screen Recording and Accessibility when the system asks for them.



In **Settings > Computer use**, you can review which apps Codex can access and revoke any app you allowed earlier.



To trigger it, mention `@Computer` or the app name in your prompt:

```
Open the app with Computer Use, reproduce the onboarding bug, and fix the
smallest code path that causes it. Run the same UI flow after each change.
```

Because Computer Use can touch apps outside your project folder, I keep these tasks narrow and read every permission prompt. On Windows, it uses foreground input, so you can't keep using the desktop while it works.

If a service already has a dedicated plugin or MCP server, use that instead. Save Computer Use for tasks that depend on what's on screen. Codex's plugin is the built-in option, and when you want the same kind of desktop control from any agent that can run a shell, agent-desktop reads the accessibility tree and acts on refs.

Connect external tools with MCP

The context Codex needs isn't always in your repository. Sometimes it's in GitHub, Slack, Notion, Google Drive, a database, or an internal service.

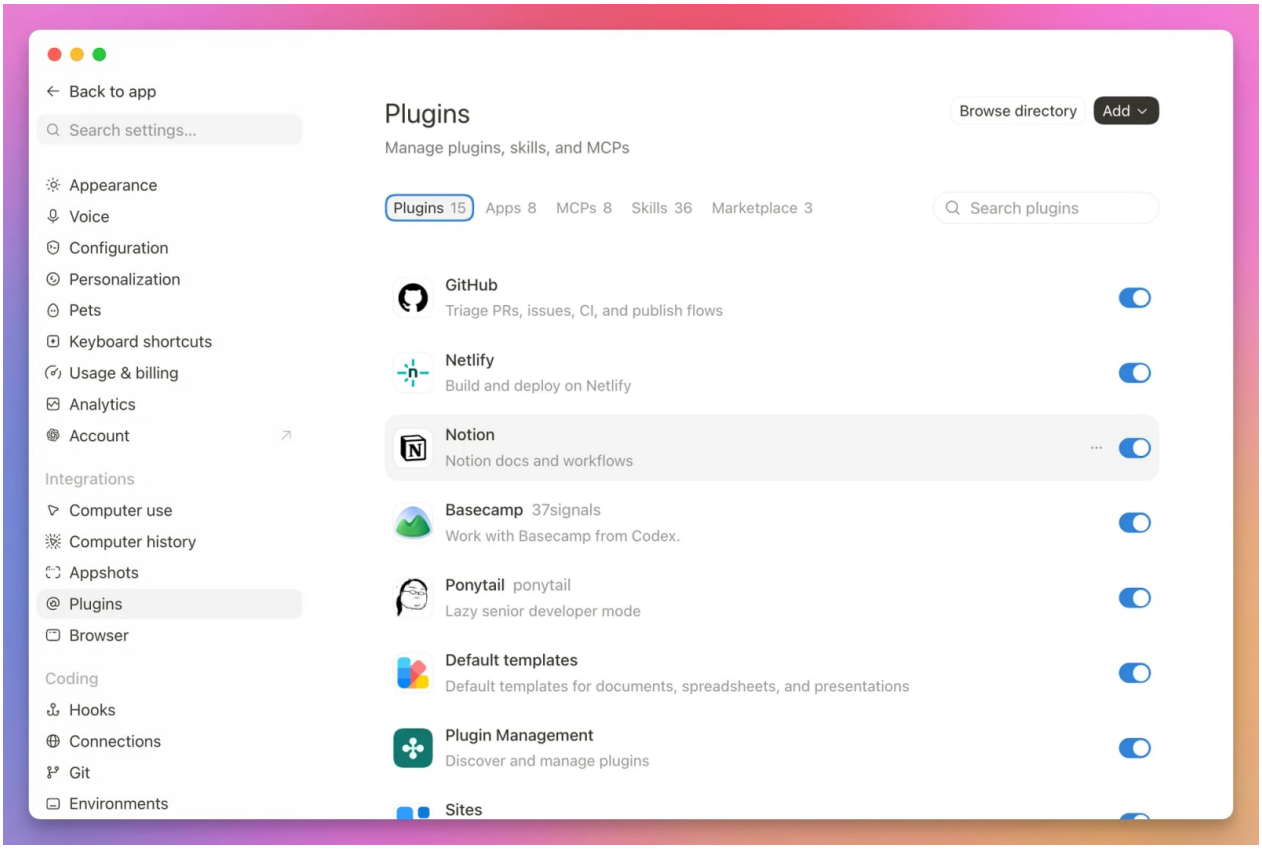
MCP stands for Model Context Protocol, and it gives Codex structured tools for working with those systems.

An MCP server defines what Codex can read or do. It can expose tools such as:

- list open pull requests
- read an issue
- query deployment logs

- create a task
- fetch a document

In the desktop app, open **Settings > Plugins > MCPs** to see what's connected and add a server.



Connect a service when you have a task for it, such as reading a failed deployment's logs. Here are the MCP servers I actually use and how they work in Codex.

Connect the OpenAI documentation server

We can test a real connection right now to check how Codex discovers the skill we're about to create. OpenAI's Docs MCP server searches and fetches official documentation. It's read-only and doesn't call the OpenAI API on your behalf.

It's a public server, so it needs no API key or OAuth sign-in. Add it from your terminal:

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

That command saves a user-level connection in `~/.codex/config.toml`, making it available in other projects too. Leave any existing entries in that file alone.

If you only want this connection in the reading-time project, skip the add command and put this table in the project's `.codex/config.toml` instead:

```
[mcp_servers.openaiDeveloperDocs]
```

```
url = "https://developers.openai.com/mcp"
```

Pick one scope for this exercise so you know where the connection lives. A project entry only loads for a trusted project, and any other machine running the repo will still need network access to the server.

Restart the client or connection if needed, then run `/mcp`. Check that `openaiDeveloperDocs` appears along with its documentation search and fetch tools, since seeing the server name alone only confirms the config entry is there.

Next, ask for a narrow lookup:

```
Use the openaiDeveloperDocs MCP server to find the current Codex skill-discovery docs.
Fetch the relevant page. Tell me where this repository's SKILL.md should live,
then compare that rule with .agents/skills/check-reading-time/SKILL.md if it exists.
Include the official source link. Do not change files.
```

You should see a search tool call followed by a page fetch, with an answer pointing to the official skill documentation. Compare the answer against the returned page. If the skill file doesn't exist yet, Codex should tell you that instead of claiming it inspected it.

If it answers from memory without calling a tool, repeat the prompt and mention the server name. If tools are missing, run `/mcp verbose` to inspect the connection and check the URL, config scope, and network policy before relying on its output.

This server never asks you to sign in, so if you see an auth prompt, double-check the URL instead of pasting credentials. Other MCP servers often require OAuth, so follow their setup instructions.

To remove the user-level setup later:

```
codex mcp remove openaiDeveloperDocs
codex mcp list
```

If you used the project config, delete only the `[mcp_servers.openaiDeveloperDocs]` table and restart the connection. Run `/mcp` again to verify which tools are left, since another scope can still provide a server with the same name.

The MCP configuration guide covers server scopes, diagnostics, and authentication.

Reuse workflows with skills

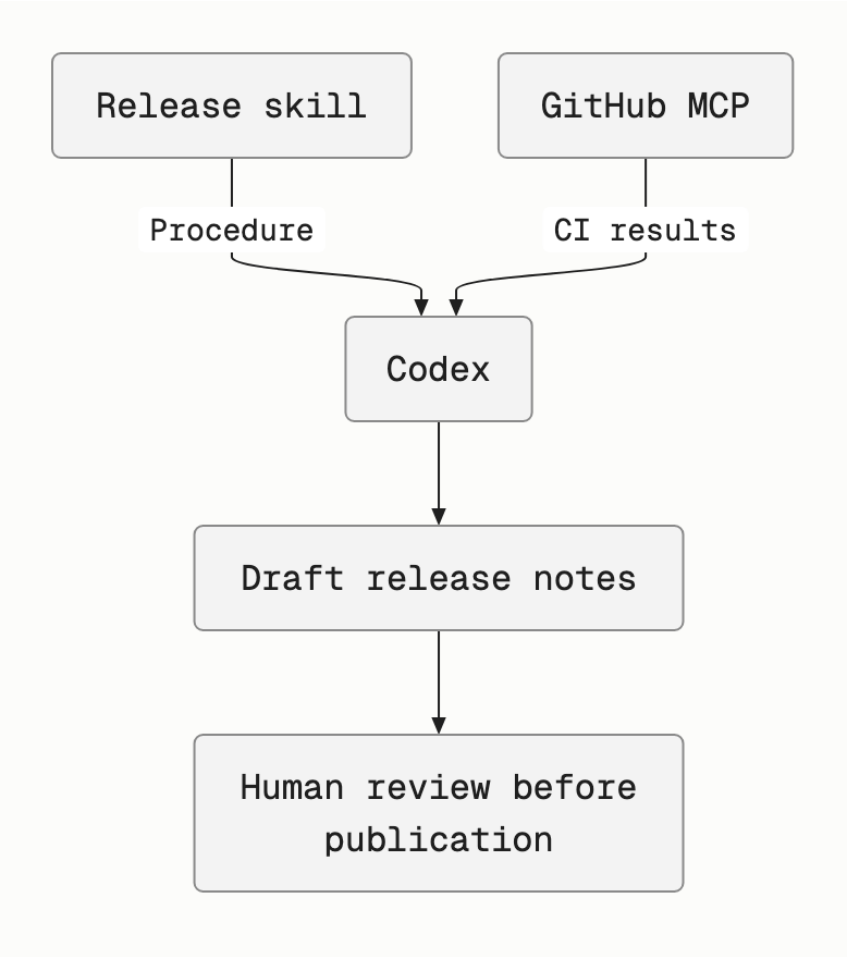
A skill teaches Codex how to perform one repeatable job. It's a folder with a `SKILL.md` file and, when needed, scripts, references, templates, or assets.

For example, a skill could explain how to:

- check a deployment before a release
- review a database migration

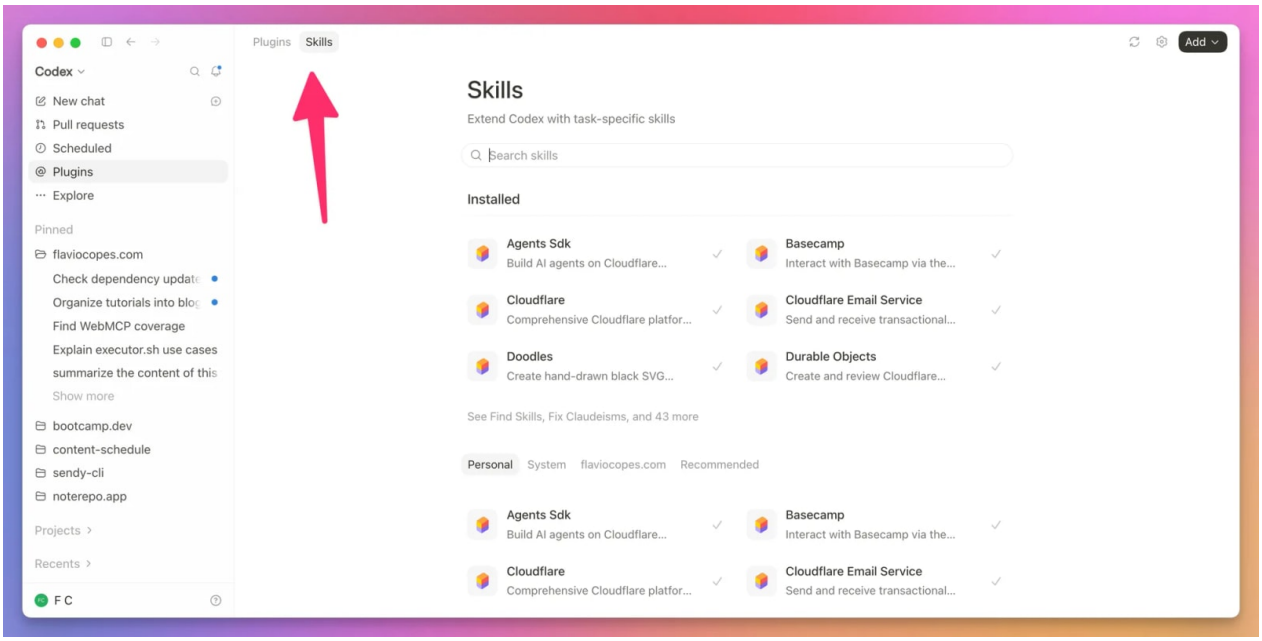
- prepare release notes
- diagnose a failed deployment

A skill tells Codex how to do the work, while an MCP server gives it live data and actions:



For example, a release skill can require passing CI and a reviewed changelog, while a GitHub connection lets Codex fetch the real CI status. The skill defines your release procedure, and the GitHub connection gives Codex access to the repo.

You can browse available skills right in the desktop app. A good skill has a narrow job, clear trigger phrases, and an output you can verify.



If you keep pasting the same instructions into prompts, turn them into a skill.

Skills add up quickly, and several end up doing nearly the same job. I built Skill Cabinet to see every skill on my Mac, which agents load it, and to merge the ones that read alike.

How to write an Agent Skill walks through building, testing, and packaging one.

Build a skill for the reading-time project

We keep checking the same things after each change: the one-minute minimum, fenced blocks, and whether the tests catch the missing behavior. Put that review procedure in a project skill.

Create its folder from the project root:

```
mkdir -p .agents/skills/check-reading-time
```

Save this complete file as `.agents/skills/check-reading-time/SKILL.md`:

```
---
name: check-reading-time
description: Review changes to this project's reading-time calculator and its tests. Use after changes
    ↪ to readingTime or when asked to check its behavior. Do not use for deployment or unrelated
    ↪ project reviews.
---

# Check the reading-time calculator

Read src/reading-time.js and test/reading-time.test.js.
Inspect git diff and git diff --cached for those files.
Run npm test and report the actual result.

Check that empty text still returns at least one minute.
Check backtick and tilde fences, including an unclosed fence.
Check that inline code still counts and wordsPerMinute still works.
For each behavior, point to a test or report the missing coverage.
Check whether the fenced-code regression fails against the original implementation.
Do not change working files to test a historical version. If that comparison
```

```
cannot be checked safely, report it as unverified.

Do not edit, stage, commit, or push files.
Return findings with file references, checks performed, and anything unverified.
```

Codex uses the `name` and `description` frontmatter to find the skill, and the Markdown body lays out the procedure. You don't need any scripts for this one.

In Codex, type `$` and select `check-reading-time`, or send:

```
$check-reading-time
Review the current calculator and tests. Report findings without changing files.
```

Watch the activity in the conversation: Codex should read the source and tests, inspect the diff, and run `npm test`. If it replies "looks good" without running the tests, it skipped the procedure. It should also tell you if any check couldn't run.

To test the skill against a known failure, reintroduce the unwanted minimum change from our recovery exercise and run it again. It should catch the failing test, and you can undo the change afterward using the reviewed recovery steps.

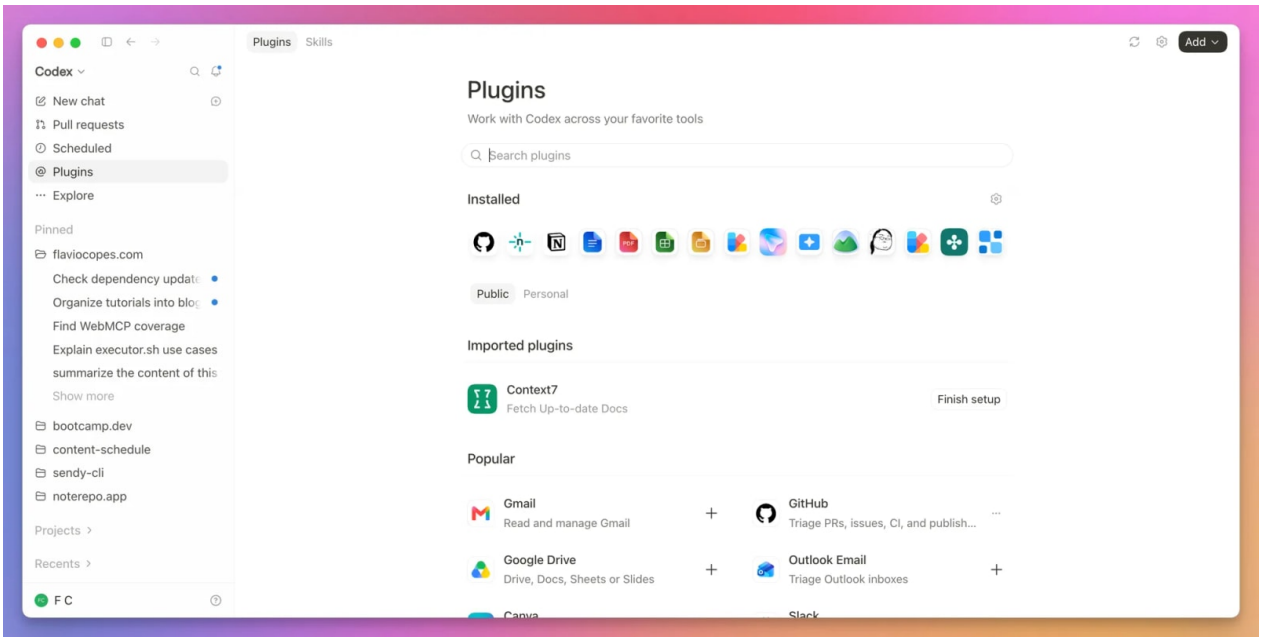
Codex picks up skill changes automatically. If the skill doesn't appear in `/skills`, check its filename and frontmatter, then restart the client. Project skills belong under `.agents/skills/` and personal skills belong under `~/.agents/skills/`. Avoid duplicate same-name skills, since Codex won't merge them. The skill authoring guide covers discovery and invocation.

Keep in mind that a skill can't enforce permissions on its own, so run it in a read-only session if you want to restrict local command writes alongside the review-only instructions.

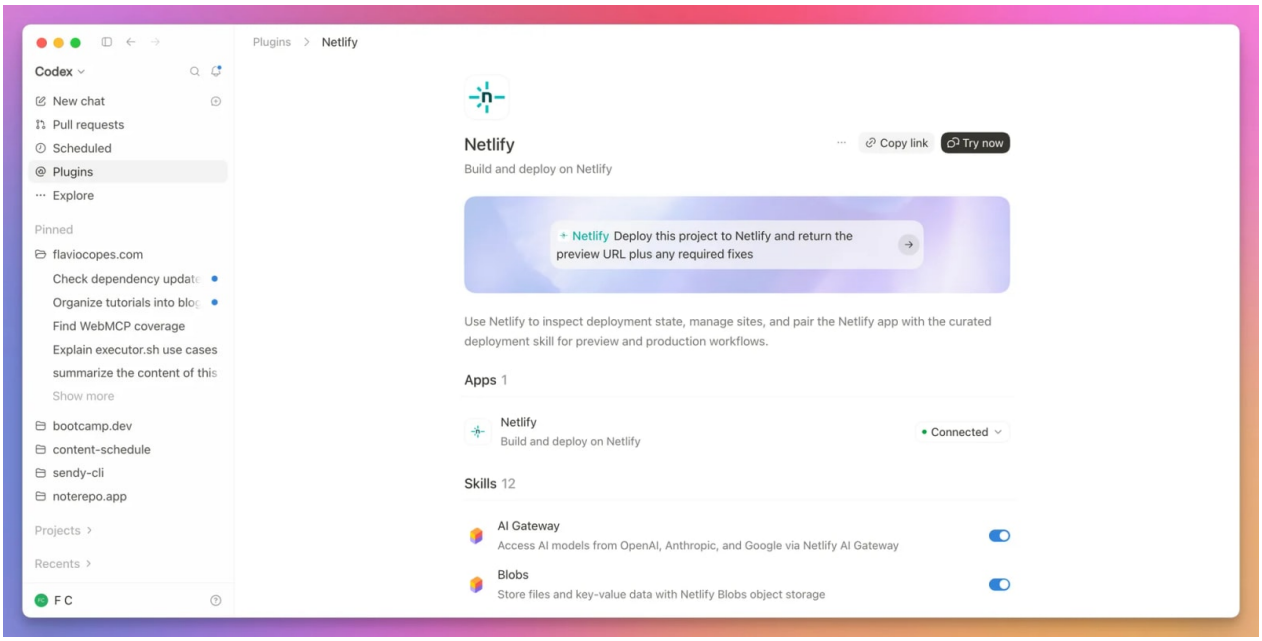
Install plugins

Plugins bundle Codex capabilities together: skills, MCP servers, hooks, browser features, and other resources.

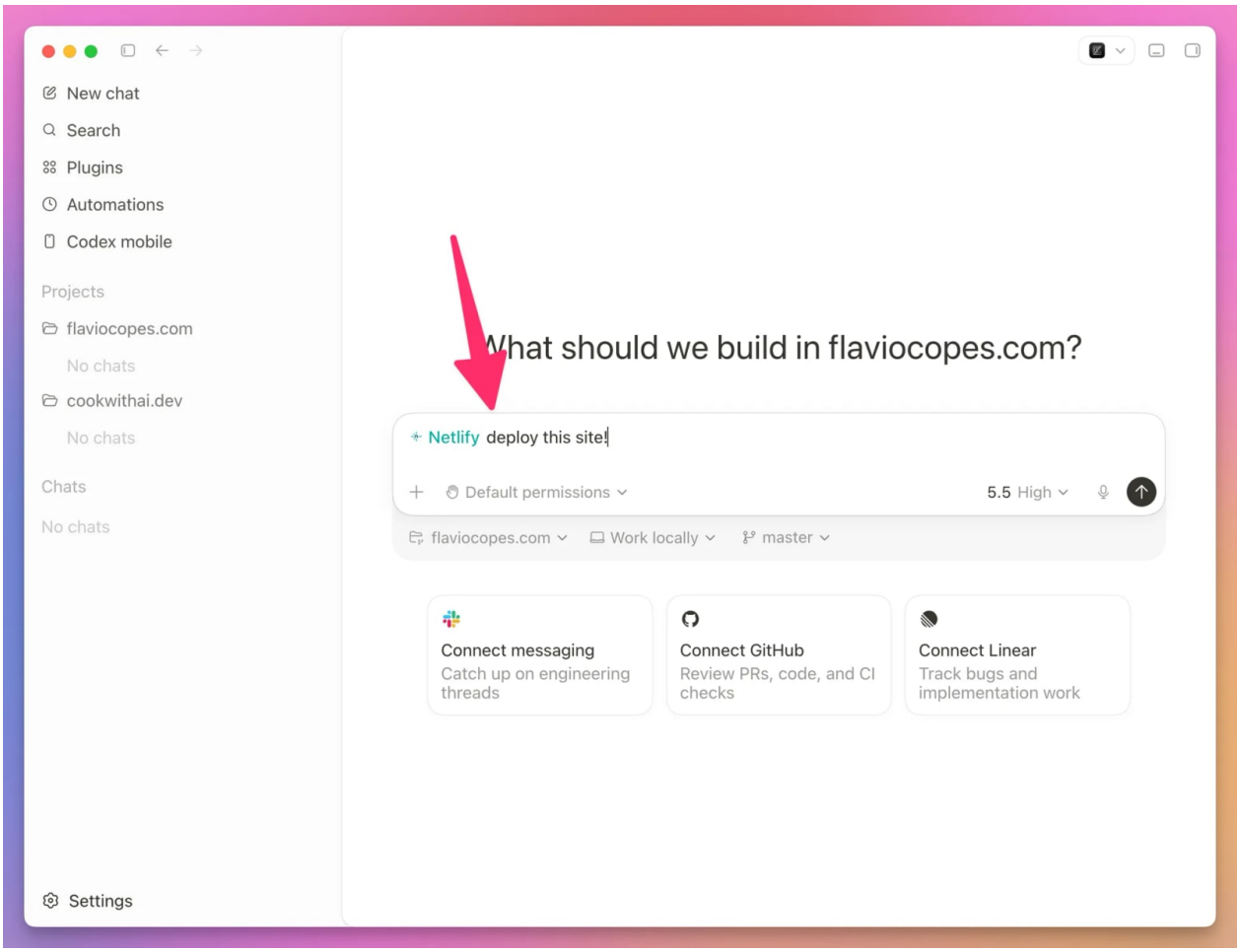
Browse them from the Plugins section:



The directory shows installed plugins and services you can connect. Open one to see the apps and skills it provides. ChatGPT and Codex share the same plugin directory, and I explained how to build and publish one (a `plugin.json` plus MCP servers and skills) in how to create and publish a ChatGPT plugin.

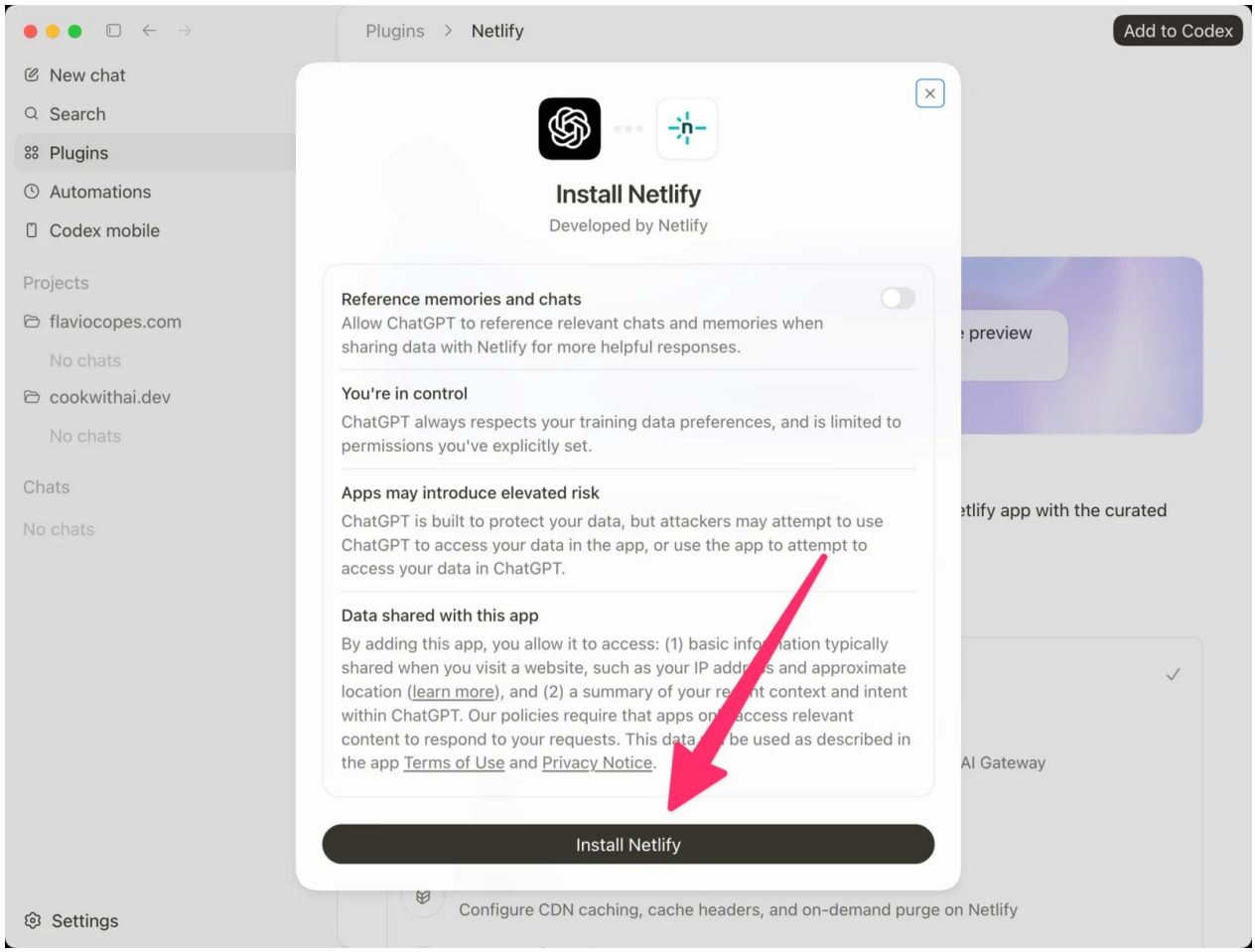


The Netlify plugin, for example, includes an app connection and a set of deployment skills. Once it's connected, you can click **Try now** or mention Netlify in a prompt.



Cloudflare, Vercel, Render, and Railway offer something similar, as a plugin or an MCP server. I compared them in Cloud platform plugins and skills for AI coding agents.

Some plugins stay completely local, while others connect to an external service and ask you to authorize an account. Check the access and data-sharing details in the install dialog before you connect one.



One plugin worth knowing about is Codex Security. It scans a repository for vulnerabilities, validates the plausible findings, and gives you evidence and remediation guidance for each one. Install it from Plugins and a **Security** item appears in the sidebar, where you set up scans and review the findings. Only scan code you own or have permission to check.

The same scanner is also available as a standalone CLI, which I covered in How to scan your code with the Codex Security CLI.

Use hooks for mechanical rules

Hooks run commands at specific points while Codex works. I use them for mechanical jobs like logging completed work, checking a command before it runs, or validating a file after an edit.

You define them in `~/.codex/hooks.json`, or in a `.codex/hooks.json` inside the project.

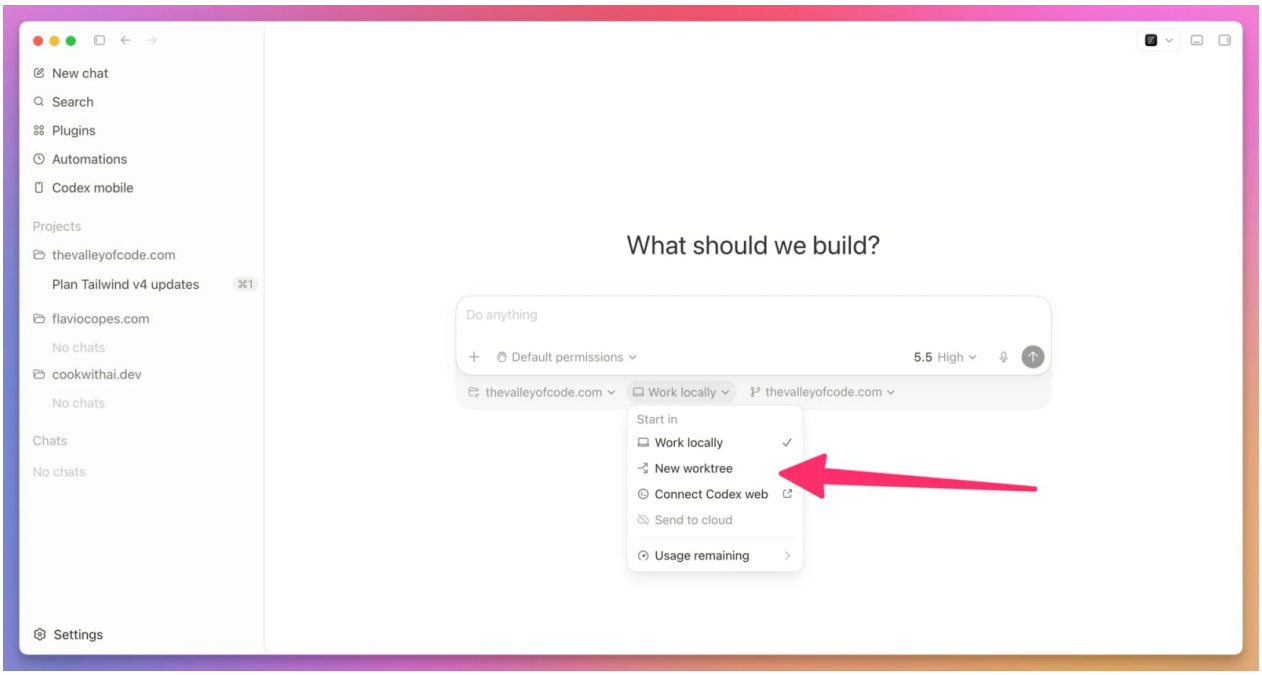
The one I use every day is a `PreToolUse` hook. Before Codex runs a `git commit`, it installs a Git `post-commit` logger in the repository, so every commit lands in the same plain text work log I keep with Cursor. I described the setup in How I use the same Git work log with Codex and ChatGPT.

Advice Codex needs to remember belongs in `AGENTS.md` or a skill instead.

Since hooks execute code on your machine, review them before turning them on, especially if they come from a plugin.

Local work, worktrees, and handoff

When you start a desktop chat for a Git repository, you can work in your normal local checkout or an isolated Git worktree.



A worktree gives the chat a separate copy of the repository, so several agents can work on the same repo without editing the same checkout.

Codex-managed worktrees live under `$CODEX_HOME/worktrees` by default. Tracked files are present, while ignored files like `.env` aren't copied unless you list them in `.worktreeinclude`:

```
# .worktreeinclude
.env
.env.local
```

You can hand a chat from a worktree to your local checkout, or move local work into a worktree. Codex handles the Git operations behind the move.

I normally work locally, one task at a time, because I want direct control over my current checkout. I use worktrees when tasks are genuinely independent, though I still avoid sending two agents into the same files since I'd have to reconcile both sets of changes afterward.

I explain the underlying Git feature in my Git Worktrees guide.

Split work across subagents

Codex can spawn subagents. Each one is a separate agent that takes one part of the job, runs in parallel with the others, and reports back to the main chat. Ask explicitly when you want Codex to delegate work, and check what your current client supports.

Subagents keep investigation output in their own threads and return summaries. The main chat gets the findings instead of every test log and stack trace, and independent work finishes faster when it runs in parallel.

Be specific about how to split the work and what each agent should return:

```
Review this branch against main with three subagents: one for security risks,
one for missing tests, one for maintainability. Wait for all three, then
summarize the findings by category with file references.
```

Use them for read-heavy work: exploration, tests, triage, review. Be careful about having several agents edit code at the same time, since they'll step on each other's changes. Each subagent does its own model and tool work, so the run costs more tokens than a single agent.

Subagents inherit the permission mode of the chat that spawned them, so set it before you delegate. In the CLI, `/subagents` switches between running subagent threads.

You can also define your own agents. Put one TOML file per agent in `~/.codex/agents/`, or in the project's `.codex/agents/`. Each file has a name, a description, and instructions, and can set its own model and reasoning effort:

```
# .codex/agents/reviewer.toml
name = "reviewer"
description = "Reviews a diff for correctness, security, and missing tests."
model_reasoning_effort = "high"
developer_instructions = """
Review without editing files. Lead with concrete findings and skip
style-only comments unless they hide a real bug.
"""
```

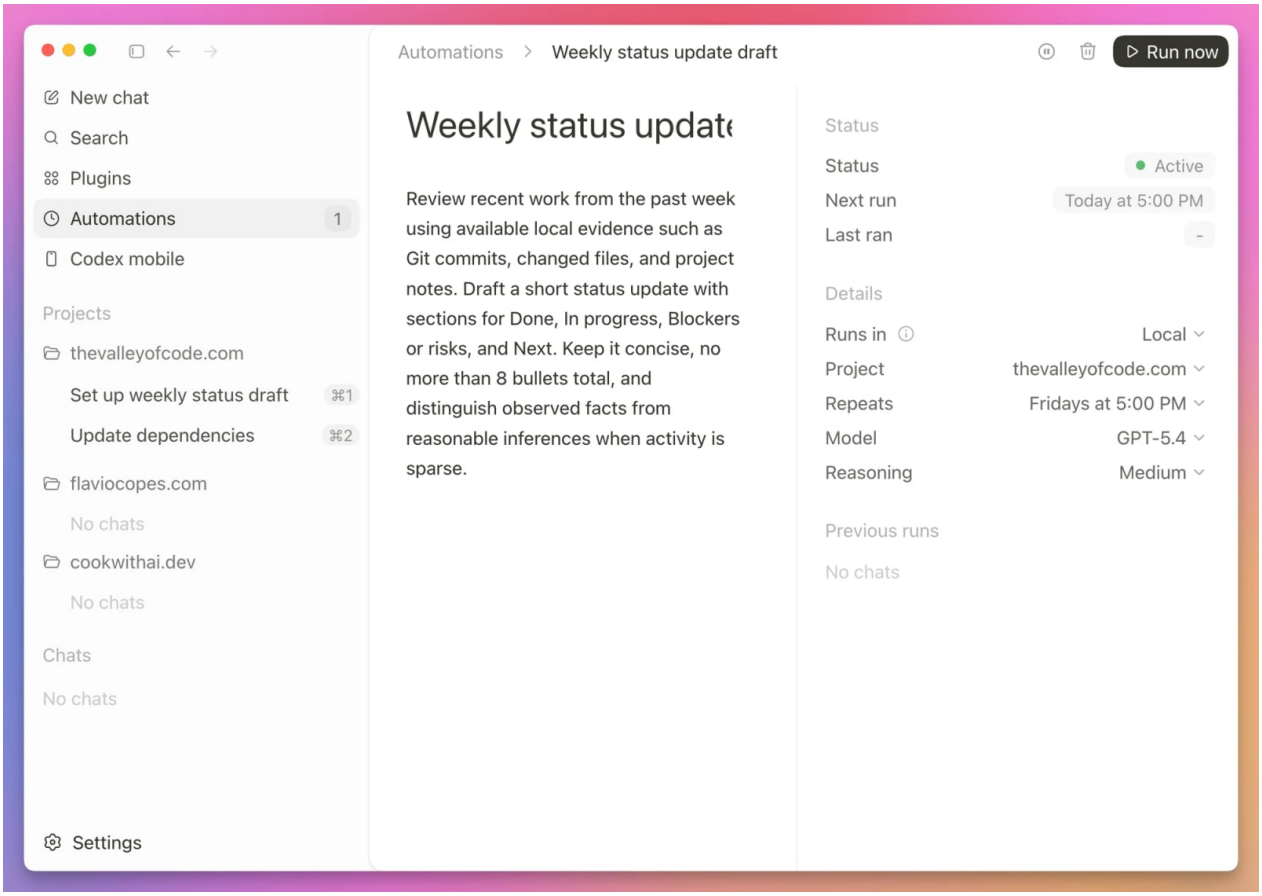
Telling an agent to review without editing guides its behavior, though it doesn't create a separate sandbox. You can also tune each agent individually, giving an explorer a cheaper model and a reviewer higher reasoning.

Create scheduled tasks

Scheduled tasks run a prompt later, or repeat it on a schedule. You can ask Codex to create one or set it up from the Scheduled section in the app.

For a standalone project task, choose the project, prompt, cadence, model, and execution environment. You can also schedule follow-ups inside an existing chat,

which keeps that chat's context. Use a standalone task when each run should be independent.



Useful scheduled tasks include:

- scan recent commits for likely bugs
- summarize yesterday's Git activity
- check outdated dependencies and propose safe upgrades
- draft a weekly changelog
- check CI failures

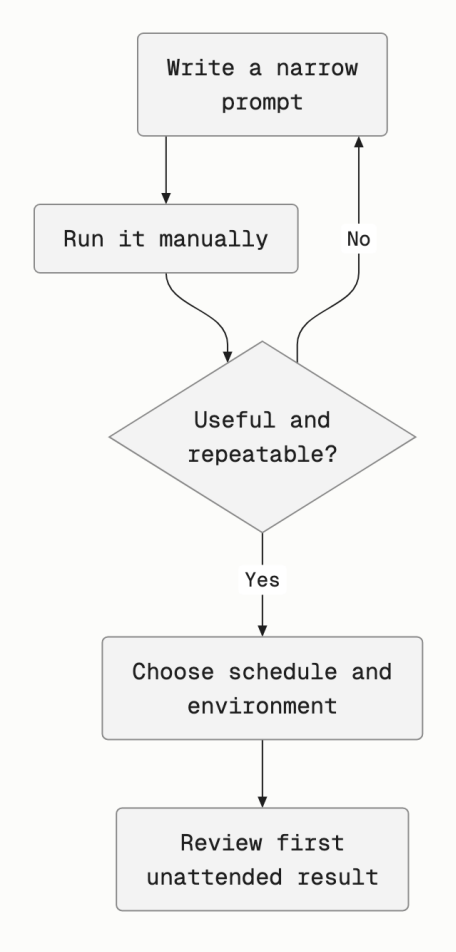
Try the prompt in a normal chat first and refine it until the result is useful without repeated corrections. I also wrote about what would make an overnight agent useful if you're thinking about leaving one running.

For local projects, your computer needs to be on and the desktop app has to be running. Scheduled tasks use your sandbox settings, and since unattended runs can't pause to negotiate every action, give them the narrowest access they need to succeed.

In a Git repository, a scheduled task can run in your local checkout or on a dedicated background worktree. Pick the worktree unless the task needs to touch the files you're editing right now, so unattended changes stay away from your unfinished work.

You can also name the exact skill a task should use by writing `$skill-name` in its prompt, so Codex doesn't have to pick the right skill on its own. The skill defines

the steps and the schedule decides when to run them, like this:



For a dependency check, start with a report. Let the task list outdated packages and describe the risks, while you keep upgrades and deployments as separate decisions. After several useful runs, you can decide whether to automate more of the workflow.

Use Codex from the terminal

The CLI runs Codex inside your terminal. Once you know the desktop workflow, it feels familiar. I wrote a separate guide on how to use the Codex CLI if you want the full tour.

I use it when I'm already working in a shell, or when I connect to a remote server over SSH. I go back to the app when I want its browser, diff, file panels, and multiple chats.

On macOS and Linux, install it with:

```
curl -fsSL https://chatgpt.com/codex/install.sh | sh
```

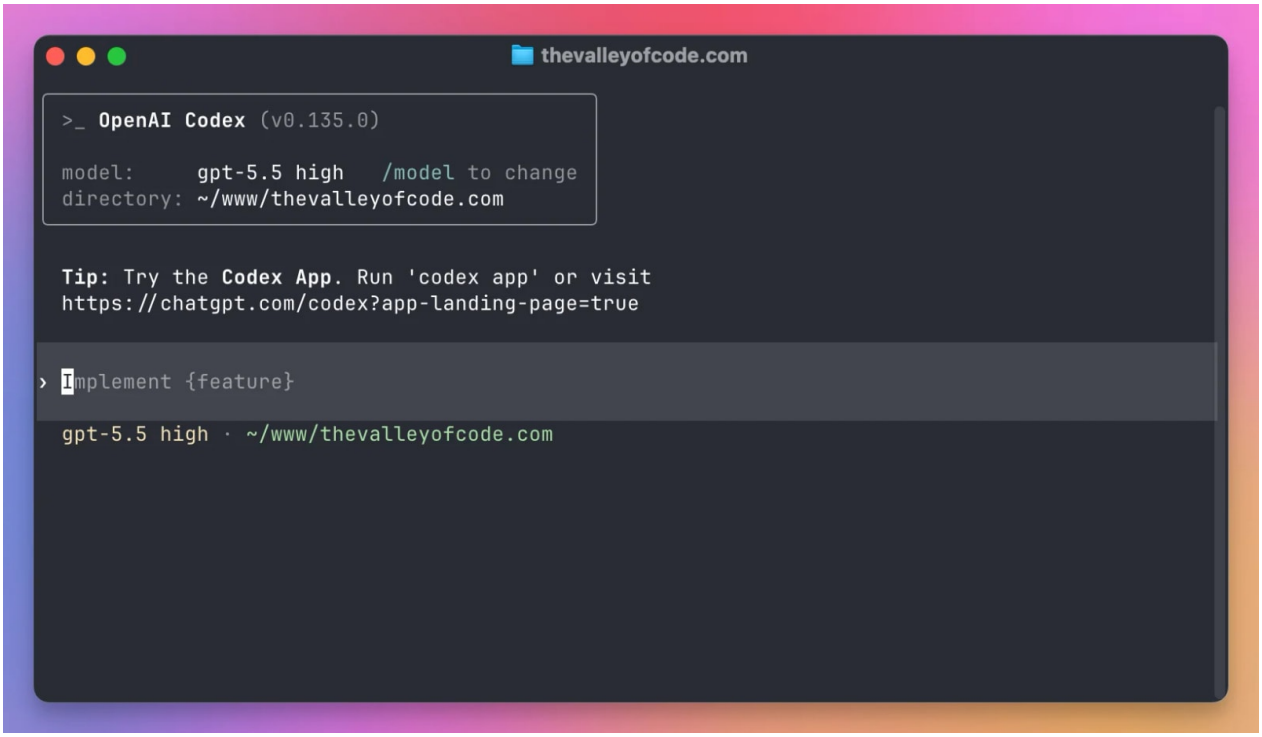
See the official CLI setup guide for Windows and the other installation options.

Move into a project and run:

```
cd ~/Sites/my-app
```

```
codex
```

The first time, Codex asks you to sign in with ChatGPT or use an OpenAI API key. ChatGPT sign-in uses the Codex allowance included with your plan. An API key bills usage through the OpenAI API.



You can open the interactive interface with a prompt already filled in:

```
codex "Explain how authentication works in this project. Don't change anything."
```

For a non-interactive task, use `codex exec`. From the reading-time project, this runs the same test suite and prints a final report:

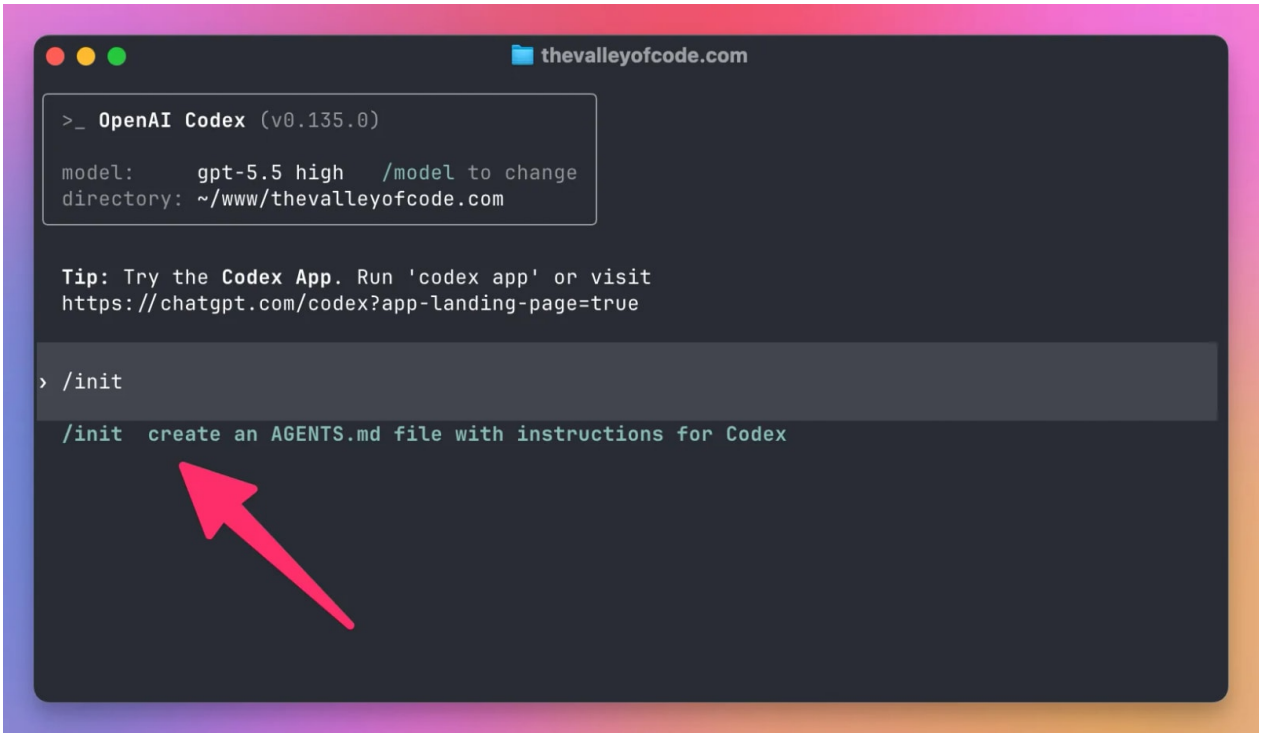
```
codex exec --sandbox read-only "Run npm test and summarize any failures. Do not edit files."
```

This exercise needs no generated files, so read-only access is enough. A different project's test runner might write a cache or build output; use `--sandbox workspace-write` when that's part of the job.

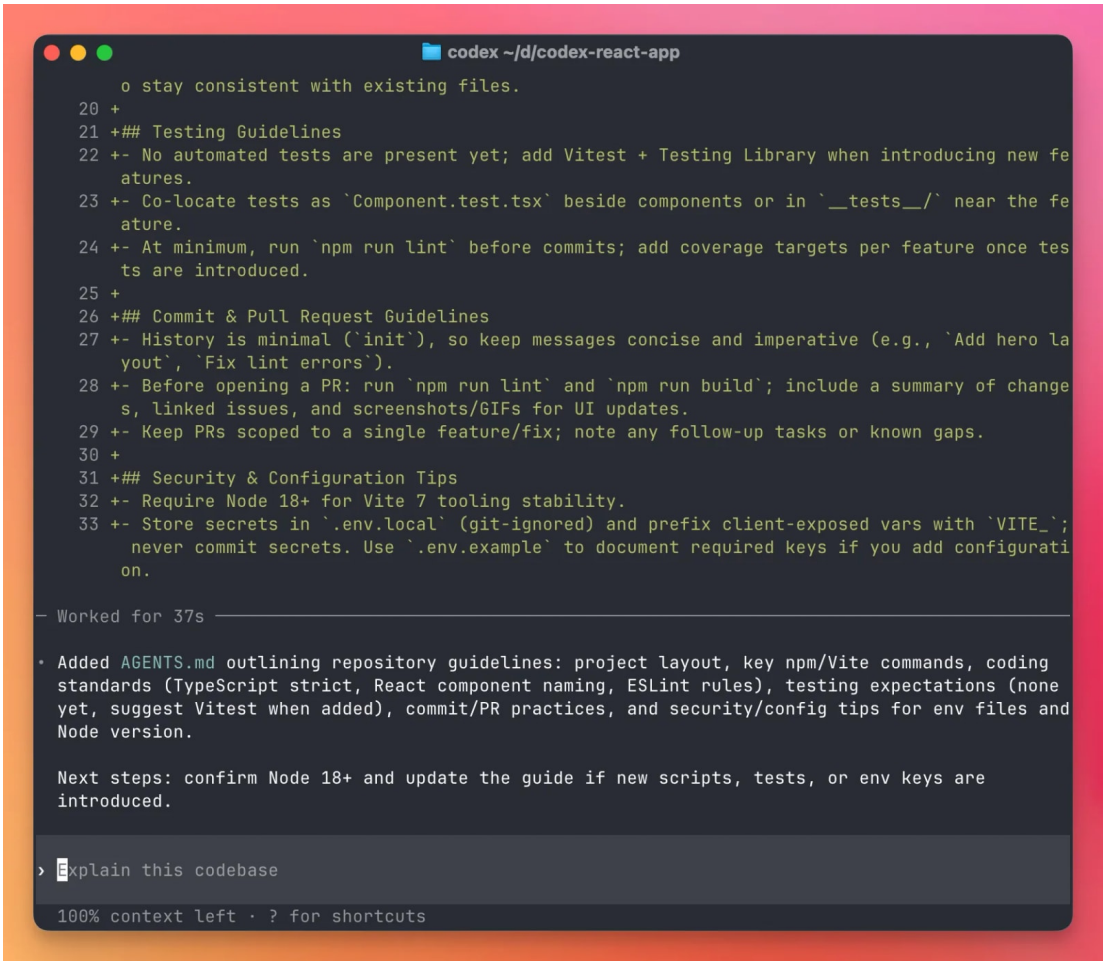
Progress goes to standard error and the final answer goes to standard output. Add `-o test-report.md` to save the answer. Your shell or the CLI writes that file, while the read-only sandbox limits the agent's commands. Read the reported result rather than trusting a clean exit code on its own.

The non-interactive mode guide covers structured output and automation. If you run into older examples that use `--full-auto`, drop that flag because it's no longer supported. In untrusted directories, `codex exec` now keeps a repository's `.codex/config.toml` and project MCP servers disabled by default even with `--sandbox workspace-write`; pass `--trust` when you want a script to load them. You can also run `codex exec --remote <ADDR>` against an existing Codex app server.

The current directory becomes the CLI workspace. Run `/init` if you want Codex to generate a starting `AGENTS.md`, then edit the file for your project.



Codex writes the generated project instructions to `AGENTS.md`.



Use `/mention` to add a file or folder to the prompt:

```
/mention src/lib/auth.js
```

You can pass one or more images with `-i` or `--image`:

```
codex -i screenshot.png "Explain this error and suggest the smallest fix"
codex --image before.png,after.png "List the visual regressions"
```

Web search normally uses an OpenAI-maintained cache. In full-access configurations, it defaults to live results. Pass `--search` when you need live results for one run:

```
codex --search "Summarize the latest release notes for this dependency"
```

Set `web_search = "live"` in `config.toml` to make that the default, or `web_search = "disabled"` to turn the tool off.

Run `/mcp` to check active MCP connections, or add a server with `codex mcp add`. Use `/skills` to browse skills and `/plugins` to browse installed and available plugins. After you install a plugin, start a new CLI session so its tools and skills load.

Fix the sign-in problem before changing code

If Codex won't start or behaves strangely, check your account and environment first:

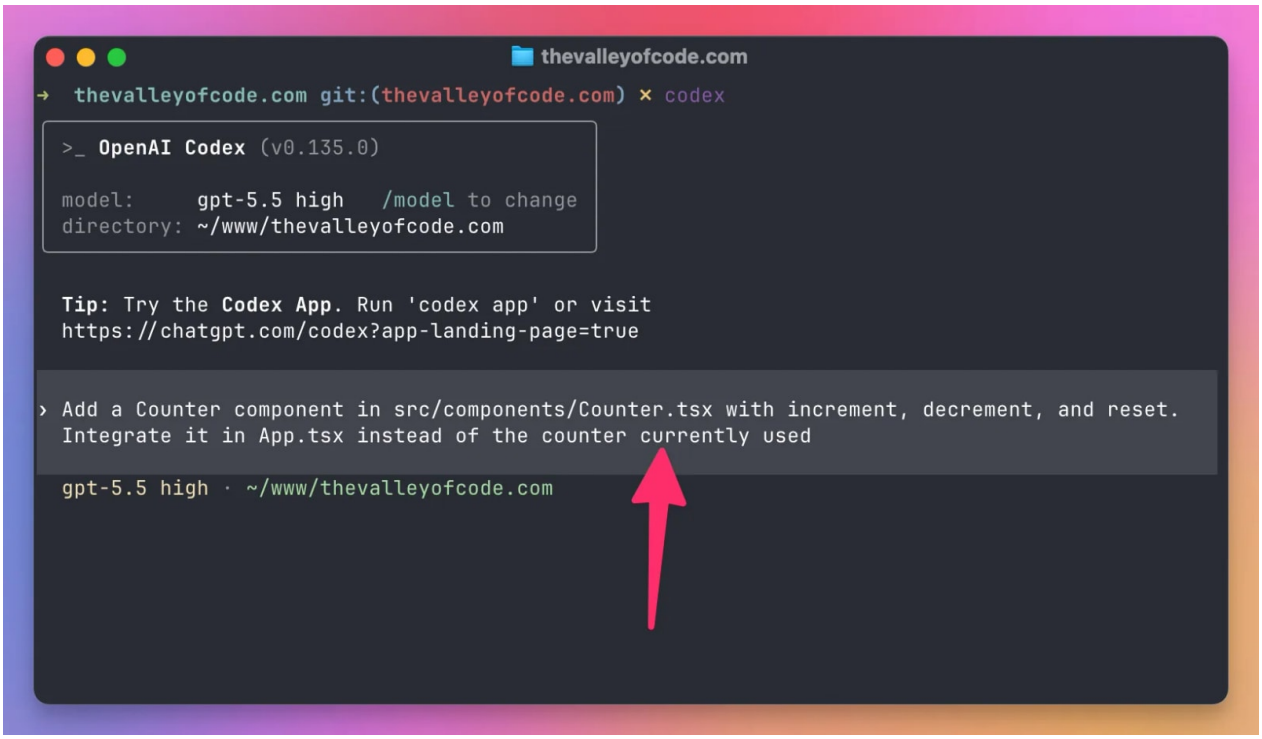
Problem	First check
`codex` isn't found	Open a new terminal and check the install directory is on `PATH`
The app and CLI show different access	Confirm the same account, workspace, and sign-in method
API-key usage appears on a bill instead of the plan	Check which sign-in method is active; API billing is separate
Cloud work is unavailable with an API key	Use an eligible ChatGPT sign-in for cloud features
A managed account blocks a feature	Check the workspace policy with its administrator
A local tool works on your laptop but fails in cloud	Check the cloud setup and its available credentials and services

Run `codex doctor` to check your local installation, configuration, and login state. Read the report before sharing it, and keep credentials out of prompts or screenshots. The authentication guide covers the supported login methods and environments.

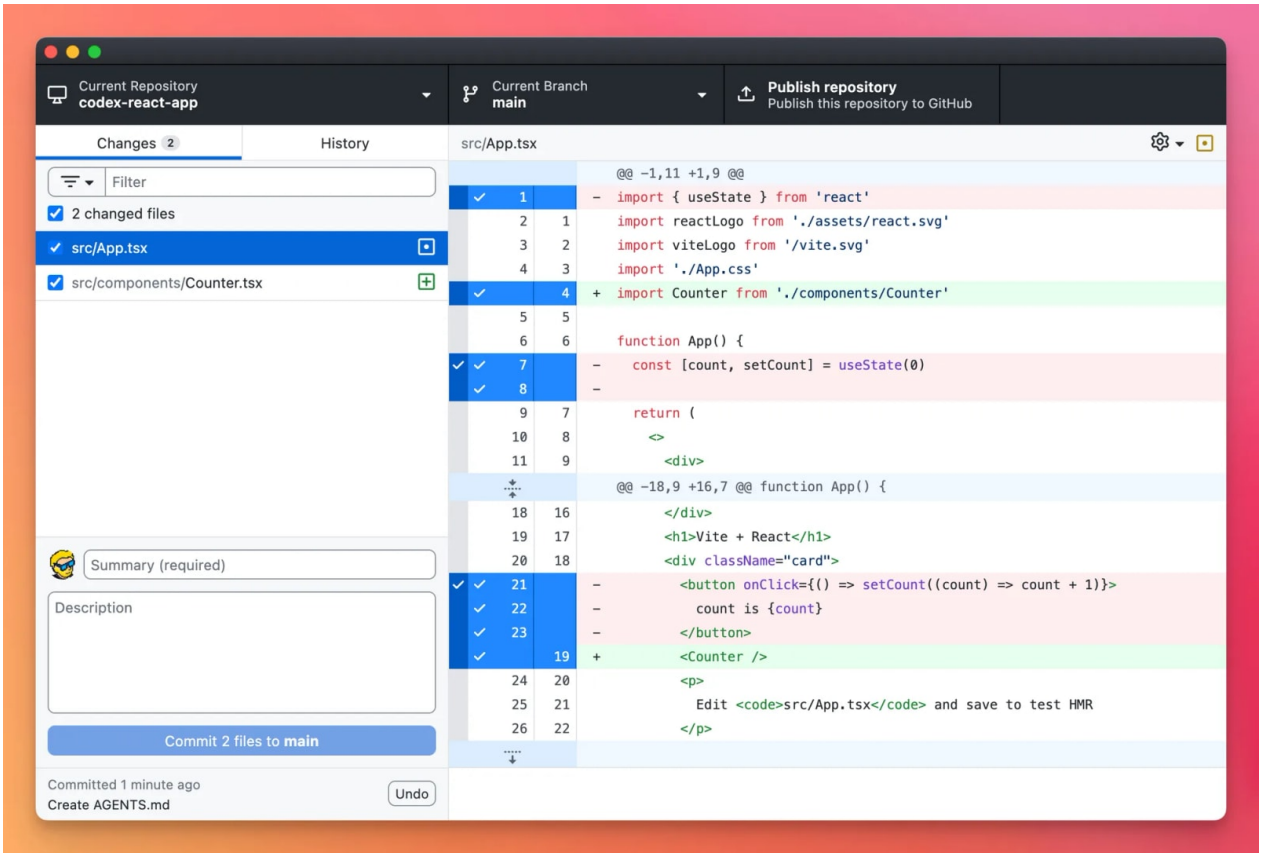
Make a change from the CLI

In the terminal, give Codex the same requirements and checks you'd use in the app.

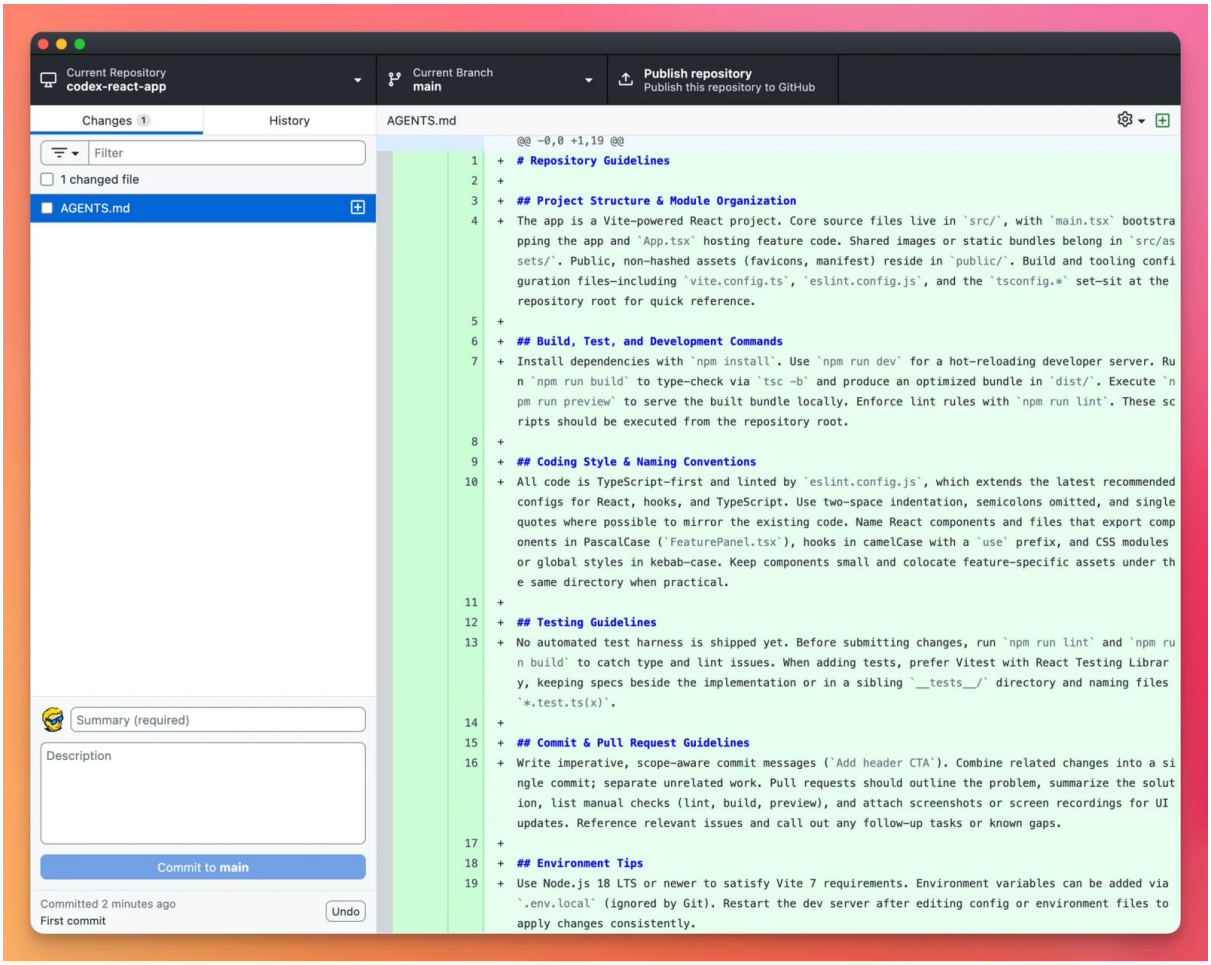
In this example I asked it to add a `Counter` component with increment, decrement, and reset controls, then use it from `App.tsx`.



Codex inspected the project, changed the files, and reported the result in the same terminal session.



You can review those files in your usual Git client too. Here they are in GitHub Desktop.



The CLI commands I use most

Type `/` to open the command list. It changes as Codex adds features, but these are the commands I use most:

- `/init` creates a starting `AGENTS.md`
- `/plan` enters Plan mode
- `/goal` starts or manages a persistent goal
- `/model` changes the model and reasoning level
- `/permissions` changes what Codex can do without asking
- `/mention` adds a file or folder to the prompt
- `/diff` shows the Git diff
- `/review` starts a separate reviewer on a diff: against a base branch, the uncommitted changes, one commit, or with your own instructions
- `/subagents` switches between running subagent threads
- `/compact` summarizes a long chat to recover context space
- `/status` shows the current model, permissions, workspace, and context usage
- `/usage` shows account usage
- `/resume` continues a saved chat
- `/fork` branches the current chat without losing the original
- `/mcp` inspects active MCP connections; `/skills` and `/plugins` browse their respective catalogs

- `/rc` (or `/remote`) manages local remote control and shows device pairing codes
- `/quit` exits

Since Codex CLI 0.171, the TUI workspace is on by default, showing searchable document previews, image previews, and subagent transcripts beside the conversation. Since 0.172, pressing `Tab` completes arguments for supported slash commands, and thread links in search results and transcripts open saved conversations and nested agents directly.

The CLI also has normal subcommands:

```
codex resume
codex review --uncommitted
codex exec "Run the tests and return only the failures"
codex doctor
```

Run `codex --help` and `codex <command> --help` for the version installed on your computer. The official command reference documents every option.

Other ways to use Codex

The IDE extension

The Codex IDE extension puts the agent beside your editor. It can use open files, your current selection, and the repository as context.

It's a good fit when a task starts from code you already have open in your editor. I still prefer the desktop app when I want the diff, terminal, browser, and several chats visible together.

Codex cloud

Codex cloud runs work in an isolated environment. You can delegate a task without keeping your computer busy. I walk through the setup in How to use Codex Cloud.

Cloud requires a ChatGPT sign-in, while API key sign-in is for local work. The cloud environment also needs its own setup, because the dependencies, environment variables, and services on your laptop aren't automatically available there.

Continue a local task from your phone

Remote access connects your phone to a computer already running the desktop app. The code and local tools stay on that host. This is useful for answering a question or checking a test result while you're away from your desk.

You need a Mac or Windows computer that stays awake and online, the current desktop and mobile apps, and the same ChatGPT account and workspace signed in on both.

On the computer, open **Settings > Connections > Control this Mac or PC**, then choose **Set up** or **Add**. Complete the verification step, scan the QR code with

your phone, and finish the connection in ChatGPT.

On the phone, open **Codex** (on Android it's called **Remote**). Choose the paired host, open the reading-time project, and continue its existing chat. Ask for a status update before sending another coding task:

```
Show the current task status, the latest npm test result,
and any question waiting for my decision. Don't change files.
```

The host's permissions and credentials still apply. If it sleeps or goes offline, that local connection can't reach its files. Save real code reviews for a screen where you can read the diff comfortably.

When you're done testing it, go back to **Settings > Connections** on the host to review your paired devices and remove any connection you don't want to keep. Never share the pairing QR code or code with anyone else.

Most people pair from the desktop app, though Codex CLI 0.171 also added `/rc` (or `/remote`) so you can manage remote control and display pairing codes from the terminal. OpenAI's remote connections guide covers platform support, verification, and disconnecting devices.

A real debugging workflow

I once woke up to an angry email because the unsubscribe link for my newsletter didn't work. The entire server was down, and the disk was 99% full.

I could have given Codex SSH access, but this was a production server with almost no free space, so I wanted to see every command. I ran each one myself and pasted the output back.

The loop looked like this:

```
I pasted df -h
Codex suggested the next safe inspection
I ran it
I pasted the result
Codex narrowed the search
```

We found that the Sendy database used only 528 MB. MySQL binary logs used around 41 GB.

Codex told me not to delete files manually from `/var/lib/mysql`. It gave me the MySQL command to purge old binary logs safely, and the disk went from 99% used to 41% used. The important detail was using the database's own maintenance command, after checking which logs could be removed. I wrote the full session in How I fixed a full Ubuntu server disk with Codex.

I don't let the agent execute everything on a risky system. In this case Codex guided the investigation, while I ran one command at a time and made the final decisions.

What I learned building a full app with Codex

I also used Codex to build a local web app that generates multi-page comics from a story and reference photos.

I started with a detailed prompt about characters, story projects, storyboards, image generation, exports, and local storage. Later I changed the stack, added reusable comic styles, switched the image API, and fixed mistakes I only noticed when I used the app.

The first prompt produced a lot of software, but it wasn't the exact product I had in mind. Some product decisions in my detailed prompt were wrong, and I only discovered missing requirements by using the app.

The generated UI needed more work too. When image generation failed silently, I asked Codex to add progress and error messages before touching the API. With the failure visible in the app and the logs, we could see what needed fixing.

That experience is why I now give Codex one clear goal at a time. I make sure I can see the result, try it myself, review the changes, and move on from there.

What I don't like about Codex

The naming is confusing. The app is called ChatGPT, Codex is a mode inside it, and most documentation pages describe ChatGPT Work and Codex together, so it takes a while to work out which sentence applies to you.

Goal mode is impressive, but it isn't how I want to work. After the PocketBase experiment, I'd rather spend that budget on ten well-scoped tasks I can review one by one.

The built-in browser doesn't know my logins. It has its own profile, so a page behind Google sign-in means signing in again inside Codex. The browser extension can drive your real Chrome instead, at the price of an extension that can read and change data on every site you visit.

Computer Use only exists in some regions, and it's a plugin rather than a built-in, so the setup has more steps than the browser.

Scheduled tasks for local projects need the computer awake and the app running. A task set for 7am does nothing if the laptop is closed. For anything that must run no matter what, I still use a cron job on a server or a scheduled GitHub Action.

When I pick Codex over Cursor or Claude Code

I use all three. Cursor gets most of my days, because I'm in the editor anyway and I like a local agent that works on my checkout, one task at a time. Claude Code is my other terminal agent, and the one I ran on a disposable server to build and ship a small app from the same box.

Codex wins for me when:

- I'm working on something visual. Clicking an element in the built-in browser and describing the change beats writing a paragraph about where it is.
- I want a task to run on a schedule, or unattended in a worktree while I do something else.
- I want the whole loop in one window: plan, change, terminal, browser, diff, commit.
- My ChatGPT plan includes it. If yours includes Codex too, you can try it within that plan's allowance.

With all three I keep the same habit of reviewing one outcome at a time. I compared the apps in Codex vs Claude Code vs Cursor, and the models in Which AI model is best for coding?. For a comparison of the terminal tools on the same task, see OpenCode vs Codex CLI on one bug fix.

My Codex workflow

This is the loop I follow for most project work:

1. Open the correct project.
2. Ask Codex to inspect the relevant code without changing it.
3. Use Plan mode if the task crosses several files or has unclear decisions.
4. Review the plan.
5. Ask Codex to implement one coherent outcome.
6. Let it run the tests and build.
7. Verify the behavior in the browser or app.
8. Read the full diff.
9. Ask for one final review.
10. Commit and push only when I understand the result.

For a long job with a precise end state, I use Goal mode. If a workflow becomes stable and repetitive, I turn the instructions into a skill and test it before scheduling anything.

None of this is specific to Codex. I wrote down where the loop comes from, after six months of working with Cursor, Claude Code, and Codex, in What I learned in the past 6 months about using agentic AI.

How I would start on an existing project

Pick a bug you can reproduce, create a Git checkpoint, and ask Codex to find the code responsible before it touches anything. Then give it one change and a concrete check.

For a web form, that check might be submitting an empty value, fixing it, saving, and reloading the page. For a CLI, it might be running the command with a known input and checking both the exit status and the output.

When Codex finishes, test the feature yourself and read the diff. If you can't explain why a changed file belongs in the task, ask about it before committing.

Try this on a small change in a repository you already know. A reproducible bug gives you an easy way to judge the result.