Workflows

Codex works best when you treat it like a teammate with explicit context and a clear definition of "done." This page gives end-to-end workflow examples for the Codex IDE extension, the Codex CLI, and Codex cloud.

If you are new to Codex, read Prompting first, then come back here for concrete recipes.

How to read these examples

Each workflow includes:

- **When to use it** and which Codex surface fits best (IDE, CLI, or cloud).
- **Steps** with example user prompts.
- **Context notes**: what Codex automatically sees vs what you should attach.
- **Verification**: how to check the output.

> **Note:** The IDE extension automatically includes your open files as context. In the CLI, you usually need to mention paths explicitly (or attach files with `/mention` and `@` path autocomplete).

Explain a codebase

Use this when you are onboarding, inheriting a service, or trying to reason about a protocol, data model, or request flow.

IDE extension workflow (fastest for local exploration)

1. Open the most relevant files.
2. Select the code you care about (optional but recommended).
3. Prompt Codex:

```
Explain how the request flows through the selected code.
```

```
Include:
- a short summary of the responsibilities of each module involved
- what data is validated and where
- one or two "gotchas" to watch for when changing this
```

Verification:

- Ask for a diagram or checklist you can validate quickly:

```
Summarize the request flow as a numbered list of steps. Then list the files involved.
```

CLI workflow (good when you want a transcript + shell commands)

1. Start an interactive session:

   ```
   codex
   ```

2. Attach the files (optional) and prompt:

   ```
   I need to understand the protocol used by this service. Read @foo.ts @schema.ts and explain the schema and request/response flow. Focus on required vs optional fields and backward compatibility rules.
   ```

Context notes:

- You can use `@` in the composer to insert file paths from the workspace, or `/mention` to attach a specific file.

Fix a bug

Use this when you have a failing behavior you can reproduce locally.

CLI workflow (tight loop with reproduction and verification)

1. Start Codex at the repo root:

   ```
   codex
   ```

2. Give Codex a reproduction recipe, plus the file(s) you suspect:

```
Bug: Clicking "Save" on the settings screen sometimes shows
"Saved" but doesn't persist the change.

Repro:
1) Start the app: npm run dev
2) Go to /settings
3) Toggle "Enable alerts"
4) Click Save
5) Refresh the page: the toggle resets

Constraints:
- Do not change the API shape.
- Keep the fix minimal and add a regression test if feasible.

Start by reproducing the bug locally, then propose a patch and run
 checks.
```

Context notes:

- Supplied by you: the repro steps and constraints (these matter more than a high-level description).
- Supplied by Codex: command output, discovered call sites, and any stack traces it triggers.

Verification:

- Codex should re-run the repro steps after the fix.
- If you have a standard check pipeline, ask it to run it:

```
After the fix, run lint + the smallest relevant test suite. Report the
 commands and results.
```

IDE extension workflow

1. Open the file where you think the bug lives, plus its nearest caller.
2. Prompt Codex:

   ```
   Find the bug causing "Saved" to show without persisting changes.
   After proposing the fix, tell me how to verify it in the UI.
   ```

Write a test

Use this when you want to be very explicit about the scope you want tested.

IDE extension workflow (selection-based)

1. Open the file with the function.
2. Select the lines that define the function. Choose “Add to Codex Thread” from command palette to add these lines to the context.
3. Prompt Codex:

   ```
   Write a unit test for this function. Follow conventions used in other tests.
   ```

Context notes:

- Supplied by “Add to Codex Thread” command: the selected lines (this is the “line number” scope), plus open files.

CLI workflow (path + line range described in prompt)

1. Start Codex:

   ```
   codex
   ```

2. Prompt with a function name:

   ```
   Add a test for the invert_list function in @transform.ts. Cover the happy path plus edge cases.
   ```

Prototype from a screenshot

Use this when you have a design mock, screenshot, or UI reference and you want a working prototype quickly.

CLI workflow (image + prompt)

1. Save your screenshot locally (for example `./specs/ui.png`).
2. Run Codex:

```
codex
```

3. Drag the image file into the terminal to attach it to the prompt.

4. Follow up with constraints and structure:

```
Create a new dashboard based on this image.

Constraints:
- Use react, vite, and tailwind. Write the code in typescript.
- Match spacing, typography, and layout as closely as possible.

Deliverables:
- A new route/page that renders the UI
- Any small components needed
- README.md with instructions to run it locally
```

Context notes:

- The image provides visual requirements, but you still need to specify the implementation constraints (framework, routing, component style).
- For best results, include any non-obvious behavior in text (hover states, validation rules, keyboard interactions).

Verification:

- Ask Codex to run the dev server (if allowed) and tell you exactly where to look:

```
Start the dev server and tell me the local URL/route to view the
prototype.
```

IDE extension workflow (image + existing files)

1. Attach the image in the Codex chat (drag-and-drop or paste).

2. Prompt Codex:

```
Create a new settings page. Use the attached screenshot as the
target UI.
Follow design and visual patterns from other files in this project.
```

Iterate on UI with live updates

Use this when you want a tight "design → tweak → refresh → tweak" loop while Codex edits code.

CLI workflow (run Vite, then iterate with small prompts)

1. Start Codex:

   ```
   codex
   ```

2. Start the dev server in a separate terminal window:

   ```
   npm run dev
   ```

3. Prompt Codex to make changes:

   ```
   Propose 2–3 styling improvements for the landing page.
   ```

4. Pick a direction and iterate with small, specific prompts:

   ```
   Go with option 2.

   Change only the header:
   - make the typography more editorial
   - increase whitespace
   - ensure it still looks good on mobile
   ```

5. Repeat with focused requests:

   ```
   Next iteration: reduce visual noise.
   Keep the layout, but simplify colors and remove any redundant
   borders.
   ```

Verification:

- Review changes in the browser "live" as the code is updated.
- Commit changes that you like and revert those that you don't.
- If you revert or modify a change, tell Codex so it doesn't overwrite the change when it works on the next prompt.

Delegate refactor to the cloud

Use this when you want to design carefully (local context, quick inspection), then outsource the long implementation to a cloud task that can run in parallel.

Local planning (IDE)

1. Make sure your current work is committed or at least stashed so you can compare changes cleanly.

2. Ask Codex to produce a refactor plan. If you have the `$plan` skill available, invoke it explicitly:

   ```
   $plan

   We need to refactor the auth subsystem to:
   - split responsibilities (token parsing vs session loading vs
   permissions)
   - reduce circular imports
   - improve testability

   Constraints:
   - No user-visible behavior changes
   - Keep public APIs stable
   - Include a step-by-step migration plan
   ```

3. Review the plan and negotiate changes:

   ```
   Revise the plan to:
   - specify exactly which files move in each milestone
   - include a rollback strategy
   ```

Context notes:

- Planning works best when Codex can scan the current code locally (entrypoints, module boundaries, dependency graph hints).

Cloud delegation (IDE → Cloud)

1. If you haven't already done so, set up a Codex cloud environment.

2. Click on the cloud icon beneath the prompt composer and select your cloud environment.

3. When you enter the next prompt, Codex creates a new thread in the cloud that carries over the existing thread context (including the plan and any local source changes).

   ```
   Implement Milestone 1 from the plan.
   ```

4. Review the cloud diff, iterate if needed.
5. Create a PR directly from the cloud or pull changes locally to test and finish up.
6. Iterate on additional milestones of the plan.

Do a local code review

Use this when you want a second set of eyes before committing or creating a PR.

CLI workflow (review your working tree)

1. Start Codex:

   ```
   codex
   ```

2. Run the review command:

   ```
   /review
   ```

3. Optional: provide custom focus instructions:

   ```
   /review Focus on edge cases and security issues
   ```

Verification:

- Apply fixes based on review feedback, then rerun `/review` to confirm issues are resolved.

Review a GitHub pull request

Use this when you want review feedback without pulling the branch locally.

Before you can use this, enable Codex **Code review** on your repository. See Code review.

GitHub workflow (comment-driven)

1. Open the pull request on GitHub.
2. Leave a comment that tags Codex with explicit focus areas:

   ```
   @codex review
   ```

3. Optional: Provide more explicit instructions.

   ```
   @codex review for security vulnerabilities and security concerns
   ```

Update documentation

Use this when you need a doc change that is accurate and clear.

IDE or CLI workflow (local edits + local validation)

1. Identify the doc file(s) to change and open them (IDE) or `@` mention them (IDE or CLI).
2. Prompt Codex with scope and validation requirements:

   ```
   Update the "advanced features" documentation to provide
   authentication troubleshooting guidance. Verify that all links are
    valid.
   ```

3. After Codex drafts the changes, review the documentation and iterate as needed.

Verification:

- Read the rendered page.