Codex: A Safe Way to Work With an AI Coding Agent

Codex is an AI coding agent. It can read a project, propose changes, and run commands in it. This guide shows you how to use it safely from the first session to a regular routine. You work on a fake practice project. You do not need the class video.

> **Check the current Codex documentation from OpenAI** before you rely on any feature, plan or limit. This guide describes safe habits that stay the same even when features change.

Part 1. The words you need

Word	Meaning
Agent	A tool that works in steps: it reads files, plans, edits and runs commands
Prompt	The instruction you give the agent
Diff	The exact lines added or removed by a change. Always read the diff before accepting
Command	Something the agent runs in the project, such as a test or an install
Approval	Your yes or no before the agent runs a command or changes a file
Sandbox	A limited area where commands run with restricted access
Repository	A project folder with its history, usually tracked with Git
Branch	A separate line of work, so changes can be reviewed before they join the main code

Part 2. Set up a practice project

1. Create a small practice project with one page and one test. Use fake data only.
2. Put the project in its own folder, and start a Git repository in it so every change is recorded.
3. Create a branch called `practice` for the agent's work.
4. Write one sentence describing what the project should do. Keep it in a file called `README.md`.

You should see: a folder with one page, one test, a README, and a clean Git status.

Part 3. Your first session

1. **Ask for a plan first.** Example: "Explain what you would change to add a hello message to the page. Do not edit any files yet."
2. **Read the plan.** If anything is unclear, ask about that step only.
3. **Approve one step at a time.** Read each proposed diff line by line.
4. **Run the tests yourself** after each accepted change.
5. **Commit with a clear message** only when the tests pass.

Pass when: the tests pass, and every changed line is one you can explain in your own words.

Part 4. Good requests and weak requests

Good request	Weak request
"Add input validation to the form on this page, and add one test."	"Make the app better."
"Explain the error on line 12 before changing anything."	"Fix everything."
"List the files you would change, then wait for my approval."	"Rewrite the whole project."
"Show me the diff for this one file."	"Just apply it."

Part 5. Safety rules

Risk	Control
A secret in the code or prompt	Store keys in environment settings. Never paste a key into a prompt or a file
A change you do not understand	Ask for a plain-English explanation of that diff before you accept it
A command that deletes or publishes	Refuse it unless you have read the exact command
Work in your main project	Use a copy or a separate branch, and review before merging
Too many files changed at once	Ask the agent to stop and list the files, then narrow the task
Installing unknown packages	Ask why each package is needed and check it on its official page first

Part 6. Reviewing a change

1. Open the diff and read each file.

2. Check that the change does only what the request asked for.
3. Check that no secret, personal data or unrelated file has changed.
4. Run the tests and read the output, not just the result.
5. Ask one question for each line you do not understand.

Part 7. Common problems and fixes

Problem	Fix
A change breaks the tests	Ask for the failing test output, then fix one thing at a time
The agent edits too many files	Stop it, ask for the list of files, then narrow the request
You cannot tell what changed	Ask for a file-by-file summary of the diff
The same error keeps returning	Ask for the cause before any new change
A command needs more access	Refuse until you know exactly why it is needed

Part 8. A regular routine

1. Start each session with a clear goal written in one sentence.
2. Ask for a plan, then approve steps one by one.
3. Review every diff and run the tests.
4. Commit, then write one line in a log: what changed and why.
5. End the session by asking for a summary of what was done and what remains.

Part 9. Checklist

Item	Done
Practice project used, not the main project	[ ]
Plan requested before any edit	[ ]
Every diff read before it was accepted	[ ]
Tests run and read after each change	[ ]
No secrets in the project or prompts	[ ]
Current Codex documentation checked	[ ]

Part 10. Quiz (answers follow)

1. Why ask for a plan before any edit?
2. What is a diff?
3. Where should a key be stored?
4. Why use a branch?
5. What do you do when a change breaks the tests?

Answers: 1. So you can check the approach before any file changes. 2. The exact lines added or removed. 3. In environment settings, never in a prompt or file. 4. So changes can be reviewed before they join the main code. 5. Read the failing output and fix one thing at a time.

Part 11. Glossary

Term	Plain meaning
Agent	A tool that works through steps on a task.
Diff	The exact changes in a file.
Approval	Your yes before a command or change runs.
Branch	A separate line of work for review.

Feature names, plans and access rules change. Check OpenAI's current developer documentation on the day you use Codex.

Part 12. Six working patterns

Most coding tasks fit one of these patterns. For each, the prompt shape and the check are the same every time.

Pattern	Prompt shape	You check
Explain code	"Explain what this file does in five lines. Do not change anything."	The explanation matches what you read yourself
Fix a bug	"The test fails with this error: . Explain the cause first, then propose the smallest fix."	The fix touches the cause, not only the symptom
Add a feature	"Add . List the files you will change, then wait for my approval."	Only the listed files change; a test covers the new behaviour
Write tests	"Add tests for : a normal case, an empty input and a bad input."	Each test fails when the code is broken on purpose

Refactor	"Rename to everywhere. Do not change behaviour."	All tests pass before and after, and the diff is only the rename
Review a change	"Review this diff for bugs, unclear names and missing tests. List findings; do not edit."	Each finding points to a real line

Part 13. Prompt templates you can copy

```
Goal: <one sentence>
Context: <the file or function, and what it does today>
Constraint: change only <files>; do not touch <files>
Plan first: list the steps and the files; wait for my approval
Done when: <the test or behaviour that proves it works>
```

Use the same five lines every time. They keep the agent focused, and they give you a clear way to judge the result.

13.1 Examples of the template

Task	Filled-in prompt
Validation	Goal: reject an empty name on the form. Context: form handler in the practice app. Constraint: change only the handler and its test. Plan first. Done when: the new test passes and the old tests still pass.
Bug	Goal: fix the wrong total when the list is empty. Context: the total function. Constraint: change only that function. Plan first. Done when: a test for an empty list passes.

Part 14. A standing instruction file

Many coding agents read a short instruction file at the root of a project, so you do not retype the same rules each session. The file name depends on the tool; check the current Codex documentation for the name it reads. A good file is short and concrete.

```
# Project rules
- Run the tests before you say a change is done.
- Change only the files named in the request.
- Never edit files under /config or any file that holds secrets.
- Ask before adding a new package; give the reason.
- Keep functions short; explain any new function in one comment.
```

Keep this file under one page. A long file is hard to follow, and it costs tokens on every session (see the tokens guide).

Part 15. Working with Git, step by step

1. **Start clean:** check that there are no uncommitted changes before you start a session.
2. **Branch:** create a branch for the task, for example `practice-validation`.
3. **Small steps:** after each accepted change that passes the tests, make a commit with a clear message.
4. **Review the history:** read the list of commits before you merge. Each one should be a small, understandable change.
5. **Undo safely:** if a change is wrong, go back to the last good commit, then try a smaller request.

Habit	Why
One task per branch	A bad change is easy to drop
Commit after each passing step	You can return to the last good state
Never work on the main branch	Mistakes cannot reach shared code
Write messages that say why	You can understand the history later

Part 16. Reading a diff, line by line

What you see	What it means	What to ask
Lines starting with +	Added lines	"Why is this line needed?"
Lines starting with -	Removed lines	"Is anything else relying on this?"
A change in a file you did not name	The agent went beyond the request	"Why did you change this file? Undo it."
A new dependency	A new package was added	"Why this package? Is it needed?"
A change to a test	The agent edited the check, not the code	"Why did the test need to change?"

> **Watch for tests changed to pass.** If a failing test is edited so that it passes, the problem is hidden, not fixed. Ask why the test changed before you accept it.

Part 17. Security in more depth

Area	Risk	Control

Secrets	A key committed to the repository	Keep keys in environment settings; add secret files to the ignore list; scan before you commit
Dependencies	A new package that is unsafe or unneeded	Ask why; check its official page; prefer packages you already use
Commands	A command that deletes files or sends data out	Read the exact command; refuse if it is unclear
Personal data	Real data in test files	Use fake data in every test and example
Outside text	A file or web page that contains instructions aimed at the agent	Treat outside text as data; do not let it change your rules
Access	The agent can reach more than it needs	Work in a copy with the least access the task needs

Part 18. Six labs (fake practice project)

1. **Explain, then change.** Ask for an explanation of one function. Pass when you agree with it, and then a one-line change works.
2. **Fix a planted bug.** Break a function on purpose. Pass when the agent explains the cause before it fixes it, and the test passes.
3. **Add a validation rule.** Pass when a new test fails without the rule and passes with it.
4. **Write three tests.** Normal, empty and bad input. Pass when each test fails if you break the code on purpose.
5. **Rename safely.** Rename one function everywhere. Pass when the diff is only the rename and all tests pass.
6. **Review a diff.** Ask for a review of a small change. Pass when each finding points to a real line.

Part 19. End-to-end project: a to-do list app (fake data)

19.1 Plan on one page

Item	Answer
Goal	A small page where you can add, complete and delete tasks
Data	Fake task titles only
Must never	Store personal data, or call a service that needs a key
Done when	All tests pass and you can explain every function

19.2 Build in seven steps

1. Ask for a plan: files, functions and tests. Approve it.
2. Ask for the add-task function and a test. Review, run, commit.
3. Ask for the complete-task function and a test. Review, run, commit.
4. Ask for the delete-task function and a test. Review, run, commit.
5. Ask for validation: empty titles and very long titles are rejected, with tests.
6. Ask for a short README that explains how to run the tests.
7. Ask for a review of the whole diff, then fix the findings one at a time.

Pass when: every commit is small, every test passes, and you can describe each function in your own words.

Part 20. Troubleshooting, extended

Problem	Likely cause	What to do
The agent keeps repeating one fix	The cause is not understood	Stop; ask for the cause with evidence from the error
Tests pass but the app is wrong	Tests do not cover the behaviour	Add a test that fails on the wrong behaviour
A huge diff appears	The request was too broad	Reject it; ask for a smaller step
The agent invents a function that does not exist	It guessed at an API	Ask it to show where the function is defined, or to check the official documentation
It cannot run a command	Limited permissions	Run the command yourself, then paste the output
Results change each time	The request is vague	Add constraints and a clear "done when"

Part 21. Team practice

- **Review rule:** a person reads every change before it is merged, whoever wrote it.
- **Small changes:** keep changes small enough to review in ten minutes.
- **Share prompts:** keep a short file of prompts that worked, with what they were for.
- **Record decisions:** when the agent proposes a design, write down what was chosen and why.
- **Onboarding:** new teammates start in a practice project before they touch real code.

Part 22. Cheat sheet

Need	Do this
Understand code	Ask for an explanation; do not allow edits
Start a change	Plan first, then approve step by step
Check a change	Read the diff, run the tests, read the output
Stay safe	Practice copy, branch, no secrets, fake data
Save progress	Commit after each passing step
Recover	Return to the last good commit and ask for a smaller step

Part 23. Extended quiz (answers follow)

1. Why change only the files you named?
2. What does it mean if a failing test was edited to pass?
3. Why commit after each passing step?
4. Why keep the standing instruction file short?
5. What do you do when the agent mentions a function you do not recognise?

Answers: 1. It keeps changes small and reviewable. 2. The problem may be hidden, not fixed; ask why. 3. You can return to the last good state. 4. A long file is hard to follow and costs tokens every session. 5. Ask where it is defined, or check the official documentation.

Part 24. Extended glossary

Term	Plain meaning
Agent	A tool that works through steps on a task.
Diff	The exact lines added or removed.
Commit	A saved snapshot of changes with a message.
Branch	A separate line of work for review.
Test	A small check that proves the code does what you expect.
Refactor	Changing the code's shape without changing its behaviour.
Dependency	A package your project relies on.
Standing instruction file	A short file of project rules the agent reads each session.

Sandbox	A limited area where commands run with restricted access.

Feature names, plans and access rules change. Check OpenAI's current developer documentation on the day you use Codex.

Part 25. When not to use an agent

Situation	Better choice	Why
A one-line change you understand	Edit it yourself	Faster and safer than explaining it
A change to security, payments or access rules	A person who owns that area, with a review	A mistake is costly; the agent may miss hidden rules
Code you do not understand at all	Ask for an explanation first, then decide	Editing code you cannot read is risky
A task with no way to test it	Write the test first, or do not delegate	You need a way to prove the result
Exact calculations	A written formula and a test	Agents can make small arithmetic mistakes

Part 26. Working in an old or unfamiliar project

1. **Map first.** Ask for a list of the main folders and what each is for. Check it against what you see.
2. **Find the entry points.** Ask where the program starts and which files handle the main requests.
3. **Run the tests before you change anything.** Note which fail already, so you do not blame new changes.
4. **Change one small thing.** Prove you can build, run and test before a larger change.
5. **Add a test around any code you will change.** Then change it, and confirm the test still passes.

Pass when: you can explain the path of one request through the project in five sentences.

Part 27. Code review checklist

Question	Pass when

Does the change do what was asked, and only that?	Yes, with no unrelated edits
Is there a test for the new behaviour?	Yes, and it fails without the change
Are names clear?	A new reader can guess what each does
Are errors handled?	Empty, missing and wrong inputs give clear results
Any secret or personal data?	None
Any new dependency?	Only with a clear reason
Is anything copied that should be shared?	No repeated blocks that should be one function

Part 28. Writing documentation with the agent

1. Ask for a README draft that explains what the project is, how to run it and how to run the tests.
2. Run every command in the README yourself. A command that does not work is a bug in the README.
3. Ask for a short comment above each function that says what it does, not how.
4. Remove anything you cannot check.

> **Check every claim.** An agent can describe code in a confident way that does not match what the code does. Documentation you have not checked is a risk.

Part 29. Cost and time awareness

Habit	Effect
Name the files in the request	The agent reads less, so it costs less and stays focused
Ask for a plan first	You catch a wrong approach before many edits
Keep a short standing instruction file	Fewer tokens per session (see the tokens guide)
Start a new session for a new task	Less old context to carry
Cap the scope	Smaller diffs are quicker to review

Plans, usage limits and pricing change. Check OpenAI's current pages before you decide how much to rely on an agent.

Part 30. Second project: a unit converter (fake data)

30.1 Plan

Item	Answer
Goal	Convert length, weight and temperature between common units
Rules	Reject unknown units and non-numbers with a clear message
Tests	At least five per unit type, including zero and negative values
Done when	All tests pass and you can explain each formula

30.2 Build in six steps

1. Write the formulas in a plain list yourself first. Exact formulas belong to you, not to the agent.
2. Ask for one function and its tests for length. Review, run, commit.
3. Repeat for weight and for temperature.
4. Ask for input validation with tests for unknown units and non-numbers.
5. Ask for a short README with three examples.
6. Ask for a review of the whole diff; fix the findings one at a time.

Pass when: every test passes, the results match a hand check on five values, and each formula matches your written list.

Part 31. Frequently asked questions

Question	Answer
Is it safe to let an agent run commands?	Only in a practice copy, with approval for each command, and with no secrets in the project.
How do I know the code is correct?	You read the diff and run tests you trust. The agent saying it is correct is not proof.
What if the agent is wrong about a library?	Check the library's official documentation, and ask the agent to show where it found the function.
Can I use it for a real project?	Yes, with a branch, small steps, review, tests and no secrets in prompts. Start with low-risk tasks.
Which should I learn first: Codex or Claude Code?	The habits are the same. Learn the habits on a practice project, then use whichever tool your team uses.

Where do I find current features?	OpenAI's developer documentation for Codex, checked on the day you use it.

Part 32. Final checklist

Item	Done
Practice project and branch used	[ ]
Plan requested and approved before edits	[ ]
Standing instruction file written and kept short	[ ]
Every diff read; tests run and read	[ ]
Commit after each passing step	[ ]
No secrets, no personal data in files or prompts	[ ]
New dependencies justified	[ ]
Documentation commands run by me	[ ]
Current Codex documentation checked	[ ]

Part 33. Prompt library

Keep a short file of prompts that worked. These are starting points; change the names to match your project.

Use	Prompt
Orient	"List the main folders and say in one line what each is for. Do not edit anything."
Find	"Where is defined and which files call it? List only; do not edit."
Plan	"I want . List the files you would change and the tests you would add. Wait for my approval."
Small change	"Change only . Add . Add one test. Show me the diff."
Debug	"This test fails: . Explain the cause in three lines before proposing a fix."
Review	"Review this diff. List bugs, unclear names and missing tests. Do not edit."
Explain to me	"Explain this function as if I am new to programming. Use one example with fake data."

Wrap up	"Summarise what changed in this session, what remains, and one risk I should check."

Part 34. A four-week practice plan

Week	Focus	Pass when
1	Explain and orient on a small project; no edits	You can describe the project in your own words
2	Small changes with tests: validation, rename, one bug fix	Every change has a passing test and a clear commit
3	Review and refactor; read every diff aloud to yourself	You catch at least one issue in a diff before accepting it
4	A small project from plan to README	All tests pass and you can explain every function

Part 35. How to tell whether you are using it well

Measure	How	Good sign
Accepted without edits	Share of diffs you accepted as they were	Not near 100 percent; you are reading and asking
Tests passing after a change	Passing runs divided by changes	High, and rising
Size of changes	Lines per commit	Small and steady
Rework	Changes you had to undo	Falling over time
Understanding	Can you explain the change without the agent?	Yes, every time

> **The test that matters most:** if you cannot explain a change in your own words, you should not accept it yet.

Part 36. A short list of habits to keep

1. Plan first. Edit second.
2. Name the files. Name the finish line.
3. Read every diff. Run every test. Read the output.
4. Commit small. Branch always.

5. No secrets. No personal data. No unchecked commands.
6. Check the official documentation when a fact matters.