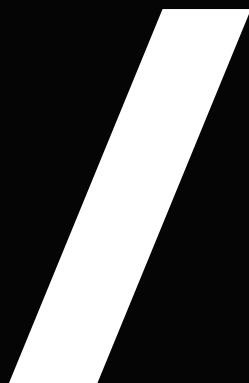


The Monolith Workspace

Turn what you know into a workspace AI can use.

A practical guide to setting up your files, giving clear instructions and building a repeatable way to work.



Build with AI. Keep The CTRL.

Give your next request a better starting point

A new client has sent you an intake form. You want to welcome them and explain what happens next. You ask AI to write a note. It sounds warm and confident. It also promises a call tomorrow, although you have not arranged one.

The missing piece was your process. The assistant needed to know that you first read the form, then email questions, then arrange a call. A polished sentence cannot supply those facts.

You could explain the process each time. Or you could save it beside the work, tell the assistant where to find it, and improve that note when your process changes.

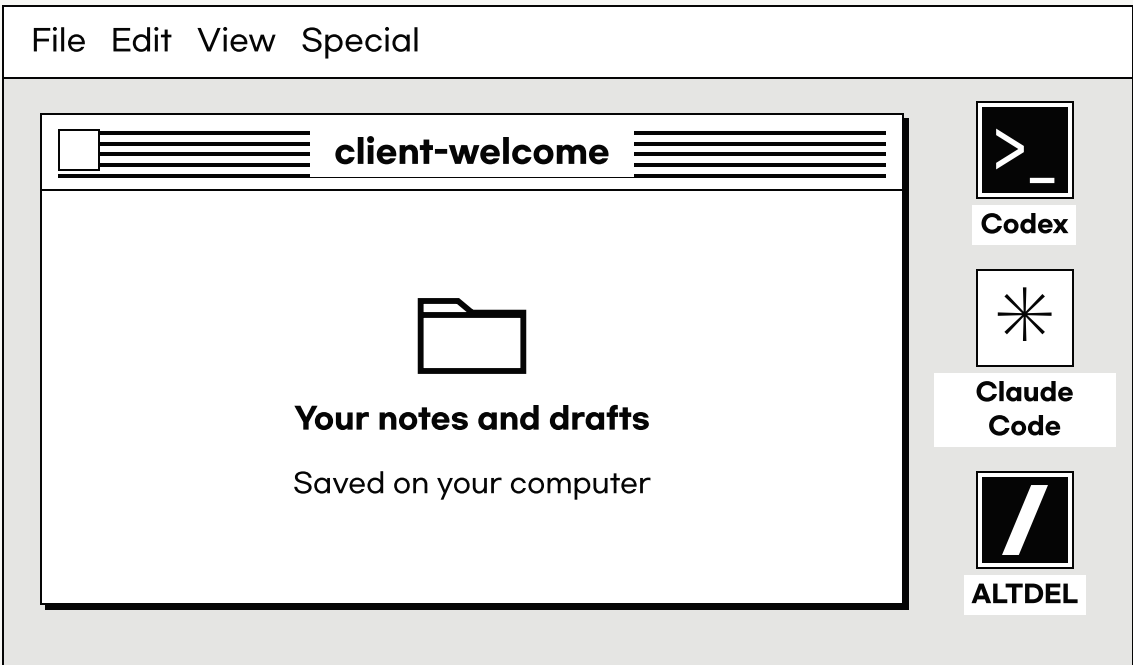
That is the starting point of the Monolith workspace: a place for the work and the instructions that make it useful.

In this guide, you will build a small **client-welcome** folder. You will ask Codex to create its notes, use them to write a welcome message, then see how the same arrangement grows with your work. You remain the person who knows whether the result is right.

The app works with your folder

You will use Codex in the desktop app for this walkthrough. A project points to a folder on your computer. When you ask it to save a file there, that file becomes part of your workspace.

A reply in chat and a saved file are different things. A reply can explain the work. A file gives you something you can open, change and use again. Ask for both when you need to inspect a result.



The method can travel to other tools that work with local files. Here, we will follow one setup path in Codex so you can concentrate on the work.

These notes are ordinary text

The notes in this workspace are **Markdown files**. Their names end in **.md**, just as a PDF file's name ends in **.pdf**. You can read and change the words inside them.

Markdown uses a few marks to organize the text. A **#** starts a heading. A line beginning with **-** becomes a list item. The marks help you find the parts of a note without needing a page-design app.

A Markdown note

A heading

A sentence in ordinary text.

- One item in a list
- Another item in the list

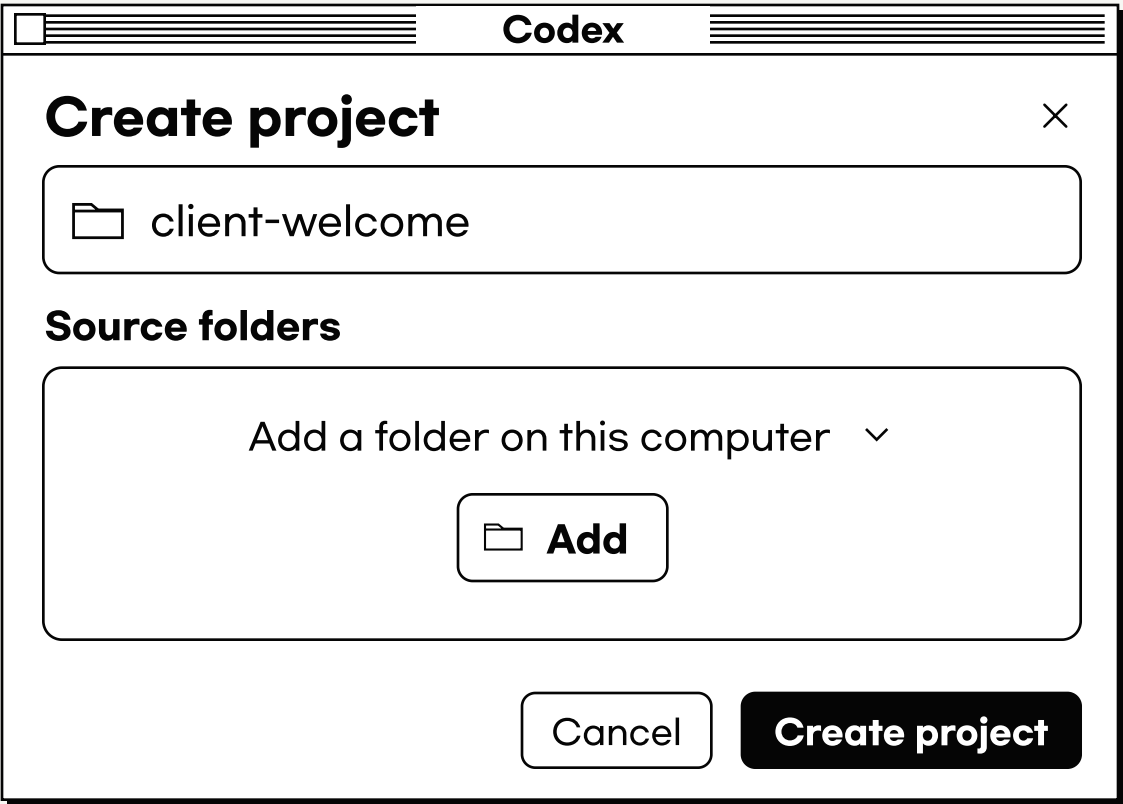
Codex can write these files from your description and edit them when you ask. You can also open them in Notepad or TextEdit. If you use TextEdit, choose plain text when saving. A separate editor is optional here.

Think of the files as instructions and working notes that you can inspect. You are not writing software. You are making your knowledge available in a form the assistant can read beside the work.

Connect a place to work

If you need the app, follow [OpenAI’s desktop setup](#). Open the app, sign in and choose Codex. Create a new project. For this walkthrough, name it **client-welcome**. The name helps you recognize this work in the app.

Under **Source folders**, choose **Add**. This connects the project to the folder that will hold your notes and drafts.

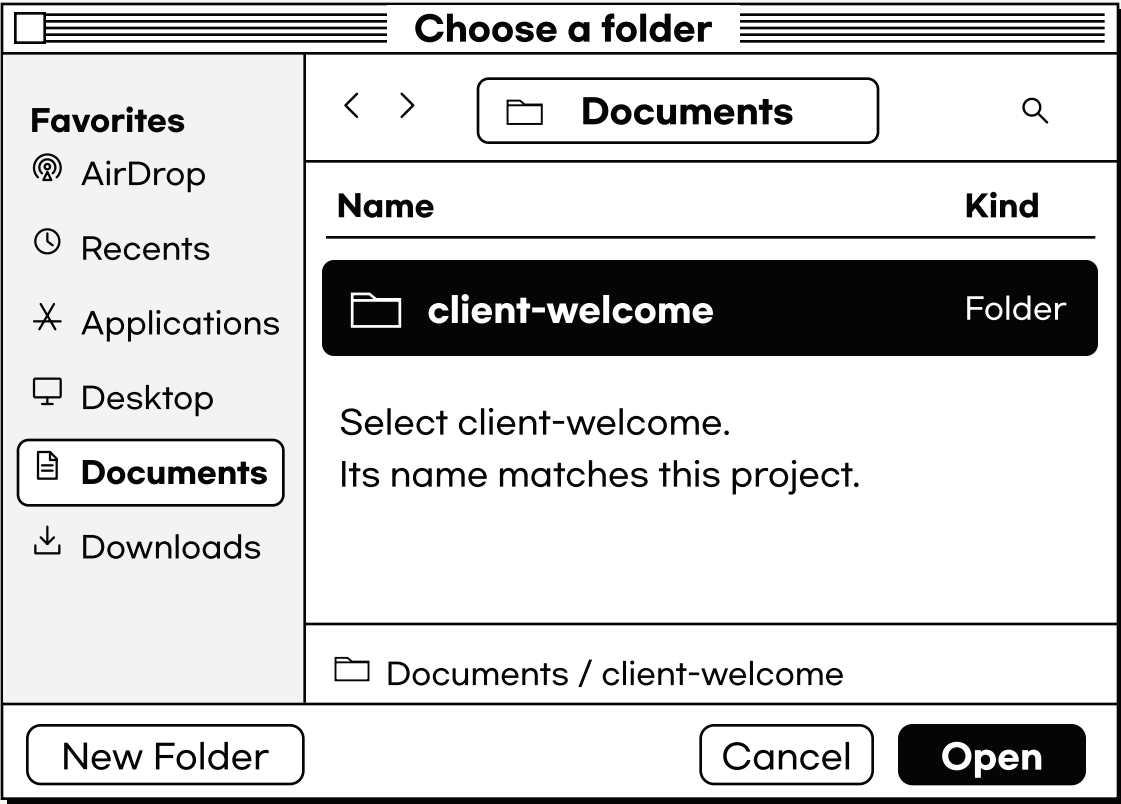


Create a project in Codex.

You are choosing where Codex should work. The next screen lets you select that place on your computer.

Choose your project folder

In the folder chooser, open **Documents**. Select **New Folder** and name it **client-welcome**. Select that folder, then choose **Open**. Back in Codex, choose **Create project**.



Choose the client-welcome folder.

Start a task inside this project. Ask, “Which folder are we working in?” Check that the reply ends with **Documents/client-welcome**. This checks the selected location. After creating the notes, open the saved-file links and check that they belong to this folder too.

Ask Codex to create the notes

You do not need to write the files by hand. Describe the work and ask Codex to put that information in the right place. For this first run, use the brief below.

In this folder, create AGENTS.md, CONTEXT.md and REFERENCES.md.

This project produces short welcome notes for new clients. Use plain language and only supplied names, fees and timing.

My first task is a welcome note for a client who has just sent an intake form. These are the confirmed steps: we read the form, email any questions, then arrange a first call. No call date or reply deadline has been set.

Put the project purpose and lasting rules in AGENTS.md. Put the goal, reader and result in CONTEXT.md. Put the confirmed steps in REFERENCES.md.

In AGENTS.md, say to read both notes before drafting. Save the files here and give me links so I can open them.

The key instruction is **save the files here**. If the response only shows text to copy, ask Codex to create the actual files in your project folder.

Next, open each file. You are checking whether Codex understood your brief before that brief shapes the work.

Keep the lasting rules together

AGENTS.md is the main guide for this Codex project. It gives the assistant a purpose and tells it what to read. Its rules apply to more than one welcome note.

 **AGENTS.md** 

Client welcome

Help me write clear welcome notes for new clients.

Before drafting, read **CONTEXT.md** and **REFERENCES.md**.

- Use plain language.
- Use only supplied names, fees and timing.
- Save drafts here when I ask for a file.

The first sentence explains the job. The next points to two files with more detail. The bullets set habits that should hold each time: plain language, supplied facts and a saved draft when requested.

The client's name and the details of one call do not belong in these lasting rules. Those details change with the job. Keeping them elsewhere lets this guide stay useful without rewriting it for each client.

Codex recognizes **AGENTS.md** as an instruction file. Names such as **CONTEXT.md** and **REFERENCES.md** are choices in this workspace; the instruction to read them makes their role clear. [OpenAI's guide](#) explains how project instructions are found.

Explain the job and supply the facts

CONTEXT.md explains the work you are trying to do.
REFERENCES.md supplies the facts or source material the assistant needs to use. Open yours and compare their meaning with these examples.

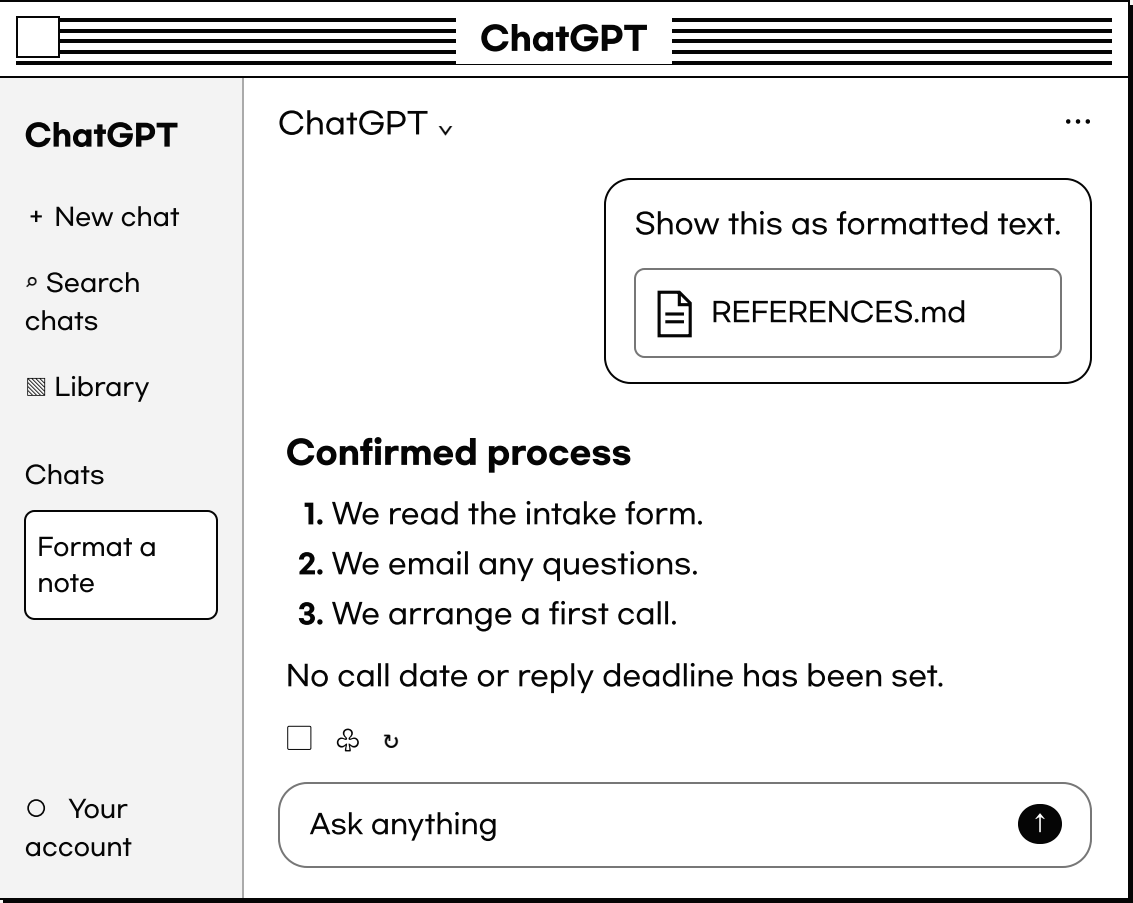
	CONTEXT.md
---	--

The context explains the goal and reader. The reference supplies the sequence of events and what has not been set.

References can also point to a brief, table or example. Say what each source is for so the assistant knows why it matters.

See the formatted note

You can view the same Markdown file as formatted text. Attach **REFERENCES.md** in ChatGPT and ask it to display the contents. The file is the input to this request.



Attach the file to display its saved text as a formatted note.

The heading and list look different, but the words carry the same facts. You can keep working with plain text in Codex. See the [Markdown syntax guide](#) for more about the marks.

Change the file through the app

Suppose you decide that a welcome note should be brief enough to read on a phone. Tell Codex to make that requirement part of the current work:

In CONTEXT.md, change the Result line to:
A welcome note no longer than 120 words.

Save the change. Leave the confirmed process in REFERENCES.md unchanged. Give me a link to the updated file.

CONTEXT.md

Current work

Goal: Explain what happens after a client sends an intake form.
Reader: A new client waiting for the next step.
Result: A welcome note no longer than 120 words.

Open the linked file and look for the new line. This is how you check a file edit: inspect the saved text, rather than relying only on the message saying it was changed.

The next task can read this length requirement from the file. If you only ask for a short note in chat, that request may be useful today, but it will not update the workspace by itself.

Use the notes to make something

The workspace now holds the purpose, the reader, the length and the process. You can ask for a real result without typing that background again.

Read AGENTS.md, CONTEXT.md and REFERENCES.md.

Write the welcome note for the client who has just sent an intake form. Explain what happens next in the order given in the references.

Use a warm, direct tone. Do not add a client name, a call date or a reply deadline. Keep it within the length set in CONTEXT.md.

Save it as welcome-note-v1.md in this folder. Show me the note and give me a link to the saved file.

welcome-note-v1.md means the first saved version of your welcome note. The name tells you what it contains before you open it. The .md ending tells you it is a Markdown text file.

Your prompt selects today's action. Your notes supply the background. Both have a job: one says what to do now; the others describe the work and the facts it should respect.

Open the result and check its meaning

Open **welcome-note-v1.md**. The words may differ each time, but the result should explain the same confirmed process. A note that meets the brief could read like this:

welcome-note-v1.md

Thanks for sending your intake form

We will read through it and email you with any questions. Then we will arrange a first call.

Thanks for sharing this with us. We look forward to learning more about your project.

Compare the draft with **REFERENCES.md**. Does it say the form comes first, followed by questions and a call? Has it promised a date or response time you did not supply? Then check the length and tone against your request.

A sentence can sound helpful and still make an unsupported promise. Checking the actual draft against your notes is how you catch that difference.

If one sentence is wrong, name it and ask Codex to revise the draft. If the process itself has changed, update the reference first, then ask for a new draft. That keeps tomorrow's work from repeating yesterday's mistake.

Read what this task needs

The welcome note worked from a few facts. Your whole business will have far more: past drafts, product notes, client questions and plans. Giving all of it to every task creates a second problem. The assistant must sort through material that may have nothing to do with the job.

A model works with a limited amount of information at once, called its **context window**. Text is counted in **tokens**, which can be words, parts of words or punctuation. Instructions, supplied material and the response all use capacity. Limits differ by model. [OpenAI explains these limits](#).

Read for this task

The project guide

The client process

The current request

Leave aside

Old website layouts

Unrelated drafts

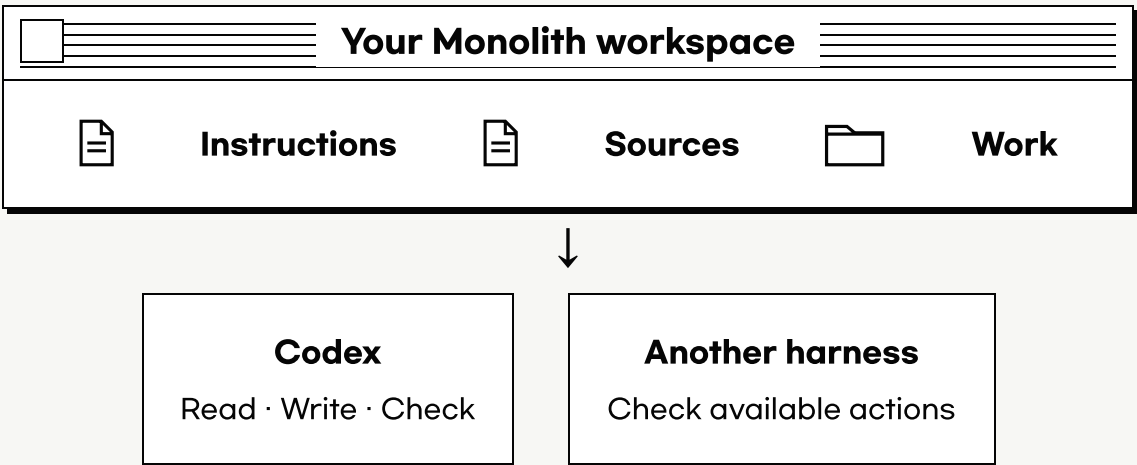
For a welcome note, the client process matters. Last month's website layout does not. A useful workspace tells the assistant where to look and what to leave aside.

The goal is to give the task the right material. More context is helpful when it answers a real question; unrelated context is still unrelated, even when it fits.

Keep the method across tools

The **model** interprets information and produces a response. The **harness** is the software around it that manages the conversation, files and tools. Codex gives you one way to work with these pieces in an app.

Your Monolith workspace stores the purpose, facts and process in readable files. That makes the method portable. You can use those files with another model or harness that can read them and carry out the required steps.



When changing apps, tell the new assistant to read the main guide. Check that it can open the source files, write the requested result and use any needed tools. Different apps may recognize different instruction filenames. Codex uses AGENTS.md; another app may need a short entry file that points to the same guidance.

A chat app with attachments can help draft from the notes you supply. A harness with file access can also save and revise them in place. The written process can stay the same while the available actions differ.

Give the request five useful parts

A good request answers five questions. Some answers may already live in your project files, so you do not need to repeat them in every message.

	Five questions for the request
Identity / role	What kind of help do you need?
Task	What should it do now?
Context	What information should it use?
Constraints	What limits must it respect?
Output format	What should the result look like?

For the welcome note, the role is a helpful editor. The task is to write the note. The context is the confirmed client process. The limits exclude invented timing and set the length. The output is a named Markdown file.

Try this on your next request: underline the action, name the source, and describe the result you expect to open. If a necessary fact is missing, supply it or ask the assistant to identify the gap.

A role helps set the kind of work. It does not give the model facts about your business that you have not supplied.

Separate a decision from the work it controls

A short note needs few choices. A longer guide needs an order, a level of detail and a clear audience. Ask for the outline before asking for every page. You can correct the plan while it is still easy to change.

Read the project notes. Propose an outline for a client welcome guide. Save it as `welcome-outline.md`. Use only supported topics and list any missing information. Stop after the outline.

Once the outline is useful, ask for one section using that saved file. Read the result, then continue. This gives each request an input you can name and an output you can inspect.



welcome-outline.md

Choose the topics and order.



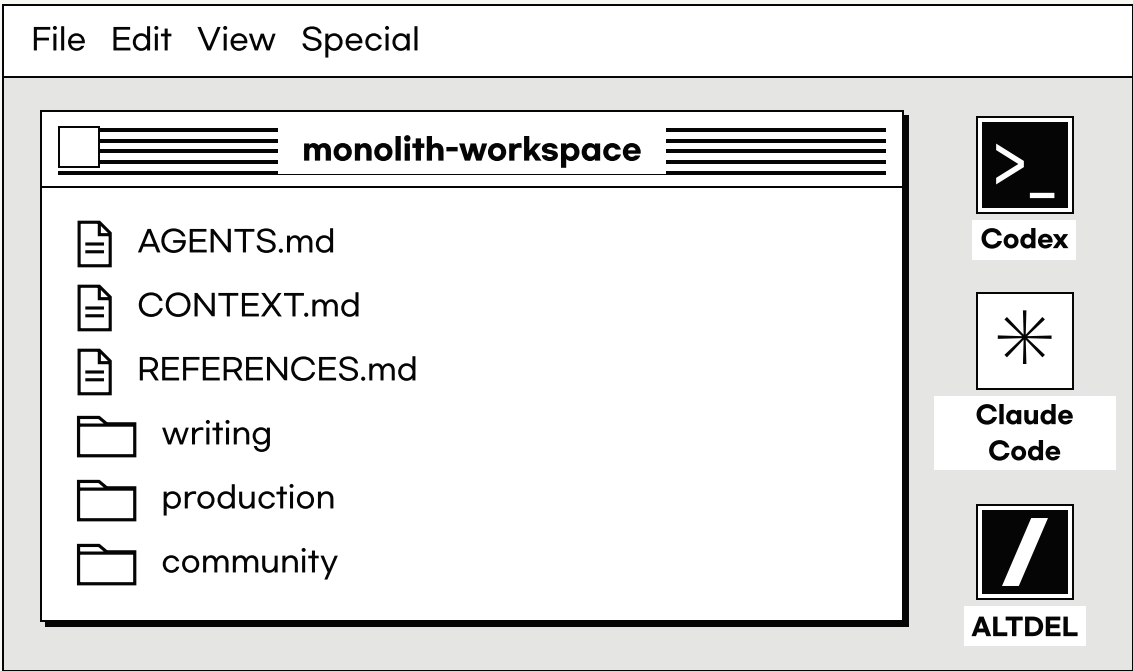
welcome-guide-v1.md

Write from the outline.

The same care applies to large source material. Name the section that matters to the task. Keep a table's headings with its values. If you divide a long document, label each part so the assistant knows what it is reading and how the parts relate.

Build a wider workspace

Your welcome note is one kind of writing. Turning it into a handout and answering audience questions need different instructions. Give each kind of work its own area.



In **monolith-workspace**, **writing** creates text, **production** turns it into assets, and **community** handles questions you supply.

Unzip **monolith-reader-kit.zip** and add **monolith-workspace** as another Codex project, using the earlier setup steps. Keep **client-welcome** as your first exercise. The new project includes a sample at **writing/welcome-note-v1.md**; it does not move your first exercise's files.

Route each kind of work

Open the expanded project's **AGENTS.md**. It should explain the shared purpose, then route each request to the area that owns the work.

AGENTS.md		
Task	Go to	Read
Draft or revise text	writing/	writing/CONTEXT.md REFERENCES.md
Plan or build a handout	production/	production/CONTEXT.md The source draft you name
Draft a member reply	community/	community/CONTEXT.md The question you supply

Read the first row as a sentence: “For writing, go to the writing folder, read its context file and use the shared reference.” The local file supplies the steps that do not belong in every task.

A production request needs the production guide and a named source draft. A community reply needs the member's actual question. The map helps the assistant choose those inputs without reading every nearby file.

Keep shared rules, such as using supplied facts and clear filenames, in the main guide. Put detailed writing or layout methods in the area that uses them.

Load the right instructions

In the monolith-workspace project, give Codex a short request:

Read AGENTS.md. Go to the writing area, read its CONTEXT.md and tell me what this area does, what it needs from me and where it saves work. Do not draft yet.



Codex / writing



Ready for your writing task

I draft and revise business text using the project references and the source you supply.

I save the named draft in writing/.

What would you like to write?

The response should explain the area before starting a draft.

This request checks the route. The response should name the sources and ask what you want to write, without inventing an assignment.

The chosen area supplies the working role. Different instructions let the same model write a note or plan a layout.

Open **writing/CONTEXT.md** and compare it with the reply. Purpose explains why the area exists. Inputs say what to read. Process describes the work. Output says what you should be able to open afterward.

Let a missing input stay visible

Now try a production request that refers to a file the template does not contain:

Read AGENTS.md and production/CONTEXT.md. Make a one-page welcome handout using writing/welcome-guide-v1.md. Before building, check that the source file exists.

Codex / production

 **writing/welcome-guide-v1.md**

Not found

I found writing/welcome-note-v1.md,
but not the guide named in your request.

Supply the guide, ask writing to create it,
or tell me to use the existing note.

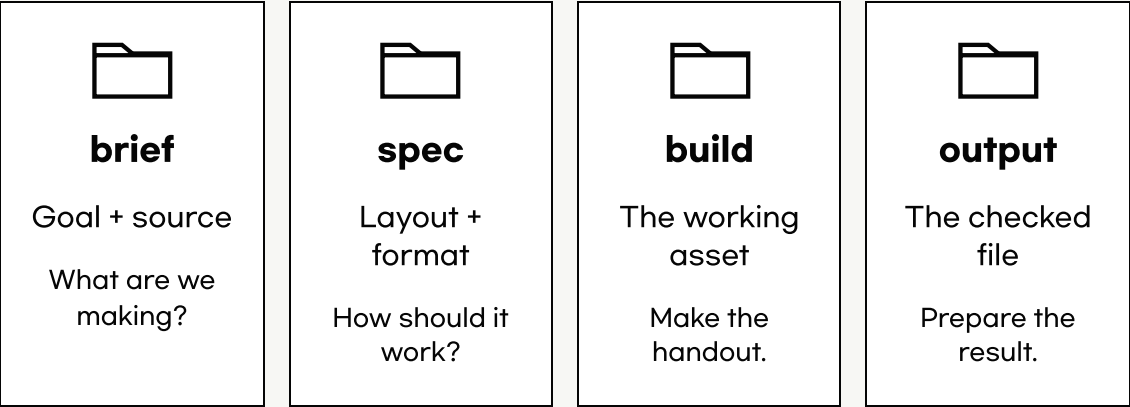
The requested guide is missing. There is a **welcome-note-v1.md**, but a note and a guide are not the same file. A useful assistant identifies that gap and explains what it needs next.

You can supply the guide, ask the writing area to create it, or choose the existing note as the source. For the next exercise, use the note.

This is a normal part of the process. A later step depends on an earlier result. Finding the missing result before building prevents a well-presented asset from being based on guessed content.

Plan the handout in stages

Production is a work area. Inside it, **brief**, **spec**, **build** and **output** are stages of this handout job. Simpler jobs may combine steps.



Each folder has its own  **CONTEXT.md**

The **brief** defines the purpose, audience and source. The **spec**, short for specification, describes the layout and format in enough detail to guide the build. **Build** makes the asset from those choices. **Output** checks the result and puts the usable file in its delivery folder.

Each stage has a local **CONTEXT.md** that names its inputs and output. The next stage reads the file made by the previous stage, alongside any source facts it still needs.

For our handout, the written note supplies the facts. The brief sets the goal. The spec decides how the one-page layout should present those facts. A layout choice can change the presentation; it should not invent a new client process.

Save each stage's result

Start with the existing welcome note. Ask Codex to work through the first two stages separately so you can see their different jobs.

Read production/CONTEXT.md and the brief stage guide. Use writing/welcome-note-v1.md to plan a one-page client welcome handout. Save the brief as production/brief/welcome-handout.md. Stop so I can read it.

After reading the brief:

Read the spec stage guide and the saved brief. Write the layout and format choices to production/spec/welcome-handout.md. Keep the source facts unchanged. Stop after saving the spec.

Once the spec is useful, ask Codex to follow **production/build/CONTEXT.md**. The example produces **welcome-handout.html**, a page you can open in a browser. HTML is a common web-page file format.

Then ask it to follow **production/output/CONTEXT.md**. That step checks the source text and requested format, and places the passing file in **production/output/**. Saving it there does not publish it on the internet.

The stages make the choices inspectable. A vague layout can be corrected in the spec before it becomes a finished page.

Choose the method and tool

A **skill** stores a reusable method. It may include instructions, examples or scripts. A **tool** performs an action: read a file, inspect a page, make an image or write an output.

Skill / a method	Tool / an action
Review a draft Read the audience and source. Check the wording. Return useful edits.	Open a saved file Read its contents. Inspect a page Show its rendered layout.

A writing area might use an available editing skill after a draft exists. Production might use a design skill for a spec and a browser tool to inspect the built page. A PDF skill belongs where a PDF is actually needed.

The local context says when a method applies, which input it uses and what result it should return. It does not need to load every skill for every job.

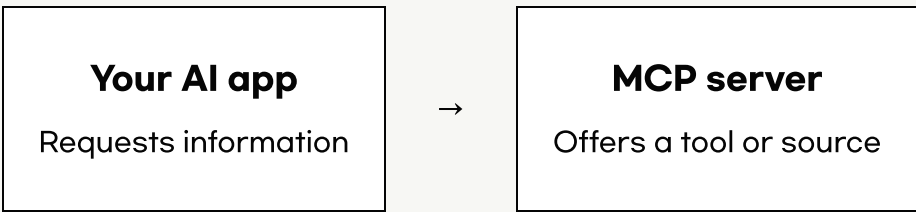
Before using one, ask the assistant to check that the skill or tool is available. If it cannot inspect a page in the current app, it should name that missing check. A successful file save alone does not prove the layout was reviewed.

For the supplied handout, use the tools already available in Codex. No new plugin is required by the template.

Connect outside information when the job needs it

Your first exercises use local files. Later, a job might need a customer record, a document in another app or a calendar entry.

MCP, the Model Context Protocol, is a standard way for an AI app to connect to services that provide tools or information. The app connects to an MCP server, which offers the capabilities it supports. The protocol does not decide the whole process for you. [The MCP overview](#) explains this arrangement.



A named question comes back to the community task.

For example, a community task might retrieve a named support question through an existing connection, then use the community guide to draft a reply. The question is the input; the guide still supplies the working instructions.

Writing “use the calendar” in a context file does not connect an account or grant access. The app needs a working connection and permission for the action. Choose the connection because a real task needs it.

The local practice folder needs no external connection. It teaches you where one would belong when your work calls for it.

Use names that lead back to the right work

A filename should help you find the result without opening every file. Include the subject, the kind of work and a version or date when those details matter.

	Find a specific result	
	welcome-note-v1.md	First draft
	welcome-note-v2.md	Revised draft
	2026-09-13-call-notes.md	Date and subject

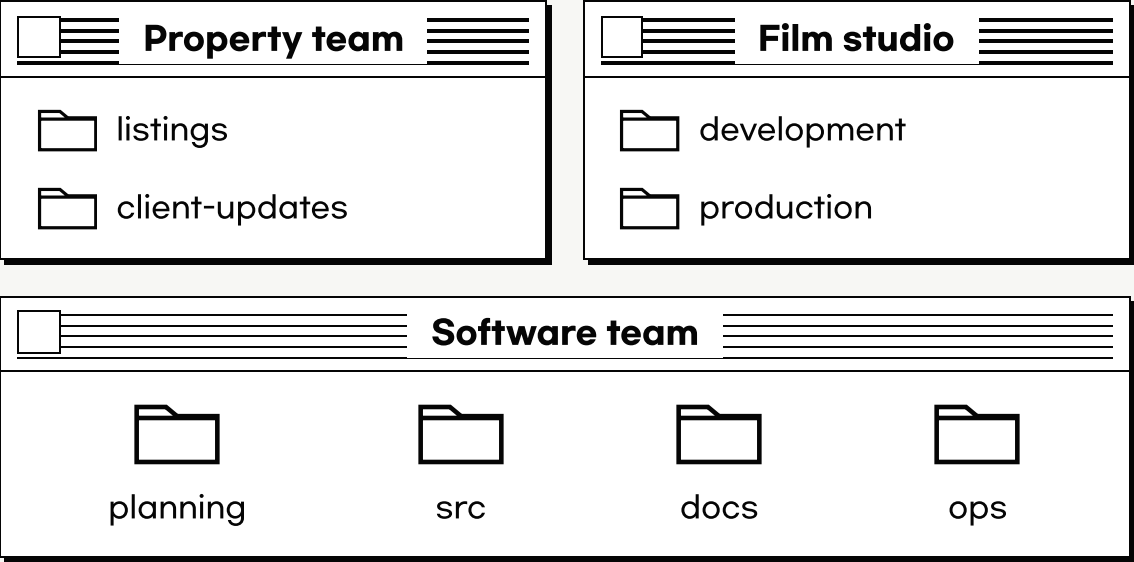
Ask for the exact file you want to continue. “Revise welcome-note-v1.md” gives a clearer starting point than “use that note we made.” In a new task, also ask the assistant to read the project guide and relevant context.

Put a lasting naming convention in AGENTS.md. Then the writing and production areas can follow the same pattern without inventing a new naming system for each task.

Use versions to distinguish drafts and stage folders to distinguish jobs. Within production/, **brief/welcome-handout.md** describes the goal; **spec/welcome-handout.md** describes the layout. Their shared subject connects them. Their folders explain which role each file plays.

Make the areas your own

Change the work described inside the files as well as the names outside them. A writing area for a property team needs different sources, language and checks from one used by a film studio.



For each area, answer four questions: What work belongs here? What source starts it? How do we do it? What file should exist when it is done?

A client business can keep client-specific facts in each client's own folder and share general templates separately. An app team can keep planning, code, documentation and operations in their existing locations, with guides that map to those locations.

Start with the areas you actually use. When a new type of work appears, write its context and add a route in the main guide. Update the audience, source material, process and output rules together so the new folder has a real job.

Repair the file that caused the problem

Use the result of a real task to decide what to improve. These seven common problems point to different repairs.

1

Overloaded main guide

Keep purpose and routes there; move detailed methods locally.

2

Missing route

Name the request, destination and file to read.

3

Too many work areas

Group work that uses the same instructions and sources.

4

Vague identity

Describe the job and reader, not a list of personality traits.

5

Stale context

Update the changed fact or process in its source file.

6

One flat file pile

Group related work and use names that reveal its purpose.

7

Structure before use

Try a real task before adding another folder or rule.

A bigger guide is not always a better guide. Make one focused correction, run a small task and open the result. Keep the changes that make the next task easier to understand and check.

Put the whole workspace to work

The welcome note gave you a first result. The larger workspace showed what happens when that result becomes an input to another kind of work.

You now have a main guide that selects an area, local context that explains the job, references that supply the facts, and working files that carry results between stages. Skills and tools help where a step needs them.

For your next task, start with the outcome you want. Ask Codex to follow the route, read the relevant context and name the inputs. If something is missing, supply it before the step that depends on it. Open each meaningful result and improve the saved instructions when the work reveals a gap.

The method belongs to your workspace. You can bring it to another model or harness by giving that tool the files and checking that it can perform the required actions. Keep your sources and output checks intact when you switch.

monolith-reader-kit.zip contains **client-welcome/** for the first exercise and **monolith-workspace/** for the wider example. The second folder includes the main guide, local contexts, four production-stage guides and the welcome note. It leaves the requested welcome guide absent so you can try the missing-input exercise.

Use the file structure to support the work you already understand. Then let real use show you what the next guide needs to say.