Create skills

Skills let teams capture institutional knowledge and turn it into reusable, shareable workflows. Skills help Codex behave consistently across users, repositories, and sessions, which is especially useful when you want standard conventions and checks applied automatically.

A **skill** is a small bundle consisting of a `name`, a `description` that explains what it does and when to use it, and an optional body of instructions. Codex injects only the skill's name, description, and file path into the runtime context. The instruction body is never injected unless the skill is explicitly invoked.

Decide when to create a skill

Use skills when you want to share behavior across a team, enforce consistent workflows, or encode best practices once and reuse them everywhere.

Typical use cases include:

- Standardizing code review checklists and conventions
- Enforcing security or compliance checks
- Automating common analysis tasks
- Providing team-specific tooling that Codex can discover automatically

Avoid skills for one-off prompts or exploratory tasks, and keep skills focused rather than trying to model large multi-step systems.

Create a skill

Use the skill creator

Codex ships with a built-in skill to create new skills. Use this method to receive guidance and iterate on your skill.

Invoke the skill creator from within the Codex CLI or the Codex IDE extension:

```
$skill-creator
```

Optional: add context about what you want the skill to do.

```
$skill-creator

Create a skill that drafts a conventional commit message based on a
short summary of changes.
```

The creator asks what the skill does, when Codex should trigger it automatically, and the run type (instruction-only or script-backed). Use instruction-only by default.

The output is a `SKILL.md` file with a name, description, and instructions. If needed, it can also scaffold script stubs (Python or a container).

Create a skill manually

Use this method when you want full control or are working directly in an editor.

1. Choose a location (repo-scoped or user-scoped).

   ```
   # User-scoped skill (macOS/Linux default)
   mkdir -p ~/.codex/skills/<skill-name>

   # Repo-scoped skill (checked into your repository)
   mkdir -p .codex/skills/<skill-name>
   ```

2. Create `SKILL.md`.

   ```
   ---
   name: <skill-name>
   description: <what it does and when to use it>
   ---

   <instructions, references, or examples>
   ```

3. Restart Codex to load the skill.

Understand the skill format

Skills use YAML front matter plus an optional body. Required fields are `name` (non-empty, at most 100 characters, single line) and `description` (non-empty, at most 500 characters, single line). Codex ignores extra keys. The body can contain any Markdown, stays on disk, and isn't injected into the runtime context unless explicitly invoked.

Along with inline instructions, skill directories often include:

- Scripts (for example, Python files) to perform deterministic processing, validation, or external tool calls
- Templates and schemas such as report templates, JSON/YAML schemas, or configuration defaults
- Reference data like lookup tables, prompts, or canned examples
- Documentation that explains assumptions, inputs, or expected outputs

<FileTree class="mt-4" tree={[{ name: "my-skill/", open: true, children: [{ name: "SKILL.md", comment: "Required: instructions + metadata", }, { name: "scripts/", comment: "Optional: executable code", }, { name: "references/", comment: "Optional: documentation", }, { name: "assets/", comment: "Optional: templates, resources", },], },]} />

The skill's instructions reference these resources, but they remain on disk, keeping the runtime context small and predictable.

For real-world patterns and examples, see agentskills.io and check out the skills catalog at github.com/openai/skills.

Choose where to save skills

Codex loads skills from these locations (repo, user, admin, and system scopes). Choose a location based on who should get the skill:

- Save skills in your repository's `.codex/skills/` when they should travel with the codebase.
- Save skills in your user skills directory when they should apply across all repositories on your machine.
- Use admin/system locations only in managed environments (for example, when loading skills on shared machines).

For the full list of supported locations and precedence, see the "Where to save skills" section on the Skills overview.

See an example skill

```
---
name: draft-commit-message
description: Draft a conventional commit message when the user asks
        for help writing a commit message.
metadata:
  short-description: Draft an informative commit message.
---
```

```
Draft a conventional commit message that matches the change summary
        provided by the user.

Requirements:

- Use the Conventional Commits format: `type(scope): summary`
- Use the imperative mood in the summary (for example, "Add", "Fix",
        "Refactor")
- Keep the summary under 72 characters
- If there are breaking changes, include a `BREAKING CHANGE:` footer
```

Example prompt that triggers this skill:

```
Help me write a commit message for these changes: I renamed
`SkillCreator` to `SkillsCreator` and updated the sidebar.
```

Check out more example skills and ideas in the github.com/openai/skills repository.

Follow best practices

- Be explicit about triggers. The `description` tells Codex when to trigger a skill.
- Keep skills small. Prefer narrow, modular skills over large ones.
- Prefer instructions over scripts. Use scripts only when you need determinism or external data.
- Assume no context. Write instructions as if Codex knows nothing beyond the input.
- Avoid ambiguity. Use imperative, step-by-step language.
- Test triggers. Verify your example prompts activate the skill as expected.

Troubleshoot skills

Skill doesn't appear

If a skill doesn't show up in Codex, make sure you enabled skills and restarted Codex. Confirm the file name is exactly `SKILL.md` and that it lives under a supported path such as `~/.codex/skills`.

If you use symlinked directories, confirm the symlink target exists and is readable. Codex also skips skills with malformed YAML or `name`/`description` fields that exceed the length limits.

Skill doesn’t trigger

If a skill loads but doesn’t run automatically, the most common issue is an unclear trigger. Make sure the `description` explicitly states when to use the skill, and test with prompts that match that description.

If two or more skills overlap in intent, narrow the description so Codex can select the correct one.

Startup validation errors

If Codex reports validation errors at startup, fix the listed issues in `SKILL.md`. Most often, this is a multi-line or over-length `name` or `description`. Restart Codex to reload skills.