ChatGPT

Best practices for your `AGENTS.md` (Codex PR-first workflow)

Large language model agents like Codex read `AGENTS.md` before doing any work. These files are concatenated from the agent's global directory, the repository root and any nested directories, with later files overriding earlier ones [1]. The file acts as a briefing packet that teaches the agent how to build, test and structure your project, which conventions to follow, how to format pull requests and what operations are allowed. Keeping the instructions short and precise makes the agent faster and easier to control [2]. The practices below combine official guidance and community experience to maximise the value of your `AGENTS.md`.

Layer your guidance

- **Use multiple scopes.** Add persistent defaults in `~/.codex/AGENTS.md` to define global rules (e.g. always run tests before merging) [1]. In each repository, place an `AGENTS.md` at the root and optionally in sub-directories. Codex merges files from the root down to your current working directory, with deeper files overriding earlier ones [1]. For monorepos, include an `AGENTS.md` (or `AGENTS.override.md`) inside each package so that packages with different frameworks or versions can have tailored instructions [3].

- **Override carefully.** Use `AGENTS.override.md` when a sub-module needs completely different guidance. Configure fallback filenames (e.g. `TEAM_GUIDE.md`) in the Codex config if your project uses a different naming convention [4]. The Codex CLI automatically injects each discovered file as a separate user-role message, clearly indicating the directory it applies to [5].

Keep it concise and actionable

- **Limit length.** Long instruction files slow the agent and bury useful signal. Factory's AGENTS.md guide recommends aiming for **≤150 lines** [2]. Use headings (`#`) and bullet lists to organise sections such as *Build & Test*, *Architecture*, *Security*, *Git Workflows* and *Conventions* [6].

- **Use concrete commands.** Wrap commands in back-ticks so they can be copied verbatim. Provide exact flags and file paths; avoid ambiguous descriptions [7]. When possible, specify file-scoped commands (e.g. `pytest path/to/test.py::test_function`, `mypy path/to/module.py`) for faster feedback [8].

- **Update alongside code.** Treat the `AGENTS.md` file like code. When build steps, test commands or conventions change, reviewers should expect a corresponding update in `AGENTS.md` [9]. To avoid duplication, link out to READMEs or design docs rather than pasting large sections of text [10].

- **Make requests precise.** Clearly state what problem the agent should solve, what files it may edit and how success is measured. Precise guidance improves the agent's plan and reduces undesired exploration [11].

Document build, test and verification commands

- **Setup, lint and type-check.** Provide exact commands for installing dependencies, running linters and performing type checks. For example:

```
# Python example – adapt to your stack
pip install -r requirements.txt   # install deps
flake8 .                          # lint
mypy .                            # type-check
pytest                            # run all tests
pytest path/to/test.py::test_func # run a single test
```

Use single-file or targeted commands to reduce turnaround time [8]. Specify the order in which to run these commands and require a green test suite before opening a PR [12].

- **Continuous integration parity.** Mirror the CI pipeline in your instructions (tests, lint, type-check, build) so that local runs match the gatekeeping conditions. Require objective proof before merging: tests pass, lint and type checks succeed and the diff is confined to permitted directories [13].

- **Test-first mode.** Encourage the agent to write or update unit tests before implementing a new feature or fixing a bug. For regressions, instruct it to reproduce the bug with a failing test and then iterate until the test passes [14].

Describe project structure, conventions and standards

- **Architecture overview.** Briefly describe major modules and data flow. Point the agent to entry points (e.g. `main.py` or `src/App.tsx`), routing files and where specific subsystems live (e.g. *components*, *models*, *queries*). This helps the agent start where a human would [15].

- **Coding style and patterns.** List the languages and versions you use, framework preferences (e.g. React functional components with hooks, or Python typing), naming conventions and folder layout. Provide do/don't lists to enforce your conventions. For example, you might specify "use vectorised pandas operations for large data sets" and "avoid loops where possible," or "use small components and focused modules; avoid monolithic files" [16]. Explain why these preferences exist so that the agent can weigh trade-offs [17].

- **Good and bad examples.** Link to real files that represent your ideal patterns and call out legacy code to avoid [18]. For example: "avoid `legacy_dashboard.py`; instead, follow `modern_dashboard.py` for structure." Real examples improve fidelity and reduce drift. [18]

- **Design system and API docs.** If the project uses a design system or external APIs, point the agent to the relevant documentation or typed clients. Indicate where tokens and components live and provide example usage [19]. This reduces guesswork when generating UI or API calls.

Provide safety and permission rules

- **Allowed versus confirmation.** List the operations the agent may perform without asking (reading files, running single-file tests, linting, type checking) and operations that require explicit approval (installing packages, deleting files, running full build or integration tests, pushing commits) [20]. Clear boundaries prevent accidental network calls, package installations or destructive actions.

- **Protect sensitive areas.** Identify directories or files that should never be edited (e.g. vendor code, generated artifacts, deployment scripts). Remind the agent never to commit secrets or modify `.env` files. Warn against disabling tests to make CI pass or adding new dependencies without justification.

Write a PR checklist and review rubric

- **Define "ready for review."** Require that branches are up to date with main, tests and linting pass, and that the diff is small and focused [12]. Encourage conventional commit prefixes (e.g. `feat:`, `fix:`) and ensure the PR description explains what changed, why it changed, how to test the change and any risks.

- **Review rubric.** Include a list of questions for the agent to self-check: Does the diff stay under a certain number of lines? Are there any unrelated formatting changes? Are public API changes documented? Are new features covered by tests? Are error messages clear? Is performance or security affected? Use this rubric to gate PRs before requesting a human review.

- **Draft vs ready.** Instruct the agent to open a draft PR when work is incomplete or tests are flaky. Only mark the PR ready once the checklist is satisfied and the code is polished.

Encourage planning, clarifications and incremental work

- **Ask when stuck.** If the agent is uncertain, instruct it to pause and ask clarifying questions or propose a short plan instead of guessing [21]. The plan can outline steps and files to edit before making changes. This avoids large speculative changes and makes it easier to adjust.

- **Small, atomic tasks.** Break work into small tasks with explicit acceptance criteria. The builder community found that tasks with tight scopes and linked files produce better results than broad open-ended tasks [22].

Use nested AGENTS.md files in monorepos

- **Per-package rules.** Large repositories often contain multiple technologies or versions. Place an `AGENTS.md` in each package or service directory to localise guidance for that area [3]. The agent automatically reads the closest file to the work it's doing, so each package can evolve independently.

Maintain and evolve the file

- **Single source of truth.** Avoid multiple conflicting instruction files; link to existing docs instead of duplicating content [10]. When something changes (e.g. switching from Jest to Vitest or from Make to Poetry), update `AGENTS.md` in the same PR so that the agent stays in sync [9].

- **Verify before merging.** Require objective evidence—passing tests, lint checks and type checks, and a clean diff—before merging agent-generated changes [13]. High test coverage (ideally 100 %) helps define what is allowed, what is forbidden and constrains the agent's behaviour [23]. If the agent drifts (e.g. modifies unrelated files or rewrites plans mid-execution), tighten the specification, revert noisy edits and restart with clearer instructions [24].

Example skeleton `AGENTS.md`

Below is an example outline you can adapt for your own projects. Replace the placeholder commands and paths with your real tooling. The comments explain the intent.

```
# Build & Test
- Install dependencies: `pip install -r
requirements.txt`  # 🛠 how to set up the environment
- Lint: `flake8 .`                                          # 🧹 enforce style
- Type check: `mypy .`                                       # 🔍 static typing
- Unit tests: `pytest`                                       # 🧪 run all tests
- Test a single file: `pytest tests/test_example.py::test_func`

# Architecture Overview
- The API lives in `src/api/` and uses FastAPI.             # 🗺 where major
modules reside
- Data models are in `src/models/` and database schemas in `src/db/schema.py`.
- Front-end React app lives in `frontend/`.                  # specify sub-
projects

# Conventions & Patterns
- Write new code in Python 3.11 and follow PEP 8.           # 🎨 coding style
- Prefer vectorised Pandas operations over loops.            # 📊 data science
convention
- Use functional React components with hooks.                # ⚛ UI patterns
- Do **not** hard-code secrets—use environment variables.  # 🔒 security rule
```

```
# Project Structure
- Source code: `src/`, tests: `tests/`.                    # 🗂 high-level layout
- SQL queries live in `queries/`.                            # for database work
- Reusable plots live in `visualisation/`.                   # example
domain-specific folder

# Safety & Permissions
- Allowed without confirmation: reading files, running `pytest` on a single
test, linting.
- Ask before: installing new packages, deleting files, pushing commits, running
full integration tests.
- Never modify files in `vendor/` or `generated/` directories.

# Git Workflow
- Always create a feature branch; never commit directly to `main`.
- Commit messages should follow Conventional Commits: `feat:`, `fix:`, etc.
- Open a draft PR if work is incomplete; mark ready once tests, lint and type
checks pass.

# PR Checklist
- □ Tests, lint and type-check all pass.
- □ PR description explains what changed, why, how to test and any risks.
- □ Diff is small and focused—no unrelated refactors.
- □ New features include unit tests; public API changes update docs.

# When Stuck
- If unsure, ask a clarifying question or propose a plan.
- Do not push speculative changes across multiple files without review.
```

Use this skeleton as a starting point and refine it with your own commands, directories and conventions. By layering clear, concise instructions and keeping them up to date, you give Codex the context it needs to create small, reviewable pull requests and follow your preferred workflow.

---

[1] [4] Custom instructions with AGENTS.md
https://developers.openai.com/codex/guides/agents-md/

[2] [6] [7] [9] [10] [11] [13] [24] AGENTS.md - Factory Documentation
https://docs.factory.ai/cli/configuration/agents-md

[3] [8] [12] [14] [15] [16] [17] [18] [19] [20] [21] [22] Improve your AI code output with AGENTS.md (+ my best tips)
https://www.builder.io/blog/agents-md

[5] Codex Prompting Guide
https://developers.openai.com/cookbook/examples/gpt-5/codex_prompting_guide/

[23] Best Practices for using Codex - Codex - OpenAI Developer Community
https://community.openai.com/t/best-practices-for-using-codex/1373143