Using PLANS.md for multi-hour problem solving

Codex and the `gpt-5.2-codex` model (recommended) can be used to implement complex tasks that take significant time to research, design, and implement. The approach described here is one way to prompt the model to implement these tasks and to steer it towards successful completion of a project.

These plans are thorough design documents, and "living documents". As a user of Codex, you can use these documents to verify the approach that Codex will take before it begins a long implementation process. The particular `PLANS.md` included below is very similar to one that has enabled Codex to work for more than seven hours from a single prompt.

We enable Codex to use these documents by first updating `AGENTS.md` to describe when to use `PLANS.md`, and then of course, to add the `PLANS.md` file to our repository.

`AGENTS.md`

`AGENTS.md` is a simple format for guiding coding agents such as Codex. We describe a term that users can use as a shorthand and a simple rule for when to use planning documents. Here, we call it an "ExecPlan". Note that this is an arbitrary term, Codex has not been trained on it. This shorthand can then be used when prompting Codex to direct it to a particular definition of a plan.

Here's an `AGENTS.md` section instructing an agent about when to use a plan:

```
# ExecPlans

When writing complex features or significant refactors, use an
        ExecPlan (as described in .agent/PLANS.md) from design to
        implementation.
```

PLANS.md

Below is the entire document. The prompting in this document was carefully chosen to provide significant amounts of feedback to users and to guide the model to implement precisely what a plan specifies. Users may find that they benefit from customizing the file to meet their needs, or to add or remove required sections.

```
# Codex Execution Plans (ExecPlans):

This document describes the requirements for an execution plan
        ("ExecPlan"), a design document that a coding agent can
        follow to deliver a working feature or system change. Treat
        the reader as a complete beginner to this repository: they
        have only the current working tree and the single ExecPlan
        file you provide. There is no memory of prior plans and no
        external context.

## How to use ExecPlans and PLANS.md

When authoring an executable specification (ExecPlan), follow PLANS.md
        _to the letter_. If it is not in your context, refresh your
        memory by reading the entire PLANS.md file. Be thorough in
        reading (and re-reading) source material to produce an
        accurate specification. When creating a spec, start from the
        skeleton and flesh it out as you do your research.

When implementing an executable specification (ExecPlan), do not
        prompt the user for "next steps"; simply proceed to the next
        milestone. Keep all sections up to date, add or split entries
        in the list at every stopping point to affirmatively state the
        progress made and next steps. Resolve ambiguities
        autonomously, and commit frequently.

When discussing an executable specification (ExecPlan), record
        decisions in a log in the spec for posterity; it should be
        unambiguously clear why any change to the specification was
        made. ExecPlans are living documents, and it should always be
        possible to restart from _only_ the ExecPlan and no other
        work.

When researching a design with challenging requirements or significant
        unknowns, use milestones to implement proof of concepts, "toy
        implementations", etc., that allow validating whether the
        user's proposal is feasible. Read the source code of
        libraries by finding or acquiring them, research deeply, and
        include prototypes to guide a fuller implementation.
```

Requirements

NON-NEGOTIABLE REQUIREMENTS:

* Every ExecPlan must be fully self-contained. Self-contained means that in its current form it contains all knowledge and instructions needed for a novice to succeed.
* Every ExecPlan is a living document. Contributors are required to revise it as progress is made, as discoveries occur, and as design decisions are finalized. Each revision must remain fully self-contained.
* Every ExecPlan must enable a complete novice to implement the feature end-to-end without prior knowledge of this repo.
* Every ExecPlan must produce a demonstrably working behavior, not merely code changes to "meet a definition".
* Every ExecPlan must define every term of art in plain language or do not use it.

Purpose and intent come first. Begin by explaining, in a few sentences, why the work matters from a user's perspective: what someone can do after this change that they could not do before, and how to see it working. Then guide the reader through the exact steps to achieve that outcome, including what to edit, what to run, and what they should observe.

The agent executing your plan can list files, read files, search, run the project, and run tests. It does not know any prior context and cannot infer what you meant from earlier milestones. Repeat any assumption you rely on. Do not point to external blogs or docs; if knowledge is required, embed it in the plan itself in your own words. If an ExecPlan builds upon a prior ExecPlan and that file is checked in, incorporate it by reference. If it is not, you must include all relevant context from that plan.

Formatting

Format and envelope are simple and strict. Each ExecPlan must be one single fenced code block labeled as `md` that begins and ends with triple backticks. Do not nest additional triple-backtick code fences inside; when you need to show commands, transcripts, diffs, or code, present them as indented blocks within that single fence. Use indentation for clarity rather than code fences inside an ExecPlan to avoid prematurely closing the ExecPlan's code fence. Use two newlines after every heading, use # and ## and so on, and correct syntax for ordered and unordered lists.

When writing an ExecPlan to a Markdown (.md) file where the content of the file *is only* the single ExecPlan, you should omit the triple backticks.

Write in plain prose. Prefer sentences over lists. Avoid checklists, tables, and long enumerations unless brevity would obscure meaning. Checklists are permitted only in the `***Progress***` section, where they are mandatory. Narrative sections must remain prose-first.

Guidelines

Self-containment and plain language are paramount. If you introduce a phrase that is not ordinary English ("daemon", "middleware", "RPC gateway", "filter graph"), define it immediately and remind the reader how it manifests in this repository (for example, by naming the files or commands where it appears). Do not say "as defined previously" or "according to the architecture doc." Include the needed explanation here, even if you repeat yourself.

Avoid common failure modes. Do not rely on undefined jargon. Do not describe "the letter of a feature" so narrowly that the resulting code compiles but does nothing meaningful. Do not outsource key decisions to the reader. When ambiguity exists, resolve it in the plan itself and explain why you chose that path. Err on the side of over-explaining user-visible effects and under-specifying incidental implementation details.

Anchor the plan with observable outcomes. State what the user can do after implementation, the commands to run, and the outputs they should see. Acceptance should be phrased as behavior a human can verify ("after starting the server, navigating to *[*http://localhost:8080/health*](http://localhost:8080/health)* returns HTTP 200 with body OK") rather than internal attributes ("added a HealthCheck struct"). If a change is internal, explain how its impact can still be demonstrated (for example, by running tests that fail before and pass after, and by showing a scenario that uses the new behavior).

Specify repository context explicitly. Name files with full repository-relative paths, name functions and modules precisely, and describe where new files should be created. If touching multiple areas, include a short orientation paragraph that explains how those parts fit together so a novice can navigate confidently. When running commands, show the working directory and exact command line. When outcomes depend on environment, state the assumptions and provide alternatives when reasonable.

Be idempotent and safe. Write the steps so they can be run multiple times without causing damage or drift. If a step can fail halfway, include how to retry or adapt. If a migration or destructive operation is necessary, spell out backups or safe fallbacks. Prefer additive, testable changes that can be validated as you go.

Validation is not optional. Include instructions to run tests, to start the system if applicable, and to observe it doing something useful. Describe comprehensive testing for any new features or capabilities. Include expected outputs and error messages so a novice can tell success from failure. Where possible, show how to prove that the change is effective beyond compilation (for example, through a small end-to-end scenario, a CLI invocation, or an HTTP request/response transcript). State the exact test commands appropriate to the project's toolchain and how to interpret their results.

Capture evidence. When your steps produce terminal output, short diffs, or logs, include them inside the single fenced block as indented examples. Keep them concise and focused on what proves success. If you need to include a patch, prefer file-scoped diffs or small excerpts that a reader can recreate by following your instructions rather than pasting large blobs.

Milestones

Milestones are narrative, not bureaucracy. If you break the work into milestones, introduce each with a brief paragraph that describes the scope, what will exist at the end of the milestone that did not exist before, the commands to run, and the acceptance you expect to observe. Keep it readable as a story: goal, work, result, proof. Progress and milestones are distinct: milestones tell the story, progress tracks granular work. Both must exist. Never abbreviate a milestone merely for the sake of brevity, do not leave out details that could be crucial to a future implementation.

Each milestone must be independently verifiable and incrementally implement the overall goal of the execution plan.

Living plans and design decisions

* ExecPlans are living documents. As you make key design decisions, update the plan to record both the decision and the thinking behind it. Record all decisions in the `Decision Log` section.
* ExecPlans must contain and maintain a `Progress` section, a `Surprises & Discoveries` section, a `Decision Log`, and an `Outcomes & Retrospective` section. These are not optional.
* When you discover optimizer behavior, performance tradeoffs, unexpected bugs, or inverse/unapply semantics that shaped your approach, capture those observations in the `Surprises & Discoveries` section with short evidence snippets (test output is ideal).
* If you change course mid-implementation, document why in the `Decision Log` and reflect the implications in `Progress`. Plans are guides for the next contributor as much as checklists for you.
* At completion of a major task or the full plan, write an `Outcomes & Retrospective` entry summarizing what was achieved, what remains, and lessons learned.

Prototyping milestones and parallel implementations

It is acceptable--and often encouraged--to include explicit prototyping milestones when they de-risk a larger change. Examples: adding a low-level operator to a dependency to validate feasibility, or exploring two composition orders while measuring optimizer effects. Keep prototypes additive and testable. Clearly label the scope as “prototyping”; describe how to run and observe results; and state the criteria for promoting or discarding the prototype.

Prefer additive code changes followed by subtractions that keep tests passing. Parallel implementations (e.g., keeping an adapter alongside an older path during migration) are fine when they reduce risk or enable tests to continue passing during a large migration. Describe how to validate both paths and how to retire one safely with tests. When working with multiple new libraries or feature areas, consider creating spikes that evaluate the feasibility of these features _independently_ of one another, proving that the external library performs as expected and implements the features we need in isolation.

Skeleton of a Good ExecPlan

```
# <Short, action-oriented description>

This ExecPlan is a living document. The sections `Progress`,
    `Surprises & Discoveries`, `Decision Log`, and `Outcomes &
    Retrospective` must be kept up to date as work proceeds.

If PLANS.md file is checked into the repo, reference the path to
    that file here from the repository root and note that this
    document must be maintained in accordance with PLANS.md.

## Purpose / Big Picture

Explain in a few sentences what someone gains after this change
    and how they can see it working. State the user-visible
    behavior you will enable.

## Progress

Use a list with checkboxes to summarize granular steps. Every
    stopping point must be documented here, even if it requires
    splitting a partially completed task into two ("done" vs.
    "remaining"). This section must always reflect the actual
    current state of the work.

- [x] (2025-10-01 13:00Z) Example completed step.
- [ ] Example incomplete step.
- [ ] Example partially completed step (completed: X; remaining:
    Y).

Use timestamps to measure rates of progress.

## Surprises & Discoveries

Document unexpected behaviors, bugs, optimizations, or insights
    discovered during implementation. Provide concise evidence.

- Observation: …
  Evidence: …

## Decision Log

Record every decision made while working on the plan in the
    format:

- Decision: …
  Rationale: …
```

Date/Author: …

Outcomes & Retrospective

Summarize outcomes, gaps, and lessons learned at major milestones or at completion. Compare the result against the original purpose.

Context and Orientation

Describe the current state relevant to this task as if the reader knows nothing. Name the key files and modules by full path. Define any non-obvious term you will use. Do not refer to prior plans.

Plan of Work

Describe, in prose, the sequence of edits and additions. For each edit, name the file and location (function, module) and what to insert or change. Keep it concrete and minimal.

Concrete Steps

State the exact commands to run and where to run them (working directory). When a command generates output, show a short expected transcript so the reader can compare. This section must be updated as work proceeds.

Validation and Acceptance

Describe how to start or exercise the system and what to observe. Phrase acceptance as behavior, with specific inputs and outputs. If tests are involved, say "run <project's test command> and expect <N> passed; the new test <name> fails before the change and passes after>".

Idempotence and Recovery

If steps can be repeated safely, say so. If a step is risky, provide a safe retry or rollback path. Keep the environment clean after completion.

Artifacts and Notes

Include the most important transcripts, diffs, or snippets as indented examples. Keep them concise and focused on what proves success.

```
    ## Interfaces and Dependencies

    Be prescriptive. Name the libraries, modules, and services to use
        and why. Specify the types, traits/interfaces, and function
        signatures that must exist at the end of the milestone.
        Prefer stable names and paths such as
        `crate::module::function` or `package.submodule.Interface`.
        E.g.:

    In crates/foo/planner.rs, define:

        pub trait Planner {
            fn plan(&self, observed: &Observed) -> Vec<Action>;
        }
```

If you follow the guidance above, a single, stateless agent -- or a human novice -- can read your ExecPlan from top to bottom and produce a working, observable result. That is the bar: SELF-CONTAINED, SELF-SUFFICIENT, NOVICE-GUIDING, OUTCOME-FOCUSED.

When you revise a plan, you must ensure your changes are comprehensively reflected across all sections, including the living document sections, and you must write a note at the bottom of the plan describing the change and the reason why. ExecPlans must describe not just the what but the why for almost everything.