Codex CLI

Agentic Engineering from First Principles

Daniel Vaughan

Codex CLI

Agentic Engineering from First Principles

Daniel Vaughan

This book is available at https://leanpub.com/codex-cli

This version was published on 2026-04-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Copyright and Trademarks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Trademarks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Disclaimer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Introduction

I Started This on a Train

The 07:42 from Audley End to Liverpool Street gives you about fifty minutes, depending on whether anything has gone wrong at Tottenham Hale. On a Tuesday in November I had my laptop open, a half-decent mobile data connection, and a vague intention to finally take Codex CLI seriously. I'd installed it a couple of weeks earlier, run a few commands, shrugged, and gone back to whatever I was already doing. That morning, I had a real problem: a Node service I'd written six months ago had started behaving oddly in staging, and I hadn't looked at it since. I didn't even have the repo fresh in my head.

I pointed Codex CLI at the project directory, described the symptom, and watched.

What happened next is the reason you're reading this book. It didn't just suggest a fix. It read the source files, traced the execution path, ran the test suite, found a second problem I hadn't mentioned, wrote a failing test for it, fixed both issues, and asked me whether it should open a pull request. By the time we pulled into Tottenham Hale—where, yes, something had gone wrong—I had a clean diff, a short explanation of the root cause, and a slight sense of unease about what I'd just seen.

I've been using AI coding assistants since the early Copilot days.[1] I've gone through phases with ChatGPT in a browser tab, Claude in a sidebar, various Copilot flavours, Cursor, Codeium, and a handful of others I've already forgotten the names of. They've all been useful in the way that autocomplete is useful. This was different. Not because it was smarter, exactly, but because it was *operating*, reading context, taking actions, managing state, recovering from errors, rather than responding. The distinction sounds subtle. It isn't.

That's when the book became necessary.

[1] GitHub Copilot, the original AI pair-programming tool, was first introduced in 2021 as a technical preview and is powered by OpenAI's code models. https://github.com/features/copilot

What Changed My Mind

The thing that actually changed my mind wasn't any single impressive output. It was understanding the architecture.

Most AI coding tools are, at their core, sophisticated text completers with a context window. You give them code; they give you more code. The quality varies, the context window keeps getting bigger, the integrations get slicker, but the underlying model is reactive. You prompt, it responds. You accept or reject. You prompt again.

Codex CLI is built on a different premise. It's an agent loop. You give it a goal, and it plans, executes, observes the results, adjusts, and continues until it's done or it's stuck. It has real tool use: it can read and write files, run arbitrary shell commands, execute tests, call external APIs.[2] It can spawn sub-agents to work in parallel on independent sub-tasks. It maintains working memory across steps. It knows when to ask for clarification and when to just get on with it.

That architecture matters because it changes what the tool is actually good for. Autocomplete is good at the line-by-line. Chat is good at the question-and-answer. Agentic execution is good at the *task*, the thing that has a beginning, a middle, and an end, that requires reading several files and running several commands and making several decisions along the way. Refactoring a module. Adding a feature end-to-end. Diagnosing a failing build. Those are the things that used to eat your afternoon. Those are the things Codex CLI is actually built for.

I think a lot of the breathless commentary about AI coding tools, and there is a lot of it, and most of it is overrated, misses this distinction. People benchmark autocomplete quality, or measure how often a model suggests the right function name. That's fine, but it's measuring the wrong thing. The interesting question isn't whether the model can write a sorting algorithm. It's whether the tool can sit alongside you in your actual workflow, your repo, your tests, your CI system, your conventions, and do useful work with real autonomy. Codex CLI's answer to that question is the most interesting I've seen.

[2]The official Codex CLI documentation describes the agent's tool use capabilities, including file read/write, shell command execution, and API calls. https://developers.openai.com/codex/cli

This book is my attempt to explain why, and to teach you how to use it properly.

Who This Book Is For

I wrote this for mid-to-senior software engineers who are already comfortable with AI-assisted development and are wondering whether agentic tools change the picture. If you've been using GitHub Copilot or Claude Code for a while, you understand what those tools are good at and where they fall short, and you're curious about what comes next: this is the book for you.

You should be comfortable in a terminal. Codex CLI is a command-line tool, and this book treats it as one. There's no GUI, no plugin, no sidebar. Everything happens in your shell. If that sounds appealing rather than alarming, you're in the right place.

You don't need deep expertise in machine learning or large language models. I'll explain what you need to know about how the underlying models work in order to use Codex CLI effectively, but this isn't an ML textbook. What I'm assuming is engineering experience: the ability to read a codebase, reason about system behaviour, debug under uncertainty. Those skills translate directly.

This book is not for managers who want to understand the AI landscape, or for non-technical readers who are curious about where software development is heading. There are good books for those audiences. This isn't one of them. I've written this for engineers, and I've tried to be concrete and specific in the way that engineers find useful.

One more thing: I'm writing from the perspective of someone who uses this daily in real work. I'm Programme Director at HCLTech AI Labs, which means I spend my time across enterprise delivery, research, and engineering teams. I've run Codex CLI on production codebases, on greenfield projects, in CI pipelines, and in local development. My opinions in this book are grounded in that use, not in benchmarks or demos.

What You Won't Find Here

I want to be honest about scope, because I think books that oversell their coverage do readers a disservice.

This is not a general introduction to prompt engineering. There are plenty of those, and some of them are excellent. This book assumes you can write a decent prompt and focuses instead on the specific patterns and idioms that work well with Codex CLI's agentic model.

This is not an LLM textbook. I won't be explaining attention mechanisms, transformer architectures, or how current frontier models are trained. Where the underlying model matters, and it does matter in specific ways, I'll explain what you need to know. But understanding neural networks is not a prerequisite for using Codex CLI effectively.

This is not a market survey. I'm not going to spend chapters comparing Codex CLI to Cursor, Devin, Aider, Claude Code, and every other tool in the current landscape. That comparison would be out of date before the book was printed. What I will do is explain what distinguishes the agentic approach from the alternatives, and let you draw your own conclusions about what that means for the tools you already use.

What you will find here is one tool, covered deeply, from someone who has used it daily for several months, with strong opinions about what works and what doesn't.

How This Book Is Organised

The book is divided into six parts.

Part 1: Orientation puts Codex CLI in context. What is it, where did it come from, and what does the agent loop actually look like in practice? If you're completely new to the tool, start here. If you've already been using it for a while, you might skim Part 1 and come back to specific chapters when they're relevant.

Part 2: Foundations covers the core mechanics in detail: configuration, context management, tool use, the trust model, and the sandbox environment. This is the part you'll return to most often as a reference. I've tried to write it so that individual chapters stand alone.

Part 3: Extension explains how to go beyond the defaults. Custom tools, MCP server integration, writing your own agent scripts, composing Codex CLI with other command-line tools. This is where things get interesting if you have specific workflows or constraints.

Part 4: Scale addresses the questions that come up when you move from personal use to team use. How do you share configuration? How do you run Codex CLI in CI? How do you audit what it's done? How do you manage cost? These are the questions I get most often from engineering teams.

Part 5: Specialised Workflows is the applied section, with specific, detailed walkthroughs of using Codex CLI for refactoring, debugging, writing tests, documentation, and code review. Each chapter in this part follows a real task from start to finish.

Part 6: Advanced steps back from individual workflows and examines the architecture: how the Agents SDK lets you treat Codex CLI as an orchestratable service, how its primitives compare to other agentic frameworks, what harness engineering is and why it matters for long-running agents, and how a small team can organise itself as an Agentic Engineering Pod. These chapters assume you have worked through Part 5 and are ready to think about the broader picture.

A Note on Version Currency

Codex CLI releases weekly. Sometimes more frequently. At time of writing, this book was developed against v0.117.x, and I've tried to note where I'm describing behaviour that was introduced recently or that I expect to change.[3]

The tool you install today may behave differently from the tool I was using when I wrote Chapter 7. That's not a reason to distrust the book; the architecture, the mental models, and the core workflows are stable, but it does mean you should treat specific flags and configuration options as starting points rather than gospel. The `codex --help` output is always more authoritative than anything printed here.

Two practical suggestions for staying current. First, read the changelog. Chapter 2 walks you through where to find it and how to read it efficiently; the Codex CLI changelog is unusually good, and the team is transparent about what's changed and why.[4] Second, check the companion repository at github.com/danielvaughan/codex-cli-book, where I maintain version-specific notes and update the code examples as the tool evolves. I also write

[3]The OpenAI Codex CLI GitHub repository hosts all releases with structured release notes. https://github.com/openai/codex/releases

[4]The Codex CLI changelog is published on the OpenAI developer portal alongside the GitHub releases page. https://developers.openai.com/codex/changelog

about new Codex CLI developments in my Substack newsletter, *Agentic Engineering*, which you can find linked from the companion repo.

How to Use This Book

If you're reading this for the first time, I'd suggest reading it cover to cover, at least through Part 3. The book is designed to build on itself, as concepts introduced early are used without re-explanation later, and the mental models in Parts 1 and 2 make everything in Parts 4 and 5 considerably easier to follow.

If you're returning to the book as a reference, each chapter in Parts 2, 3, and 4 is designed to stand alone. The index is detailed, and I've tried to cross-reference liberally. If you want to know how to configure context windows, go to Chapter 15. If you want to understand the trust model, go to Chapter 9. You shouldn't need to re-read surrounding chapters to get what you need.

If you're leading a team through this book, running internal workshops or trying to bring a group of engineers up to speed on agentic tooling, Part 5 is designed for that purpose. Each workflow chapter is structured so that it can be run as a hands-on exercise, and the companion repo includes the starter codebases you'll need.

* * *

Right. Let's get into it.

Getting Your Bearings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 1. What Is Codex CLI?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Before Codex: A Brief History

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Codex model (2021)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

GitHub Copilot and the autocomplete paradigm (June 2021)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The ChatGPT shift (November 2022)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Problem with Autocomplete

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The 2026 Agentic Landscape

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Category 1: IDE-Integrated Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Category 2: Terminal Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Category 3: Cloud and Hosted Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Where They All Stand

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Defining Codex CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Terminal-first

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Headless and scriptable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Sandboxed at the kernel layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

AGENTS.md-aware

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Open source under MIT

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Why the Terminal Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Composability with Unix tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CI/CD integration without ceremony

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

No GUI required

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Core Proposition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Reads the whole repository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Runs your tests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Makes commits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Follows AGENTS.md

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 2. Getting Started with Codex CLI

Chapter 1 made the case for what Codex CLI is and why the agentic model matters. This chapter answers the practical question: how do you get it running, how do you authenticate, and what does a real first session look like? The rest of the book assumes you've done this.

Learning Objectives

You will be able to:

- Install Codex CLI and verify the installation
- Complete the authentication flow appropriate for your situation
- Run a real first session and understand what you're seeing

* * *

The Bigger Picture: Four Surfaces, One Agent

Before you install anything, it helps to know what you're installing into. Codex CLI is not just a CLI tool, it is a single agent with four points of entry: the **CLI**, the **Desktop App**, the **IDE Extension**, and **Cloud**.[1] All four surfaces share the same intelligence layer, the same configuration (`~/.codex/config.toml`, `AGENTS.md`), and the same persistent memory. Skills you build and context you accumulate carry across every surface.

The CLI you are about to install is the foundational surface, open-source, local-first, and the one this book focuses on. But everything you learn here

[1]The four-surface architecture is described in the OpenAI developer documentation. https://developers.openai.com/codex/cli

applies when you later open the Desktop App to run parallel agent threads, use the IDE extension to work inline with your editor, or fire off a long-running task to Cloud. Think of the CLI as the primary instrument in a quartet: it does the job on its own, and it plays well with the others.

Chapter 3 covers what is new in detail, but keep this mental model in mind as you set things up. You are not installing a standalone tool; you are joining a system.

* * *

Prerequisites

Codex CLI requires Node.js 22 or later[2], a hard requirement. On macOS and Linux, manage Node versions with `nvm` or `fnm`:

```
fnm install 22 && fnm use 22
node --version   # v22.x.x or later
```

Codex CLI runs natively on macOS and Linux. Windows is supported via WSL2 only (WSL1 lacks the required kernel features for sandboxing).[3]

On macOS, both Apple Silicon and Intel are supported on macOS 13 Ventura or later.[4] On Linux, the sandbox layer requires `bwrap` (bubblewrap): `sudo apt install bubblewrap` if not pre-installed.

* * *

Installation

npm (recommended)

[2]Codex CLI requires Node.js 22.0.0 or later. https://nodejs.org/en/about/previous-releases

[3]macOS and Linux platform support is documented in the Codex CLI system requirements. https://developers.openai.com/codex/cli/system-requirements

[4]macOS 13 Ventura or later on Apple Silicon or Intel. https://developers.openai.com/codex/cli/system-requirements

```
npm install -g @openai/codex
codex --version
```

> **Tip:** If you see permission errors, configure npm to use a directory in your home folder: `npm config set prefix ~/.npm-global` and add `~/.npm-global/bin` to your `PATH`.

To upgrade an existing installation, Codex now provides a built-in update command (PR #19933, v0.126.0):

```
codex update                          # self-update via the package manager used
↪  to install
```

This replaces the previous workaround where typing `codex update` confusingly launched an interactive agent session with "update" as the prompt. The new subcommand detects your installation method (npm, Homebrew, or binary) and triggers the appropriate update mechanism. If you're on a version before v0.126.0, use the manual approach:

```
npm install -g @openai/codex@latest   # npm manual upgrade
```

Homebrew (macOS and Linux)

```
brew install --cask codex
```

The Homebrew cask installs a pre-built binary directly, so no Node.js dependency is required. Homebrew handles updates through the standard `brew upgrade` mechanism.

Direct binary download

Pre-built binaries are available from the GitHub releases page for environments where neither npm nor Homebrew is appropriate.[5] Download the archive for your platform, extract it, rename the binary to `codex`, and place it on your `PATH`:

[5] Pre-built binaries and release notes are published on the Codex GitHub releases page. https://github.com/openai/codex/releases

```
tar xzf codex-aarch64-apple-darwin.tar.gz
mv codex-aarch64-apple-darwin /usr/local/bin/codex
chmod +x /usr/local/bin/codex
```

Verifying the installation

Confirm everything is working:

```
codex --version
```

You should see output like `codex 0.119.0` (or your installed version). If the command is not found, ensure the installation directory is on your `PATH`.

As a more thorough check, run the built-in diagnostics:

```
codex doctor
```

This validates your Node.js version, sandbox capability, network connectivity to the OpenAI API, and authentication status in a single pass. Fix any warnings before continuing.

Shell completions

Tab-completion makes flag and subcommand discovery effortless. Codex CLI ships with built-in completion generation for bash, zsh, and fish (plus PowerShell and Elvish).[6]

Bash:

```
mkdir -p ~/.local/share/bash-completion/completions
codex completion bash > ~/.local/share/bash-completion/completions/codex
```

Zsh:

[6]The `codex completion` subcommand supports bash, zsh, fish, powershell, and elvish. https://developers.openai.com/codex/cli/reference

```
mkdir -p ~/.zsh/completions
codex completion zsh > ~/.zsh/completions/_codex
```

Then add to `~/.zshrc` before `compinit`:

```
fpath=(~/.zsh/completions $fpath)
autoload -Uz compinit
compinit
```

Fish:

```
codex completion fish > ~/.config/fish/completions/codex.fish
```

Fish picks up completion files from this directory automatically.

* * *

Authentication

Path 1: ChatGPT subscription (device-code flow)

```
codex auth login
```

Codex CLI prints a short code and opens a URL. Log in, enter the code, and the CLI stores a credential token in `~/.codex/auth.json` (owner read/write only).[7]

Path 2: API key

```
export OPENAI_API_KEY="sk-..."
```

Add to your shell profile for persistence. With an API key, you're billed per token. Start with a low usage cap while learning typical session costs.

Path 3: Enterprise configuration

[7] ChatGPT Plus and Pro subscription limits for Codex usage. https://openai.com/pricing

```
codex config set api_base_url "https://your-org.openai.azure.com/..."
codex config set api_key_source "azure_ad"
```

Chapter 22 covers enterprise deployment in detail.

* * *

Your First Session

Clone the example repository and launch Codex CLI:

```
git clone https://github.com/openai/codex-quickstart-example.git
cd codex-quickstart-example
codex
```

Type your task:

```
Find the bug in main.js and fix it. Run the tests before and after to confirm
↪  the fix.
```

Codex reads files, runs `npm test` (which fails), traces the bug, writes the fix, runs tests again (which pass), and summarises the change. The whole process takes twenty to forty seconds. Every tool call is visible in the session log.

This is the agent loop: plan, execute, observe, adjust. The result is a diff applied to the filesystem with a test run confirming it works.

* * *

Interactive REPL vs. `codex exec`

The **interactive REPL** (`codex` with no arguments) is a persistent session with conversation history. Use it for exploration, multi-step tasks, and learning.

`codex exec` runs non-interactively and exits on completion. Use it for scripted tasks, CI/CD pipelines, and well-defined automation:

```
codex exec "update all TypeScript interfaces in src/types/ to add the new
↪  userId field"
codex exec --json "analyse test coverage and report modules under 80 per cent"
```

Heuristic: if you'd type it into a terminal, use the REPL. If you'd write it in a Makefile, use `codex exec`.

* * *

Where Configuration Lives

`~/.codex/config.toml`: user-level defaults (model, approval mode, sandbox settings):

```
model           = "gpt-5.4"
approval_policy = "on-request"
sandbox_mode    = "workspace-write"
```

`AGENTS.md`: project-level instructions read at session start. Chapter 8 is devoted to it.

Environment variables override `config.toml`:

Variable	Purpose
`OPENAI_API_KEY`	API key for direct access
`CODEX_MODEL`	Override default model
`CODEX_SANDBOX`	Override sandbox mode
`CODEX_LOG_LEVEL`	Set log verbosity

Resolution order (later wins): built-in defaults -> `~/.codex/config.toml` -> project `.codex/config.toml` -> environment variables -> command-line flags.

* * *

Summary

- The CLI is one of four Codex surfaces (CLI, Desktop App, IDE Extension, Cloud) that share the same intelligence layer, configuration, and memory.
- Codex CLI requires Node.js 22+ and runs on macOS and Linux natively; Windows via WSL2.
- Install via `npm install -g @openai/codex`, `brew install --cask codex`, or direct binary download. Run `codex doctor` to verify.
- Set up shell completions for bash, zsh, or fish to speed up daily use.
- Three auth paths: device-code flow for subscriptions, `OPENAI_API_KEY` for API access, enterprise config for managed deployments.
- The REPL is for exploratory work; `codex exec` is for scripted and CI/CD usage.
- Configuration lives at three levels with a clear resolution order. Skills and context carry across all four surfaces.

Exercises

Exercise 2-1. Install Codex CLI, run `codex --help`, and identify three flags you didn't know existed. Write one sentence for each explaining when you'd use it.

Exercise 2-2. Create a minimal `AGENTS.md` in a current project with the test command, one coding convention, and one off-limits directory. Run a session and observe whether the instructions take effect.

Exercise 2-3. Run `codex exec` on a bounded task ("find all TODO comments and write a TODOS.md listing them"). Then run the same task interactively. Compare what the interactive session showed you that `codex exec` did not.

Chapter 3. What's New in Codex CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Three Themes of 2026

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Model Landscape

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Model deprecation timeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Subagents: Generally Available

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Hooks System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

New CLI Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex App Computer Use

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Thread Automations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enterprise Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

v0.122.0 Stable — April 20, 2026

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

`/side` conversations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Plan Mode with fresh context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Deny-read glob policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Slash command queueing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Reproducible CI with `--ignore-user-config` and `--ignore-rules`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Windows and Intel Mac desktop app

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Tool discovery and image generation enabled by default

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

v0.123.0 Stable — April 23, 2026

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Amazon Bedrock native provider (PR #18744)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

`/mcp verbose` diagnostics command

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Plugin MCP loading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Built-in Code Review skill (PR #18746)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Remote thread store (PR #18714)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Agent identity auth becoming first-class (PR #18785)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Expected: Goal Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Context fragments as plugin injection points (PR #18794)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

v0.124.0 Stable — April 23, 2026

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Hooks marked STABLE

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Quick reasoning controls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Hooks observe MCP tools and apply_patch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Inline hook configuration in config.toml

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Fast service tier default

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Remote plugin marketplace enhancements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

rust-v0.125.0-alpha.1

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

v0.125.0 Stable and the v0.126.0 Alpha Cycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Staying Current

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The version trail

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Reading the changelog efficiently

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 4. Benchmarks and Real-World Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Benchmark Landscape

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Benchmark progression: five years of coding model evolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

SWE-bench: Gold Standard to Cautionary Tale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What SWE-bench actually measures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

SWE-bench Verified and the contamination problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

SWE-bench Pro: the honest measure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Terminal-Bench 2.0: CLI-First Benchmarking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Structure and motivation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Current scores

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Running Terminal-Bench yourself

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Scaffolding Effect

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What the Numbers Actually Mean for Your Team

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Match the benchmark to your workload

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Account for scaffold inflation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Model selection within Codex CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Running Your Own Benchmarks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

April 2026 Benchmarks: CocoaBench, HiL-Bench, and AAR

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The failure taxonomy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

AAR: 37.2% accuracy across 9,374 trajectories

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CocoaBench: multi-modal composition at 55% failure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

HiL-Bench: the help-seeking gap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What the new benchmarks mean for your configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

SlopCodeBench: Measuring Quality, Not Just Completion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Benchmark Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Code Review Benchmarks and Adoption Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 5. Competing Tools and When to Use Each

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Two-Tier Landscape

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

A note on the “should you even use this book?” question

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Terminal Tier: The CLI Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Claude Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Gemini CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Aider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cross-Agent Interoperability: The New Competitive Dimension

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Aider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cline CLI 2.0

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

OpenCode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Goose by Block

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

IDE Tier: The Editor Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

GitHub Copilot

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cursor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Windsurf

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

GitKraken Desktop 12.0

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Kiro (Amazon)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Ecosystem Convergence: The Reverse Bridge Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Decision Framework

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Multi-Tool Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Ecosystem at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Protocol Landscape: MCP and A2A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The superapp convergence (April 2026)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The SaaS Readiness Dimension

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

When to stay with what you have

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The DNA of Coding Agents: System Prompt Comparison

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Prompt architecture differences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The proactiveness spectrum

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 6. Codex in the Wild: Interfaces and Community

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Usage Limits: The Hidden Variable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What the Benchmarks Actually Tell You

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Core Personality Difference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Practical Handoff Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Power Stack: Using Both

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The token burn crisis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Team Adoption Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Interfaces: Desktop, CLI, and IDE

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Terminal CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Desktop App

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The IDE Extension

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Interface Decision Framework

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Voice: Realtime V2 and Background Agent Streaming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Configure Once, Use Everywhere

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Foundations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 7. Prompting Codex CLI Effectively

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Why Codex CLI Prompting Is Different

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The cost of a vague run

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Anatomy of an Effective Prompt

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Task Scoping and Reasoning Effort

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Plan mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

PLANS.md for multi-hour tasks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The checkpoint pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Iterative Prompting and Mid-Session Corrections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Prompt Patterns for Common Tasks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Bug fixes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Refactoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Parallelisable work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Moving Durable Context Out of Prompts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

AGENTS.md

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Custom system prompts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 8. AGENTS.md: Patterns and Pitfalls

In Chapter 7 you learned to craft prompts that guide Codex CLI through a single task: frame the goal clearly, supply the right context, constrain the scope. That skill matters. But prompts are ephemeral. Every new session starts blank, and if your team has discovered that Codex CLI needs to run `make test` before committing, or must never touch generated files under `src/generated/`, someone has to remember to say so—every time. Multiply that across a team of ten engineers running dozens of Codex CLI sessions a day, and the gap between what Codex CLI does by default and what your project actually needs becomes a daily source of friction.

`AGENTS.md` closes that gap. It's a Markdown file you commit to your repository, or store in your home directory for personal defaults, and Codex CLI reads it automatically at the start of every session. Think of it as the written constitution for your AI collaborator: the standing rules, project conventions, and hard limits that apply regardless of what task you've asked for. Where Chapter 7 was about the art of the single prompt, this chapter is about the infrastructure that makes every prompt better without you having to repeat yourself.

Learning Objectives

You will be able to:

- Write a complete `AGENTS.md` from scratch using the starter template in this chapter
- Structure a monorepo with layered `AGENTS.md` files so each service or package gets precisely the instructions it needs
- Diagnose why a rule isn't taking effect and trace the active instruction chain
- Prune a bloated `AGENTS.md` without breaking the behaviours that depend on it

- Treat instruction files with the same version-control discipline as production code

* * *

What AGENTS.md Is and How Codex CLI Reads It

`AGENTS.md` is a plain Markdown file. Codex CLI imposes no schema on it; any Markdown is valid. What makes it special is when and how Codex CLI reads it: before your prompt, automatically, on every run.[1]

The discovery chain

Codex builds its instruction set by walking the filesystem from the broadest scope to the narrowest, concatenating every `AGENTS.md` it finds along the way. The full resolution order is:

```
1. ~/.codex/AGENTS.override.md    ← highest-precedence global override
   OR ~/.codex/AGENTS.md          ← standard global defaults

2. <git-root>/AGENTS.override.md  ← repo override (beats repo AGENTS.md)
   OR <git-root>/AGENTS.md        ← standard repo rules

3. <intermediate-dirs>/AGENTS.override.md  ← per each directory between root
↪   and cwd
   OR <intermediate-dirs>/AGENTS.md

4. <cwd>/AGENTS.override.md       ← the directory you invoked Codex CLI from
   OR <cwd>/AGENTS.md
```

The concatenation order means that content from a closer file wins when it conflicts with content from a farther file. Your home-directory `AGENTS.md` sets personal defaults; the repo-root `AGENTS.md` sets team defaults; a subdirectory `AGENTS.md` sets service-specific rules. Each layer can add to, tighten, or override anything above it.

A few operational details worth knowing from the start:

[1]The AGENTS.md file is injected into the model context before the user prompt on every Codex CLI run. The full specification of the file format and discovery mechanism is documented at the OpenAI developer site. https://developers.openai.com/codex/guides/agents-md

- Codex CLI rebuilds this chain fresh on every run; there's no cache to clear when you edit a file.
- Empty files are silently skipped; they don't reset or interrupt the chain.
- The combined size of all loaded files is capped by the `project_doc_max_bytes` setting, which defaults to approximately 5 MB (raised from the original 32 KiB ceiling). Files that push past this limit are silently truncated—a sharp edge covered in detail in the section "AGENTS.md for Monorepos and Multi-Service Repos" below, and again in Chapter 10.[2]
- The `.override.md` variant at any level takes precedence over the plain `.md` variant at the same level.

Codex CLI resolves the git root, not your current working directory, as the starting point for step 2. If you invoke Codex CLI from inside `services/payments/`, the discovery chain starts at the repository root, then traverses down through intermediate directories to your current location. This means repo-root rules always apply, even deep in a subdirectory.

What the file looks like

There's no required structure. In practice, the files that work best are divided into a handful of focused sections, written in plain imperative prose that Codex CLI can follow without ambiguity. Here's a minimal example for a TypeScript web service:

[2]The `PROJECT_DOC_MAX_BYTES` constant in `codex-rs/core/src/config/mod.rs` is now set to `5_000_000` (approximately 5 MB), raised from the original `32 * 1024`. Exceeding this limit still causes silent truncation with no warning displayed in the TUI. https://github.com/openai/codex/issues/7138

```
# Project: Checkout Service

## Overview
Node.js REST API for the checkout flow. Written in TypeScript, tested with Vitest,
deployed via GitHub Actions to AWS Lambda.

## Commands
- Build: `npm run build`
- Test: `npm test` (runs Vitest; must pass before any commit)
- Lint: `npm run lint`
- Type-check: `npm run typecheck`

## Commit conventions
- Use Conventional Commits: `feat:`, `fix:`, `chore:`, etc.
- Scope to the affected module: `feat(cart): add coupon validation`
- Never commit directly to `main`—always through a PR

## Style
- TypeScript strict mode is enabled; never use `any` or `// @ts-ignore`
- Imports are sorted alphabetically within each group (ESLint enforces this)
- All async functions return `Promise<T>` explicitly

## Constraints
- Do not modify files under `src/generated/`—they are auto-generated from OpenAPI
- Do not install new dependencies without listing them in the PR description
- All new endpoints must include an integration test in `tests/integration/`
```

This file is 22 lines of Markdown. Codex CLI reads it before your first message, so you never need to restate that the test command is `npm test` or that `src/generated/` is off-limits.

Keep the "Commands" section at the top of the file. Codex CLI references it repeatedly when deciding how to verify its own work. The faster it finds the test command, the less likely it is to invent one.

Thinking in tiers: constitution, specialist, skill

A flat `AGENTS.md` works well for small-to-medium projects. Once a codebase crosses roughly 20,000 lines, however, you may notice the single file growing unwieldy—too many rules for too many domains, all loaded into every session. Aristidis Vasilopoulos's empirical study of a 108,000-line C# distributed system

offers a useful mental model: a three-tier knowledge architecture that maps directly onto Codex CLI's existing primitives.[3]

Tier	Role	Codex CLI Mapping	Loaded when
T1 — Constitution	Non-negotiable rules, routing table	`AGENTS.md`	Every session
T2 — Specialist	Domain-specific expertise	Custom agents in `.codex/agents/*.toml`	Per task type
T3 — Knowledge base	Reference specs, API docs	MCP servers	On-demand query

The constitution (Tier 1) is the `AGENTS.md` you already know. It stays lean—under 500 lines—and contains only what every session needs: coding standards, architectural boundaries, forbidden patterns, and crucially, routing rules that tell the agent when to delegate to a specialist. Tier 2 agents hold the deep domain knowledge that would bloat the constitution if included inline. Tier 3 pushes rarely needed reference material behind MCP servers, keeping the context window lean.

You don't need all three tiers on day one. Most projects live happily with Tier 1 alone. The model is useful because it gives you a principled answer to the question "where should this piece of context live?" If an instruction applies to every session, it belongs in `AGENTS.md`. If it applies only when working on a specific subsystem, it belongs in a specialist agent or a subdirectory `AGENTS.md`. If it is reference material queried occasionally, it belongs behind an MCP server.

Routing rules and architectural boundaries. When you do adopt specialists, the constitution becomes the routing table. The pattern is straightforward: encode trigger conditions directly in `AGENTS.md` so the agent knows when to delegate:

[3]Vasilopoulos, A. (2026). "Codified Context: Infrastructure for AI Agents in a Complex Codebase." arXiv:2602.20478v1. The paper tracks a 108,000-line C# distributed system across 283 development sessions, formalising the three-tier knowledge architecture (constitution, specialist agents, MCP knowledge servers). https://arxiv.org/abs/2602.20478

```
## Routing Rules

When the task involves **network protocols or sync logic**, delegate to
the `network-protocol-designer` agent before making changes.

When the task involves **coordinates, camera, or spatial transforms**,
delegate to the `coordinate-wizard` agent.

## Architectural Boundaries

- ECS components MUST NOT hold references to MonoBehaviours
- Network messages MUST be defined in the shared assembly
- All coordinate transforms go through CoordinateService
```

Architectural boundaries belong in the constitution because they are non-negotiable invariants—the kind of rule that, if violated, breaks the system regardless of which subsystem is being changed. Vasilopoulos's study found that over 57 per cent of agent invocations were routed to project-specific specialists rather than generic tool agents, validating the investment in domain-specific context.[4] The maintenance overhead was modest: roughly one to two hours per week for the entire three-tier infrastructure.

* * *

Essential Sections: Commits, Testing, Style

Most effective `AGENTS.md` files converge on the same set of sections. This isn't coincidence—it reflects the categories of information that are genuinely hard for a language model to infer from source code alone.

Commands

The single most important section. List every command Codex CLI might need to run to verify its work:

[4] Vasilopoulos, A. (2026). "Codified Context: Infrastructure for AI Agents in a Complex Codebase." arXiv:2602.20478v1. The paper tracks a 108,000-line C# distributed system across 283 development sessions, formalising the three-tier knowledge architecture (constitution, specialist agents, MCP knowledge servers). https://arxiv.org/abs/2602.20478

```
## Commands
- Install dependencies: `npm ci`
- Build: `npm run build`
- Unit tests: `npm test`
- Integration tests: `npm run test:integration` (requires Docker; skip in CI if
↪  not available)
- Lint: `npm run lint -- --fix`
- Type-check: `npm run typecheck`
- Local dev server: `npm run dev` (port 3000)
```

Be explicit about prerequisites. "Requires Docker" in the integration tests line above tells Codex CLI not to blindly run that command in an environment where Docker isn't available. That kind of conditional note pays off when Codex CLI operates in automated pipelines.

Commit conventions

If your project follows a commit message format, such as Conventional Commits,[5] a custom format, or a required ticket prefix, state it here. Codex CLI will use it when generating commit messages. Without it, you'll get generic messages like "fix bug" or "update files":

```
## Commit conventions
- Format: `<type>(<scope>): <description>`
- Types: feat, fix, docs, style, refactor, test, chore
- Scope is required; use the module or package name
- Description is imperative mood, lowercase, no trailing period
- Breaking changes: add `BREAKING CHANGE:` footer in the commit body
- Reference issues: `Closes #123` in the commit body
```

Style and conventions

Code style rules that aren't captured by your linter belong here. The linter enforces formatting; `AGENTS.md` handles intent:

[5] Conventional Commits is a specification for adding human- and machine-readable meaning to commit messages, using a structured `<type>(<scope>): <description>` format. https://www.conventionalcommits.org/en/v1.0.0/

```
## Code style
- Prefer `const` over `let`; never use `var`
- Error handling: always use the project's `AppError` class, not raw `Error`
- Logging: use `logger.info/warn/error` from `src/lib/logger.ts`; never
  ↪ `console.log` in production paths
- Database queries: use the `db` singleton from `src/lib/db.ts`; never create a
  ↪ new connection
```

Constraints and off-limits areas

This is where you prevent Codex CLI from making well-intentioned but destructive changes:

```
## Constraints
- Do NOT modify: `src/generated/`, `schema/migrations/`, `package-lock.json`
- Do NOT install packages without approval
- Do NOT change environment variable names—they are referenced in deployment
  ↪ configs
- All changes to `src/auth/` require a comment explaining the security
  ↪ implications
```

The "Do NOT" framing is deliberate. Affirmative instructions ("only modify X") require Codex CLI to reason about what isn't X. Negative instructions are more reliable for hard limits.

`AGENTS.md` is not a security boundary. Codex CLI can be instructed to ignore it in a prompt. For files you genuinely must protect, use filesystem permissions or repository branch protection rules. `AGENTS.md` is guidance, not enforcement.

* * *

Project-Specific Context: What to Include and Omit

The hardest editorial decision in writing `AGENTS.md` is what to leave out. The default `project_doc_max_bytes` ceiling was raised from 32 KiB to ~5 MB, but a generous limit doesn't make bloat free—every line you add still consumes tokens and adds cognitive load to every session. The goal isn't completeness—it's signal density.

What to include

Repo layout. A brief map of where things live helps Codex CLI navigate without reading every directory:

```
## Repo layout
- `src/api/` — Express route handlers and middleware
- `src/services/` — Business logic; no direct DB access here
- `src/models/` — Sequelize model definitions and migrations
- `src/lib/` — Shared utilities (logger, db, config)
- `tests/` — Unit tests mirror the `src/` structure
- `scripts/` — One-off maintenance scripts; not production code
```

Non-obvious dependencies. Anything that would surprise an engineer reading the repo for the first time:

```
## Architecture notes
- The API is stateless; all session state lives in Redis (see
  ↪ `src/lib/session.ts`)
- Feature flags are read from environment variables at startup—no hot reload
- Rate limiting is handled at the gateway layer; don't add it in application
  ↪ code
```

Recurring gotchas. If your team hits the same misunderstanding repeatedly, document it once here:

```
## Common mistakes to avoid
- Do not use Sequelize's `sync()` method—migrations are managed by Umzug
- The `User` model's `id` field is a UUID, not an integer—don't compare with
  ↪ `==`
- `process.env` is only safe to access inside functions, not at module load time
```

What to omit

Information already in code. If the answer is obvious from reading the source, don't duplicate it in `AGENTS.md`. Codex CLI reads your source code too.

Aspirational rules. Don't put rules in `AGENTS.md` that you don't actually enforce. If test coverage isn't measured and no one will check, saying "maintain 80 per cent coverage" trains Codex CLI to write tests for coverage, not correctness.

Personal preferences that aren't team policy. The home-directory `~/.codex/AGENTS.md` is the right place for your personal style. The repo-root `AGENTS.md` should reflect team consensus.

Process documentation. Deployment runbooks, on-call procedures, and architecture decision records belong in your docs system, not in `AGENTS.md`. Link to them if Codex CLI needs them; don't paste them inline.

* * *

Common Mistakes and How They Manifest

The rules don't apply

You wrote a constraint, but Codex CLI ignores it. Before assuming a bug, check the instruction chain. Run this to see exactly what Codex CLI loaded:

```
codex --ask-for-approval never \
  "List the AGENTS.md files you loaded this session, in order, with their full
  ↪  paths."
```

You can also inspect the session log directly:

```
grep -i agents ~/.codex/log/codex-tui.log
```

The log shows exactly which files were found and concatenated. Table 9-1 lists the most common failure modes.

Table 9-1. Common AGENTS.md diagnostic scenarios

Symptom	Likely cause
Override not applying	Wrong filename—use `AGENTS.override.md`, not `AGENTS-override.md`

Symptom	Likely cause
Old rules persisting after edit	A closer `AGENTS.override.md` is still winning over your updated `AGENTS.md`
File silently ignored	File is empty, or the combined chain already hit `project_doc_max_bytes`
Rules applying in the wrong directory	Codex CLI resolves from the git root; run from the target directory
Constraint respected inconsistently	The rule is buried below the truncation point imposed by `project_doc_max_bytes`

The file is too prescriptive or contradicts itself

`AGENTS.md` that micromanages every decision produces brittle sessions. If you're writing rules about how to name local variables, you've gone too far. Reserve `AGENTS.md` for things that can't be inferred from reading the code.

When levels of the instruction chain conflict—the repo-root says "use `npm test`" and a subdirectory says "use `make test`"—Codex uses the more specific (closer) rule. The problem is unintentional contradictions, usually from updating the root file without checking subdirectory overrides. The diagnostic command above catches these.

Treat your `AGENTS.md` files like code. Review them in pull requests, assign ownership in subdirectories, and add them to your onboarding checklist.

* * *

AGENTS.md for Monorepos and Multi-Service Repos

Monorepos are where the hierarchical scope chain earns its complexity budget. A single repo might contain a React frontend, a Node API, a Python data

pipeline, and Terraform infrastructure—each with different test commands, style rules, and off-limits files. A single flat `AGENTS.md` at the root either becomes unmanageably large or forces every service to share the lowest common denominator.

The solution is to distribute instructions to the directories that need them.

Recommended structure

```
my-monorepo/
├── AGENTS.md                        ← 3-4 KiB: repo layout, global CI commands, shared conventions
├── services/
│   ├── api/
│   │   └── AGENTS.md                ← 2 KiB: Node/TypeScript rules, API-specific patterns
│   ├── payments/
│   │   └── AGENTS.override.md       ← 2 KiB: PCI rules that override any parent leniency
│   └── frontend/
│       └── AGENTS.md                ← 2 KiB: React conventions, Vitest setup, CSS modules
└── infra/
    └── AGENTS.md                    ← 2 KiB: Terraform patterns, never-touch state files
```

When Codex CLI works in `services/api/`, it loads the repo root plus `services/api/AGENTS.md`, roughly 5–6 KiB of focused, relevant instructions. It doesn't see the Terraform rules or the payments PCI constraints. That's intentional.

Using override files in monorepos

The `AGENTS.override.md` variant is especially useful for services that need to tighten rules inherited from the root. In the structure above, `services/payments/AGENTS.override.md` might contain:

```
## Payments Service Rules

**Test command:** `make test-payments` (NOT the root `npm test`)
**Never modify:** `src/generated/`, `schema/migrations/`
**Security rule:** All fields that handle PII must be annotated with `@PII`.
  Flag any additions that lack this annotation.
**Before committing:** Run `make audit-pci`. It must exit 0.
```

The `override` variant signals intent: these rules are deliberately replacing, not merely extending, whatever the parent said. Using a plain `AGENTS.md` in the same location would extend the parent; using `AGENTS.override.md` makes the replacement explicit and visible.

Managing the project_doc_max_bytes limit

The `project_doc_max_bytes` setting caps the combined size of all files in the resolved chain. The default was raised from 32 KiB to approximately 5 MB (`5_000_000` bytes in `PROJECT_DOC_MAX_BYTES`), so most projects will never hit the ceiling. Nonetheless, silent truncation—with no warning in the terminal or logs—remains a real failure mode reported by users in production (see the GitHub issue tracker), and context bloat still wastes tokens even when files fit.[6] The layered structure above is the primary defense: by distributing instructions, each invocation loads only the relevant subset, keeping token costs low.

If you genuinely need more space than the ~5 MB default, you can raise the cap in `~/.codex/config.toml`:

```
[agent]
project_doc_max_bytes = 10000000   # ~10 MB
```

Raising the limit increases token usage per session. Before doing so, audit your existing `AGENTS.md` files for content that can be pruned, moved to source code comments, or linked rather than inlined. See "What Not to Do: Common AGENTS.md Mistakes" in this chapter for pruning strategies.

Silent truncation means rules that appear near the bottom of a large `AGENTS.md` may never be read. Put your most important constraints—especially off-limits files and required test commands—at the top of the file, not the bottom.

[6] A GitHub issue documents the silent truncation behaviour and community requests for a visible warning in the `/stats` TUI display. https://github.com/openai/codex/issues/7138

Override files for CI/CD pipelines

In automated pipelines you often want Codex CLI to behave differently than it does interactively: no clarifying questions, stricter verification requirements, no commits. Rather than modifying your shared `AGENTS.md`, generate a temporary override file for the duration of the pipeline run:

```
# In your GitHub Actions workflow step:
cat > .codex/AGENTS.override.md << 'EOF'
## CI Mode
- Never ask clarifying questions—make your best judgment and proceed
- Always run the full test suite before completing any task
- If tests fail, explain the failure and stop; do not attempt a fix
- Do not make commits; only propose changes as diffs
EOF

codex exec "$TASK" --full-auto

rm .codex/AGENTS.override.md
```

The override file is created before the run and deleted after. Your shared `AGENTS.md` is never touched, and the CI behaviour is fully auditable from the workflow definition.

Fallback filenames

If your team already maintains a well-structured `CONTRIBUTING.md` or `TEAM_GUIDE.md`, you can tell Codex CLI to treat it as an `AGENTS.md` equivalent—without migrating its contents:[7]

```
# ~/.codex/config.toml
[agent]
project_doc_fallback_filenames = ["CONTRIBUTING.md", "TEAM_GUIDE.md",
↪  ".agents.md"]
```

Codex CLI checks for these filenames at each directory level, using the same priority order as `AGENTS.md`. This is particularly useful when adopting Codex CLI on an existing team that has already invested in developer documentation.

[7]The `project_doc_fallback_filenames` configuration key lets Codex CLI treat arbitrary filenames (e.g., `CONTRIBUTING.md`) as AGENTS.md equivalents, following the same discovery priority order. https://developers.openai.com/codex/guides/agents-md

You get the benefit immediately whilst migrating content to a dedicated `AGENTS.md` over time.

`AGENTS.md` is not exclusive to Codex. In December 2025 OpenAI donated the specification to the Agentic AI Foundation (AAIF) under the Linux Foundation—the same governance structure behind Kubernetes and Node.js—alongside Anthropic's Model Context Protocol and Block's goose.[8] By early 2026, AAIF had grown to 146 member organisations including JPMorgan Chase, Red Hat, Autodesk, and UiPath.[9] Over 30 agent products now read `AGENTS.md` natively, including Codex CLI, GitHub Copilot, Cursor, Gemini CLI, Jules, Windsurf, Amp, Devin, Aider, JetBrains Junie, Zed, Warp, and Factory.[10] More than 60,000 open-source repositories have adopted the file. One well-maintained `AGENTS.md` per repository guides all AI tooling your team uses—no need to maintain parallel `CLAUDE.md`, `.cursorrules`, and `.windsurfrules` files for different tools. The practical recommendation from the community is the 80/20 rule: write 80 per cent of your instructions in `AGENTS.md`, then maintain tool-specific files only for features that genuinely require them.[11]

* * *

Testing and Validating Your AGENTS.md

Stale `AGENTS.md` content is worse than no content: it actively misleads Codex CLI. Before merging any change, run these three validation commands:

[8] AGENTS.md was donated to the Agentic AI Foundation (AAIF) under the Linux Foundation in December 2025, alongside Anthropic's Model Context Protocol and Block's goose. https://openai.com/index/agentic-ai-foundation/

[9] Linux Foundation. "Agentic AI Foundation Welcomes 97 New Members." 24 February 2026. Total membership reached 146 organisations; David Nalley (AWS) appointed governing board chair. https://www.linuxfoundation.org/press/agentic-ai-foundation-welcomes-97-new-members

[10] AGENTS.md official specification site, accessed April 2026. Lists 25+ supported tools and 60,000+ project adoption. https://agents.md/

[11] SmartScope. "AGENTS.md Cross-Tool Unified Management Guide." February 2026. Recommends the 80/20 shared-base approach. https://smartscope.blog/en/generative-ai/github-copilot/github-copilot-agents-md-guide/

```
# Does Codex CLI know how to run the tests?
codex --ask-for-approval never "How do I run the tests for this project?"

# Does Codex CLI know what it must not touch?
codex --ask-for-approval never "Which files or directories in this repo should
↪  never be modified?"

# Is the instruction chain correct?
codex --ask-for-approval never \
  "List every AGENTS.md file you loaded this session, in order, with full paths
  ↪  and approximate byte sizes."
```

If the answers to the first two are wrong, your `AGENTS.md` isn't working. If the third reveals unexpected or missing files, you have a chain configuration problem.

To catch truncation, ask Codex CLI for the last rule it can see from the repo-root file. If it names a rule from the middle of your file, you've hit the size limit. Move critical rules up.

Add a step to your pull request template asking authors to verify whether their changes affect any `AGENTS.md` file. A migration that renames the test command is a breaking change for Codex, just as it is for CI.

* * *

* * *

What Not to Do: Common AGENTS.md Mistakes

A well-maintained `AGENTS.md` is a force multiplier. A bloated one actively works against you—costing token budget, burying critical rules, and causing the agent to follow instructions that produce no useful change. Understanding how files go wrong helps you write better ones from the start and prune existing ones without fear.

How files grow without bound

AGENTS.md files don't start large. They start as three or four lines. The problem is the growth curve. Every time the agent does something wrong, the instinct is to add an instruction. The agent used `npm` when the project uses `bun`—add a line. Each addition is individually rational. Collectively they create a different problem.

A February 2026 paper from ETH Zurich's SRI Lab—the most rigorous empirical evaluation of repository-level context files conducted to date—found that LLM-generated context files reduce task success rates by an average of 3 per cent compared to having no context file at all, while increasing inference costs by over 20 per cent. Human-written files improved success by only 4 per cent with a cost increase of up to 19 per cent.[12] The content that drives positive outcomes is specifically non-standard tooling instructions and non-inferable team decisions—not architectural overviews or framework conventions.

Never commit the output of `/init` or any other auto-generation command without a thorough line-by-line human review. Generated files are accurate-sounding and actively harmful to agent performance until you've pruned the redundant content.

What the ETH Zurich paper actually tells you

The paper's headline numbers have already prompted some practitioners to conclude "just delete your AGENTS.md." That misreads the results. Here is what the evidence actually supports, and where it stops.

The benchmarks used well-documented open-source repositories. These projects already have comprehensive READMEs, contributing guides, inline comments, and years of Stack Overflow coverage baked into the model's training data. Corporate monorepos have none of that ambient documentation. The less publicly documented your codebase is, the *more* a concise context file matters—not less.

The negative signal came from LLM-generated files, not from context files as a category. Auto-generated files are padded with inferred-from-code

[12] Gloaguen et al., "Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents?", ETH Zurich SRI Lab, arXiv:2602.11988 (February 2026). The paper introduces AGENTbench, a benchmark of 138 real-world Python tasks. https://arxiv.org/abs/2602.11988

content the model already knows, which adds cost and dilutes the signal. The takeaway is not "context files are useless"—it is "never auto-generate and commit without rigorous human review."

The truncation limit is vindicated, not challenged. The finding that shorter, human-curated files outperform longer generated ones aligns with the hard cap Codex CLI enforces (now ~5 MB by default, up from the original 32 KiB). The higher ceiling doesn't change the lesson: brevity wins. One real code snippet showing your non-obvious build invocation beats three paragraphs describing your architecture.

The content that moved the needle was narrow and specific: non-standard tooling instructions, non-inferable team decisions, and custom build or test commands. Architectural overviews and framework conventions—the bulk of most auto-generated files—contributed nothing. If your AGENTS.md is mostly architecture prose, the paper explains why it is not helping.

The correct conclusion is not "delete your AGENTS.md." It is "prune ruthlessly, keep only what the model cannot infer from your code and its training data, and never commit auto-generated content without review."

The compounding costs of oversized instructions

There are four concrete categories of harm from a bloated file:

The obedience trap. Every line in your `AGENTS.md` is a commitment to additional agent steps. An instruction to "check the architectural overview before making any changes" will generate a file read on every task, including trivial one-line fixes.

Silent truncation. Codex CLI enforces a hard combined limit via `project_doc_max_bytes` (now ~5 MB by default, up from the original 32 KiB). There is no warning. The TUI does not indicate truncation has occurred. Instructions at the end of a long file—in a reactively grown file, often the most recently added and most urgently needed—are the first to disappear.

The lost-in-the-middle effect. Even below the truncation threshold, attention is not uniform. Information in the middle of a long context receives less reliable attention than content near the top. A short file with critical rules near the top outperforms a longer file with the same rules buried at line 140.

Inference cost. Every token in your `AGENTS.md` is processed on every agent invocation. A file that is twice as long costs roughly twice as much to include in

context, across every session—including the ones where none of the additional content is relevant.

A pruning checklist

Work through your `AGENTS.md` in one sitting, applying these checks to each instruction:

Is it about repository structure, architecture, or layout? Remove it. Agents navigate repositories by reading them. Navigation instructions add cost and rarely change outcomes.

Is it a generic best practice? ("Write clean, readable code." "Prefer clarity over cleverness.") Remove it. These are baked into the model.

Does it describe a framework convention? If your `AGENTS.md` says "use React hooks rather than class components," that's part of the React ecosystem's current practice. The model knows it.

Does it overlap with content in your README or another context file? Remove it. The agent reads those files. You're paying twice for one instruction.

Could it move to a nested `AGENTS.md`? If an instruction is only relevant to a specific subdirectory, move it down to an `AGENTS.md` in that subdirectory.

Could it move to a Skill? If an instruction is only relevant when the agent is performing a specific workflow—deploying, running load tests—it belongs in a Skill file loaded on demand, not in the baseline context.

Here is the same project before and after a single pruning pass:

Before—reactively grown:

```
# Project Context

This is a Python web application using FastAPI and PostgreSQL...

## Architecture
The application is divided into three layers: the API layer (src/api/),
the service layer (src/services/), and the data layer (src/db/)...

## Testing
Run tests with pytest. We use pytest-asyncio for async tests. Make sure
to write tests for all new functionality. Tests should be readable and
well-documented. Aim for high coverage.

```

```
## Code Style
Follow PEP 8. Use type hints everywhere. Write docstrings for all public
functions. Keep functions small and focused. Write clean, readable code.

## Database
We use Alembic for migrations. Never run migrations directly—always use
`alembic upgrade head`. Don't modify existing migration files.

## Git
Write descriptive commit messages. Reference JIRA tickets in commits using
the format PROJ-XXXX. Use feature branches, never commit directly to main.

## Dependencies
Use pip to install dependencies. Add them to pyproject.toml.
```

After—pruned:

```
# Project Instructions

## Tooling
- Use `uv` not `pip` for all package operations: `uv add`, `uv sync`
- Run tests: `uv run pytest -x --tb=short`

## Non-obvious rules
- JIRA ticket reference required in all commits: format PROJ-XXXX
- Never run Alembic migrations without `--dry-run` first in local dev
- Deploy script is scripts/deploy.sh—requires explicit approval before use
```

The pruned version is 11 lines. The original is 57. The three instructions that remain are all non-inferable from reading the codebase and have concrete behavioural impact.

Before adding anything: four inclusion tests

An instruction earns inclusion only if it passes all four:

1. **Failure-backed.** You have seen the agent fail without this instruction in a repeatable pattern—not just once.
2. **Non-inferable.** The information isn't discoverable by an agent reading the codebase directly.
3. **Always-relevant.** The instruction applies to the majority of tasks the agent performs in this directory.

4. **Triggerable.** You can describe a concrete task scenario where following the instruction produces a measurably different outcome than not having it.

If an instruction fails any of these tests, it does not belong in the file.

Treating AGENTS.md like code

When you add or remove an instruction, write a commit message that explains why. "Add uv instruction: agent was using pip on three consecutive sessions despite uv being available" is a useful commit message. "Add uv instruction" is not—it gives you no way to evaluate whether the instruction is still needed when you review it later.

Review `AGENTS.md` changes in pull requests with the same scrutiny as code changes. Add a recurring task (quarterly is a reasonable cadence) to audit the file against recent agent sessions, remove instructions that aren't showing up as behavioural influences, and check that the combined byte size is lean enough to avoid wasting tokens (even though the ~5 MB default ceiling is generous).

The fastest way to test whether an instruction is doing work is to remove it temporarily and run a few representative tasks. If the agent's behaviour is identical, the instruction was dead weight.

* * *

Starter Template

The following template is ready to copy into any new project. Fill in the bracketed placeholders and delete any sections that don't apply.

```
# Project: [Project Name]

## Overview
[One to three sentences: what the project does, primary language/framework,
↪  deployment target.]

## Commands
- Build: `[build command]`
- Test: `[test command]` (must pass before committing)
- Lint: `[lint command]`
- Type-check: `[typecheck command]` (if applicable)
- Start dev server: `[dev command]` (port [port])

## Repo layout
- `src/` — application source
- `tests/` — test files (mirror `src/` structure)
- `scripts/` — maintenance scripts; not production code
- [Add other top-level directories with one-line descriptions]

## Commit conventions
- Format: `<type>(<scope>): <description>`
- Types: feat, fix, docs, style, refactor, test, chore
- [Add any project-specific requirements, e.g., ticket references]

## Code style
- [Language/framework version]
- [Key style rules that linters don't catch]
- [Preferred patterns for error handling, logging, database access]

## Constraints
- Do NOT modify: [list generated or off-limits paths]
- Do NOT install new dependencies without noting them in the PR description
- [Any other hard limits]

## Architecture notes
- [Non-obvious dependencies or patterns]
- [Recurring gotchas to avoid]
```

* * *

The Plans.md Technique

Alex Embiricos, product lead for Codex at OpenAI, advocates a discipline that complements `AGENTS.md`: before writing any code, prompt Codex to generate

a milestone-based implementation plan in a `Plans.md` file.[13] The workflow is straightforward. First, describe the feature or project goal and ask Codex to produce a plan broken into numbered milestones, each with concrete deliverables. Then execute one milestone at a time, prompting Codex to read `Plans.md` before starting each step and to update it with findings before moving on.

The technique works because it forces task decomposition up front, creates an audit trail of planned versus actual work, reduces context drift on long-running tasks, and gives humans a natural checkpoint for course-correction between milestones. OpenAI used Plans.md internally to build the Sora Android app in 28 days—the plan kept Codex aligned with the overall architecture across dozens of sessions.[14]

If you already maintain an `AGENTS.md`, the two files serve different roles: `AGENTS.md` encodes standing rules that apply to every session, while `Plans.md` encodes the specific execution roadmap for a current initiative. Together they give the agent both the project's constitution and its marching orders.

Validating AGENTS.md with Agnix

As your `AGENTS.md` files grow across a monorepo, manual review may not catch every structural problem. Agnix is an open-source config linter purpose-built for `AGENTS.md` validation: it checks for duplicate rules, detects contradictions between layers of the instruction chain, and flags content likely to be silently truncated by the `project_doc_max_bytes` limit.[15] Running it as a pre-commit hook or CI step turns `AGENTS.md` quality into an automated gate rather than a manual discipline.

* * *

[13] Embiricos, A. "Advanced Codex Workflows." *How I AI* podcast, Lenny's Newsletter, 12 January 2026. Embiricos describes the Plans.md technique and its use during the Sora Android development. https://www.lennysnewsletter.com/p/this-week-on-how-i-ai-the-power-users

[14] Embiricos, A. "Advanced Codex Workflows." *How I AI* podcast, Lenny's Newsletter, 12 January 2026. Embiricos describes the Plans.md technique and its use during the Sora Android development. https://www.lennysnewsletter.com/p/this-week-on-how-i-ai-the-power-users

[15] Agnix is a community-maintained linter for AGENTS.md files that validates structure, detects layer conflicts, and warns about truncation risk. https://github.com/agnix-ai/agnix

The File Map Pattern: Addressing the Navigation Failure Mode

Research published in April 2026 identifies navigation as the dominant failure mode for coding agents—accounting for 27–52% of all agent failures, far exceeding tool-use errors at less than 17% (see Chapter 4 for the full failure taxonomy). The AAR study, analysing 9,374 agent trajectories across diverse codebases, found that agents achieve only 37.2% accuracy on action recognition tasks, with the majority of failures stemming from agents getting lost in codebase structure rather than misusing tools once they find the right files.[16]

The file map pattern is the single highest-impact `AGENTS.md` addition supported by this research. A file map gives the agent a navigational aid that directly addresses the dominant failure mode: instead of reading directory after directory to build a mental model of the codebase, the agent starts every session with an explicit map of where things live and what they do.

The file map template

```
## File Map

### Core Application
- `src/api/routes/` — HTTP route handlers; each file maps to one API resource
- `src/api/middleware/` — Express middleware (auth, rate-limit, error-handler)
- `src/services/` — Business logic; no direct DB access; called by route
  ↪ handlers
- `src/models/` — Sequelize model definitions; one file per table
- `src/lib/db.ts` — Database singleton; all queries go through this
- `src/lib/logger.ts` — Structured logger; use this, never console.log

### Generated (DO NOT MODIFY)
- `src/generated/` — OpenAPI codegen output; regenerated on build
- `prisma/client/` — Prisma client; regenerated from schema

### Tests
- `tests/unit/` — Mirrors `src/` structure; run with `npm test`
- `tests/integration/` — Docker-dependent; run with `npm run test:integration`
- `tests/fixtures/` — Shared test data; JSON and SQL seed files

```

[16] AAR (Agent Action Recognition) study, April 2026. 9,374 agent trajectories analysed; 37.2% action recognition accuracy; navigation errors account for 27–52% of all failures, compared to <17% for tool-use errors. See also Chapter 4 for the full failure taxonomy from CocoaBench, HiL-Bench, and AAR.

```
### Infrastructure
- `infra/terraform/` — AWS infrastructure; never modify without approval
- `scripts/` — One-off maintenance scripts; not production code
- `.github/workflows/` — CI/CD pipelines; changes require platform team review
```

Why file maps work

The file map works because it converts an exploration problem into a lookup problem. Without a file map, the agent must issue multiple tool calls to read directories, open files, and build a mental model of the codebase structure—each of those calls consuming context window tokens and introducing opportunities for the agent to take a wrong turn. With a file map, the agent starts with the answer to "where does X live?" already in its context.

The file map is particularly effective for:

- **Monorepos**, where the directory structure is deep and the agent may spend a significant portion of its context budget simply navigating to the right service.
- **Codebases with non-obvious layouts**, where convention alone does not reveal the purpose of directories (e.g., a `lib/` directory that contains both shared utilities and database code).
- **Legacy codebases**, where historical naming conventions create confusion (e.g., a `helpers/` directory that actually contains core business logic).

Adoption data

The effectiveness of navigation aids in `AGENTS.md` is supported by adoption data: over 60,000 open-source repositories have adopted `AGENTS.md` as of April 2026, and community analysis of the most effective files consistently identifies repository layout and file map sections as the highest-impact content.[17] The file map pattern has become the standard recommendation from the AAIF community for addressing the navigation failure mode identified in the April 2026 benchmark research.

* * *

[17] AGENTS.md official specification site, accessed April 2026. Lists 25+ supported tools and 60,000+ project adoption. https://agents.md/

Summary

`AGENTS.md` is the mechanism that makes Codex CLI a reliable team collaborator rather than a capable but amnesiac assistant. Key points from this chapter:

- Codex CLI resolves an instruction chain by concatenating `AGENTS.md` files from `~/.codex/` to the repository root to your current working directory, with closer files winning on conflicts.
- `AGENTS.override.md` at any level takes precedence over `AGENTS.md` at the same level—useful for temporary personal overrides, service-specific rules, and CI/CD pipeline behaviour.
- The four essential sections are: commands (build, test, lint), commit conventions, code style, and constraints (off-limits files and required behaviours).
- The `project_doc_max_bytes` limit (now ~5 MB by default, up from the original 32 KiB) is enforced silently. Put critical rules first, distribute instructions across subdirectories, and validate the chain with the diagnostic commands in this chapter.
- In monorepos, layered `AGENTS.md` files let each service receive exactly the instructions relevant to it, with no cross-contamination.
- Fallback filenames let you point Codex CLI at an existing `CONTRIBUTING.md` or similar file without migration.
- For larger projects, a three-tier knowledge architecture—constitution (always loaded), specialist agents (per task type), and MCP knowledge servers (queried on demand)—provides a principled framework for deciding where each piece of context lives. Routing rules and architectural boundaries belong in the constitution; deep domain knowledge belongs in specialist agents.
- `AGENTS.md` is an open standard under Linux Foundation governance, supported by over 30 agent products and adopted by more than 60,000 repositories. Write tool-agnostic instructions in `AGENTS.md` and reserve tool-specific files for features that genuinely require them.
- ETH Zurich's AGENTbench study found that LLM-generated context files reduce task success rates by 3 per cent while increasing costs by 20 per cent. The content that drives positive outcomes is specifically non-standard tooling instructions and non-inferable team decisions—not architectural overviews or framework conventions.

- A pruning checklist—removing architectural content, generic practices, framework conventions, and anything inferable from the codebase—can reduce a reactively grown file by 60–80 per cent without losing any instruction that was actually doing useful work.
- Before adding any instruction, apply four tests: failure-backed, non-inferable, always-relevant, triggerable. An instruction that fails any of these four doesn't belong in the file.
- Treat `AGENTS.md` like code: review changes in pull requests, write commit messages that explain the failure case behind each addition, run the three-question validation test, and schedule quarterly pruning cycles against recent session samples.

* * *

Exercises

1. Write an `AGENTS.md` for a project you work on. Use the starter template as a starting point. Focus on the commands section first—can you list every command Codex CLI would need to build, test, lint, and type-check the project?
2. Run the three-question validation test against your new `AGENTS.md`. Do Codex CLI's answers to "how do I run the tests" and "what files should I never touch" match what you wrote? If not, revise the file until they do.
3. Take a monorepo you have access to and sketch a layered `AGENTS.md` structure for it. Which rules belong at the repo root, which belong at the service level, and which (if any) should use `AGENTS.override.md`? Estimate the byte size of each file and verify the total is lean enough to avoid unnecessary token spend (the ~5 MB default ceiling is generous, but bloat still wastes inference budget).
4. **The pruning pass.** Take an existing `AGENTS.md`—yours or a public project's—and apply the pruning checklist from this chapter. For each instruction, classify it as: architectural/structural (remove), generic best practice (remove), framework convention (remove), or non-inferable team decision (keep). Count the lines in each category. Apply the four inclusion tests to the survivors and note how many pass all four.

5. **The removal experiment.** Identify one instruction in your `AGENTS.md` that you believe is doing useful work but have never empirically verified. Remove it on a test branch. Run five representative agent tasks. Compare the results to five equivalent sessions with the instruction present. Does agent behaviour change? This exercise builds the habit of treating `AGENTS.md` maintenance as an empirical practice rather than a precautionary one.

Chapter 9. Approval Modes and Trust Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Trust Model: What Codex Can Touch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Four Approval Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Untrusted: maximum caution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

On-request: the practical default

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Never: fully automated pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The composite shortcut flags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Auto-Approve: When to Use It and When Not To

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Sandboxing: Filesystem and Network Restrictions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Configuring the sandbox in config.toml

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Network allowlisting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Kernel-Level vs. Hook-Based Sandboxing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

macOS: Seatbelt

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Linux: bubblewrap, Landlock, and seccomp

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Windows: restricted tokens and ACLs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Approval Mode Strategy for Teams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Granular approval policy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Named profiles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Distinct approval IDs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Rejection feedback

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Denylist-only sandbox networking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Project-level configuration and trust levels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Admin enforcement with requirements.toml

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The approval mode ladder

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Programmatic Approval with PermissionRequest Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

PermissionRequest Hook Deepening: Toward a Programmatic Policy Engine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

YOLO mode TUI header display

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Guardian Failure Modes and Escalation Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

False-positive fatigue

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Silent business-logic failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Escalation deadlocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The asymmetric feedback problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Deny-Read Glob Policies (v0.122.0)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CI Reproducibility Flags (v0.122.0)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

`--ignore-user-config`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

`--ignore-rules`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Combining the flags for deterministic pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 10. Debugging and Diagnosing Agent Failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Reading the Session Transcript

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Using Approval Mode as a Diagnostic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

TUI Slash Commands for Live Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Diagnosing AGENTS.md Failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Using `--debug` to verify loading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Common silently-failing patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Authentication failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Testing Commands Under the Sandbox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Validating Execution-Policy Rules Offline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Context Overflow Symptoms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Recovering a Runaway Session

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Structured Logging and `--debug`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Module-Targeted Tracing with RUST_LOG

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The codex debug Subcommand

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Terminal notifications and agent alerting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

OpenTelemetry Tracing for Agent Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What Gets Traced

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exporting to Backends

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Debugging Agent Loops with Traces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Model Catalog Introspection with `codex debug models`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Statsig Analytics for Enterprise Usage Tracking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

A Diagnostic Workflow Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 11. Model Selection and Reasoning Effort

Chapter 9 explored approval modes, the mechanism that governs *what* Codex CLI is permitted to do. You learned how to tune the agent's autonomy from fully supervised `untrusted` mode all the way to unsupervised `never` mode, and how the trust boundary between human oversight and machine action shifts at each setting. Approval modes answer the question of control: will the agent ask before acting, or proceed on its own judgment?

This chapter pivots to a related but distinct concern: once you've decided how much autonomy to grant, which model should execute the task, and at what level of reasoning effort? These two knobs—model and reasoning effort—determine how well Codex CLI performs and how much that performance costs you. Model selection determines the ceiling of capability; reasoning effort determines how close to that ceiling the model operates for any given request. Choosing the wrong model for a task wastes quota and produces work of inappropriate quality in either direction: over-engineered output from an expensive model on a trivial lookup, or shallow results from a lightweight model on a hard architectural problem. By the end of this chapter, you will have a systematic approach to matching both model and effort to task across every scenario you are likely to encounter.

Learning Objectives

You will be able to:

- Configure a persistent model and effort default in `~/.codex/config.toml` and override it per-invocation with `--model` and `-e` flags
- Build named profiles that let you switch between daily-driver, deep-investigation, and quota-saving configurations with a single flag
- Use the `/effort` slash command to change reasoning depth during an active session without restarting

- Write a custom subagent definition that pins a worker to `gpt-5.4-mini` with reduced effort
- Design multi-subagent task graphs that allocate reasoning budget where it produces the most return
- Estimate monthly Codex CLI quota spend for a small team and identify where switching models saves the most

* * *

The Available Models

Codex CLI supports five models as of late April 2026.[1] Set the active model with `--model`, the `model` key in `config.toml`, or `/model` during an interactive session. The default is `gpt-5.4`, though teams with access to GPT-5.5 should evaluate it for their most demanding tasks.

Table 13-1 summarises the five models side by side. The "Quota cost" column uses `gpt-5.4` as the 100 per cent baseline; the other values are relative to that baseline.

Table 13-1. Codex model comparison (April 2026)

Model	Context window	Quota cost	Max effort	Best use cases
`gpt-5.5`	400,000 tokens	~200 per cent (2x base-line)	`xhigh`	Frontier tasks requiring maximum capability, lowest hallucination rate, complex multi-file architecture

[1] OpenAI, "Codex Models," https://developers.openai.com/codex/models, accessed April 2026. Lists the four currently supported models and their capabilities.

Model	Context window	Quota cost	Max effort	Best use cases
`gpt-5.4`	200,000 tokens	100 per cent (baseline)	`xhigh`	Daily driver, orchestration, complex debugging, strong value option for routine work
`gpt-5.4-mini`	200,000 tokens	~30 per cent	`high`	Subagent workers, codebase exploration, parallel file review, quota-sensitive sessions
`gpt-5.3-codex`	200,000 tokens	–	`xhigh`	SWE-bench–style tasks, large multi-file refactors, hours-long autonomous engineering runs
`gpt-5.3-codex-spark`	–	–	–	Real-time iteration (Pro subscribers, research preview only)

GPT-5.5 (“Spud”), released on April 23, 2026, is the new frontier model.[2] It scores 82.7 per cent on Terminal-Bench 2.0 (SOTA), 58.6 per cent on SWE-bench Pro, and 88.7 per cent on SWE-bench. The 400K context window in Codex is large enough to hold a substantial portion of most codebases. The 60 per cent reduction in hallucinations compared to previous versions is especially significant for enterprise teams where fabricated file paths and invented API endpoints create costly verification overhead. API pricing is $5/$30 per million tokens – double GPT-5.4 – but OpenAI reports better token efficiency that may offset the sticker price for many tasks. If the model solves the problem in one pass instead of two, or in 800 output tokens instead of 1,500, the effective cost per task may stay roughly the same.

The quota cost for `gpt-5.3-codex` is unlisted in official documentation at the time of writing; treat it as comparable to `gpt-5.4` until OpenAI publishes a

[2] GPT-5.5 (“Spud”) released April 23, 2026. 400K context in Codex, Terminal-Bench 2.0 at 82.7% (SOTA), SWE-bench Pro at 58.6%, 60% hallucination reduction, API pricing $5/$30 per M tokens. https://openai.com/index/gpt-5-5/

specific ratio.

> **Warning:** OpenAI periodically deprecates older model versions. The `gpt-5.2-codex` and all 5.1 variants (`gpt-5.1-codex`, `gpt-5.1-codex-max`, `gpt-5.1-codex-mini`) have been removed from the ChatGPT model picker.[3][4] If your `config.toml` or subagent definitions reference a deprecated model, migrate to a supported replacement—see the migration table below. API-key authenticated users can still access any API-supported model or configure a custom model provider.

The key insight from Table 13-1 is that capability and cost are not perfectly correlated. `gpt-5.4-mini` is not a weaker `gpt-5.4`; it is a different model optimised for different work. Treating it as "cheaper but worse" misses the design intent. For the tasks it's built for—bounded subtasks, file-level analysis, exploration—it delivers appropriate quality at a fraction of the cost.

Migrating from deprecated models

If your team is still running on older models, the migration path is straightforward. The table below maps every deprecated model to its recommended replacement.

Table 13-1b. Deprecated model migration map

Deprecated model	Replacement	Notes
`gpt-5.2-codex`	`gpt-5.3-codex` or `gpt-5.4`	Fully removed for ChatGPT auth on April 14, 2026; API retirement June 5, 2026[5]

[3] Daniel Vaughan, "Codex CLI Model Lifecycle: Navigating Deprecations, Migrations, and the GPT-5.x Transition," https://danielvaughan.com/codex-resources/articles/2026-04-07-codex-cli-model-lifecycle-deprecations-migrations/, 7 April 2026. Comprehensive timeline of Codex model releases, deprecation waves, and migration guidance for the April 2026 landscape.

[4] OpenAI, "Codex Changelog," https://developers.openai.com/codex/changelog, accessed April 2026. Records the April 7 model picker changes: `gpt-5.2-codex` and all 5.1 variants removed from the ChatGPT picker, with full removal for ChatGPT-authenticated users on April 14.

Deprecated model	Replacement	Notes
`gpt-5.1-codex`	`gpt-5.3-codex` or `gpt-5.4`	Deprecated April 1, 2026[6]
`gpt-5.1-codex-max`	`gpt-5.3-codex` or `gpt-5.4`	Deprecated April 1, 2026[7]
`gpt-5.1-codex-mini`	`gpt-5.4-mini`	Deprecated April 1, 2026[8]
`gpt-5-codex` (original)	`gpt-5.4`	Deprecated April 1, 2026[9]
`gpt-5-codex-mini` (original)	`gpt-5.4-mini`	Deprecated April 1, 2026[10]

To migrate, audit your configuration files and CI/CD workflows for hardcoded model strings (`grep -rn "gpt-5\.\(0\|1\|2\)" ~/.codex/ .codex/`), update any `AGENTS.md` references, and test with the replacement model before rolling out to the full team.

> **Tip:** ChatGPT-authenticated users whose `config.toml` references a removed model will silently fall back to the current default (`gpt-5.4`). API-key users get a hard error—their pipelines break immediately. Either way, don't rely on fallback behaviour; update your configuration explicitly.

[5] GitHub Blog, "GPT-5.1 Codex, GPT-5.1-Codex-Max, and GPT-5.1-Codex-Mini deprecated," https://github.blog/changelog/2026-04-03-gpt-5-1-codex-gpt-5-1-codex-max-and-gpt-5-1-codex-mini-deprecated/, 3 April 2026. The entire GPT-5.0 and GPT-5.1 Codex family was deprecated in the April 1 deprecation wave.

[6] GitHub Blog, "GPT-5.1 Codex, GPT-5.1-Codex-Max, and GPT-5.1-Codex-Mini deprecated," https://github.blog/changelog/2026-04-03-gpt-5-1-codex-gpt-5-1-codex-max-and-gpt-5-1-codex-mini-deprecated/, 3 April 2026. The entire GPT-5.0 and GPT-5.1 Codex family was deprecated in the April 1 deprecation wave.

[7] GitHub Blog, "GPT-5.1 Codex, GPT-5.1-Codex-Max, and GPT-5.1-Codex-Mini deprecated," https://github.blog/changelog/2026-04-03-gpt-5-1-codex-gpt-5-1-codex-max-and-gpt-5-1-codex-mini-deprecated/, 3 April 2026. The entire GPT-5.0 and GPT-5.1 Codex family was deprecated in the April 1 deprecation wave.

[8] GitHub Blog, "GPT-5.1 Codex, GPT-5.1-Codex-Max, and GPT-5.1-Codex-Mini deprecated," https://github.blog/changelog/2026-04-03-gpt-5-1-codex-gpt-5-1-codex-max-and-gpt-5-1-codex-mini-deprecated/, 3 April 2026. The entire GPT-5.0 and GPT-5.1 Codex family was deprecated in the April 1 deprecation wave.

[9] GitHub Blog, "GPT-5.1 Codex, GPT-5.1-Codex-Max, and GPT-5.1-Codex-Mini deprecated," https://github.blog/changelog/2026-04-03-gpt-5-1-codex-gpt-5-1-codex-max-and-gpt-5-1-codex-mini-deprecated/, 3 April 2026. The entire GPT-5.0 and GPT-5.1 Codex family was deprecated in the April 1 deprecation wave.

[10] OpenAI, "Retiring GPT-4o and older models," https://openai.com/index/retiring-gpt-4o-and-older-models/, February 2026. Confirms GPT-5.2 Thinking retires from the API on 5 June 2026.

Independent Assessment: Nathan Lambert on GPT-5.4

The most detailed independent technical assessment of GPT-5.4 for agentic coding comes from Nathan Lambert's Interconnects analysis.[11] Lambert's core thesis: GPT-5.4 is the first model that excels across **four dimensions simultaneously** – correctness, ease of use, speed, and cost – rather than trading them off against each other.

"Death by a thousand cuts" fixed. Previous GPT models (5.1, 5.2, even 5.3-codex) would reliably fail on ancillary operations – git commands, package management, system tools (LaTeX, ffmpeg), file manipulation. Each failure required manual recovery. GPT-5.4 handles these operations reliably, eliminating the "random failure" tax on agentic sessions. For subagent workflows where a single failed `git add` can derail an entire pipeline, this reliability improvement is transformative.

Token efficiency. Lambert cites benchmark data showing GPT-5.4 achieves peak performance with substantially fewer tokens than predecessors. This directly impacts cost and speed in multi-agent workflows where token budgets compound across subagents.

Precise instruction following vs interpretation. GPT-5.4 follows instructions precisely, which Lambert contrasts with Claude's more "interpretive" approach. For agentic orchestration, precision is critical – when you spawn five subagents with specific mandates, you need them to do exactly what they were told, not creatively reinterpret the task. This is a model characteristic, not a quality judgment: Claude's interpretive approach excels at exploration and ambiguous problems, while GPT-5.4's precision excels at delegated execution.

Benchmark gap. Lambert identifies a significant gap in the evaluation landscape: existing benchmarks (SWE-bench, Terminal-Bench) measure correctness but **not speed or cost**. These are the practical differentiators that determine whether agents are usable in production. No cross-model-family comparison for computational efficiency exists yet.

The adoption data supports Lambert's assessment: 3 million weekly active Codex users (50 per cent growth in under one month), 70+ per cent monthly

[11] Nathan Lambert, "GPT-5.4 Is a Big Step for Codex," Interconnects, https://www.interconnects.ai/p/gpt-54-is-a-big-step-for-codex, April 2026. First serious external technical assessment of GPT-5.4 for agentic coding, with specific analysis of ancillary operation reliability and token efficiency.

token usage increase, and enterprise adoption from Cisco, Nvidia, Ramp, Rakuten, and Harvey.[12]

Competitive Landscape: Claude Opus 4.7

Lambert's assessment noted Claude's "interpretive" approach as a contrasting design philosophy to GPT-5.4's precise instruction following. That contrast sharpened on April 16, 2026, when Anthropic released Claude Opus 4.7—a significant update to the model most frequently encountered as an alternative in agentic coding workflows.[13]

The headline numbers: a 13 per cent improvement on coding benchmarks over Opus 4.6, 3.75-megapixel vision support (relevant for agents that process screenshots, diagrams, or UI mockups), and a new tokenizer—all at the same price as Opus 4.6. Opus 4.7 is also the first model deployed under Anthropic's Project Glasswing cybersecurity framework, which imposes additional pre-deployment security evaluation. For teams using Codex CLI with API-key authentication and a custom model provider pointed at the Anthropic API, Opus 4.7 is a drop-in replacement for Opus 4.6 that delivers meaningfully better coding performance without a cost increase.

The benchmark story is worth stating explicitly: Opus 4.7 scores 87.6 per cent on SWE-bench Verified, establishing a new ceiling for agent performance on the benchmark that has become the de facto standard for evaluating agentic coding systems.[14] That number matters for model selection because it provides a concrete reference point. When you are deciding whether to route a complex, multi-file engineering task to an OpenAI model or to an Anthropic model via a custom provider (or via the new Bedrock native provider), SWE-bench Verified is the closest thing the field has to an apples-to-apples comparison on realistic software engineering work. At 87.6 per cent, Opus 4.7 is competitive with or ahead of every other publicly benchmarked model on this metric as of April 2026.

[12] OpenAI, "Codex product metrics," April 2026. 3M WAU, 70%+ monthly token usage growth, and enterprise adoption figures cited in Lambert's analysis and confirmed by OpenAI's public statements.

[13] Anthropic, "Claude Opus 4.7 release," April 16, 2026. 13% coding benchmark improvement over Opus 4.6, 3.75-megapixel vision, new tokenizer, same pricing. First model deployed under Anthropic's Project Glasswing cybersecurity framework.

[14] Anthropic, "Claude Opus 4.7 SWE-bench Verified results," April 2026. 87.6% on SWE-bench Verified, the highest publicly reported score as of April 2026. SWE-bench Verified is a human-validated subset of SWE-bench designed to reduce noise from ambiguous or poorly specified test cases.

The broader takeaway for model selection is that the competitive landscape continues to compress. The benchmark gap Lambert identified—no cross-family comparison for speed and cost—still exists, but each successive release from both OpenAI and Anthropic narrows the raw capability gap and shifts the decision toward secondary factors: tokenizer efficiency, vision capability, security posture, and ecosystem integration. For Codex CLI users, the practical implication is straightforward: if you are already using a custom model provider for Anthropic models, update to Opus 4.7; if you are evaluating alternatives to the default OpenAI models, Opus 4.7 raises the bar that any alternative must clear.

Cloud Model Providers: Azure, Bedrock, and Multi-Cloud

API-key authenticated users are not limited to OpenAI's model family. Codex CLI supports custom model providers configured in `config.toml` under `[model_providers]`. The `wire_api` setting maps a provider's native API to the Responses API contract Codex CLI expects, so the rest of your configuration (prompts, tools, approval policies) works without modification.

The provider landscape expanded significantly with the addition of Amazon Bedrock as a native provider (#18744). Enterprise teams on AWS can now use Bedrock models—Claude via Bedrock, Titan, Mistral—without third-party proxy configuration. Combined with the existing Azure OpenAI support, Codex CLI now has first-party providers for all three major clouds:

Cloud	Provider mechanism	Example models
Azure	Azure OpenAI model provider	GPT-5.4 (Azure-hosted), GPT-5.4-mini
AWS	Amazon Bedrock native provider	Claude (via Bedrock), Titan, Mistral
GCP	Custom model provider (Vertex AI)	Gemini, Claude (via Vertex)

The multi-cloud story matters most for enterprise teams with existing cloud commitments. If your organisation mandates that all inference traffic stays within AWS, the Bedrock provider eliminates the need for an external proxy or gateway—you configure the provider in `config.toml`, authenticate

with your existing AWS credentials, and Codex CLI routes requests through Bedrock natively. The same logic applies to Azure shops that were already using Azure OpenAI. GCP support via Vertex AI requires a custom provider configuration but follows the same `wire_api` pattern.

For teams evaluating cross-cloud strategies, the practical implication is that model selection is no longer constrained by cloud vendor. You can run OpenAI models through Azure, Anthropic models through Bedrock, and Google models through Vertex—all from the same Codex CLI configuration, switching between them with profiles or the `/model` command.

Local Model Providers

Local inference servers—Ollama, vLLM, llama.cpp—can also serve as model backends using the same `[model_providers]` configuration. The `wire_api = "responses"` value handles the protocol mapping.

Profile switching makes this practical: define a `[profiles.local]` that targets a local model for exploration and a `[profiles.cloud]` that targets the flagship model for implementation work. Local models are best suited for bounded, mechanical tasks—file exploration, formatting, simple lookups—where latency and cost advantages outweigh the capability gap. The full setup is beyond this chapter's scope; see the article index for detailed configuration guides.

* * *

Reasoning Effort: The Second Knob

Where model selection determines *which* model runs your task, reasoning effort determines *how deeply* that model thinks before it responds. The two knobs are orthogonal: you can run `gpt-5.4` at `minimal` effort (fast, shallow) or at `xhigh` effort (slow, deep), and you can set `gpt-5.4-mini` anywhere from `minimal` to `high`.

Before reasoning models emit a single word of response, they produce a hidden stream of internal tokens—sometimes called reasoning tokens or

thinking tokens—that represent the model working through the problem.[15] You do not see these tokens in the conversation, but you pay for them, and they have a direct effect on response quality for hard problems.

Reasoning effort is the dial that controls how many of those internal tokens the model is permitted to use.[16] The `model_reasoning_effort` configuration key controls this setting, with five valid values: `minimal`, `low`, `medium`, `high`, and `xhigh`. `medium` is the recommended interactive default.

The key insight is that reasoning tokens are not free quality. They're a compute budget you are allocating on behalf of the task. Allocating too little budget to a hard problem produces mediocre output. Allocating too much to an easy problem burns tokens without improving the result—the model does not use the budget it does not need for trivial work, but you still wait longer. Matching effort to task type is therefore an engineering discipline, not just a preference.

> **Note:** Reasoning tokens appear in your billing but not in the conversation transcript. If a response seems expensive relative to its visible length, reasoning tokens are almost always the explanation.

What each level does

`minimal` uses the fewest reasoning tokens and returns responses as fast as the model can produce them. Use it for tasks where the answer is mechanical: renaming a variable, adding a docstring to a simple function, reformatting a JSON file, or applying a single-line fix where the correct edit is unambiguous. If the task requires any multi-step reasoning or judgment, `minimal` will frequently produce incomplete or incorrect output.

`low` allocates a small but non-trivial reasoning budget. It handles well-defined, narrowly scoped tasks competently: generating boilerplate from a clear specification, performing a data transformation where the mapping rules

[15] OpenAI, "Introducing OpenAI o1-preview," https://openai.com/index/introducing-openai-o1-preview/, September 12, 2024, accessed March 2026. OpenAI's o1-preview was the first model to publicly expose reasoning tokens—a hidden scratchpad of internal computation that the model uses before generating its visible response. The model may generate anywhere from a few hundred to tens of thousands of reasoning tokens depending on problem complexity.

[16] The `model_reasoning_effort` key and its five valid values (`minimal`, `low`, `medium`, `high`, `xhigh`) are defined in the Codex configuration reference. https://developers.openai.com/codex/config-reference

are fully stated, or filling in a template with known values. The distinction from `minimal` is that `low` can follow a short chain of logic without getting lost.

`medium` is the general-purpose level and OpenAI's recommended default for interactive coding work. It balances quality and cost well enough that most developers never need to touch the setting during day-to-day work. Multi-file edits, standard refactors, moderate debugging tasks, and ordinary feature additions all sit comfortably within what `medium` can handle reliably.

`high` enables significantly more thorough reasoning. Suitable for complex bug investigation, architectural decisions, code reviews that require synthesised judgment across many files, or planning a multi-step refactor where getting the plan right matters more than getting it fast. Expect responses to be slower and more expensive than `medium`, but expect the quality ceiling to be meaningfully higher on hard problems.

`xhigh` is the maximum and the specialist level. The model thinks extensively before responding, which is noticeably slower and proportionally more expensive. It is specifically designed for non-latency-sensitive tasks where quality matters more than speed: long-horizon autonomous tasks that run unattended, hard algorithmic problems, deep security audits, or compliance reviews that need to catch subtle edge cases.[17] The official guidance is to use `xhigh` only when evaluation data shows a clear benefit that justifies the additional latency and cost. It's not supported by `gpt-5.4-mini`.

Table 13-2 summarises the decision logic for effort levels.

Table 13-2. Effort level decision guide by task type

Effort level	Reasoning token budget	Typical speed	Use when…
`minimal`	Fewest	Fastest	Task is mechanical: rename, reformat, single-line fix

[17] OpenAI, "Reasoning models," https://platform.openai.com/docs/guides/reasoning, accessed March 2026. The `xhigh` reasoning effort level is designed for non-latency-sensitive tasks that require extended deliberation: security audits, hard algorithmic problems, and long-horizon autonomous workflows validated on math, science, coding, and visual reasoning benchmarks.

Effort level	Reasoning token budget	Typical speed	Use when…
`low`	Low	Fast	Task is well-defined and narrowly scoped: boilerplate, data transforms
`medium`	Moderate	Responsive	Default interactive work: everyday coding, standard refactors
`high`	High	Slower	Multi-file refactors, hard debugging, architectural decisions, plan mode
`xhigh`	Most	Slowest	Long-horizon autonomous tasks, hard algorithms, security audits

Warning: `xhigh` can produce reasoning token bills 8–15 times higher than `medium` for the same task. Reserve it for problems that genuinely require it and confirm the cost before running it at scale.

The plan_mode_reasoning_effort key

Plan mode, Codex CLI's interactive planning interface, has its own reasoning effort override. Setting `plan_mode_reasoning_effort` in your `config.toml` changes the effort level used specifically during planning sessions without affecting regular sessions:

```
model = "gpt-5.4"
model_reasoning_effort = "medium"
plan_mode_reasoning_effort = "high"
```

This configuration runs everyday tasks at `medium` effort but spends more reasoning tokens during the planning phase, where getting a thorough upfront plan pays dividends across many subsequent implementation steps. When `plan_mode_reasoning_effort` is not set, plan mode falls back to its own built-in default, which is currently `high`—a deliberate choice because plan quality compounds through the entire implementation that follows.

> **Note:** Setting `plan_mode_reasoning_effort = "none"` means "use no extended reasoning in plan mode"; it does not mean "inherit the global setting." If you want plan mode to follow the global `model_reasoning_effort`, omit the `plan_mode_reasoning_effort` key entirely.

Quick reasoning controls (v0.124.0)

As of v0.124.0, you can adjust reasoning effort during an active session using keyboard shortcuts: `Alt+,` decreases effort by one level and `Alt+.` increases it.[18] This provides immediate, tactile control over the cost-quality trade-off without interrupting your flow to type `/effort`. The current effort level is displayed in the TUI status bar.

The typical workflow: start a session at `medium`, bump to `high` with `Alt+.` when you hit a hard problem, then drop back to `medium` or `low` with `Alt+,` for mechanical follow-up tasks. This is faster than the `/effort` slash command and encourages more granular effort management within a single session.

Overriding effort per session

You can override effort per session using the `-e` flag at the command line:

```
codex -e high "Refactor the auth module to use JWT"
codex -e minimal "Add a docstring to this function"
codex -e xhigh "Find the race condition in this event loop"
```

The `/effort` slash command lets you change effort mid-session without restarting. Type it at the Codex CLI TUI prompt:

```
/effort high       # raise effort for the next request in this session
/effort medium     # return to the interactive default
/effort minimal    # fast mode for quick mechanical tasks
```

This is useful when a session shifts type: you start a session for everyday coding (`medium`), encounter a hard debugging problem that warrants deeper

[18] Quick reasoning controls (`Alt+, / Alt+.`) introduced in Codex CLI v0.124.0, April 23, 2026. Keyboard shortcuts for adjusting reasoning effort during active sessions.

reasoning (`high`), then return to mechanical edits (`minimal`) afterward. The `/effort` command changes only the current session; it does not modify `config.toml`.

> **Tip:** The `-e` flag is appropriate for one-off overrides. For any effort setting you reach for more than once a week, encode it in a profile instead.

Automation runs and the effort caveat

A known issue affects automated (scheduled) Codex runs: they execute at `medium` reasoning effort even when the global `model_reasoning_effort` is set to `xhigh`.[19] Automation definition files currently support `schedule`, `prompt`, and `cwd` fields but not a `reasoning` field. If your automated workflows rely on higher effort for quality, either verify that they produce acceptable output at `medium`, or monitor the upstream issue for when per-automation reasoning configuration is available.

Dynamic model routing and the /fast command

The `/fast` command switches the active session to `service_tier=priority`, which is approximately 1.5x faster at roughly 2x the token cost. It targets real-time iteration where latency matters more than cost—rapid prototyping, tight feedback loops, live debugging sessions where waiting an extra few seconds per turn breaks flow.

The natural workflow is **scaffold then harden**: use `/fast` for initial code generation and scaffolding, then switch back to standard reasoning (or `/effort high`) for review and hardening. This cost-quality routing pattern means you spend premium tokens only where depth matters, not on the exploratory work that precedes it.

The `/model` command lets you switch models during an active session without restarting. Combined with `/effort` and `/fast`, you have three runtime knobs for tuning the cost-quality-latency trade-off mid-session:

[19] The automation reasoning effort limitation—scheduled runs executing at `medium` regardless of the global `model_reasoning_effort` setting—is a known issue documented in the Codex configuration reference and changelog. https://developers.openai.com/codex/config-reference

- `/model` controls which model handles the next turn
- `/effort` controls how deeply that model reasons
- `/fast` controls the service tier (latency versus cost)

All three take effect immediately and persist for the remainder of the session. None of them modify `config.toml`.

When quota is exhausted, Codex CLI receives 429 (rate limit) or 529 (overloaded) responses and applies exponential backoff automatically. The CLI does not surface these retries visually in most cases—a long pause between responses is the primary signal that backoff is occurring. For API-key users, rate limits are per-organisation; for subscription users, they map to the weekly and hourly quota described in Chapter 17.

* * *

Task Taxonomy: Matching Model and Effort to Task

The most common failure mode in model selection is applying one model uniformly to all tasks. This section gives you a decision framework that's specific enough to apply without ambiguity.

Table 13-3. Task taxonomy—model and effort by task type

Task type	Recommended model	Recommended effort
Quick lookup, grep, extraction	`gpt-5.4`	`minimal`
File search, codebase exploration (subagent)	`gpt-5.4-mini`	`low`
Routine feature implementation	`gpt-5.4`	`medium`
Code review of a pull request	`gpt-5.4`	`medium`

Task type	Recommended model	Recommended effort
Complex bug investigation	`gpt-5.4`	`high`
Architectural design or refactor planning	`gpt-5.4`	`high`
Large-scale multi-file refactor	`gpt-5.3-codex`	`high`
Hours-long autonomous engineering run	`gpt-5.3-codex`	`high`
Hardest problems where evals confirm benefit	`gpt-5.4` or `gpt-5.3-codex`	`xhigh`

The last row deserves emphasis: don't reach for `xhigh` as a first instinct when a task seems hard. Use it when you have evaluation data—test suite results, output quality comparisons, or explicit feedback—that confirms it produces materially better results than `high`. The cost and latency difference is real, and for most hard problems, `high` is sufficient.

> **Note:** If you are unsure which category a task falls into, start at `medium` and move up only if the output reveals a reasoning gap. Moving up proactively costs tokens; moving up reactively costs one retry.

Cost and latency trade-offs

The cost and latency impact of effort tuning is driven almost entirely by reasoning tokens. Table 13-4 gives approximate multipliers relative to `medium` as a baseline. These figures are derived from community benchmarks and are model-dependent; exact values are not publicly documented by the model provider.[20]

Table 13-4. Approximate latency and cost multipliers by effort level

[20] OpenAI, "Reasoning best practices," https://developers.openai.com/api/docs/guides/reasoning-best-practices, accessed March 2026. OpenAI's guidance confirms that reasoning tokens are billed as output tokens but are not visible in the API response. The cost multipliers in Table 13-4 are approximate figures from community benchmarks; exact values vary by model and task type and are not officially published.

(relative to `medium`)

Effort level	Reasoning token multiplier	Relative latency	Notes
`minimal`	~0.1x	Very fast	Negligible reasoning overhead
`low`	~0.3x	Fast	70 per cent cost reduction vs `medium`
`medium`	1x (baseline)	Responsive	Default; suitable for most interactive work
`high`	~3–5x	Slower	Significant quality gain on hard problems
`xhigh`	~8–15x	Slowest	Specialist use only; verify cost before scaling

There is one important efficiency note if you are using `gpt-5.3-codex`: that model achieves better benchmark performance than `gpt-5.4` at the same `medium` effort while using approximately 30 per cent fewer thinking tokens. If you are on the SWE-specialist model, the quality/cost trade-off at `medium` is already better than it would be on the standard tier.

The latency impact compounds in interactive sessions. Switching from `medium` to `high` for a single response might add a few seconds; running an entire session at `high` adds those seconds to every turn, which affects the feel of the tool significantly. This is another reason to use profiles rather than a blanket effort increase: you pay the latency cost only when you have deliberately opted into it for a session that warrants it.

Named profiles in practice

The cleanest way to operationalize Table 13-3 is with named profiles in your `~/.codex/config.toml`. Instead of remembering which model and effort combination to use for each task, you define the presets once and invoke them by name:

```toml
[profiles.daily]
model = "gpt-5.4"
model_reasoning_effort = "medium"
approval_policy = "on-request"
sandbox_mode = "workspace-write"

[profiles.deep]
model = "gpt-5.4"
model_reasoning_effort = "xhigh"
approval_policy = "on-request"
sandbox_mode = "workspace-write"

[profiles.quick]
model = "gpt-5.4-mini"
model_reasoning_effort = "low"
approval_policy = "never"
sandbox_mode = "read-only"

[profiles.ci]
model = "gpt-5.4-mini"
model_reasoning_effort = "low"

[profiles.swe-bench]
model = "gpt-5.3-codex"
model_reasoning_effort = "high"
approval_policy = "never"
sandbox_mode = "danger-full-access"

[profiles.audit]
model = "gpt-5.4"
model_reasoning_effort = "xhigh"
```

You activate a profile with the `--profile` flag:

```
codex --profile deep "trace the root cause of this deadlock"
codex --profile quick "what does this function return?"
codex --profile swe-bench "implement all failing tests in the suite"
```

Warning: Named profiles are currently experimental and may change or be removed in future releases. They are not yet supported in the Codex IDE extension. For production workflows, fall back to explicit `--model` and `--config` flags if profile behaviour changes unexpectedly.

* * *

Cost Modelling: Estimating Monthly Spend

Quota management becomes a genuine concern for teams sharing a Codex CLI subscription. This section walks through a worked example for a three-person team to illustrate how model and effort selection affect monthly consumption.

Baseline assumptions

Assume a three-person team using Codex CLI as their primary development assistant, with the following approximate daily workload per developer:

- 20 routine tasks (feature implementation, code review, quick questions): `gpt-5.4` at `medium`
- 5 exploration or search tasks (codebase browsing, file review): `gpt-5.4-mini` at `low`
- 2 hard tasks (complex debugging, architectural decisions): `gpt-5.4` at `high`
- 1 occasional long-running task per week, shared across the team: `gpt-5.3-codex` at `high`

Per-developer daily quota consumption

Using `gpt-5.4` at 100 per cent quota as the baseline unit, and `gpt-5.4-mini` at 30 per cent:

- 20 routine tasks at 100 per cent each = 2,000 per cent of baseline
- 5 exploration tasks at 30 per cent each = 150 per cent of baseline
- 2 hard tasks at 100 per cent each (effort multiplier assumed moderate) = 200 per cent of baseline
- Daily total: approximately 2,350 units per developer

Across three developers: roughly 7,050 units per day, or approximately 211,500 units per 30-day month.

The mini substitution effect

If you substitute `gpt-5.4-mini` for `gpt-5.4` on all 20 routine tasks:

- 20 routine tasks at 30 per cent each = 600 per cent of baseline (down from 2,000 per cent)
- Daily saving per developer: 1,400 units
- Monthly saving across three developers: approximately 126,000 units, roughly 60 per cent of the baseline

The implication is significant: the majority of Codex CLI quota consumption for most teams comes from routine tasks that don't require flagship model capability. Routing those tasks to `gpt-5.4-mini` is the single highest-impact optimisation available.

> **Tip:** Start a session with `gpt-5.4-mini` for exploration and context-building, then switch to `gpt-5.4` for the implementation decision. Use the `/model` slash command during the session to switch without losing your context window.

Prompt Caching: The Third Cost Lever

Model selection and the mini substitution effect control *which* model processes your tokens. Prompt caching controls *how much you pay per token* on the model you have already chosen. The two strategies are complementary: switching routine tasks to `gpt-5.4-mini` reduces the per-task cost by 70 per cent; layering prompt caching on top reduces the per-token input cost by a further 50 per cent on cached tokens. Applied together, 40–60 per cent total savings are achievable on a typical team's monthly spend.

The mechanism is straightforward. When Codex CLI sends a request to the API, a significant portion of the input—the system prompt, your `AGENTS.md` content, tool definitions, and the earlier turns of the conversation—is identical across consecutive requests. The API recognises these repeated token prefixes and serves them from cache rather than reprocessing them. Cached input tokens are billed at a 50 per cent discount compared to uncached input tokens.

You do not need to enable prompt caching explicitly; the API applies it automatically when token prefixes match. What you *can* control is how consistently your prompts produce cacheable prefixes. The key strategies are:

1. **Keep static content at the front of the prompt.** System instructions, `AGENTS.md` content, and tool definitions should appear before dynamic content (user messages, tool results). Codex CLI already structures its prompts this way by default, but custom model provider configurations or MCP server injections can disrupt the prefix ordering.
2. **Avoid unnecessary prompt variation.** Randomised greetings, timestamps injected into system prompts, or per-request metadata that changes on every call break the cache prefix and force full reprocessing. If you are injecting context via hooks (Chapter 13), ensure the injected content is deterministic across requests within the same session.
3. **Use longer sessions rather than many short ones.** Each new session starts with a cold cache. A single session with 20 turns amortises the uncached first request across 19 cached follow-ups. Twenty separate single-turn sessions pay the full uncached price 20 times.
4. **Structure multi-agent workflows for prefix reuse.** When dispatching work to subagents, ensure they share the same system prompt and tool definitions where possible. Subagents that diverge in their prompt preamble each maintain a separate cache, reducing the overall hit rate.

For the full treatment of prompt caching mechanics, measurement techniques, and advanced strategies for maximising cache hit rates, see the dedicated article: `articles/2026-04-21-codex-cli-prompt-caching-maximise-cache-hits-cost-reduction.md`.[21]

The cost modelling implication is significant. Revisiting the three-person team example: if 70 per cent of input tokens across all sessions hit the cache (a realistic figure for teams running multi-turn interactive sessions), the effective input token cost drops by 35 per cent (70 per cent of tokens at 50 per cent discount). Combined with the mini substitution strategy for routine tasks, total monthly spend can drop by 40–60 per cent compared to a naive configuration that runs `gpt-5.4` at `medium` for everything with no attention to caching behaviour.

* * *

[21] Vaughan, D., "Codex CLI Prompt Caching: Maximise Cache Hits and Reduce Costs," codex-resources, April 2026. Detailed treatment of prompt caching mechanics, measurement techniques, and strategies for maximising cache hit rates. See `articles/2026-04-21-codex-cli-prompt-caching-maximise-cache-hits-cost-reduction.md`.

Model Selection in Automated Pipelines

When Codex CLI runs in CI/CD pipelines or as part of scheduled automation, model and effort selection become first-class engineering concerns rather than occasional adjustments. Several properties of automated contexts change the calculus compared to interactive use.

Configuration resolution in automation

Automated runs inherit from `config.toml` in the normal precedence order (CLI flags override project config, which overrides user config). However, the automation reasoning effort caveat applies: automation runs currently execute at `medium` effort regardless of the global `model_reasoning_effort` setting. Design your automation prompts to produce acceptable results at `medium` reasoning effort. If a task genuinely requires `high` or `xhigh`, run it interactively rather than scheduled.

In automated pipelines, tasks are typically well-specified and narrowly scoped by design. Running them at `high` or `xhigh` does not improve the output; it increases the bill and slows the pipeline. Default automated workflows to `low` unless specific tasks require more.

Automated pipelines also often involve large numbers of repetitions. A task that costs 3x at `high` versus `low` might seem manageable for a single run, but if that task runs 500 times per day across your organisation's repositories, the multiplier becomes a budget line item that engineering leadership will notice.

> **Warning:** Avoid setting a single high effort level as a global default in CI profiles. One ill-advised `model_reasoning_effort = "xhigh"` in a shared config file can generate unexpected costs across every automated run in the organisation.

Subagent model and effort configuration

The most important application of model and effort selection in automated pipelines is the orchestrator/worker pattern. You define a lightweight worker agent that handles parallelisable subtasks, while the orchestrating session uses the flagship model for planning and judgment.

Worker agents are defined as individual TOML files in `~/.codex/agents/` (personal, available in all sessions) or `.codex/agents/` (project-scoped):

```toml
# .codex/agents/refactor-worker.toml
name = "refactor-worker"
description = "Applies a single, well-defined refactor to one file. Takes exact instructions. Does not make judgment calls."
developer_instructions = """
Apply the specified transformation to the specified file. Make only the changes described.
Return a summary of what was changed. Do not edit other files. Do not spawn additional agents.
"""
model = "gpt-5.4-mini"
model_reasoning_effort = "low"
sandbox_mode = "workspace-write"
```

The orchestrating session—running `gpt-5.4` at `medium` or `high` effort—plans the refactor, identifies the files, and dispatches work to the `refactor-worker` agent. Each worker operates on a narrow, bounded file-level task where `gpt-5.4-mini` at `low` effort is entirely appropriate.

You can also assign effort at the individual agent level in a multi-agent TOML file, allocating reasoning budget precisely where it produces the most return:

```toml
[[agents]]
id = "planner"
prompt = "Break this feature into 5 non-overlapping implementation tasks"
reasoning_effort = "high"

[[agents]]
id = "implementer-1"
prompt = "Implement task 1: add the database migration"
reasoning_effort = "low"

[[agents]]
id = "tester"
prompt = "Write unit tests for the migration"
reasoning_effort = "low"

[[agents]]
id = "reviewer"
prompt = "Review all changes for security issues"
reasoning_effort = "high"
```

The pattern here is the core of subagent economics: the orchestrator and the reviewer—roles that require judgment—run at `high`, while the implementers

and testers—roles that execute a well-defined specification—run at `low`. This allocation can reduce total token spend by 50–70 per cent compared to running all agents at `medium`, while maintaining quality precisely where quality matters.

Purpose-built models for agentic sub-tasks

The orchestrator/worker pattern allocates different *effort levels* to different roles, but the next evolution allocates different *models*. Codex CLI's own guardian reviewer illustrates this shift. Prior to v0.122.0-alpha.3, the reviewer that checks proposed changes before they are applied used the same generic `gpt-5.4` as every other component. In v0.122.0-alpha.3, the reviewer switched to `codex-auto-review`, a specialised model variant tuned specifically for automated code review. The result is both higher review accuracy—fewer false positives that block valid changes, fewer missed issues that slip through—and lower cost, because a purpose-built model can reach the same quality ceiling with fewer reasoning tokens than a general-purpose frontier model applied to the same narrow task.

The significance extends beyond a single release note. As agentic systems mature, expect sub-tasks to acquire their own optimised models rather than sharing a single frontier model across every role. Review is the first concrete example, but the pattern generalises: purpose-built models for planning, code generation, testing, security analysis, and other agentic sub-tasks are the logical next steps. For practitioners designing multi-agent workflows, the implication is that the `model` field in your agent TOML definitions is not just a choice between flagship and mini—it will increasingly be a choice between general and specialist, with specialist models offering better quality-per-token on the tasks they were built for.

> **Note:** Fields omitted from a subagent definition inherit from the parent session: `model`, `model_reasoning_effort`, `sandbox_mode`, `mcp_servers`, and `skills.config` all fall through if not explicitly set. Pin the model explicitly in your worker definitions to prevent workers from accidentally inheriting an expensive flagship model from the orchestrator.

> **Tip:** A useful heuristic for subagent effort allocation: if the agent's job is to decide what to do, use `high`; if its job is to do what it was told, use `low`.

Controlling concurrency

The `[agents]` section of `config.toml` governs how many subagents can run simultaneously and how deeply they can nest:

```
[agents]
max_threads = 6
max_depth = 1
```

The default `max_depth = 1` allows direct child agents to spawn but prevents deeper nesting. Increase this value only if you specifically need recursive delegation—unbounded nesting depth can produce runaway token consumption that is difficult to detect before quota is exhausted.

* * *

Part 2 Summary

Part 2 covered the five pillars of working effectively with Codex CLI at an intermediate level. Chapter 7 established how to write prompts that produce reliable, reproducible results. Chapter 8 introduced the `AGENTS.md` instruction file as the mechanism for giving the agent persistent, project-specific context. Chapter 10 addressed what happens when that file grows out of control, and how to keep it maintainable. Chapter 9 drew the line between what the agent is trusted to do autonomously and what requires human approval. This chapter closed the loop by showing you how to control the quality and cost of every agent action through deliberate model selection and reasoning effort allocation.

Together, these five concerns—prompting, instruction files, instruction hygiene, approval modes, and model selection and reasoning effort—form the operational foundation for any serious Codex CLI workflow. They're the decisions you make once, encode in configuration, and revisit when something isn't working. Master them, and you have reliable control over an agent that can do substantial engineering work on your behalf.

Part 3 turns to the extension stack: the additional capabilities you bolt onto the foundation. The Model Context Protocol, hooks, and the skills ecosystem

each add a new dimension to what Codex CLI can reach, intercept, and automate, without requiring you to change how the core agent works.

> **What's Next:** Part 3 is the extension stack: the three mechanisms that let you wire Codex CLI into the rest of your tooling and encode team knowledge in reusable form. Chapter 12 introduces the Model Context Protocol, which gives the agent typed, authenticated access to external systems: your databases, issue trackers, and internal APIs. Chapter 13 covers hooks, the lifecycle callbacks that let you inject context, gate prompts, and fire side effects at session boundaries. Chapter 14 brings both threads together with skills, the packaging format for portable, reusable agent workflows that install in one command and work across every adopting platform.

System Prompt Architecture: How Model Selection Affects the Agent's Instructions

An April 2026 system prompt comparison analysis reveals a dimension of model selection that most practitioners are unaware of: Codex CLI ships different system prompts for different model generations, meaning your model choice affects not just capability and cost but the fundamental instructions the agent operates under.[22]

The three prompt tiers in Codex CLI are:

[22] Vaughan, D., "System Prompts Compared: Codex CLI vs Gemini CLI vs Claude Code," codex-resources, April 2026. Analysis revealing Codex CLI's unique per-model prompt switching architecture with three prompt tiers: full (24K tokens), compact (7K tokens), and inline (~14.7K tokens).

Model tier	System prompt size	Example models	Design intent
Base models (full prompt)	~24,000 tokens	`gpt-5.4`, `gpt-5.4-mini`	Comprehensive instructions for general-purpose models that need detailed guidance on tool use, safety, and coding conventions
Fine-tuned Codex models (compact prompt)	~7,000 tokens	`gpt-5.3-codex`, `gpt-5.3-codex-spark`	Minimal instructions; the model's fine-tuning already encodes coding behaviour that the base model needs spelled out
Inline models (intermediate prompt)	~14,700 tokens	GPT-5.4 inline variant	Mid-size prompt for models that handle specific invocation patterns

This per-model prompt switching is unique to Codex CLI. Claude Code and Gemini CLI use a single system prompt regardless of the model version—their prompt may vary by feature configuration, but not by model generation. Codex CLI's approach acknowledges that a fine-tuned Codex model already "knows" coding conventions that a base model needs to be told, and avoids wasting context window tokens restating what the model has already learned.

The practical implication: when you switch from `gpt-5.4` to `gpt-5.3-`

`codex`, you are not just changing the model—you are also reducing the system prompt overhead by approximately 17,000 tokens. For context-constrained sessions (large codebases, many MCP servers), this reduction can be significant. Conversely, when you switch from a Codex model to a base model, the larger system prompt consumes more of your context budget, leaving less room for your `AGENTS.md`, conversation history, and tool results.

* * *

Summary

- As of April 2026, Codex CLI supports four models: `gpt-5.4` (flagship, recommended default), `gpt-5.4-mini` (fast and quota-efficient, roughly 30 per cent of the cost of `gpt-5.4`), `gpt-5.3-codex` (specialist software engineering model for SWE-bench–style work), and `gpt-5.3-codex-spark` (real-time iteration, Pro only, research preview). All older models—`gpt-5.2-codex` and the entire 5.1 family—are being removed from the ChatGPT model picker as of April 7, with full removal on April 14; API-key users retain access to any API-supported model
- Reasoning effort is a second, orthogonal dimension controlled by `model_reasoning_effort`: `minimal` and `low` for fast, shallow tasks; `medium` for routine interactive work; `high` for complex debugging and architectural decisions; `xhigh` only when evaluation data justifies it
- The `plan_mode_reasoning_effort` key lets you independently tune effort for Plan mode without affecting regular sessions; omit it to inherit the built-in default of `high`
- Per-session overrides are available via the `-e` flag at invocation and the `/effort` slash command mid-session; for recurring configurations, encode them in named profiles
- Automation runs execute at `medium` effort regardless of the global `model_reasoning_effort` setting—a known limitation to design around; default CI workflows to `low` with targeted `high` for judgment roles
- The orchestrator/worker pattern assigns `gpt-5.4` to planning and judgment and `gpt-5.4-mini` to parallelisable subtasks; allocating `high` to orchestrators and `low` to executors can reduce total token spend by 50–70 per cent without sacrificing quality where it matters

- Amazon Bedrock is now a native provider (#18744), joining Azure OpenAI to give Codex CLI first-party cloud provider support across AWS, Azure, and GCP (via custom provider); enterprise teams can route inference through their existing cloud without third-party proxies
- Claude Opus 4.7 scores 87.6 per cent on SWE-bench Verified, the highest publicly reported score as of April 2026, establishing a new ceiling for agentic coding performance and raising the bar for model selection decisions involving custom providers
- Prompt caching provides a third cost lever beyond model selection and effort tuning: cached input tokens are billed at a 50 per cent discount, and structured prompting strategies can achieve 40–60 per cent total savings when combined with the mini substitution effect

Exercises

1. **Profile audit.** Create named profiles in your `~/.codex/config.toml`—`daily`, `deep`, `quick`, and `ci`—matching the configurations in this chapter. Run the same moderately complex prompt (such as "explain the data flow through this module") using the `daily`, `deep`, and `quick` profiles and compare the response quality and time-to-first-token. Note the point at which the `daily` and `deep` profiles diverge.
2. **Worker agent definition.** Write a project-scoped subagent definition at `.codex/agents/explorer.toml` that pins the model to `gpt-5.4-mini` with `low` effort and `read-only` sandbox mode. Design it for codebase exploration: give it developer instructions that tell it to return a structured summary of what it finds without making edits. Invoke it explicitly from a `gpt-5.4` orchestrating session and verify that the worker inherits the correct model rather than the orchestrator's model.
3. **Cost estimation exercise.** Using the worked example in this chapter as a template, estimate your own monthly Codex CLI quota consumption based on your actual daily task mix. Identify the two or three task types that consume the most quota, and propose a model substitution strategy—moving tasks from `gpt-5.4` to `gpt-5.4-mini` where appropriate—that would reduce your estimate by at least 40 per cent without sacrificing meaningful output quality.
4. **Effort level experiment.** Choose one task in your workflow where you suspect `xhigh` would meaningfully outperform `medium`—a hard debugging

problem, a security review, or a complex algorithmic design. Run it at both levels, compare the outputs, and assess whether the cost multiplier (8–15x) was justified by the quality difference.

5. **Subagent task graph design.** Design a three-agent task graph in a TOML file for a task you work on regularly. Assign model and effort levels to each agent based on the orchestrator/executor heuristic from this chapter. Estimate the token cost of running all three at `medium` versus your targeted allocation, using the multipliers in Table 13-4.
6. **Build a model decision tree for your team.** Using the task taxonomy from this chapter as a starting point, create a customised model and effort selection decision tree specific to your team's work. Include at least six task types you encounter regularly. For each, specify the model, reasoning effort level, and profile name you would use. Share the decision tree with one colleague and refine it based on their feedback. Add the final version to your team's AGENTS.md or engineering handbook so it becomes a shared resource rather than individual knowledge.

The Extension Stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 12. MCP: Consuming and Serving

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What MCP Is—and What It Isn't

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Architecture: Hosts, Clients, and Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP and A2A: Complementary Protocol Layers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP outputSchema: Typed Tool Outputs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP Tool Annotations: Risk Vocabulary for Approval Policy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP Tool Namespacing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Parallel Tool Calls and External Event Injection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Parallel MCP tool execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Turn item injection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Connecting Codex CLI to MCP Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Stdio transport

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

HTTP transport

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Restricting tool exposure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Connecting to Common Servers (GitHub, Browser, Database)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Context Cost of MCP: What Gets Loaded

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enterprise MCP: Authentication, Scoping, and Restrictions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP Elicitations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Common MCP Gotchas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Building a Simple MCP Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Serving MCP from Codex

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex CLI as an MCP Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Starting the server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Registering with Claude Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The inverted direction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Beyond Read-Only: Write-Back Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Ticketing System Integration (Jira, Linear)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Communication Platform Integration (Slack, Teams)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Message quality matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Bidirectional Database Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The read-first invariant

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Fixture data generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Sandbox-Aware MCP Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP Resilience and Governance Hardening

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Safety Boundaries for Write-Enabled Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Explicit write confirmation hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Idempotency by design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Audit logs for all writes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Least-privilege credentials

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Remote MCP Executor Stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enterprise implications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Current status

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Docker MCP Toolkit: Containerised Tool Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Profile template cards

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Credential management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Connecting to Codex CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Remote HTTP MCP Transport

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What changed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enterprise implications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Context Fragments as MCP Injection Points

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

How fragments work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What this enables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP Debugging and Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The `/mcp verbose` command (v0.123.0)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Schema bloat and the system prompt tax

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Plugin MCP loading (v0.123.0)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP zombie process fix (PR #19753)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The MCP server explosion: late April 2026

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 13. Hooks: Intercepting the Agent Lifecycle

MCP extends the agent's reach outward; hooks give you control over the agent's own lifecycle: when a session opens, when a user submits a prompt, and when a session closes. Hooks inject context, enforce policy, log behaviour, and send notifications—without changing a single line of your `AGENTS.md`.

Learning Objectives

You will be able to:

- Configure all five hook events in `.codex/hooks.json`
- Write shell and Python hook scripts that inject context, enforce prompt policies, run test suites, and scan for secrets
- Reason about hook timeouts, error handling, and the trade-offs of synchronous execution

* * *

The Hook System: Overview and Events

Hooks are shell-command callbacks that fire at defined points in the agent lifecycle.[1] They run synchronously and communicate results through stdout, stderr, and exit codes. As of v0.124.0 (April 23, 2026), hooks are **stable** — no

[1]The Codex CLI hooks system is described in the OpenAI Codex CLI changelog and documentation at https://github.com/openai/codex, accessed March 2026. Hook events `SessionStart`, `PreToolUse`, `PostToolUse`, `UserPromptSubmit`, and `Stop` were introduced across the v0.114.0–v0.117.0 release series.

feature flag is required.[2] Hooks are enabled by default when a `hooks.json` file is discovered in an active config layer, or when hook definitions are present inline in `config.toml`.

Prior to v0.124.0, hooks required an explicit opt-in via `[features] codex_hooks = true` in `config.toml`. This flag still exists for managed environments that need to *disable* hooks via `requirements.toml`, but the default has flipped — hooks are on unless explicitly turned off.

Hooks are fully supported on Windows. The same `hooks.json` configuration works on macOS, Linux, and Windows without platform-specific changes.

Codex CLI reads hook configuration from two sources: `.codex/hooks.json` from the project root, and inline `[[hooks]]` definitions in `config.toml` (added in v0.124.0). Both are hot-read; edits take effect without restarting. Table 16-1 maps the five lifecycle events to their primary use cases.

Table 16-1. Hook events and their characteristics

Event	Fires when	Stdout effect	Exit 2 effect	Can block?
`SessionStart`	Session opens	Injected into model context	Error logged; session continues	No
`PreToolUse`	Before each tool call executes	Logged or ignored	Tool call rejected; error shown	Yes (`Bash` only)
`UserPromptSubmit`	Prompt submitted, before model sees it	Appended to prompt	Prompt rejected; user sees stderr	Yes

[2] Codex CLI v0.124.0, released April 23, 2026. Hooks marked as stable, configurable in config files, and expanded to observe MCP tools alongside apply_patch and Bash sessions. https://github.com/openai/codex/releases/tag/v0.124.0

Event	Fires when	Stdout effect	Exit 2 effect	Can block?
`PostToolUse`	After each tool call completes	Logged or ignored	Error logged; execution continues	No
`Stop`	Session closes	None	Error logged; teardown continues	No

The `hooks.json` object is keyed by event name. Each event maps to an array of hook groups; each group contains a `hooks` array of individual hook objects:

```
{
  "hooks": {
    "SessionStart": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.codex/scripts/inject-context.sh",
            "statusMessage": "Loading project context…",
            "timeout": 15000
          }
        ]
      }
    ]
  }
}
```

The nested `hooks` array inside each group is not a mistake. The outer array holds hook groups; each group holds individual hooks. This structure supports conditional group activation.

Inline configuration in config.toml (v0.124.0)

As of v0.124.0, hooks can be defined inline in `config.toml`, collapsing the two-file setup into a single configuration surface:

```toml
# .codex/config.toml
[[hooks]]
event = "SessionStart"
command = "bash ~/.codex/scripts/inject-context.sh"
status_message = "Loading project context..."
timeout = 15000

[[hooks]]
event = "PostToolUse"
command = "bash ~/.codex/scripts/auto-test.sh"
status_message = "Running tests..."
timeout = 60000
```

Both `hooks.json` and inline `config.toml` definitions are supported simultaneously. When both exist, hooks from all sources fire in discovery order. The inline format is also supported in managed `requirements.toml` for enterprise environments that enforce hook policies centrally.

MCP and apply_patch observation (v0.124.0)

Prior to v0.124.0, `PreToolUse` and `PostToolUse` hooks fired only for `Bash` tool calls. v0.124.0 extends hook observation to **MCP tool calls** (prefixed `mcp__<server>__<tool>`) and `apply_patch`.[3] This means a `PreToolUse` hook with a matcher like `"mcp__docs__search"` can intercept, audit, or block MCP tool invocations before they execute.

The MCP observation pattern enables a new class of hook: the **MCP audit hook**, which logs every MCP tool invocation with full request and response payloads for compliance review.

[3]MCP and apply_patch hook observation, PR #18391 and v0.124.0 release. PreToolUse and PostToolUse hooks now fire for MCP tool calls and apply_patch in addition to Bash commands.

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "mcp__*",
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.codex/scripts/mcp-audit.sh",
            "statusMessage": "Auditing MCP call...",
            "timeout": 5000
          }
        ]
      }
    ]
  }
}
```

* * *

SessionStart: Configuring the Environment

`SessionStart` executes once before the model processes any user input—the right place for dynamic context that would go stale in `AGENTS.md`. On exit `0`, Codex CLI injects the script's stdout into the model's context before the first turn. Keep this output structured and brief: every token counts against your context window.

```
#!/usr/bin/env bash
# ~/.codex/scripts/inject-context.sh

BRANCH=$(git rev-parse --abbrev-ref HEAD 2>/dev/null || echo "unknown")
TICKET=$(echo "$BRANCH" | grep -oE '[A-Z]+-[0-9]+' | head -1)
DIRTY=$(git status --porcelain 2>/dev/null | wc -l | tr -d ' ')

echo "## Session Context"
echo "- Branch: $BRANCH"
echo "- Jira ticket: ${TICKET:-none detected}"
echo "- Uncommitted files: $DIRTY"

if [ -f ".codex/project-notes.md" ]; then
  echo ""
  cat .codex/project-notes.md
fi
```

Wire this script into your `hooks.json`:

```
{
  "hooks": {
    "SessionStart": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.codex/scripts/inject-context.sh",
            "statusMessage": "Loading branch context…",
            "timeout": 10000
          }
        ]
      }
    ]
  }
}
```

Exit 2 logs the error but continues the session (unlike `UserPromptSubmit`, where exit 2 blocks the prompt). If context injection fails, the session opens without the injected context.

As of v0.116.0, Codex CLI does not expose a `source` field distinguishing a fresh `startup` from a `resume` or context-compaction trigger. Whether `SessionStart` re-fires after `/clear` or an automatic compaction is not documented in the official changelog. Test this behaviour in your environment before writing scripts that assume they run exactly once per process lifetime.

* * *

UserPromptSubmit: Shaping Input Before the Agent Acts

`UserPromptSubmit` fires after you press Enter but before the model sees the text—and before the prompt enters session history. It can augment prompts with boilerplate or reject prompts that violate policy.

Augmenting prompts

On exit `0`, Codex CLI appends the hook's stdout to your prompt:

```
#!/usr/bin/env bash
# ~/.codex/scripts/append-test-status.sh
# Append current test suite status so the agent always knows what's failing

FAILING=$(python -m pytest --tb=no -q 2>&1 | tail -5)
if [ -n "$FAILING" ]; then
  echo ""
  echo "### Current test status"
  echo "$FAILING"
fi
```

A Python equivalent for complex augmentation logic:

```
#!/usr/bin/env python3
# ~/.codex/scripts/append-context.py
import subprocess, sys, os

result = subprocess.run(
    ["python", "-m", "pytest", "--tb=no", "-q"],
    capture_output=True, text=True
)
if result.returncode != 0:
    lines = result.stdout.strip().splitlines()[-5:]
    print("\n### Current test status")
    print("\n".join(lines))
```

Blocking prompts

Exit 2 rejects the prompt entirely—it never reaches the model and never enters history:

```
#!/usr/bin/env bash
# ~/.codex/scripts/prompt-policy.sh
# Block requests to push directly to main

PROMPT="${CODEX_PROMPT:-}"

if echo "$PROMPT" | grep -qi 'push.*main\|force.push\|--force'; then
  echo "Prompt blocked: direct push to main is prohibited. Open a PR instead."
  ↪  >&2
  exit 2
fi
```

Whether the prompt text is passed to the hook via an environment variable like `$CODEX_PROMPT`, via stdin, or via a temp file is not confirmed in the official documentation as of v0.116.0. Community implementations have used all three approaches. Test the mechanism against your installed version before shipping a policy script.

Set a tight `timeout` on `UserPromptSubmit` hooks. Because this hook blocks the prompt from proceeding, a hung or slow script creates a frustrating experience. Five seconds is usually enough for policy checks; thirty seconds is reasonable for test runners if you genuinely need that feedback before the model acts.

* * *

PreToolUse: Intercepting Shell Commands

`PreToolUse` fires before each tool call executes—the only hook that can prevent a tool call from running. In v0.117.0, it fires only for `Bash` tool calls; hooks matching `"Write"`, `"Edit"`, or other tool names do not fire. Configure with a `matcher` of `"Bash"`:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.codex/scripts/pre-tool-policy.sh",
            "statusMessage": "Checking command policy…",
            "timeout": 5000
          }
        ]
      }
    ]
  }
}
```

Exit `2` rejects the tool call. Exit `0` lets it proceed.

To block file-write operations to certain paths, use Codex CLI's `sandbox_mode` or `AGENTS.md` instructions rather than `PreToolUse`. No `PreToolUse` equivalent exists for non-Bash tools.

* * *

PostToolUse: Observing Without Blocking

`PostToolUse` fires after each tool call completes. The tool has already run; you observe the aftermath rather than gatekeep the action. Well-suited for:

- Running the test suite after every file write and injecting results back into context
- Scanning modified files for secrets or credentials immediately after they are written[4]
- Appending a structured log entry to an audit trail for every shell command executed

[4]The concept of "test on every save" using post-write hooks is a pattern from continuous testing workflows; see, for example, the pytest documentation on continuous integration at https://docs.pytest.org/en/stable/how-to/usage.html, accessed March 2026.

- Posting a notification when the agent runs a particularly sensitive operation

A "test on every save" hook in Python:

```python
#!/usr/bin/env python3
# ~/.codex/scripts/post-tool-test-runner.py
import subprocess, sys, json, os

# Read tool call details from stdin if provided
tool_info = {}
try:
    import json
    tool_info = json.load(sys.stdin)
except Exception:
    pass

tool_name = tool_info.get("tool", os.environ.get("CODEX_TOOL_NAME", ""))

# Only run tests after file-write tools
if tool_name not in ("write_file", "str_replace_editor", ""):
    sys.exit(0)

result = subprocess.run(
    ["python", "-m", "pytest", "--tb=short", "-q", "--no-header"],
    capture_output=True, text=True, timeout=60
)

if result.returncode != 0:
    # Exit 0 but write to stdout so results are available as context
    print("### Post-write test results")
    print(result.stdout[-2000:])  # trim to avoid flooding context

sys.exit(0)
```

`PostToolUse` stdout handling in v0.117.0 alpha is not yet fully specified. The pattern of exiting `0` and writing structured output to stdout is consistent with other events, but verify the exact injection behaviour against your version's changelog before building production workflows around it.

* * *

Stop: Teardown and Cleanup

`Stop` fires when the session ends. It does not inject into model context; exit 2 logs an error but does not block teardown:

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.codex/scripts/on-stop.sh",
            "statusMessage": "Wrapping up…",
            "timeout": 10000
          }
        ]
      }
    ]
  }
}
```

A typical `on-stop.sh`:

```
#!/usr/bin/env bash
# ~/.codex/scripts/on-stop.sh

TIMESTAMP=$(date -u +"%Y-%m-%dT%H:%M:%SZ")
BRANCH=$(git rev-parse --abbrev-ref HEAD 2>/dev/null || echo "unknown")

# Audit log
echo "$TIMESTAMP  branch=$BRANCH  user=$USER" >> ~/.codex/session-audit.log

# Desktop notification (platform-specific)
if command -v notify-send &>/dev/null; then
  notify-send "Codex" "Session complete on $BRANCH"
elif command -v osascript &>/dev/null; then
  osascript -e "display notification \"Session complete on $BRANCH\" with title
  ↪  \"Codex\""
fi

# Slack webhook (if configured)
if [ -n "$CODEX_SLACK_WEBHOOK" ]; then
  curl -s -X POST "$CODEX_SLACK_WEBHOOK" \
    -H 'Content-type: application/json' \
    --data "{\"text\":\"Codex session ended on \`$BRANCH\` at $TIMESTAMP\"}" \
```

```
    > /dev/null
fi
```

* * *

Writing Robust Hooks

Set timeouts aggressively. A 30-second `UserPromptSubmit` hook means waiting 30 seconds before the model acts on every prompt. Typical values: 5 seconds for policy checks, 10–15 seconds for context injection, 30–60 seconds for test runners.

Guard stdout size. `SessionStart` and `UserPromptSubmit` inject stdout into model context. An unbounded log dump can silently consume thousands of tokens. Trim to what the model actually needs.

Handle missing dependencies gracefully. A failed dependency (missing venv, absent Slack token) must exit `0` silently rather than exit `2` and block the session. Reserve exit `2` for deliberate policy rejections.

Understand hook status rendering. Hook status messages render in a dedicated live area in the TUI. The display appears after a 300ms delay (avoiding flicker for fast hooks), lingers for 600ms after completion so you can read the result, and collapses parallel hooks into a compact status line when multiple hooks fire simultaneously. Design your `statusMessage` strings to be informative at a glance—they may share screen space with other hooks running in parallel.

Test hooks in isolation. Run your hook script from the terminal before adding it to `hooks.json`; verify exit code, stdout, and stderr match expectations.

Add `set -euo pipefail` to bash hooks during development to catch errors early, then replace it with more selective error handling for production. Silent failures are harder to debug than loud ones when you are iterating.

* * *

Real Hook Patterns: Enforcement, Audit, and Notification

Style checks on every prompt

Wire a linter check into `UserPromptSubmit` so the model always knows the current lint state before it edits code:

```
#!/usr/bin/env bash
# Append ruff output to every prompt
OUTPUT=$(ruff check . --output-format=concise 2>&1 | head -20)
if [ -n "$OUTPUT" ]; then
  echo ""
  echo "### Linter state (ruff)"
  echo "$OUTPUT"
fi
```

Secret scanner on every file write

Use `PostToolUse` to scan for secrets immediately after the agent writes:[5]

```
#!/usr/bin/env bash
# ~/.codex/scripts/secret-scan.sh
# Requires: gitleaks (https://github.com/gitleaks/gitleaks)

if ! command -v gitleaks &>/dev/null; then
  exit 0
fi

OUTPUT=$(gitleaks detect --source . --no-git -q 2>&1)
if [ $? -ne 0 ]; then
  echo "### Secret scan warning"
  echo "$OUTPUT" | head -20
fi
```

[5] Gitleaks, "Find secrets with Gitleaks," https://github.com/gitleaks/gitleaks, accessed March 2026. Gitleaks is an open-source SAST tool for detecting hardcoded secrets (API keys, passwords, tokens) in git repositories, with over 19,000 GitHub stars and 20 million Docker downloads as of early 2026.

MITM detection on network operations

Security-conscious deployments can use `PostToolUse` hooks to detect man-in-the-middle attacks during agent network operations (PR #18168, shipped in v0.122.0-alpha.5). The pattern targets network-related tools—`curl`, `wget`, API calls—and inspects SSL certificate chains for anomalies that suggest interception or DNS manipulation. If the hook detects a suspicious certificate (unexpected issuer, shortened chain, mismatched SAN), it writes a warning to stdout and exits 2 to flag the session:

```
#!/usr/bin/env bash
# ~/.codex/scripts/mitm-detect.sh
# PostToolUse hook: check for certificate chain irregularities

TOOL_CMD="${CODEX_TOOL_CMD:-}"

# Only inspect network-related tool calls
echo "$TOOL_CMD" | grep -qiE 'curl|wget|http|fetch|requests' || exit 0

# Extract the most recent target host from the command
HOST=$(echo "$TOOL_CMD" | grep -oP 'https?://\K[^/: ]+' | head -1)
[ -z "$HOST" ] && exit 0

# Verify certificate chain; flag unexpected issuers or short chains
CERT_INFO=$(echo | openssl s_client -connect "$HOST:443" -servername "$HOST"
↪  2>/dev/null)
ISSUER=$(echo "$CERT_INFO" | openssl x509 -noout -issuer 2>/dev/null)
DEPTH=$(echo "$CERT_INFO" | grep -c "^ [0-9]")

if echo "$ISSUER" | grep -qiE 'mitmproxy|burp|charles|fiddler|zscaler'; then
  echo "### MITM warning" >&2
  echo "Suspicious certificate issuer detected for $HOST: $ISSUER" >&2
  exit 2
fi

if [ "$DEPTH" -lt 2 ]; then
  echo "### MITM warning" >&2
  echo "Unexpectedly short certificate chain for $HOST (depth=$DEPTH)" >&2
  exit 2
fi
```

Wire this into `PostToolUse` with a `Bash` matcher so it inspects every shell command the agent executes. In enterprise environments where agents make outbound API calls through corporate proxies, this pattern catches the gap between “the agent was told to call an API” and “the response actually came

from the intended server." Pair it with the secret scanner hook to cover both credential leakage and transport-layer interception.

Audit log

```
#!/usr/bin/env bash
ENTRY=$(python3 -c "
import json, os, time, subprocess

branch = subprocess.getoutput('git rev-parse --abbrev-ref HEAD')
print(json.dumps({
    'ts': time.strftime('%Y-%m-%dT%H:%M:%SZ', time.gmtime()),
    'user': os.environ.get('USER', 'unknown'),
    'branch': branch,
    'cwd': os.getcwd(),
}))
")
echo "$ENTRY" >> ~/.codex/audit.jsonl
```

Slack notification on session end

```
#!/usr/bin/env bash
[ -z "$CODEX_SLACK_WEBHOOK" ] && exit 0

BRANCH=$(git rev-parse --abbrev-ref HEAD 2>/dev/null || echo "unknown")
MSG="Codex session ended | branch: \`$BRANCH\` | user: $USER"

curl -s -X POST "$CODEX_SLACK_WEBHOOK" \
  -H 'Content-type: application/json' \
  --data "{\"text\":\"$MSG\"}" > /dev/null
```

PermissionRequest hooks: active governance

PR #17563, shipped in v0.122.0-alpha.7, extends the hook system from observation-only to active governance. Shell commands, unified execution, and network approval prompts now fire hook events. For the first time, external scripts can programmatically approve or deny permission requests,

with fallback to the standard TUI approval flow when the hook returns no decision.[6]

The hook receives the full request context — command, sandbox state, and Guardian review outcome — and returns one of three verdicts:

```
{"decision": "allow"}
{"decision": "deny", "reason": "Write to /etc not permitted for this agent role"}
{"decision": null}
```

When `decision` is `null` (or the hook exits without writing a decision), the standard TUI prompt appears as a fallback. This design means PermissionRequest hooks extend the approval system without replacing it.

Four patterns have emerged in early adoption:

1. **CI/CD auto-approval** — a hook checks whether a command matches a known-safe allowlist (`npm test`, `make lint`) and auto-approves; unknown commands escalate to human review.
2. **Policy-as-code** — the hook calls an OPA or Cedar policy engine to evaluate the request against project, branch, and role constraints.
3. **Audit trail logging** — every approval decision is forwarded to a SIEM or observability platform with full context, independent of the verdict.
4. **Multi-agent delegation scopes** — an orchestrator hook approves sub-agent actions only when they fall within the parent agent's delegated authority.

The net effect is that the approval system becomes a programmable security layer rather than a user interface feature. See Chapter 9 for the full coverage of PermissionRequest hooks within the approval mode architecture.

Self-evolving agent retraining loop

The most sophisticated hook application to date is the self-evolving agent pattern mapped from the OpenAI Cookbook retraining loop.[7] The pattern uses

[6] PermissionRequest hooks, PR #17563, merged in v0.122.0-alpha.7, April 2026. Shell, network, and unified execution approval prompts now fire hook events. https://github.com/openai/codex/pull/17563

[7] OpenAI Cookbook, "Building Self-Evolving Agents with Retraining Loops," April 2026. Four-stage retraining loop (baseline, evaluation, optimisation, deployment) reference architecture.

a closed feedback loop: a `PostToolUse` hook runs composite grading after each tool execution, and when scores fall below threshold, a metaprompt agent generates improved instructions that replace the current `AGENTS.md`.

The composite grading pipeline uses four complementary graders to prevent single-metric optimisation:

Grader	Type	Purpose
Entity preservation	Rule-based	Ensures critical terms survive transformations
Length discipline	Deterministic	Maintains target output size within tolerance
Semantic consistency	Cosine similarity	Guards against content drift
Holistic quality	LLM-as-judge	Captures nuanced signals that rules miss

The hook fires on `PostToolUse` with a `Bash` matcher. On each invocation, the grading script evaluates the tool's output against these four dimensions. When the average score drops below the configured threshold (typically 0.85), the system triggers a metaprompt optimisation pass that revises the `AGENTS.md` instructions. The improved version is committed to git, creating an auditable history of prompt evolution with eval scores recorded in commit messages.

This pattern closes the loop between execution and improvement: the agent's own output drives its instruction refinement. It requires significant upfront investment in eval infrastructure and is appropriate only for high-stakes, high-volume workflows where the evaluation cost (approximately 74,000 tokens per benchmark run via Skill Creator V2) is justified by the deployment stakes. For most teams, the simpler hook patterns described earlier in this chapter are the right starting point.

OpenTelemetry metrics for hook runs

As of v0.122.0-alpha.5 (PR #18026), hooks emit structured OpenTelemetry spans, closing the observability gap for hook-heavy enterprise deployments.

Each hook invocation produces a span tagged with the event type (`SessionStart`, `PreToolUse`, `PostToolUse`, `UserPromptSubmit`, `Stop`), the hook command, its exit code, and its wall-clock duration. These spans flow through any OTEL-compatible collector, so teams already using Datadog, Grafana, Honeycomb, or similar stacks can monitor hook execution latency, failure rates, and invocation-type distribution without custom plumbing.

A practical starting point: configure your OTEL collector endpoint and enable the hooks telemetry exporter:

```
# .codex/config.toml
[telemetry]
otel_endpoint = "http://localhost:4317"   # gRPC collector
otel_hooks = true
```

With spans flowing, you can build dashboards that answer questions like "which `PreToolUse` hooks add the most latency?" or "what percentage of `PostToolUse` invocations exit non-zero?" For teams running dozens of hooks across multiple event types, this turns hook performance from a black box into a first-class observable subsystem. Pair OTEL metrics with the audit-log pattern above for both real-time dashboards and durable offline records.

As of v0.124.0 (April 23, 2026), hooks are officially **stable**. The configuration format — both JSON (`hooks.json`) and inline TOML (`config.toml`) — is considered production-grade. The hooks API has matured through the v0.117.0–v0.124.0 release series, gaining OTEL telemetry, MITM detection patterns, PermissionRequest hooks, MCP observation, and apply_patch coverage. Pin your Codex CLI version when deploying hooks in team environments, and review the changelog before upgrading.

Hook listing and configuration APIs (PR #19778)

PR #19778, in the v0.126.0 alpha cycle (April 2026), introduces programmatic APIs for listing and managing hooks at runtime.[8] Prior to this change, hooks were exclusively file-driven: you edited `hooks.json` or `config.toml` and the

[8]Hook listing and configuration APIs, PR #19778, v0.126.0 alpha cycle, April 2026. Introduces programmatic runtime management of hooks alongside existing file-based configuration. https://github.com/openai/codex/pull/19778

agent discovered them on the next read cycle. The new APIs expose three capabilities:

1. **List** — enumerate all registered hooks across all configuration sources (JSON, inline TOML, managed `requirements.toml`), with their event bindings, matchers, timeouts, and source precedence.
2. **Enable/disable** — toggle individual hooks without removing them from configuration. This is particularly useful in CI pipelines where a slow `PostToolUse` hook should run in local development but be skipped in headless `codex exec` runs.
3. **Runtime inspection** — query hook execution statistics (invocation count, average duration, failure rate) during a session, complementing the OTEL telemetry pipeline with in-process diagnostics.

The shift from file-only to programmatic management means hooks can now be governed by the same infrastructure-as-code tooling that manages the rest of the development environment. An enterprise deployment can use the listing API to audit which hooks are active across a fleet of developer machines, and the enable/disable API to enforce policy centrally without modifying individual configuration files.

Plugin-bundled hooks

A new distribution pattern has emerged alongside the programmatic APIs: **plugin-bundled hooks**, where hooks are shipped inside plugins rather than configured as standalone scripts.[9] When a plugin includes a `hooks/` directory with hook definitions, installing the plugin automatically registers those hooks — no manual `hooks.json` editing required.

This pattern solves a deployment friction that has plagued hook adoption: distributing hooks across a team required each developer to copy scripts and update their configuration files. With plugin-bundled hooks, installing a plugin like `codex marketplace add security-scan` can register a `PostToolUse` secret scanner and a `PreToolUse` command policy checker in a single operation.

[9] Plugin-bundled hooks pattern, observed in the v0.125–v0.126 plugin ecosystem, April 2026. Plugins ship hook definitions in a `hooks/` directory that are auto-registered on installation, removing the need for manual configuration.

The hooks inherit the plugin's lifecycle: uninstalling the plugin removes its hooks, and updating the plugin updates its hooks. Managed `requirements.toml` policies can mandate specific plugins (and therefore their bundled hooks), giving enterprise administrators a clean enforcement path: require a plugin, get its hooks.

Delegated patch approval rendering (PR #19709)

When a patch approval is delegated — for example, to a Guardian auto-review model or to a custom PermissionRequest hook — the TUI now renders the delegation details: who reviewed the patch, what risk level was assigned, and whether the approval was automatic or escalated.[10] Previously, delegated approvals appeared identically to direct user approvals, making it difficult to audit which patches were machine-reviewed versus human-reviewed. The rendering change improves transparency in enterprise approval chains where different patches flow through different review authorities. For teams using the composite approval pattern (hooks for low-risk changes, human review for high-risk changes), the visual distinction makes the trust boundary visible at a glance.

* * *

Summary

`SessionStart` injects fresh context before the first turn. `PreToolUse` fires before each `Bash` call and can block it. `UserPromptSubmit` intercepts every prompt before the model sees it—augment or reject. `PostToolUse` observes each tool call after it completes: run tests, scan for secrets, append observations. `Stop` handles teardown: audit logs, notifications, Slack webhooks.

All five hooks share the same protocol: exit `0` with stdout for success, exit `2` with stderr to block, any other code to fail silently. Hooks run synchronously, so timeouts matter; output injected into context counts against your context window, so keep it trimmed.

[10] Delegated patch approval rendering, PR #19709, v0.126.0-alpha.8, April 2026. TUI renders delegation details (reviewer identity, risk level) for delegated patch approvals. https://github.com/openai/codex/pull/19709

`AGENTS.md` sets static rules; hooks inject dynamic state. Sandbox approval modes gate tool calls before execution; `PostToolUse` observes them after. MCP extends the agent's reach outward; hooks shape its lifecycle.

PermissionRequest hooks (PR #17563) extend the hook system from observation to active governance: shell, network, and unified execution approval prompts now fire hook events, and external scripts can return `allow`, `deny`, or `null` (fall through to TUI) verdicts. This is the first time hooks can programmatically approve or deny permission requests rather than merely logging them.

The self-evolving agent retraining loop (OpenAI Cookbook pattern) closes the feedback gap by wiring `PostToolUse` hooks to a composite grading pipeline—entity preservation, length discipline, semantic consistency, and LLM-as-judge—so that production observations feed back into fine-tuning without manual curation.

> **See also: Chapter 23 (Testing and Evaluation).** The `PostToolUse` test-on-save pattern is one half of the agent testing loop. Chapter 23 covers the other: designing a test suite that is hermetic, deterministic, and fast enough for an agent to run reliably after every file write.

Exercises

1. **Branch-aware context injection.** Write a `SessionStart` hook that reads the current git branch, extracts a Jira or GitHub issue number from the branch name (e.g., `feat/GH-1234-add-login`), and injects a one-paragraph summary by calling `gh issue view`. Test on a branch with and without an issue number.
2. **Test-on-save observer.** Implement a `PostToolUse` hook that runs `pytest -q --tb=line` after every file write and writes a summary line to stdout. Verify the summary appears in context by asking the agent "what is the current test status?" immediately after it writes a file.
3. **Full lifecycle integration.** Combine all five hooks into a single `hooks.json`. Set conservative timeouts, add `set -euo pipefail` to each bash script, and document each hook in a README inside `.codex/scripts/`. Run three sessions and review the audit log. Identify

at least one case where a hook added unexpected latency and explain how you would reduce it.

Chapter 14. The Skills Ecosystem: Using and Writing Skills

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Part 1: The Consumer's View — Using and Browsing the Ecosystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What skills are and how they work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The agentskills.io standard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

gh skill: Supply-Chain-Secure Skill Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Skills worth installing from the community

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Installing, managing, and versioning skills

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Skills vs. MCP vs. Hooks: choosing the right abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Part 2: The Producer's View — Writing Your Own Skills

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The SKILL.md format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The three-tier loading model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Writing the description field

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The agents/openai.yaml file: policy and MCP dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Building and publishing a skill

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The skill over-engineering trap: a five-tier spectrum

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Evaluating skills with Skill Creator V2

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Testing skills in CI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Testing skills with the eval pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Customisation Stack: Five Layers of Agent Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Case study: Nx monorepo agent skills

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Case study: The codex-pr-body skill — first official team-authored project skill

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Plugin System: From Skills to Distributable Packages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Plugin architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Marketplace discovery and distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Installation policies and enterprise governance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Install and remove lifecycle via app-server RPC

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Built-in Code Review skill

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Reusable workflow patterns: composing skills and plugins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cross-agent skill portability and ecosystem maturity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Scale and Automation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 15. Context Window Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Quadratic Growth Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Thread Resume and Fork: Preserving Context Without Restarting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What Consumes Context (and What Doesn't)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The /compact Command and Automatic Summarisation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Background Prefix Compaction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Sub-Agent Delegation as Context Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Strategies for Large Codebases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Monitoring Context Usage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Context Compaction Architectures: Cross-Tool Comparison

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex CLI: Server-Side Encryption and Session Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Claude Code: Transparent Summaries with Custom Instructions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

OpenCode: Selective Pruning with Protected Outputs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cross-Tool Comparison

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Model Lineage Context Compaction Breakthrough

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Plan Mode and Fresh-Context Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Context Fragments Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Prompt Caching: Economics of Long Sessions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Structured Prompting for Maximum Cache Hits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Persistent Context: The Two-Phase Memory Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP Memory Servers: Persistent Cross-Session Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Built-In Memory Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Memory Lifecycle Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Team Memory: The Gap Beyond Individual Recall

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Context Failure Modes: A Taxonomy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

1. Context Poisoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

2. Context Distraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

3. Context Confusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

4. Context Clash (the O3 finding)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The tool paradox as a context management finding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Augment's five-phase Apollo methodology

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

RTK and the four compression strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 16. Sub-Agents and Parallel Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Sub-Agent Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The TOML Subagent Definition Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Task Decomposition: What to Parallelise

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

spawn_agents_on_csv: Fan-Out Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Aggregating Results and Handling Failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Path-Based Sub-Agent Addressing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MultiAgentV2 Path Addressing and the v4 Agent API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Feedback Cascade

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Side Conversations: Lightweight Sub-Agent Alternative

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Beyond Built-In Subagents: External Swarm Orchestration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The three-tier orchestration model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Shell fan-out with worktree isolation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cross-session knowledge sharing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Named Exec Environments: Per-Agent Sandbox Isolation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exec Policy Propagation to Sub-Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cascade Thread Archive Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Multi-Agent Architecture Patterns Beyond Codex

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The composable three-layer stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cross-tool subagent landscape

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP dummy tools and subagent resilience

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Known Failure Mode: MCP Process Tree Leak

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Defensive patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Metric Freedom: When to Use Multi-Agent vs Single-Agent

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The formula

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Applying F to your tasks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The mailbox drain revert (alpha.11)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

When Not to Parallelise

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 17. Cost Management and Quota Strategy

Chapter 15 covered context windows: how they work, how they fill, and how to manage the token budget within a single session. This chapter covers the other budget: money. Context windows and cost are related (more context means more tokens means higher cost) but they're not the same problem. A session can be well within context window limits and still be unexpectedly expensive. A team can be cost-efficient while systematically exhausting context windows. Both problems require deliberate management.

This chapter teaches you how Codex CLI quota actually works, how to estimate what a team workflow will cost before you commit to it, how to configure cost controls in your deployment, and how to build monitoring so you're not surprised by an invoice.

Learning Objectives

You will be able to:

- Estimate the monthly API cost of a Codex-enabled engineering team
- Configure model routing for cost control using profile inheritance
- Build a `postTaskComplete` hook that logs session costs to a file or webhook
- Apply the cost-quality decision matrix to select the right model for a given task category
- Maximise prompt cache hit rates to reduce input token costs by 40–60%
- Use Plan Mode context-usage visibility to prevent costly context-bloat sessions
- Evaluate Amazon Bedrock as an alternative billing path for enterprise deployments

* * *

How Codex CLI quota works

Before discussing how to control cost, be precise about what cost means in the Codex CLI context, because the answer differs depending on how you access the service.

Subscription quota (ChatGPT Plus and Pro)

ChatGPT Plus and Pro subscriptions include a Codex CLI quota that is expressed in compute-equivalent units rather than raw tokens. The exact mapping varies by model: `gpt-5.4` consumes quota faster than `gpt-5.4-mini`, and reasoning models consume quota differently again because of their additional inference-time compute.[1]

The key properties of subscription quota:

- It resets monthly, not daily.
- It's not denominated in dollars; you won't see a per-token price because you're paying a flat monthly fee.
- It doesn't roll over. Unused quota at the end of the month is not credited.
- Exceeding the quota doesn't terminate sessions in progress; it throttles new sessions and eventually requires switching to API-key billing.

For individual developers doing exploratory work and personal projects, subscription quota is usually sufficient. For teams, it's usually not; teams generating significant output quickly exceed per-user monthly allocations, and per-user subscription accounts don't aggregate across team members.

Codex-only seats

OpenAI offers a codex-only seat type designed for teams that need API-level access without a full ChatGPT subscription. The pricing is $0 per seat per month plus token consumption at published API rates—no per-seat fee, no rate limits, and administrator-monitored usage via the organisation dashboard. This model suits teams that want granular cost attribution per developer without the overhead of per-user subscription management.

[1]ChatGPT Plus and Pro Codex quota limits and their relationship to model choice are documented in the OpenAI help centre. Limits are subject to change at OpenAI's discretion. https://help.openai.com/en/articles/codex-quota

Codex Subscription API

The Codex Subscription API, announced alongside GPT-5.5, provides programmatic access to subscription-tier capabilities.[2] This enables automated workflows – CI/CD pipelines, scheduled agents, and programmatic task submission – to use subscription quota rather than API-key billing. For teams that have already budgeted for ChatGPT subscriptions, the Subscription API avoids the need for a separate API billing relationship. It also provides a new billing dimension: teams can allocate subscription capacity to automated workflows and reserve API-key billing for burst or overflow usage.

Model deprecation and cost planning

OpenAI announced a deprecation schedule effective April 14, 2026, removing `gpt-5.2-codex`, `gpt-5.1-codex-mini`, `gpt-5.1-codex-max`, `gpt-5.1-codex`, `gpt-5.1`, and `gpt-5` for ChatGPT-authenticated users. If your team's cost model depends on any of these models, migrate to `gpt-5.4` or `gpt-5.4-mini` before the cutover. API-key users retain access to any API-supported model, but pricing tiers may shift as deprecated models are decommissioned.

Per-tier cost comparison

The following table estimates daily and monthly costs for different usage profiles, based on API pricing and observed usage patterns from comparable agentic coding tools:

Table 20-1a. Estimated daily cost by usage tier (API billing, April 2026)

Usage profile	Typical daily tokens	Est. daily cost	Est. monthly cost (22 days)
Light (exploration, Q&A)	50K–150K	$1–3	$22–66

[2] Codex Subscription API announced April 23, 2026. Provides programmatic access to GPT-5.5 and other models via subscription-tier capacity, enabling automated workflows to use subscription quota rather than API-key billing.

Usage profile	Typical daily tokens	Est. daily cost	Est. monthly cost (22 days)
Moderate (feature work, reviews)	300K–600K	$4–8	$88–176
Heavy (multi-agent, CI pipelines)	1M–3M	$10–25	$220–550
Team of 5 (orchestrator/worker)	5M–15M aggregate	$50–125	$1,100–2,750
Team of 50 (enterprise)	50M–150M aggregate	$500–1,250	$11,000–27,500

These estimates assume a mix of `gpt-5.4` for orchestration and `gpt-5.4-mini` for workers. Teams running all tasks on `gpt-5.5` should expect approximately 2x higher costs for the orchestration component, though token efficiency improvements may partially offset this. Teams running all tasks on `gpt-5.4` should expect 3–5× higher costs for the worker component. The 1 billion tokens per week figure sometimes cited in community discussions is a team-level aggregate—no individual developer on a standard workflow approaches that volume.

Eligible ChatGPT plans now default to the **fast service tier** (v0.124.0), which reduces latency for interactive sessions. This does not affect pricing but improves the cost-efficiency of interactive workflows where developer wait time has its own cost.

API pricing

When you use Codex CLI with an API key, billing is per-token at published rates.[3] The costs that matter for Codex CLI are:

Table 20-1. Approximate model cost ratios for Codex CLI sessions (April 2026)

[3] OpenAI API pricing for all models is published at the OpenAI pricing page. Prices shown in this chapter are approximate as of March 2026 and will change. Always verify current prices before making financial projections. https://openai.com/pricing

Model	Input (per 1M tokens)	Output (per 1M tokens)	Relative cost vs. gpt-5.4
`gpt-5.5`	$5.00	$30.00	2.0x
`gpt-5.4`	$2.00	$8.00	1.0x
`gpt-5.4-mini`	$0.40	$1.60	0.2x
`gpt-5.4-nano`	$0.10	$0.40	0.05x
`gpt-5.4-mini` (reasoning)	$1.10	$4.40	~0.6x (plus reasoning tokens)
`gpt-5.4` (high reasoning)	$10.00	$40.00	~5x

GPT-5.5 pricing at $5/$30 per million tokens represents a 2x increase over GPT-5.4.[4] The sticker price is double, but OpenAI reports better token efficiency — the model delivers better results with fewer tokens for most tasks. If GPT-5.5 solves a problem in one pass instead of two, or in 800 output tokens instead of 1,500, the effective cost per task may stay roughly the same. Teams should run representative tasks and compare total token usage against GPT-5.4 before committing to the upgrade. For routine tasks, GPT-5.4 at half the price remains the better value option.

The key observation remains that `gpt-5.4-mini` is approximately one-fifth the cost of `gpt-5.4` and covers a large fraction of agentic use cases well.[5]

What counts as usage

In an agentic session, tokens are consumed by:

- **The system prompt.** This includes the `AGENTS.md` content, the active profile configuration, and the Codex CLI system prompt itself. This is charged on every API call in the session.

[4]GPT-5.5 API pricing announced April 23, 2026: $5.00/M input tokens, $30.00/M output tokens — double GPT-5.4's $2.50/$15.00 rates. OpenAI reports better token efficiency that may offset the price increase for many tasks. https://openai.com/index/gpt-5-5/

[5]Benchmark comparisons between `gpt-5.4` and `gpt-5.4-mini` on coding tasks are discussed in the OpenAI model release documentation. For agentic coding tasks with well-defined objectives, `gpt-5.4-mini` achieves 85–90 per cent of `gpt-5.4` performance at 20 per cent of the cost. https://openai.com/gpt-4-1

- **The conversation history.** Every previous message, tool call, and tool result that's still in the context window is charged as input tokens on every API call.
- **The task description.** Your prompt.
- **Tool call results.** File contents, command output, search results: all of this comes back as input tokens on the subsequent API call.
- **Model output.** The agent's response, reasoning tokens (for o-series), and any generated content.

The expensive part of an agentic session is not the model's output; it's the accumulated conversation history. A session that reads ten files, each 500 tokens, and runs ten tool calls, adds 5,000+ tokens to the context on every subsequent API call. In a session with twenty API calls, that's 100,000 tokens of accumulated history cost on top of the actual output tokens.

This is why `/compact` has a financial benefit as well as a context management one. Compacting a session reduces the token cost of every subsequent API call by replacing a long detailed history with a shorter summary.

* * *

Estimating Team Costs

Abstract pricing tables are less useful than concrete examples. Here is a worked estimate for a realistic team deployment.

Scenario: five-engineer team, orchestrator/worker pattern

Assumptions:

- 5 engineers, each running Codex CLI for approximately 3 hours of active sessions per working day
- Orchestrator/worker pattern: each task uses one orchestrator call (on `gpt-5.4`) that spawns 2–3 worker agents (on `gpt-5.4-mini`)
- Average orchestrator session: 40,000 input tokens, 4,000 output tokens
- Average worker session: 25,000 input tokens, 3,000 output tokens

- Average of 2.5 workers per task
- 15 tasks per engineer per day (mix of short and long)

Per-day cost per engineer:

Orchestrator calls (`gpt-5.4`): 15 × (40,000 × $2.00/1M + 4,000 × $8.00/1M) = 15 × ($0.08 + $0.032) = 15 × $0.112 = **$1.68**

Worker calls (`gpt-5.4-mini`): 15 × 2.5 × (25,000 × $0.40/1M + 3,000 × $1.60/1M) = 37.5 × ($0.01 + $0.0048) = 37.5 × $0.0148 = **$0.55**

Total per engineer per day: $1.68 + $0.55 = **$2.23**

Per-month team cost (22 working days):

$2.23 × 5 engineers × 22 days = **$245/month**

This is a rough estimate (real sessions vary significantly) but it establishes order of magnitude. For a five-engineer team, Codex API costs in the orchestrator/worker pattern are roughly comparable to a single cloud service instance. This is substantially below the developer time cost of the tasks being automated.

The estimate also illustrates the leverage of the model routing decision. If the team ran all workers on `gpt-5.4` instead of `gpt-5.4-mini`, the worker cost would jump from $0.55 to approximately $2.75 per engineer per day, a 5× increase for the worker component, raising the total from $245 to about $750/month. Model routing matters.

The billing transparency crisis

As of April 2026, a significant community concern has emerged around billing transparency. A 491-comment thread on the OpenAI community forum reports users experiencing 5–10× faster token consumption than their mental models predict.[6] The root cause is the accumulated conversation history problem described above: users underestimate how quickly the context window fills with tool call results, and each subsequent API call re-processes the entire history. A session that feels like "a few questions" may have consumed 200,000+ input tokens by its tenth API call.

[6]OpenAI Community Forum, "Codex CLI token consumption significantly exceeds expectations," April 2026 (491 comments). Users report 5–10× faster quota depletion than anticipated, primarily due to accumulated conversation history in agentic sessions. https://community.openai.com/t/codex-token-consumption

The frustration has only deepened. By mid-April 2026, a 530-comment megathread on the Codex community forums crystallised the core grievance: users cannot reliably distinguish agent-initiated API requests from user-initiated ones, making accurate billing attribution nearly impossible.[7] Reports of "runaway token consumption"—sessions where background agent activity silently inflates costs—have eroded subscription confidence, particularly among teams that expected predictable monthly spend. Enterprise teams are now demanding granular per-session and per-task cost attribution, functionality that the current dashboard does not provide.

A concrete example from the community illustrates the risk for API-key users: a Reddit user ran a single prompt—"Go through this codebase and prophylactically fix any issues for MacOS"—against a 7,000-line codebase using GPT-5.4 at xHigh reasoning effort. The session consumed approximately 4 million tokens across 34 API requests in 7 minutes, costing $3.50 from a single prompt. Context was compacted mid-response, meaning the actual pre-compaction token accumulation was even higher. This is not an edge case; it is what happens when a broad, open-ended prompt meets a capable model at maximum reasoning effort on a non-trivial codebase.

This is not a billing error. It is the expected cost of agentic sessions with deep context. But the gap between user expectations and actual consumption is real, and it is exacerbated by three factors: subscription quota displays that show remaining percentage rather than token counts, the absence of per-session cost breakdowns in the ChatGPT interface, and the compounding effect of multi-agent workflows where each subagent has its own context accumulation curve.

The practical response is monitoring. Use the `postTaskComplete` hook (described later in this chapter) to log per-session token counts. Compare those counts to your projections weekly. If actual consumption consistently exceeds projections by more than 2×, your session depth assumptions need revision.

> **Note:** OpenAI has added a compaction analytics event (PR #17155) that tracks how often `/compact` is triggered and how many tokens it saves. This data may surface in the usage dashboard in future

[7] Codex Community Forums, "Token billing transparency and agent-initiated requests," April 2026 (530 comments). The thread documents widespread difficulty attributing API costs to specific agent actions versus user-initiated requests, with multiple reports of runaway consumption in background agent workflows. https://community.openai.com/t/codex-billing-transparency

releases, providing the per-session cost visibility that is currently missing.

The token burn crisis: cross-tool adoption blocker

By mid-April 2026, token economics has become the primary adoption blocker across the agentic coding landscape — ahead of capability, reliability, or security. The agents-radar #649 tracker identifies the issue as cross-tool: Codex CLI #14593 (550 comments, 225 upvotes), Claude Code #38335 (642 comments), and Copilot CLI #2591 are the most active issue threads in each project's GitHub tracker.[8] The core grievance is consistent across tools: predictable economics is now evaluated before capability when teams choose whether to adopt or expand agentic tooling.

The strategic implication is significant. A year ago, the first question enterprise evaluators asked was "can it do X?" Today, the first question is "can we predict what X will cost?" Teams that cannot model their token spend with reasonable accuracy do not scale beyond pilot phase, regardless of the agent's capabilities.

Wasted spend: the 28.5% agentic PR failure rate

MSR 2026 research across 933,000 agentic pull requests provides the first large-scale empirical evidence of wasted token spend.[9] Ehsani et al. found that 28.5% of agentic PRs fail to merge — and the dominant failure mode is not bad code but social and workflow misalignment: reviewer abandonment (38% of failures), duplicate submissions (23%), and unwanted features (4%).

Every failed PR represents tokens spent on work that never merges. The MSR data also shows that agent-introduced symbols survive a median of only 3 days compared to 34 days for human-introduced symbols, and agent code has a churn rate of 7.33% versus 4.10% for human code.[10] These metrics mean that

[8] agents-radar #649, "Token economics as primary adoption blocker," April 2026. Cross-references: Codex CLI #14593 (550 comments), Claude Code #38335 (642 comments), Copilot CLI #2591. https://github.com/agents-radar/radar/issues/649

[9] Ehsani, R. et al., "Where Do AI Coding Agents Fail? An Empirical Study of Failed Agentic Pull Requests in GitHub," MSR 2026. https://arxiv.org/abs/2601.15195

[10] Pham, D. & Ghaleb, T. A., "Code Change Characteristics and Description Alignment: A Comparative Study of Agentic versus Human Pull Requests," MSR 2026. https://arxiv.org/abs/2601.17627

even merged agentic code is revised faster, compounding the effective token cost per surviving line of code.

The practical response is to reduce the failure rate before optimising token efficiency. Chapter 20 covers CI/CD pipeline design informed by these findings. For cost management purposes, the key insight is that reducing the agentic PR failure rate from 28.5% to 15% — achievable through smaller PRs, pre-submission CI validation, and explicit scope constraints in `AGENTS.md` — would eliminate roughly half of all wasted agentic spend.

The SLM-as-judge alternative for review loops

Cross-model review — writing code with one agent and reviewing it with another — has become the standard quality pattern, but the token economics are steep. A three-provider review loop (Claude Opus + GPT-5.4 + Gemini 2.5 Pro) costs $0.47–$0.50 per merge request. At 40 merge requests per developer per month, that is $20/developer/month on review alone — $12,000 annually for a 50-person team.[11]

The February 2026 ICSE paper "Improving Code Generation via Small Language Model-as-a-judge" demonstrated that fine-tuned SLMs achieve judging performance competitive with models 5–25x larger.[12] An SLM triage layer using GPT-4o Mini ($0.15/$0.60 per M tokens) or Haiku 4.5 ($1.00/$5.00) pre-screens every merge request. Only those flagged as potentially problematic — typically 25–35% of routine MRs — proceed to a full frontier-model review. This reduces average review cost from ~$0.27 (two-model) to approximately $0.09 per MR, a roughly 3x reduction.

```
flowchart TD
    A[Code Diff] --> B[SLM Triage Judge<br/>GPT-4o Mini / Haiku 4.5]
    B -->|Score below threshold| C[Fast-Track Merge<br/>~70% of MRs]
    B -->|Score above threshold| D[Full Review<br/>Claude Sonnet / o3]
    D --> E[Human Review<br/>Critical paths only]
```

The two-model review loop (implement with one, review with another) remains the cost-quality sweet spot. The third reviewer adds cost faster

[11] Vaughan, D., "The Real Cost of Multi-Model Review Loops," codex-resources, April 2026. Cost modelling across single, two, and three-model review architectures.

[12] Crupi et al., "Improving Code Generation via Small Language Model-as-a-judge," IEEE/ACM ICSE 2026. https://arxiv.org/abs/2602.11911

than quality — research shows diminishing returns past two independent reviewers. Invest in an SLM triage layer and prompt caching (both Anthropic and OpenAI offer ~90% input savings for cached context) rather than adding a third frontier-model reviewer.

The Basic/Advanced tier split signal

Signals from OpenAI's product roadmap suggest a two-tier Codex offering is in preparation: a Basic tier covering standard coding features, and an Advanced tier gating capabilities such as Scratchpad parallel agent launcher, live app preview, and additional developer tooling.[13] Pricing details remain unconfirmed, but the pattern is clear—enterprise-grade features (multi-agent orchestration, extended sandbox environments) will likely concentrate in the Advanced tier. Teams doing budget planning for the next quarter should account for potential tier-gated features and model the cost impact of upgrading select seats rather than the entire organisation.

With over 3 million weekly active users as of early April 2026—a 50 per cent increase from the 2 million reported in March—rate-limit pressure has also increased. Users on shared subscription tiers report more frequent throttling during peak hours. API-key users are less affected but should factor higher baseline contention into their cost models.

Skill evaluation costs

Teams investing in skill development should budget for evaluation costs separately from production usage. Skill Creator V2 (Chapter 14) consumes approximately 74,000 tokens per benchmark run, running 10–20 parallel variant comparisons. A team that iterates through five benchmark cycles while tuning a skill will spend roughly 370,000 tokens—equivalent to $0.74 on `gpt-5.4` input pricing, or $0.15 on `gpt-5.4-mini`. These costs are modest per skill but add up for teams maintaining a library of 20+ skills with regular refinement cycles.

Factor skill evaluation into your monthly cost projections. A reasonable default is 5 per cent of total Codex CLI spend allocated to skill development and evaluation.

[13] References to a Basic/Advanced product split have appeared in OpenAI API changelog entries and community manager responses as of April 2026. No official pricing has been published. Features mentioned for the Advanced tier include Scratchpad parallel agent launcher and live app preview.

Projection variables to adjust

This estimate is a starting point. The variables that move it most significantly:

- **Session depth.** The 40,000 input token assumption for orchestrators is moderate. Sessions in a large legacy codebase with many file reads can reach 100,000+ input tokens. Measure your actual sessions with `--debug` output and adjust.
- **Workers per task.** More parallelism is more cost. Three workers costs 20 per cent more than 2.5; five workers costs double.
- **Task frequency.** If engineers run Codex CLI for discovery and exploration (many short sessions) rather than deep task completion (fewer long sessions), the cost profile shifts. Exploration sessions are short but generate overhead per session that's proportionally higher.

* * *

Configuring Cost Ceilings

Projections are useful, but hard limits are better. Codex CLI provides several mechanisms for setting per-session and per-profile cost ceilings.

Per-session token limits

The `max_tokens_per_session` configuration key sets a hard limit on the total tokens (input plus output) that can be consumed in a single session. When the limit is reached, the session is terminated gracefully: the agent completes the current tool call and then exits with a message explaining the limit was reached.

In `~/.codex/config.toml`:

```
1 max_tokens_per_session = 200000
```

This is a blunt instrument: it does not distinguish between an efficient session and one wasting tokens on repeated reads. But it prevents runaway sessions from generating unexpectedly large bills.

`max_tokens_per_session` counts tokens at the API level, not at the context window level. A session that uses `/compact` to reduce the context window still accumulates total tokens across all API calls before and after compaction. The ceiling is cumulative, not a snapshot.

Model routing for cost control

A more effective approach routes different task types to different models based on their cost-quality requirements. Codex CLI supports this through named profiles selected per invocation:

```
[profiles.explore]
model              = "gpt-5.4-mini"
max_tokens_per_session = 100000

[profiles.commit]
model              = "gpt-5.4"
max_tokens_per_session = 300000

[profiles.reason]
model              = "gpt-5.4-mini"
reasoning_effort   = "high"
max_tokens_per_session = 150000

default_profile    = "explore"
```

Invoke a specific profile with:

```
codex --profile commit "implement the new user authentication flow"
```

The `explore` profile is the default: it uses the cheaper model and has a lower token ceiling, appropriate for exploratory sessions where you're browsing code, asking questions, or running quick experiments. The `commit` profile uses the full-capability model for tasks that will produce real output. The `reason` profile uses `gpt-5.4-mini` with `high` reasoning effort for tasks that benefit from extended reasoning.

Profile inheritance

Profiles support inheritance to avoid repeating common settings:

```
[profiles.base]
sandbox_mode    = "workspace-write"
approval_policy = "on-request"

[profiles.explore]
inherits = "base"
model    = "gpt-5.4-mini"
max_tokens_per_session = 100000

[profiles.commit]
inherits = "base"
model    = "gpt-5.4"
max_tokens_per_session = 300000
```

The `explore` and `commit` profiles inherit the sandbox and approval settings from `base` and only specify what's different. This keeps the config DRY and makes it easier to update shared settings across profiles.

* * *

Monitoring and Alerting with Hooks

Configuration controls what Codex CLI does before and during a session. Hooks let you react to what happened after. The `postTaskComplete` hook fires when a session completes, whether successfully or not, and receives structured session data including token counts, model used, duration, and the final task status.

Logging to a file

A `postTaskComplete` hook can append session cost data—model, input tokens, output tokens, session ID—to a local JSONL file for later analysis. The `CODEX_HOOK_DATA` environment variable provides session metadata as JSON. After a few days of sessions, aggregate the log with `jq` to compute daily and weekly cost estimates.

See Chapter 13 for the complete hook implementation.

Logging to a webhook

For teams, individual log files are insufficient; you need centralised data. A `postTaskComplete` hook that posts `CODEX_HOOK_DATA` to a webhook URL (set via environment variable) lets your backend aggregate by user, project, team, and model.

See Chapter 13 for the complete hook implementation.

Alerting on threshold exceedance

A `postTaskComplete` hook can send a Slack notification when a single session exceeds a token threshold (e.g., 500,000 tokens—roughly $1.50–$2.00 per session at current rates). This catches runaway sessions before they accumulate significant cost.

See Chapter 13 for the complete hook implementation.

* * *

Cost-Quality Decision Matrix

Not all tasks require the best model. Many agentic engineering tasks are well-served by cheaper models, and routing them correctly cuts costs without sacrificing quality.

The decision framework

Table 20-2. Task categories and model recommendations

Task category	Recommended model	Reasoning
Code search and navigation	`gpt-5.4-nano` or `gpt-5.4-mini`	Finding things requires reading and pattern-matching, not synthesis
Formatting and linting passes	`gpt-5.4-nano`	Deterministic transformations; quality difference is negligible
Test generation for known patterns	`gpt-5.4-mini`	Good models for well-established idioms
Documentation generation	`gpt-5.4-mini`	Fluency matters more than deep reasoning
Exploratory refactoring	`gpt-5.4-mini`	Draft quality is acceptable; engineer reviews
Complex multi-file refactoring	`gpt-5.4`	Coherence across many files requires full capability
Architecture design and review	`gpt-5.4` with `high` reasoning effort	High-stakes reasoning; cost is justified
Security audit	`gpt-5.4-mini` with `xhigh` reasoning effort	Adversarial reasoning benefits from extended inference
Novel algorithm implementation	`gpt-5.4-mini` with `xhigh` reasoning effort	Hard problems benefit from the "reasoning tax"
Bug diagnosis in large codebases	`gpt-5.4`	Context coherence matters; mini may lose the thread

The principle: use cheaper models for tasks with strong patterns (the model is filling in a template) and more capable models for tasks with weak patterns

(the model is solving a novel problem).

The "reasoning tax" on high-effort configurations

Models configured with `xhigh` or `high` reasoning effort apply extended inference at generation time: they "think" before producing output, generating reasoning tokens billed as output tokens but not shown to the user. This reasoning tax makes them cost more per call than the same model at `medium` effort, but for problems where that reasoning is necessary (logic errors, complex state machines, multi-constraint optimisation) the quality improvement justifies the cost.

The reasoning tax is not worth paying for routine tasks. Running `gpt-5.4` at `xhigh` effort to generate a standard REST endpoint is expensive and doesn't produce better results than `gpt-5.4` at `medium` effort on that task. The routing decision should depend on whether the task actually benefits from extended reasoning, not on which model is "best" in benchmarks.

A quick heuristic: if you could describe the correct output from the task in a precise specification before the agent runs, the task probably doesn't need high reasoning effort. If the agent needs to explore solution space and evaluate trade-offs, the reasoning tax earns its cost.

* * *

Per-Reasoning-Effort Token Consumption

Empirical data from the token usage article (April 2026) quantifies the cost impact of reasoning effort settings with concrete token consumption figures. The data confirms what practitioners have suspected: the cost multipliers for higher reasoning effort are steeper than most teams budget for, and the quality returns are non-linear.

Table 20-3. Observed token consumption by reasoning effort level (gpt-5.4, representative coding task)

Effort level	Input tokens	Output tokens	Reasoning tokens	Total tokens	Relative cost vs medium
`minimal`	~12,000	~800	~200	~13,000	~0.15x
`low`	~15,000	~1,200	~1,500	~17,700	~0.35x
`medium`	~20,000	~2,500	~8,000	~30,500	1.0x (baseline)
`high`	~25,000	~4,000	~35,000	~64,000	~3.5x
`xhigh`	~30,000	~6,000	~120,000	~156,000	~12x

The reasoning token column is the cost driver. At `xhigh`, reasoning tokens account for over 75% of total consumption. These tokens are billed as output tokens at output token rates—meaning the effective per-session cost of `xhigh` is dominated by invisible computation the user never sees in the response.

KV-cache economics: cache miss = 10x cost

WEKA's infrastructure research provides a critical insight for enterprise cost planning: a KV-cache miss—where the model must recompute cached key-value pairs rather than reusing them from a previous turn—costs approximately 10x more than a cache hit. In practice, this means that session interruptions (network disconnects, process restarts, `/compact` operations) carry a hidden cost multiplier. A session that runs continuously benefits from cache hits on accumulated context; a session that restarts pays the full recomputation cost.

The practical implication: minimise session restarts for long-running tasks. If you must restart, use `codex resume --last` to preserve as much cached context as possible rather than starting a fresh session.

Stanford bimodal ROI distribution

Stanford's 120,000-developer study (reviewed across 22 conference talks in April 2026) provides a sobering correction to vendor ROI claims. The distribution of productivity gains from AI coding assistants is bimodal, not normal: the median gain is 10–15%, not the 30–60% figures commonly cited in vendor marketing. The top quartile of developers sees gains of 40–60%, while the bottom quartile sees negligible or even negative productivity impact.

This bimodal distribution has direct cost implications: teams that budget for a 40% productivity gain across all developers and price their AI tooling investment against that assumption will be disappointed. Budget against the median (10–15% gain) and treat higher returns as upside.

The tool paradox: fewer tools, better accuracy

A counterintuitive finding from the 22-talk review: reducing the number of available tools from 46 to 19 relevant tools improved agent accuracy by 44%. This is the "tool paradox"—more tools create more context overhead and more decision-making complexity for the model. Every tool description consumes context window tokens (see Chapter 15 on context window management) and every additional tool increases the probability that the model will select the wrong one.

The cost relevance is direct: each unnecessary MCP server connected to your session is not just a security surface—it is a cost multiplier. The tool descriptions consume tokens on every API call, and the increased decision complexity leads to longer reasoning chains and more frequent tool-use errors.

* * *

Prompt Caching: the Largest Single Cost Lever

The previous sections focused on choosing the right model and the right reasoning effort. Prompt caching is a complementary lever that reduces the cost of whichever model you choose. OpenAI's prompt caching mechanism is automatic for any API request whose prefix exceeds 1,024 tokens. Cached input tokens are billed at a 50% discount – no opt-in required, no additional fee for cache writes.[14]

In a typical Codex CLI session, the system prompt, tool definitions, AGENTS.md content, and early conversation turns form a stable prefix that is resent on every API call. With an 85% cache hit rate – achievable with default

[14]OpenAI, "Prompt Caching," API documentation. Cached input tokens are billed at 50% of the standard input rate. The mechanism is automatic for requests exceeding 1,024 tokens and uses exact prefix matching on 128-token boundaries. https://developers.openai.com/api/docs/guides/prompt-caching

settings and stable configuration — the effective cost of input tokens drops by 40–60% over the life of the session.[15]

The savings compound with model routing. A team that routes exploration tasks to `gpt-5.4-mini` (0.2× the cost of `gpt-5.4`) and then achieves an 85% cache hit rate on those sessions is paying roughly 0.2 × 0.55 = 0.11× the baseline `gpt-5.4` uncached input cost — a 92% reduction from the naive starting point. Neither lever alone achieves that; the combination does.

Four practices to maximise cache hits

1. **Keep system prompts stable.** Dynamic content such as timestamps, build numbers, or session identifiers in the system prompt invalidates the prefix on every session. Move volatile data into the user message instead.
2. **Place AGENTS.md at the start and keep it deterministic.** AGENTS.md content is aggregated into the developer-role prefix. If these files include auto-generated metadata (line counts, last-modified dates), every invocation produces a different prefix. Pin AGENTS.md discovery to specific paths and avoid dynamically generated content.
3. **Maintain consistent tool ordering.** MCP server discovery order must be deterministic. Adding, removing, or reordering MCP servers mid-session changes the prefix and invalidates cached state. Declare all servers upfront in `config.toml` and prefer disabling over removing.
4. **Batch similar tasks to share warmed caches.** Parallel sessions against the same repository share the same system instructions and tool definitions. Setting a consistent `prompt_cache_key` environment variable across related sessions increases the probability that they hit the same cached prefix:

```
CODEX_PROMPT_CACHE_KEY="myproject-main" codex "fix the auth bug"
CODEX_PROMPT_CACHE_KEY="myproject-main" codex "add unit tests for auth"
```

For a detailed treatment of the caching mechanism, prefix layout, anti-patterns, and monitoring, see the companion article “Prompt Caching in Codex CLI: How the Agent Loop Stays Linear and How to Maximise Cache

[15] OpenAI, “Prompt Caching,” API documentation. Cached input tokens are billed at 50% of the standard input rate. The mechanism is automatic for requests exceeding 1,024 tokens and uses exact prefix matching on 128-token boundaries. https://developers.openai.com/api/docs/guides/prompt-caching

Hits" (`articles/2026-04-21-codex-cli-prompt-caching-maximise-cache-hits-cost-reduction.md`).

* * *

Context-Usage Visibility from Plan Mode

Codex CLI v0.122.0 introduced Plan Mode, which separates planning from implementation. When you run a task in Plan Mode, the agent produces a structured plan and then pauses before executing it. The plan output includes a context-usage indicator showing how much of the context window the planning phase consumed.

This is a cost management tool as much as a workflow tool. If the plan alone consumes 60% of the context window, the implementation phase will start with limited headroom, leading to early compaction (which resets the prompt cache) or context overflow (which terminates the session). Either outcome wastes tokens.

The practical workflow is:

1. Run the task in Plan Mode.
2. Check the context-usage indicator in the plan output.
3. If context usage is above 50%, consider splitting the task into smaller subtasks before approving implementation.
4. If context usage is below 30%, the task has comfortable headroom and can proceed as-is.

This avoids the most expensive failure mode in agentic sessions: a task that runs deep into the context window, triggers compaction, loses cache benefits, and then produces incomplete output that requires a second session. Catching context bloat at the plan stage — before any implementation tokens are spent — is strictly cheaper than discovering it after ten tool calls.

* * *

Bedrock Pricing as an Alternative to Direct API

For teams already operating on AWS, Amazon Bedrock provides access to OpenAI models (including `gpt-5.4` and `gpt-5.4-mini`) through the Bedrock provider rather than through OpenAI's API directly. The token-level pricing is comparable, but the billing model differs in ways that matter for enterprise cost management.

Bedrock enables:

- **Committed-use discounts.** AWS offers provisioned throughput pricing with committed spend agreements, which can reduce effective per-token cost below published on-demand rates for teams with predictable volume.
- **AWS billing consolidation.** Codex CLI API costs appear on the same AWS invoice as compute, storage, and other infrastructure, simplifying cost attribution and avoiding a separate OpenAI billing relationship.
- **Enterprise pricing tiers.** Organisations with AWS Enterprise Discount Programs (EDPs) may receive additional discounts on Bedrock inference that are not available through direct OpenAI API billing.

To configure Codex CLI to use the Bedrock provider, set the provider in your configuration:

```
[model]
provider = "bedrock"
model    = "gpt-5.4"
```

The trade-off is operational: Bedrock introduces an additional layer between Codex CLI and the model, which may add latency and may lag behind OpenAI's direct API in supporting new model releases or features. Teams should verify that prompt caching behaviour is preserved through the Bedrock provider, as cache routing depends on infrastructure that may differ between providers.

For teams spending above $5,000/month on Codex CLI API costs, the committed-use discount alone can offset the operational overhead. Below that threshold, direct API billing is simpler and provides faster access to new features.

Token Compression: RTK and the Cost Impact

RTK (Rust Token Killer), with 36.4K GitHub stars as of April 2026, is the most widely adopted community tool for reducing agent token consumption.[16] Operating as a transparent proxy between Codex CLI and the model API, RTK achieves 60–90 per cent token reduction through structural deduplication, semantic compression, diff-aware encoding, and schema compaction (see Chapter 15 for the technical detail).

The cost implications are direct: a session that consumes 800K tokens at $5 per million input tokens ($4.00) reduces to 160K–320K tokens ($0.80–$1.60) – a 60–80 per cent cost saving per session. At team scale, the savings compound. A five-engineer team running 20 sessions per day moves from approximately $80/day to $16–$32/day. Over a month, that is the difference between $2,400 and $480–$960.

RTK integrates with Codex CLI by setting it as a model provider proxy in `config.toml`. No changes to skills, hooks, or AGENTS.md are required:

```
[model_providers.rtk]
endpoint = "http://localhost:8080/v1"
wire_api = "responses"
```

The proactive usage prompt (PR #19769, v0.126.0 alpha) provides a complementary signal: when Codex detects that a session is consuming tokens faster than expected, it logs a structured warning that surfaces the consumption rate.[17] For cost-conscious teams, this early warning can trigger a session split or model downgrade before the budget ceiling is hit.

* * *

Summary

- Codex CLI subscription quota is expressed in compute-equivalent units and resets monthly; API billing is per-token at published rates. The two

[16] RTK (Rust Token Killer), https://github.com/nicholasgriffintn/rtk, accessed April 2026. Token compression proxy achieving 60–90% context reduction. 36.4K GitHub stars.

[17] Proactive usage prompt, PR #19769, v0.126.0 alpha cycle, April 2026. Surfaces structured warnings when token consumption exceeds expected rates. https://github.com/openai/codex/pull/19769

are different products with different cost profiles.

- In an API-billed agentic session, the most significant cost driver is the accumulated conversation history, not the model output. This makes context management (including `/compact`) a financial decision as well as a quality one.
- A five-engineer team running Codex CLI in an orchestrator/worker pattern can expect roughly $200–300/month in API costs, dominated by orchestrator calls on `gpt-5.4`. Workers on `gpt-5.4-mini` are approximately one-fifth the cost per token.
- Model routing via named profiles with inheritance is the most effective cost control mechanism. Route exploratory tasks to `gpt-5.4-mini` or `gpt-5.4-nano`; route complex multi-file work and security-sensitive tasks to `gpt-5.4` with high reasoning effort.
- The `postTaskComplete` hook is the primary mechanism for cost monitoring. Log to a local file for individual tracking; log to a webhook for team aggregation.
- A billing transparency gap has created a community concern: users report 5–10x faster token consumption than expected, driven by accumulated conversation history and the inability to distinguish agent-initiated from user-initiated requests. This has escalated into a cross-tool token burn crisis (agents-radar #649): predictable economics is now evaluated before capability as the primary adoption criterion. MSR 2026 research shows that 28.5% of agentic PRs fail to merge, meaning nearly a third of all agentic token spend produces work that never ships.
- The SLM-as-judge alternative reduces review loop costs by roughly 3x: an SLM triage layer (GPT-4o Mini or Haiku 4.5) pre-screens merge requests, routing only flagged ones to frontier-model review. Two-model review remains the cost-quality sweet spot; adding a third reviewer adds cost faster than quality.
- Prompt caching provides a 50% discount on cached input tokens, translating to 40–60% overall input cost savings at typical cache hit rates. Four practices maximise cache hits: stable system prompts, deterministic AGENTS.md, consistent tool ordering, and batching similar tasks with a shared `prompt_cache_key`. Combined with model routing, prompt caching can reduce input costs by up to 92% versus the naive baseline.
- Plan Mode (v0.122.0) exposes context-usage visibility before implementation begins, letting you catch context-bloat tasks at the planning stage and split them before wasting tokens on sessions that will hit compaction or overflow.

- Amazon Bedrock provides an alternative billing path for teams on AWS, enabling committed-use discounts, enterprise pricing tiers, and AWS billing consolidation. Teams spending above $5,000/month on API costs should evaluate Bedrock for the committed-use discount alone.
- Monitor per-session token counts with hooks and compare to projections weekly. Budget planning should also account for a potential Basic/Advanced tier split that may gate enterprise features.
- Skill evaluation costs (approximately 74,000 tokens per Skill Creator V2 benchmark run) should be budgeted separately—a reasonable default is 5 per cent of total Codex CLI spend.
- With 3 million weekly active users as of April 2026 (up 50 per cent from March), rate-limit pressure has increased. Factor higher baseline contention into cost models.

Exercises

Exercise 20-1. Using the worked example in the Estimating Team Costs section as a template, build a cost estimate for your own team's Codex usage over the past week. If you don't have actual usage data, use the `--debug` output from five representative sessions to measure average token counts per session type. Then compare your estimate to the actual cost on your OpenAI usage dashboard. Note which assumptions in the template were most different from your reality.

Exercise 20-2. Configure three profiles in your `~/.codex/config.toml`: one for exploration (`gpt-5.4-mini`, low token ceiling), one for commits (`gpt-5.4`, higher ceiling), and one for reasoning tasks (`gpt-5.4-mini` with `high` effort, moderate ceiling). Add a `postTaskComplete` hook that appends session cost data to a local log file. Run at least ten sessions over two days using the appropriate profile for each task. At the end, analyse the log and calculate what the total cost would have been if you'd used the commit profile for all sessions instead of routing by task type.

Exercise 20-3: Build a team cost attribution system. If your team shares an API key or uses Codex Cloud, design a cost attribution system. Create separate API keys (or cost tags via the OpenAI API metadata field) for at least three different use cases (interactive development, CI pipeline, automated review). Write a short script or shell alias that sets the correct key/tag based on context.

After one week, pull the usage breakdown from the OpenAI usage dashboard and verify that the attribution is working correctly. Calculate the per-use-case cost and identify which case is most expensive per token and which is most expensive in total.

Exercise 20-4: Audit your most expensive session. From your usage dashboard or local cost log, identify the most expensive Codex session you have run this month. Open the `transcript.jsonl` for that session (in `~/.codex/sessions/`) and analyse the token breakdown: how much was system prompt, how much was conversation history at the end, and how much was tool call results? Identify the single largest contributor and design one change to your workflow (prompt, context management, or configuration) that would reduce cost for similar sessions by at least 20%.

Chapter 18. Multi-Agent Orchestration Patterns

Chapter 15 showed you how to reason about context windows: what fits inside a single agent's working memory, how to compress history, and when a task exceeds what one agent can hold. That constraint is the natural entry point into orchestration: if one agent cannot hold everything, you need multiple agents and rules for how they communicate. Context size tells you *what* each agent can do; orchestration patterns tell you *how* to wire them together so their combined output is coherent. This chapter covers the four patterns that appear most often in production Codex CLI pipelines (Sequential Gated Chain, Parallel Worker Swarm, Wave-Based Hybrid, and Cross-Model Review Loop) and the practical guidance you need to choose among them, debug them when they fail, and avoid the anti-patterns that quietly corrupt results.

Learning Objectives

You will be able to:

- Select the appropriate orchestration pattern for a given task based on its dependency structure and latency requirements
- Implement a gated handoff between two agents using structured output validation
- Diagnose silent worker failures in a parallel swarm and add the instrumentation to surface them
- Configure a cross-model review loop at the appropriate automation tier and use the feedback cascade primitive to drive convergence
- Use `/side` and `/fork` conversation branching to implement the supervisor pattern for lightweight multi-agent coordination
- Evaluate when conversation branching is sufficient versus when programmatic orchestration (swarms, waves) is required
- Describe how Goal Mode's persistent objectives and token budgets change the orchestration outer loop

* * *

The Three-Tier Orchestration Landscape

Before diving into individual patterns, it helps to understand the three tiers at which multi-agent orchestration operates in the Codex CLI ecosystem:

Tier 1: In-process orchestration. The four patterns in this chapter (Sequential Gated Chain, Parallel Worker Swarm, Wave-Based Hybrid, Cross-Model Review Loop) all operate within a single Codex CLI session or a set of coordinated sessions on a single machine. The parent agent controls the lifecycle, and all coordination happens through Codex's built-in subagent primitives. This is the right tier for tasks that fit within one developer's workflow.

Tier 2: Local shell fan-out. When tasks exceed the six-thread subagent limit, external orchestrators (shell scripts, GNU Parallel, custom Python harnesses) launch multiple independent Codex CLI processes, each in its own git worktree. Chapter 16 covers this tier in detail. The coordination primitive shifts from Codex's internal subagent system to filesystem and process management.

Tier 3: Cloud fan-out. Codex Cloud Tasks run each unit of work as a cloud-hosted session with its own compute, context window, and rate-limit allocation. This tier is appropriate for overnight batch processing, CI/CD-triggered analysis, and workloads that exceed local machine capacity.

The patterns in this chapter focus on Tier 1 but are composable upward. A Wave-Based Hybrid can use shell fan-out for wide waves. A Sequential Gated Chain can delegate individual stages to Cloud Tasks. Understanding which tier you're operating at helps you choose the right coordination primitives and set realistic expectations about latency, cost, and failure handling.

Emerging patterns

The orchestration landscape is evolving rapidly. Google's Agent-to-Agent (A2A) protocol (Chapter 5, Chapter 12) defines how agents from different vendors discover and delegate to each other, pointing toward cross-tool orchestration without MCP-server bridging. Persistent orchestration frameworks decouple the orchestrator's lifecycle from any individual task, enabling always-on scheduling and container-isolated worker pools for proactive workflows

like nightly code quality sweeps. Codex CLI's own conversation branching primitives (`/side` and `/fork`, v0.122.0) and the upcoming Goal Mode (v0.123) represent a parallel evolution: moving orchestration inside the CLI session itself, with the developer or a parent agent acting as supervisor. These developments extend the patterns in this chapter beyond discrete task graphs, but the newer primitives (particularly Goal Mode) have not yet reached production maturity as of early 2026.

Pattern 1: Sequential Gated Chain

The simplest orchestration topology is a chain: Agent A produces output, that output becomes the input to Agent B, Agent B's output feeds Agent C, and so on. Each agent in the chain has a single upstream producer and a single downstream consumer. The chain terminates when the last agent delivers a final artefact.

Simple does not mean safe.[1] A naive chain assumes every agent always produces valid output. In practice, agents fail silently. A summariser returns an empty string when the source document is binary. A code reviewer returns a markdown block when the next agent expects JSON. A classifier returns a label outside the allowed vocabulary. If you pass these outputs downstream without checking, the failure propagates and amplifies; the final artefact is wrong, and the error is nearly impossible to trace back to its origin without re-running the entire chain.

The *gated* variant addresses this by inserting a validation step between every pair of agents. The gate is not a new LLM call; it is a deterministic function that asserts structural and semantic invariants on the upstream output before allowing it to flow downstream. A gate might check that a JSON blob matches a Pydantic schema, that a word count falls within a range, or that required keys are present and non-null. If the gate fails, it either retries the upstream agent with augmented instructions or raises an exception that halts the chain with a clear error message.

[1]The ReAct framework (Yao et al., 2022) introduced the paradigm of interleaving reasoning traces and actions in LLM agents, and showed that multi-step pipelines require explicit reasoning at each stage to avoid silent propagation of errors. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," arXiv:2210.03629, ICLR 2023. https://arxiv.org/abs/2210.03629

```python
# Minimal gated chain: two agents with a schema gate between them
from codex import agent, gate
from pydantic import BaseModel

class ResearchSummary(BaseModel):
    topic: str
    key_findings: list[str]   # must have at least one item
    confidence: float          # 0.0–1.0

def run_research_chain(query: str) -> str:
    # Stage 1: research agent produces structured summary
    raw = agent("researcher").run(
        f"Research this topic and return JSON: {query}"
    )

    # Gate: validate structure before passing downstream
    summary = gate(raw, schema=ResearchSummary, retries=2)

    # Stage 2: writer agent consumes validated summary
    report = agent("writer").run(
        f"Write a 500-word report from this
        ↪  summary:\n{summary.model_dump_json()}"
    )
    return report
```

Notice that `gate()` takes the raw output and the expected schema. If parsing fails on the first attempt, Codex CLI re-invokes the researcher with the validation error appended to the prompt, giving it a chance to self-correct. After `retries` attempts the gate raises `GateValidationError`, which you catch at the orchestration layer.

Sequential chains are the right choice when your task is inherently linear: when step N genuinely cannot start until step N-1 is complete and its output has been validated. Document translation pipelines (extract, translate, reformat) and multi-stage code generation pipelines (spec, scaffold, implement, test) are canonical examples. The tradeoff is latency: stages execute serially, so total wall-clock time is the sum of all stage latencies rather than the maximum.

Pattern 2: Parallel Worker Swarm

When your task decomposes into many independent units of work—rows in a CSV, files in a repository, tickets in a backlog—you can fan out across a swarm of worker agents that execute in parallel. The orchestrator spawns workers, each

worker handles one unit, and the orchestrator collects and aggregates results when all workers complete.

Codex CLI exposes this pattern through `spawn_agents_on_csv`, a built-in helper that reads a CSV file, spins up one agent per row, passes each row's fields as prompt variables, and streams results back to a collector:[2]

```
from codex import spawn_agents_on_csv, collect_results

# tickets.csv has columns: ticket_id, title, description, priority
results = spawn_agents_on_csv(
    csv_path="tickets.csv",
    agent_name="ticket-classifier",
    prompt_template=(
        "Classify ticket {ticket_id}: '{title}'\n"
        "Description: {description}\n"
        "Current priority: {priority}\n"
        "Return JSON: {{'category': str, 'suggested_priority': str,
        ↪  'rationale': str}}"
    ),
    max_concurrency=8,      # cap simultaneous workers
    timeout_per_worker=30,  # seconds before a worker is considered failed
)

summary = collect_results(results, on_failure="log_and_skip")
```

The `max_concurrency` parameter is essential. Without it you run unbounded parallelism (discussed in the anti-patterns section below), which exhausts API rate limits, saturates your event loop, and produces a thundering-herd failure when the rate limiter begins dropping requests. Set `max_concurrency` to a value derived from your API tier's requests-per-minute limit divided by an estimate of average task duration.

The aggregation plan is the other half of the pattern, and engineers routinely neglect it. Before you fan out, decide: what does the collected output look like, and who produces it? In the ticket classifier example, `collect_results` returns a list of parsed result objects. A second pass (often a simple Python function rather than another agent) rolls these into a summary CSV or a Jira bulk update payload. If you skip this step and treat the raw list as your final artefact, you have fan-out without aggregation, which is a different anti-pattern described later.

[2] Codex CLI subagents documentation describing the `spawn_agents_on_csv` fan-out primitive and result collection. https://developers.openai.com/codex/subagents

The swarm pattern trades simplicity for throughput. Its failure mode is partial completion: some workers succeed, some fail, and you need a strategy for each category of failure. The `on_failure` parameter accepts `"log_and_skip"` (record the failure and continue), `"retry_once"` (retry failed workers exactly once before skipping), or `"raise"` (halt on first worker failure). Choose based on whether partial results are acceptable. For a batch analytics job, `"log_and_skip"` is usually fine. For a code-generation pipeline where every file must be produced, `"raise"` or `"retry_once"` is more appropriate.

Pattern 3: Wave-Based Hybrid

Many real-world tasks are neither purely sequential nor purely parallel. They have a dependency graph: some units of work depend on others, but within each dependency tier the work is independent. The Wave-Based Hybrid pattern maps directly onto this structure by grouping work into waves, where all items in a wave are independent of each other (and therefore parallelisable) but each wave depends on the completion of the previous one.

Consider a monorepo refactor: you cannot update callers of a function until you have updated the function's signature, but you can update all callers in parallel once the signature is settled.[3] The dependency graph has two tiers. Wave 1 contains the one task that has no dependencies (update the function signature). Wave 2 contains all the caller-update tasks, which depend on Wave 1's output.

```
from codex import wave_executor, WavePlan

plan = WavePlan([
    # Wave 1: no dependencies
    {
        "id": "update-signature",
        "agent": "refactor-agent",
        "prompt": "Update function signature in lib/core.py per spec",
        "depends_on": [],
    },
    # Wave 2: depends on wave 1
    {
        "id": "update-caller-api",
```

[3] Wave-based scheduling is a natural extension of the ReAct agent architecture to multi-agent settings, where dependency graphs between tasks determine execution order. https://arxiv.org/abs/2210.03629

```
        "agent": "refactor-agent",
        "prompt": "Update callers in api/ using updated signature from
        ↪  {update-signature.output}",
        "depends_on": ["update-signature"],
    },
    {
        "id": "update-caller-cli",
        "agent": "refactor-agent",
        "prompt": "Update callers in cli/ using updated signature from
        ↪  {update-signature.output}",
        "depends_on": ["update-signature"],
    },
    {
        "id": "update-caller-tests",
        "agent": "refactor-agent",
        "prompt": "Update test stubs using updated signature from
        ↪  {update-signature.output}",
        "depends_on": ["update-signature"],
    },
])

executor = wave_executor(plan, max_concurrency_per_wave=4)
final_results = executor.run()
```

The `wave_executor` resolves the dependency graph at startup, partitions tasks into waves, runs each wave as a mini-swarm, validates that all tasks in a wave completed successfully before proceeding, and passes upstream outputs as template variables to downstream prompts via the `{task-id.output}` interpolation syntax.

Wave boundaries are explicit quality gates. If any task in Wave N fails, the executor halts before starting Wave N+1. This is the correct default: proceeding with downstream tasks when upstream tasks have failed produces cascading corrupted artefacts that are expensive to clean up. You can override this with `on_wave_failure="skip_dependents"` to allow waves whose dependency chain is intact to proceed, but this requires careful analysis of your dependency graph to avoid surprising interactions.

The wave-based pattern is the most powerful of the three and also the most complex to reason about. Its primary failure mode is dependency misconfiguration: you declare that task B depends on task A, but you forget that task C also depends on A's output, so C runs in Wave 1 with stale data. Always draw your dependency graph explicitly before encoding it in a `WavePlan`.

Choosing the Right Pattern

The decision between the four patterns reduces to three questions: are the units of work independent of each other, do some units depend on outputs produced by others, and does the task benefit from structurally independent review?

	Work is independent	**Work has dependencies**
Single unit per stage	Sequential Gated Chain	Sequential Gated Chain
Many units	Parallel Worker Swarm	Wave-Based Hybrid
Quality-critical output	Cross-Model Review Loop	Cross-Model Review Loop (embedded in chain or wave)

If you have a single linear pipeline where each stage produces one artefact consumed by the next, use a Sequential Gated Chain. If you have many independent units of the same type to process in bulk, use a Parallel Worker Swarm. If you have many units with inter-dependencies, use a Wave-Based Hybrid. If the output is high-stakes and benefits from structurally independent judgement—security-sensitive code, public API changes, data migrations—layer a Cross-Model Review Loop on top of whichever base pattern you choose.

Table 21-1 gives a more detailed comparison across the dimensions that matter most in production.

Table 21-1. Orchestration pattern comparison

Dimension	Sequential Gated Chain	Parallel Worker Swarm	Wave-Based Hybrid	Cross-Model Review Loop
Canonical use case	Multi-stage document pipeline	Bulk classification, batch analysis	Monorepo refactor, dependent deployments	Security-sensitive code review, adversarial quality gate
Implementation complexity	Low	Medium	High	Low (Level 1) to High (Level 3)
Latency	Sum of stage latencies	Max worker latency	Sum of wave latencies	+30–90 s per review round
Primary failure mode	Gate validation error propagates	Partial completion with silent failures	Dependency misconfiguration	Infinite feedback loop without re-entrancy guard
Recovery strategy	Retry upstream agent with error context	Retry failed workers; skip or halt	Halt at wave boundary; fix failed tasks before continuing	Cap iteration rounds; escalate to human on non-convergence
Best for teams with	< 5 agents in pipeline	Homogeneous worker tasks	Heterogeneous tasks with known dependency graph	High-stakes changes requiring independent judgement

When your workload doesn't fit cleanly into one pattern, compose them.

A wave-based executor can use a gated chain as the agent logic inside each wave task. A sequential chain can call a worker swarm as one of its stages. The patterns are composable primitives, not mutually exclusive categories.

Pattern 4: Cross-Model Review Loop

The three patterns above assume all agents share the same model. In practice, the most consequential quality gains come from routing different *models* into producer and reviewer roles. A model reviewing its own output suffers from sycophancy bias: the same reasoning shortcuts that produced a bug will rationalise it during review.[4] Cross-model review sidesteps this by placing a structurally different model in the critic seat—different training data, different blind spots, genuinely independent judgement. When both models flag the same issue, confidence is high. When only one flags it, the disagreement itself is the signal worth investigating.

This pattern has three concrete automation tiers, each trading setup complexity for hands-off operation.

Level 1: SKILL.md — Manual Trigger

A `SKILL.md` file placed in `.claude/skills/` defines a slash command that any agent can parse. The command exports the current diff, invokes `codex exec` in read-only sandbox mode for an independent review, and feeds findings back for the builder to address:[5]

```
# Review invocation (read-only sandbox enforces immutability)
codex exec -m gpt-5.3-codex -s read-only \
  "Review this plan against the codebase. Respond PASS or CONCERNS with
  ↪  details."
```

If the verdict is `CONCERNS`, the builder addresses the findings and resubmits. The loop caps at five rounds before escalating to a human. A worthwhile

[4] Cross-model adversarial review: using multiple AI models to catch agent blind spots. The builder-critic architecture and the "two doctors, same patient" heuristic for structurally independent review. https://asdlc.io/patterns/adversarial-code-review/

[5] SmartScope, "Automating the Claude Code × Codex Review Loop – Three Levels," March 2026. Three automation tiers from SKILL.md manual trigger through stop-hook automatic trigger to multi-AI pipeline governance. https://smartscope.blog/en/blog/claude-code-codex-review-loop-automation-2026/

refinement: after the loop converges, spawn a *fresh* Codex session for a final audit. This eliminates context bias from the iterative conversation and catches systemic issues the loop may have normalised.[6]

Setup time is under five minutes. Use Level 1 to validate that cross-model review catches real issues in your codebase before investing in automation infrastructure.

Level 2: Stop Hook — Automatic Trigger

Level 2 eliminates the manual invocation by hooking into the agent lifecycle. When the builder agent attempts to complete a turn, a Stop hook intercepts the exit and triggers the cross-model review automatically.[7] The hook communicates its decision via exit codes: exit 0 with `{"decision": "block", "reason": "..."}` feeds the reason back as a continuation prompt; exit 0 without blocking JSON permits the stop.

The critical implementation detail is a re-entrancy guard. Your stop hook must check a `stop_hook_active` flag before spawning another review. Without this guard, the review's own completion triggers another stop hook, creating an infinite loop—a concrete instance of the unbounded-feedback anti-pattern.[8]

Two plugins implement this tier. OpenAI's `codex-plugin-cc` provides a single-command review gate that blocks Claude Code from finalising changes until Codex has reviewed them.[9] The community `claude-review-loop` plugin takes a more aggressive approach, spawning up to four parallel Codex sub-agents (diff review, holistic review, framework-specific review, UX review) and deduplicating their findings before presenting a consolidated verdict.[10]

---

[6] SmartScope, "Automating the Claude Code × Codex Review Loop — Three Levels," March 2026. Three automation tiers from SKILL.md manual trigger through stop-hook automatic trigger to multi-AI pipeline governance. https://smartscope.blog/en/blog/claude-code-codex-review-loop-automation-2026/

[7] OpenAI, "Hooks – Codex," OpenAI Developers, 2026. Hook lifecycle, Stop hook semantics, and exit-code-based decision protocol. https://developers.openai.com/codex/hooks

[8] SmartScope, "Automating the Claude Code × Codex Review Loop — Three Levels," March 2026. Three automation tiers from SKILL.md manual trigger through stop-hook automatic trigger to multi-AI pipeline governance. https://smartscope.blog/en/blog/claude-code-codex-review-loop-automation-2026/

[9] OpenAI, "codex-plugin-cc," GitHub, March 2026. Official Codex plugin for Claude Code providing review gate, adversarial review, and background task delegation. https://github.com/openai/codex-plugin-cc

[10] Hamel Husain, "claude-review-loop," GitHub, 2026. Community plugin spawning parallel Codex sub-agents for multi-dimensional review with deduplication. https://github.com/hamelsmu/claude-review-loop

Level 3: Multi-AI Pipeline Governance

Level 3 moves beyond a single reviewer to an orchestrated pipeline where different AI systems handle distinct quality dimensions. The `claude-codex` plugin implements a three-reviewer sequential chain: Claude Sonnet implements, a second Claude Sonnet reviews, Claude Opus provides a deeper architectural review, and Codex CLI acts as the final gate.[11] Each reviewer independently validates against OWASP Top 10 vulnerabilities, and the pipeline enforces sequential dependencies—Review 2 cannot start until Review 1 approves. If any reviewer requests changes, a fix task and re-review are automatically created, with explicit iteration limits (typically 10–15 rounds) to prevent runaway loops.

The three levels map to team scale: Level 1 for solo developers validating the approach, Level 2 for small teams that need automatic triggers, Level 3 for organisations requiring formalised quality gates with audit trails.

The Feedback Cascade: A Coordination Primitive

Across all three levels, the underlying coordination primitive is the same: the **feedback cascade**. One agent produces an artefact, a second agent evaluates it and emits structured feedback, and the first agent consumes that feedback to produce a revised artefact. The cascade terminates on a pass verdict or an iteration cap.

What distinguishes the feedback cascade from a simple retry is that the feedback is *structured and directional*. The reviewer does not merely say "try again"; it returns an ordered list of violations with file-and-line evidence, each tagged by severity. The builder consumes these as a prioritised work list rather than a vague instruction to improve. This structure is what makes the loop converge: each round addresses specific findings rather than groping toward an undefined standard.

The cascade is composable with the other three patterns. A gated chain can use a feedback cascade at any stage where the gate detects a semantic rather than structural failure. A wave-based executor can embed a cascade within a single wave task—the wave boundary ensures the cascade converges before downstream tasks begin. A parallel swarm can run independent cascades

[11] Z-M-Huang, "claude-codex: Multi-AI orchestration plugin," GitHub, 2026. Three-reviewer sequential pipeline with OWASP validation and iteration-capped fix loops. https://github.com/Z-M-Huang/claude-codex

per worker, with a judge agent sampling a percentage of cascades for quality assurance rather than reviewing every one.

The feedback cascade also explains why the re-entrancy guard in Level 2 is essential. Without it, the cascade's termination condition—the reviewer emitting a `PASS` verdict—itself triggers a new cascade, and the system never reaches a fixed point. Bounded iteration (typically three to five rounds) and explicit termination signals are non-negotiable when embedding cascades in automated pipelines.

Agentmaxxing: Human-Coordinated Cross-Vendor Parallelism

The patterns above assume orchestration happens within Codex CLI's own subagent system or through shell-level fan-out of Codex processes. A different approach dispenses with programmatic orchestration entirely and puts the human developer in the coordinator role across multiple CLI agent products simultaneously. This pattern—colloquially known as "agentmaxxing"—involves running Codex CLI, Claude Code, Gemini CLI, or other agentic tools in parallel, each in its own git worktree, each tackling a different task or a different slice of the same problem.

The setup is straightforward: create separate worktrees (`git worktree add ../feature-auth`, `git worktree add ../feature-api`), launch a different agent in each terminal, and coordinate by reviewing diffs and merging branches yourself. The human provides the decomposition, the task assignment, and the integration—the roles that a programmatic orchestrator would handle in Tiers 1–3. The agents provide raw throughput. This is not a replacement for Codex-native subagent orchestration; it is a complementary pattern for situations where the tasks are large enough to warrant full independent agent sessions and where cross-vendor model diversity provides genuine coverage benefits (different models catch different classes of bugs, as the Cross-Model Review Loop demonstrates). The tradeoff is that coordination overhead falls entirely on the developer, which limits the pattern to teams comfortable managing parallel branches and resolving merge conflicts manually.

Conversation Branching as a Supervisor Pattern

The patterns above treat agents as separate processes coordinated by an external orchestrator. Codex CLI v0.122.0 introduced a different primitive: conversation branching within a single session, using the `/side` and `/fork` slash commands. These commands turn the developer (or a parent agent) into a lightweight supervisor that can monitor, query, and redirect work without the overhead of a full orchestration harness.

`/side` and `/fork`: The Two Branching Primitives

`/side` creates an ephemeral, in-memory fork of the current conversation. The side thread inherits the parent's context as read-only reference, supports only a limited set of non-modifying commands (`/copy`, `/diff`, `/mention`, `/status`), and vanishes when you close it. The main agent continues running in the background, uninterrupted.[12]

`/fork` creates a persistent branch. It clones the full conversation history into a new on-disk session that you can `/resume` later. Unlike `/side`, a fork has full command access and can modify files. You can also fork from an earlier point in the transcript by scrolling back and pressing Enter at the desired message.[13]

The two primitives sit at opposite ends of a weight spectrum:

Dimension	`/side`	`/fork`
Persistence	In-memory only	On-disk session
File modifications	Blocked	Allowed
Context inheritance	Read-only reference	Full clone

[12] Codex CLI v0.122.0 introduced `/side` (ephemeral forks) and enhanced `/fork` (persistent branches) as conversation branching primitives. See "Codex CLI Conversation Branching: /side, /fork, and Plan Mode Workflows," codex-resources, April 2026.

[13] Codex CLI v0.122.0 introduced `/side` (ephemeral forks) and enhanced `/fork` (persistent branches) as conversation branching primitives. See "Codex CLI Conversation Branching: /side, /fork, and Plan Mode Workflows," codex-resources, April 2026.

Dimension	`/side`	`/fork`
Token cost	Minimal (~500–2,000 tokens)	Full (clones entire history; prompt caching mitigates API cost)
Resumable	No	Yes (`/resume`)
Available commands	4 (limited set)	Full

The Supervisor Pattern

When you combine `/side` and `/fork` with Codex CLI's subagent system, a supervisor pattern emerges. The supervisor is an agent (or a developer) that holds the high-level plan and delegates bounded units of work to forked subagents, using side threads to monitor progress and ask clarifying questions without interrupting execution.

```
flowchart TD
    S["Supervisor (main thread)"] -->|"/fork — API work"| F1["Fork 1: REST API updates"]
    S -->|"/fork — SDK work"| F2["Fork 2: Client SDK regeneration"]
    S -->|"/fork — docs work"| F3["Fork 3: Documentation updates"]
    S -.->|"/side — status check"| F1
    S -.->|"/side — quick question"| F2

    F1 -->|"result"| M["Merge / review in main thread"]
    F2 -->|"result"| M
    F3 -->|"result"| M

    style S fill:#fff3e0
    style F1 fill:#e3f2fd
    style F2 fill:#e3f2fd
    style F3 fill:#e3f2fd
    style M fill:#e8f5e9
```

The supervisor decomposes a task (e.g., "update the API, the client SDK, and the documentation"), forks a subagent for each domain, and coordinates their outputs. While the forks are executing, the supervisor can open `/side` threads to check on progress or verify assumptions without injecting tokens into any fork's working context. Each fork operates in its own context window, so documentation updates do not consume tokens needed for API code changes.

This pattern maps naturally onto the three-tier landscape from the beginning of this chapter. At Tier 1, the supervisor and its forks run within a single Codex CLI session. At Tier 2, each fork can run in its own terminal with a dedicated git worktree to prevent file conflicts. The branching primitives provide the coordination mechanism that shell fan-out otherwise handles through filesystem conventions.

Composing Branching with Plan Mode

Plan Mode (`/plan` or `Shift+Tab`) gained fresh-context implementation in v0.122.0. After a planning phase that may consume 30–50% of the context window with file reads and architectural discussion, the developer can choose to start implementation in a clean context, carrying forward only the plan itself (typically written to a `PLANS.md` file).[14]

This composes with the supervisor pattern in a three-phase workflow:

1. **Plan** in the main thread. Gather context, discuss architecture, produce a plan.
2. **Fork** bounded implementation tasks from the plan. Each fork starts with a reference to the plan document rather than the full planning history.
3. **Side-verify** assumptions during implementation. Use `/side` to check documentation, confirm API signatures, or validate edge cases without polluting the implementation context.

The key insight is that branching reduces the tension between thorough planning and lean implementation. Without branching, you choose between a context window bloated by planning tokens or a fresh session that lacks the planning context. With branching, each phase gets its own context budget.

When Branching Replaces Heavier Orchestration

Not every multi-agent workflow needs a `WavePlan` or a `spawn_agents_on_csv` call. If your task decomposes into three to five bounded subtasks with a human or parent agent available to coordinate, conversation branching may be sufficient. The decision criteria:

[14] Codex CLI v0.122.0 introduced `/side` (ephemeral forks) and enhanced `/fork` (persistent branches) as conversation branching primitives. See "Codex CLI Conversation Branching: /side, /fork, and Plan Mode Workflows," codex-resources, April 2026.

- **Use branching** when the number of parallel work streams is small (under ~6), the supervisor is a human or a single parent agent, and the subtasks are heterogeneous enough that a shared prompt template would not help.
- **Use programmatic orchestration** when the number of work units is large (dozens or more), the units are homogeneous, or the dependency graph requires automatic wave scheduling.

Branching and programmatic orchestration are not mutually exclusive. A wave-based executor can use `/fork` to isolate individual wave tasks that benefit from conversational context. A supervisor using `/fork` can delegate one of its forks to a `spawn_agents_on_csv` swarm for the bulk-processing portion of a larger task.

Goal Mode: From Task Execution to Objective Tracking (Preview)

The orchestration patterns covered so far—chains, swarms, waves, review loops, and conversation branching—all operate on discrete tasks: bounded units of work with clear start and end conditions. Goal Mode, expected in Codex CLI v0.123, introduces a different abstraction: persistent objectives that span multiple tasks and track progress against a token budget.[15]

The Shift from Tasks to Objectives

In current Codex CLI workflows, you issue a prompt, the agent executes, and the session ends (or you issue another prompt). Each interaction is a self-contained task. Goal Mode wraps a sequence of tasks in a persistent objective: a named goal with a description, success criteria, and a token budget that bounds how much work the agent can do before pausing for human review.

The implementation is tracked in a five-PR stack (#18073–#18077) on the Codex CLI repository.[16] The core mechanics, based on pre-release documentation:

[15] Goal Mode is tracked in PRs #18073–#18077 on the Codex CLI repository. The feature introduces persistent objectives with token budgets, expected in v0.123. https://github.com/openai/codex/pulls

[16] Goal Mode is tracked in PRs #18073–#18077 on the Codex CLI repository. The feature introduces persistent objectives with token budgets, expected in v0.123. https://github.com/openai/codex/pulls

- **Goal declaration.** The developer defines an objective with explicit success criteria and a token budget. The goal persists across sessions.
- **Progress tracking.** As the agent works toward the goal, Codex CLI tracks token expenditure and task completion against the declared criteria.
- **Budget gates.** When token spend reaches the declared budget, the agent pauses and surfaces a progress report. The developer can extend the budget, adjust the goal, or terminate.
- **Multi-session continuity.** A goal can span multiple Codex CLI sessions. Resuming a session with an active goal picks up where the previous session left off.

Orchestration Implications

Goal Mode changes the orchestration landscape in two ways.

First, it moves the iteration boundary. In current patterns, the developer decides when to re-invoke an agent or adjust a plan. With Goal Mode, the agent operates autonomously within its budget, re-planning and re-executing as needed until it either meets the success criteria or exhausts the budget. The developer's role shifts from task-level direction to objective-level oversight—closer to a manager reviewing milestones than a supervisor issuing individual instructions.

Second, it composes with conversation branching to enable longer-running supervisor workflows. A supervisor agent can declare a goal, fork subtasks, track their progress against the budget, and make course corrections—all within a single persistent objective rather than a series of ad-hoc sessions. This is the natural evolution of the supervisor pattern described in the previous section: branching provides the coordination mechanism, and Goal Mode provides the persistence and budgeting layer.

```
flowchart LR
    G["Goal: Migrate auth to JWT"]
    G --> S1["Session 1: Plan + scaffold"]
    S1 --> S2["Session 2: Implement core"]
    S2 --> S3["Session 3: Tests + cleanup"]
    S3 --> D{"Success criteria met?"}
    D -->|Yes| Done["Goal complete"]
    D -->|No, budget remaining| S3
    D -->|No, budget exhausted| R["Pause for human review"]

    style G fill:#fff3e0
    style Done fill:#e8f5e9
    style R fill:#ffebee
```

Practical Guidance

Goal Mode is in preview as of April 2026 and has not reached production stability. The five-PR stack is still under active review. If you are evaluating it:

- Start with small, well-defined objectives where the success criteria are unambiguous (e.g., "all tests pass," "migration script runs without errors"). Vague objectives ("improve code quality") will exhaust the budget without converging.
- Set conservative token budgets initially. You can always extend; you cannot reclaim wasted tokens.
- Use Goal Mode in combination with existing patterns rather than as a replacement. A goal can orchestrate a wave-based executor internally, using the budget gate as an additional safeguard against runaway execution.

Goal Mode does not eliminate the need for the patterns in this chapter. Chains, swarms, waves, and review loops remain the right tools for structuring the work *within* a goal. What Goal Mode adds is the outer loop: the persistent objective that ties multiple orchestration runs together and provides the budgeting discipline that prevents open-ended agent execution from becoming an unbounded cost centre.

Debugging Orchestration Failures

Orchestration failures have a property that single-agent failures don't: the error you observe is often far removed in time and position from its cause. A worker that silently returned an empty string in Wave 1 may not produce a visible error until Wave 3's agent attempts to parse its output. By then, the stack trace points at Wave 3, not Wave 1.

The most effective defence is structured logging at every agent boundary. Under the hood, Codex CLI's agentic loop uses an asynchronous submission/event queue architecture: clients push operations via `submit(Op)` and the engine emits `EventMsg` notifications back through `next_event()`.[17] Every tool call flows through a centralised `ToolRouter` that classifies it and routes it through an approval gate before dispatching to one of three execution backends (built-in shell, `apply_patch` for file edits, or MCP server integration).[18] This means every agent boundary already produces a stream of typed events you can tap into. Codex CLI's `OrchestratorLogger` attaches a unique trace ID to each orchestration run and records every agent invocation, its input hash, its output hash, and its gate result. When a failure occurs, you replay the trace to identify the earliest point where output diverged from expectations:

```
from codex import OrchestratorLogger

logger = OrchestratorLogger(
    run_id="refactor-2026-03-27",
    log_level="DEBUG",        # logs every gate evaluation
    persist_to="./run-logs/", # writes JSONL per wave
)

executor = wave_executor(plan, logger=logger, max_concurrency_per_wave=4)
```

With `log_level="DEBUG"`, every gate evaluation is logged with the raw agent output, the parsed result, and the validation verdict. When you open the JSONL file for a failed run, you filter for `"gate_result": "fail"` and immediately find the earliest silent failure.

[17] Architecture overview of the Codex CLI agentic loop: submission/event queue pattern, ToolRouter dispatch, approval gate state machine, and kernel-level sandbox enforcement via Landlock LSM and Seatbelt profiles. https://deepwiki.com/openai/codex

[18] Architecture overview of the Codex CLI agentic loop: submission/event queue pattern, ToolRouter dispatch, approval gate state machine, and kernel-level sandbox enforcement via Landlock LSM and Seatbelt profiles. https://deepwiki.com/openai/codex

A second debugging technique is checkpointing. Long-running orchestrations—especially wave-based ones that take minutes or hours—should checkpoint completed wave outputs to disk or a key-value store. If a run fails in Wave 3, you resume from the Wave 2 checkpoint rather than re-running from scratch:

```
executor = wave_executor(
    plan,
    checkpoint_dir="./checkpoints/refactor-2026-03-27/",
    resume_from_checkpoint=True,  # skip waves whose outputs are already on disk
)
```

When workers fail silently—returning a well-formed response that is semantically wrong—you need a different approach: semantic validation. Structural gates catch format errors; semantic validation uses a lightweight judge agent to evaluate whether the output makes sense given the input. This adds latency and cost, so reserve it for high-stakes stages rather than applying it universally.

Silent semantic failures are the hardest class of orchestration bug. A worker that returns `{"category": "billing", "suggested_priority": "P1"}` for every ticket regardless of content will pass all structural gates. The only way to catch this is to sample worker outputs during development and inspect them manually, or to include a judge agent that evaluates a random 10 per cent of results.

Orchestration Anti-Patterns to Avoid

Three anti-patterns appear so frequently in production Codex CLI deployments that they deserve explicit treatment.

Orchestrators that don't check worker output. The most common mistake is treating agent output as trusted input to the next stage. Every handoff between agents is a trust boundary. The upstream agent's output may be malformed, truncated, or semantically wrong. Always insert a gate—even a minimal one that checks for non-empty output and valid JSON structure—at every handoff. The cost is negligible; the debugging time saved is substantial.

Fan-out without an aggregation plan. Spawning a swarm of workers is easy. Deciding in advance what you will do with their collective output is harder

and more important. Before you write a single line of fan-out code, write the aggregation function. Know whether you need a merged list, a summary artefact, a consensus vote among conflicting worker results, or a diff applied to a shared artefact. If you cannot describe the aggregation in one sentence, your task decomposition is wrong; revisit it before parallelising.

Unbounded parallelism. It is tempting to set `max_concurrency` to a large number or omit it entirely on the assumption that "more parallel is faster." In practice, unbounded parallelism causes three problems. First, API rate limits kick in and most workers begin failing with 429 errors, which your error handling may not distinguish from semantic failures.[19] Second, if workers share any mutable state—a file, a database row, a cache entry—concurrent writes corrupt it. Third, the cost of debugging 500 simultaneous failed workers is orders of magnitude higher than debugging 10. Set `max_concurrency` conservatively, profile your pipeline under realistic load, and increase it incrementally.

Omitting `max_concurrency` in `spawn_agents_on_csv` or `wave_executor` is a footgun on CSVs with more than a few dozen rows. Codex CLI will attempt to spawn one agent per row concurrently. For a 10,000-row CSV, this means 10,000 simultaneous API calls. Even if your rate limit allows it, the memory overhead of 10,000 in-flight coroutines will degrade performance significantly. Always set an explicit `max_concurrency`.

A fourth anti-pattern worth naming is **reviewers with write access.** When you embed a cross-model review loop in your orchestration, the reviewer agent must run in a read-only sandbox. A reviewer that can modify the codebase can mask its own findings—or worse, introduce changes that the builder never intended. Codex CLI enforces this at the kernel level through Landlock LSM on Linux and Seatbelt profiles on macOS, not at the application layer, so a misbehaving agent cannot circumvent the restriction by spawning a child process.[20] Always pass `--sandbox read-only` (or the equivalent `ReadOnly` sandbox policy) for any agent in a reviewer role.

A fifth anti-pattern worth naming is **shared mutable context between parallel workers.** Workers in a swarm are designed to be stateless and

[19] Codex CLI command-line reference documenting `max_concurrency`, approval modes, and rate-limit behaviour in agentic pipelines. https://developers.openai.com/codex/cli/reference

[20] Architecture overview of the Codex CLI agentic loop: submission/event queue pattern, ToolRouter dispatch, approval gate state machine, and kernel-level sandbox enforcement via Landlock LSM and Seatbelt profiles. https://deepwiki.com/openai/codex

independent. If you find yourself passing a shared mutable object—a list being appended to, a counter being incremented, a file being written to—into worker closures, you are creating a race condition. Use the collect-then-reduce model: workers write to their own isolated outputs, and aggregation happens after all workers complete.

If you're unsure whether your workers are truly independent, apply the commutativity test: would the final result be the same if the workers ran in a different order? If yes, they're safe to parallelise. If no, there's a hidden dependency you need to make explicit in your wave plan.

* * *

Summary

Orchestration patterns give you a vocabulary for structuring multi-agent work. The Sequential Gated Chain handles linear pipelines where each stage feeds the next, and its gates prevent silent failures from propagating downstream. The Parallel Worker Swarm handles bulk independent work through fan-out and requires an explicit aggregation plan and a bounded concurrency cap. The Wave-Based Hybrid handles tasks with dependency graphs by grouping independent work into waves and enforcing wave boundaries as implicit quality gates. The Cross-Model Review Loop layers structurally independent judgement on top of any of the other three patterns, using the feedback cascade primitive—structured findings, bounded iteration, explicit termination—to drive convergence between a builder and a reviewer drawn from different model families.

When choosing among these patterns, start with the dependency structure of your task. Linear dependencies map to chains. Independent units map to swarms. Partial dependencies map to waves. When the output is high-stakes, add a cross-model review loop at the appropriate automation tier. When your task doesn't fit neatly, compose the patterns: a chain can invoke a swarm, a swarm can run chain logic per worker, a wave can contain either, and a feedback cascade can embed within any stage.

Conversation branching (`/side` and `/fork`, introduced in v0.122.0) adds a lightweight supervisor pattern to the toolkit. The supervisor decomposes work into forked subagents, monitors them through ephemeral side threads, and coordinates results—without the setup cost of a programmatic orchestration harness. Branching composes with Plan Mode's fresh-context implementation to keep each phase of a workflow within its own context budget. Use branching when the number of parallel streams is small and heterogeneous; reach for programmatic orchestration when work units are numerous and homogeneous.

Goal Mode (expected v0.123) extends orchestration from discrete tasks to persistent objectives. By wrapping multiple sessions in a named goal with success criteria and a token budget, it shifts the developer's role from task-level direction to objective-level oversight. Goal Mode does not replace the patterns in this chapter; it provides the outer loop that ties them together and imposes budgeting discipline on open-ended agent work.

Debugging orchestration failures requires structured logging at agent boundaries, checkpointing for long runs, and semantic validation for the cases where structural gates aren't sufficient. The anti-patterns to avoid are ungated handoffs, fan-out without aggregation, unbounded parallelism, reviewers with write access, and shared mutable context—each of which is easy to introduce and expensive to diagnose.

These four core patterns operate within a three-tier orchestration landscape: in-process (this chapter), local shell fan-out (Chapter 16), and cloud fan-out (Codex Cloud Tasks). They exist alongside emerging patterns—A2A protocol for cross-tool agent coordination, persistent orchestration for always-on agent infrastructure, and conversation branching and Goal Mode for supervisor-driven workflows—that extend orchestration beyond discrete task graphs.

The next chapter extends these patterns to persistent, long-running agents that maintain state across multiple invocations—moving from orchestration of discrete task graphs to coordination of continuously running agent processes.

Exercises

1. **Implement a gated chain.** Write a two-agent chain that (a) extracts action items from a meeting transcript and (b) formats them as a GitHub issue body. Insert a gate between the agents that validates the extractor's

output against a Pydantic schema requiring at least one action item with an owner and a due date. Test the gate by providing a transcript that produces an empty list and verify that the gate triggers a retry.

2. **Profile a parallel swarm.** Create a CSV with 50 rows of synthetic data (ticket title and description). Run `spawn_agents_on_csv` with `max_concurrency` values of 2, 5, 10, and 20. Record total wall-clock time and the number of API errors at each setting. Plot the results and identify the concurrency level at which additional parallelism stops improving throughput.
3. **Build a wave plan from a dependency graph.** Take the following task list for a data pipeline refactor: (a) update the schema definition file, (b) update the ingestion script to use the new schema, (c) update the transformation script to use the new schema, (d) update the ingestion tests, (e) update the transformation tests, (f) run the integration test suite. Draw the dependency graph, identify the wave boundaries, and encode the plan as a `WavePlan` object. Add checkpointing so that a failure in the test wave doesn't require re-running the earlier waves.
4. **Add semantic validation to a swarm.** Extend the ticket classifier from the Pattern 2 section to include a judge agent that samples 10 per cent of worker outputs and evaluates whether the assigned category is plausible given the ticket description. Log any results where the judge disagrees with the worker. Run the pipeline on a 100-row CSV and report the disagreement rate. See Chapter 16 for techniques to use judge feedback to improve worker prompt quality over time.
5. **Implement a supervisor workflow with conversation branching.** Design a task that decomposes into three bounded subtasks (e.g., updating an API, regenerating a client SDK, and updating documentation). Use `/fork` to create a subagent for each subtask, and use `/side` to verify an assumption during one of the forks without interrupting it. Document the token cost of each fork versus what a single-thread approach would have consumed. Pair each fork with a separate git worktree to prevent file conflicts.
6. **Evaluate Goal Mode's budget gate.** (Requires v0.123 or later.) Define a persistent goal with explicit success criteria ("all unit tests pass after migrating module X from callback style to async/await") and a token budget of 50,000 tokens. Run the goal and observe where the budget gate triggers. Record how many sessions the goal spans and whether the success criteria were met within the budget. Adjust the budget and re-run, noting the relationship between budget size and goal completion

rate.

7. **Implement a cross-model feedback cascade.** Write a SKILL.md file that defines a `/cross-review` command. The command should: (a) export the current `git diff` to a temporary file, (b) invoke `codex exec` in read-only sandbox mode with a different model than the builder, (c) parse the reviewer's verdict as `PASS` or `CONCERNS` with a list of findings, (d) if `CONCERNS`, feed the findings back to the builder and re-invoke the reviewer on the updated diff, and (e) cap the loop at three rounds before escalating. Add a re-entrancy guard so the review's own completion does not trigger a nested review. Test the cascade on a deliberately buggy diff and verify that the loop converges within the iteration cap.

Chapter 19. Worktrees and Isolated Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Git Worktrees: A Brief Recap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Why Worktrees Matter for Agentic Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

One Agent, One Worktree: The Isolation Principle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Worktrees in the Codex Desktop App

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CLI Worktree Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Creating a worktree for a task

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Running agents in parallel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Inspecting worktree state

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Merging Agent Work Back to Main

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Sequential merge (simple cases)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Rebase merge (linear history)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Merge queue (high-concurrency cases)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Worktree setup scripts and local environments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Worktree Lifecycle in CI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Basic lifecycle script

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

GitHub Actions example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Parallel matrix jobs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Worktree Workflow Patterns by Team Size

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 20. CI/CD Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Running Codex in Non-Interactive Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The openai/codex-action GitHub Action

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Automated Code Review on Every PR

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Test Generation on Merge

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Dependency Update Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

GitLab CI/CD Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Code Quality in .gitlab-ci.yml

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Analytics Pipeline: CI/CD Observability Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Token Usage Tracking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Steering Metadata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Thread Context Denormalisation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Practical CI/CD Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Self-Healing CI/CD: From Observation to Autonomous Remediation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Level 1: Observer (suggest mode)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Level 2: Gatekeeper (auto-edit mode)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Level 3: Healer (full-auto mode)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Pipeline Doctor Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CI Autofix: Self-Repair Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Project-Level Skills: The codex-pr-body Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Stacked PRs as a Review-Scaling Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Structured Output and Session Resume

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Safety Strategies for CI Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Remote Sandbox Configuration: Hostname-Pattern Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Hermetic Execution Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Empirical Evidence: What 33,000 Agentic PRs Reveal About Pipeline Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 21. Security Hardening

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Threat Model for Agentic Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Prompt Injection: Attack Patterns and Defences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

How the attack works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Defences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Filesystem Restrictions and Sandboxing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

How the OS-Level Sandbox Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Sandbox Denial Detection and Escalation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Remote Sandbox Configuration by Hostname Pattern (#18763)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Guardian Review and the Trust Ladder

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Agent Identity and Biscuit Tokens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Agent Identity as First-Class Auth Mode (v0.123 Alpha)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Identity and Authentication Hardening (v0.126.0 Alpha)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Deny-Read Glob Policies (v0.122.0)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CI Isolation: `--ignore-user-config` and `--ignore-rules`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Network Allowlisting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Secret Management for Agent Environments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Audit Logging and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP as an Attack Surface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CVE-2025-61260: The Codex CLI Config Injection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

OWASP MCP Top 10 Mapping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

MCP Tool Annotations as Security Mechanism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enterprise MCP Hardening Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Supply Chain Attacks: The Axios Incident and Binary Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Verifying Codex CLI binary signatures post-rotation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Case study: the Axios supply chain compromise

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enterprise Defense-in-Depth: The Five-Layer Security Model

Layer 1: Filesystem Isolation — Managed Deny-Read (PR #17740)

Layer 2: Execution Environment Isolation — Named Exec Environments (PR #17741)

Layer 3: Permission Negotiation — Hook Suggestions (PR #17739)

Layer 4: IPC Scoping — Unix Socket Allowlists (PR #17654)

Layer 5: Tool Context Awareness — Sandbox State in MCP Metadata (PR #17763)

The Defense-in-Depth Trust Table

Empirical Security Findings: The Amplifying.ai Benchmark

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Benchmark Methodology and Results

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

AGENTS.md as a Security Requirements Specification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cross-Model Review as a Security Defence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Sandbox Enforcement in Practice: The YOLO Proxy Fix (PR #17742)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Compliance Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

GDPR

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

SOC 2

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 22. Enterprise Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Distributing config.toml at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

RBAC: Three Admin Roles and Access Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

SCIM group synchronisation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Managed Policies: requirements.toml

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Policy tier design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

requirements.toml format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Deploying managed policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Runtime controls (v0.116.0)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Managed configuration patterns for fleet management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

AGENTS.override.md: Enforcing Policy Across Teams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Onboarding an Engineering Team

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Measuring ROI: Metrics That Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex Cloud vs Self-Hosted

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Multi-Cloud Provider Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Agent Identity Authentication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Governance Frameworks for Agentic AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The three-stage guardrails pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The managed deny-read stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Remote MCP Executor Stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Governance APIs and Data Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Analytics Dashboard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Statsig integration for usage governance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Compliance API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Rollout Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

External Governance Frameworks: Microsoft AGT and Forrester ADS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Microsoft Agentic Governance Toolkit (AGT)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Forrester Agentic Decision Support (ADS) Framework

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Remote MCP Executor Stack: enterprise deployment implications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Multi-environment support (v0.123.0)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex Labs and GSI Network

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

AWS AgentCore managed harness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Workspace agents and Slack integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Permission Profile Phase 3 (PRs #19772–#19776)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex Orchestrator: fleet management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

SSH remote connections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enterprise onboarding resources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 23. Testing and Evaluation Strategy for Agentic Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What Makes a Test Suite Agent-Friendly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Hermetic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Deterministic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Single-command invocable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Designing for Agent Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Seed fixtures and mock services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Test isolation patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Environment variable injection for agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Feedback Signal Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The gate command pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Using Codex CLI to audit your test suite

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What to ask for

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Making the audit actionable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Evaluation Beyond Unit Tests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Integration test strategies for agentic workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Property-based testing for agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Sampling-based evaluation for long-horizon tasks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Building an Evaluation Harness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Structure of a minimal harness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Connecting to SWE-bench style evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CI integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

TDD as an Agent Feedback Loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Structuring Failing Tests for Codex

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The AGENTS.md TDD Convention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Test-First for Bug Fixes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The 4-File Durable Memory Pattern for Long-Horizon Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

April 2026 Evaluation Frameworks: CocoaBench, HiL-Bench, and AAR

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Using the failure taxonomy as a test coverage checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CocoaBench as an evaluation framework

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

HiL-Bench for approval mode calibration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

AAR for AGENTS.md effectiveness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

Specialised Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 24. AI Code Review

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Why AI Code Review Works (and Where It Doesn't)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Empirical evidence: the c-CRAB benchmark

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Configuring Codex for Code Review

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Selecting the review model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Calibrating strictness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The /review Command

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

PR Integration: Automated Review on Every PR

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

GitHub cloud review

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CI review with codex-action

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Structured Output Code Review with `codex exec --output-schema`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Structured Output Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Platform-Specific Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cross-Model Adversarial Review

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Writing Review Checklists in AGENTS.md

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Human-AI Review Collaboration Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Review-Fix Loop: A Three-Level Maturity Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Level 1: Manual review

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Level 2: Autonomous stop-hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Level 3: Multi-AI governance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 25. Frontend Engineering with React and TypeScript

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Frontend-Specific AGENTS.md Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Component Generation and Scaffolding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Test Generation for React Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Explorer/Worker Sub-Agent Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Accessibility Audit Automation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Design-to-Code Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex Browser Use: Visual Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The build-and-verify loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Comparison with Claude Computer Use and Devin

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

When to use browser verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 26. Python Team Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Python-Specific AGENTS.md

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Monorepo overrides

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Pytest Integration and Test Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Fixture discovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Coverage thresholds and test markers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Automated test expansion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Type Hints and Docstring Automation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Typing style in AGENTS.md

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Docstring generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

uv, ruff, and Modern Python Toolchain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

pyproject.toml as the single source of truth

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Hooks for automatic enforcement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Data Pipeline Code Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Pandas and Polars patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Spark workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Multi-Service Python Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

FastAPI service conventions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Async patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 27. Web Search and Research Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enabling Web Search in Codex CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The three modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Runtime override

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Sandbox mode interaction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Per-profile configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enterprise controls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Known limitation: minimal reasoning effort

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Research-to-Code Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Encoding the pattern in AGENTS.md

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Dependency Research and Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Staying Current with API Changes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Knowledge-Augmented Agents: MCP Knowledge Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Context7: version-specific library documentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

OpenAI Docs MCP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Playwright: when you need the live page

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Combining Web Search with Sub-Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

A parallel dependency evaluation pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Research parallelism without sub-agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 28. Codebase Migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Why Codex CLI Excels at Migration Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Planning the Migration with Codex CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Encoding the transformation rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Incremental Migration Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Validation Strategies During Migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Snapshot tests and migration safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Migrating from Claude Code to Codex CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Step 1: CLAUDE.md to AGENTS.md

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Step 2: Skills

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Step 3: Subagents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Step 4: Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Step 5: Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Framework and Language Version Migrations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Architecture and Vision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 29. The Agents SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What the Agents SDK Adds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The SDK Architecture: Agents, Handoffs, Tools, and Guardrails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex CLI as an MCP Server in SDK Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Building a Designer-Developer-Tester Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The RECOMMENDED_PROMPT_PREFIX pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Wiring the three-stage pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Scaling to a Project Manager orchestrator

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Tracing and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

SDK vs CLI: Choosing the Right Level

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

TypeScript SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Installation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Core Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Basic Agent Creation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Streaming Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Tool Use in TypeScript

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Differences from the Python SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

CI/CD Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Recent SDK Developments (April 2026)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 30. Agentic Primitives Compared

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Four Primitives: Agents, Handoffs, Tools, Guardrails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

How Codex CLI Implements Each Primitive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Agents: AGENTS.md as the persistent agent spec

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Handoffs: sub-agents as the delegation mechanism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Tools: MCP as the universal tool interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Guardrails: hooks and approval modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

LangChain and LangGraph

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

AutoGen and CrewAI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Google Gemini Agents and ADK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Choosing a Primitives Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Codex CLI's Custom Agent TOML in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Extending the Primitives: Custom Agents in TOML vs Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 31. Harness Engineering for Long-Running Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What Harness Engineering Is

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The three pillars

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The WORKFLOW.md Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

State Persistence Across Agent Runs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Error Recovery and Resumption

The Proof-of-Work Principle

Symphony: A Harness in Practice

NanoClaw: A persistent orchestration reference

What Symphony tells you about where your own harness should go

SaaS Readiness and Production Embedding

Local Model Providers as Harness Components

Docker MCP Toolkit as Harness-Level Isolation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Built-in Code Review Skill

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

App Server JSON-RPC for Custom Harness Clients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

LangChain Deep Agents: Harness Middleware as Competitive Advantage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Factory Model: Osmani's Production Agent Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Agent Harness Engineering: Osmani's Taxonomy and the Ratchet Principle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Academic Validation: The Harness Engineering Survey

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Cross-Tool Harness Portability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Crate Extraction as Harness Engineering at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Forward Deployed Engineer as Harness Practitioner

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Chapter 32. The Agentic Engineering Pod

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Learning Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Section 1: Why Three Roles?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The problem with traditional team structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The three-person rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The key structural insight

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Section 2: The Context Architect: Human Role

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What the Context Architect owns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The autonomy ladder

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

AGENTS.md as the Context Architect's primary artefact

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Specification as a forcing function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Distributing the Context Architect's craft as a plugin

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Context Architect TOML configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Section 3: The Value Engineer: Human Role

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What the Value Engineer owns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Briefing engineering in practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Value Engineer's Codex session pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Proportional review depth

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Value Engineer TOML configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Section 4: The Quality Engineer: Human Role

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What the Quality Engineer owns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Deterministic tool interface design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The hooks the Quality Engineer authors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Plugin-enforced hooks: making bypass structurally impossible

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Verification economics: cost-tiered gates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Quality Engineer TOML configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Section 5: The Pod in Practice: A Feature Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Step 1: Context Architect frames the feature

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Step 2: Value Engineer orchestrates implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Step 3: Quality Engineer's automated gates fire

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Flow diagram

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Section 6: The Agentic Pod Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Principle 1: Fix time/budget, flex scope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Principle 2: The three-person rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Principle 3: Sketch first, code last

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Section 7: Failure Modes and Pod Anti-Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Context debt

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Gate bypassing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Single-point quality failure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Role collapse

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Scope creep

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Section 8: Distributing the Pod with Plugins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The configuration problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The plugin as pod infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Monolithic versus composable plugins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Enterprise governance through the marketplace

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What plugins do not change

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Metric Freedom and Benchmark Failure Modes: Tools for the Pod

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Metric Freedom (F) as a pod decision tool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Benchmark failure modes as pod coverage dimensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Closing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Conclusion: The Agentic Engineer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

A Note from the Author, Or Rather, the Tool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

How This Book Was Made

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

What This Means for You

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

The Book as Artefact

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

An Invitation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

A Face for the Voice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

Bibliography

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

1. Research Papers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

2. OpenAI Sources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

3. Standards and Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

4. Frameworks and Protocols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

5. Benchmarks and Evaluations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

6. Developer Tooling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.

7. Industry Reports and Blogs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/codex-cli.