Build and Deploy Anywhere with GPT-5 Codex

Booklet

Build and Deploy Anywhere with GPT-5 Codex

A hands-on guide to agentic software engineering

From first install to production deployment

Corrected & Updated Edition — 2026

Reflects the Codex CLI and GPT-5-Codex model family (recommended default: gpt-5.5)

Table of Contents

Chapter 1 — Introduction to Agentic Engineering

Software engineering has always evolved through tooling. The compiler turned symbolic logic into executable artifacts. The integrated development environment reduced cognitive friction by surfacing syntax errors in real time. Version control enabled distributed collaboration. Continuous integration automated quality checks and deployment. Each generation of tools shifted the boundary of what an individual engineer could accomplish.

Yet through all of it, one principle held: the human engineer orchestrated the process. Tools executed instructions but did not decide what to build, how to sequence work, or how to reconcile competing constraints. Even as automation increased, agency remained human.

Agentic software engineering changes that distribution of responsibility. In this paradigm, an intelligent system does not merely respond to prompts or complete lines of code. It interprets goals, constructs plans, executes actions in a development environment, evaluates intermediate outcomes, and adjusts. It can create repositories, generate modules across directories, run test suites, inspect failures, patch defects, commit changes, and open pull requests. It participates in engineering work as an actor within the workflow.

The term agentic is deliberate. In artificial-intelligence research, an agent perceives its environment, reasons about objectives, plans actions, executes them, and updates its internal state based on outcomes. Applied to software engineering, this means a system that does more than generate text: it operates inside an environment, interacts with tools, and pursues structured goals over time.

This book is grounded in a specific, real tool: OpenAI Codex. Codex is a coding agent that runs in your terminal, inside your IDE, in the cloud, and on GitHub, powered by the GPT-5-Codex family of models. We will use it to build and deploy a real application end to end. But the ideas transfer: the discipline of working with an agent matters more than any single vendor.

To appreciate the shift, consider a concrete example. In a traditional workflow, a product manager requests a dashboard displaying podcast analytics. A developer designs the structure, initializes a project, installs dependencies, scaffolds files, implements chart components, wires up data parsing, writes tests, commits changes, and submits a pull request. The process is sequential and human-driven.

In an agentic workflow, the developer states a high-level objective:

> *"Create a React-based analytics dashboard using this CSV dataset. Include charts for downloads, episode duration, and guest frequency."*

The agent responds not with a single code block but with a structured plan. It scaffolds a project, installs dependencies, generates components, parses the dataset, runs the development server, and surfaces a preview. If a test fails, it inspects the output and patches the code. If the repository

lacks version control, it initializes Git and commits with descriptive messages. The difference is not syntactic productivity; it is operational participation.

This does not mean the human engineer disappears. On the contrary, human responsibility becomes more concentrated. The engineer defines constraints, validates architectural decisions, reviews diffs, and decides whether the agent's output aligns with broader system goals. Agency shifts from manual execution to structured oversight.

Consider a failing test in a Node.js project:

```
npm test

> FAIL  src/utils/parseCsv.test.js
  × parses numeric downloads field correctly

    Expected: 1200
    Received: "1200"
```

A generative assistant can explain that the field must be converted to a number. An agent can open the file, identify the parsing logic, modify it, re-run the tests, and verify that the failure disappears — all without further prompting. It closes the loop between reasoning and execution. That loop is the core of agentic engineering.

The rise of such systems reflects broader technical shifts. Models now operate with large context windows, reasoning across many files at once. Tool-orchestration layers let them run shell commands, manipulate files, and interface with version control. Multimodal capabilities let them interpret screenshots and diagrams and map visual elements back to source code. For instance, a developer can highlight a paragraph in a running web application and ask the agent to remove it; a multimodal agent can locate the corresponding JSX block and produce a precise diff.

The implications extend beyond convenience. As agents operate at repository scale, architectural clarity becomes more important. Clean module boundaries, consistent naming, and deterministic builds improve not only human comprehension but machine reliability. In tangled codebases with implicit dependencies, agents struggle to reason effectively. In well-structured systems, they thrive.

The governance dimension is critical. When an agent modifies a repository, accountability must remain clear. Audit logs must record actions. Approval gates must protect high-risk operations. Rollback must be reliable. The more capable the agent, the more disciplined the oversight must be. Agentic engineering is powerful, but it is not self-justifying; it must be embedded in structured processes.

In the chapters that follow, we move from concepts to applied practice. We will install and configure Codex, build a real analytics dashboard with it, refactor and extend that dashboard, drive changes from screenshots, work inside an IDE, run parallel variants in the cloud, automate

code review, and — as the title promises — deploy the result. Along the way you will see real commands, real configuration, and corrected code you can run.

Chapter 2 — The Engineering Landscape and the Rise of Agent Models

Agentic engineering did not appear in isolation. It is the result of decades of incremental evolution in developer tooling, automation practice, and model capability. To understand why agent models are a structural shift rather than a feature upgrade, it helps to examine the landscape they enter.

For most of modern software development, the workflow has followed a stable pattern. A requirement is defined. An engineer designs a solution. Code is written in an IDE. Tests run locally. Changes are committed and pushed. Peers review diffs. Continuous integration validates builds. Finally, the code is deployed. Even in highly automated environments, the engineer remains the central coordinator. Tools accelerate compilation, testing, linting, and dependency resolution, but they do not decide what to do next.

Over the past few years, large language models began assisting in this process. Developers could describe a function and receive an implementation, request a unit test and obtain boilerplate, or paste an error message and receive an explanation. These generative tools reduce friction within tasks. But they remain fundamentally reactive: they respond to a prompt, they do not manage a workflow, and they do not execute shell commands or evaluate build output unless a human feeds that output back in.

The difference becomes evident with a realistic integration failure. Suppose a developer modifies a schema in a TypeScript backend, forgetting to update a validation layer. Running the build produces:

```
tsc
error TS2322: Type 'string | undefined' is not assignable to type 'string'.
```

A generative assistant can explain the type mismatch and suggest a fix. But it will not, by itself, open the repository, locate all dependent references, adjust the schema, re-run the build, verify resolution, commit the change, and open a pull request. An agent can. The defining feature of agent models is not improved code synthesis; it is operational participation through a closed loop: perceive, reason, plan, act with tools, observe, adapt.

In a software-engineering context, "environment" means more than text. It includes repository structure, the file system, build outputs, test results, version-control state, and sometimes running applications. An agent can inspect directory trees, read configuration, detect missing dependencies, and interpret CI failures. It can formulate a plan such as:

1. Modify the controller to accept pagination parameters.
2. Update the service layer to handle offsets.
3. Add integration tests.
4. Run the test suite.

5. Fix failing assertions.
6. Commit changes.

The plan may not be exposed as a visible list, but internally the agent operates with structured task decomposition rather than isolated responses. Traditional generative tools extend the IDE; agentic systems extend the workflow.

Tool orchestration is another important dimension. Modern agents can invoke shell commands, manipulate files, interact with Git, call APIs, and operate inside containers. Consider the difference between asking for a Dockerfile and having the system generate it, build the container, detect a missing dependency, modify the Dockerfile, rebuild, and confirm that the container starts. The latter requires integration with execution environments — exactly what Codex's sandboxed execution provides.

As these systems gain tools, memory becomes equally important. Memory here is not merely a larger context window; it is structured state. The agent must remember what it has attempted, which files changed, what errors occurred, and which sub-goals remain. In practice this appears as iterative behaviour. An agent runs a test suite, reads a failure, inspects the rendering logic, notices a misconfigured scale domain, updates the initialization, reruns, and confirms success. Each cycle refines the solution based on observed output.

This evolution affects organizational structure. When agents can scaffold projects, refactor repetitive patterns, and implement cross-cutting features, the human engineer's focus shifts from writing each function toward defining constraints and evaluating results. Prompt design becomes workflow design. Repository hygiene becomes machine-readability. Test coverage becomes an executable specification against which agents iterate.

Architecture must adapt too. Agents reason more effectively over modular, well-structured repositories. Clear separation of concerns, explicit interfaces, and deterministic builds improve reliability. When modules are predictable and responsibilities explicit, the agent can localize changes and reduce unintended side effects. Poorly structured systems amplify error propagation.

As we move forward, the central question is no longer whether agents can generate code. It is whether they can reliably participate in engineering processes at scale, under governance, within complex socio-technical systems.

Chapter 3 — Architecture of a Software Engineering Agent

If agentic engineering is a shift in workflow, then the architecture of the agent becomes as significant as the architecture of the application it modifies. A software-engineering agent is not simply a language model wrapped in an interface. It is a coordinated system composed of context ingestion, structured reasoning, tool invocation, state tracking, and feedback integration.

At a conceptual level, the agent operates in a loop: it perceives context, reasons about objectives, plans actions, executes those actions in a real environment, observes outcomes, and adapts. This perception–reasoning–execution–feedback cycle is not metaphorical; it is implemented through explicit components. Codex is a concrete instance of this architecture, and understanding how it maps to these layers makes the rest of the book practical rather than abstract.

Context ingestion

Given an objective such as "Add pagination to the episodes endpoint," the agent must first determine where the endpoint resides, which modules handle data access, whether pagination already exists, and what test coverage is present. Codex indexes the working directory, reads files on demand, and honours project-specific guidance placed in an AGENTS.md file (covered in Chapter 4). If Git is present, it can inspect history to understand recent changes. When version control is absent, its ability to reason about deltas is reduced.

Structured reasoning and planning

Once context is established, the agent decomposes the goal into sequenced actions. For pagination that might include modifying the controller to accept limit and offset, updating the data-access layer, adjusting response schemas, adding tests, running the suite, and fixing regressions. In Codex you can influence how much deliberation the model spends via the reasoning effort setting (minimal, low, medium, high, or xhigh), selected with the /model command or set in configuration. Higher effort suits complex, multi-file changes; lower effort suits quick edits.

Tool invocation and execution

Execution happens through tool orchestration. Unlike a purely generative system, the agent interacts with a real environment. It may run shell commands, create or modify files, run builds, and launch a development server:

```
npm install
npm test
git checkout -b feature/pagination
```

Crucially, these actions run inside a sandbox whose permissions you control (Chapter 4). Each action produces artifacts — logs, error messages, diffs — that become new perceptual inputs for the reasoning loop.

Imagine the test suite produces:

```
FAIL src/api/episodes.test.ts
  ● returns paginated results
    Expected length: 10
    Received length: 0
```

The agent reads this not as prose but as structured feedback indicating a failure in the data-retrieval path. It revisits the data-access layer, discovers that the offset is applied before filtering, patches the logic, and reruns the suite. The loop continues until the objective condition — passing tests — is satisfied or a blocking ambiguity requires human input.

Version control as a state subsystem

Version-control integration plays a particularly important role. Git is not merely storage; it becomes a state-tracking subsystem. Commits are checkpoints. Diffs are explicit change sets. Branches are alternative solution paths. An agent that understands Git can isolate its own modifications and present structured pull requests for review. Consider a diff generated after implementing pagination:

```
+ router.get("/episodes", async (req, res) => {
+   const limit = Number(req.query.limit) || 10;
+   const offset = Number(req.query.offset) || 0;
+   const episodes = await episodeService.getPaginated(limit, offset);
+   res.json(episodes);
+ });
```

This diff is an artifact embedded in repository history. The agent may create a branch, commit with a descriptive message, and open a pull request. Human oversight re-enters the loop: the engineer reviews the diff, checks architectural consistency, and decides whether to merge.

Multimodal perception and governance

Beyond code and logs, Codex can accept images — screenshots, annotated UI captures, design mockups — as input, mapping visual context to source code (Chapter 7). And architecture alone does not guarantee safe operation: governance must be embedded in the system. An agent with unrestricted file-system access and commit privileges poses risk. Responsible configurations therefore scope autonomy through the approval and sandbox controls described next. All actions are observable; audit trails remain inspectable; human-in-the-loop checkpoints are a structural requirement, not an optional add-on.

Chapter 4 — Project Setup: Installing and Configuring Codex

With the architecture established, we move from abstraction to execution. This chapter installs Codex, authenticates it, configures its permissions, and scaffolds a real project: an interactive analytics dashboard for a podcast dataset. Everything here is intended to be reproducible.

Note Version note (current as of 2026): the commands and model names in this book reflect the Codex CLI and the GPT-5-Codex model family as of mid-2026. Codex updates frequently. Rather than pinning older identifiers, prefer the current recommended model (at the time of writing, gpt-5.5) and let sensible defaults update. Always confirm specifics against the official documentation at developers.openai.com/codex.

Installing the Codex CLI

There are three supported ways to install the CLI. The official standalone installer is the simplest and ships a self-contained binary:

```
# macOS / Linux — official installer (recommended)
curl -fsSL https://chatgpt.com/codex/install.sh | sh
```

Alternatively, install via Homebrew on macOS, or via npm if you prefer the Node toolchain (npm requires Node.js 22 or later):

```
# macOS — Homebrew
brew install --cask codex

# Any platform with Node.js 22+
npm install -g @openai/codex
```

Note Correction from earlier drafts: the npm package is @openai/codex, but it is only one of three install paths, and it requires Node 22+. On Windows, Codex runs natively in PowerShell or under WSL2.

Authenticating

On first launch, Codex asks how you want to sign in. The recommended path is to sign in with your ChatGPT account, which draws on the Codex usage included with your plan; alternatively you can authenticate with an OpenAI API key billed through the platform.

```
codex        # launches the interactive TUI; prompts for sign-in on first run
# or explicitly:
codex login  # sign in with ChatGPT (opens a browser)
```

Codex usage is included with ChatGPT Plus, Pro, Business, Edu, and Enterprise plans (Enterprise may require administrator enablement). This authentication step — omitted from many quick-start walkthroughs — is mandatory; without it, no commands will run.

Understanding approval and sandbox modes

Before letting an agent touch your machine, understand how Codex constrains itself. Codex separates two concerns: what it is allowed to do without asking (the approval policy) and where its file-system and network access are confined (the sandbox). The named presets you will see in the interface are:

- Read Only — Codex can read files and answer questions but cannot edit or run commands without approval. This is the default in folders that are not Git repositories.
- Auto — Codex can read, edit, and run commands within the current workspace, but asks for approval before acting outside it or accessing the network. This is the default inside a Git repository.
- Full Access — Codex can read/write anywhere and use the network without prompts. Powerful and dangerous; reserve it for throwaway environments.

The same behaviour is available through explicit flags, which is how you drive Codex in scripts and CI:

```
--ask-for-approval <untrusted|on-request|on-failure|never>
--sandbox <read-only|workspace-write|danger-full-access>
--full-auto     # convenience: workspace-write with minimal prompting
```

The sandbox is enforced by the operating system, not merely by the model's good behaviour. On macOS, Codex runs commands under Apple's Seatbelt (sandbox-exec); on Linux it uses bubblewrap (bwrap) with seccomp filtering. This is why "the agent has commit privileges" need not mean "the agent can delete your home directory": least-privilege execution is a real, OS-level boundary.

Scaffolding the dashboard

Our dataset is a CSV of one hundred podcast episodes. Each record includes episode number, guest name, release date, download count, duration, and listener completion rate. The goal: let the host team explore trends, identify high-performing guests, analyse episode length, and track engagement through interactive visualizations.

The stack is modern and mainstream: React 19 for the UI, Vite for tooling and dev server, and D3 for custom SVG charts (rather than a prebuilt charting abstraction, to preserve flexibility). Create a working directory, initialise Git first — so Codex has version-control context and the Auto approval default — and launch Codex:

```
mkdir poddata && cd poddata
git init
codex
```

Note Correction from earlier drafts: initialise Git before starting, not after. Codex uses the repository as a state subsystem and enables its Auto mode inside a Git repo. Starting without Git triggers a warning that diff-tracking and rollback are limited — and needlessly runs your first edits with less context.

Now give Codex a high-level objective rather than implementation details:

> *"Create a React 19 + Vite project that analyses podcast metrics from data/data.csv. Use D3 for at least six meaningful charts. Ask clarifying questions if necessary."*

Codex performs contextual reasoning first — checking the directory, confirming no project exists, noting the Git repo — then decomposes the task: initialize the project, install dependencies, scaffold source directories, establish data-parsing utilities, create chart components, configure scripts, and verify the build. As it works, file-creation and command events stream to the terminal, and (in Auto mode) it pauses to request approval before steps that leave the workspace.

A correct Vite + React project layout looks like this. Note in particular where index.html lives:

```
poddata/
  index.html            <-- project ROOT (Vite entry HTML), NOT in public/
  package.json
  vite.config.js
  public/
    data/data.csv       <-- static assets served at /data/data.csv
  src/
    main.jsx
    App.jsx
    utils/parseCsv.js
    charts/DownloadsChart.jsx
    charts/DurationChart.jsx
    charts/GuestFrequencyChart.jsx
    charts/CompletionRateChart.jsx
```

Note Correction from earlier drafts: in a Vite project, index.html belongs at the project root, not in public/. Placing it in public/ is a Create-React-App convention and will not work with Vite. Vite's index.html references the entry module directly with <script type="module" src="/src/main.jsx">.

The generated package.json should target current, supported versions. React 19 and D3 7 are current; Vite has advanced well beyond version 5, so pin a recent major:

```
{
  "name": "poddata-dashboard",
  "version": "0.1.0",
  "private": true,
```

```
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^19.0.0",
    "react-dom": "^19.0.0",
    "d3": "^7.9.0"
  },
  "devDependencies": {
    "vite": "^8.0.0",
    "@vitejs/plugin-react": "^5.0.0"
  }
}
```

Note Correction from earlier drafts: the previous version pinned "vite": "^5.0.0". Vite 5 is several majors out of date; use a current release (8.x at the time of writing). React 19 and D3 7.9 remain current and are unchanged. Also declare "type": "module" and include @vitejs/plugin-react.

The correct Vite entry HTML and React bootstrap:

```
<!-- index.html (project root) -->
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Podcast Metrics Dashboard</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

```
// src/main.jsx
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

The dataset lives at public/data/data.csv so it is served at the URL /data/data.csv in both development and production. Loading and numeric coercion happen once, centrally, in App.jsx:

```
// src/App.jsx
import { useEffect, useState } from "react";
import * as d3 from "d3";

export default function App() {
  const [data, setData] = useState([]);

  useEffect(() => {
    d3.csv("/data/data.csv").then((parsed) => {
      setData(
        parsed.map((d) => ({
          episode: +d.episode,
          guest: d.guest,
          downloads: +d.downloads,
          duration: +d.duration,
          completion: +d.completion,
          releaseDate: new Date(d.releaseDate),
        }))
      );
    });
  }, []);

  return (
    <div>
      <h1>Podcast Metrics Dashboard</h1>
      {/* Charts will be rendered here */}
    </div>
  );
}
```

Start the dev server and confirm a clean baseline before adding visualization logic:

```
npm install
npm run dev
```

By the end of scaffolding the repository contains a functional React application, configured dependencies, initialized version control, central data-ingestion logic, and placeholder chart components. No visualization logic is implemented deeply yet, but the structural foundation is stable: the project builds, the server runs, the data loads. This stability is essential — agentic iteration works best from a clean, reproducible baseline.

Chapter 5 — Code Generation: From Requirements to Components

With the scaffold in place and the dev server running cleanly, the project transitions from structure to substance. The repository now contains a React application, dependency configuration, CSV ingestion, and version-control history. What it does not yet contain is meaningful visualization logic. This is where the agent converts abstract requirements into concrete components.

The original objective was intentionally high-level: "at least six meaningful charts." No chart types were specified. The agent must infer appropriate visualizations from the available fields. Download counts suggest a bar or line chart. Duration lends itself to distribution analysis. Guest frequency implies aggregation by category. Completion rate invites comparison across episodes. Release date suggests time-series trends.

Rather than embedding all logic in a single file, the agent decomposes responsibilities into components and utilities. It creates a charts/ directory, assigns one component per visualization, and keeps data transformation separate from rendering. The result is not a monolithic dashboard but a structured collection of reusable modules — which, as later chapters show, is exactly what makes safe agent-driven refactoring possible.

A representative chart component

The DownloadsChart component renders episode download counts as a bar chart, integrating React's lifecycle with D3's imperative DOM manipulation. The code below is idiomatic for React 19 + D3 7 and runs in the Vite project scaffolded in Chapter 4:

```
// src/charts/DownloadsChart.jsx
import * as d3 from "d3";
import { useEffect, useRef } from "react";

export default function DownloadsChart({ data }) {
  const svgRef = useRef(null);

  const width = 800;
  const height = 400;
  const margin = { top: 20, right: 20, bottom: 40, left: 50 };

  useEffect(() => {
    if (!data || data.length === 0) return;

    const svg = d3.select(svgRef.current);
    svg.selectAll("*").remove();

    const innerWidth = width - margin.left - margin.right;
    const innerHeight = height - margin.top - margin.bottom;

    const g = svg
      .append("g")
```

```
      .attr("transform", `translate(${margin.left},${margin.top})`);

    const x = d3
      .scaleBand()
      .domain(data.map((d) => d.episode))
      .range([0, innerWidth])
      .padding(0.2);

    const y = d3
      .scaleLinear()
      .domain([0, d3.max(data, (d) => d.downloads)])
      .nice()
      .range([innerHeight, 0]);

    g.selectAll("rect")
      .data(data)
      .enter()
      .append("rect")
      .attr("x", (d) => x(d.episode))
      .attr("y", (d) => y(d.downloads))
      .attr("width", x.bandwidth())
      .attr("height", (d) => innerHeight - y(d.downloads))
      .attr("fill", "#4e79a7");

    g.append("g")
      .attr("transform", `translate(0,${innerHeight})`)
      .call(d3.axisBottom(x));

    g.append("g").call(d3.axisLeft(y));
  }, [data]);

  return <svg ref={svgRef} width={width} height={height} />;
}
```

Several details are worth noting. Width and height are defined outside the effect hook so they are available both to the rendering logic and to the returned <svg> element — a common mistake in ad hoc D3 integrations is to declare dimensions inside useEffect and then reference them outside. The margin convention creates an inner drawing region, and .nice() on the Y scale rounds the domain to readable bounds. React is not used to render individual bars declaratively; instead the component lets D3 manage the SVG subtree inside the <g> element, which is the correct pattern when combining D3's data-binding model with React's lifecycle.

Note Practical note: with one hundred episodes, a band scale produces one hundred x-axis ticks, which crowds the axis. A production build would thin the tick labels, for example .call(d3.axisBottom(x).tickValues(x.domain().filter((_, i) => i % 10 === 0))). This is the kind of refinement a human reviewer should request — the code is correct but the presentation can be improved.

Aggregation and time-series charts

A GuestFrequencyChart groups records by guest name. d3.rollups returns an array of [key, value] entries, which map cleanly to chartable objects:

```
const grouped = d3.rollups(
  data,
  (v) => v.length,
  (d) => d.guest
).map(([guest, count]) => ({ guest, count }));
```

For time-series visualization, the agent parses release dates into JavaScript Date objects (already done centrally in App.jsx) and uses a time scale:

```
const x = d3
  .scaleTime()
  .domain(d3.extent(data, (d) => d.releaseDate))
  .range([0, innerWidth]);
```

Each chart follows the same structural template: isolate the SVG, clear the prior render, compute scales, bind data, render axes. Consistency across modules reflects repository-scale reasoning rather than one-off code emission. Inside App.jsx, the agent composes the dashboard, passing already-normalized data down as props so each chart avoids redundant parsing:

```
import DownloadsChart from "./charts/DownloadsChart";
import DurationChart from "./charts/DurationChart";
import GuestFrequencyChart from "./charts/GuestFrequencyChart";

// inside the returned JSX:
<DownloadsChart data={data} />
<DurationChart data={data} />
<GuestFrequencyChart data={data} />
```

Tests as an executable specification

Even when explicit test generation is not requested, the agent may scaffold minimal tests using Vitest (the natural choice in a Vite project). A rendering-integrity test confirms the component mounts with valid props:

```
// src/charts/DownloadsChart.test.jsx
import { render } from "@testing-library/react";
import { describe, it } from "vitest";
import DownloadsChart from "./DownloadsChart";

describe("DownloadsChart", () => {
  it("renders without crashing", () => {
    const sample = [
      { episode: 1, downloads: 120, duration: 45, completion: 0.7 },
    ];
```

```
    render(<DownloadsChart data={sample} />);
  });
});
```

Note This test needs a DOM environment. In vite.config.js set test: { environment: "jsdom" } and install @testing-library/react and jsdom as dev dependencies. The purpose is structural assurance, not visual validation; richer SVG assertions can be layered later.

What distinguishes this from a conventional tutorial is not the code itself but the transition from a structured plan to an integrated implementation. The agent examined repository context, reused parsing utilities, normalized data centrally, created modular components, and preserved consistent patterns. Each component becomes a node in a coherent system rather than a disconnected artifact.

Chapter 6 — Refactoring, Feature Addition, and Diffs

Once the dashboard renders correctly, the next phase is not expansion but evolution. Real engineering rarely consists of writing greenfield modules in isolation. More often it involves modifying existing structures, introducing new behaviour into stable systems, and ensuring enhancements integrate cleanly. In this chapter the feature request is deceptively simple:

> *"Add zoom and pan behaviour to all charts so users can drag to pan and use the mouse wheel to zoom in and out."*

What appears to be a minor UI enhancement is, in practice, a cross-cutting concern. It affects chart rendering, SVG grouping structure, and event handling, and it touches every visualization component. It therefore provides a realistic demonstration of agent-driven refactoring.

The agent begins not by editing files but by re-evaluating repository context: the structure of each chart, how SVG elements are organized, whether there is a consistent <g> container pattern, and whether dimensions are hard-coded. Zoom in D3 works by applying a transform to a container element. In the current implementation, bars and axes are appended directly inside a single <g> with margin translation applied. For zoom to work correctly, the chart must distinguish between a static outer container and a transformable inner content group.

Rather than duplicating logic across components, the agent extracts a reusable utility in src/utils/zoomBehavior.js that operates on a specific content group and accepts dimensions explicitly:

```
// src/utils/zoomBehavior.js
import * as d3 from "d3";

export default function applyZoom(svg, contentGroup, width, height) {
  const zoom = d3
    .zoom()
    .scaleExtent([1, 8])
    .translateExtent([[0, 0], [width, height]])
    .extent([[0, 0], [width, height]])
    .on("zoom", (event) => {
      contentGroup.attr("transform", event.transform);
    });

  svg.call(zoom);
  return zoom;
}
```

This uses the D3 v6+ zoom API, in which the handler receives the event object and reads event.transform. The function takes both the root svg selection and the inner contentGroup, avoiding fragile assumptions about selectors, and constrains the zoom and translate extents to the drawing region. It is now reusable and deterministic.

To support a transformable content group while preserving axis clarity, the agent restructures DownloadsChart's grouping hierarchy. A rootGroup handles margins and axis positioning; a contentGroup nested inside holds only the elements that should scale and translate under zoom:

```
// key structural change inside the useEffect
const rootGroup = svg
  .append("g")
  .attr("transform", `translate(${margin.left},${margin.top})`);

const contentGroup = rootGroup.append("g");

// bars now render into contentGroup (not directly into a single g)
contentGroup.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("x", (d) => x(d.episode))
  .attr("y", (d) => y(d.downloads))
  .attr("width", x.bandwidth())
  .attr("height", (d) => innerHeight - y(d.downloads))
  .attr("fill", "#4e79a7");

// axes attach to rootGroup so they remain visually anchored
rootGroup.append("g")
  .attr("transform", `translate(0,${innerHeight})`)
  .call(d3.axisBottom(x));
rootGroup.append("g").call(d3.axisLeft(y));

// behaviour attached last, after elements exist
applyZoom(svg, contentGroup, width, height);
```

The structural change is subtle but critical. The rootGroup handles margins and axes; the contentGroup contains only elements that should scale and translate. Because the contentGroup starts with no transform of its own, the zoom handler can safely overwrite its transform without disturbing the margins on rootGroup. From a version-control perspective, the change appears as a clear, coherent diff rather than scattered edits:

```
+ import applyZoom from "../utils/zoomBehavior";

- const g = svg.append("g")
-   .attr("transform", `translate(${margin.left},${margin.top})`);
+ const rootGroup = svg.append("g")
+   .attr("transform", `translate(${margin.left},${margin.top})`);
+ const contentGroup = rootGroup.append("g");

- g.selectAll("rect")
+ contentGroup.selectAll("rect")

+ applyZoom(svg, contentGroup, width, height);
```

Because the agent has Git context, it commits with a descriptive message:

```
feat(charts): add reusable zoom/pan behaviour with content-group separation
```

The agent then repeats the pattern for the other components. It does not blindly paste code: it verifies whether each chart already has a grouping structure, and because the zoom utility transforms the content group generically, it remains compatible with a time scale or a linear scale. After implementing the feature, the agent runs the dev server and validates that scroll-wheel zoom scales the chart, click-and-drag pans, axes remain anchored, and no JavaScript errors appear in the console.

Human oversight remains indispensable. Performance must be evaluated: with large datasets, zoom transforms can cause rendering lag, and the reviewer must decide whether to introduce throttling or canvas-based rendering. Agentic refactoring accelerates execution, but architectural judgment remains human. Because the logic is centralized in applyZoom, future modifications — such as double-click reset — require editing only one file.

Chapter 7 — Visual Inputs and Screenshot-Driven Modifications

Until now the agent has operated primarily on textual inputs: prompts, repository files, terminal logs, and diffs. But Codex is multimodal — it can accept images such as screenshots, annotated UI captures, and design mockups, and integrate them into the engineering loop. In the CLI you attach an image by referencing its path (for example, dragging a file into the terminal or passing its path), and in the IDE extension you can paste or attach a screenshot directly.

This capability changes how user-interface changes propagate from intent to implementation. In conventional workflows, a designer produces a mock-up, a developer interprets it, edits the JSX or CSS, rebuilds, and verifies. The translation from visual intent to code depends entirely on human interpretation. With a multimodal agent, that translation becomes partially automated: the agent perceives the visual hierarchy, associates elements with component boundaries, and maps them back to source files. The loop remains the same — perceive, plan, act, verify — but perception now includes pixels.

Consider the dashboard's homepage, which renders a headline followed by a short descriptive paragraph:

```
<div className="container">
  <h1>Podcast Metrics Dashboard</h1>
  <p className="intro">
    This dashboard explores episode performance and guest behavior.
  </p>
  <DownloadsChart data={data} />
</div>
```

A stakeholder decides the introductory paragraph is unnecessary. They take a screenshot of the running application, draw an arrow to the paragraph, and annotate it: "Remove this text paragraph." The agent receives the annotated image along with repository context. Its task is not merely to delete a line; it must determine which component rendered the highlighted element, whether the element is static or generated dynamically, whether removal has layout implications, and whether other components depend on its presence.

The agent correlates visible text in the screenshot with strings in the codebase, locates the literal sentence in src/App.jsx, verifies the <p className="intro"> element is not conditionally rendered or reused, and generates a structured patch:

```
--- a/src/App.jsx
+++ b/src/App.jsx
@@ export default function App() {
   return (
     <div className="container">
       <h1>Podcast Metrics Dashboard</h1>
-      <p className="intro">
-        This dashboard explores episode performance and guest behavior.
-      </p>
       <DownloadsChart data={data} />
```

```
    </div>
  );
```

The agent applies the patch, triggers the dev server if necessary, and verifies the build remains clean. With hot module reloading active, the preview updates immediately and the paragraph disappears. What previously required manual file search and editing is reduced to a perceptual mapping followed by a deterministic diff.

More nuanced requests require deeper reasoning. Suppose a designer circles a chart title and writes, "Move this above all charts and centre it." The agent must identify that the title currently lives inside a chart component, extract it into a parent container, update layout styling, and verify that other charts are unaffected. Or consider an annotation that reads, "Make this bar colour reflect completion rate." Now the visual change implies data binding:

```
- .attr("fill", "#4e79a7");
+ .attr("fill", (d) => d3.interpolateBlues(d.completion));
```

The agent must confirm that completion exists in the parsed dataset and is normalized between 0 and 1; if not, it must adjust parsing logic. A seemingly aesthetic request can cascade into data-layer adjustments. Screenshot-driven modification is powerful but not infallible: ambiguous annotations can produce incorrect mappings, dynamically generated SVG elements may have no direct JSX to modify, and visual changes can introduce side effects such as altered spacing or reduced contrast. Best practice is to treat visual modifications as semi-autonomous operations: the agent executes, the human verifies. As always, these operations should run on a branch and produce reviewable commits rather than bypassing version control.

Chapter 8 — IDE Integration and Interactive Engineering

So far the agent has operated primarily through the CLI. Effective as that is, the integrated development environment remains the centre of gravity for most engineers — it is where code is read, modified, reviewed, debugged, and committed. Codex meets developers there through an IDE extension for Visual Studio Code and its forks (including Cursor and Windsurf) and for JetBrains IDEs (IntelliJ, PyCharm, WebStorm, and others).

Installed, Codex appears not as a separate terminal session but as a sidebar panel alongside the editor. The file tree remains visible. The diff viewer remains available. The integrated terminal is still at the bottom. What changes is that a new participant has entered the workspace — one that can scan the repository, edit files, run commands, and report progress in real time. The experience shifts from "invoke the agent" to "work with the agent."

In a typical session the developer opens the dashboard project. The Codex panel offers a conversational interface; the editor shows the files; the terminal sits below. Repository context is available automatically — no manual copying of files into prompts is required — and you can pull specific files into context with an @-reference (for example, @src/charts/DownloadsChart.jsx). The developer types:

> *"Add hover tooltips to all charts showing episode number and downloads."*

The agent does not respond with a snippet alone. It scans the src/charts directory, identifies files containing D3 draw logic, evaluates whether a tooltip abstraction already exists, and — finding none — plans a shared utility. Inline diffs appear in the editor; insertions are highlighted, deletions marked, and the developer can expand each change to inspect context. Simultaneously, the integrated terminal runs the dev server, and if the build fails, the error surfaces in the panel for the agent to interpret and correct.

Calibrating autonomy

The extension exposes the same approval controls as the CLI, letting teams calibrate how much the agent does on its own. In a chat-oriented mode the agent behaves as an advisor: it explains, suggests patterns, and proposes refactors, but the developer applies changes. In an agent mode it performs multi-file edits but pauses before irreversible operations, prompting for confirmation. In a full-access mode it operates with expanded authority — applying changes, running builds, and committing — with human involvement shifting toward review. The existence of these modes reflects an important principle: agentic engineering is not binary. Autonomy is adjustable, and the model and reasoning effort are switchable from within the editor as well.

To make interaction concrete, revisit the zoom-and-pan feature, this time from inside the IDE. The developer types a request; a new file appears in the tree (src/utils/zoomBehavior.js), rendered in the editor and highlighted as newly created; inline diffs appear in DownloadsChart.jsx. If something looks wrong — perhaps the zoom extent is too restrictive — the developer comments directly in the panel:

> *"Limit zoom scale to 4x instead of 8x."*

The agent updates the utility accordingly. After applying edits across all relevant components, it triggers the dev server, the preview refreshes, and the charts respond to scroll and drag. Finally the agent offers to create a branch and commit. A single confirmation produces a clean feature branch and structured commit history — all without leaving the IDE. For long-running jobs, the extension can also delegate work to Codex Cloud and let you monitor it from the editor, a capability we use in the next chapter.

This integration introduces new cognitive patterns. Engineers must learn to review larger diffs efficiently; instead of writing twenty lines they may review two hundred generated lines, moving from syntactic construction to structural validation. Debugging also changes character: when a bug emerges in generated code, the developer must understand not only the surface implementation but the abstraction the agent introduced. Agentic productivity increases velocity, but it does not eliminate the need for comprehension — and it rewards clean, well-named, modular repositories, in which the agent can localize changes confidently.

Chapter 9 — Cloud Execution, Parallel Variants, and A/B Engineering

Local CLI and IDE sessions assume a single working copy and a largely linear progression. Codex Cloud dissolves that linearity. Accessible at chatgpt.com/codex and delegatable from the web, the IDE extension, or GitHub, it runs agents in isolated, OpenAI-managed containers — each a sandboxed universe with its own file system, its own branch, and its own build artifacts. You can configure the environment (repository, setup scripts, available tools), optionally grant internet access, run multiple tasks in parallel, and have the agent open pull requests directly.

When you submit a feature request in cloud mode, the sequence differs from local execution. The environment provisions a container, clones the repository, installs dependencies, and creates a working branch. Because containers are independent, the agent can explore multiple implementation strategies at once. Suppose the request is:

> *"Implement hover tooltips showing episode number and downloads."*

The agent may create separate branches and implement each independently inside its own container:

```
feature/hover-tooltips-v1
feature/hover-tooltips-v2
```

Variant 1 implements tooltips using a simple absolutely positioned overlay managed through React state. Variant 2 embeds tooltip rendering directly in D3, using pointer events and SVG-aware positioning to ensure compatibility with the zoom behaviour introduced earlier. Both are built and tested automatically:

```
npm install
npm run build
npm test
```

Logs are captured in full. If one variant fails to compile, its container is marked unsuccessful without affecting the other. In Variant 1, the agent creates a Tooltip component:

```
// src/charts/Tooltip.jsx
export default function Tooltip({ x, y, content }) {
  return (
    <div
      style={{
        position: "absolute",
        left: x,
        top: y,
        background: "#333",
        color: "#fff",
        padding: "4px 8px",
        borderRadius: "4px",
```

```
        pointerEvents: "none",
      }}
    >
      {content}
    </div>
  );
}
```

Each chart tracks mouse position and renders the tooltip conditionally. The approach is React-centric but may conflict with D3 transforms during zoom. In Variant 2, the agent embeds tooltip logic directly in D3, binding events to the content group so tooltips respect existing transforms:

```
contentGroup.selectAll("rect")
  .on("mouseover", function (event, d) {
    tooltip
      .style("opacity", 1)
      .html(`Episode ${d.episode}<br/>Downloads: ${d.downloads}`);
  })
  .on("mousemove", function (event) {
    tooltip
      .style("left", `${event.pageX + 10}px`)
      .style("top", `${event.pageY - 20}px`);
  })
  .on("mouseout", function () {
    tooltip.style("opacity", 0);
  });
```

The agent presents a structured comparison — files modified, architecture notes, compatibility with zoom — and can deploy preview builds for each variant so the developer can open both side by side and interact with them. The developer evaluates the live previews and selects Variant 2 for correct alignment during zoom. What was previously a single attempt has become structured parallelism: multiple solutions explored without sequential developer time.

Once a variant is chosen, the agent pushes the selected branch and opens a pull request with a structured summary. Continuous-integration pipelines trigger; tests run in isolated CI environments; linting and formatting checks execute; and, if configured, performance benchmarks detect regressions in render time or bundle size. A typical status might resemble:

```
CI Status: Passed
Tests: 42 passing
Lint: No issues
Bundle size delta: +3.2 KB
```

Cloud-based workflows enable not only parallel development but live experimentation. Instead of choosing a variant subjectively, a team may deploy both to segmented user traffic, route cohorts to each implementation, and collect metrics such as hover-activation rate or interaction latency. This power must be governed carefully: unrestricted variant generation risks branch proliferation and review fatigue, and each container consumes compute. Clean branching conventions and

clear archival or deletion of rejected variants keep repositories disciplined. A/B engineering is only as effective as the quality of its measurement framework; if success metrics are ambiguous, the agent may optimize for the wrong signal.

Chapter 10 — Automated Code Review and Pull Request Workflows

By this stage the agent has scaffolded projects, implemented features, refactored code, interpreted screenshots, worked inside the IDE, and generated parallel variants. The next evolution is subtler but equally significant: the agent begins to evaluate code rather than merely produce it. Codex provides this both as a local review agent and, more prominently, as a GitHub integration.

GitHub code review

With the Codex GitHub app connected to a repository, you can trigger a review by commenting on a pull request, or enable automatic review of every new PR. Codex reads the diff in the context of the whole repository and posts a standard GitHub review, flagging high-priority issues and honouring any review guidance you place in AGENTS.md:

```
# in a PR comment
@codex review

# follow-up, after reviewing the findings
@codex fix the P1 issue
```

The agent scans the diff holistically. It examines structural consistency (are imports aligned with repository conventions? are utilities reused rather than duplicated?), performance implications (does the new tooltip implementation attach excessive event listeners or trigger unnecessary re-renders?), and accessibility (are ARIA attributes missing? is keyboard interaction preserved?). It then annotates the pull request with inline comments grounded in repository context rather than generic warnings.

Suppose the selected tooltip implementation inadvertently layers an invisible SVG rectangle above the chart points, intercepting pointer events. The diff might include:

```
svg.append("rect")
  .attr("fill", "transparent")
  .attr("width", width)
  .attr("height", height);
```

The agent attaches a comment directly to this block: "The transparent overlay is rendered after chart elements and may intercept pointer events, preventing tooltips from triggering. Consider appending this overlay before rendering data points, or setting pointer-events: none." This is not a generic lint rule; it is behavioural reasoning grounded in how event propagation works in SVG layering.

Automated review extends beyond static diff analysis. Once the pull request exists, the agent (or your CI) can orchestrate the full pipeline: builds, test suites, linters, static analysis, and dependency-vulnerability checks. For example:

```
Build: Success
Tests: 42 passed, 0 failed
Lint: No warnings
Bundle size delta: +3.2 KB
Dependency scan: No new vulnerabilities detected
```

If a test fails and the failure is localized and deterministic, the agent may push a corrective commit to the same branch, re-run CI, and update the PR. In more advanced configurations, policy gates apply: if the diff touches a sensitive module — authentication logic, payment processing, infrastructure configuration — it is flagged as requiring mandatory human review, and merge automation is restricted to low-risk changes. The agent does not replace governance; it encodes it.

A full review loop

7. The tooltip feature branch is pushed and a pull request opened.
8. CI executes and passes.
9. Codex posts two inline comments: one about pointer-event layering, another suggesting memoization of a frequently recalculated scale.
10. The developer addresses the pointer-event issue but declines the memoization suggestion, noting the dataset is capped at 100 rows.
11. The agent re-evaluates the updated diff and confirms no regressions remain.
12. A "Ready to merge" label is applied.

This loop illustrates an important principle: the agent's review is advisory but reasoned. Humans can accept, modify, or override suggestions; the interaction resembles peer review rather than command execution. Once approval criteria are satisfied, the agent may merge, tag the commit, and update changelog documentation:

```
git merge feature/hover-tooltips-v2
git tag v1.3.0
```

```
## v1.3.0
- Added D3-integrated hover tooltips compatible with zoom/pan.
- Improved pointer-event layering.
```

Every action is logged. Audit records include timestamp, triggering event, files modified, CI results, and merge outcome. In regulated environments, such logs become compliance artifacts. Automation without traceability would be reckless; automation with traceability becomes disciplined acceleration. As agents assume review authority, effective systems embed risk classification and confidence scoring — based on test-coverage presence, diff size, similarity to previously validated changes, and the absence of security-sensitive files — so that low-confidence

changes escalate to manual review while high-confidence changes may proceed under predefined policy.

Chapter 11 — Configuring Codex for a Repository (AGENTS.md, config.toml, MCP)

By the time agents are building features, refactoring code, and reviewing pull requests, a practical question emerges: how do you steer an agent's behaviour for a specific repository and team? Codex answers this with two configuration mechanisms — project-level guidance in AGENTS.md and machine-level settings in config.toml — plus support for the Model Context Protocol (MCP) to extend the agent with external tools. This chapter is the hands-on complement to the architectural discussion of Chapter 3.

AGENTS.md: project instructions

AGENTS.md is a plain-Markdown file that Codex reads before doing work. It is where you record conventions the agent should follow: how to run the build and tests, coding standards, directory responsibilities, and review guidance. You can generate a starter file from inside the interactive session with the /init command. Files are discovered hierarchically — a global file in ~/.codex/AGENTS.md, a repository-root AGENTS.md, and nested AGENTS.md files in subdirectories — and concatenated root-first, so guidance closer to the working directory takes precedence. A minimal example for the dashboard:

```
# AGENTS.md

## Stack
- React 19 + Vite, D3 7 for charts. Package manager: npm.

## Commands
- Install: `npm install`
- Dev server: `npm run dev`
- Tests: `npm test` (Vitest + jsdom)
- Build: `npm run build`

## Conventions
- One chart component per file under src/charts/.
- Parse and normalize CSV once in App.jsx; pass data down as props.
- Shared logic goes in src/utils/. Do not duplicate utilities.

## Review guidance
- Flag accessibility regressions (missing ARIA, lost keyboard nav).
- Treat anything under src/auth/ as sensitive: require human review.
```

config.toml: machine and profile settings

While AGENTS.md describes the project, ~/.codex/config.toml describes how the CLI itself behaves. Common options include the model, reasoning effort, default approval policy, and sandbox mode. Named profiles let you keep several configurations side by side — for example, a cautious default and a more autonomous profile for throwaway environments:

```
# ~/.codex/config.toml
model = "gpt-5.5"
model_reasoning_effort = "medium"   # minimal | low | medium | high | xhigh
approval_policy = "on-request"      # untrusted | on-request | on-failure | never
sandbox_mode = "workspace-write"    # read-only | workspace-write | danger-full-access

[profiles.autonomous]
approval_policy = "never"
sandbox_mode = "workspace-write"
```

Configuration precedence runs from most to least specific: explicit CLI flags override project-scoped config, which overrides the selected profile, which overrides the user config, which overrides built-in defaults. In practice, keep durable preferences in config.toml and use flags for one-off deviations.

Extending Codex with MCP servers

Codex supports the Model Context Protocol, an open standard for connecting agents to external tools and data sources. MCP servers are declared in config.toml. A stdio server is launched by a command; a streamable-HTTP server is reached by URL:

```
# ~/.codex/config.toml
[mcp_servers.docs]
command = "npx"
args = ["-y", "some-mcp-server"]
env = { API_KEY = "..." }

[mcp_servers.internal_api]
url = "https://mcp.example.com/sse"
enabled = true
```

MCP configuration is shared across Codex surfaces — CLI, IDE, and desktop app — so a server you add once is available wherever you work. Because MCP servers can grant the agent new capabilities, treat them with the same least-privilege discipline as any other tool: enable only what a task needs, and prefer servers you control or trust.

Why structure amplifies agents

These mechanisms make explicit a theme running through the book: agentic effectiveness correlates with structural clarity. When an agent scans a repository it relies on structural signals — directory organization, naming consistency, import graphs, test boundaries, dependency declarations. Predictability matters more than cleverness. A repository that is modular, well-documented, instrumented, and governed becomes fertile ground for agents; a tangled, inconsistent one resists them. Configuration files turn that clarity from an implicit stylistic choice into an explicit, machine-readable contract.

Chapter 12 — Deploying the Dashboard

The book's title promises deployment, and this chapter delivers it concretely. Having built, refactored, and reviewed the dashboard, we now ship it. Because the project is a standard Vite build, it produces static assets that can be hosted almost anywhere — a static host, a container, or a CI-driven pipeline. We will cover all three, and show how Codex participates in each.

Producing a production build

Vite compiles the application into an optimized, static bundle in the dist/ directory. Confirm the build and preview it locally before deploying:

```
npm run build      # outputs static assets to dist/
npm run preview    # serves the production build locally for a final check
```

Because the CSV lives under public/data/, Vite copies it verbatim into dist/data/data.csv, so the deployed app fetches /data/data.csv exactly as it did in development. This is the payoff of placing static assets in public/ from the start.

Option A — Static hosting (Vercel or Netlify)

For a client-side app, a static host is the simplest path. Both Vercel and Netlify detect a Vite project automatically; the build command is npm run build and the publish directory is dist. From the terminal you can deploy directly:

```
# Vercel
npm i -g vercel
vercel            # first run links the project; follow prompts
vercel --prod     # promote to production

# Netlify
npm i -g netlify-cli
netlify deploy --build --prod --dir=dist
```

A single-page app that uses client-side routing needs a rewrite rule so deep links resolve to index.html. For Netlify, add a public/_redirects file (copied into dist/ by Vite):

```
/*    /index.html   200
```

For Vercel, an equivalent rewrite lives in vercel.json at the project root:

```
{
  "rewrites": [{ "source": "/(.*)", "destination": "/index.html" }]
}
```

Option B — Containerized deployment (Docker)

To "deploy anywhere" in the literal sense, package the app as a container. A multi-stage Dockerfile builds the static bundle in a Node image, then serves it from a small Nginx image:

```
# Dockerfile
# ---- build stage ----
FROM node:22-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# ---- serve stage ----
FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

To support client-side routing under Nginx, add a small config that falls back to index.html, and copy it in the serve stage:

```
# nginx.conf
server {
  listen 80;
  location / {
    root /usr/share/nginx/html;
    try_files $uri $uri/ /index.html;
  }
}
```

```
# add to the serve stage of the Dockerfile
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

Build and run the container locally, then push the image to any registry and run it on any container platform:

```
docker build -t poddata-dashboard .
docker run -p 8080:80 poddata-dashboard   # visit http://localhost:8080
```

Option C — Continuous deployment with GitHub Actions and Codex

For repeatable delivery, wire the build into CI so every merge to main deploys automatically. A minimal workflow builds the project and uploads the artifact (here, to GitHub Pages; swap the final step for your host):

```
# .github/workflows/deploy.yml
name: Deploy
```

```
on:
  push:
    branches: [main]
permissions:
  contents: read
  pages: write
  id-token: write
jobs:
  build-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: 22 }
      - run: npm ci
      - run: npm run build
      - uses: actions/upload-pages-artifact@v3
        with: { path: dist }
      - uses: actions/deploy-pages@v4
```

Note If your app is served from a subpath (as GitHub Pages project sites are), set Vite's base option in vite.config.js — for example base: "/poddata-dashboard/" — so asset URLs resolve correctly. On a root domain (Vercel, Netlify, a container) the default base of "/" is correct.

Codex fits naturally into this pipeline. You can add the official Codex GitHub Action or the @codex review integration from Chapter 10 so that every pull request is reviewed before it can trigger a deploy, and you can ask Codex to generate or fix the workflow file itself. Deployment then becomes part of the same governed, agent-assisted loop as development: a change is proposed on a branch, reviewed by Codex and a human, merged, and shipped automatically — with every step logged and reversible.

Rollback and safety

Deploy pipelines should be reversible. Tag releases so you can redeploy a known-good version; keep previous container images in your registry; and, for progressive delivery, use your host's blue-green or canary features to shift traffic gradually while monitoring error rates. If Codex is permitted to merge and deploy low-risk changes automatically, pair that autonomy with automated rollback triggers so a spike in errors reverts to the last stable release without waiting for manual intervention.

Chapter 13 — Metrics, Evaluation, and Productivity

Once an organization adopts agentic workflows, the question inevitably arises: is this actually making us better? Productivity in software engineering has always been difficult to measure, and the agentic era complicates it further. When code is generated partly by machines, lines of code lose meaning. When agents open pull requests autonomously, commit counts cease to represent effort. When review cycles shorten but diff sizes grow, superficial metrics mislead.

Productivity in the agentic era is best framed as value delivered per unit of combined human–agent effort while maintaining or improving quality. That definition forces measurement across several dimensions: delivery speed, quality and reliability, human cognitive load, economic efficiency, and business impact. No single number captures this; a system of measurement is required.

The industry already possesses mature frameworks that agentic workflows extend rather than replace. The DORA metrics — deployment frequency, lead time for changes, change-failure rate, and time to restore service — remain foundational. If agent adoption reduces lead time and change-failure rate simultaneously, the signal is strong; if deployment frequency rises but failure rate also rises, caution is warranted. The SPACE framework — satisfaction, performance, activity, communication, and efficiency — adds the human dimension: engineers relieved from repetitive boilerplate often report improved focus, but if cognitive load rises from constant review of machine-generated code, satisfaction may decline.

Beyond these, agentic workflows introduce new operational variables. Teams can track the proportion of commits authored by agents versus humans (a high ratio may indicate effective delegation — or over-reliance), the human override rate on agent suggestions (high override rates point to prompt or repository-clarity problems), the rollback rate of agent-merged changes, and variant count in parallel workflows (many experimental branches with few meaningful evaluations signals unsustainable review overhead).

One of the clearest measurable signals is human-time reallocation. Engineers frequently report saving hours per week by delegating repetitive tasks — test scaffolding, formatting, dependency updates. But the important question is not how much time is saved; it is how that time is used. If saved time is reinvested into architectural improvement, performance, and technical-debt reduction, system health improves. If it is consumed reviewing low-quality output, gains evaporate.

Measurement should be longitudinal rather than snapshot-based. Early adoption often increases pull-request size as the agent modifies multiple files coherently; review time per PR may rise even as feature cycle time falls. Over time, as teams refine prompts and repository structure, diff sizes normalize. Compare metrics quarterly rather than weekly to avoid overreacting to short-term fluctuations, and examine correlations: does reduced human coding time correlate with improved test coverage? does increased experimentation correlate with lower defect-escape rates?

Overemphasis on a single metric distorts incentives. Optimizing purely for throughput yields excessive variants; measuring only human hours saved neglects quality; tracking only deployment frequency encourages fragmented changes. A disciplined adoption process begins with a baseline — current lead time, failure rate, review duration, developer satisfaction — captured before integration, then introduces agent workflows gradually and monitors change over several sprints. A mature measurement system evaluates delivery speed, system reliability, human cognitive impact, economic efficiency, and business outcomes together. Resource efficiency also becomes non-trivial at scale: monitoring token usage per feature, compute cost per experiment, and container runtime keeps automation economically sustainable. When measured thoughtfully, agentic workflows reveal their true value not in lines of code generated but in systems delivered faster, safer, and with greater strategic leverage.

Chapter 14 — Safety, Governance, and Ethical Considerations

As agentic systems transition from assistants to active participants, the risk profile of development changes fundamentally. A model that suggests code is one thing; a model that modifies repositories, executes shell commands, merges branches, and triggers deployments is something else. The difference is autonomy, and autonomy introduces both leverage and new risk surfaces.

Classes of risk

Four classes of risk deserve particular attention. Goal-interpretation drift: an agent receiving a high-level instruction such as "improve dashboard rendering performance" may memoize components, refactor pipelines, or alter rendering strategies — each locally reasonable yet potentially divergent from architectural intent. Scale of modification: an agent can alter dozens of files in seconds, so the blast radius of an incorrect assumption grows proportionally. Tool-chain execution risk: agents that invoke shell commands or manage dependencies expand the operational attack surface; a poorly constrained agent could run destructive commands or introduce insecure packages. And review opacity: a large, well-structured diff can appear trustworthy, tempting reviewers to accept it without reconstructing the underlying reasoning — trust without verification becomes a systemic vulnerability.

These risks do not invalidate agentic engineering; they demand structured governance. Fortunately, Codex's design provides concrete levers for each, which earlier chapters introduced and this one ties together.

Least-privilege execution

A safe environment enforces least-privilege access. As covered in Chapter 4, Codex runs commands inside an OS-enforced sandbox — Seatbelt on macOS, bubblewrap with seccomp on Linux — and separates the approval policy from the sandbox scope. Keep the default read-only or workspace-write sandbox for normal work and reserve danger-full-access for disposable environments. Dangerous operations such as recursive deletion or direct production-database modification should be disallowed by policy, not merely discouraged. Dependency management is another concern: packages introduced by an agent should pass vulnerability scanning and license checks automatically in CI.

Accountability and traceability

When a change is authored by an agent, responsibility must remain human. The organization deploying the agent, and the engineers supervising it, retain accountability for outcomes, and this principle should be reflected in workflow design. Pull-request templates can require explicit human sign-off for high-impact modules. Branch-protection rules can prevent automatic merging in directories related to authentication, billing, or infrastructure. CI can enforce mandatory review when diff size exceeds a threshold. Every agent action — from prompt to plan to command

execution to merge — should be logged, with audit records capturing timestamps, affected files, CI results, and approving reviewers. In regulated environments, this documentation becomes a compliance artifact. Agent autonomy without traceability is operationally irresponsible.

Human-in-the-loop, calibrated

Human-in-the-loop does not mean micromanaging every action; it means defining thresholds at which human review becomes mandatory. For low-risk UI changes, automated merging after passing tests may be acceptable. For architectural refactors or security-sensitive modules, human approval must be non-negotiable. This can be implemented technically through branch-protection rules requiring reviews for specified paths, CI checks that classify changes by impact domain, and automated labelling that escalates high-risk modifications. Because agents introduce changes rapidly, monitoring must operate at equivalent speed: post-deployment telemetry should be integrated tightly with merge automation so that if error rates spike after an agent-merged change, rollback triggers automatically or demands immediate human review.

Bias, ethics, and human dignity

Bias in engineering workflows manifests differently than in text generation — in algorithm selection, data-filtering logic, or presentation choices. An agent modifying a recommendation pipeline may inadvertently amplify biases in historical data; a change to sorting logic could affect which content receives prominence. Teams should conduct bias audits for features that affect user visibility, resource allocation, or decision-making, holding agent-generated code to the same fairness and accessibility standards as human-authored code. Organizations also have a responsibility to train engineers in prompt design, architectural governance, and agent supervision: blind delegation erodes mastery and increases systemic fragility. Equitable access to agentic tools matters too, so productivity gains are shared rather than concentrated. Human dignity in engineering means preserving meaningful agency, judgment, and accountability even as automation expands. The objective is not maximal restriction nor maximal autonomy; it is calibrated collaboration. Safety, transparency, accountability, and ethical alignment must be treated as core architectural requirements — not afterthoughts layered onto an automated pipeline.

Chapter 15 — The Future of Agentic Engineering

Throughout this book we have used a single engineering agent — Codex — to scaffold a project, refactor it, interpret screenshots, review pull requests, run parallel variants in the cloud, and deploy the result. Yet what lies ahead is not merely more capable single agents. The real inflection point arrives when multiple specialized agents collaborate across the entire lifecycle of software delivery.

Today's workflows typically centre on one primary model operating within a tool-enabled loop. In the near future we should expect specialization: different agents assuming different responsibilities, coordinated through orchestration layers that manage state, memory, and policy constraints. Imagine a pipeline in which a feature begins not with a human typing code but with a planning agent ingesting backlog items and producing a structured implementation roadmap. A code-generation agent translates that plan into scaffolding and modules. A test agent synthesizes unit and integration tests from contract definitions. A review agent evaluates diffs for architectural compliance and risk. A deployment agent pushes changes through staged environments. A monitoring agent observes runtime behaviour and feeds insights back into the system. These agents share context through persistent memory and communicate via defined protocols rather than ad hoc prompts — an architecture resembling a distributed system of reasoning components rather than a single conversational interface. The building blocks already exist: multi-surface execution, MCP-based tool integration, and cloud orchestration are steps along this path.

As these workflows mature, they will cease to be optional enhancements and become embedded in enterprise platforms. Development environments will include native agent orchestration. CI/CD systems will expose structured interfaces for agent-triggered workflows. Cloud providers will offer first-class agent execution with secure sandboxing and policy control. In such an environment, the boundary between "developer tool" and "intelligent collaborator" dissolves; the agent is not a plugin but part of the development fabric. This transformation demands organizational redesign, not just new tooling: governance, security, and compliance frameworks must account for machine actors, and engineering leadership must treat agentic integration as structural change.

As automation increases, the human contribution does not disappear; it shifts upward in abstraction. Engineers spend less time implementing routine scaffolding and more time designing architectural constraints, defining evaluation metrics, and supervising agent workflows. Prompt construction evolves into specification design. Code review becomes model oversight. Refactoring becomes architectural steering. New skill clusters emerge: structuring repositories for machine legibility, interpreting agent-generated reasoning, and detecting subtle misalignment. Importantly, deep technical understanding remains essential — an engineer who cannot reason about distributed systems or rendering performance cannot effectively supervise an agent modifying them. Mastery does not diminish in value; it becomes more critical.

One of the most significant frontiers is persistence. Today's agents operate largely within session-bounded context; the future demands continuity. Agents will reference durable knowledge stores capturing architectural decisions, historical trade-offs, performance benchmarks, and policy constraints across months or years, rather than re-inferring repository norms repeatedly. This persistent memory introduces new governance challenges — historical context must remain accurate, and architectural decisions must be versioned not only in code but in machine-readable constraints. Research in automated verification and constraint-based generation will increasingly integrate with agentic workflows, so that agents operate within bounded correctness frameworks — ensuring generated modifications satisfy type constraints, security policies, or architectural invariants before execution. Safety becomes programmable. And the most transformative shift may occur after deployment, as agents monitor runtime metrics, detect anomalies, propose remediations, implement candidate fixes in isolated branches, and deploy validated improvements through controlled rollouts. Maintenance becomes proactive rather than reactive; software evolves continuously under supervised autonomy, with human architects overseeing strategic direction.

Change will be incremental rather than instantaneous. Engineering remains a complex socio-technical system; human creativity, domain expertise, and ethical judgment cannot be automated wholesale. What changes is the distribution of effort: typing decreases, orchestration increases, execution becomes partially automated, and strategy becomes central. Organizations that prepare deliberately — investing in modular, well-documented repositories, developing internal agent workflows and policy constraints before scaling autonomy, training engineers in specification and oversight, and instrumenting pipelines for audit and rollback — will experience disproportionate leverage. The value of the engineer will lie less in keystrokes and more in judgment.

Epilogue — Engineering in the Age of Collaboration

When the first integrated development environments appeared, they did not replace programmers; they changed how programmers thought. When version control became standard, it did not reduce engineering; it restructured collaboration. When continuous integration and cloud infrastructure emerged, they did not diminish human expertise; they shifted it toward systems thinking. Agentic software engineering belongs in that lineage of structural shifts.

This book traced a progression: defining agentic engineering, installing and configuring a real agent, building an application with it, refactoring and extending that application, driving changes from screenshots, working inside the IDE, running parallel variants in the cloud, automating review, and deploying the result under governance. Each step demonstrated a simple but profound idea: when systems can plan and act, the nature of engineering changes — not because humans disappear, but because collaboration changes.

In traditional workflows, engineering skill often expressed itself through direct code authorship. In agentic workflows, that authorship becomes partially abstracted, and the engineer operates one level higher: defining constraints instead of writing every function, supervising structured transformations instead of manually refactoring, specifying intent and evaluating artifacts instead of implementing repetitive scaffolding. But orchestration demands deeper understanding, not less. To supervise an agent modifying a rendering pipeline, one must understand rendering pipelines; to validate a refactor across distributed services, one must grasp distributed systems. Delegation without comprehension is not leverage; it is risk.

The recurring themes of this book — modular repositories, clean branch discipline, audit logging, least-privilege execution, human-in-the-loop checkpoints, rollback strategies — are not bureaucratic layers. They are the interfaces through which humans and agents collaborate, and the structural safeguards that let autonomy scale responsibly. A modular codebase with explicit interfaces lets agents reason locally without unintended side effects. A well-instrumented pipeline lets them validate their own changes. A governed branching strategy lets autonomy proceed without chaos. Autonomy is seductive — the ability to scaffold, refactor, review, and merge at machine speed feels transformative — but without governance, acceleration becomes instability.

As agents grow more capable, it becomes tempting to ask whether software will eventually build itself. The question misunderstands the trajectory. Agents excel at execution, pattern synthesis, and structured reasoning within defined constraints. Humans remain uniquely capable of contextual judgment, cross-domain abstraction, ethical reasoning, and strategic foresight. Engineering is not only the act of implementing code; it is the act of deciding what should exist. Agents can help us build systems; they cannot determine why those systems matter. The human edge shifts upward — from syntax to strategy, from typing to intent, from mechanics to meaning.

Software increasingly mediates economic systems, public services, healthcare, finance, communication, and education. As engineering becomes more automated, the ethical weight of building systems does not decrease — it increases. When an agent modifies authentication flows

or recommendation algorithms, it operates within a system designed by humans; accountability, bias mitigation, and ethical alignment remain human responsibilities. The future of agentic engineering must therefore be grounded in responsibility. Speed is not the ultimate goal; sustainable, trustworthy systems are.

The opportunity before us is nonetheless extraordinary. Imagine engineering teams freed from repetitive boilerplate, exploring multiple implementation variants in parallel, responding to production feedback automatically, and focusing creative energy on architectural and strategic challenges. Imagine organizations where experimentation cycles shrink from months to days, where quality improves alongside velocity, where software evolves intelligently rather than stagnates. This is not a distant future; the foundations already exist. Agentic software engineering is not about replacing developers. It is about amplifying them — building systems that help us build better systems, shifting the locus of effort from mechanical repetition to meaningful design, and embedding autonomy within carefully constructed boundaries.

Engineering, at its core, has always been about extending human capability through tools. Agentic systems are simply the next tool — more powerful, more complex, and more demanding of our judgment than any before. The question is not whether they will change how we build software. They already have. The question is whether we will architect our systems, our teams, and our culture wisely enough to harness that change responsibly. The future of engineering will not be written solely by machines. It will be written by humans who understand how to collaborate with them.