OpenAI

Building an AI-native engineering team

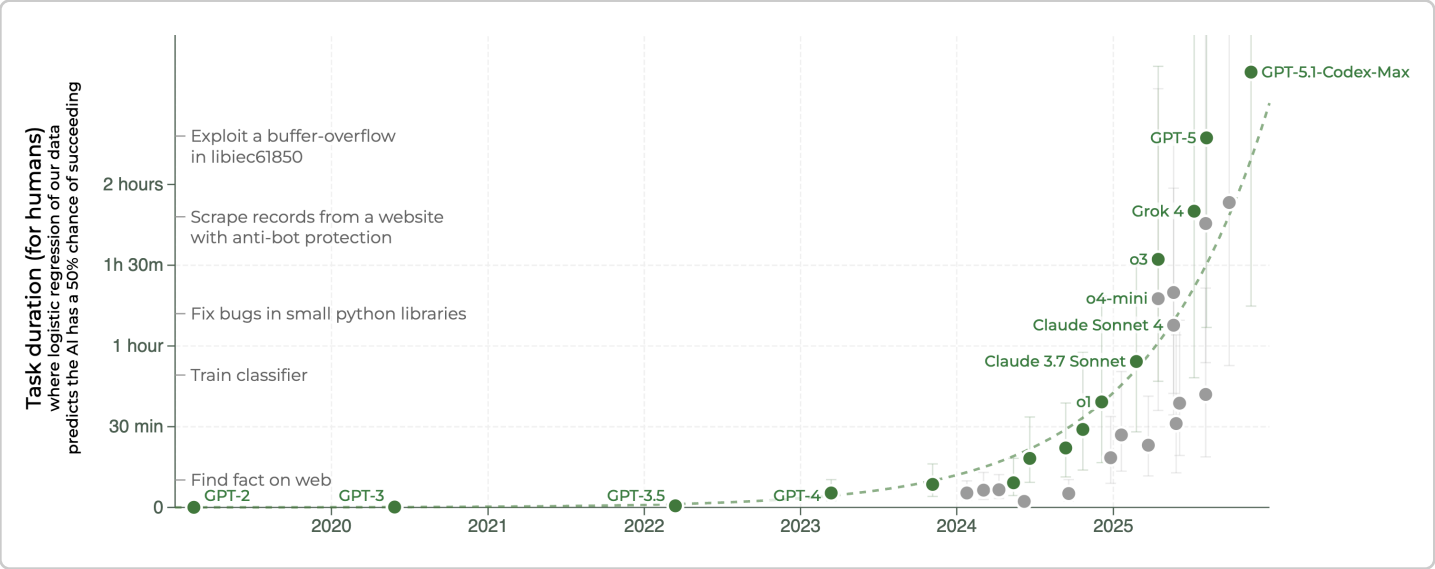
How coding agents accelerate the software development lifecycle

Introduction

AI models are rapidly expanding the range of tasks they can perform, with significant implications for engineering. Frontier systems now sustain multi-hour reasoning: as of August 2025, METR found that leading models could complete **2 hours and 17 minutes** of continuous work with roughly **50% confidence** of producing a correct answer.

This capability is improving quickly, with task length doubling about every seven months. Only a few years ago, models could manage about 30 seconds of reasoning – enough for small code suggestions. Today, as models sustain longer chains of reasoning, the entire software development lifecycle is potentially in scope for AI assistance, enabling coding agents to contribute effectively to planning, design, development, testing, code reviews, and deployment.

The time-horizon of software engineering tasks different LLMs can complete 50% of the time



METR, Measuring AI Ability to Complete Long Tasks

In this guide, we'll share real examples that outline how AI agents are contributing to the software development lifecycle with practical guidance on what engineering leaders can do today to start building AI-native teams and processes.

AI coding: from autocomplete to agents

AI coding tools have progressed far beyond their origins as autocomplete assistants. Early tools handled quick tasks such as suggesting the next line of code or filling in function templates. As models gained stronger reasoning abilities, developers began interacting with agents through chat interfaces in IDEs for pair programming and code exploration.

Today's coding agents can generate entire files, scaffold new projects, and translate designs into code. They can reason through multi-step problems such as debugging or refactoring, with agent execution also now shifting from an individual developer's machine to cloud-based, multi-agent environments. This is changing how developers work, allowing them to spend less time generating code with the agent inside the IDE and more time delegating entire workflows.

Advancement	What it enables
Unified context across systems	A single model can read code, configuration, and telemetry, providing consistent reasoning across layers that previously required separate tooling.
Structured tool execution	Models can now call compilers, test runners, and scanners directly, producing verifiable results rather than static suggestions.
Persistent project memory	Long context windows and techniques like compaction allow models to follow a feature from proposal to deployment, remembering previous design choices and constraints.
Evaluation loops	Model outputs can be tested automatically against benchmarks—unit tests, latency targets, or style guides—so improvements are grounded in measurable quality.

At OpenAI, we have witnessed this firsthand. Development cycles have accelerated, with work that once required weeks now being delivered in days. Teams move more easily across domains, onboard faster to unfamiliar projects, and operate with greater agility and autonomy across the organization. Many routine and time-consuming tasks, from documenting new code and surfacing relevant tests, maintaining dependencies and cleaning up feature flags are now delegated to Codex entirely.

However, some aspects of engineering remain unchanged. True ownership of code—especially for new or ambiguous problems—still rests with engineers, and certain challenges exceed the capabilities of current models. But with coding agents like Codex, engineers can now spend more time on complex and novel challenges, focusing on design, architecture, and system-level reasoning rather than debugging or rote implementation.

In the following sections, we break down how each phase of the SDLC changes with coding agents — and outline the concrete steps your team can take to start operating as an AI-native engineering org.

Plan

Teams across an organization often depend on engineers to determine whether a feature is feasible, how long it will take to build, and which systems or teams will be involved. While anyone can draft a specification, forming an accurate plan typically requires deep codebase awareness and multiple rounds of iteration with engineering to uncover requirements, clarify edge cases, and align on what is technically realistic.

How coding agents help

AI coding agents give teams immediate, code-aware insights during planning and scoping. For example, teams may build workflows that connect coding agents to their issue-tracking systems to read a feature specification, cross-reference it against the codebase, and then flag ambiguities, break the work into subcomponents, or estimate difficulty.

Coding agents can also instantly trace code paths to show which services are involved in a feature — work that previously required hours or days of manual digging through a large codebase.

What engineers do instead

Teams spend more time on core feature work because agents surface the context that previously required meetings for product alignment and scoping. Key implementation details, dependencies, and edge cases are identified up front, enabling faster decisions with fewer meetings.

Delegate	AI agents can take the first pass at feasibility and architectural analysis. They read a specification, map it to the codebase, identify dependencies, and surface ambiguities or edge cases that need clarification.
Review	Teams review the agent's findings to validate accuracy, assess completeness, and ensure estimates reflect real technical constraints. Story point assignment, effort sizing, and identifying non-obvious risks still require human judgment.
Own	Strategic decisions — such as prioritization, long-term direction, sequencing, and tradeoffs — remain human-led. Teams may ask the agent for options or next steps, but final responsibility for planning and product direction stays with the organization.

Getting started checklist

- [ ] Identify common processes that require alignment between features and source code. Common areas include feature scoping and ticket creation.
- [ ] Begin by implementing basic workflows, for example tagging and deduplicating issues or feature requests.
- [ ] Consider more advanced workflows, like adding sub-tasks to a ticket based on an initial feature description. Or kick off an agent run when a ticket reaches a specific stage to supplement the description with more details.

Design

The design phase is often slowed by foundational setup work. Teams spend significant time wiring up boilerplate, integrating design systems, and refining UI components or flows. Misalignment between mockups and implementation can create rework and long feedback cycles, and limited bandwidth to explore alternatives or adapt to changing requirements delays design validation.

How coding agents help

AI coding tools dramatically accelerate prototyping by scaffolding boilerplate code, building project structures, and instantly implementing design tokens or style guides. Engineers can describe desired features or UI layouts in natural language and receive prototype code or component stubs that match the team's conventions.

They can convert designs directly into code, suggest accessibility improvements, and even analyze the codebase for user flows or edge cases. This makes it possible to iterate on multiple prototypes in hours instead of days, and to prototype in high fidelity early, giving teams a clearer basis for decision-making and enabling customer testing far sooner in the process.

What engineers do instead

With routine setup and translation tasks handled by agents, teams can redirect their attention to higher-leverage work. Engineers focus on refining core logic, establishing scalable architectural patterns, and ensuring components meet quality and reliability standards. Designers can spend more time evaluating user flows and exploring alternative concepts. The collaborative effort shifts from implementation overhead to improving the underlying product experience.

Delegate	Agents handle the initial implementation work by scaffolding projects, generating boilerplate code, translating mockups into components, and applying design tokens or style guides.
Review	The team reviews the agent's output to ensure components follow design conventions, meet quality and accessibility standards, and integrate correctly with existing systems.
Own	The team owns the overarching design system, UX patterns, architectural decisions, and the final direction of the user experience.

Getting started checklist

- [ ] Use a multi-modal coding agent that accepts both text and image input
- [ ] Integrate design tools via MCP with coding agents
- [ ] Programmatically expose component libraries with MCP, and integrate them with your coding model
- [ ] Build workflows that map designs → components → implementation of components
- [ ] Utilize typed languages (e.g. Typescript) to define valid props and subcomponents for the agent

Build

The build phase is where teams feel the most friction, and where coding agents have the clearest impact. Engineers spend substantial time translating specs into code structures, wiring services together, duplicating patterns across the codebase, and filling in boilerplate, with even small features requiring hours of busy-work.

As systems grow, this friction compounds. Large monorepos accumulate patterns, conventions, and historical quirks that slow contributors down. Engineers can spend as much time rediscovering the “right way” to do something as implementing the feature itself. Constant context switching between specs, code search, build errors, test failures, and dependency management adds cognitive load — and interruptions during long-running tasks break flow and delay delivery further.

How coding agents help

Coding agents running in the IDE and CLI accelerate the build phase by handling larger, multi-step implementation tasks. Rather than producing just the next function or file, they can produce full features end-to-end — data models, APIs, UI components, tests, and documentation — in a single coordinated run. With sustained reasoning across the entire codebase, they handle decisions that once required engineers to manually trace code paths.

With long-running tasks, agents can:

- Draft entire feature implementations based on a written spec.
- Search and modify code across dozens of files while maintaining consistency.
- Generate boilerplate that matches conventions: error handling, telemetry, security wrappers, or style patterns.
- Fix build errors as they appear rather than pausing for human intervention.
- Write tests alongside implementation as part of a single workflow.
- Produce diff-ready changesets that follow internal guidelines and include PR messages.

In practice, this shifts much of the mechanical “build work” from engineers to agents. The agent becomes the first-pass implementer; the engineer becomes the reviewer, editor, and source of direction.

What engineers do instead

When agents can reliably execute multi-step build tasks, engineers shift their attention to higher-order work:

- Clarifying product behavior, edge cases, and specs before implementation.
- Reviewing architectural implications of AI-generated code instead of performing rote wiring.
- Refining business logic and performance-critical paths that require deep domain reasoning.
- Designing patterns, guardrails, and conventions that guide agent-generated code.
- Collaborating with PMs and design to iterate on feature intent, not boilerplate.

Instead of “translating” a feature spec into code, engineers concentrate on correctness, coherence, maintainability, and long-term quality, areas where human context still matters most.

Delegate	Agents draft the first implementation pass for well-specified features — scaffolding, CRUD logic, wiring, refactors, and tests. As long-running reasoning improves, this increasingly covers full end-to-end builds rather than isolated snippets.
Review	Engineers assess design choices, performance, security, migration risk, and domain alignment while correcting subtle issues the agent may miss. They shape and refine AI-generated code rather than performing the mechanical work.
Own	Engineers retain ownership of work requiring deep system intuition: new abstractions, cross-cutting architectural changes, ambiguous product requirements, and long-term maintainability trade-offs. As agents take on longer tasks, engineering shifts from line-by-line implementation to

Example Engineers, PMs, designers, and operators at Cloudwalk use Codex daily to turn specs into working code whether they need a script, a new fraud rule, or a full microservice delivered in minutes. It removes the busy work from the build phase and gives every employee the power to implement ideas at remarkable speed.

Getting started checklist

- [ ] Start with well specified tasks
- [ ] Have the agent use a planning tool via MCP, or by writing a PLAN.md file that is committed to the codebase
- [ ] Check that the commands the agent attempts to execute are succeeding
- [ ] Iterate on an AGENTS.md file that unlocks agentic loops like running tests and linters to receive feedback

Test

Developers often struggle to ensure adequate test coverage because writing and maintaining comprehensive tests takes time, requires context switching, and deep understanding of edge cases. Teams frequently face trade-offs between moving fast and writing thorough tests. When deadlines loom, test coverage is often the first thing to suffer.

Even when tests are written, keeping them updated as code evolves introduces ongoing friction. Tests can become brittle, fail for unclear reasons, and can require their own major refactors as the underlying product changes. High quality tests let teams ship faster with more confidence.

How coding agents help

AI coding tools can help developers author better tests in several powerful ways. First, they can suggest test cases based on reading a requirements document and the logic of the feature code. Models can be surprisingly good at suggesting edge cases and failure modes that may be easy for a developer to overlook, especially when they have been deeply focused on the feature and need a second opinion.

In addition, models can help tests up to date as code evolves, reducing the friction of refactoring and avoiding stale tests that become flaky. By handling the basic implementation details of test writing and surfacing edge cases, coding agents accelerate the process of developing tests.

What engineers do instead

Writing tests with AI tools doesn't remove the need for developers to think about testing. In fact, as agents remove barriers to generating code, tests serve a more and more important function as a source of truth for application functionality. Since agents can run the test suite and iterate based on the output, defining high quality tests is often the first step to allowing an agent to build a feature.

Instead, developers focus more on seeing the high level patterns in test coverage, building on and challenging the model's identification of test cases. Making test writing faster allows developers to ship features more quickly and also take on more ambitious features.

Delegate	Engineers will delegate the initial pass at generating test cases based on feature specifications. They'll also use the model to take a first pass at generating tests. It can be helpful to have the model generate tests in a separate session from the feature implementation.
Review	Engineers must still thoroughly review model-generated tests to ensure that the model did not take shortcuts or implement stubbed tests. Engineers also ensure that tests are runnable by their agents; that the agent has the appropriate permissions to run, and that the agent has context awareness of the different test suites it can run.
Own	Engineers own aligning test coverage with feature specifications and user experience expectations. Adversarial thinking, creativity in mapping edge cases, and focus on intent of the tests remain critical skills.

Getting started checklist

- [ ] Guide the model to implement tests as a separate step, and validate that new tests fail before moving to feature implementation.
- [ ] Set guidelines for test coverage in your AGENTS.md file
- [ ] Give the agent specific examples of code coverage tools it can call to understand test coverage

Review

On average, developers spend 2–5 hours per week conducting code reviews. Teams often face a choice between investing significant time in a deep review or doing a quick “good enough” pass for changes that seem small. When this prioritization is off, bugs slip into production, causing issues for users and creating substantial rework.

How coding agents help

Coding agents allow the code review process to scale so every PR receives a consistent baseline of attention. Unlike traditional static analysis tools (which rely on pattern matching and rule-based checks) AI reviewers can actually execute parts of the code, interpret runtime behavior, and trace logic across files and services. To be effective, however, models must be trained specifically to identify P0 and P1-level bugs, and tuned to provide concise, high-signal feedback; overly verbose responses are ignored just as easily as noisy lint warnings.

What engineers do instead

At OpenAI, we find that AI code review gives engineers more confidence that they are not shipping major bugs into production. Frequently, code review will catch issues that the contributor can correct before pulling in another engineer. Code review doesn’t necessarily make the pull request process faster, especially if it finds meaningful bugs – but it does prevent defects and outages.

Delegate vs. review vs. own

Even with AI code review, engineers are still responsible for ensuring that the code is ready to ship. Practically, this means reading and understanding the implications of the change. Engineers delegate the initial code review to an agent, but own the final review and merge process.

Delegate	Engineers delegate the initial coding review to agents. This may happen multiple times before the pull request is marked as ready for review by a teammate.
Review	Engineers still review pull requests, but with more of an emphasis on architectural alignment; are composable patterns being implemented, are the correct conventions being used, does the functionality match requirements.
Own	Engineers ultimately own the code that is deployed to production; they must ensure it functions reliably and fulfills the intended requirements.

Example Sansan uses Codex review for race conditions and database relations, which are issues humans often overlook. Codex has also been able to catch improper hard-coding and even anticipates future scalability concerns.

Getting started checklist

- [ ] Curate examples of gold-standard PRs that have been conducted by engineers including both the code changes and comments left. Save this as an evaluation set to measure different tools.
- [ ] Select a product that has a model specifically trained on code review. We've found that generalized models often nitpick and provide a low signal to noise ratio.
- [ ] Define how your team will measure whether reviews are high quality. We recommend tracking PR comment reactions as a low-friction way to mark good and bad reviews.
- [ ] Start small but rollout quickly once you gain confidence in the results of reviews.

Document

Most engineering teams know their documentation is behind, but find catching up costly. Critical knowledge is often held by individuals rather than captured in searchable knowledge bases, and existing docs quickly go stale because updating them pulls engineers away from product work. And even when teams run documentation sprints, the result is usually a one-off effort that decays as soon as the system evolves.

How coding agents help

Coding agents are highly capable of summarizing functionality based on reading codebases. Not only can they write about how parts of the codebase work, but they can also generate system diagrams in syntaxes like mermaid. As developers build features with agents, they can also update documentation simply by prompting the model. With AGENTS.md, instructions to update documentation as needed can be automatically included with every prompt for more consistency.

Since coding agents can be run programmatically through SDKs, they can also be incorporated into release workflows. For example, we can ask a coding agent to review commits being included in the release and summarize key changes. The result is that documentation becomes a built-in part of the delivery pipeline: faster to produce, easier to keep current, and no longer dependent on someone “finding the time.”

What engineers do instead

Engineers move from writing every doc by hand to shaping and supervising the system. They decide how docs are organized, add the important “why” behind decisions, set clear standards and templates for agents to follow, and review the critical or customer-facing pieces. Their job becomes making sure documentation is structured, accurate, and wired into the delivery process rather than doing all the typing themselves.

Delegate	Fully hand off low-risk, repetitive work to Codex like first-pass summaries of files and modules, basic descriptions of inputs and outputs, dependency lists, and short summaries of pull-request changes.
Review	Engineers review and edit important docs drafted by Codex like overviews of core services, public API and SDK docs, runbooks, and architecture pages, before anything is published.
Own	Engineers remain responsible for overall documentation strategy and structure, standards and templates the agent follows, and all external-facing or safety-critical documentation involving legal, regulatory, or brand risk.

Getting started checklist

- [ ] Experiment with documentation generation by prompting the coding agent
- [ ] Incorporate documentation guidelines into your AGENTS.md
- [ ] Identify workflows (e.g. release cycles) where documentation can be automatically generated
- [ ] Review generated content for quality, correctness, and focus

Deploy & maintain

Understanding application logging is critical to software reliability. During an incident, software engineers will reference logging tools, code deploys, and infrastructure changes to identify a root cause. This process is often surprisingly manual and requires developers to tab back and forth between different systems, costing critical minutes in high pressure situations like incidents.

How coding agents help

With AI coding tools, you can provide access to your logging tools via MCP servers in addition to the context of your codebase. This allows developers to have a single workflow where they can prompt the model to look at errors for a specific endpoint, and then the model can use that context to traverse the codebase and find relevant bugs or performance issues. Since coding agents can also use command line tools, they can look at the git history to identify specific changes that might result in issues captured in log traces.

What engineers do instead

By automating the tedious aspects of log analysis and incident triage, AI enables engineers to concentrate on higher-level troubleshooting and system improvement. Rather than manually correlating logs, commits, and infrastructure changes, engineers can focus on validating AI-generated root causes, designing resilient fixes, and developing preventative measures.This shift reduces time spent on reactive firefighting, allowing teams to invest more energy in proactive reliability engineering and architectural improvements.

Delegate	Many operational tasks can be delegated to agents — parsing logs, surfacing anomalous metrics, identifying suspect code changes, and even proposing hotfixes.
Review	Engineers vet and refine AI-generated diagnostics, confirm accuracy, and approve remediation steps. They ensure fixes meet reliability, security, and compliance standards.
Own	Critical decisions stay with engineers, especially for novel incidents, sensitive production changes, or situations where model confidence is low. Humans remain responsible for judgment and final sign-off.

Example Virgin Atlantic uses Codex to strengthen how teams deploy and maintain their systems. The Codex VS Code Extension gives engineers a single place to investigate logs, trace issues across code and data, and review changes through Azure DevOps MCP and Databricks Managed MCPs. By unifying this operational context inside the IDE, Codex speeds up root cause discovery, reduces manual triage, and helps teams focus on validating fixes and improving system reliability.

Getting started checklist

- [ ] Connect AI tools to logging and deployment systems: Integrate Codex CLI or similar with your MCP servers and log aggregators.
- [ ] Define access scopes and permissions: Ensure agents can access relevant logs, code repositories, and deployment histories, while maintaining security best practices.
- [ ] Configure prompt templates: Create reusable prompts for common operational queries, such as “Investigate errors for endpoint X” or “Analyze log spikes post-deploy.”
- [ ] Test the workflow: Run simulated incident scenarios to ensure the AI surfaces correct context, traces code accurately, and proposes actionable diagnostics.
- [ ] Iterate and improve: Collect feedback from real incidents, tune prompt strategies, and expand agent capabilities as your systems and processes evolve.

Conclusion

Coding agents are transforming the software development lifecycle by taking on the mechanical, multi-step work that has traditionally slowed engineering teams down. With sustained reasoning, unified codebase context, and the ability to execute real tools, these agents now handle tasks ranging from scoping and prototyping to implementation, testing, review, and even operational triage. Engineers stay firmly in control of architecture, product intent, and quality — but coding agents increasingly serve as the first-pass implementer and continuous collaborator across every phase of the SDLC.

This shift doesn't require a radical overhaul; small, targeted workflows compound quickly as coding agents become more capable and reliable. Teams that start with well-scoped tasks, invest in guardrails, and iteratively expand agent responsibility see meaningful gains in speed, consistency, and developer focus.

If you're exploring how coding agents can accelerate your organization or preparing for your first deployment, reach out to OpenAI. We're here to help you turn coding agents into real leverage—designing end-to-end workflows across planning, design, build, test, review, and operations, and helping your team adopt production-ready patterns that make AI-native engineering a reality.