构建 AI 原生工程团队

Figure 1: 编码智能体如何加速软件开发生命周期

简介

AI 模型正在迅速扩展其可执行任务的范围，这对软件工程领域产生了深远影响。前沿系统现在能够进行长达数小时的持续推理：截至 2025 年 8 月，METR 发现领先的模型可以持续工作 2 小时 17 分钟，并以约 50% 的准确率给出正确答案。

这种能力正在迅速提高，任务长度大约每七个月翻一番。就在几年前，模型的推理时间还只有区区 30 秒——仅够给出简短的代码建议。如今，随着模型能够进行更长时间的推理，整个软件开发生命周期都可能纳入 AI 协助的范围，使编码智能体能够有效地为规划、设计、开发、测试、代码审查和部署做出贡献。

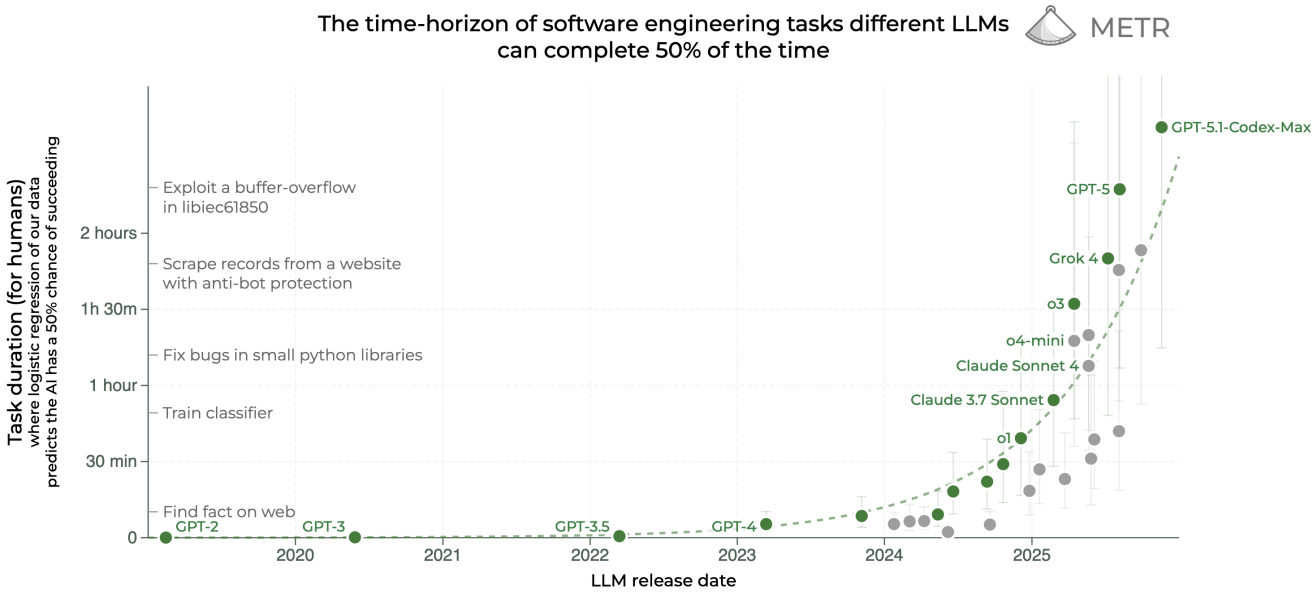

Figure 2: 不同大语言模型以 50% 成功率完成软件工程任务的时间跨度

METR，衡量 AI 完成长任务的能力

在本指南中，我们将分享真实案例，概述 AI 智能体如何为软件开发生命周期做出贡献，并为工程领导者提供切实可行的指导，说明他们如何开始构建 AI 原生团队和流程。

AI 编码：从自动补全到智能体

AI 编码工具已经远远超出了其作为自动补全助手的起源。早期的工具处理快速任务，例如建议下一行代码或填充函数模板。随着模型获得更强的推理能力，开发人员开始通过 IDE 中的聊天界面与智能体交互，进行结对编程和代码探索。

如今的编码智能体可以生成整个文件，搭建新项目，并将设计转化为代码。它们可以推理多步骤问题，例如调试或重构，智能体的执行也正从单个开发人员的机器转向基于云的多智能体环境。这正在改变开发人员的工作方式——少花时间在 IDE 里敲代码，多花精力在委托和指挥整个工作流上。

技术演进与赋能

跨系统的统一上下文

单个模型可以读取代码、配置和遥测数据，提供跨层的连贯推理，而这在以前需要单独的工具。

结构化工具执行

模型现在可以直接调用编译器、测试运行器和扫描器，产生可验证的结果，而不是静态建议。

持久的项目记忆

长上下文窗口和压缩等技术允许模型跟踪功能从提议到部署的全过程，记住以前的设计选择和约束。

评估循环

模型输出可以针对基准（单元测试、延迟目标或风格指南）自动测试，因此改进建立在可衡量的质量之上。

在 OpenAI，我们亲身见证了这一变化。开发周期大幅缩短——原本数周的工作，现在几天就能交付。团队跨领域协作更加顺畅，更快地融入陌生的项目，并在整个组织中以更大的敏捷性和自主性工作。许多常规且耗时的任务，从编写新代码的文档、识别相关测试，到维护依赖项、清理功能标志，现在都完全交由 Codex 处理。

然而，工程的某些方面保持不变。代码的真正责任人——尤其是对于新问题或模糊的需求——仍然是工程师，某些挑战超出了当前模型的能力。但有了像 Codex 这样的编码智能体，工程师现在可以将更多时间花在复杂和新颖的挑战上，专注于设计、架构和系统级推理，而不是调试或机械性的实现。

在接下来的部分中，我们将分解软件开发生命周期（SDLC）的每个阶段如何随着编码智能体而变化——并概述您的团队可以采取的具体步骤，以开始作为一个 AI 原生工程组织运作。

规划 (Plan)

组织中的团队通常依赖工程师来确定功能是否可行，构建需要多长时间，以及将涉及哪些系统或团队。虽然任何人都可以起草规范，但形成准确的计划通常需要对代码库的深入理解以及与工程部门的多轮迭代，以发现需求，澄清边缘情况，并就技术可行性达成一致。

编码智能体如何提供帮助

AI 编码智能体在规划和范围界定期间为团队提供即时的、基于代码的洞察。例如，团队可以构建工作流，将编码智能体连接到他们的问题跟踪系统，以读取功能规范，将其与代码库交叉引用，然后标记歧义，将工作分解为子组件，或估计难度。

编码智能体还可以立即追踪代码执行路径，以显示功能涉及哪些服务——这项工作以前需要数小时或数天的人工挖掘大型代码库。

工程师转而做什么

智能体能够自动识别关键上下文——这些信息过去需要通过会议来与产品团队对齐并确定范围——因此团队得以将更多时间投入核心功能开发。关键的实现细节、依赖项和边缘情况被提前识别，从而能够以更少的会议做出更快的决策。

角色转变

委托 (Delegate)

AI 智能体可以对可行性和架构分析进行初步处理。它们读取规范，将其映射到代码库，识别依赖项，并发现需要澄清的歧义或边缘情况。

审查 (Review)

团队审查智能体的发现以验证准确性，评估完整性，并确保估计反映真实的技术约束。故事点分配、工作量评估以及识别潜在风险仍需人类判断。

拥有 (Own)

战略决策——如优先级、长期方向、排序和权衡——仍然由人类主导。团队可能会向智能体询问选项或后续步骤，但规划和产品方向的最终责任在于组织。

入门清单

- □ 识别需要功能和源代码之间对齐的常见流程。常见领域包括功能范围界定和工单创建。
- □ 从实施基本工作流开始，例如标记和去重问题或功能请求。
- □ 考虑更高级的工作流，例如根据初始功能描述向工单添加子任务。
- □ 或者当工单达到特定阶段时启动智能体运行，以用更多细节补充描述。

设计 (Design)

设计阶段往往被基础设置工作拖慢。团队花费大量时间连接样板代码，集成设计系统，以及完善 UI 组件或流程。设计稿与实现之间的偏差会导致返工和漫长的反馈周期，而探索替代方案或适应需求变化的精力有限，也会延迟设计验证。

编码智能体如何提供帮助

AI 编码工具通过搭建样板代码、构建项目结构以及即时实施设计标记或风格指南，极大地加速了原型设计。工程师可以用自然语言描述所需的功能或 UI 布局，并接收符合团队惯例的原型代码或组件存根。

它们可以将设计直接转换为代码，建议可访问性改进，甚至分析代码库中的用户流或边缘情况。这使得在数小时而不是数天内迭代多个原型成为可能，并及早进行高保真原型设计，为团队提供更清晰的决策基础，并在流程中更早地启用客户测试。

工程师转而做什么

随着常规设置和转换任务由智能体处理，团队可以将注意力转移到更高价值的工作上。工程师专注于完善核心逻辑，建立可扩展的架构模式，并确保组件符合质量和可靠性标准。设计师可以花更多时间评估用户流和探索替代概念。协作努力从实现开销转移到改善底层产品体验。

角色转变

委托 (Delegate)

智能体通过搭建项目、生成样板代码、将设计稿转换为组件以及应用设计标记或风格指南来处理初始实现工作。

审查 (Review)

团队审查智能体的输出，以确保组件遵循设计惯例，符合质量和可访问性标准，并与现有系统正确集成。

拥有 (Own)

团队拥有总体设计系统、UX 模式、架构决策以及用户体验的最终方向。

入门清单

- □ 使用接受文本和图像输入的多模态编码智能体。
- □ 通过 MCP 将设计工具与编码智能体集成。
- □ 使用 MCP 以编程方式公开组件库，并将它们与您的编码模型集成。
- □ 构建映射工作流：设计 → 组件 → 组件实现。
- □ 利用类型化语言（例如 Typescript）为智能体定义有效的 props 和子组件。

构建 (Build)

构建阶段是团队感到摩擦最大的地方，也是编码智能体影响最明显的地方。工程师花费大量时间将规范转化为代码结构，将服务连接在一起，在代码库中复制模式，并填充样板代码，即使是很小的功能也需要数小时的繁琐工作。

随着系统的增长，这种摩擦会加剧。大型单一代码库积累了模式、惯例和历史怪癖，减慢了贡献者的速度。工程师花费在重新发现做某事的“正确方法”上的时间可能与实现功能本身的时间一样多。在需求规范、代码搜索、构建错误、测试失败和依赖项管理之间频繁的上下文切换，大大增加了认知负荷——长运行任务期间的中断会打破心流并进一步延迟交付。

编码智能体如何提供帮助

在 IDE 和 CLI 中运行的编码智能体通过处理更大的、多步骤的实现任务来加速构建阶段。它们不仅仅是生成下一个函数或文件，而是在一次协调的运行中端到端地生成完整的功能——数据模型、API、UI 组件、测试和文档。凭借对整个代码库的持续推理，它们处理曾经需要工程师手动跟踪代码路径的决策。

对于长运行任务，智能体可以：

- 根据书面规范起草整个功能实现。
- 搜索和修改数十个文件中的代码，同时保持一致性。
- 生成符合惯例的样板代码：错误处理、遥测、安全包装器或风格模式。
- 在构建错误出现时修复它们，而不是暂停等待人工干预。
- 作为单个工作流的一部分，与实现一起编写测试。
- 生成符合内部准则并包含 PR 消息的 diff-ready 变更集。

在实践中，大部分机械性的”构建工作”从工程师转移给了智能体。智能体成为初步实现者；工程师成为审查者、编辑和方向的来源。

工程师转而做什么

当智能体可以可靠地执行多步骤构建任务时，工程师将注意力转移到更高阶的工作：

- 在实施前澄清产品行为、边缘情况和规范。
- 审查 AI 生成代码的架构影响，而不是执行机械性的组件连接。
- 完善需要深刻领域推理的业务逻辑和性能关键路径。
- 设计指导智能体生成代码的模式、护栏和惯例。
- 与 PM 和设计协作迭代功能意图，而不是样板代码。

工程师不再将功能规范“翻译”成代码，而是专注于正确性、连贯性、可维护性和长期质量，这些领域人类的上下文理解仍然最为重要。

角色转变

委托 (Delegate)

智能体起草定义明确的功能的初步实现——脚手架、CRUD 逻辑、连接、重构和测试。随着长运行推理的改进，这越来越多地涵盖完整的端到端构建，而不是孤立的片段。

审查 (Review)

工程师评估设计选择、性能、安全性、迁移风险和领域对齐，同时纠正智能体可能错过的细微问题。他们塑造和完善 AI 生成的代码，而不是执行机械工作。

拥有 (Own)

工程师保留对需要深刻系统直觉的工作的所有权：新的抽象、横切架构变更、模糊的产品需求和长期可维护性权衡。随着智能体承担更长的任务，工程从逐行实现转变为架构设计。

示例

Cloudwalk 的工程师、PM、设计师和操作员每天都使用 Codex 将规范转化为可用代码，无论他们需要脚本、新的欺诈规则还是几分钟内交付的完整微服务。它消除了构建阶段的繁琐工作，并赋予每位员工以惊人的速度实现想法的能力。

入门清单

- □ 从定义明确的任务开始。
- □ 让智能体通过 MCP 使用规划工具，或者通过编写提交到代码库的 PLAN.md 文件。
- □ 检查智能体尝试执行的命令是否成功。
- □ 迭代 AGENTS.md 文件，该文件解锁智能体循环，如运行测试和 linter 以接收反馈。

测试 (Test)

开发人员经常难以确保足够的测试覆盖率，因为编写和维护全面的测试需要时间，需要上下文切换，以及对边缘情况的深刻理解。团队经常面临在快速移动和编写彻底测试之间的权衡。当截止日期临近时，测试覆盖率通常是第一个受害者。

即使编写了测试，随着代码的演进保持它们更新也会引入持续的摩擦。测试可能会变得脆弱，因不清楚的原因而失败，甚至可能随底层产品变化而需要大幅重构。高质量的测试让团队更有信心地更快发布。

编码智能体如何提供帮助

AI 编码工具可以通过几种强大的方式帮助开发人员编写更好的测试。首先，它们可以根据阅读需求文档和功能代码的逻辑建议测试用例。模型在建议边缘情况和故障模式方面可能出奇地好，这些对于开发人员来说可能很容易被忽视，特别是当他们专注于功能并需要第二意见时。

此外，模型可以帮助测试随着代码的发展保持最新，减少重构的摩擦，并避免过时的测试变得不稳定。通过处理测试编写的基本实现细节和识别边缘情况，编码智能体加速了开发测试的过程。

工程师转而做什么

使用 AI 工具编写测试并不能消除开发人员思考测试的需要。事实上，随着智能体消除了生成代码的障碍，测试作为验证应用功能正确性的权威依据，其重要性愈发凸显。由于智能体可以运行测试套件并根据输出进行迭代，因此定义高质量的测试通常是允许智能体构建功能的第一步。

相反，开发人员更多地关注测试覆盖率的整体模式，在模型识别的测试用例基础上进行补充和优化。使测试编写更快允许开发人员更快地发布功能，并挑战更有难度的需求。

角色转变

委托 (Delegate)

工程师会委托智能体初步生成基于功能规范的测试用例。他们还将使用模型完成测试代码的初步实现。让模型在与功能实现分开的会话中生成测试可能会有所帮助。

审查 (Review)

工程师必须仍然彻底审查模型生成的测试，以确保模型没有走捷径或生成空壳测试。工程师还要确保测试能够被智能体运行——包括智能体拥有适当的执行权限，并能识别可用的测试套件。

拥有 (Own)

工程师拥有将测试覆盖率与功能规范和用户体验期望对齐的所有权。逆向思维、发现边缘情况的创造力以及对测试意图的把握仍然是关键技能。

入门清单

- □ 引导模型将实现测试作为一个单独的步骤，并验证新测试在转移到功能实现之前失败。
- □ 在您的 AGENTS.md 文件中设置测试覆盖率指南。
- □ 给智能体特定的代码覆盖率工具示例，它可以调用这些工具来了解测试覆盖率。

代码审查 (Review)

平均而言，开发人员每周花费 2-5 小时进行代码审查。团队经常面临选择：是投入大量时间进行深度审查，还是对看似微小的更改进行快速的”足够好”的通过。当优先级判断失误时，错误会溜进生产环境，给用户带来问题并造成大量的返工。

编码智能体如何提供帮助

编码智能体允许代码审查过程扩展，以便每个 PR 都收到一致的基线关注。与传统的静态分析工具（依赖于模式匹配和基于规则的检查）不同，AI 审查者能够真正执行部分代码，解释运行时行为，并跟踪跨文件和服务的逻辑。然而，为了有效，模型必须经过专门训练以识别 P0 和 P1 级别的错误，并进行调整以提供简洁、高信号的反馈；过于冗长的响应就像嘈杂的 lint 警告一样容易被忽略。

工程师转而做什么

在 OpenAI，我们发现 AI 代码审查让工程师更有信心，确信他们不会将重大错误发布到生产环境。AI 代码审查常能在早期发现问题，让开发者无需他人介入即可自行修正。代码审查不一定会加快拉取请求的处理速度，特别是当它发现了重要问题时——但它确实能有效防止缺陷和故障。

委托 vs. 审查 vs. 拥有

即使有 AI 代码审查，工程师仍然负责确保代码准备好发布。实际上，这意味着阅读和理解变更的影响。工程师将初始代码审查委托给智能体，但拥有最终审查和合并过程。

角色转变

委托 (Delegate)

工程师将初始编码审查委托给智能体。这可能会在拉取请求被标记为准备好由队友审查之前发生多次。

审查 (Review)

工程师仍然审查拉取请求，但更强调架构对齐；是否实施了可组合的模式，是否使用了正确的惯例，功能是否符合要求。

拥有 (Own)

工程师最终拥有部署到生产环境的代码；他们必须确保其可靠运行并满足预期要求。

示例

Sansan 使用 Codex 审查竞争条件和数据库关系，这是人类经常忽视的问题。Codex 还能捕捉不当的硬编码，甚至预测未来的可扩展性问题。

入门清单

- □ 策划由工程师进行的黄金标准 PR 示例，包括代码更改和留下的评论。将其保存为评估集以衡量不同的工具。
- □ 选择一款专门针对代码审查进行过训练的产品。我们发现通用模型经常吹毛求疵，信噪比低。
- □ 定义您的团队将如何衡量审查是否高质量。我们建议跟踪 PR 评论反应，作为标记好坏审查的低摩擦方式。
- □ 从小处着手，待对审查效果建立信心后再快速推广。

文档 (Document)

大多数工程团队都知道他们的文档落后了，但发现赶上成本很高。关键知识通常由个人持有，而不是捕获在可搜索的知识库中，现有的文档很快就会过时，因为更新它们会把工程师从产品工作中拉走。即使团队进行文档冲刺，结果通常是一次性的努力，一旦系统演变就会过时。

编码智能体如何提供帮助

编码智能体在阅读代码库并总结功能方面表现出色。它们不仅能撰写关于代码库各部分工作原理的说明，还能生成 Mermaid 等格式的系统架构图。随着开发人员利用智能体构建功能，只需简单提示模型即可同步更新文档。借助 AGENTS.md，系统可以自动在每个提示词中包含按需更新文档的指令，从而确保持续的一致性。

由于编码智能体可以通过 SDK 以编程方式运行，因此可以将其集成到发布工作流中。例如，我们可以让智能体审查发布版本中包含的提交记录，并总结关键变更。这样一来，文档就成为了交付流水线的内建环节：生成速度更快，维护更容易，且不再依赖人工“抽时间”来更新。

工程师转而做什么

工程师的角色从手工编写每份文档，转变为塑造和监督整个文档系统。他们负责规划文档结构，补充决策背后的关键“原因”，为智能体设定清晰的标准和模板，并审查关键或面向客户的内容。他们的工作重心变成了确保文档的结构化、准确性以及与交付流程的衔接，而不再是亲自完成所有的撰写工作。

角色转变

委托 (Delegate)

将低风险、重复性的工作完全移交给 Codex，如文件和模块的第一遍摘要、输入和输出的基本描述、依赖项列表以及拉取请求更改的简短摘要。

审查 (Review)

工程师在任何内容发布之前，审查和编辑由 Codex 起草的重要文档，如核心服务概述、公共 API 和 SDK 文档、运行手册和架构页面。

拥有 (Own)

工程师仍然负责整体文档策略和结构、智能体遵循的标准和模板，以及所有涉及法律、监管或品牌风险的面向外部或安全关键的文档。

入门清单

- ☐ 尝试通过提示编码智能体生成文档。
- ☐ 将文档指南纳入您的 AGENTS.md。
- ☐ 识别可以自动生成文档的工作流（例如发布周期）。
- ☐ 审查生成内容的质量、正确性和重点。

部署与维护 (Deploy & Maintain)

了解应用程序日志记录对于软件可靠性至关重要。在事件期间，软件工程师将参考日志记录工具、代码部署和基础设施变更以识别根本原因。这个过程往往需要大量手动操作，需要开发人员在不同系统之间来回切换，在像事件这样的高压情况下花费关键的几分钟。

编码智能体如何提供帮助

使用 AI 编码工具，除了代码库的上下文之外，您还可以通过 MCP 服务器提供对日志记录工具的访问。这让开发人员可以使用单一工作流，提示模型查看特定端点的错误，然后模型可以使用该上下文遍历代码库并查找相关的错误或性能问题。由于编码智能体还可以使用命令行工具，它们可以查看 git 历史记录以识别可能导致日志跟踪中捕获的问题的特定更改。

工程师转而做什么

通过自动化日志分析和事件分类的繁琐方面，AI 使工程师能够专注于更高级别的故障排除和系统改进。工程师不再手动关联日志、提交和基础设施变更，而是专注于验证 AI 生成的根本原因，设计弹性修复，并制定预防措施。这种转变减少了花在被动灭火上的时间，使团队能够将更多精力投入到主动可靠性工程和架构改进中。

角色转变

委托 (Delegate)

许多操作任务可以委托给智能体——解析日志、识别异常指标、识别可疑代码更改，甚至提议热修复。

审查 (Review)

工程师审查和完善 AI 生成的诊断，确认准确性，并批准补救步骤。他们确保修复符合可靠性、安全性和合规性标准。

拥有 (Own)

关键决策留在工程师手中，特别是对于新颖的事件、敏感的生产变更或模型置信度低的情况。人类仍然负责判断和最终签署。

示例

维珍大西洋航空使用 Codex 加强团队部署和维护系统的方式。Codex VS Code 扩展为工程师提供了一个单一的地方来调查日志，跨代码和数据跟踪问题，并通过 Azure DevOps MCP 和 Databricks Managed MCP 审查更改。通过在 IDE 内统一这种操作上下文，Codex 加速了根本原因的定位，减少了手动分类，并帮助团队专注于验证修复和提高系统可靠性。

入门清单

- □ 将 AI 工具连接到日志记录和部署系统：将 Codex CLI 或类似的工具与您的 MCP 服务器和日志聚合器集成。
- □ 定义访问范围和权限：确保智能体可以访问相关的日志、代码存储库和部署历史记录，同时保持安全最佳实践。
- □ 配置提示模板：为常见的操作查询创建可重用的提示，例如“调查端点 X 的错误”或“分析部署后的日志峰值”。
- □ 测试工作流：运行模拟事件场景，以确保 AI 识别正确的上下文，准确跟踪代码，并提出可操作的诊断。
- □ 迭代和改进：从真实事件中收集反馈，调整提示策略，并随着系统和流程的发展扩展智能体能力。

结论

编码智能体正在通过承担传统上减慢工程团队速度的机械、多步骤工作来改变软件开发生命周期。凭借持续的推理、统一的代码库上下文以及执行真实工具的能力，这些智能体现在处理从范围界定和原型设计到实现、测试、审查甚至操作分类的任务。工程师牢牢控制着架构、产品意图和质量——而编码智能体正日益成为初步实现者，以及贯穿整个开发生命周期的持续协作者。

这种转变不需要彻底的改革；随着编码智能体变得更有能力和更可靠，小的、有针对性的工作流会迅速产生叠加效应。从范围明确的任务开始，投资于护栏，并迭代扩展智能体责任的团队在速度、一致性和开发人员专注度方面看到了有意义的收益。

如果您正在探索编码智能体如何加速您的组织或准备您的第一次部署，请联系 OpenAI。我们在这里帮助您将编码智能体转化为真正的杠杆——设计跨越规划、设计、构建、测试、审查和操作的端到端工作流，并帮助您的团队采用使 AI 原生工程成为现实的生产就绪模式。

版权

Figure 3: 扫一扫关注“西滨 AI 随想”公众号和你一起学 AI