《氛围编程：编程的未来》

Vibe Coding：编程的未来

作者：Addy Osmani

在美国印刷。

由O’Reilly Media, Inc.出版，地址：1005 Gravenstein Highway North, Sebastopol, CA 95472。

O’Reilly图书可用于教育、商业或销售推广用途。大多数书目也提供在线版本(http://oreilly.com)。如需更多信息，请联系我们的企业/机构销售部门：800-998-9938或corporate@oreilly.com。

- 策划编辑：Louise Corrigan
- 开发编辑：Sarah Grey
- 制作编辑：Katherine Tozer
- 内页设计师：David Futato
- 封面设计师：Karen Montgomery
- 插图师：Kate Dullea
- 2025年8月：第一版

早期版本修订历史

- 2025-04-17：第一次发布

请查看http://oreilly.com/catalog/errata.csp?isbn=9798341634756了解发布详情。

979-8-341-63470-1

简要目录（尚未最终确定）

第1章：Vibe转变：有意图的编程（不可用）

第2章：提示的艺术：与AI有效沟通（不可用）

第3章：70%问题：AI辅助编程的残酷真相（可用）

第4章：突破70%：最大化人类贡献（可用）

第5章：理解生成的代码：审查、完善、掌握（不可用）

第6章：AI驱动的原型设计：工具和技术（不可用）

第7章：使用AI构建Web应用（不可用）

第8章：AI生成代码的安全性和可靠性（不可用）

第9章：Vibe Coding的伦理影响（不可用）

第10章：程序员的解构：个人软件（不可用）

第11章：超越代码生成：AI的扩展作用（不可用）

第12章：Vibe程序员工具包：高级技术（不可用）

第1章 70%问题：实际有效的AI辅助工作流程

早期读者须知

通过早期版本电子书，您可以获得最早期形式的书籍——作者在写作时的原始未编辑内容——这样您就可以在这些标题正式发布前很久就利用这些技术。

这将是最终书籍的第3章。

如果您对我们如何改进本书内容和/或示例有意见，或者发现本章缺少材料，请联系编辑sgrey@oreilly.com。

基于AI的编程工具在某些任务上表现得惊人出色。它们擅长生成样板代码、编写常规函数，并让项目*大部分*完成。实际上，许多开发者发现AI助手可以实现一个初始解决方案，覆盖大约70%的需求。

Peter Yang的一条推文完美地概括了我在实践中观察到的现象：

> *作为非工程师使用AI编程的诚实反思：*
>
> *它可以让你完成70%，但最后30%很令人沮丧。它不断地前进一步后退两步，出现新的bug、问题等等。*
>
> *如果我知道代码是如何工作的，我可能可以自己修复它。但由于我不懂，我怀疑自己是否真的学到了很多。*

使用AI编程的非工程师发现自己撞上了一堵令人沮丧的墙。他们可以惊人地快速完成70%的工作，但最后30%变成了收益递减的练习。

这个”70%问题”揭示了AI辅助开发当前状态的重要事实。初始进展感觉神奇——你可以描述你想要什么，像v0或Bolt这样的AI工具会生成一个看起来令人印象深刻的工作原型。但随后现实来了。

70%通常是工作中直接的、模式化的部分——那种遵循成熟路径或常见框架的代码。正如一位*Hacker News*评论者观察到的，AI在处理软件的“偶然复杂性”（重复的、机械的东西）方面表现出色，而”本质复杂性”——理解和管理问题的固有复杂性——仍然落在人类肩上。用Fred Brooks的经典术语来说，AI解决附带问题，但不是开发的内在困难。

这些工具在哪些方面存在不足？经验丰富的工程师经常提到存在”最后一公里”的差距。AI 可以生成一个看似合理的解决方案，但最后 30% 的工作——处理边界情况、优化架构和确保可维护性——“需要严肃的人类专业知识”。

例如，AI 可能会给你一个在基本场景下技术上可行的函数，但它不会自动考虑异常输入、竞争条件、性能约束或未来需求，除非明确告知。AI 可以让你完成大部分工作，但最后关键的 30%（边界情况、保持可维护性和可靠的架构）需要严肃的人类专业知识。

此外，AI 有一个已知的倾向，即生成令人信服但不正确的输出。它可能引入微妙的错误或”幻想出”不存在的函数和库。Steve Yegge 诙谐地比喻今天的 LLM 就像”极其高效的初级开发者”——速度极快且充满热情，但”可能被改变心智的药物影响”，容易想出疯狂或不可行的方法。

用 Yegge 的话说，LLM 可以吐出乍一看很精致的代码，但如果一个经验不足的开发者天真地说”看起来不错！“并采用它，在接下来的几周里就会发生喜剧（或灾难）。AI 并不真正*理解*问题；它拼接通常有意义的模式。只有人类才能识别一个看似良好的解决方案是否隐藏着长期的陷阱。Simon Willison 呼应了这一点，在看到 AI 提出了一个*只有对问题领域有深入理解的高级工程师*才能识别出有缺陷的迷人巧妙设计之后。教训是：AI 的信心远远超过了它的可靠性。

关键的是，当前的 AI 不会创造超出其训练数据的根本性新抽象或策略。它们不会为你发明新颖的算法或创新的架构——它们重新组合已知的内容。它们也不会为决策承担责任。正如一位工程师指出的：“AI 不会有比其训练数据包含的内容’更好的想法’。它们不会为自己的工作承担责任。”

所有这些意味着创造性和分析性思维——*决定构建什么*、*如何构建*以及*为什么*——牢牢地保留在人类领域。总而言之，AI 是开发者的力量倍增器，处理重复性的 70% 工作并给我们提供生产力的”涡轮增压”。但它*不是*能够替代人类判断的万能药。软件工程剩余的 30%——困难的部分——仍然需要只有经过培训的、深思熟虑的开发者才能带来的技能。这些是值得关注的持久技能，第 6 章专门讨论它们。正如一个讨论所说：“AI 是一个强大的工具，但它不是万能药……人类判断和良好的软件工程实践仍然至关重要。”

我观察到团队在利用 AI 进行开发时有两种不同的模式。让我们称它们为”引导者”和”迭代者”。两者都在帮助工程师（甚至非技术用户）减少从想法到执行（或 MVP）的差距。

首先，有*引导者*，他们通常将新项目从零带到 MVP。像 Bolt、v0 和截图转代码 AI 这样的工具正在彻底改变这些团队引导新项目的方式。这些团队通常：

- 从设计或粗略概念开始
- 使用 AI 生成完整的初始代码库
- 在几小时或几天内（而不是几周）获得工作原型
- 专注于快速验证和迭代

结果可能令人印象深刻。我最近看到一个独立开发者使用 Bolt 在极短时间内将 Figma 设计转换为工作的 web 应用程序。虽然还没有准备好投入生产，但足以获得最初的用户反馈。

第二阵营，*迭代者*，在他们的日常开发工作流程中使用像 Cursor、Cline、Copilot 和 WindSurf 这样的工具。这不那么引人注目，但可能更具变革性。这些开发者：

- 使用 AI 进行代码完成和建议
- 利用 AI 进行复杂的重构任务
- 生成测试和文档
- 将 AI 用作解决问题的”结对程序员”

但问题是：虽然这两种方法都能显著加速开发，但它们带来的隐性成本并不立即显现。

当你观察一个高级工程师使用像 Cursor 或 Copilot 这样的 AI 工具工作时，看起来像魔法。他们可以在几分钟内搭建整个功能，包括测试和文档。但仔细观察，你会注意到关键的一点：他们不只是接受 AI 的建议。他们不断将生成的

代码重构为更小、更专注的模块。他们添加 AI 遗漏的全面错误处理和边界情况处理，加强其类型定义和接口，并质疑其架构决策。换句话说，他们正在应用多年来艰难获得的工程智慧来塑造和约束 AI 的输出。AI 正在加速他们的实现，但他们的专业知识是保持代码可维护的关键。

常见失败模式

初级工程师经常会错过这些关键步骤。他们更容易接受 AI 的输出结果，这导致了我所说的”纸牌屋代码”——看起来完整但在现实压力下会崩塌。

倒退两步

接下来通常发生的情况遵循一个我称为”倒退两步”的可预测模式（如[图1-1]所示）：

- 你试图修复一个小bug。
- AI 建议一个看起来合理的改动。
- 这个修复破坏了其他功能。
- 你要求 AI 修复新问题。
- 这又创造了两个新问题。
- 不断重复。

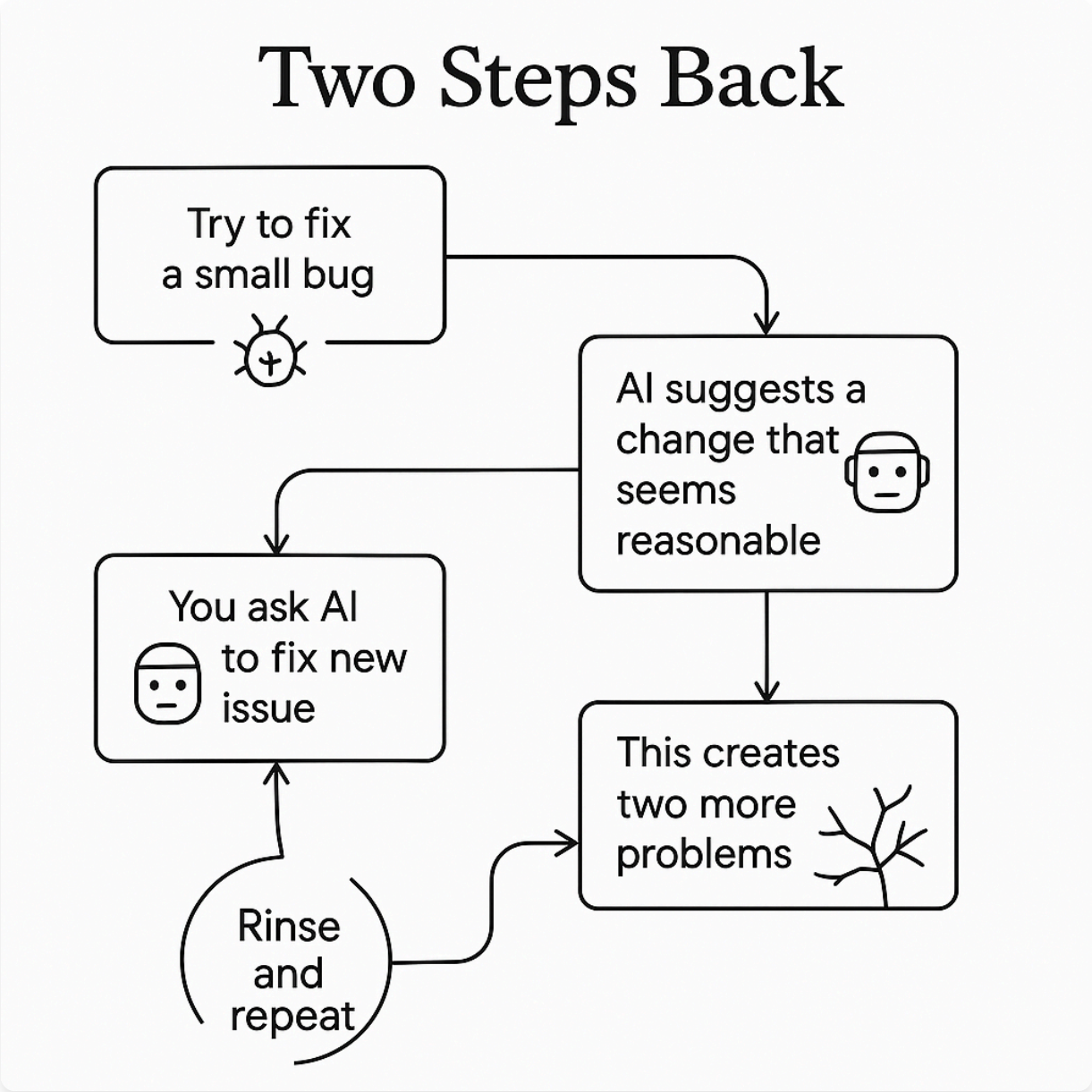

图 1-1. “倒退两步”反模式

这个循环对非工程师来说特别痛苦，因为他们缺乏理解实际出了什么问题的心智模型。当有经验的开发者遇到bug时，他们可以基于多年的模式识别经验推理潜在原因和解决方案。没有这种背景，你本质上是在与你不完全理解的代码玩打地鼠游戏。这就是我在本书前言中提到的”知识悖论”：**高级工程师和开发者使用 AI 来加速他们已经知道如何做的事情，而初级人员试图使用它来学习*要做什么*。**

这个循环对使用”自举者”模式的非工程师来说特别痛苦，因为他们缺乏构建MVP时解决这些问题所需的心智模型。然而，即使是经验丰富的”迭代者”，如果过度依赖 AI 建议而没有深度验证，也可能陷入这种打地鼠陷阱。

这里有一个更深层的问题：让 AI 编码工具对非工程师可及的东西——它们代你处理复杂性的能力——实际上可能阻碍学习。当代码只是”出现”而你不理解底层原理时，你就不会发展调试技能。你会错过学习基础模式。你无法推

理架构决策，因此你难以维护和发展代码。这创造了一种依赖关系，你需要不断回到 AI 模型来修复问题，而不是发展自己处理它们的专业知识。

这种依赖风险可能随着更自主的”智能体”AI 系统的兴起而加深。这些智能体不仅仅建议代码片段：它们可以规划、执行和迭代整个开发任务，只需最少的人工输入。在我写这篇文章的2025年初，像 Cline、Devin AI 和 Claude Code 这样的工具已经展示了自主调试、测试甚至部署代码的能力。虽然这承诺了效率，但也引入了新的挑战。没有对底层流程的扎实理解，用户可能发现自己越来越依赖这些智能体，当事情出错时无法有效干预。

此外，随着智能体 AI 系统越来越多地集成到开发工作流程中，级联错误的可能性增加。一个自主智能体可能做出一系列单独看起来合理但整体上将项目引向意外方向的决策。没有审计和纠正这些决策的专业知识，用户面临在不稳定基础上构建的风险。

本质上，虽然智能体 AI 为软件开发提供了强大工具，但也放大了基础知识的重要性。为了有效和负责任地利用这些工具，用户必须培养对软件原理的深度理解，确保他们保持对开发过程的控制，而不是成为被动观察者。

演示质量陷阱

这正在成为一种模式：团队使用 AI 快速构建令人印象深刻的演示。快乐路径运行得很美好。投资者和社交网络被震撼了。但当真实用户开始四处点击时？事情就开始分崩离析。

我亲眼见过：对普通用户毫无意义的错误消息，使应用程序崩溃的边缘情况，从未清理的混乱UI状态，完全被忽视的可访问性，在较慢设备上的性能问题。这些不仅仅是低优先级的bug——它们是人们容忍的软件和人们喜爱的软件之间的区别。

创建真正的自助服务软件——用户永远不需要联系支持的那种——需要一种不同的心态，这完全关于打磨的失落艺术。你需要痴迷于错误消息，在慢连接和真实的非技术用户那里测试，让功能可被发现，并优雅地处理每个边缘情况。这种对细节的关注（也许）无法由 AI 生成。它来自同理心、经验和对工艺的深度关怀。

真正有效的方法：实用工作流模式

在我们深入本书第二部分的编码之前，我们需要讨论现代开发实践以及 AI 辅助编码如何适应团队工作流程。毕竟，软件开发不仅仅是编写代码——它是一个包含规划、协作、测试、部署和维护的完整工作流程。而氛围编码(vibe coding)不是一个独立的新奇事物——它可以融入敏捷方法论和DevOps实践中，在保持质量和可靠性的同时增强团队的生产力。

在本节中，我们将探讨团队成员如何集体使用氛围编码工具而不互相干扰，如何平衡 AI 建议和人类洞察，以及持续集成/持续交付(CI/CD)管道如何整合 AI 或适应 AI 生成的代码。我还将涉及版本控制策略等重要考虑因素。

在观察了几十个团队后，以下是我在单人和团队工作流程中看到的三种一致有效的模式：

- *AI作为初稿撰写者*，AI模型生成初始代码，然后开发者进行完善、重构和测试。
- *AI作为结对编程伙伴*，开发者与AI持续对话，保持紧密的反馈循环、频繁的代码审查，并提供最少的上下文信息
- *AI作为验证者*，开发者仍然编写初始代码，然后使用AI来验证、测试和改进代码

在本节中，我将逐一介绍每种模式，讨论工作流程和成功的技巧。

AI作为初稿撰写者

在要求AI模型开始起草任何代码之前，确保团队中的每个人都达成一致是很重要的。沟通是关键，这样开发者就不会要求他们的AI助手执行重复的任务或生成冲突的实现。

在每日站会（敏捷工作流的基础）中，不仅值得讨论你正在做什么，还要讨论是否计划将AI用于某些任务。例如，两个开发者可能正在开发不同的功能，但都涉及日期格式化的实用函数——如果两人都要求AI创建一个 `formatDate` 辅助函数，你可能会得到两个相似的函数。提前协调（“我将生成一个我们都可以使用的日期实用程序”）可以防止重复。

成功集成AI工具的团队通常从就编码标准和提示实践达成一致开始。例如，团队可能决定采用一致的风格（代码检查规则、项目约定），甚至将这些指导原则输入到他们的AI工具中（一些助手允许提供风格偏好或示例代码来引导输出）。正如Codacy博客所指出的，通过让AI熟悉团队的编码标准，你可以获得更统一、更容易让每个人使用的生成代码。在实际层面上，这可能意味着在项目README中有一个”AI使用技巧”部分，在那里你记录诸如”我们只使用函数式组件”或”优先使用Fetch API而不是Axios”之类的内容，开发者在提示AI时可以记住这些内容。

另一个实践是使用工具的*协作功能*（如果可用）。一些AI辅助IDE允许用户分享他们的AI会话，或者至少分享他们使用的提示。如果开发者A通过提示获得了复杂组件的出色结果，与开发者B分享该提示（也许通过问题跟踪器或团队聊天）可以节省时间并确保一致性。

至于使用版本控制，基本原则保持不变——但有所不同。使用Git（或其他版本控制系统）在现代开发中是不可协商的，这在氛围编码(vibe coding)中不会改变。实际上，当AI快速生成代码时，版本控制变得更加关键。提交充当安全网来捕获AI的错误；如果AI生成的更改破坏了某些东西，你可以恢复到以前的提交。

一种策略是在使用AI辅助时更频繁地提交。每次AI产生你接受的重要代码块（如生成功能或进行重大重构）时，考虑使用清晰的消息进行提交。频繁的提交确保如果你需要分析问题或撤消AI引入的部分代码，历史记录足够精细。

另外，尽量隔离不同的AI引入的更改。如果你让AI在不同区域进行许多更改并将它们一起提交，那么如果出现问题就更难分离。例如，如果你使用代理来优化性能，它也调整了一些UI文本，请分别提交这些更改。（你的两个提交消息可能是”优化列表渲染性能 [AI辅助]“和”更新训练完成消息的UI文案 [AI辅助]“）。描述性的提交消息很重要；一些团队甚至标记有大量AI参与的提交，只是为了可追溯性。这不是关于责备，而是关于理解代码的来源——标记为”[AI]“的提交可能向审查者表明代码可能需要对边缘情况进行额外彻底的审查。

从本质上讲，团队应该将AI使用视为开发对话的正常部分：分享经验、成功的技术，以及关于什么不应该做的警告（如”Copilot建议为X使用过时的库，要小心”）。

审查和完善对这种模式至关重要。开发者应该手动审查和重构代码的模块性，添加全面的错误处理，编写彻底的测试，并在完善代码时记录关键决策。下一章将详细介绍这些流程。

AI作为结对编程伙伴

传统的结对编程涉及两个人类在一个工作站协作。随着AI的出现，出现了一种混合方法：一个人类开发者与AI助手一起工作。这种设置可能特别有效，提供人类直觉和机器效率的融合。

在人类-AI配对中，开发者与AI交互以生成代码建议，同时也审查和完善输出。这种动态允许人类利用AI在处理重复性任务（如编写样板代码或生成测试用例）方面的速度，同时保持监督以确保代码质量和相关性。

例如，在集成新库时，开发者可能提示AI起草初始集成代码。然后开发者审查AI的建议，与官方文档交叉参考以验证准确性。这个过程不仅加速了开发，还促进了知识获取，因为开发者深度参与AI的输出和库的复杂性。

让我们将此与传统的人类-人类结对编程进行比较：

- *Human-AI配对*提供快速代码生成，能够高效处理日常任务。这对单独开发者或团队资源有限的情况特别有益。
- *Human-human配对*在复杂问题解决场景中表现出色，在这些场景中，细致的理解和协作头脑风暴是必不可少的。它促进共同所有权和集体代码理解。

两种方法都有其优点，您在它们之间的选择可以根据项目的复杂性、资源可用性和开发过程的具体目标来指导。

AI结对编程的最佳实践

为了最大化AI辅助开发的好处，考虑以下实践：

为不同任务启动新的AI会话
这有助于保持上下文清晰度，并确保AI的建议与手头的特定任务相关。

保持提示专注和简洁
提供清晰和具体的指令可以提高AI输出的质量。

频繁审查和提交更改
定期集成和测试AI生成的代码有助于及早发现问题并保持项目动力。

保持紧密的反馈循环
持续评估AI的贡献，根据需要提供纠正或改进，以指导其学习并改善未来的建议。

AI作为验证器

除了代码生成之外，AI还可以作为有价值的验证器，协助代码审查和质量保证。AI工具可以分析代码中的潜在错误、安全漏洞以及是否遵循最佳实践。例如，DeepCode和Snyk的AI驱动代码检查器等平台可以识别缺少输入清理或不安全配置等问题，在开发环境中直接提供可操作的见解。Qodo和TestGPT等平台可以自动生成测试用例，确保更广泛的覆盖范围并减少手动工作。许多AI工具可以协助监控应用程序性能，检测可能表明潜在问题的异常。

通过将AI验证器集成到开发工作流程中，团队可以提高代码质量，减少缺陷的可能性，并确保符合安全标准。这种主动的验证方法补充了人工监督，导致更强大和可靠的软件。这些工具通过处理重复和耗时的任务来提高QA过程的效率和有效性，让人工测试人员专注于质量保证的更复杂和细致的方面。

将AI融入开发过程，无论是作为结对程序员还是验证器，都提供了增强生产力和代码质量的机会。通过深思熟虑地集成这些工具，开发人员可以利用人类和人工智能的优势。

为了最大化AI和人类能力在QA中的好处，我推荐几个最佳实践：

- 使用AI进行初始评估和初步扫描以识别明显问题。
- 优先考虑对关键区域的人工审查，如复杂功能、用户体验以及AI可能存在局限性的区域。
- 培养持续协作的环境，让AI工具和人工测试人员协同工作，通过持续的反馈循环来改善AI性能和人类决策制定。

Vibe编码的黄金法则

对你想要的内容要具体和清晰
在与AI交互时清楚地表达你的需求、任务和结果。精确的提示产生精确的结果。

始终验证AI输出是否符合你的意图
AI生成的代码必须始终与你的原始目标进行检查。在接受之前验证功能、逻辑和相关性。

将AI视为初级开发人员（需要监督）
将AI输出视为需要你仔细监督的草稿。提供反馈、改进，并确保质量和正确性。

使用AI扩展你的能力，而不是取代你的思考
利用AI自动化常规或复杂任务，但始终积极参与问题解决和决策制定。

在生成代码之前与团队进行前期协调
在开始AI驱动开发之前，与你的团队就AI使用标准、代码期望和实践达成一致。

将AI使用视为开发对话的正常部分
定期与你的团队讨论AI经验、技术、成功和陷阱。将AI标准化为集体改进的另一个工具。

通过单独提交在Git中隔离AI更改
在版本控制中清楚地识别和分离AI生成的更改，以简化审查、回滚和跟踪。

确保所有代码，无论是人类还是AI编写的，都经过代码审查
通过让所有贡献都经过相同严格的审查过程来保持一致的标准，提高代码质量和团队理解。

不要合并你不理解的代码
除非你彻底理解AI生成代码的功能和含义，否则不要集成它。理解对于可维护性和安全性至关重要。

优先考虑文档、注释和ADR
为AI生成的代码清楚地记录理由、功能和上下文。良好的文档确保长期清晰度并减少未来的技术债务。

分享和重复使用有效的提示
记录导致高质量AI输出的提示。维护经过验证的提示库，以简化未来的交互并增强一致性。

定期反思和迭代
定期审查和改进你的AI开发工作流程。利用过去经验的见解持续增强团队的方法。

通过遵守这些黄金法则，你的团队可以有效利用AI，在保持清晰度、质量和控制的同时提高生产力。

[1] 本章基于最初发表在我的Substack通讯*Elevate with Addy Osmani*上的一篇文章，“The 70% Problem: Hard Truths about AI-Assisted Coding”，2024年12月4日。

第2章。超越70%：最大化人类贡献

早期发布读者须知

通过早期发布电子书，您可以获得最早形式的书籍——作者在写作时的原始和未编辑内容——这样您就可以在这些技术正式发布之前很长时间就利用它们。

这将是最终书籍的第4章。

如果您对我们如何改进本书中的内容和/或示例有意见，或者如果您注意到本章中缺少材料，请联系编辑*sgrey@oreilly.com*。

您已经看到像Cursor、Cline、Copilot和WindSurf这样的AI编码助手如何改变了软件构建方式，承担了大部分繁重工作和样板代码——大约70%。[1] 但是，将玩具解决方案与生产就绪系统区分开来的最后”30%“工作呢？这个差距包括困难的部分：理解复杂需求、架构可维护的系统、处理边界情况以及确保代码正确性。换句话说，虽然AI可以生成*代码*，但它经常在*工程*方面存在困难。

Tim O’Reilly在反思几十年的技术变革时提醒我们，每次自动化的飞跃都改变了我们*如何*编程，但没有改变我们*为什么*需要熟练程序员。我们面临的不是编程的终结，而是”我们今天所知编程的终结”，这意味着开发者的角色在演变，而不是消失。

今天工程师面临的挑战是拥抱AI在其擅长领域（前70%）的能力，同时加倍投入剩余30%所需的持久技能和洞察力。本文深入探讨专家见解，以确定哪些人类技能仍然至关重要。我们将探索高级和中级开发者应该继续利用什么，以及初级开发者必须投资什么才能与AI一起蓬勃发展。

因此，本章的目标是为您提供实用指导，以最大化那不可替代的30%的价值，为各个级别的工程师提供可操作的要点。

高级工程师和开发者：与AI一起利用您的经验

如果您是高级工程师，您应该将AI编码工具的出现视为扩大影响力的机会——如果您以正确的方式利用您的经验。高级开发者通常拥有深厚的领域知识、对可能出错情况的直觉以及做出高级技术决策的能力。

这些优势是AI无法单独处理的30%的一部分。本节探讨经验丰富的开发者如何最大化其价值。

成为架构师和主编

让AI处理代码的初稿，而您专注于架构解决方案，然后完善AI的输出。在许多组织中，Steve Yegge写道，我们可能会看到一种转变，即团队只需要”高级助理”，他们”(a)描述要完成的任务（即创建提示），以及(b)审查结果工作的准确性和正确性。“拥抱这种模式。作为高级开发者，您可以将复杂需求转化为AI助手的有效提示或规范，然后用您的批判性眼光审查产生的每一行代码。您实际上是在与AI结对编程——它是快速打字员，但您是大脑。

在审查过程中保持高标准：确保代码符合您组织的质量、安全性和性能基准。通过充当架构师和编辑，您可以防止”高审查负担”压垮您。（一个警告：如果初级员工只是将原始AI输出直接交给您，请推回——建立一个流程，要求他们必须首先验证AI生成的工作，这样您就不是唯一的安全网。）

将AI用作大型计划的力量倍增器

高级工程师经常推动大型项目或处理初级人员无法单独应对的复杂重构。AI可以通过在您的指导下处理大量机械性更改或探索替代方案来增强这些努力。Steve Yegge引入了术语*聊天导向编程*(CHOP)来描述这种工作风格——“通过迭代提示优化进行编码”，AI作为合作者。利用CHOP在您承担的工作上更有野心。

拥有AI协助降低了项目值得投入时间的门槛，因为可能需要几天时间的工作现在可以在几小时内完成。因此，高级开发者可以尝试那些”如果...会很好”的项目，这些项目以前似乎略微遥不可及。

关键是保持引导思维：您决定追求哪些工具或方法，并将各个部分整合成一个连贯的整体。您的经验使您能够筛选AI的建议——接受那些合适的，拒绝那些不合适的。

指导和设定标准

高级工程师的另一个关键作用是指导经验较少的团队成员有效使用AI和遵循永恒的最佳实践。您可能拥有初级人员可能看不到的陷阱的宝贵知识，如内存泄漏、差一错误和并发危险。

随着初级工程师现在可能通过AI生成代码，教会他们如何自我审查和测试这些代码变得很重要。通过展示如何彻底测试AI贡献的代码来树立榜样，并鼓励质疑和验证机器输出的文化。一些组织（甚至包括律师事务所）已经制定了规则，如果有人使用AI生成代码或写作，他们必须*披露并自己验证结果*——而不仅仅是假设高级同事会发现错误。

作为高级工程师，在你的团队中倡导这样的规范：欢迎AI，但需要勤勉。通过这种方式指导初级工程师，你可以减轻一些监督负担，并帮助他们更快地成长到那30%的技能组合中。

继续培养领域专精和前瞻性

你的广泛经验和背景比以往任何时候都更重要。高级开发人员通常拥有关于公司为什么以某种方式构建东西或行业如何运作的历史知识。这种领域专精让你能够发现新手不会注意到的AI错误。

继续投资于深度理解问题领域。这可能意味着跟上业务需求、用户反馈或影响软件的新法规。AI不会自动纳入这些考虑因素，除非你告诉它。当你将领域洞察与AI的速度相结合时，你会获得最佳结果。

同时，利用你的前瞻性来引导AI。例如，如果你知道快速修复会在以后造成维护痛苦，你可以指导AI实现更可持续的解决方案。相信你多年来磨练的直觉——如果一个代码片段看起来”不对劲”或好得令人难以置信，就深入研究。十次中有九次，你的直觉发现了AI没有考虑到的东西。能够预见代码的二阶和三阶效应是高级工程师的标志；不要让AI的便利性钝化这种习惯。相反，将其应用到AI产生的任何东西上。

磨练你的软技能和领导力

随着AI承担一些编码工作，高级开发人员可以将更多精力投入到工程的人文方面。这包括与利益相关者沟通、主持设计会议，以及做出使技术与业务战略保持一致的判断决策。Tim O’Reilly和其他人建议，随着日常编码变得更容易，价值转向决定*构建什么*以及*如何协调*复杂系统。

高级工程师通常是协调和看大局的人。承担起那个角色。自愿编写架构路线图、评估采用哪些工具（AI或其他）、或定义你的组织的AI编码指南。这些是AI无法完成的任务——它们需要经验、人类判断力，通常还需要跨团队共识建设。通过增强你的领导存在感，你确保自己不仅仅是一个代码生成器（可被另一个工具替代），而是指导团队的不可或缺的技术领导者。

简而言之，继续做经验丰富的开发人员最擅长的事情：见树又见林。

AI将帮助你砍伐更多树木，但仍然需要有人决定*砍哪些树*以及*如何*用木材建造稳固的房屋。你的判断力、战略思维和指导能力现在比以往任何时候都更加重要。有效利用AI的高级开发人员比不利用AI的开发人员生产力要高得多——但真正出色的将是那些应用人类优势来放大AI输出的人，而不仅仅是让它自由运行。

正如一位Reddit用户观察到的，“AI是编程力量倍增器”，“大大提高了高级程序员的生产力”。倍增效应是真实的，但被倍增的是你的专业知识。保持这种专业知识敏锐并处于开发过程的中心。

中级工程师：适应和专业化

如果你是中级工程师，你可能面临最大的进化压力。许多传统上占用你时间的任务——实现功能、编写测试、调试简单问题——正变得越来越自动化。

这并不意味着淘汰；它意味着提升。重点从编写代码转向更专业化的知识，下面的章节将探讨这一点。

学会管理系统集成和边界

随着系统变得更加复杂，理解和管理组件之间的边界变得至关重要。这包括API设计、事件模式(event schemas)和数据模型——所有这些都需要仔细考虑业务需求和未来的灵活性。加深你的计算机科学基础，包括获得对以下学科的高级理解：

- 数据结构和算法
- 分布式系统原理
- 数据库内部结构和查询优化
- 网络协议和安全

这些知识帮助你理解AI生成代码的含义，并做出更好的架构决策。

也要学会处理边缘情况和模糊性。现实世界的软件充满了奇怪的场景和不断变化的需求。AI默认倾向于解决一般情况。由开发人员来询问”如果…怎么办？“并探索弱点。

这里持久的技能是批判性思维和前瞻性——列举边缘情况、预测失败并在代码或设计中解决它们。这可能意味着考虑null输入、网络中断、异常用户操作或与其他系统的集成。

构建你的领域专业知识

理解业务上下文或用户环境将揭示通用AI根本不知道的边界情况。经验丰富的工程师习惯性地考虑这些场景。系统地练习测试边界和质疑假设。专精于人类理解仍然至关重要的复杂领域。

通用领域包括：

- 具有监管要求的金融系统
- 涉及隐私问题的医疗保健系统
- 具有严格性能要求的实时系统
- 机器学习基础设施

软件工程特定领域包括前端和后端工程、移动开发、DevOps和安全工程等。领域专业知识提供了当前AI工具所缺乏的上下文，帮助您做出更好的决策，决定在何处以及如何应用这些工具。

掌握性能优化和DevOps

虽然LLMs可以建议基本优化，但识别和解决系统级性能问题需要对整个技术栈的深入理解，从数据库查询模式到前端渲染策略。随着代码生成变得更加自动化，理解系统在生产环境中如何运行变得更加宝贵。专注于以下领域：

- 监控和可观察性
- 性能分析和优化
- 安全实践和合规性
- 成本管理和优化

专注于代码审查和质量保证

随着AI编写大量代码，严格审查和测试这些代码的能力变得更加关键。Steve Yegge强调：“每个人都需要在测试和审查代码方面变得更加认真”。将AI生成的代码视为初级开发人员的输出——您是负责捕获错误、安全漏洞或草率实现的代码审查者。这意味着要加强您在单元测试、集成测试和调试方面的技能。

编写良好的测试是一项持久的技能，它迫使您理解规范并验证正确性。明智的做法是假设在得到证明之前什么都不起作用。正如Builder.io CEO Steve Sewell指出的那样，AI通常产生”功能性但优化糟糕的代码”，直到您通过迭代改进来指导它。

培养测试思维：验证每个关键逻辑路径，使用静态分析或linter，如果AI提供的代码不符合您的质量标准，不要害怕重写。即使您遵循上一章讨论的”AI作为验证器”模式，质量保证也不是简单地外包给AI的领域——这是人类勤勉发光的地方。当软件不按预期工作时，您需要真正的问题解决能力来诊断和修复它。AI可以协助调试（例如，建议可能的原因），但它缺乏对应用程序运行的特定上下文的真正理解。人类测试人员拥有特定领域的知识和对用户期望的理解，这是AI目前缺乏的。这种洞察力在评估潜在问题的相关性和影响时至关重要。诊断复杂错误通常需要创造性的问题解决能力和考虑广泛因素的能力——这些都是固有的人类技能。评估软件行为的伦理影响，如公平性和可访问性，需要人类的敏感性和判断力。

能够推理复杂的bug——重现它、隔离原因、理解底层系统（操作系统、数据库、库）——是一项永恒的工程技能。这通常需要对基础知识（内存和状态如何工作、并发性等）的深刻理解，这是初级开发人员必须通过实践学习的。使用AI作为助手（它可能解释错误消息或建议修复），但*不要盲目依赖它*。方法性地排除故障并在调试时应用第一性原理的技能使优秀的开发人员脱颖而出。这也是一个反馈循环：调试AI编写的代码将教会您下次更好地提示AI或避免某些模式。

学习系统思维

软件项目不仅仅是孤立的编码任务；它们存在于用户需求、时间表、遗留代码和团队流程的更大背景中。AI对大局没有内在的感知，比如您项目的历史或某些决策背后的理由（除非您明确地将所有这些信息输入到提示中，这通常是不现实的）。人类需要承载这种上下文。

这里的持久技能是系统思维——理解系统一个部分的变化如何影响另一个部分，软件如何服务于业务目标，以及所有运动部件如何连接。[2] 这种整体视角让您恰当地使用AI输出。例如，如果AI建议一个巧妙的捷径，但它与监管要求或公司惯例相矛盾，您会发现它，因为您知道上下文。要点是学习项目的背景并阅读设计文档，这样您就可以培养关于什么合适、什么不合适的判断力。

保持适应性——永不停止学习

最后，一个元技能：学习新工具和适应变化的能力。AI辅助开发领域正在快速发展。保持开放心态并学会如何有效使用新AI功能的工程师将保持领先地位——Tim O’Reilly 建议，“渴望学习新技能”的开发者将从AI中获得最大的生产力提升。投入精力深入学习*基础知识*，保持对新技术的好奇心。这种结合使你能够将AI作为工具来利用，而不会对它产生依赖。

这是一种平衡：利用AI加速你的成长，但也要偶尔在没有AI的情况下练习，确保你没有跳过核心学习（一些开发者会定期进行”AI戒断“来保持他们原始的编码技能敏锐）。简而言之，做一个不断学习的工程师——这是任何时代都能保证职业发展的技能。

擅长跨职能沟通

随着实现时间的减少，在业务需求和技术解决方案之间进行转换的能力变得更有价值。能够与产品经理、设计师和其他利益相关者有效沟通的工程师将变得越来越有价值。这里的重点领域包括：

- 需求收集和分析
- 技术写作和文档
- 项目规划和估算
- 团队领导和指导

学习系统设计和架构

中级工程师不再花费数天时间实现新功能，而是可能花时间设计能够优雅处理扩展和故障模式的强健系统。这需要对分布式系统原理、数据库内部机制和云基础设施有深入理解——这些是LLM目前提供有限价值的领域。

练习设计解决大规模真实世界问题的系统。无论代码如何生成，这些技能都保持价值，因为它们需要理解业务需求和工程权衡。

设计一个连贯的系统需要理解权衡、约束和超越编写几个函数的”大局观”。AI可以生成代码，但不会自动为复杂问题选择最佳架构。

整体设计——组件如何交互、数据如何流动、如何确保可扩展性和安全性——是需要人类洞察力的那30%的一部分。这包括：

- 负载均衡和缓存策略
- 数据分区和复制
- 故障模式和恢复程序
- 成本优化和资源管理

高级开发者长期以来一直在磨练这项技能，中级和初级开发者应该积极培养它。从模式和原则的角度思考（如关注点分离和模块化）——这些指导AI生成的解决方案走向可维护性。记住，*稳固的架构不会偶然出现*；它需要有经验的人来掌舵。

使用AI!

记住AI应该成为你工作流程的一个组成部分——这不是需要抵制的东西。将AI融入日常工作的实际方法包括：

- 搭建初始代码结构
- 快速原型和概念验证
- 结对编程进行更快的调试和问题解决
- 建议优化和替代方法
- 处理重复的代码模式，让你专注于架构和设计决策

涉足UI和UX设计

有一种日益增长的说法认为中级软件工程师应该”直接辞职”——纯工程技能将随着AI处理实现细节而变得过时。虽然这个结论被夸大了，但关于工程之外技能（如设计）重要性的讨论值得审视。在2024年12月X上的一个具有代表性的交流中，@nullpointered写道：

> *如果你是一个有三年职业经验的软件工程师：现在就辞职吧。CS领域再也没有工作了。一切都结束了。这个领域在1.5年内就不会存在了。*

@garrytan在转推中回复：

> *学习X设计和产品设计，你将变得比你能想象的更强大。*

成功的软件创造一直需要的不仅仅是编码能力。正在改变的不是工程的消亡，而是纯实现障碍的降低。这种转变实际上使工程判断和设计思维变得更加关键，而不是更少。

考虑一下是什么让Figma、Notion或VS Code这样的应用成功的。这不仅仅是技术卓越——而是对用户需求、工作流程和痛点的深入理解。这种理解来自：

- 用户体验设计思维
- 深入的领域知识
- 对人类心理和行为的理解
- 考虑性能、可靠性和可扩展性的系统设计
- 商业模式对齐

最好的工程师一直都不仅仅是编码者。他们是既理解技术约束又理解人类需求的问题解决者。随着AI工具降低实现的摩擦，这种整体理解变得更加有价值。

然而，这并不意味着每个工程师都需要成为UX设计师。相反，这意味着要发展更强的产品思维能力，与设计师和产品经理建立更好的协作技能。这意味着要更多地思考用户，理解他们的心理和行为模式，并学会做出支持用户体验目标的技术决策。你已经达到了技术优雅的境界：现在需要通过密切关注实际用户需求来平衡这一点。

Tan接着发推说：

> *UX、设计、对工艺的实际奉献将在下一个时刻占据中心舞台*

> *真正制造人们想要的东西。软件和编码不会是门槛因素。在多个领域中成为博学者并且聪明/高效的能力，才能共同创造出色的软件。*

未来属于那些能够架起人类需求与技术解决方案之间桥梁的工程师——无论是通过自己培养更好的设计敏感性，还是通过与专业设计师更有效的协作。

初级开发者：与AI共同茁壮成长

如果你是一名初级或经验较少的开发者，你可能会对AI感到既兴奋又焦虑。AI助手可以编写你自己可能不知道如何编写的代码，这可能会加速你的学习。然而，有标题声称”初级开发者的死亡“，暗示入门级编程工作面临风险。与普遍猜测相反，虽然AI正在显著改变早期职业经验，但初级开发者*并没有过时*。

你需要主动发展技能，确保你贡献的价值超越AI能够产出的内容。通过实现基本CRUD应用程序和简单功能来学习的传统路径将会演变，因为这些任务正变得越来越自动化。

考虑一个典型的初级任务：按照现有模式实现一个新的API端点。以前，这可能需要一天的编码和测试时间。借助AI帮助，实现时间可能会降到一个小时，但关键技能变成了：

- 充分理解现有系统架构，以便正确指定需求
- 审查生成的代码是否存在安全隐患和边界情况
- 确保实现与现有模式保持一致性
- 编写验证业务逻辑的全面测试

这些技能无法通过纯粹的教程学习或AI提示来获得——它们需要生产系统的实践经验和高级工程师的指导。

这种演变为早期职业开发者带来了挑战和机遇。入门级职位的门槛可能会提高，需要更强的基础知识来有效审查和验证AI生成的代码。然而，这种转变也意味着初级工程师可能在职业生涯的早期就能解决更有趣的问题。

以下是如何投资自己以有效处理那30%差距的方法。

学习基础知识：不要跳过”为什么”

很容易对每个问题都依赖AI寻求答案（“如何在Python中做X？”）而从不真正吸收底层概念。请抵制这种冲动。将AI作为导师使用，而不仅仅是答案自动售货机。例如，当AI给你一段代码时，询问*为什么*它选择那种方法，或让它逐行解释代码。

确保你理解数据结构、算法、内存管理和并发等概念，而不总是依赖AI。原因很简单：当AI的输出错误或不完整时，你需要自己的心理模型来识别和修复它。如果你没有主动参与理解AI为什么生成某些代码，你实际上可能学得更少，这会阻碍你的成长。所以花时间阅读文档，从头编写小程序，巩固你的核心知识。这些基础知识是持久的；即使你周围的工具发生变化，它们也会为你服务。

练习不依赖AI安全网的问题解决和调试

要建立真正的信心，有时你必须独自飞行。许多开发者提倡做一个”无AI日”或定期限制AI帮助。这确保你仍然可以仅凭自己的技能解决问题，这对于避免技能萎缩很重要。你会发现这迫使你真正思考问题的逻辑，这反过来使你更好地使用AI（因为你可以更智能地指导它）。

此外，每当你在AI生成的代码中遇到bug或错误时，在要求AI修复之前先*自己*跳进去调试。通过逐步调试器或添加print语句来查看问题所在，你会学到更多。

将AI建议视为提示，而不是最终答案。随着时间的推移，解决任务中那些最后的棘手部分将在AI挣扎的领域建立你的技能——这正是让你有价值的地方。

专注于测试和验证

作为初级开发者，你能培养的最好习惯之一就是为代码编写测试。如果你使用AI生成代码，这一点更加重要。

当你从LLM获得一段代码时，不要假设它是正确的——质疑它。编写单元测试（或使用手动测试）来查看它是否真正处理了需求和边界情况。这实现了两个目标：它捕获AI输出中的问题，并训练你在信任实现之前思考预期行为。

你甚至可以使用AI来帮助编写测试，但*你*要定义测试什么。Steve Yegge关于认真对待测试和代码审查的建议适用于所有级别。如果你培养了仔细验证工作的声誉（无论是否有AI辅助），资深同事会更加信任你，你也能避免他们觉得你只是在向他们”倾倒”有问题代码的情况。

在实际工作中，开始将测试作为开发的组成部分，而不是事后的想法。学习如何使用测试框架，如何进行探索性手动测试，以及如何系统性地复现bug。这些技能不仅让你在30%的工作中表现更好，还能加速你对代码真正运行方式的理解。

记住：如果你发现了AI引入的bug，*你*刚刚做了AI做不到的事情——这就是附加价值。

培养对可维护性的敏感度

初级开发者往往专注于”让它工作”。但在AI时代，获得一个基本的工作版本很容易——AI可以做到这一点。更难的部分（也是你应该关注的）是编写可读、可维护和简洁的代码。

开始培养对良好代码结构和风格的敏感度。将AI的输出与你知道的最佳实践进行比较；如果AI代码混乱或过于复杂，主动重构它。例如，如果LLM给你一个做太多事情的50行函数，你可以将其拆分为更小的函数。如果变量名不清楚，重新命名它们。

本质上，假装你在审查同事的代码，并像同事写的一样改进AI的代码。这将帮助你内化良好的设计原则。随着时间的推移，你会开始以产生更清洁代码的方式提示AI（因为你会指定你想要的风格）。软件维护者（通常在几个月或几年后工作）会感谢你，你将证明你的思考不仅仅是”让它运行”——你在像工程师一样思考。保持事物的可维护性恰好在那个人类驱动的30%中，所以从职业生涯开始就把它作为你的关注点。

明智地发展你的提示和工具技能

不可否认，“提示工程”——有效与AI工具交互的技能——是有用的。作为初级开发者，你绝对应该学习如何向AI提问，如何给它提供适当的上下文，以及如何在提示上迭代以改进输出（本书第2章是一个很好的起点）。这些是可以让你脱颖而出的新技能（许多有经验的开发者也还在摸索！）。然而，记住良好的提示往往是理解问题的代理。如果你发现无法让AI做你想要的事情，可能是因为*你*需要首先澄清自己的理解。把这作为一个信号。

一个策略是在要求AI实现之前，用简单的英语自己概述解决方案。另外，尝试不同的AI工具（Copilot、Claude等）以了解它们的优势和劣势。你越熟练使用这些助手，就越有生产力——但永远不要把它们的输出当作绝对正确的。把AI想象成一个超级充电的Stack Overflow：一个帮助工具，不是权威。

你甚至可以使用AI构建小的个人项目来推动你的极限（“我能在AI的帮助下构建一个简单的Web应用程序吗？”）。这样做会教你如何将AI集成到开发工作流中，这是带入团队的一个很好的技能。只是要与不依赖这个安全网的工作时间保持平衡，如前面提到的。

寻求反馈和指导

最后，一个能加速你成长的持久技能是寻求反馈和向他人学习的能力。如果你忽略AI的建议，它不会生气，但你的人类队友和导师对你的发展是无价的——特别是在软技能、领导力、沟通和处理办公室政治方面。

不要犹豫询问资深开发者为什么他们偏好一种解决方案而不是另一种，特别是当它与AI建议的不同时。与更有经验的同事讨论设计决策和权衡——这些对话揭示了资深工程师的思考方式，这对你来说是宝贵的。在代码审查中，对关于你的AI编写代码的评论要特别接受。如果审查者指出”这个函数不是线程安全的”或”这种方法会有扩展问题”，花时间理解根本问题。这些正是AI可能错过的事情，你想学会捕捉它们。随着时间的推移，你会建立一个考虑事项的心理清单。

此外，寻找结对编程的机会（即使是远程的）。也许你可以与一个在工作流中使用AI的资深人员”结对”——你会观察到他们如何提示AI以及如何纠正它。但更重要的是，你会看到他们如何沟通、领导讨论和处理微妙的团队动态。对反馈开放并积极寻求指导将帮助你从做AI能做的任务成熟到做只有人类能做的高价值任务。在某种意义上，你试图尽可能高效地获得通常伴随经验而来的智慧。这让你不仅仅是房间里的另一个程序员——这让你成为团队渴望保留和提升的那种工程师。

沟通和协作

构建软件是一项团队运动。AI不参加会议（谢天谢地）——人类仍然必须与其他人类交谈以澄清需求、讨论权衡和协调工作。强大的沟通技能和以往一样有价值。练习提出好问题和清楚地描述问题（对同事和对AI都是如此）。

有趣的是，向AI发出提示本身就是一种交流形式；它要求你精确表达你想要的内容。这与核心工程技能重叠：*需求分析*。[3] 如果你能制定清晰的提示或规范，这意味着你已经深思熟虑了问题。

此外，分享知识、编写文档和审查他人代码是AI无法替代的协作技能。在未来，当开发者”与”AI合作时，团队中人与人的协作——确保解决正确的问题——仍然至关重要。一个新兴趋势是，开发者可能会更多地专注于高层设计讨论（通常有AI参与）和协调任务，本质上承担更多指挥者的角色。沟通和领导技能将在指挥者的位置上很好地为你服务。

转变思维：从消费到创造

值得注意的是，在AI时代，初级开发者需要转变思维：你需要从只是*消费解决方案*转向*创造理解*。过去，你可能需要费力地研读文档才能最终编写一个功能；现在AI可以直接给你提供一个解决方案。如果你只是消费它（复制粘贴然后继续），你并没有太多成长。

相反，将AI给出的每个解决方案用作学习案例。剖析它，尝试它，考虑你自己可能如何达到这个解决方案。通过将AI输出视为交互式学习材料而非终极答案，你确保自己——人类——在持续提升。这样，AI不是替代你的成长，而是加速它。

许多专家认为，虽然AI可能会减少对大型初级”编码苦力”团队的需求，但它也*提高了*成为初级开发者的标准。这个角色正在转变为能够有效与AI合作并快速攀升价值链的人。如果你采用上述习惯，你将作为一名初级开发者脱颖而出，不仅仅带来AI能带来的东西（任何公司都可以通过订阅获得），而是带来洞察力、可靠性和持续改进——这些是未来高级开发者的特质。

用持久的工程技能让你的职业生涯面向未来

总之，要在AI增强的开发世界中蓬勃发展，各级工程师都应该加倍投入AI无法（尚未）复制的持久技能和实践。无论我们的工具变得多么先进，这些能力都将保持关键作用。特别地，专注于：

- 加强你的系统设计和架构专业知识
- 练习系统思维并保持对大局的上下文理解
- 磨练批判性思维、问题解决和前瞻性技能
- 在专业领域建立专业知识
- 审查代码、测试、调试和质量保证
- 提升沟通和协作技能
- 适应变化
- 持续学习，保持基础知识扎实的同时获得新技能并更新知识
- 使用AI

这些技能构成了软件工程中的人类优势。它们是持久的，因为它们不会随着下一个框架或工具变化而过期；实际上，AI的崛起使它们更加突出。Simon Willison 论述，AI辅助实际上使强大的编程技能*更加*有价值，而不是更少，因为那些有专业知识的人可以更有效地利用工具。

在不熟练的手中，强大的机器可能是危险的或被浪费的，但在有能力的手中，它是变革性的。在AI时代，经验丰富的工程师就像有了新的先进副驾驶的资深飞行员：旅程可以走得更快更远，但飞行员仍必须引导风暴并确保安全着陆。

软件工程一直是一个持续变化的领域——从汇编语言到高级编程，从本地服务器到云计算，现在从手动编码到AI辅助开发。每次飞跃都自动化了编程的某些方面，但每次开发者都适应了并找到了更多要做的事情。正如Tim O’Reilly指出，过去的创新”几乎总是为开发者带来更多工作、更多增长、更多机会”。AI的崛起也不例外。AI不是让开发者变得无关紧要，而是重塑了成功所需的技能组合。平凡的70%编码变得更容易；挑战性的30%成为我们价值中更大的一部分。

为了最大化那30%的人类价值，专注于永恒的工程技能：深度理解问题、设计清洁的解决方案、审查代码质量，以及考虑用户和上下文。经验丰富的程序员从AI中获得更多，因为他们知道如何引导它以及当它出错时该做什么。那些将这些技能与AI工具结合的人将超越只拥有其中之一的人。实际上，专家之间正在形成的共识是，AI是有技能者的工具：即”LLM(大语言模型)是为专业用户设计的专业工具”。这意味着我们每个人都有责任成为那个”专业用户”——培养让我们有效使用这些新工具的专业知识。

最终，软件工程的技艺不仅仅是编写能工作的代码。而是编写*工作良好*的代码——在真实世界环境中，随着时间推移，在不断发展的需求下。今天的AI模型可以帮助编写代码，但还不能确保代码在所有这些维度上都工作良好。这是开发者的工作。

通过加强上述技能，高级开发者可以继续引领和创新，中级开发者可以深化专业知识，初级开发者可以加速掌握之路。AI将处理越来越多的例行工作，但你的创造力、直觉和深思熟虑的工程实践将把原始输出转化为真正有价值的东西。AI是一个强大的工具，但关键在于我们如何使用它。良好的工程实践、人类判断力和学习意愿将始终是必不可少的。

在实际应用中，无论你是在与一个”积极的初级”AI结对编程来编写函数，还是在审查充满AI生成代码的差异，都不要忘记运用你独特的人类视角。问问自己：这解决了*正确的*问题吗？其他人能够理解和维护这个吗？风险和边缘情况有哪些？这些问题是你的责任。编程的未来确实将涉及更少的手动输入每个分号，更多的是指导和策划——但仍需要有智慧正确执行的开发者掌舵。

最终，出色的软件工程始终关乎解决问题，而不仅仅是编写代码。AI并没有改变这一点：它只是挑战我们将解决问题的能力提升到下一个层次。拥抱这个挑战，你将在我们行业的新篇章中蓬勃发展。

[1] 本章基于我首次在Substack时事通讯《Elevate with Addy Osmani》上发表的两篇文章：《Beyond the 70%: Maximizing the Human 30% of AI-Assisted Coding》，首次发表于2025年3月13日，以及《Future-Proofing Your Software Engineering Career》，首次发表于2024年12月23日。

[2] 要了解更多关于系统思维的内容，请查看Donella H. Meadows著的《Thinking in Systems: A Primer》第2版（Rizzoli出版社，2008年），以及Peter M. Senge著的《The Fifth Discipline: The Art and Practice of the Learning Organization》（Crown出版社，2010年）。

[3] 有关此主题的更多信息，请参见Mark Richards和Neal Ford著的《Fundamentals of Software Architecture》第2版（O’Reilly出版社，2025年）以及Mark Richards、Neal Ford和Raju Gandhi著的《Head First Software Architecture》（O’Reilly出版社，2024年）。

关于作者

Addy Osmani是Google Chrome的高级工程主管，专注于开发者体验、性能和AI驱动的软件开发工具。他拥有超过20年的行业经验，致力于构建网络技术，并撰写了多本关于软件工程最佳实践的书籍。

他在AI驱动的开发者工具方面拥有丰富经验，测试和评估了GitHub Copilot、OpenAI Codex、v0.dev、Cursor和Cline等新兴平台。他在AI辅助软件开发方面的写作影响了成千上万的开发者，他在Google Chrome的领导工作帮助塑造了网络性能和AI增强开发者工作流程的未来。

本书凝聚了他在软件工程方面的深厚专业知识和AI驱动编程助手的实践经验，为开发者提供了将AI集成到日常工作流程中的实用策略，并适应快速变化的软件开发环境。