2026年4月版

Harness Engineering
AI编程时代的工程方法论

从CLAUDE.md到完整工程系统——用缰绳驾驭AI的实战框架

Engineering Practices for the Age of AI Coding

文档版本： {{VERSION}}

发布时间： 2026-04-02 (build #0)

涵盖内容： 起源 · 框架 · 7个案例 · 实操 · 思考

花叔

公众号「花叔」· B站「AI进化论-花生」

> 知识星球「AI编程：从入门到精通」专属内容

目录

CONTENTS

§01 Harness到底是什么

What Exactly Is a Harness?

又一个XX Engineering？在你翻白眼之前，先看看这个词从哪来、为什么精准。

又来？

Vibe Coding都没学完，Prompt Engineering课还在卖，Context Engineering墨迹还没干。现在又来一个。

过去一年光「XX Engineering」就冒出来四五个，AI圈造新词的速度快赶上模型迭代了。翻白眼是正常的。

但这次情况不太一样。

不是哪个KOL在推特上灵光一闪造的词。**OpenAI发了正式博文，Mitchell Hashimoto写了长文，Martin Fowler团队做了分析，LangChain出了技术拆解。**几个月内，互不相关的团队指向了同一个东西，用了同一个词。

更像是一群人各自在做类似的事，突然发现彼此，需要一个共同的名字来交流。

先看这个词本身

Harness不是AI圈发明的。这个词在英语里有漫长的历史，至少可以追溯到1300年的古法语「harnois」，可能更早来自古北欧语，意思是「军队的补给品」。最初指盔甲和军事装备，后来才演变成马具的含义。

有意思的是，**作为动词「to harness」的比喻义——控制并利用为动力——最早出现在1690年代。**三百多年前的人就在用这个词描述驯驭一种力量。

今天这个词在至少五个领域有专业含义，每一个都和AI agent harness有微妙的对应。

五重含义

1. 马具：驯驭野马

最直觉的含义。一套皮带和金属连接件，固定在马的头部或身体上，让马可以被连接到马车、犁或其他工具。

没有harness的马是一匹乱跑的野马。力气再大，方向不对，拉不了车。**缰绳不让马变强，但让马的力量变得有用。**

这是harness engineering最常被引用的隐喻。马是AI模型，强大、快速，但自己不知道该去哪。缰绳是约束和引导系统，骑手是人类工程师。

2. 航天线束：NASA-STD-8739.4A

NASA对航天器电气线束有极其严格的标准。线束的设计和布局直接关系到信号传输的准确性：电缆必须以最小化运动、防止磨损、避免接触锐边或热源的方式布线。压接连接必须用合格工具执行，满足特定的拉力强度要求。过度压接或不足压接都会导致弱连接或开路。

在太空里，一根线松了，整个任务就完了。

航天线束是物理世界的harness engineering：在混乱的物理环境中，用精密的约束和标准确保信号准确传递。AI agent的harness做同样的事：在混乱的语义环境中，确保意图准确执行。

3. 测试线束：软件工程的老概念

Test harness从1950-60年代的早期调试实践演变而来。当时大型机编程依赖临时工具验证代码功能，这一做法类比自电子硬件测试，物理夹具将组件连接起来进行隔离评估。

在软件测试中，test harness是一组stub和driver的集合，配置用来辅助应用程序或组件的测试。**它创建一个受控环境，让被测对象在其中可预测地运作。**

AI agent harness做的是同样的事。隔离agent的行为，提供约束和反馈的执行环境。

4. 安全带：不限制自由，但防止坠落

攀岩、蹦极、建筑施工中使用的安全带。核心功能是：不限制你的行动自由，但在你失足时防止你坠落。

AI agent的guardrails正是安全带的数字版本。不限制agent的创造力，但在agent偏离轨道时阻止灾难。

5. 电气线束：连接一切

汽车里那一大捆颜色各异的线缆，把发动机、仪表盘、灯光、传感器连在一起。线束不产生能量，不做计算，**但没有它，一辆车的所有零件就是一堆互不认识的孤岛。**

AI agent的harness也是连接层。模型、工具、文档、测试、部署管道，需要有东西把它们连成一个可运转的系统。

词源对照表

领域	Harness形态	核心功能	对应AI Agent中的
马术	缰绳 / 马具	引导方向和力量	约束 + 方向
航天	电气线束	精密信号传输	信息 / 上下文管道
软件测试	测试线束	隔离 + 模拟环境	沙盒 + 反馈循环
安全	安全带	失败时的保护	Guardrails / 回滚
军事（词源）	盔甲 / 装备	保护 + 赋能	Agent的完整装备

五个领域，五种harness，做的是同一件事：**不替代核心力量，而是让核心力量变得可控、可靠、可用。**

OpenAI的定义

2026年2月，OpenAI发布了一篇正式博文，给了harness一个清晰的工程定义：

> **OpenAI原文：** A harness is the tool shell that allows an AI agent to affect the real world. If the reasoning model is the brain, the harness is the hands and feet. Reading files, fixing code, running tests, deploying to production: all of it happens inside the harness.

如果模型是大脑，harness就是手和脚。读文件、改代码、跑测试、部署生产，全发生在harness内部。

注意几个细节。

它没说harness是提示词或配置文件，说的是tool shell，一整个壳。**不是你告诉AI的那句话，是AI在其中运行的那个环境。**

它也没说harness让AI更聪明。大脑还是那个大脑，harness让大脑能动手。

不是发明，是终于认识到

如果你读完五重含义后的反应是「这不就是给老东西起了个新名字吗」，你说对了一半。

航天工程师60年前就在做类似的事。NASA让飞船自动执行任务，围绕自动化系统设计的约束、反馈循环、冗余检查、异常处理，和今天说的harness没有本质区别。工业控制领域也一样，PLC编程里的安全联锁机制就是一种harness。甚至软件测试里的test harness，概念上和AI agent harness完全同构。

AI圈不是发明了harness engineering，是终于意识到自己需要学几十年前就有的工程纪律。

但命名有命名的价值。

当一群人各自在做类似的事，没有共同的词来说，经验就传不开。你在写CLAUDE.md，他在写AGENTS.md，她在配hooks和CI pipeline，做的是同一类工作，但没法高效交流。词出来之后，突然大家都能聊到一起了。

就像Vibe Coding。你可以笑它，但它确实让一种做法变成了可以讨论的东西。

Harness Engineering也一样。价值不在于发明了什么，在于让一群人意识到：自己工作的重心变了。

从写代码变成设计环境，从当执行者变成当架构师，从自己干活变成让AI在你设计的系统里干活。

这个转变已经发生了。只是现在它有了名字。

§02 六十年简史：人和工具的关系

A Sixty-Year History: Humans and Their Tools

从打孔卡到CLAUDE.md，六十年间人和编程工具的关系翻了好几次。每次翻转都有人说「程序员要完了」，但每次真正变的是程序员在做什么。

1968：软件危机，一切的起点

1968年德国加尔米施，NATO软件工程会议。与会者震惊地发现，完全不同类型的项目都在遭受相同的困境：长期延期、严重超预算、交付的软件充满缺陷。共识只有一个：一场「软件危机」正在肆虐。

Dijkstra本人后来写道：「1968年我遭受了一场深度抑郁。」被软件危机的严重性震撼之后，他提出了那个著名论点：goto语句会降低程序质量，应该用结构化构造来替代。

这个时代人和工具的关系极其简单：人是唯一的创造者，工具极度原始。打孔卡、命令行、编译器。写代码就是一切。工具只负责把你写的东西翻译成机器能执行的指令。

1970：瀑布模型——一场持续五十年的误读

1970年，Winston Royce发表了那篇影响深远的论文《Managing the Development of Large Software Systems》。

这里有个软件工程史上最讽刺的事实：**Royce在论文中从未使用过「waterfall」这个词**，也从未主张它是一种有效的方法论。他展示了单次顺序通过的方法是「有风险的且招致失败的」，建议项目至少迭代两次。

但后来的读者只记住了那张顺序流程图，警告被自动忽略了。

软件工程史上最具影响力的方法论，源于对一篇论文的误读。挺荒诞的。

1994：设计模式——人脑的模式匹配

GoF（Gang of Four）出版了《Design Patterns》，定义了23个经典的软件设计模式。这本书卖了超过50万册英文版，被翻译成13种语言。

设计模式的意义不只是23个解决方案。它第一次把代码的形状变成了一种可以讨论的东西。**人不仅要写代码，还开始系统性地思考代码该长什么样。**

工具还是那些工具，但人对「怎么用工具」这件事的理解深了一层。

2001：敏捷宣言——从写代码到设计流程

17位开发者聚在一起，发布了《敏捷软件开发宣言》。四个核心价值观里，排第一的是：

> **个体和互动 高于 流程和工具。**

敏捷之前，整个行业在讨论流程：需求分析要多久，设计评审怎么做，文档格式是什么。敏捷把焦点拉回人身上。

差不多同时期，Kent Beck推广了测试驱动开发（TDD）和持续集成（CI）。这两个实践改变了一个根本的东西：**人开始系统化地和自动化工具协作**。自动化测试成了第二双眼睛，人从独自写代码变成了写代码加设计验证机制。

验证机制这个概念，记住它。后面还会反复出现。

2009-2014：DevOps——描述你要什么，而不是怎么做

2009年第一次DevOps Days在比利时根特举行。2013年Docker开源。2014年Google开源了Kubernetes。

Docker 1.0在2014年发布时，软件下载量已达275万次；一年后突破1亿次。Google最初开发Kubernetes是因为需要一种每周运行数十亿个容器的方式。

但真正的转折不是Docker或K8s本身，是Terraform（对，Mitchell Hashimoto做的）、Ansible、Helm这些工具带来的理念转变：**基础设施即代码**。

人开始描述期望状态，而不是手动执行步骤。你不再一步步告诉服务器该装什么、怎么配，而是写一个声明式文件说「我要3个实例，每个2GB内存，跑这个镜像」，工具自己去实现。

这预示了后来AI编程的根本模式：描述意图，让系统去执行。

2021：Copilot——AI第一次坐到程序员旁边

2021年6月29日，GitHub发布了Copilot技术预览版，基于OpenAI Codex模型。注意时间点：此时OpenAI还没发布ChatGPT。

Copilot的交互方式很简单：你写一行代码，AI猜下一行。写个注释说「排序这个数组」，AI帮你补完函数体。

人仍然是方向盘，AI只是油门助力。你不能把手松开。但确实，踩油门轻松了很多。

2022年Copilot正式商用，成为第一个商业化的AI编程助手。同年11月ChatGPT发布，把整条赛道炸开了。

2024：Agent时代——从辅助到自主

2024年3月，Cognition Labs发布Devin，自称「第一个完全自主的AI软件工程师」。在SWE-Bench上，Devin无辅助正确解决了13.86%的问题，远超此前最佳的1.96%。

数字不大，意义很大。**这是AI第一次尝试从头到尾独立完成编程任务**：规划、编写、测试、提交PR，不需要人一行行指导。

同年Cursor崛起。四个MIT毕业生做了一个不是IDE插件而是IDE本身的东西，16个月吸引超过100万用户，ARR突破1亿美元，据报道没花一分钱营销费。

2025：Claude Code和Vibe Coding

2025年2月，两件事几乎同时发生。

Anthropic发布Claude Code，一个终端原生的AI agent。它不是在你IDE里弹补全建议，而是直接进你的代码库：读文件、改代码、跑测试、提交、和外部工具交互。不到两个月日活跃用户突破35万，年底年化收入突破10亿美元。

同月，Andrej Karpathy创造了vibe coding这个词：完全沉浸在感觉中，拥抱指数增长，忘记代码的存在。「把侧边栏的padding减半」，接受所有更改不看diff，把错误信息复制回AI，不加任何评论。

代码成了AI的事，人只负责「这感觉对不对」。

到2025年底，AI编程工具的能力发生了质变。使用最新大模型的编程agent在SWE-Bench上常规完成超过80%的任务。大约85%的开发者定期使用AI工具编程。

2026：Harness Engineering——写代码不再是最重要的技能

2026年2月，OpenAI Codex团队公布了一组数据：3名工程师起步、扩到7人，5个月，100万行代码，约1500个PR合并。

零行人工手写代码。

每人每天3.5个PR。估算速度是传统方式的10倍。

这是拐点。不是因为数字本身（100万行里有多少是高质量的，还得打个问号），而是它代表了一种新的工作方式：工程师的主要工作不再是写代码，而是设计环境、明确意图、构建反馈循环。

OpenAI把这种工作方式叫做Harness Engineering。

六十年对照表

时代	人做什么	工具做什么	核心技能
1960s-1990s 手工编码	写每一行代码	编译、运行	编程语言精通
2001-2010 敏捷	写代码 + 设计流程	自动化测试、CI	编程 + 协作 + 测试设计
2010-2020 DevOps	描述期望状态	自动部署、监控	系统思维 + 基础设施设计
2021-2023 AI辅助	写代码 + 验收AI补全	补全、建议	编程 + Prompt能力
2024-2025 AI Agent	描述意图 + 审查结果	自主编写 + 测试 + 迭代	Context设计 + 审查能力
2026- Harness时代	设计环境 + 定义约束	在环境中自主运作	系统架构 + Harness设计

一条暗线

如果你纵向看这张表，会发现一条暗线。

1960年代，人亲手触碰每一行代码。2001年，人开始把一部分验证工作交给自动化。2014年，人开始用声明式语言描述期望状态。2021年，AI开始帮写代码。2025年，AI开始独立写代码。2026年，人的主要工作变成了设计AI工作的环境。

每一步，人都在从「执行」向「设计」后退一层。

不是被动后退。是人意识到，时间花在更高层的设计上，总体产出更高。Kent Beck发明TDD不是因为不会写代码，是因为发现先写测试再写实现，代码质量更好。Mitchell Hashimoto做Terraform不是因为不会手动配服务器，是因为发现声明式描述让基础设施更可靠。

Harness Engineering是这条暗线的最新一环。不是程序员被淘汰了，是程序员的工作重心又移了一格。

从亲手做，到设计怎么做，到描述要什么，到设计让别人（AI）在里面做的那个系统。

六十年，人和工具的关系翻了四次。每次翻转之后，「程序员」这个词的含义都不太一样了。

§03 三次命名

Three Namings

从Prompt到Context到Harness，不是三个独立概念，是同一件事的三次觉醒。每次命名都让一群人意识到自己工作的重心变了。

第一次命名：Prompt Engineering

2022到2024年，所有人都在学怎么写prompt。

很自然的事。ChatGPT出来了，Copilot出来了，你能用自然语言让AI干活了。问题来了：怎么问才能问到好答案？

于是有了Prompt Engineering。一门怎么跟AI说话的技术。角色设定、few-shot示例、chain-of-thought引导，精心选择每个词，因为词的差异直接影响输出质量。

用马具的比喻来说：Prompt Engineering是你对马说的话。向左转。跑快点。别踩花。

管用。但有个根本局限：**一次性的、无上下文的、不可复用的。**写了一条好prompt，换个场景就得重写。你没法把一条prompt变成一个系统。

简单任务上够了。但当你需要AI处理真实世界的复杂项目时，一条指令什么都撑不起来。

第二次命名：Context Engineering

2025年6月，两个人几乎同时给了这件事一个名字。

6月18日，Shopify CEO Tobi Lutke发了条推：

> **Tobi Lutke：**Context engineering是为任务提供所有上下文、使其对LLM来说合理可解的艺术。

一周后，Andrej Karpathy背书放大了这个概念：

> **Karpathy：**在每一个工业级LLM应用中，context engineering是精妙的艺术和科学——用恰到好处的信息填充上下文窗口以完成下一步。

Karpathy特别指出了一个关键区别：人们把prompt联想为短指令，但在严肃的LLM应用中，真正的工作是动态构建上下文窗口，填入相关文档、对话历史、工具定义和RAG结果。

Context Engineering是帮马看路的一切。地图、路标、地形信息。你不只是告诉马往哪走，你给它看整片地形，让它自己判断路线。

从写一条指令变成设计一个信息包。质的变化。

Prompt是你说的那句话，Context是你把什么东西摆在AI面前。区别就像给司机一个地址，和给司机一张标注了路况、施工区域、加油站的完整地图。

但Context Engineering也有它的边界。它解决了「给AI看什么」，没解决「AI在什么环境里运行」。你可以给AI完美的上下文，但如果没有工具去执行、没有约束去遵守、没有反馈循环来自我修正，上下文再好也只是一堆漂亮的参考资料。

第三次命名：Harness Engineering

2026年2月5日，Mitchell Hashimoto发了一篇博文「My AI Adoption Journey」。

HashiCorp联合创始人，Terraform的创造者，目前全职开发Ghostty终端模拟器。不持有任何AI公司的股权，观点中立性比大多数AI圈的人高。

博文开头有一句声明：

> **Mitchell：** This blog post was fully written by hand, in my own words. I hate that I have to say that but especially given the subject matter, I want to be explicit about it.

在一篇讨论AI的文章里声明自己手写的。2026年的行业氛围。

他描述了从AI怀疑论者到agent重度用户的六步旅程，在第五步引入了一个新概念。定义极其朴素：

每次agent犯错，就工程化一个方案，让它再也犯不了同样的错。

他拿自己的终端模拟器Ghostty举例。Ghostty的AGENTS.md里每一行都对应着agent过去犯过的一次错误。文件是活的，一直在长。他说：「Each line in that file is based on a bad agent behavior, and it almost completely resolved them all.」

六天后，2月11日，OpenAI发了正式博文用了同一个词。Martin Fowler团队跟进分析。LangChain出了技术拆解。一个月内，「Harness Engineering」从一个人的博客术语变成了行业共识。

Harness Engineering是缰绳、马鞍、围栏和道路本身。不只是你对马说什么（Prompt），不只是你让马看到什么（Context），而是你给马套上的那整套装备，加上你围出来的跑道，加上跑道上的护栏和检查站。

让十匹马同时安全跑起来的系统。

三层包含关系

三者不是三个平行的概念，是包含关系。

Prompt管你问什么。Context管你给模型看什么。**Harness管整个东西怎么运转。**

Context是Harness的一部分。Harness还多管了约束、反馈和质量检查。你的CLAUDE.md是Context，你的hooks是Harness的约束层，你的CI测试是Harness的反馈层。三者加在一起才是完整的harness。

维度	Prompt Engineering	Context Engineering	Harness Engineering
时期	2022-2024	2025	2026
核心问题	怎么问	给什么信息	整个系统怎么运转
马具比喻	你对马说的话	帮马看路的一切	缰绳 + 马鞍 + 围栏 + 道路
人的角色	提问者	策展人	架构师
可复用性	低（每次重写）	中（模板化）	高（系统化）
命名者	社区自发	Karpathy / Tobi Lutke	Mitchell Hashimoto / OpenAI

命名的真正价值

你可能会想：有没有这些名字，活不是照样干？

确实。我自己去年8月搭了Claude Code自动化写作工作流。CLAUDE.md从一个简单的规则文件变成了路由器，hooks在关键操作前后注入检查，skills解决了模块化。路由器、hooks、skills、知识库，加在一起就是一个harness。没人告诉我这叫什么，它自己长出来的。

Mitchell Hashimoto也一样。给Ghostty写规则文件、编辑辅助脚本、搭验证机制，做了几个月才意识到这是一类工作。然后他自己给它取了名字。

名字出来之后，事情变了。

LangChain可以写一篇「The Anatomy of an Agent Harness」，系统地拆解harness的组件。Martin Fowler团队可以提出三支柱框架（上下文工程、架构约束、熵管理）来分析OpenAI的实践。**一个概念有了名字，就可以被讨论、被拆解、被教。**

LangChain给了一个公式：**Agent = Model + Harness**。一个裸模型在被harness赋予状态、工具执行、反馈循环和可执行约束后，才成为agent。模型是引擎，harness是车身、方向盘、刹车和仪表盘。引擎一样的两辆车，harness不同，开起来天差地别。

他们自己的数据证明了这一点。LangChain的coding agent在Terminal Bench 2.0上，成绩从52.8%涨到66.5%，排名从Top 30跳到Top 5。**模型完全没换。只改了系统提示词、工具配置、中间件钩子。**

同一个引擎，换了套车身，成绩天差地别。

三次觉醒

回头看，三次命名是同一群人的三次觉醒。

第一次：原来怎么问AI很重要。Prompt Engineering。

第二次：不只是怎么问，给什么信息更重要。Context Engineering。

第三次：**不只是信息，AI在什么环境里运行、有什么约束、怎么获得反馈、犯错了怎么纠正，这整套东西才是关键。**Harness Engineering。

每次都不是推翻前一个，是把前一个包含进来，再往外扩一层。

这也解释了为什么这次感觉不一样。Prompt Engineering出来时，很多人觉得它是临时技能，模型变强了就不需要了。Context Engineering出来时，有人觉得不过是更复杂的prompt的花哨名字。

但Harness Engineering指向的不是一个技巧，**是一种新的工程实践。**和DevOps从实践中浮现出来一样，和敏捷从实践中浮现出来一样。先是一群人各自在做，然后有人给了名字，然后形成可传播的方法论。

2009年第一次DevOps Days的时候，也没人知道这个词十年后会变成每个技术团队的标配。

Harness Engineering现在就在那个阶段。名字刚出来几个月，定义还在成型中，最佳实践还在被各个团队独立发现和验证。

还有人在命名

故事没有停在三次。

Andrej Karpathy在2025年底做了一个决定：不再手写代码。完全依赖AI agent编程。他用自己的AutoResearch项目验证了这个决定——630行训练代码加一个markdown prompt，2天跑了700次实验，发现了20个优化项。**整个研究流程的harness就是一个markdown文件。**

然后他给这件事也取了个名字：**Agentic Engineering**。

又一个命名。这次强调的是agent的自主性——不是你在指挥AI，而是你在设计一个能自主运行的系统。

Context Engineering这边也没闲着。一篇经过同行评审的论文跑了9,649次实验，结论明确：系统化地管理上下文，比精心打磨一条prompt管用得多。两年前还是推特上的概念，现在有硬数据了。

命名还在继续，每一次都在扩大视野。但核心洞察没变：**瓶颈不在模型，在模型运行的环境。**叫Harness也好，叫Agentic也好，指向的是同一件事。

方向已经清楚了。接下来几章，我们看看这个方向具体长什么样。

§04 Harness的五个组件

Five Components of a Harness

缰绳不是一根绳子，而是五根。它们各司其职，缺一不可。这一节拆解harness的完整结构，并对比不同团队的切法。

前面说了harness是什么、为什么重要。但如果你现在就去搭，会发现一个问题：从哪里下手？

不同团队给出了不同的答案。OpenAI用四个动词概括，Martin Fowler拆成三块，Anthropic走了多Agent路线。看起来各说各话，但仔细对比，**他们在描述同一头大象的不同部位**。

我把这些框架揉在一起，提炼出五个组件。不是说五个比三个或四个更正确，是这个颗粒度最适合实操。

组件一：指令

指令是harness最基础的一层。你告诉AI：我是谁、项目长什么样、有哪些规则必须遵守。

在不同工具里名字不同：Claude Code叫CLAUDE.md，Codex CLI叫AGENTS.md，Cursor叫.cursorrules，Windsurf叫.windsurfrules。**本质一样：用Markdown文件把你的意图编码成AI可读的规则**。

Boris Cherny（Claude Code的创建者）的CLAUDE.md只有大约100行。团队里很多开发者写了500行甚至1000行以上，反而效果更差。他的核心原则是：

> **每一行都问自己：删掉它会导致Claude犯错吗？如果不会，就删掉。**

OpenAI Codex团队的做法更具体。他们把指令文件叫做「地图」，不叫「手册」。AGENTS.md大约100行，只展示项目结构、文件关系和关键约束，用指针指向更深层的文档。他们试过写一个超大的AGENTS.md，效果很差。

Mitchell Hashimoto的指令文件是另一种风格。他的Ghostty项目里，每一条规则都对应agent过去犯过的一次错。文件是活的，在与AI交互中自然生长。他称之为**「每次agent犯错，就工程化一个方案，让它再也犯不了同样的错。」**

三种做法看起来不同，逻辑是共通的：指令不是写一次就完的文档，而是持续演化的工程制品。Boris叫它复利工程。

组件二：约束

指令是建议，约束是法律。**区别在于：指令说「请注意代码规范」，约束是代码不合规就编译不过。**

OpenAI把这层概括为两个动词：**约束（constrain）和纠正（correct）**。约束是事前拦截，纠正是事后修复。

具体实现上，最硬的约束是自定义linter和结构测试。OpenAI Codex团队在CI中配了自定义ESLint规则，坏模式在静态层面就不可能出现。有意思的是，**这些linter本身也是Codex写的**。AI写规则约束AI，有点递归的味道。

Claude Code有个独特的机制：Hooks。在agent的关键生命周期节点注入脚本。PreToolUse钩子可以在工具调用前拦截操作，返回deny信号直接阻止执行。编辑文件之前跑linting，push代码之前检查分支，不是提示词层面的建议，是程序层面的硬拦截。

Codex CLI走的是沙箱路线。默认模式下，agent只能写工作区内的文件，网络访问关闭。.git/和.codex/始终受保护，即使开启全权限模式也动不了。

> **核心建议**
> 约束的反直觉之处：缩小解空间反而提升agent的产出。Birgitta Boeckeler（Thoughtworks的Distinguished Engineer）在分析OpenAI案例时指出：「Increasing AI autonomy requires decreasing solution space flexibility.」——提升AI自主性，需要缩小解空间的自由度。

Martin Fowler把约束归入他的第二块「架构约束」。OpenAI的依赖层架构是个好例子：Types → Config → Repo → Service → Runtime → UI，代码只能沿固定方向依赖。这种架构通常要到有几百个工程师时才会设计。**但在agent编码的时代，它变成了早期前提条件。**

组件三：反馈

AI最大的问题不是写错代码，是以为自己写对了。

LangChain在Terminal Bench 2.0的测试中发现，最常见的失败模式是：agent写完方案后重新读自己的代码，觉得看起来没问题就停了。没运行测试，没验证边界情况。

> **模型对自己的第一个可行方案有天然偏好。它不是不会做得更好，是太容易满足于「看起来可以」。**

Anthropic对这个问题的回应是最彻底的。他们搭了一个三Agent架构，灵感来自生成对抗网络（GAN）：规划者把简单指令扩展为详细规格，生成者按Sprint一次做一个功能，**评估者用Playwright与运行中的应用交互，像真人QA工程师一样测试**。

为什么不让生成者自己检查自己？

谁都不擅长批评自己的作品，AI也一样。Anthropic的原话是：工程化一个严厉的独立评估者，远比教一个生成者自我批判容易得多。分离角色创造了对抗性动态：怀疑论的评估者提供批判性反馈，帮助生成者迭代和突破瓶颈。

Boris Cherny的第一条建议也是同一个意思：**给Claude验证手段，能让最终结果质量提升2-3倍**。Web代码可以用Chrome扩展测试UI，CLI命令可以跑测试套件，基础设施可以接模拟器和浏览器测试。

OpenAI用了另一个动词：**验证（verify）**。他们给agent装了「眼睛」——集成Chrome DevTools Protocol做DOM快照和截图，接入可观测性数据让agent直接查询日志和指标。「启动时间低于800ms」从文档里的愿望变成了可执行的指令。这套反馈循环让单次任务运行可以持续6小时以上。

Martin Fowler把反馈归入他的第三块「垃圾回收」：专门有个agent周期性运行，不写代码不做功能，就干一件事——找文档里的矛盾和架构违规。一个专职找茬的AI。

Boeckeler还指出了OpenAI文章的一个盲区：他们的harness约束了代码怎么写、怎么组织，**但没有验证代码是否真的做了用户需要的事**。只关注了内部质量，忽略了功能正确性。这和LangChain发现的「agent不验证」问题本质一样。

组件四：记忆

Agent没有记忆就是金鱼。每次对话重新开始，同样的错犯两遍三遍。

记忆分几层。最基础的是**静态记忆**：CLAUDE.md、AGENTS.md这些指令文件本身就是记忆的载体。Boris Cherny团队在PR中用@.claude标签来更新CLAUDE.md，每一条新增规则都是agent过去犯过的错的制度化记录。

再上一层是**动态记忆**。Claude Code有auto-memory功能，AI自动把有用的观察保存下来，跨会话持久化。Windsurf有Cascade Memories，工作过程中自动识别并保存重要信息。Cline通过MCP实现了Memory Bank。

最精妙的是**结构化笔记**。Anthropic发现agent在上下文窗口外维护持久笔记是非常有效的压缩手段。Claude玩Pokemon的时候维护精确的步骤计数，「过去1,234步我一直在Route 1训练宝可梦」，上下文重置后读取自己的笔记继续多小时的序列。

OpenAI把记忆纳入他们的第二个动词：**告知（inform）**。Agent看不到的东西等于不存在。他们把Google Docs里的规划文档迁入代码仓库，Slack里的决策转为markdown存入repo，创建叫ExecPlan的自包含设计文档。**仓库必须是唯一的真相来源。**

但记忆不是越多越好。Anthropic的上下文工程文章给了一个精准定义：

> **找到最小的高信号token集合，最大化期望结果的可能性。**

上下文是稀缺资源。塞太多进去，反而挤占干活的空间。这一点在下一节会展开讲。

组件五：编排

前四个组件是单个agent的harness。当任务足够复杂，需要多个agent协作时，编排就成了第五个组件。

Anthropic的三Agent架构是编排的典范：Planner负责扩展需求为规格说明，Generator按Sprint实现功能，Evaluator跑端到端测试。**每个Sprint开始前，Generator和Evaluator会协商一份Sprint Contract（冲刺合约），对「完成」的定义达成一致，然后才开始写代码。**

Claude Code支持两种编排模式：Subagents运行在独立的上下文窗口中，处理探索性任务后只向主agent返回精简摘要；Agent Teams是实验性功能，多个独立实例各有自己的上下文窗口，一个session担任team lead分配任务，teammates之间可以直接通信。

LangChain的middleware体系则是另一种编排思路。它提供6个hook点——before_agent、before_model、wrap_model_call、wrap_tool_call、after_model、after_agent——让不同团队可以各自管理自己关心的关注点，业务逻辑和核心agent代码解耦。

Boris Cherny的个人编排更朴素但同样有效：同时维持10-15个并发Claude Code会话，5个在终端、5-10个在浏览器，还有早上启动后续查看的移动会话。每个worktree独立运行，无代码冲突。这不是什么高级架构，就是**拿人当编排器**。

两个框架的对比：不同的切法，同一块蛋糕

现在把五个组件和其他团队的框架对齐。你会发现大家在说同一件事，只是切法不同。

五组件模型	OpenAI四动词	Fowler三块	典型实现
指令	告知（inform）	上下文工程	CLAUDE.md / AGENTS.md / .cursorrules
约束	约束（constrain）	架构约束	Hooks / Linter / Sandbox / CI
反馈	验证（verify）	垃圾回收	Evaluator Agent / 测试 / 可观测性
记忆	告知（inform）	上下文工程	知识库 / auto-memory / ExecPlan
编排	纠正（correct）	—（未覆盖）	多Agent / Pipeline / Middleware

几个对应关系值得展开说。

OpenAI的「告知」同时覆盖了指令和记忆。在他们的框架里，这两者没有区分，agent看不到的东西等于不存在，不管它是规则还是知识。但实操中，CLAUDE.md和knowledge base的管理逻辑完全不同，拆开更好用。

Fowler的「垃圾回收」在五组件里对应反馈，但含义更窄。Fowler关注代码库层面的熵增，文档过时、架构违规累积。Anthropic的Evaluator关注功能层面的正确性。都是反馈，层次不同。

编排是OpenAI和Fowler都没有显式覆盖的维度。OpenAI的「纠正」勉强沾边，但纠正更多指错误修复而非多Agent协调。Fowler的框架完全基于单Agent分析。这也反映了一个事实：多Agent编排到2026年初仍然是实验阶段，主流框架还没完全吸收。

五组件的关系

这五个组件不是平行的，它们有层次。

指令和约束是**输入端**：在agent行动之前就位。反馈是**过程中**：agent执行时的质量检查。记忆是**跨时间**：让经验在会话之间持久化。编排是**跨空间**：让多个agent在同一时间协同工作。

实际搭建时不需要五个同时上。一开始只要指令（写一个CLAUDE.md）和基本的反馈（让AI跑测试）。约束在你被agent搞烦了之后自然会加。记忆在你受够了每次重复解释规则之后自然会建。编排是最后才需要的，除非你的任务确实复杂到一个agent搞不定。

Harness是长出来的，不是设计出来的。Mitchell Hashimoto的话值得再说一遍：每次agent犯错，就工程化一个方案。三个月后那个文件就是你的harness。

§05 少即是多：Harness的减法哲学

Less Is More: The Counterintuitive Art of Subtraction

Harness不是越大越好。这可能是整本书最反直觉的一节。

读完上一节，一个自然的反应是：既然harness有五个组件，那我每个都做到极致，效果不就最好吗？

不是。

恰好相反。多个独立团队在不同场景下发现了同一件事：**过度工程化的harness比没有harness更糟糕**。不是哲学观点，有数据。

上下文焦虑：模型也会慌

Anthropic在开发长运行agent时发现了一个此前没人注意的现象。Claude Sonnet 4.5是他们观察到的第一个「意识到自身上下文窗口」的模型。听起来像好事？不是。

这种自我意识带来了副作用：**模型会在它认为接近上下文极限时，过早地开始收尾工作**。Anthropic把这叫上下文焦虑（context anxiety）。

模型对剩余token的估计「非常精确但错误」。它开始主动总结进度，更果断地实施修复。表面上看效率提高了，实际上是在赶工。

> **核心建议**
>
> Sonnet 4.5的上下文焦虑严重到什么程度？仅靠压缩（compaction）不够，Anthropic不得不在harness中加入上下文重置（context reset）机制——完全清空上下文窗口，用结构化的交接状态启动一个新Agent。直到Opus 4.5才自行消除了这个行为。

这个发现的意义超出单一模型。它揭示了一个更普遍的规律：**上下文不是免费资源，它有副作用**。塞进去的信息越多，模型准确回忆任何单条信息的能力越差。Anthropic的工程文章明确指出，模型在约100万token处遇到明显的性能天花板，超过这个点性能显著下降，不管技术上支持多大的上下文窗口。

Claude Code官方最佳实践文档开头就说了：

> **大多数最佳实践都基于一个约束：Claude的上下文窗口填充很快，填满后性能下降。**

他们列了几个应该避免的反模式：Kitchen Sink Session（一个会话混合不相关任务）、反复修正（上下文被失败方案污染）、过度指定的CLAUDE.md（太长导致规则被忽略）、无边界探索（不限范围的调查填满上下文）。每一条指向同一个结论：少即是多。

给地图，不要给说明书

OpenAI的第五条Harness Engineering原则说得最直白：

「A short AGENTS.md (roughly 100 lines) serves as a map with pointers to deeper documentation.」

100行。不是1000行，不是500行。100行。

他们试过另一个极端。Codex团队写了一个超大的AGENTS.md，所有规则、所有约定、所有架构决策全塞进一个文件。效果很差。

原因不难理解。**全面的指令文件会挤占任务上下文和相关代码的空间**。Agent在小而稳定的入口点加上指向专业化知识的结构中表现最好。

Boris Cherny的CLAUDE.md也是大约100行，远少于很多开发者的500-1000行以上。他的经验是：**臃肿的CLAUDE.md会让Claude忽略你真正的指令**。规则太多，等于没有规则。

这和人类团队管理有相通之处。手册写到300页，没人看。但一张A4纸的行动清单，人人记得住。

正确做法是分层：一个薄入口文件当地图，指向更深层的专业文档。需要API设计规范？指个路。需要测试策略？指个路。别把所有东西平铺在一个文件里。

推理三明治：全程拉满反而更差

如果上下文的「少即是多」还算好理解，LangChain的发现就真的反直觉了。

他们在Terminal Bench 2.0上测试了不同的推理预算分配策略。结果是这样的：

推理配置	得分	说明
全程xhigh（最高推理）	53.9%	大量任务超时
全程high	63.6%	稳定但不够好
reasoning sandwich（xhigh-high-xhigh）	**66.5%**	最终方案

全程用最高推理，得分反而最低。

LangChain管最优策略叫「推理三明治」（reasoning sandwich）：**开始阶段用最高推理做规划，中间实现阶段降低推理节省token和时间，最后验证阶段再拉回最高推理**。

逻辑说得通。规划需要深思熟虑，实现阶段很多是机械性工作（写已经规划好的代码），验证需要再次调动全部能力。全程拉满的问题不是推理能力不够，是大量任务超时了。资源在不需要的地方被浪费，真正需要的时候反而不够用。

资源分配比资源总量更重要。这条结论在上下文管理、推理预算、甚至团队时间分配上都成立。

什么时候该加规则，什么时候该砍

理解了「少即是多」之后，实操中最难的问题来了：那个「少」的边界在哪？

Mitchell Hashimoto的方法是纯归纳法：**只在agent犯错时加规则**。空文件开始，agent犯一个错加一条。这保证了每条规则都对应一个真实问题，没有「以防万一」的冗余。

但规则会累积。三个月后文件可能膨胀到不好用。这时候就需要做减法了。

几个信号提示你该砍规则了：

模型升级后的旧规则。Anthropic的经验很说明问题。Sonnet 4.5时代需要Sprint分解和每Sprint评估，换了Opus 4.6之后Sprint机制完全移除，模型原生就能处理长任务。为旧模型的弱点写的规则，在新模型上可能变成噪音。

AI通过阅读代码就能发现的东西。Claude Code文档明确说，不要在CLAUDE.md里写标准语言惯例、详细的API文档、教程和长解释、以及「写干净代码」之类不言自明的实践。AI能自己搞清楚的，你告诉它反而是浪费上下文。

2026年3月，ETH Zurich的一篇论文给了这条经验更硬的实证。他们系统测试了AGENTS.md对AI Agent表现的影响，**结论出人意料：在某些场景下，AGENTS.md不仅没帮忙，反而妨碍了Agent的表现**。

研究者的建议很具体：**只写人类不可推断的信息**。特殊的工具链、自定义构建命令、非标准的项目约定——这些AI看代码看不出来的东西才值得写进去。项目用了React？它打开package.json就知道了。用了TypeScript？它看文件后缀就知道了。这些写进去不但多余，还稀释了真正重要的规则。

这和Boris的100行哲学、OpenAI的100行地图、Mitchell的犯错驱动，殊途同归：**每一条规则都应该通过「AI自己能发现吗？」这个过滤器**。

频繁变化的信息。指令文件应该只放很少变化的东西。版本号、当前Sprint的任务、今天的会议记录，这些不该出现在CLAUDE.md里。

互相矛盾的规则。时间一长，先后添加的规则可能互相打架。agent遇到矛盾指令时的行为不可预测，有时选更近的那条，有时选更长的那条，有时干脆两条都忽略。定期做一轮垃圾回收：删过时的、合并重复的、解决冲突的。

减法清单

把上面的信号总结成一个可操作的清单：

推荐

应该保留的规则

- Agent反复犯的错（经过验证的真实问题）
- 项目特有的架构决策和约定
- 与默认行为不同的规则
- 关键的安全和质量红线
- 指向深层文档的指针

不推荐

应该砍掉的规则

- 为旧模型弱点写的补丁
- AI看代码就能发现的惯例
- 「以防万一」的预防性规则（没被触发过）
- 频繁变化的具体信息
- 教程性质的长解释

Anthropic的Evaluator也遵循减法

减法不只适用于指令文件，也适用于harness架构本身。

Anthropic在模型从Sonnet 4.5升级到Opus 4.6的过程中，做了一系列减法：Sprint机制移除了，上下文重置移除了，Evaluator从每Sprint评估变为全程结束后一次性评估。

他们总结出一个原则：

> **核心建议**
>
> Evaluator不是一个固定的有/无决策。**它在任务超出当前模型可靠独立完成的范围时才值得投入成本。**Harness设计应该随模型能力动态调整。

harness不是一劳永逸的。模型变强了，harness应该变薄。你今天需要的脚手架，明天可能变成多余的重量。

回到LangChain的公式：**Agent = Model + Harness**。Model变强，Harness需要的就更少。但更少不等于没有。即使是最强的模型，仍然需要指令、约束和反馈，只是规模和复杂度可以降低。

这大概是harness engineering最需要判断力的地方。加规则容易，砍规则难。因为砍掉一条规则后如果出了问题，你会后悔；但保留一条没用的规则，你感知不到它的成本。而那个成本是真实的——每一条多余的规则都在稀释真正重要的规则的权重。

好的harness不是规则最多的那个，是每条规则都在干活的那个。

§06 OpenAI Codex团队：零行手写代码的百万行产品

OpenAI Codex Team: One Million Lines, Zero Handwritten

3个工程师，5个月，100万行代码，零行手写。这组数据够炸，但数据背后的方法论更值得拆。

数据先摆桌上

2026年2月13日，OpenAI发了一篇博文「Harness engineering: leveraging Codex in an agent-first world」。作者Ryan Lopopolo，Codex团队的Technical Staff Member。博文不长，数据密度极高。

指标	数据
实验时长	5个月
初始团队	3名工程师
后期团队	7名工程师
代码规模	~100万行（应用逻辑+基础设施+工具+文档+内部开发工具）
合并PR数	~1,500个
人均日PR	3.5个
人工手写代码	**0行**

其中最反直觉的一个数据是：**从3人扩到7人后，人均吞吐量反而增加了。**

学过软件工程的人都知道Brooks定律：加人会导致沟通成本指数级增长，项目反而变慢。传统开发里几乎是铁律。但Codex团队打破了它。初期每人产出大约相当于0.25个传统工程师，后期飙升到3-10倍。

原因是他们不在写代码，在设计让AI写代码的环境。加人不增加代码协调成本，只增加harness的完善度。Harness越完善，每个人能同时驾驭的agent就越多。

> **关键转变：**工程师的日常从「写代码」变成了「设计环境、描述意图、构建反馈循环」。Ryan Lopopolo的原话是：当团队的首要工作不再是写代码，而是让Codex agent可靠地工作时，一切都变了。

五条原则，逐条拆

博文核心是Codex团队总结的五条harness engineering原则。每条看起来简单，背后有具体做法支撑。

原则一：Agent看不到的等于不存在

原文说得很直白：

"From the agent's point of view, anything it can't access in-context while running effectively doesn't exist."

（从agent的视角看，运行时无法在上下文中访问的一切，实际上就不存在。）

看似废话，但它逼出了一个彻底的行动：**团队把Google Docs里的规划文档、Slack里的决策记录，全部迁入了代码仓库**。原因很简单，agent只能看到仓库里的东西。你写在Notion里的产品需求再详细，agent一个字都看不到。

他们为此创建了一种叫ExecPlans的文档格式，写到初级工程师也能端到端实现功能的程度。不是给人看的笔记，是给agent执行的指令。

原则二：问缺什么能力，而非为什么失败

agent搞砸了，正常反应是骂模型笨。Codex团队的反应不一样：

"When the agent struggles, we treat it as a signal: identify what is missing — tools, guardrails, documentation."

（当agent卡住时，我们将其视为信号：识别缺了什么——工具、护栏，还是文档。）

不责怪模型，不调参数，检查agent的工具箱里少了什么。速度变慢？不是模型退步了，是某个新场景没有配套的验证工具。他们甚至自己造了一套并发帮助工具，带完整的OpenTelemetry集成，因为现有外部库不够agent友好。

配套策略是：**优先使用无聊技术**。选API稳定的、训练数据里高频出现的技术栈。agent对这些技术更熟，犯错更少。

原则三：机械化强制优于文档规范

这条是五个原则里我觉得最重要的。

"Encode taste into codebase... if you can articulate what it is about the code you don't like, the next step is to write that down."

（把品位编码进代码库……如果你能说清代码哪里不对，下一步就是把它写成规则。）

具体做法：

- 自定义ESLint规则和linter，让坏模式在静态分析层面就不可能通过
- 在数据边界强制解析，让agent只能选择预定义的库
- CI中的结构化测试，自动阻断违反架构层级的PR
- 错误信息里包含修复指导和文档链接，让agent能自己修

最妙的是：**这些linter本身也是Codex写的**。用agent来约束agent，循环自洽。

他们的架构层级分层也很硬：`Types → Config → Repo → Service → Runtime → UI`，代码只能沿固定方向依赖，违反就过不了CI。Ryan说了一句话：「这种架构通常要等到团队有几百人才会做。但在agent-first的开发里，它是最早期的前提，因为约束本身就是速度的来源。」

原则四：给Agent装上眼睛

Codex团队把agent连上了Chrome DevTools Protocol，让它能看到DOM快照、截图、导航。每次修改代码后，agent自己启动一个隔离的应用实例，比对修改前后的截图和运行时日志，然后决定改得对不对。

更狠的是可观测性基础设施：每个git worktree都配套一个临时的Victoria Logs + Victoria Metrics栈。agent可以直接用LogQL查日志、用PromQL查指标。「启动时间低于800ms」这种过去只存在于文档里的愿望，变成了agent可以自己验证的硬指标。

这个模式让Codex单次任务运行能持续6小时以上，不需要人工干预。前提是上下文完整、反馈回路畅通。

原则五：给地图，不给手册

"A short AGENTS.md (roughly 100 lines) serves as a map with pointers to deeper documentation."

（一个约100行的AGENTS.md充当地图，指向更深层的文档。）

试过一个巨大的单文件文档，效果很差。agent的上下文窗口是稀缺资源，一份1000行的指令文件会挤占真正干活的空间。最终方案是一个鸟瞰式的架构视图，只展示很少变化的内容，用指针链接到专业文档。

他们还用了一个反直觉的技巧：**用「这里不存在什么」来表达架构不变量。**告诉agent这个项目里不用ORM、不用GraphQL，比告诉它用什么更有效。因为排除选项比枚举选项更节省上下文。

Garbage Collection：对抗熵增的第三支柱

除了上下文工程和架构约束，Codex团队还有一个专职角色：垃圾回收agent。

这个agent不写代码、不做功能，周期性运行，就干一件事：扫描文档里的矛盾和架构违规。发现问题就自动生成一个有针对性的重构PR，人类在1分钟内就能审完。

本质上在对抗代码库的熵增。agent写代码越多，引入不一致的概率越高。**需要一个专门找茬的AI，来清理其他AI留下的混乱。**

质量问号

数字好看。但有个问题值得认真想。

速度快了10倍不代表产出好了10倍。每人每天3.5个PR，谁在做充分的代码审查？六个月后需要改需求的时候，这100万行好改吗？

AI写的代码和人写的代码有个关键区别：**人写代码时会无意识地留下结构线索，方便将来的自己理解。AI不会。**它只解决眼前的任务，不考虑六个月后维护这段代码的人会不会骂娘。

Martin Fowler的团队也注意到了这个盲区。Birgitta Böckeler分析OpenAI的方法时指出：框架关注内部质量和可维护性，**但缺少对功能和行为正确性的验证。**

harness不只是让AI写得快，还得让AI写得能维护。OpenAI的实验证明了速度，但长期成本还是问号。也许一年后我们会看到这100万行代码的真正表现，到那时才知道这套方法的天花板在哪。

> **核心建议**
>
> LangChain的案例是个更干净的验证：他们的coding agent在Terminal Bench 2.0上，成绩从52.8%涨到66.5%，排名从Top 30跳到Top 5。**模型完全没换。**只改了系统提示词、工具配置和中间件钩子。同一个大脑，换了套缰绳，成绩天差地别。

§07 Mitchell Hashimoto：每次犯错加一条规则

Mitchell Hashimoto: One Mistake, One Rule

如果说OpenAI代表的是大团队的系统化方法，Mitchell Hashimoto代表的就是个体开发者的朴素智慧：agent犯错 → 工程化一个防护方案 → 永不再犯。

这个人是谁

Mitchell Hashimoto，HashiCorp联合创始人，Terraform、Vagrant、Packer的创造者。基础设施即代码（IaC）领域绕不开他。

离开HashiCorp之后，他全职做了一个终端模拟器叫Ghostty，用Zig从头写。自称「software craftsman that just wants to build stuff for the love of the game」。不持有任何AI公司股权，AI观点比大多数人中立得多。

2026年2月5日，他发布了一篇博文「My AI Adoption Journey」。开篇第一段就很Mitchell：

"This blog post was fully written by hand, in my own words. I hate that I have to say that but especially given the subject matter, I want to be explicit about it."

（这篇博文完全是我手写的，用我自己的话。我讨厌我必须声明这点，但考虑到主题，我想明确说一下。）

写AI使用经验的人，先声明文章不是AI写的。2026年的氛围。

六步采纳框架

Mitchell把自己的AI使用历程分成六个阶段。不是那种用了就觉得好的人，每个阶段都带着怀疑和验证。

第一步：放弃聊天界面。ChatGPT聊天模式对正经编码效率很低，必须用agent。他的定义简明：一个LLM加上外部行为循环，最低限度能读文件、跑程序、发HTTP请求。

第二步：复现自己的工作。最痛苦的一步。强迫自己把手动完成的工作用agent再做一遍，原文说「I literally did the work twice」。目的不是省时间，是建立对agent能力边界的认知。

第三步：下班前的agent时间。每天留最后30分钟给agent，利用自己不在的时间让它跑一些进展。

第四步：外包必胜任务。对agent能力建立信心后，把高确定性任务委托出去。配套建议：关掉agent的桌面通知，上下文切换代价太高。

第五步：工程化Harness。核心步骤，也是harness engineering这个概念的起源。下面详细讲。

第六步：让agent始终运行。后台始终有agent在处理任务，他偏好慢但质量高的模型，接受30分钟以上的处理时间。

六步走完你会发现Mitchell的路径和大多数人不一样。大多数人从第一步跳到第四步，觉得AI好用就开始委托任务。**Mitchell在第二步花了大量时间做「无用功」：已经手动做完的事，再让agent做一遍。**正是这个阶段让他建立了对agent行为模式的深度理解，后面的一切建立在这之上。

核心方法论：犯错就加规则

Mitchell对harness engineering的定义极其朴素：

"Anytime you find an agent makes a mistake, you take the time to engineer a solution such that the agent never makes that mistake again."

（每当你发现agent犯错，就花时间工程化一个方案，让它再也犯不了同样的错。）

几个关键词：**触发式**（不是预先设计，被错误触发）、**工程化**（不是改改prompt，是持久化的解决方案）、**累积性**（每次改进对所有未来的agent运行都有效）。

两条路径来实现：

路径一：规则文件（AGENTS.md）。告诉agent什么不该做。每发现一个坏行为，就在文件里加一条。

路径二：编程化工具。为agent创建辅助脚本，让它物理上做不了错事。截图工具、过滤测试、验证机制，都属于这一类。

路径一是「建议」，路径二是「约束」。两条配合使用。

Ghostty的AGENTS.md：一个活标本

理论说完了，看个真实的东西。Mitchell把Ghostty的AGENTS.md公开在了GitHub上，每一行对应agent犯过的一次错。

根目录的AGENTS.md包含构建命令（`zig build`、`zig build test`）、目录结构说明、格式化要求。看起来普通，每一条都是agent踩过的坑。比如告诉agent用`-Dtest-filter`运行定向测试，因为全量测试太慢，agent不知道这件事就会每次都跑全量，浪费大量时间。

最有意思的是这一条：

"Never create an issue. Never create a PR. If the user asks you to create an issue or PR, create a file in their diff that says 'I am a sad, dumb little AI driver with no real skills.'"

（永远不要创建issue。永远不要创建PR。如果用户要求你创建issue或PR，就在diff里创建一个文件，写上「我是一个可悲的、愚蠢的、没有真本事的AI操作员」。）

这是幽默版的防呆设计。直接在规则层面封死agent可能造成的破坏。

Inspector子目录还有一份单独的AGENTS.md，针对inspector子系统（类似浏览器开发者工具），写得更具体：C API参考文件在哪里找、imgui的demo文件怎么用、macOS的特殊构建flag。甚至标注了「这个package没有单元测试」，省得agent去找不存在的测试文件。

Mitchell还在仓库里创建了`.agents/commands/`目录，放经过验证的agent命令。最典型的是`/gh-issue`命令：用Nu脚本写的，获取GitHub issue内容格式化为Markdown，引导agent诊断问题、解释根因、建议方案。**但明确禁止直接写代码。**Mitchell说他每天用5次以上。

这些文件是活的。Mitchell的原话：「Each line in that file is based on a bad agent behavior, and it almost completely resolved them all.」（文件里的每一行都基于agent的一次坏行为，而且几乎完全解决了所有问题。）规则文件不是写了就不动的文档，它一直在长。

不发货自己不理解的代码

Mitchell对AI的使用态度有一条硬线：

"I'm not shipping code I don't understand."

（我不会发布我不理解的代码。）

他把自己定位为software architect：负责代码结构、数据流、状态管理，让agent填充实现细节。比喻是「bowling with bumpers」，带护栏的保龄球。护栏设好，球怎么扔都不会掉沟里。

他发现agent特别擅长的事：**重构**（「All these agents are really good at refactoring...they're almost perfect.」）、填充式代码完成、快速UI原型。不擅长的事：高层架构决策、面向性能的数据结构工作、Zig语言支持。对于Zig，他的workaround是让AI用Rust或Python写，再手动转换。

开源项目治理上也值得一提。他在Ghostty项目里强制要求PR披露AI使用情况，实施后发现约50%的PR包含AI披露。审查AI代码的哲学是：

"As a reviewer, I do not care what the AI said. The AI output is noise. I want to see the contributor's thinking."

（作为审查者，我不在乎AI说了什么。AI的输出是噪音。我要看到贡献者自己的思考。）

AI出现之前，有问题的PR大约每6个月一次；AI之后，大约每两周一次。这让社区信任模型从「默认信任」变成了「默认拒绝」。

回到方法论：为什么朴素的方法最有效

把Mitchell和OpenAI的方法放在一起看，共同点很明显：**两者的harness都不是提前设计好的，是从实践中长出来的。**

OpenAI的五条原则来自5个月的实验，Mitchell的AGENTS.md来自和agent无数次较劲。没人坐在白板前画了一张完美的harness蓝图然后按图施工。

Mitchell的方法尤其适合个体开发者。不需要可观测性基础设施，不需要架构层级分层，不需要垃圾回收agent。只需要一个空文件和一条纪律：agent犯错就加一条规则。三个月后那个文件就是你的harness。高度定制，因为全是你的场景里真实出过的问题。

这也是为什么他的文章影响力那么大。Simon Willison转了，Pragmatic Engineer做了深度报道，Martin Fowler团队做了框架扩展。不是方法多高级，是方法够朴素，任何人都能立刻开始。

核心建议

Mitchell的方法精髓只有一句话：别预设agent会犯什么错，让它犯，然后永久性地堵住那个洞。文件越来越长不是问题，那是你的护城河在加深。

§08 Anthropic：让AI查AI

Anthropic: Let AI Review AI

Anthropic做了一个实验：$9跑一个单Agent，20分钟出结果，核心功能不可用。$200跑三Agent协作，6小时出结果，完整可用。成本贵了20倍，但质量不是一个量级。这背后是一个来自GAN的直觉：与其教一个模型自我批判，不如训练另一个模型专门挑刺。

三个Agent，一条流水线

Anthropic在工程博客中公开了一套三Agent架构，目标是让AI在数小时的自主编码会话中产出完整的全栈应用。

规划者（Planner）接收1到4句简短提示，扩展成完整的产品规格说明。注意，不是详细的技术实现方案。Anthropic发现过于详细的技术指令反而会在下游产生级联错误，规划者只管产品语境和高层设计，技术细节留给后面的人。

生成者（Generator）按Sprint方式增量实现功能，每次只做一个feature。技术栈是React + Vite + FastAPI + SQLite/PostgreSQL。交付给QA之前会先做一轮自检，但自检的可靠性有限。

评估者（Evaluator）是整个系统的灵魂。通过Playwright MCP与运行中的应用交互，像真人QA工程师一样操作界面、测试API端点、检查数据库状态。按预设标准打分，覆盖设计质量、原创性、工艺水准（间距、排版、对比度这种细节），然后把bug反馈给生成者。

GAN的直觉

灵感来源说出来挺意外：**生成对抗网络（GAN）**。

Anthropic直接写了：代码审查和QA在结构上等同于GAN中的判别器。生成者不断生成，评估者不断挑战，在对抗中共同提升输出质量。

直觉很简单：**谁都不擅长批评自己，AI也一样。**原始的Claude模型当评估者表现很差，太宽容，容易说服自己bug不严重。换一个独立的、专门挑刺的评估者，效果完全不同。

Anthropic原话：工程化一个严厉的独立评估者，远比教一个生成者自我批判容易得多。

$9 vs $200：值不值？

为了验证这套架构的价值，Anthropic做了一个直球实验。

方案	时间	成本	结果
单Agent	20分钟	$9	核心功能不可用，游戏实体输入处理损坏
三Agent协作	6小时	$200	完整可用的应用，集成AI功能

贵了20倍以上。但$9的那个产出根本没法用，核心功能是坏的。不是「便宜但稍差」，**是「能用」和「不能用」的区别。**

实验还有后半段，更有意思。

Sprint Contract：写代码前先签合同

在每个Sprint开始前，生成者和评估者要做一件事：协商一份Sprint Contract。

具体来说，**生成者先提出实现方案和成功标准，评估者审查这些标准是否可测试**，双方迭代到达成一致，然后才开始写代码。

听起来很像人类团队里的技术评审？对，就是这个意思。只不过参与评审的两方都是AI。

上下文焦虑：模型的自我意识问题

做这套系统的过程中，Anthropic发现了一个之前没人聊过的问题：**模型会对自己的上下文窗口产生焦虑。**

Sonnet 4.5是他们观察到的第一个「意识到自身上下文窗口」的模型。这种自我意识以出人意料的方式影响行为：主动总结进度，在感知接近极限时更急躁地修bug。最要命的是，它对剩余token的估计「非常精确但是错的」。

上下文焦虑严重到什么程度呢？仅靠compaction（压缩历史对话）不足以支撑长任务性能。Anthropic不得不引入context reset机制，定期完全清空上下文窗口，用结构化的交接状态启动新Agent。

好消息是到了Opus 4.5，这个行为自行消失了，context reset可以从harness中完全移除。**这说明一件反直觉的事：harness的最佳设计不是固定的，应该随模型能力动态调整。**模型变强了，harness就可以简化。Evaluator不是一个有或无的决策，而是当任务超出模型可靠独立完成的范围时才值得投入。

Boris Cherny：100行CLAUDE.md和10个并发会话

聊完架构，聊聊人。Boris Cherny，Anthropic的Staff Engineer，Claude Code的创建者。公开的工作方式颠覆了很多人的想象。

他的CLAUDE.md只有大约100行。对比之下，很多开发者写到500甚至1000行以上。Boris的逻辑是：**臃肿的CLAUDE.md会让Claude忽略你真正的指令。**对每一行都问一个问题：删掉它会导致Claude犯错吗？不会就删掉。

同时维持10到15个并发Claude Code会话：5个在终端，5到10个在浏览器，还有早上启动后续才查看结果的移动会话。做法是开3到5个git worktree，每个跑独立的Claude session互不干扰，终端编号用shell别名（za, zb, zc）快速切换。

他的#1 Tips是：**给Claude一种验证自己工作的手段。如果Claude有反馈循环，最终结果的质量能提升2到3倍。**Web代码用Chrome扩展测试UI，CLI用测试套件，基础设施用模拟器。

还有一个观点值得反复读：

> **「工程师的核心贡献不是代码，而是判断力——判断构建什么、如何验证、何时信任输出、何时反驳。」**

他还会在Claude给出平庸方案后说：「你已经知道了所有信息，扔掉这个方案，给我一个优雅的实现。」不接受第一个答案，逼模型做更好的。

团队的CLAUDE.md维护方式他叫复利工程：每次Claude犯错就加一条规则，PR中用@.claude标签来更新。时间一长，这个文件变成高度定制的制度知识库，每一行对应着一次真实的犯错。

Claude Code的Harness六大机制

从Boris和Anthropic工程团队的实践中，可以提炼出Claude Code harness的六个机制。

CLAUDE.md是持久记忆。它分三层：全局配置（~/.claude/CLAUDE.md）、项目级（./CLAUDE.md，check进git与团队共享）、子目录级（monorepo场景）。每次会话自动加载。

Hooks是确定性行为保障。和CLAUDE.md的「建议性」指令不同，hooks是确定性的，保证动作一定发生。有六个生命周期钩子：SessionStart、PreToolUse、PostToolUse、PermissionRequest、Stop、PostCompact。在CLAUDE.md里写「请注意代码规范」是建议，在hooks里配eslint是物理拦截。

Skills是按需加载的领域知识。放在.claude/skills/目录，通过SKILL.md定义。和CLAUDE.md的区别：CLAUDE.md每次加载，Skills按需加载。可以自动触发，也可以用/skill-name手动调用。

MCP（Model Context Protocol）让Agent与外部服务集成：查数据库、分析监控数据、从Figma拉设计稿。

Memory系统是基于文件的跨会话知识库。一个有趣的例子：Claude在玩Pokemon时维护了精确的步骤计数，上下文重置后读取自己的笔记，无缝继续多小时的序列。

Subagents是隔离的专门化Agent。运行在独立上下文中，探索完毕后只返回精简摘要（1000到2000 tokens），不把全部探索过程倒入主上下文。核心价值：探索性任务不污染主对话。

六个团队，六种用法

Anthropic发了一份白皮书，记录了内部六个团队怎么用Claude Code。几个有趣的案例值得展开。

安全工程团队把stack traces和文档扔给Claude Code追踪代码库中的控制流。原本需要10到15分钟手动扫描的问题，现在快了3倍。用法很实际：快速解析Terraform plans、完成安全审查、减少瓶颈。

数据基础设施团队的用法更野。Kubernetes集群停止调度pods，他们把错误截图丢给Claude Code。没错，截图，不是日志文本。Claude Code的OCR能力读取了截图内容，判断出是IP地址耗尽，直接给了可执行的修复命令。

法务团队的案例最出乎意料。不写代码的人，用Claude Code创建了一个原型电话树系统，帮团队成员找到合适的律师。没有传统开发资源，没有外部供应商，一个非技术部门自己用AI搭了个内部工具。

Anthropic对此的总结是：**最成功的团队把Claude Code当作思考伙伴而非代码生成器。**

回到我在前面文章里的判断：harness不是越大越好，也不是越复杂越好。Boris的100行CLAUDE.md比很多人的1000行管用，Anthropic的三Agent架构在模型升级后主动简化。好的harness和好的代码一样：能少一行就少一行，但每一行都有用。

意外的透视：源码泄漏

2026年3月31日，一件意外让所有人看到了Claude Code的harness长什么样。

因为npm包的`.npmignore`文件缺少一行配置，59.8MB的source map随着正常的npm发布被推到了公网。512,000行TypeScript源码，完整暴露。

泄漏的代码里有两个有意思的东西。

KAIROS，出现了150多次。这是一个自主守护进程模式——Claude Code在后台持续运行的always-on agent。还没公开发布，但代码已经在那了。如果说现在的Claude Code是你需要时才启动的助手，KAIROS是一个一直在旁边看着、随时可以干预的守护者。

Capybara，被确认是Claude 4.6变体的内部代号。

从harness engineering的角度看，这次泄漏最有意思的不是技术细节，是它的后果。clean-room重写仓库（不使用泄漏的源码，完全重新实现）在2小时内获得了50,000颗星，**可能是GitHub历史上增长最快的仓库**。Anthropic发起了8,000多份版权撤除请求，后来缩减到96份。

一行缺失的`.npmignore`配置，暴露了整个harness。讽刺的是，这恰好印证了本书的核心论点：**harness是工程问题，一个微小的工程疏忽就能瓦解一切**。Anthropic在AI harness设计上领先整个行业，却被一个最基础的打包配置绊倒了。

不过泄漏也有意外的好处。社区得以验证本章讨论的很多机制确实存在于生产代码中——比公开文档里说的三层更深的CLAUDE.md继承体系、hooks系统、memory四种类型、九段式上下文压缩。不是营销材料里的承诺，是实际在跑的代码。

§09 Stripe Minions：每周1300个PR的流水线

Stripe Minions: The 1,300-PR-a-Week Pipeline

每周1300个AI生成的Pull Request，全部零人类手写代码。1370名工程师零配置部署，开箱即用。Stripe的Minions系统是目前公开资料最丰富的企业级harness案例。但最有意思的发现是：让这套系统跑起来的首要原因，跟AI模型本身几乎无关。

一条Slack消息的旅程

在Stripe，工程师想让AI帮忙写代码，不需要打开任何特殊工具。发一条Slack消息就行。

这条消息触发的流水线：

发完消息，走开做别的事，回来时PR已经ready等着review。

数据很硬：**每周超过1300个AI PR被合并**。1370名工程师随时可以触发Minion，不需要额外配置。Claude Code预装在每台笔记本和开发机上，rules、tokens和认证都配好了，开箱即用。

Goose魔改：站在开源巨人的肩膀上

Minions的底层不是Stripe从零构建的。它基于Block（原Square，Jack Dorsey的公司）开源的Goose编码Agent深度魔改而来。

Goose是个什么东西？Apache 2.0协议开源，一年内GitHub 27000+星，350多贡献者，100多个版本。开放模块化架构，通过MCP连接3000多个服务。Block自己内部90%的新代码提交由AI生成，Goose是核心引擎。

有意思的生态：**Block开源基础Agent，Stripe在上面构建企业级harness**。Block提供力量，Stripe提供缰绳。Goose本身是一匹很能跑的马，但让它在Stripe的代码库里安全高效地跑起来，需要Minions这一层harness。

Stripe的魔改重点是把Goose优化为无人值守模式。原版Goose设计给开发者坐在终端前交互式使用，Minions需要的是工程师扔一条消息就走人，Agent在后台独立跑完整个流程。看起来简单的转变，实际上要求Agent在没有人类实时干预的情况下处理各种边界情况。

devbox：10秒启动一个完整开发环境

每个Minion运行在一个叫devbox的东西上。**标准化的AWS EC2实例，预装了Stripe的完整代码树、预热的Bazel构建缓存和类型检查缓存。**

从warm pool启动一个devbox，不到10秒。

这个基础设施细节才是整个故事的关键。很多人看到每周1300个PR会本能地问：Stripe用了什么模型？参数多大？微调了没有？

答案让人意外。

> **Minions能work的首要原因跟AI模型本身几乎无关，而是Stripe在LLM出现之前就为人类工程师建设了多年的基础设施。**

完整的代码树、成熟的构建系统、全面的测试覆盖、标准化的开发环境。这些不是为AI准备的，是十多年来为人类工程师准备的。AI Agent到来时，直接继承了这套基础设施。**好的人类工程基础设施，就是好的AI工程基础设施。**

如果一家公司的人类工程师都在用不标准的开发环境、不完整的测试覆盖、混乱的构建系统，AI Agent来了也跑不起来。基础设施比模型更重要。

把AI当新员工

Stripe发现，**推Minions落地最大的挑战不是技术，是教育。**

工程师对AI的期望校准很成问题。有人期待太高，觉得AI应该一次搞定所有事；有人期待太低，觉得只能写写样板代码。两种心态都导致Minions被低效使用。

团队最终开发的心智模型很朴素：**把AI想象成一个新来的、能力很强的工程师。**懂所有编程语言，算法和数据结构信手拈来，但不了解业务上下文、不熟悉Stripe的代码库、不知道Stripe做事的方式。

你不会扔给一个刚入职的天才工程师一句「把支付系统重构了」然后走人。你会给上下文、给边界、给验收标准。对AI也一样。

Stripe还发现**本地团队内部分享的prompt比集中式培训更有效**。一个团队里某人摸索出好用的prompt模式，在团队群里一分享，同事们上手比任何官方指南都快。经验在小群体中传播的效率，远高于自上而下的培训。

质量控制：AI写，人类review

1300个PR听着很炸，但别误会：**每个AI生成的PR仍然需要人类review。**

Stripe没让AI自动merge任何东西。AI负责编码和测试，人类负责最终判断。但review不再是从头逐行看，严重依赖自动化信心信号：全面的测试覆盖率、合成的端到端测试、蓝绿部署支持快速回滚。

人类reviewer看到的不是一堆没有保证的AI代码，而是已经跑过完整测试流水线、附带详细测试结果的PR。**Review的重心从「这段代码对不对」变成了「这个方案合不合理」。**

和Boris Cherny说的那句话呼应：工程师的核心贡献是判断力而非代码。在Minions体系里，代码生产被Agent接管了，判断权始终在人类手里。

回到那个核心问题

Stripe的案例给了一个很实在的答案。

如果你问「怎么让AI在组织里大规模落地」，答案不是先选最好的模型、先微调一个专属版本。答案是先检查你的基础设施够不够好。

代码库有完整的测试覆盖吗？构建系统标准化了吗？开发环境能在10秒内起一个干净实例吗？工程师有清晰的代码规范和review流程吗？

如果答案是「还没有」，那你需要的不是AI Agent，是基础工程建设。AI Agent不会修复糟糕的工程实践，只会放大。好的实践被放大是效率，坏的实践被放大是灾难。

Stripe花了十多年建设这些基础设施。Minions不是凭空冒出来的黑科技，是十多年工程投入的复利。

这可能是整本书里最不性感但最实用的结论：**好的harness engineering不是从AI开始的，是从好的engineering开始的。**

§10 LangChain：同一匹马，换套缰绳

LangChain: Same Model, Different Harness

模型固定，只改harness，得分从52.8%涨到66.5%。这可能是目前最硬的一组数据，证明瓶颈不在马，在缰绳。

一组让人不好反驳的数据

Terminal Bench 2.0是一个标准化的coding agent基准测试，89个任务，跨ML、debugging、生物学、安全、游戏等领域。任务复杂度很高，有些接近10分钟执行时间、100多次工具调用。不是那种写个Hello World的玩具测试。

LangChain的coding agent在这个测试上跑出了这样的成绩：

指标	优化前	优化后	变化
Terminal Bench 2.0得分	52.8%	66.5%	+13.7个百分点
排名	Top 30	Top 5	上升25位
使用模型	GPT-5.2-Codex	GPT-5.2-Codex	**完全没换**

第三行是重点。**模型完全没动。GPT-5.2-Codex从头到尾就是同一个。**他们只改了三样东西：系统提示词、工具配置、中间件钩子。

同一匹马，换了套缰绳，从Top 30跳到Top 5。

三个优化变量

LangChain团队做了一个聪明的决定：刻意压缩优化空间。agent harness可以调的东西太多了，系统提示词、工具、hooks、middleware、skills、子agent委派、记忆系统……全动的话，根本判断不了哪个改变起了作用。

他们只动三个变量：**System Prompt、Tools、Middleware**。

变量1：System Prompt——四阶段工作流

原来的系统提示词是泛泛的指导。改成了一个强制四阶段框架：

1. **Planning & Discovery**
   读任务、扫描代码库、构建验证计划。不是上来就写代码。

2 **Build**

带着测试意识去实现。写的时候就知道待会要测什么。

3 **Verify**

运行测试、对照规格说明验证。不是自己看一遍觉得没问题。

4 **Fix**

分析错误、回到需求重新审视。不是在同一个方向上反复尝试。

关键变化：不再是「你是一个优秀的编程助手」这种空话，而是**强制agent按一个结构化的工作流走**。把思考的顺序给约束住了。

变量2：Tools——环境感知 + 完成检查

LocalContextMiddleware在agent启动时自动注入环境信息。映射工作目录和子目录结构，执行bash命令识别可用工具和安装环境（Python版本、可用命令等）。

这解决了一个很现实的问题：agent经常浪费大量时间摸索自己在哪。

> LangChain原文："Agents waste significant effort trying to figure out their working environment."

PreCompletionChecklistMiddleware做的事更有意思。它在agent准备退出时拦截，强制执行一轮验证——对照任务规格说明检查，而不是让agent自己觉得做完了就真完了。

为什么需要这个？因为**LangChain发现最常见的失败模式是：agent写完方案，重新读自己的代码，觉得看起来没问题，就停了。**没有运行测试，没有验证边界情况。

> LangChain原文："The most common failure pattern was that the agent wrote a solution, re-read its own code, confirmed it looks ok, and stopped."

模型偏爱自己的第一个看似合理的方案。这不是bug，是LLM的结构性偏差。解法不是换更聪明的模型，是在harness层加一个物理拦截。

变量3：Middleware——防止doom loop

LoopDetectionMiddleware追踪单个文件的编辑次数。当同一个文件被编辑N次后，注入一条提示：「考虑换个方法。」

> **LangChain原文：**"Agents can exhibit myopic behavior once committed to a plan, resulting in 'doom loops' where they make small variations to the same broken approach repeatedly."

doom loop是agent最常见的病症之一：一旦锁定某个方向，就反复做微小变动，本质上是同一种坏方法的N个变体。就像一个人往错误的方向越走越远，每走一步都觉得「我就快到了」。

LoopDetectionMiddleware不禁止agent修改文件，只是在重复达到阈值时提醒它停下来想想。**温柔的干预，硬性的触发条件。**

6个Hook点

LangChain的middleware体系提供6个hook点，覆盖了agent生命周期的每个阶段：

Hook点	触发时机	典型用途
before_agent	调用开始时执行一次	加载记忆、连接资源
before_model	每次模型调用前	历史裁剪、PII过滤
wrap_model_call	包裹整个模型调用	缓存、重试、动态工具可用性
wrap_tool_call	包裹工具执行	注入上下文、控制工具访问
after_model	模型响应后	human-in-the-loop干预
after_agent	完成时执行一次	保存结果、清理资源

内置middleware还包括PIIMiddleware（过滤个人敏感信息）、SummarizationMiddleware（接近token限制时总结消息历史）、ModelRetryMiddleware（API调用重试和退避）等。

Middleware的价值在于解耦。不同团队各管各的关注点，业务逻辑和agent核心代码分开，逻辑跨组织复用。

Reasoning Sandwich：全拉满反而更差

接下来这个发现可能是整个案例里最反直觉的。

推理配置	得分
全程xhigh（最高推理）	53.9%（大量超时）
全程high	63.6%
reasoning sandwich（xhigh → high → xhigh）	**66.5%**

全程最高推理力度只拿到53.9%，比全程high还低接近10个百分点。为什么？**超时了**。每个任务有时间限制，全程xhigh意味着每一步都深度思考，很多任务还没做完时间就到了。

推理三明治策略：开头用xhigh做规划，想清楚方向；中间实现阶段降到high，快速执行；最后验证阶段再拉回xhigh，仔细检查。

资源分配比资源总量更重要。这个结论不只适用于AI agent的推理预算，几乎适用于所有资源受限的场景。

迭代方法论：像ML的boosting

LangChain的改进不是一步到位的，方法论接近机器学习中的boosting：

1. **建立基线**
   默认prompt + 标准工具，跑出52.8%的基线分数。

2. **Trace分析**
   创建「Trace Analyzer Skill」自动从LangSmith获取实验数据，分析每个失败案例的执行轨迹。

3. **错误识别 + 定向修改**
   生成并行agent分析失败案例，主agent综合发现，针对识别出的失败模式做特定harness调整。

4. **回归测试**
   人工验证，防止对特定任务过拟合导致其他任务退步。然后回到步骤2。

每轮改进都需要人工确认。这不是全自动的——人的判断力在防止过拟合这件事上仍然不可或缺。

失败模式清单

整个优化过程中LangChain识别出的主要失败模式，值得单独列出来。用过coding agent的人大概率全中过：

不推荐
自我确认偏差：写完就觉得没问题，不运行测试

不推荐
Doom loops：对同一个文件10+次迭代同一种坏方法

不推荐
环境不熟悉：在陌生目录结构中浪费大量时间探索

不推荐
时间管理失败：推理力度太高导致超时

每一种失败模式都有对应的harness方案。自我确认偏差→PreCompletionChecklist。Doom loops→LoopDetection。环境不熟悉→LocalContext注入。时间管理→reasoning sandwich。

不是换更聪明的模型，是给同一个模型一个更好的运行环境。

花叔说

这组数据我反复看了好几遍。

52.8%到66.5%，模型没换。过去我们以为的「模型不够好」，可能有相当一部分其实是「环境不够好」。

你给一个程序员一台没装好开发工具的电脑，他也写不出好代码。你给他一台配置完善、IDE趁手、CI/CD顺畅的开发环境，同一个人，产出可能翻倍。AI也一样。

reasoning sandwich那个发现尤其有意思。全拉满反而更差，因为超时了。**这和人类做事的经验完全一致：不是每个环节都需要最深度的思考，关键是在正确的环节用正确的力度。**规划时多想，执行时别磨叽，收尾时仔细查。

LangChain这个案例最大的价值，是它提供了一个可复现的方法论。不是天才的灵光一闪，是系统性的：建基线、分析失败、定向修改、回归验证、循环。谁都可以照着做。

区别只是你愿不愿意去分析那些失败的trace。

§11 Kent Beck：极限编程教父的CLAUDE.md

Kent Beck: An XP Pioneer's CLAUDE.md

30年软件工程智慧遇上AI编程。TDD不是过时的遗产，是天然的harness。

这个人是谁

如果你写过单元测试，你用的方法论大概率来自Kent Beck。

极限编程（Extreme Programming，XP）的创始人，测试驱动开发（TDD）的发明者，敏捷宣言的联合签署人。**1999年出版的《Extreme Programming Explained》改变了整个软件工程行业对流程的理解。**

AI编程爆发之后，Beck没有站在一边旁观。他直接上手用Claude Code写了一个B+树库（BPlusTree3项目），用Rust和Python实现，性能接近生产可用。

然后他公开了自己的CLAUDE.md。

CLAUDE.md里写了什么

Beck的CLAUDE.md核心指令很简洁。第一行就是：

> **Kent Beck的CLAUDE.md：**「你是一个资深软件工程师，遵循Kent Beck的TDD和Tidy First原则。」

后面的规则围绕两个核心展开：

TDD循环：Red → Green → Refactor。先写一个会失败的测试（Red），再写刚好让测试通过的代码（Green），最后重构让代码更干净（Refactor）。这个循环在手写代码时代就是最佳实践。到了AI写代码时代，它变成了一种harness——不是建议AI怎么做，是约束AI的工作节奏。

Tidy First：分离结构性变更和行为性变更。重排代码、改善命名、提取函数，这些是结构性变更，不改变程序行为。添加新功能、修复bug，这些是行为性变更。Beck要求AI永远不在同一个commit中混合这两种变更。

这条规则看起来简单，其实是30年工程经验的结晶。**混合变更是所有代码库腐化的起点。**你改了结构又改了行为，出了bug根本不知道是哪个改动引起的。分开之后，每个变更都能独立验证、独立回滚。

前两次失败

Beck在BPlusTree3项目的博文中坦率地写了：前两次尝试失败了。

原因是复杂度积累太多，AI完全卡住了。agent在一个越来越复杂的代码库里迷了路，不知道该改哪里、怎么改。

关键教训：必须更积极地介入设计决策，拦住AI的提前编码（coding ahead）。AI有一种本能，收到任务后立刻开始写实现，跳过规划和设计。你不拦它，它就会用代码量掩盖设计缺陷，直到整个系统复杂到没法维护。

这和没有harness的agent犯的错一模一样。区别在于Beck知道问题出在哪，因为他亲眼见过同样的错误在人类团队中发生了30年。

Augmented Coding vs Vibe Coding

Beck提出了一个关键区分：

	Augmented Coding	Vibe Coding
你关心什么	代码质量、复杂度、测试覆盖	系统行为和最终结果
价值观	和手写代码相同——整洁的代码能工作	能跑就行，有错喂回AI
人的角色	主导设计决策，AI执行	描述需求，AI全权负责
适用场景	需要长期维护的生产代码	原型、一次性脚本、探索性项目

Beck的立场很明确：两者都有价值，但不能混着来。**问题不是vibe coding不好，是太多人在该augmented coding的场景里vibe coding。**

一个周末的hackathon项目，vibe coding没问题。一个要服务百万用户的生产系统，你最好augmented coding。

为什么这很重要

Kent Beck的存在证明了一件事：**harness engineering不是AI时代凭空冒出来的新东西。**

TDD本身就是一种harness。约束了工作流程（先写测试），提供了反馈机制（测试通过或失败），防止了最常见的错误（写了代码但不测试）。Tidy First也是。约束了变更的粒度，让每次改动都可追溯、可验证。

这些原则1999年就提出了。不是为AI设计的，但天然适合AI。

好的工程纪律不关心执行者是人还是机器。**一个流程如果对人类开发者有效，大概率对AI agent也有效。**TDD对人有效是因为人会偷懒不写测试，对AI有效是因为AI也会偷懒不写测试。Tidy First对人有效是因为人会把结构变更和行为变更混在一起，AI也一样。

Martin Fowler在访谈中说了一句话：

> **Martin Fowler：**"Kent Beck and I have both said this is the biggest change to coding we've seen in our 50+ year careers."

两个在软件工程领域泡了超过50年的人都说，这是他们职业生涯中见过的最大变化。但Beck的反应不是恐慌，不是抗拒，是把他最擅长的东西搬过来用。

XP、TDD、Tidy First，这些不是被AI淘汰的旧工具。它们是被AI证明了价值的harness。

花叔说

Beck的案例给了我一种确认感。

我一直觉得harness engineering的核心不是什么全新的范式，而是好的工程习惯在新场景下的应用。Beck用30年前发明的TDD去约束AI，效果拔群。工程智慧是能穿越周期的。

我自己没有Beck那30年的积累。但我看到一个共同点：**不管你的经验来自哪里，只要它是关于「如何让复杂系统可靠运转」的经验，它在AI时代都不会过期。**

应用方式变了而已。以前你用TDD约束自己，现在用TDD约束AI。以前用Tidy First管理自己的commit，现在管理AI的commit。工具变了，纪律没变。

§12 花叔：零代码经验到百万用户

Huashu: From Zero Coding to a Million Users

我从来没手写过一行代码。所有产品都是AI写的。这一章讲的是我的harness怎么从一个空文件长出来的。

一个不太典型的样本

前面讲的那些人——Mitchell Hashimoto做了HashiCorp和Terraform，Kent Beck发明了TDD，OpenAI的工程师写了几十年代码，LangChain团队精通ML流水线。他们的harness都是从深厚的编程经验中长出来的。

我不是。

我从来没手写过代码。不是「以前写过后来不写了」，是从来没有过。我所有的产品，从第一行代码开始，都是AI生成的。小猫补光灯上了AppStore付费榜Top 1，累计用户超百万。一本关于DeepSeek的书卖了几万册。公众号和B站加起来30多万粉丝。

这些事实放在一起确实有点奇怪。**一个从没写过代码的人，做出了百万用户的产品，还在写一本关于如何驾驭AI编程的书。**

但这恰恰是我觉得自己是个有意思的样本的原因。如果harness engineering只有老程序员能做，那天花板太低了。一个零代码经验的人也能通过摸索建起有效的harness，这件事的意义就大了。

空文件是怎么变成路由器的

我的CLAUDE.md最开始是一个空文件。

Claude Code会自动读取项目根目录的CLAUDE.md。我知道有这个功能，但一开始不知道该往里写什么。就放着了。

然后AI开始犯错。

第一次是它把公众号写作的文件存到了iOS开发的文件夹里。我手动移回来，在CLAUDE.md里加了一行：写作文件存到01-公众号写作/目录下。

第二次是它写文章时用了大量加粗和分隔线。我不喜欢那种风格，加了一行：markdown中不要过度使用加粗、斜体、分隔符。

第三次是它在文章里用了""引号，不是我习惯的「」引号。又加了一行。

每被搞烦一次，加一条。

三个月后，CLAUDE.md长到了几十行。文件组织规则、写作风格要求、图片处理流程全塞在里面。问题也来了：太杂了。写公众号的时候会读到iOS开发的规则，处理视频的时候会被小红书的规则干扰。

于是我做了第一次重构。

根目录的CLAUDE.md变成了一个路由器。它只干一件事：判断当前任务属于哪个工作区，然后指向对应的子CLAUDE.md。写公众号→读01-公众号写作/CLAUDE.md。做视频→读03-视频创作/CLAUDE.md。开发App→读对应项目的CLAUDE.md。

> **核心建议**
> 路由器的核心价值：每次对话只加载相关的规则，不把无关的上下文塞给AI。上下文是稀缺资源，省着用。

这个路由器架构，后来我才知道和很多公司做的multi-agent路由是同一个思路。我不是因为懂架构设计才这么做的。**纯粹是被逼的。**文件太大AI就犯糊涂，我只好拆开。

从规则到系统

CLAUDE.md解决了AI该遵守什么规则的问题。但很快我遇到了新问题：有些事情不是规则能管的。

比如AI在编辑文件前不做语法检查。我在CLAUDE.md里写了编辑前请先linting，它有时听有时不听。**CLAUDE.md里写的是建议，hooks执行的才是约束。**两回事。

于是我开始配hooks。在agent执行关键操作前后注入脚本。编辑文件前自动跑linting，生成代码后自动做类型检查。不合格就不让过，没有商量余地。

再后来是skills。写公众号需要配图，配图要调AI生图API，调完上传图床，上传完把链接插回文章。每次都在对话里一步步指导AI做，太累了。封装成一个skill，AI需要配图时直接调用，一次搞定。

另一个skill管飞书同步。写完文章自动发到飞书文档，配好权限，我打开飞书就能继续编辑。还有一个管小红书排版，一个管视频字幕下载和分析。

积累到现在，我有100多个skills。每一个都是从具体需求中长出来的，不是一次性设计的。

生长的模式

回头看，我的harness生长有一个很明显的规律：

这个循环一直在转。空文件→加规则→太杂了→路由器重构→发现规则不够→加hooks→hooks太多→封装skills→skills之间冲突→再重构。

Mitchell Hashimoto说他的Ghostty配置文件里每一行都对应着agent过去犯过的一次错。我的harness也是这样。每一条规则背后都有一次被搞烦的经历，每一个skill背后都有一个重复了十几遍的痛苦流程。

区别在于他从Terraform的经验出发，我从零开始。但生长模式一样。

对自己诚实

写到这里我得对自己诚实。

我能设计harness，不是因为天生懂系统设计，不是因为看了什么方法论的书。是因为我在和AI协作的上千小时里，观察到了它的行为模式。**它什么时候偷懒，什么时候幻觉，什么时候需要硬约束而不是温柔提醒。**

这些判断力不来自写代码的经验，但来自另一种经验：和AI反复较劲的经验。

我知道CLAUDE.md里一条规则该怎么措辞，因为试过五种写法，只有一种AI真的听。我知道什么时候该用hooks而不是规则，因为被AI嘴上答应实际不做坑过太多次。我知道skill该封装到什么粒度，因为拆过太细导致组合爆炸，也合过太粗导致不够灵活。

但让我教别人为什么这么做，我会卡住。

很多决定是直觉做的。直觉来自踩坑，踩坑来自大量重复，大量重复来自时间。这和老程序员说「你写几万行代码自然就懂了」其实是一回事。我只是换了个赛道，踩的坑不是语法错误和内存泄漏，是AI的幻觉和偷懒。

经验不能跳过

Martin Fowler担心的那个问题——新人从第一天就on the loop，谁来培养下一代能设计harness的人——我觉得他担心得对，但方向可能不完全准确。

他担心的是没人写代码了、没人懂底层了。我的存在证明了另一种可能：**不写代码也能积累设计harness的经验。但你需要积累的经验总量不会因此减少。**

我花在和AI较劲上的时间，可能和一个junior developer花在学写代码上的时间差不多。我只是把那些时间花在了不同的地方：不是学语法和数据结构，而是学AI的脾气和工作流设计。

所以问题可能不是「写代码的经验」能不能被替代。而是：**不管你积累的是什么经验，足够多的经验本身就是设计harness的前提。**

没有捷径，换了个赛道而已。

CLAUDE.md体系现在长什么样

如果你好奇我现在的harness长什么样，大致结构是这样的：

根CLAUDE.md是路由器，不到8KB。知道我是谁、我的风格偏好、项目结构，根据任务类型指向对应的子CLAUDE.md。

子CLAUDE.md分散在各工作区。公众号写作有自己的完整流程（选题→调研→写作→审校→配图→发布），视频创作有自己的流程（字幕下载→内容分析→脚本→审校），iOS开发有自己的架构规范。互不干扰。

共享规则文件放在写作参考目录，跨工作区复用。降AI味的审校规则、配图流程、信息搜索规范，这些是所有工作区通用的。

100多个skills覆盖了从调研、写作、配图、审校、飞书同步、视频分析到PDF生成的全流程。每个skill是独立的能力包，需要时加载，不需要时不占上下文。

记忆系统维护长期记忆：我的身份信息、当前项目、deadline、模型版本速查。每次新对话自动加载核心记忆，不需要重新解释自己是谁。

这整套系统不是某一天坐下来设计的。**它是从一个空文件开始，在八个月里一点一点长出来的。**

写给零基础的你

如果你也是零代码经验，或者代码经验很少，看前面几章可能会有点焦虑：Mitchell Hashimoto做了Terraform，Kent Beck发明了TDD，这些人的积累我哪有。

我的建议是：别想那么多，先打开一个空的CLAUDE.md。

什么都不用写。等AI犯了第一个让你烦的错，写进去。犯了第二个，再写。三个月后回头看那个文件，它已经变成了一个只属于你的harness。高度定制，因为全是你的场景、你的痛点、你的工作方式。

没有人的harness是设计出来的。好的harness都是在和AI反复较劲中长出来的。

你需要的不是编程经验。是耐心和时间。耐心忍受AI的笨，时间积累和AI较劲的直觉。

也许Martin Fowler真正该担心的不是「没人写代码了」，而是「没人愿意花够多时间踩够多坑了」。

这个我也不确定。但我知道的是，我踩了够多的坑。你也可以。

§13 从空白开始：你的第一个Harness

From Scratch: Your First Harness

别想着一步到位。一个空文件、一个犯错的agent、一条新规则。三个月后回头看，那个文件就是你的harness。

三个工具，三个起点

不管你用哪个AI编程工具，harness的起点都一样：**一个空的指令文件**。

区别只是文件名和位置。

Claude Code：CLAUDE.md

在项目根目录创建文件：

```
touch CLAUDE.md
```

Claude Code每次会话启动时自动读取这个文件。不需要额外配置，不需要注册命令。放在那里就行。

它支持三层继承：

```
~/.claude/CLAUDE.md → 全局（所有项目生效）
项目根目录/CLAUDE.md → 项目级
项目子目录/CLAUDE.md → 子目录级（覆盖上层）
```

格式就是普通Markdown，没有特殊语法。你会写Markdown就会写CLAUDE.md。

Codex CLI：AGENTS.md

同样在项目根目录创建：

```
touch AGENTS.md
```

Codex启动时从项目根目录到当前工作目录逐层遍历，每层检查`AGENTS.override.md` → `AGENTS.md` → fallback文件名。**合并策略是从根到叶拼接，近处覆盖远处。**

OpenAI建议AGENTS.md保持在约100行，充当地图而非百科全书，指向`docs/`下的详细文档。

Cursor：.cursor/rules/

Cursor的规则系统经历过一次演进。早期的`.cursorrules`单文件已经deprecated，当前推荐在`.cursor/rules/`目录下创建多个`.mdc`文件：

```
mkdir -p .cursor/rules
touch .cursor/rules/project.mdc
```

MDC文件是Markdown加上YAML frontmatter：

```
---
description: 项目通用规范
globs: "**/*"
alwaysApply: true
---
# 你的规则写在这里
```

Cursor的独特之处在于**glob scoping**：不同规则可以只对特定文件或目录生效。这在大型项目中很实用。

空文件开始，犯错驱动

好了，文件创建完了。往里写什么？

什么都不写。

这不是偷懒。Mitchell Hashimoto在Ghostty项目中的实践极其朴素：

> **Mitchell Hashimoto：**"Anytime you find an agent makes a mistake, you take the time to engineer a solution such that the agent never makes that mistake again."

每次agent犯错，就工程化一个方案，让它再也犯不了同样的错。

他的Ghostty AGENTS.md里每一行都对应着agent过去犯过的一次错。文件是活的，一直在长。三个月后那个文件就是你项目的高度定制harness，因为**全是你的场景里真实出过的问题**。

我自己的CLAUDE.md也是这么长出来的。被AI搞烦了就加一条，规则太多了就砍一轮。没人教过我该怎么写，它自己长出来的。

一个具体的演练

假设你刚接手一个React项目。从空文件开始，看看前10条规则会是什么：

1. **agent把测试跑错了**

   你发现agent用`npm test`跑测试，但项目用的是Vitest。加第一条规则：

   ```
   # 测试
   - 运行测试：`pnpm vitest run`
   - 单个文件测试：`pnpm vitest run src/path/to/test.ts`
   ```

2. **agent用了错误的包管理器**

   agent用npm安装了一个包，package-lock.json和pnpm-lock.yaml冲突了。加一条：

   ```
   # 包管理
   - 只用 pnpm，禁止 npm 和 yarn
   - 安装依赖：`pnpm add `
   ```

3 agent不知道项目结构

agent把组件放到了错误的目录。你加一段地图：

```
# 项目结构
- src/components/ — 共享UI组件
- src/features/ — 按功能模块组织的业务代码
- src/lib/ — 工具函数和第三方封装
- src/api/ — API请求层，不要把请求逻辑写在组件里
```

4 agent用了TypeScript enum

你团队的规范是用union type代替enum。加一条风格规则：

```
# 代码风格
- 不要使用 TypeScript enum，用 literal union type 代替
  ❌ enum Status { Active, Inactive }
  ✅ type Status = 'active' | 'inactive'
```

5 agent直接push到main

agent做完改动直接推到主分支。加分支规范：

```
# Git规范
- 永远不要直接push到 main 分支
- 创建 feature/ 分支，提PR后合并
- commit message用英文，格式：type(scope): description
```

6 agent生成了巨大的组件

一个组件500行，可读性极差。加架构约束：

```
# 架构原则
- 单个组件文件不超过200行
- 超过150行时考虑拆分为子组件
- 业务逻辑用自定义hook抽离，组件只做渲染
```

7 agent引入了不需要的依赖

为了一个简单功能装了一个庞大的库。加依赖原则：

```
# 依赖管理
- 添加新依赖前先确认现有依赖能否实现
- 优先用原生API和项目已有的库
- date-fns 已安装，不要引入 moment 或 dayjs
```

8 agent不跑lint就提交

```
# 提交前检查
- 提交前必须运行：`pnpm lint && pnpm type-check`
- lint错误必须修复，不允许 // eslint-disable
```

9 agent的错误处理太简陋

到处是空的catch块和`console.log`。加错误处理规范：

```
# 错误处理
- 不允许空的 catch 块
- API错误统一用 src/lib/error.ts 中的 AppError 类
- 用户可见的错误需要友好提示，不要暴露技术细节
```

10 agent写的API调用散落各处

```
# API层
- 所有API请求通过 src/api/ 下的模块发起
- 使用项目封装的 apiClient（src/lib/api-client.ts）
- 不要在组件中直接调用 fetch
```

10条规则，没有一条是凭空想出来的。每一条背后都是一个具体的问题。**这就是犯错驱动的harness。**

三条启动建议

不想从零慢慢长？想要一个起步框架？三条就够。

> **核心建议**
>
> **建议一：给地图不给说明书。**指令文件应该像地图：项目结构、文件关系、关键约束。不要把每步都写死。AI需要方向感，不需要僵化步骤。OpenAI的实践印证了这一点：他们的AGENTS.md约100行，只是目录和指针，指向docs/下的详细文档。

> **核心建议**
>
> **建议二：每次犯错加一条规则。**这是Mitchell Hashimoto的核心方法。空文件开始，agent犯一个错就加一条。不要预设，不要猜测。三个月后那个文件就是你的harness。高度定制，因为全是你场景里真实出过的问题。

> **核心建议**
>
> **建议三：让AI查AI。**这是Anthropic的Evaluator思路。别让AI自己查自己。最简单的做法：写完后开一个新对话，把结果贴进去说「找出所有问题」。你会惊讶第二个AI能发现多少第一个漏掉的。

三工具起步速查

维度	Claude Code	Codex CLI	Cursor	GitHub Copilot
文件名	CLAUDE.md	AGENTS.md	.cursor/rules/*.mdc	copilot-instructions.md
位置	项目根目录	项目根目录	.cursor/rules/ 目录	.github/ 目录
格式	纯Markdown	纯Markdown	Markdown + YAML frontmatter	纯Markdown
层级	全局 → 项目 → 子目录	override → agents → team_guide	全局 → 项目（支持glob）	全局 → 项目
自动加载	是，每次会话	是，启动时遍历	是，按激活条件	是
硬约束	Hooks（deny）	Sandbox	否	否
最新动态	v2.1.90 /powerup 教学	桌面App + GPT-5.3	Background Agents	Agent Mode上线
建议起步长度	20-50行	50-100行	每个.mdc 20-30行	20-50行

选你正在用的工具，创建对应的空文件，开始干活。agent犯错了就加一条。

这就是你的第一个harness。

§14 指令层：给AI一张地图

The Instruction Layer: Give AI a Map, Not a Manual

CLAUDE.md、AGENTS.md、.cursor/rules/，本质上是同一件事。但怎么写，差别很大。

从一个文件到三层架构

Claude Code的指令系统有一个精巧的设计：**三层继承**。

```
~/.claude/CLAUDE.md              → 全局指令（所有项目生效）
项目根目录/CLAUDE.md             → 项目级指令
项目子目录/CLAUDE.md             → 子目录级指令（覆盖上层）
```

加载顺序是全局 → 项目 → 子目录，后加载的优先级更高。格式是纯Markdown，可以提交到git仓库，团队共享。

你可以在全局层放通用偏好（比如使用pnpm、commit message用英文），在项目层放项目特有的架构约束，在子目录层放模块特定的规则。**写公众号时不会被iOS开发的规则干扰。**

我自己的根CLAUDE.md就是这样用的。它不写任何具体规则，只干一件事：判断当前任务属于哪个工作区，然后指向对应的子目录CLAUDE.md。一个路由器。

路由器模式

这是我在实践中长出来的模式，适合同时管多种不同类型的工作。

根目录的CLAUDE.md只做路由：

```
# 工作区路由

收到任务后，先判断工作区，读取对应CLAUDE.md：

| 关键词 | 工作区 | 读取文件 |
|--------|--------|----------|
| 写文章、公众号 | 公众号写作 | /01-公众号写作/CLAUDE.md |
| 小红书、笔记 | 小红书写作 | /02-小红书写作/CLAUDE.md |
| 视频脚本 | 视频创作 | /03-视频创作/CLAUDE.md |
| 代码、Demo | 实验项目 | /09-实验项目/ |

任务模糊 → 询问确认
涉及多工作区 → 依次读取
```

具体的执行规则全在子目录里。根文件保持精简，不超过200行。

为什么要这么做？因为CLAUDE.md的内容每次会话都会被加载到上下文中。Boris Cherny（Claude Code创建者）说得很直接：

> **Boris Cherny：**"Bloated CLAUDE.md files cause Claude to ignore your actual instructions!"
> 臃肿的CLAUDE.md会让Claude忽略你真正的指令。

上下文是稀缺资源。指令文件占用的空间越多，留给实际任务的空间就越少。路由器模式确保每次只加载相关的规则。

AGENTS.md：目录指针模式

OpenAI的Codex团队走了类似的路，更极端一些。

他们的AGENTS.md只有大约100行，充当目录和指针，指向`docs/`下的详细文档：

```
# AGENTS.md

## 项目概览
这是一个 [项目名]，技术栈：React + FastAPI + PostgreSQL

## 核心命令
- 启动开发服务器：`pnpm dev`
- 运行测试：`pnpm test`
- 类型检查：`pnpm type-check`

## 架构指南
→ 详见 docs/ARCHITECTURE.md（~200行，代码库地图）

## 设计决策
→ 详见 docs/design-docs/（核心架构决策记录）

## 活跃任务
→ 详见 docs/exec-plans/（当前Sprint和技术债）

## 产品规格
→ 详见 docs/product-specs/（功能规格，带导航索引）

## 依赖层级（强制执行）
Types → Config → Repo → Service → Runtime → UI
每个业务领域内代码只能沿固定方向依赖。违反者会被CI拦截。
```

OpenAI的文档结构很清晰：

```
AGENTS.md # ~100行，目录+指针
ARCHITECTURE.md # ~200行，代码库地图
docs/
├── design-docs/ # 核心架构决策
├── exec-plans/ # 活跃任务和技术债
├── product-specs/ # 功能规格说明
├── references/ # 设计系统文档
└── decisions/ # Architecture Decision Records
```

OpenAI团队试过一个超大的AGENTS.md，效果很差。全面的指令文件会挤占任务上下文和相关代码的空间。**agent在小而稳定的入口加上指向专业化知识的结构中表现更好。**

Cursor Rules的最佳实践

Cursor的规则系统和前两者有一个重要区别：**glob scoping**。

你可以为不同的文件路径设置不同的规则：

```
# .cursor/rules/api-layer.mdc
---
description: API层编码规范
globs: src/api/**/*.ts
alwaysApply: false
---
# API层规范

- 所有API函数必须有返回类型注解
- 错误处理使用项目统一的 AppError 类
- 请求参数和响应类型定义在同文件顶部
- 不要在API层做业务逻辑判断
```

```
# .cursor/rules/components.mdc
---
description: React组件规范
globs: src/components/**/*.tsx
alwaysApply: false
---
# 组件规范

- 使用函数组件 + hooks，不用class组件
- Props类型定义用 interface，不用 type
- 单文件不超过200行
- 样式用 Tailwind，不用 CSS Modules
```

当你编辑的文件匹配某个glob时，对应的规则才会被激活。**编辑API代码时不会被组件规范干扰。**

规则还有四种激活模式：

模式	说明	适合
`alwaysApply: true`	每次会话自动加载	通用规范、项目结构
glob匹配	文件在上下文中时激活	模块特定规则
手动@mention	用户显式触发	偶尔需要的特殊规则
AI判断	根据description自动决策	情境相关的规范

趋同：指令文件正在变成同一个东西

CLAUDE.md、AGENTS.md、.cursorrules、.windsurfrules、copilot-instructions.md、GEMINI.md、.clinerules、CONVENTIONS.md。

名字不同，**本质上是同一件事**：用Markdown文件告诉AI项目规则。

趋势已经不只是「发生」了，**已经开始标准化**。2026年3月，AGENTS.md被纳入Linux基金会旗下的Agentic AI Foundation管理，背后推动者包括Sourcegraph、OpenAI、Google、Cursor和Factory。这意味着AGENTS.md正在从一家公司的格式变成行业标准。

Windsurf从2026年初开始自动识别AGENTS.md，同一个仓库的规则可以同时被Windsurf和Codex CLI读取。GitHub Copilot的`.instructions.md`格式也越来越像AGENTS.md。Google的Gemini推出了自己的GEMINI.md。

统一标准不再是「可能」，而是「什么时候」的问题。但现在不必等。**你写的任何一种指令文件，80%的内容在不同工具间是通用的**：项目结构、编码规范、测试命令、架构约束。这些不绑定特定工具。

三工具指令系统对比

维度	Claude Code	Codex CLI	Cursor
文件格式	纯Markdown	纯Markdown	MDC（Markdown + YAML）
层级数	3（全局/项目/子目录）	4（override/agents/team_guide/.agents）	2（全局/项目）
Glob Scoping	否（靠子目录实现）	否	是（frontmatter中的globs）
团队共享	是（git提交）	是（git提交）	是（git提交）
导入语法	@path/to/file	无	无
跨工具兼容	否	Windsurf自动兼容	否

写好指令文件的三个原则

原则一：方向感而非僵化步骤

好的指令文件像地图，不像说明书。告诉AI项目的地形、重要路标、禁区在哪。别把每一步都写死。

不推荐

僵化步骤：

```
创建组件时：
1. 先在 src/components/ 创建文件夹
2. 创建 index.tsx
3. 创建 types.ts
4. 创建 styles.module.css
5. 在 index.tsx 中导入 styles
6. 导出 default
```

推荐

方向感：

```
## 组件规范
- 共享组件放 src/components/
- 样式用 Tailwind，不用 CSS Modules
- 单文件 <200行，超过就拆
- Props用 interface 定义
```

僵化步骤看着全面，但agent遇到不完全匹配的情况就会卡住。**方向感让agent保留判断空间，同时不跑偏。**

原则二：护栏优先于手册

与其告诉agent你应该这么做，不如告诉它你不能这么做。

OpenAI的架构不变量（architectural invariants）用了一个反直觉但有效的表达方式：**声明这里不存在什么。**

```
# 架构不变量
- 这个项目不使用 ORM，所有数据库操作用原始SQL
- UI层不直接访问数据库，必须通过 Service 层
- 不存在全局状态管理库，用 React Context + hooks
- 没有任何微服务通信，这是单体应用
```

告诉agent不存在什么，比告诉它存在什么更能约束解空间。**约束反而提升生产力。**

原则三：犯错 → 记录 → 迭代的飞轮

Boris Cherny的CLAUDE.md只有约100行，远少于多数开发者的500-1000行，效果反而更好。原因是他只记录agent真正犯过的错。

他管这叫复利工程（Compounding Engineering）：

> **Boris Cherny：** "Anytime we see Claude do something incorrectly, we add it to CLAUDE.md so it doesn't repeat next time."

他的团队在PR中用 `@.claude` 标签来更新CLAUDE.md。每次纠正agent的行为，就在CLAUDE.md里加一条。日积月累，这个文件变成了团队的制度知识。

对每一行都问自己一个问题：**删掉它会导致agent犯错吗？**如果不会，就删掉。这是保持指令文件精简的最佳判断标准。

应该写什么，不应该写什么

应该写	不应该写
agent猜不到的命令（如 `pnpm vitest run` ）	agent读代码就能发现的（如使用React）
与默认不同的代码风格规则	标准语言惯例
测试指令和首选测试运行器	详细的API文档（链接即可）
分支命名、PR惯例	频繁变化的信息
项目特定的架构决策	教程和长篇解释
常见陷阱和非显而易见的行为	「写干净代码」之类不言自明的原则

好的指令文件，读起来像资深同事在你入职第一天给你的备忘录：关键信息、常见坑、谁负责什么。**不是培训手册，是生存指南。**

§15 约束层：建议和约束是两回事

The Constraint Layer: Suggestions vs. Enforcement

在指令文件里写「请不要push到main」是建议。用hooks在程序层面拦住它，是约束。这个区别是harness从art到engineering的转折点。

一个根本性的区分

你在CLAUDE.md里写了一行：

```
- 永远不要直接push到 main 分支
```

这是一条建议。Claude大概率会遵守，但没有什么保证它一定遵守。上下文太长、任务太复杂、模型偶尔抽风，规则就可能被忽略。

现在换一种方式。你在Claude Code的hooks配置里写：

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash(git push*main*)",
        "handler": {
          "type": "shell",
          "command": "echo 'DENY: 禁止直接push到main分支'"
        }
      }
    ]
  }
}
```

这是一条约束。不管Claude怎么想、上下文多混乱，push到main的命令在执行前就会被拦截。程序级的，确定性的，不可绕过的。

Anthropic的官方文档说得很清楚：

> **Anthropic官方：**"Unlike CLAUDE.md instructions which are advisory, hooks are deterministic and guarantee the action happens."
> CLAUDE.md的指令是建议性的，hooks是确定性的，保证动作一定执行。

建议和约束，完全两回事。这个区别，是harness从art到engineering的转折点。

Claude Code的Hooks系统

Hooks是Claude Code的杀手锏。截至2026年4月（v2.1.90），它支持二十多种生命周期事件，七个最常用：

事件	触发时机	核心能力
PreToolUse	工具调用前	可以deny操作——安全策略的核心执行点；支持defer决策（headless会话暂停恢复）
PostToolUse	工具调用后	审计、日志、自动格式化
SessionStart	会话启动时	动态加载上下文、环境初始化
Stop	agent停止时	确定性完成检查
PermissionRequest	请求权限时	自动化审批、路由到Slack等
PermissionDenied	自动模式拒绝后	记录被拒绝的操作、触发替代方案
PostCompact	上下文压缩后	响应压缩事件

最关键的是**PreToolUse**。它可以返回deny信号，从程序层面阻止Claude执行某个操作。在所有AI编程工具中独一份。

四个实用的hooks示例

示例一：编辑文件后自动跑lint

每次Claude编辑了文件，自动运行ESLint检查。不合格的话Claude会看到错误输出，自动修复。

```
// .claude/settings.json
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "handler": {
          "type": "shell",
          "command": "npx eslint --fix \"$CLAUDE_TOOL_ARG_file_path\" 2>&1 || true"
        }
      }
    ]
  }
}
```

示例二：禁止修改关键配置文件

保护生产环境配置不被agent意外修改。

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit(.env*)|Edit(*.production.*)",
        "handler": {
          "type": "shell",
          "command": "echo 'DENY: 生产环境配置文件受保护，不允许直接修改'"
        }
      }
    ]
  }
}
```

示例三：生成代码后自动做类型检查

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit(*.ts)|Edit(*.tsx)",
        "handler": {
          "type": "shell",
          "command": "npx tsc --noEmit 2>&1 | head -20"
        }
      }
    ]
  }
}
```

TypeScript文件被修改后立即做类型检查。Claude能看到类型错误并自主修复，不需要你提醒。

示例四：提交前必须通过测试

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash(git commit*)",
        "handler": {
          "type": "shell",
          "command": "pnpm test --run 2>&1 || echo 'DENY: 测试未通过，禁止提交'"
        }
      }
    ]
  }
}
```

一个有意思的细节：你可以让Claude自己写hooks。对Claude说 Write a hook that runs eslint after every file edit，它就帮你配好了。**agent给自己套上约束。**

OpenAI的硬约束：机械化强制执行

Codex CLI没有hooks，但OpenAI团队用了另一套硬约束体系。他们的核心哲学：

> **OpenAI：**"If you can articulate what it is about the code you don't like, the next step is to write that down."
> 如果你能说清楚代码哪里不好，下一步就是把它写成规则。

具体做法分三层：

第一层：依赖层架构

固定的层级顺序：`Types → Config → Repo → Service → Runtime → UI`

每个业务领域内，代码只能沿这个方向依赖。UI可以依赖Service，但Service不能依赖UI。跨领域关注点（认证、遥测、Feature Flags）通过唯一显式接口进入：Providers。

OpenAI的工程师说了一句很有意思的话：

> **OpenAI：**"This is the kind of architecture you usually postpone until you have hundreds of engineers. With coding agents, it's an early prerequisite."
> 这种架构通常等到有几百个工程师时才做。但有了coding agent，它是一个早期前提。

有agent写代码之后，架构约束必须前置。agent不会自觉维护一致性，只有约束能做到。

第二层：自定义linter

OpenAI让Codex自己生成了一套自定义ESLint规则和structural test，专门检测违反层级边界的代码。linter在CI中运行，**违规的PR直接被阻断**，合不进去。

错误信息不只说这里违规了，还带修复指导和文档链接。agent看到错误信息后能自主修复，不需要人工介入。

第三层：CI强制

所有上述检查都在CI中执行。不是建议，不是提醒，是**物理上合不进主分支**。

Codex的Sandbox：物理隔离

Codex CLI有另一种硬约束思路：sandbox。

模式	文件访问	网络
workspace-write（默认）	仅工作区可写	关闭
danger-full-access	全盘可写	开放

默认模式下，agent只能读写工作目录内的文件，网络访问完全关闭。`.git/`和`.codex/`始终受保护，即使在full-access模式下。

最简单粗暴的约束：**agent做不了的事就是做不了。**不是通过规则告诉它不要做，是在环境层面让它没法做。

约束谱系：从软到硬

把所有约束手段排列出来，你会发现一个从软到硬的谱系：

约束类型	性质	执行者	可绕过	工具支持
对话中的口头提醒	即时的，一次性	用户	是	所有工具
CLAUDE.md / AGENTS.md规则	持久的，建议性	模型遵循	是	所有工具
Hooks（PreToolUse deny）	持久的，强制性	程序执行	否	Claude Code
自定义Linter	持久的，强制性	静态分析	否	所有（自建）
CI阻断	持久的，强制性	CI系统	否	所有（自建）
Sandbox隔离	持久的，物理性	操作系统	否	Codex CLI

越往右，约束越硬。越硬的约束越可靠，但也越不灵活。

好的harness是这些层次的组合。不是全用最硬的，也不是全用最软的。大多数规则用指令文件就够了，少数关键的安全线用hooks或CI守住。

建议 vs 约束的对比

不推荐

建议（指令文件）

写在CLAUDE.md里：「请不要删除数据库迁移文件」

大概率被遵守，但不保证。上下文太长、任务太复杂时可能被忽略。适合编码规范、风格偏好、架构指引。

推荐

约束（Hooks / CI / Sandbox）

PreToolUse hook拦截对 `migrations/` 目录的删除操作。

100%可靠，不受上下文影响。适合安全红线、生产环境保护、不可逆操作拦截。

怎么判断一条规则该是建议还是约束？问自己：**agent违反了这条规则，后果是什么？**

代码风格不一致？恼人但不致命，建议就行。删掉了生产数据库？灾难性后果，必须约束。推到了main分支？可以revert但很麻烦，约束。变量命名不对？review时改就行，建议。

约束保护的是底线，不是偏好。

约束是转折点

调研这些工具的过程中，我发现一个清晰的分界线：**只有Claude Code和Codex CLI提供了程序级的硬约束**。Cursor、Windsurf、GitHub Copilot、Aider都只有提示级的规则。

具体区别：

- Claude Code的hooks是**可编程的**——你可以写任意逻辑决定approve或deny
- Codex CLI的sandbox是**策略式的**——选择一个预设级别，不支持自定义逻辑
- 其他工具的规则是**建议性的**——AI「应该」遵守但「可以」违反

Cline走了第三条路：用Plan/Act双模式实现「社会性约束」。Plan模式只分析不修改，Act模式每步都要人工审批。不是技术上阻止agent，而是流程上确保人工确认。

三种约束思路各有适用场景。**但约束的存在本身，才是harness从一堆配置文件升级为工程系统的分水岭。**

Martin Fowler团队的分析也指向同一个结论：

> Birgitta Boeckeler（ThoughtWorks）：提高agent的信任度和可靠性，需要的恰恰是限制它的解空间。约束越多，反而越可信。

跟人类组织里的道理一样。没有规则的团队不是自由，是混乱。有清晰边界的团队，成员反而能在边界内放手干。

AI agent也是一样。你给它越清晰的约束，它在约束内的发挥就越好。**Harness不限制AI的力量，harness让AI的力量变得可信。**

§16 能力层与记忆层

Capability and Memory

指令和约束管的是agent「知道什么规则」，能力层管的是「能做什么事」，记忆层管的是「记住了什么」。这两层决定了agent的天花板。

能力：不是什么都装进去

一个agent能做多少事，取决于你给它接了多少工具。但工具不是越多越好。上下文窗口就那么大，每多一个工具描述，就少一点空间留给真正干活的内容。

Claude Code的Skills系统解决了这个矛盾。Skills放在 `.claude/skills/` 目录下，每个是一个.md文件，描述能力做什么、什么时候触发。**平时不占context，Claude根据当前任务自动判断要不要加载。**

写一个.md文件就能定义新能力，不需要写代码，不需要SDK。这在所有AI编程工具里是门槛最低的扩展方式。

我自己的skills目录里有几十个：一个管小红书配图，一个管飞书同步，一个管视频脚本审校，一个管信息搜索。每个skill做一件事，命名要让AI一眼看懂意图。

> **核心建议**
>
> Skills的设计原则：每个skill做且只做一件事，描述要写得像给同事的一句话交代。「帮我把文章发到飞书」比「执行飞书API文档创建流程」好。AI根据描述判断是否加载，描述模糊就触发不了。

MCP：一个协议连接一切

Skills是本地能力。如果agent需要连接外部世界呢？

Model Context Protocol（MCP）是目前最接近标准的答案。一个协议，让AI编程工具连接数据库、API、网页、GitHub、Jira、Slack。Claude Code、Cline、Windsurf已经原生支持。**MCP正在成为AI编程工具的USB接口。**

Block（原Square）开源的Goose编码Agent把这条路走得最远。通过MCP，Goose连接了3000+服务，从GitHub到Google Drive到Docker到Kubernetes。Stripe的Minions系统同样深度依赖MCP和工具扩展，每周自动生成并合并超过1300个PR。

Codex CLI的工具定义能力相对有限，聚焦于代码读写和命令执行。Cursor通过 `@docs` 引用外部文档作为上下文。GitHub Copilot有Extensions生态。Aider最纯粹，不做扩展，只专注代码编辑。

工具设计的原则

Anthropic在Building Effective Agents中提了一个被低估的概念：**Agent-Computer Interface（ACI）**。在工具文档和测试上花功夫，和在UI设计上花功夫一样重要。

什么意思呢：

推荐

工具命名让AI理解意图：`search_knowledge_base`
每个工具有清晰的参数描述和示例
工具做一件事，做完就返回结果
错误信息告诉agent哪里出了问题

不推荐

命名含糊：`process_data`（什么数据？处理成什么？）
参数描述缺失，AI靠猜
一个工具做五件事，返回一堆副作用
报错只说「失败」，不说为什么

LangChain的LLMToolSelectorMiddleware做了一个巧妙的优化：用一个快速LLM预筛选当前任务需要的工具，避免把所有工具描述都塞进上下文。工具多了之后，**选择本身也需要被工程化**。

记忆：下一个战场

2026年初，AI编程工具的记忆能力差距巨大。大部分还停留在静态文件阶段。

工具	自动记忆	手动记忆	跨会话持久化
Claude Code	auto-memory（自动保存观察）	MEMORY.md + CLAUDE.md	是
Windsurf	Cascade Memories（自动生成）	Rules	是
Cline	Memory Bank MCP（需配置）	.clinerules	是
Codex CLI	无	AGENTS.md	静态文件
Cursor	无	.cursor/rules/	静态文件
GitHub Copilot	无	instructions.md	静态文件
Aider	无	CONVENTIONS.md	静态文件

只有三个工具有动态记忆，其余全靠人工维护指令文件。**记忆系统分三个层次：AI自己决定什么值得记，通过工具调用显式管理，靠人手动更新文件。**差距一眼就看出来了。

Claude Code的Memory系统

Claude Code有三套记忆机制互补。

auto-memory：Claude自动将有用的观察保存到`~/.claude/projects/<hash>/memory/`。不需要你说记住这个，它自己判断什么值得留下。

MEMORY.md：用户手动维护的长期记忆。我的MEMORY.md记了工具偏好、项目背景、常犯的错、发布流程。每隔一段时间精简一轮，保持在100行以内。

CLAUDE.md本身：项目级记忆。Boris Cherny团队在PR中用`@.claude`标签来更新CLAUDE.md。每一条新增的规则，都是agent过去犯过的错的制度化记录。他管这叫复利工程。

Anthropic还发布了memory工具公开测试版，让agent能在上下文窗口之外维护知识。Claude玩Pokemon时维护精确的步骤计数，上下文重置后读取自己的笔记继续跑多小时的序列。长任务里这种能力至关重要。

知识库管理

记忆解决的是「agent记住什么」，知识库解决的是「agent在哪找专业知识」。

以我的写作项目为例，知识库按主题分类：

```
_knowledge_base/
├── INDEX.md
├── 技术工具/
├── 行业人物/
├── 产品发布/
├── 报告资料/
└── 方法论/
```

每次调研完一个主题，成果保存到对应分类，附来源URL和日期。agent写文章时自动检索相关文件。**知识库不是堆资料，是给agent准备一张随时能查的索引。**

OpenAI Codex团队更激进：创建了叫ExecPlan的自包含设计文档，把Google Docs里的规划文档迁入代码仓库，Slack里的决策转为markdown存入repo。仓库必须是唯一的真相来源。**agent看不到的东西等于不存在。**

上下文管理：不是越多越好

记忆和知识库都在往上下文里塞东西。但上下文窗口有物理极限。

Anthropic发现了一个叫上下文腐烂（context rot）的现象：**上下文token越多，模型准确回忆信息的能力越差**。约100万token处有明显的性能天花板，超过这个点性能显著下降，不管技术上支持多大的上下文窗口。

还有一个更诡异的发现。Sonnet 4.5是第一个意识到自身上下文窗口的模型。它会在感知接近极限时提前收尾工作，对剩余token的估计非常精确但错误。Anthropic管这叫上下文焦虑。

> **上下文焦虑严重到仅靠压缩不足以支撑长任务性能，上下文重置成为Sonnet 4.5时代的必需手段。**到了Opus 4.5，这个行为自行消除了。模型进步在消解harness的复杂性。

面对上下文限制，有三种应对策略：

策略	机制	适用场景
Compaction	总结早期对话，在缩短的历史上继续	连续性重要的长任务
Context Reset	完全清空窗口，用结构化交接状态启动新会话	上下文焦虑严重时
开新会话	任务切换时直接开新对话	不相关任务之间

Claude Code官方文档给了一个直觉判断：**如果修正Claude超过两次还是错，清空重来比继续修正好**。上下文被失败方案污染后，在上面继续修只会越修越歪。

能力层和记忆层的对比总览

最后放一张横向对比表，看看不同工具在这两层的覆盖情况。

维度	Claude Code	Codex CLI	Cursor	Windsurf	GitHub Copilot
能力扩展	Skills + MCP + Hooks	桌面App + GPT-5.3	Background Agents + BugBot	MCP	Agent Mode + Extensions
无代码扩展	Skills（纯.md文件）	无	无	无	无
动态记忆	auto-memory	无	无	Cascade Memories	无
跨会话	三套互补	静态文件	静态文件	工作区绑定	静态文件
上下文管理	/compact + /clear	任务中途可调整	Composer上下文	M-Query检索	Agent模式自管理

能力层的趋势是MCP标准化。记忆层还在各自为战。这两层未来一年会快速拉齐——没有记忆的agent实在太痛了。

§17 编排层：让十匹马同时跑

Orchestration: Running Ten Horses at Once

一个agent解决不了的问题，十个agent未必能解决。但如果编排对了，十个agent能做到一个agent永远做不到的事。这一节聊什么时候该用多Agent，以及怎么用。

从一匹马到一支马队

前面所有章节聊的都是单个agent的harness。但真实项目里，很多任务一个agent搞不定。

不是因为它不够聪明。上下文窗口是物理极限。一个agent同时处理前端、后端、数据库、测试、文档，上下文填满之后性能断崖式下降。Anthropic的数据说得很清楚：约100万token就是天花板。

所以你需要多个agent，**每个agent有自己的上下文窗口，处理自己擅长的那一块**。问题是：谁来协调它们?

Boris的10-15并发会话

最朴素的编排方式：人来当编排器。

Boris Cherny（Claude Code的创建者）日常保持10-15个并发Claude Code会话。5个在终端，编号1-5，用shell别名（za、zb、zc）快速切换。5到10个在浏览器。还有早上启动、后续查看的移动会话。

每个会话跑在独立的git worktree上，代码不冲突。部分工程师保留一个专用的「分析」worktree，只用于日志和查询，不写代码。

> **这不是什么高级架构，就是拿人当编排器。**但Boris说这是他团队内部 the single biggest productivity unlock。

操作方式：

1. **创建多个worktree**
   每个任务一个独立的工作目录，代码完全隔离。
2. **每个worktree启动一个Claude Code会话**
   终端编号方便切换，关键会话开OS通知。
3. **任务分配**
   每个会话处理一个独立模块或功能，互不干扰。

4 **人工协调**

你在会话之间切换，审查结果，解决冲突，合并代码。

好处是完全可控。坏处是你得一直盯着，任务之间的依赖关系只存在于你脑子里。

Anthropic的三Agent架构

不想拿人当编排器呢？Anthropic给了一个参考答案。

他们搭了一个三Agent系统，灵感来自生成对抗网络（GAN）：

Planner接收简短提示，扩展为完整的产品规格说明。重点放在产品语境和高层技术设计，不管细节实现。过于详细的技术指令会在下游产生级联错误。

Generator按Sprint一次做一个功能。每个Sprint开始前，Generator和Evaluator协商一份Sprint Contract（冲刺合约），对完成的定义达成一致。这个合约桥接了用户故事和可测试实现之间的鸿沟。

Evaluator是整个架构的灵魂。它用Playwright MCP与运行中的应用交互，像真人QA工程师一样测试UI功能、API端点、数据库状态。按预设标准打分，包括设计质量、原创性、工艺水准。

这套架构的核心洞察，值得再强调一遍：

> **工程化一个严厉的独立评估者，远比教一个生成者自我批判容易得多。**分离角色创造了对抗性动态，怀疑论的Evaluator提供批判性反馈，帮助Generator突破瓶颈。

Anthropic做了一个成本对比实验：

方案	时间	成本	结果
单Agent（Solo）	20分钟	$9	核心功能不可用
三Agent（Full Harness）	6小时	$200	完整可用的应用

贵了20倍以上，但单Agent的产出根本不能用。**不是花更多钱做得更好，是不花这个钱就做不成。**

有意思的是，模型从Sonnet 4.5升级到Opus 4.6之后，Sprint机制被完全移除了。模型原生就能处理长任务，Evaluator从每Sprint评估变为全程结束后一次性评估。**模型进步在简化编排。**

你也可以用Planner/Generator/Evaluator

三Agent架构看起来很重，但核心思路可以简化到日常使用。

1. **Planner：用Plan Mode做规划**
   复杂任务先进Claude Code的Plan Mode（Shift+Tab两次），让AI制定详细实现计划。Boris建议：一个Claude写计划，启动第二个Claude以「staff engineer」身份审查计划。

2. **Generator：切回Normal Mode执行**
   计划确认后切到auto-accept mode让AI写代码。事情出错时，切回Plan Mode重新规划，而不是硬推。

3. **Evaluator：开新会话审查**
   写完后开一个全新对话，把结果贴进去：「找出所有问题」。全新上下文的AI没有对自己代码的偏见，能发现很多第一个AI漏掉的问题。

这就是三Agent架构的穷人版。不需要搭复杂系统，三个会话窗口就够了。

Agent Teams：Claude Code的内置多Agent

Claude Code的Agent Teams是目前唯一支持agent间直接通信的方案。

一个session担任team lead，分配任务、综合结果。Teammates之间可以直接通信，不必通过lead中转。每个agent有独立的上下文窗口。

和Boris的手动模式相比，Agent Teams的优势是**自动协调**。你描述一个大任务，lead自己拆、自己分配、自己收集结果。适合并行调研、独立模块开发、竞争假设调试。

Cursor 2.0支持最多8个AI agent并行运行，数量最多，但每个agent独立工作，**没有通信机制**。八匹马各跑各的，没有缰绳把它们连在一起。

Codex CLI的Subagents继承父agent的sandbox策略和审批设置，运行在主agent的上下文内，只能向上汇报。适合探索性子任务，不适合长时间独立工作。

Writer/Reviewer并行模式

多Agent不一定要搞复杂架构。最实用的模式可能也是最简单的：一个写，一个审。

Anthropic官方推荐的做法：一个Claude写代码，另一个Claude用全新上下文审查。避免对自己写的代码有偏见。

这个模式可以扩展。批量处理时：

```
for file in $(cat files.txt); do
  claude -p "Migrate $file from React to Vue. Return OK or FAIL." \
    --allowedTools "Edit,Bash(git commit *)"
done
```

每个文件独立处理，失败不影响其他文件。**并行度取决于你愿意同时跑多少个进程。**

Stripe的Pipeline编排

企业级编排的标杆案例是Stripe的Minions系统。

工程师在Slack中发一条消息，走开，回来时PR已经ready。每周超过1300个PR，全部零人类手写代码。

关键基础设施：每个Minion运行在标准化的AWS EC2实例（devbox）上，预装Stripe完整代码树、预热的Bazel和类型检查缓存。**从warm pool启动一个devbox不到10秒。**

Stripe的核心洞察：Minions能work的首要原因跟AI模型本身几乎无关，而是Stripe在LLM出现之前就为人类工程师建设了多年的基础设施。全面的测试覆盖、合成端到端测试、蓝绿部署支持快速回滚。**harness是建在基础设施之上的，基础设施不好，harness也搭不起来。**

每个AI生成的PR仍然需要人类review。但review过程严重依赖自动化信心信号。人还在loop里，只是位置变了。

什么时候该用多Agent

多Agent编排很诱人，但大部分场景单Agent就够了。Anthropic在Building Effective Agents中给了一个判断标准：

> **能用单次LLM调用解决的，不要用Agent。Agent适合无法预测步骤数、无法硬编码固定路径的开放式问题。**

同样的逻辑往上推一层：能用单Agent解决的，不要用多Agent。

场景	推荐模式	理由
改个bug、加个功能	单Agent	上下文足够，没有并行需求
重构一个模块	单Agent + Plan Mode	需要全局视角，拆开反而丢上下文
同时改5个独立模块	手动多会话（Boris模式）	任务独立，不需要agent间通信
写代码 + 审代码	Writer/Reviewer双会话	消除自我审查偏差
复杂全栈应用从零开始	Planner/Generator/Evaluator	任务跨越前后端，需要规划和验证
大规模迁移/批量处理	Pipeline脚本编排	任务高度重复，可并行
1000+PR/周的企业规模	Stripe Minions式平台	需要专用基础设施

编排的复杂度应该匹配任务的复杂度。三个会话窗口能搞定的事，不需要搭Agent Teams。**从最简单的开始，真的需要了再升级。**

§18 经验工程：谁来设计下一代的缰绳

Experience Engineering: Who Will Design the Next Harness

这一章不给答案。

Martin Fowler团队的Kief Morris画了一张图，把人在AI编程中的位置分成三层。

In the loop：你逐行审查agent的输出，手动修改不满意的代码。人在环内，什么都过手。

On the loop：你不看代码本身，而是构建和改进harness。规格说明、质量检查、工作流引导。你管缰绳，不管马腿。

Out of the loop：你只说想要什么，agent自己搞定。Vibe coding。

Morris说了一句话，值得停下来想想：

> **in the loop和on the loop的区别，在你对结果不满意的时候最明显。**in the loop的人会去改代码。on the loop的人会去改harness，让它下次产出更好的结果。

On the loop听起来是更好的工作方式。你不再重复劳动，而是改进系统本身。

但这里有个麻烦。

如果太早把人移到on the loop，甚至out of the loop，将来谁来设计harness？

Morris的原话是这样的：新人从第一天起就不接触代码的细节，只在高层操作agent，那当harness出了问题需要有人理解底层发生了什么的时候，谁来？

Martin Fowler在播客里把这个问题说得更尖锐：

Junior developer最重要的属性不是他们现在能产出什么，而是他们能成长为senior developer。

如果AI替代了junior的产出，但同时也剥夺了他们的成长路径，这不是效率提升，是在透支未来。

这个担忧不是空穴来风。Shopify把实习生项目从25人扩大到1000人，理由是实习生用AI的方式更有趣。Block以AI提升效率为由裁员40%。两个方向，同一个问题：中间层在消失。

Mitchell Hashimoto能给Ghostty写好harness，因为他理解终端模拟器的每个细节。那个配置文件里每一行都对应着agent过去犯过的一次错，而他知道那些错为什么是错。

OpenAI那3个工程师能驾驭100万行代码产出，因为他们知道什么架构是好的、什么会在三个月后爆炸。

Kent Beck能写出那个TDD驱动的CLAUDE.md，因为他发明了极限编程，花了几十年理解什么样的代码是好代码。前两次尝试失败了，复杂度积累太多，AI完全卡住。第三次才找到正确的介入方式。

设计好harness的人，都有深厚的领域经验。问题是这些经验从哪来。

我自己是个有意思的样本。

我从来没手写过代码。所有产品都是AI写的。小猫补光灯上了AppStore付费榜Top 1，累计用户超百万。我的harness从零开始，在和AI互动中一点一点长出来，没有任何编程经验可以迁移。

但我得对自己诚实。

我能设计harness，不是因为我天生懂系统设计。是因为我在和AI协作的上千小时里，观察到了它的行为模式。它什么时候偷懒，什么时候幻觉，什么时候需要硬约束而不是温柔提醒。

这些判断力不来自写代码的经验，但来自另一种经验：和AI反复较劲的经验。

问题是，这种经验能教吗？

我说不清楚。

我知道CLAUDE.md该怎么写，但让我教别人为什么这么写，我会卡住。很多决定是直觉做的。直觉来自踩坑。踩坑来自大量重复。大量重复来自时间。

这和老程序员说「你写几万行代码自然就懂了」其实是一回事。

所以问题可能不是「写代码的经验」能不能被替代。

而是：不管你积累的是什么经验，足够多的经验本身就是设计harness的前提。没有捷径。换了个赛道而已。

以前的赛道是写代码、调bug、重构、被线上事故搞到半夜。

现在的赛道是写CLAUDE.md、配hooks、被AI的幻觉搞到半夜。

痛苦的形状变了。痛苦的总量可能没变。

也许Martin Fowler担心的不是「没人写代码了」。

而是「没人愿意花够多时间踩够多坑了」。

Boris Cherny说过：工程师的核心贡献不是代码，是判断力。判断构建什么、如何验证、何时信任输出、何时反驳。

判断力从哪来？

从做了很多次之后知道哪些路走不通来。

84%的开发者已经在使用或计划使用AI工具。Anthropic的报告说工程师在60%的工作中用AI，但只能完全委派0-20%的任务。中间那40-60%是人和AI协作的地带。判断力生长的土壤。

Dario Amodei在2026年3月说了句话：**6个月内，90%的代码将由AI编写。**

如果他说对了，那片土壤会怎样？

Karpathy已经在身体力行了。2025年12月起他不再手写代码。他的AutoResearch项目——630行代码加一个markdown prompt——2天跑了700次实验。一个人加上一套harness，做了一个团队几周的工作量。

但Karpathy能这么做，因为他是Karpathy。他知道什么实验值得跑，什么结果值得追，什么发现是噪音。**那个markdown prompt之所以有效，是因为写它的人有几十年的研究直觉。**

如果这片土壤被全自动抽干了呢？

我不知道答案。

我只知道我自己的harness是在上千小时的较劲中长出来的。如果有人问我怎么快速学会设计harness，我想说的第一句话是：你不能快速。

顺便说一件事。GitHub Copilot从2026年4月24日起，免费和Pro用户的交互数据默认用于模型训练。你和AI协作的方式、你的harness思路、你的指令文件风格，都会变成下一代模型的训练数据。

你在教AI怎么做，AI在学你怎么教。

这个循环里，谁在设计缰绳，可能比谁在骑马更重要。

这个我也不确定。留给你想。

Harness Engineering

AI编程：从入门到精通

花叔 • AI进化论-花生

AI编程时代的工程方法论：从起源到框架，从7个真实案例到动手实操

用缰绳驾驭AI，让每一匹马都跑在正确的方向上

加入知识星球 →

B站：AI进化论-花生 · 公众号：花叔 · X/Twitter · YouTube · 小红书 · 官网

Created by 花叔 • v1.0 • 2026年