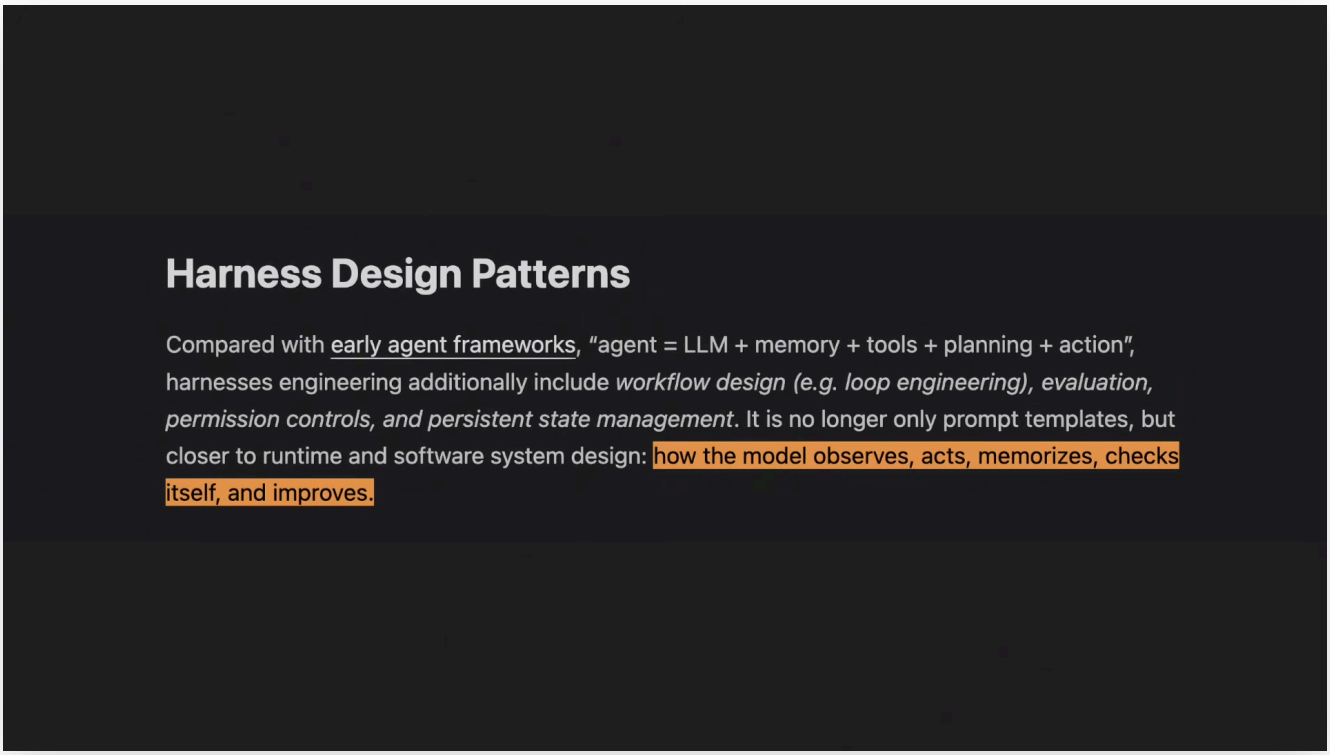
AI自我递归改进先要靠Harness工程 | Lilian Weng最新长文 | 反馈循环 | 三个设计模式 | 核心智能 | ACE | MCE | Meta-Harness | STOP - 图文解说

> 一张关键画面对应一段完整解说，按原视频时间推进。

01. Harness 的定义与递归自我改进（00:00–02:48）



场景 1

完整内容

大家好，这里是最佳拍档，我是大飞。递归自我改进这个话题在 AI 圈讨论了半个多世纪了，但最近一年呢，突然有了清晰的落地路径。

很多人呢对于 AI 自我进化的想象都是模型直接改写自己的权重，越改越强，最终触发智能爆炸。但是如果你关注前沿的研究进展，就能够发现。

短期内真正能够跑通的自我改进路径，可能和你想的完全不一样。它不靠改模型参数，而是靠优化包裹在模型外面的那套系统，也就是 Harness 工程。

刚刚呢，OpenAI 的前应用研究主管翁荔发表了最新的博客文章，系统梳理了这个领域的最新研究进展。今天呢，我们就顺着他的思路，把这件事情给讲清楚。

递归自我改进呢，这个概念呢，最早可以追溯到1965年，I.J. Good提出了超智能机器的定义，说这样的系统能够在所有智力活动上超过人类。

还能够设计出更好的机器来改进自己。到了2008年，埃利泽尤德考斯基正是用递归自我改进来描述这种反馈循环，指的是 AI 用当前的智能改进产生智能的认知机制本身。

放到今天的 AI 语境里，这个反馈循环呢，有两种可能的形式。一种呢是模型直接改写自己的权重，另一种更宽泛的形式是模型改进自己的训练流水线和部署系统。

反过来催生出性能更强的下一代模型。这里呢特意提到了部署系统，因为在翁丽看来，原始模型和真实场景之间的这层部署系统，重要性可能不亚于模型本身的原生智能。而 Harness 也就是模型的支撑系统，就是 AI 部署里的核心组件。

简单来说，Harness 就是包括在基础模型外面的整套系统。它负责编排执行流程，决定模型怎么思考规划，怎么调用工具行动，怎么感知和管理上下文。

怎么存储产物？怎么评估结果？我们熟悉的 Claude、Codex 这类成功的编码 Agent 的产品，核心竞争力很大程度上就来自于成熟的 Harness 设计。

和早期大家熟悉的 Agent 等于大模型加记忆加工具加规划加行动的框架相比，Harness 工程新增了工作流设计、评估、权限控制。

持久状态管理这些内容。它已经不只是写提示词模板了，更偏向运行时和软件系统设计。核心是解决模型怎么观察、行动、记忆、自检和迭代的问题。

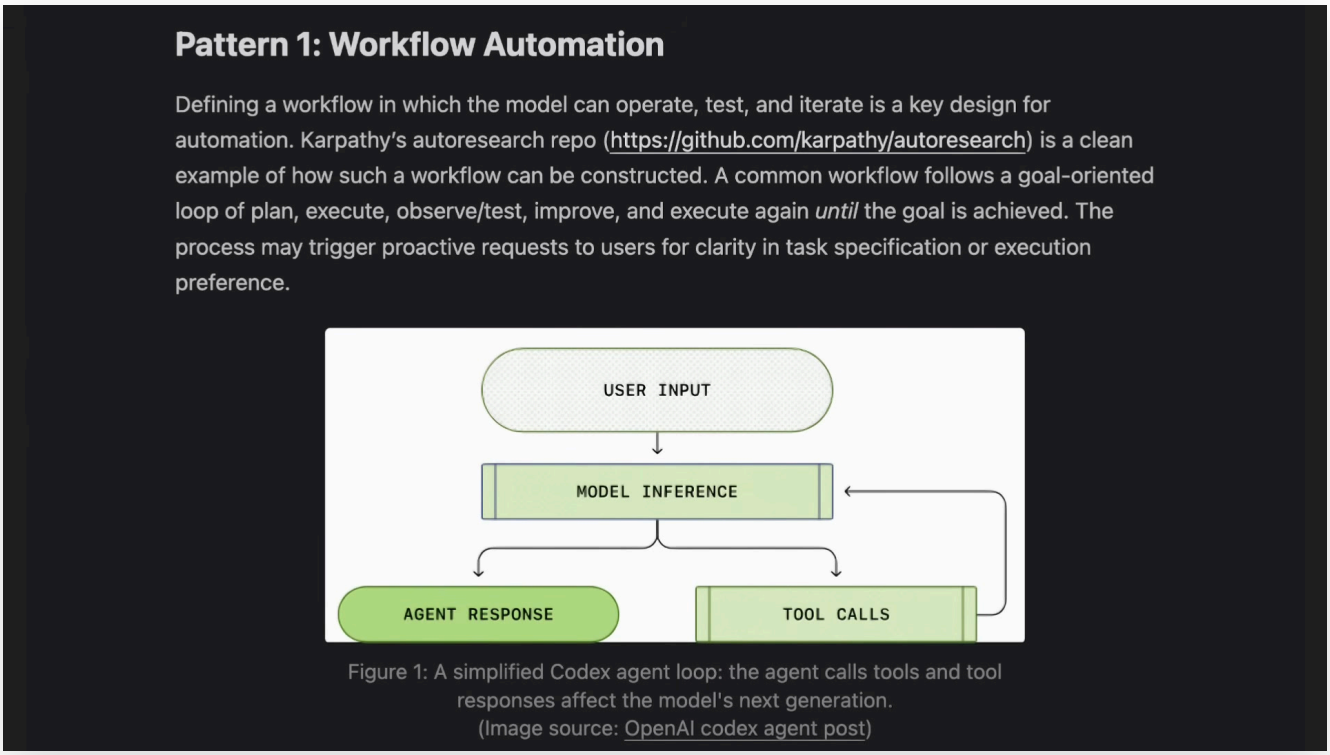
设计上的原则是尽量简单通用，才能够保证泛化性。很多思路呢，都参考了现有的软件工程实践，也能够复用模型预训练里学到的相关知识。

这个定位呢，其实和操作系统很像。把复杂的逻辑封装在内部，对外保持简洁的接口。未来行业里的配置和工具接口这些协议，大概率也会逐步的标准化。

画面说明

画面显示标题为 Harness Design Patterns 的深色幻灯片，正文把传统的 Agent 框架与 Harness 工程进行对比，并突出工作流、评估、权限控制、持久状态、观察、行动、记忆、自检和迭代等关键词。

02. 工作流自动化（02:48–03:33）



场景 2

完整内容

具体来看呢，Harness 有三个反复出现的核心设计模式。第一个是工作流自动化，给模型定一套可以操作、测试、迭代的工作流，是自动化的核心设计。

安德烈卡帕西的 Auto Research 就是一个很干净的例子。通用的工作流都是目标导向的循环，先做计划，再执行，然后观察结果或者是测试效果。

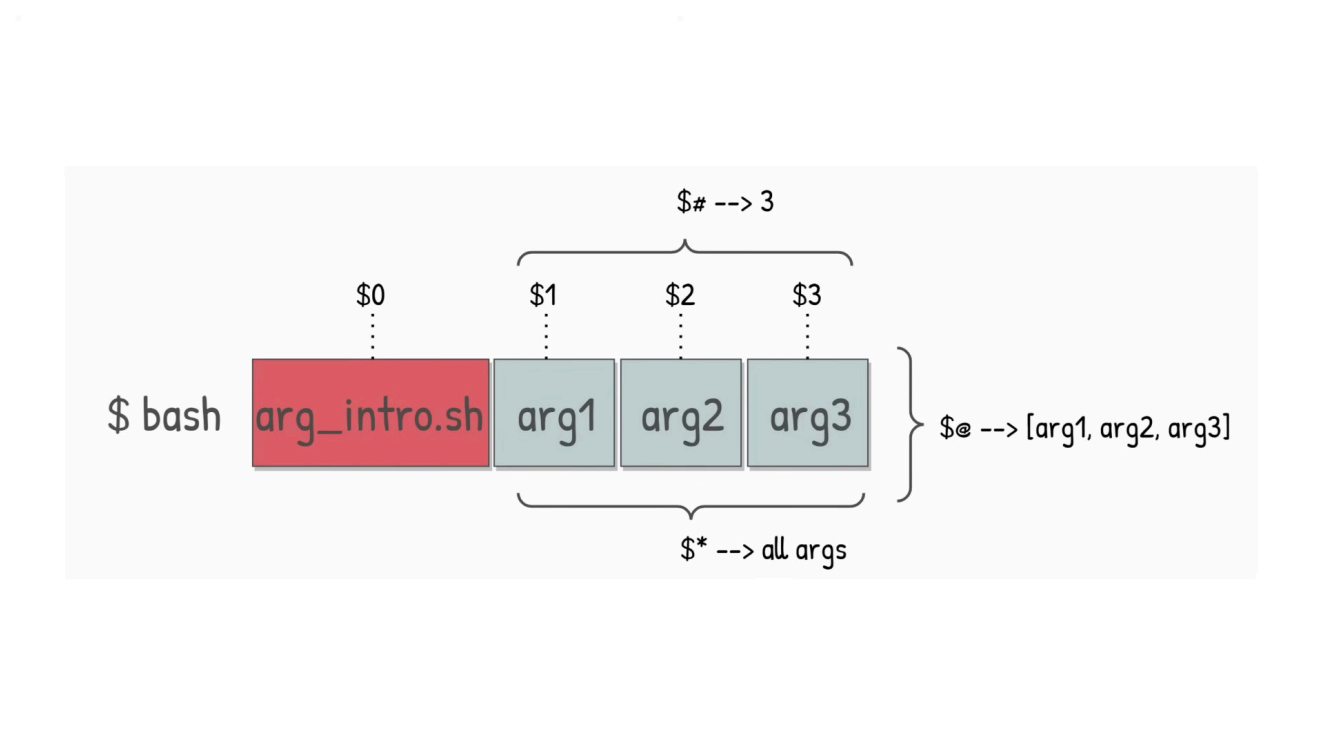
接着呢根据反馈改进，再执行下一轮，直到达成目标。过程中呢，如果遇到任务描述不清晰，或者是执行偏好不明确的情况，也可以主动的向用户确认。

和静态的提示词模板不一样，这种工作流呢更强调模型在运行时分析自己的执行轨迹和失败案例，一步一步地迭代推进。第二个模式是把文件系统作为持久级。

画面说明

画面显示 Pattern 1: Workflow Automation，并给出 User Input、Model Inference、Agent Response、Tool Calls 组成的循环示意图。

03. 文件系统作为持久记忆（03:34–04:07）



场景 3

完整内容

长周期的 Agent 系统里，一个很常见的需求是管理丰富的状态和产物。Harness 不可能把整个工作流和所有的日志都塞在上下文里，更合理的方式是把持久状态存在文件中。

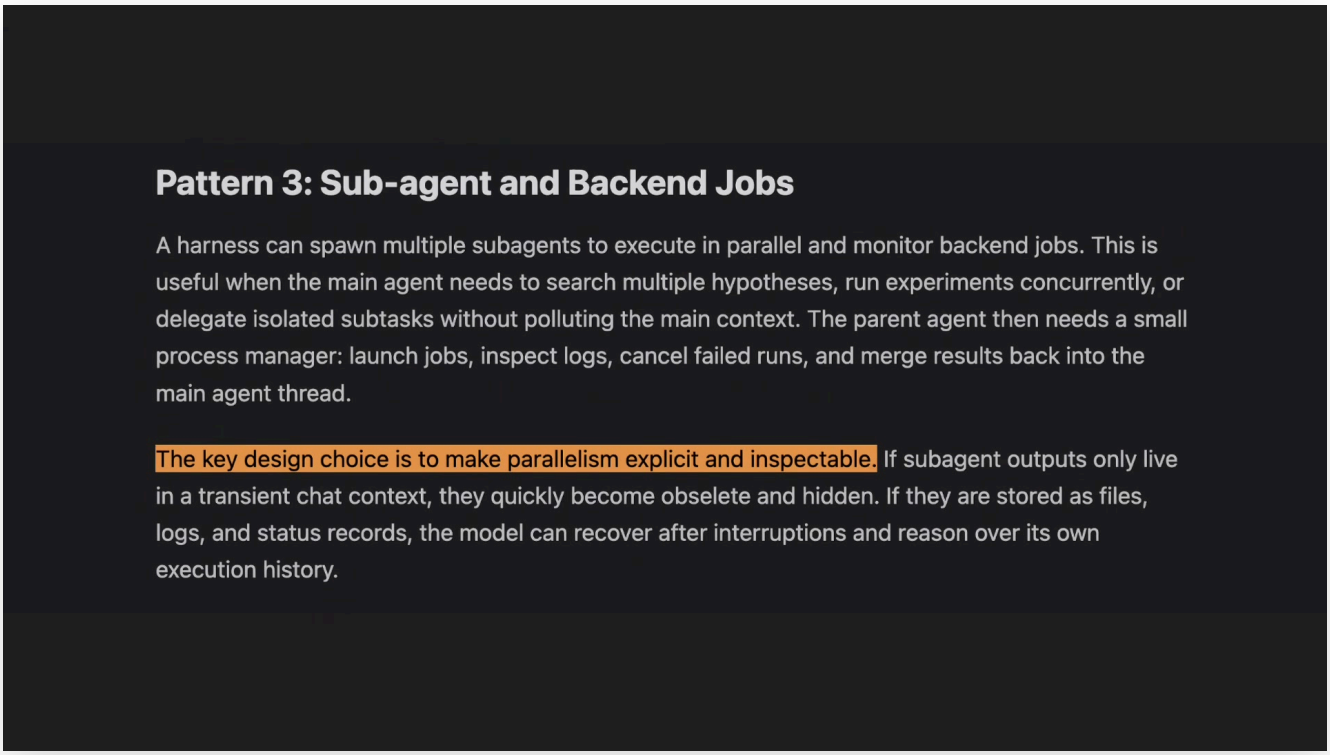
长周期的 Agent 运行中，实验日志、代码差异、论文摘要、错误栈、历史执行轨迹这些产物长度很快就会超过模型训练时的上下文窗口。

而读写编辑文件系统本来就是大模型的基础能力，通常呢靠 bash 命令实现。所以用文件这种简单形式来管理持久记忆。

画面说明

画面显示一个 bash 命令参数展开示意图，用来说明文件和命令行工具如何成为长周期 Agent 的持久化操作接口。

04. 子 Agent 与后台任务（04:07–05:04）



场景 4

完整内容

天然能够随着核心模型能力的提升而变强。第三个模式是子 Agent 和后台任务。一个 Harness 可以生成多个子 Agent 并行执行。

同时监控后台任务的状态。当主 Agent 需要同时验证多个假设和并行跑实验，或者是把孤立的子任务委派出去，不污染主上下文的时候。

这个模式就非常的有用。这时候呢，主 Agent 呢就需要一个轻量的进程管理器，负责启动任务、查看日志、取消失败的运行，最后呢把结果合并回主线程。这里的关键设计是

并行过程必须是明确可观测的。如果子 Agent 的输出只存在临时的聊天上下文里，很快就会失效，也没法追溯。但是如果都存成文件、日志、状态记录。

模型就算中途被打断，也能够恢复。还能够基于自己的完整执行历史做推理。说到这里呢，我们可以用最成熟的编码 Agent 的 Harness 来做个具体的例子。

画面说明

画面显示 Pattern 3: Sub-agent and Backend Jobs，橙色标注强调并行任务必须可观测、可检查，并保留日志和状态以便恢复。

05. 编码 Agent 的 Harness 工具链（05:04–06:13）

Group	Tool definitions
File system	- File discovery: `glob` , `grep` , `ls` - File read: `read` , `read_many` - File modification: `write` (a whole new file); `edit` (string exact-match replacement); `multi_edit` ; `apply_patch` (applies a structured patch/diff)
Shell execution	Run commands: `bash` , `PowerShell`
IO	`lsp` , git tools like `git_status` , `git_diff` , `git_commit`
External context	MCP tools, Skills
Web search	`web_search` , `web_fetch` , browser tools
Artifacts	Read docs, images; generate HTML, images
Backend processes	Such as: `CronCreate` , `CronDelete` , `CronList`
Agent delegation	Such as: `spawn_agent` , `resume_agent` , `wait_agent` , `list_agents` , `close_agent` , `interrupt_agent` , etc.

场景 5

完整内容

现在主流的编码 Agent 不管是 Claude Code Codex 还是 Cursor 这类产品，核心接口其实已经趋同了。它们的通用循环都是先观察整个代码仓库，然后做规划，接着搜索和读取相关的文件。

在编辑或者打补丁，写测试，然后运行检查错误，循环往复，直到任务完成。有了这套工具集，编码 Agent 呢，就能够在给定仓库里开发和调试问题。

就像人类开发者用 IDE 工作一样。这些 Agent 的工具集其实大同小异，最基础的是文件操作系统，比如说查找文件的 glob grep。

ls 读取文件的单文件读取、批量读取、修改文件的全量写入、精确字符串替换的编辑、批量编辑，还有应用结构化补丁。

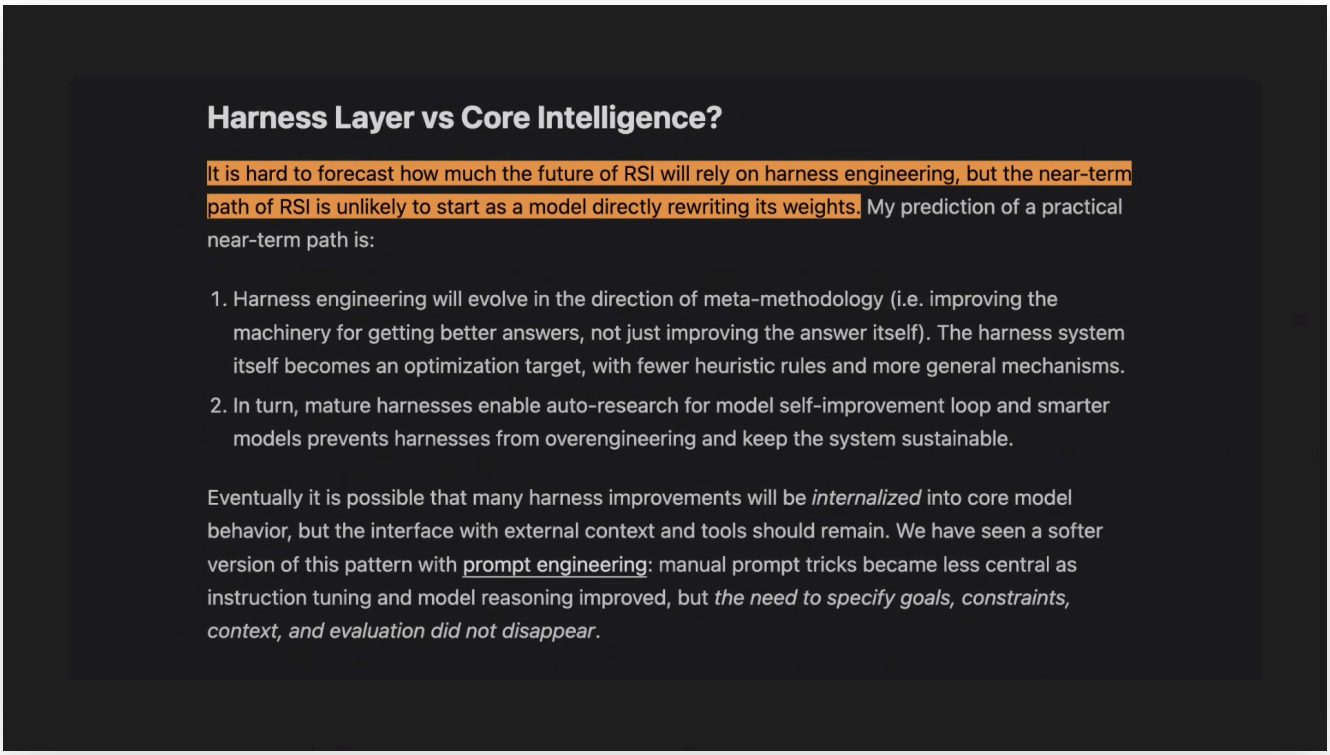
然后是 shell 执行，跑 bash 或者是 PowerShell 命令。再往上呢，是开发相关的工具，比如说语言服务器协议，Git 的状态查看、差异对比、提交操作。

还有外部上下文类的 MCP 工具、技能包、网页搜索和页面抓取、生成文档、图片、 HTML 这类产物的能力、管理后台定时任务的接口，以及生成、恢复、等待、中断子 Agent 的委派工具。

画面说明

画面显示编码 Agent 的工具清单，包括文件系统读写、Shell、Git、MCP、网页搜索、产物生成以及子 Agent 管理等类别。

06. Harness 层与核心智能（06:14–07:26）



场景 6

完整内容

聊完设计和案例，很多人自然会问一个问题，未来的递归自我改进到底更多是靠 Harness 层，还是靠核心智能呢？翁丽的判断是，很难预判长期的比例。

但近期的自我改进路径大概率不会从模型直接改写权重开始。他对近期落地路径的预测是这样的，Harness 工程会朝着原方法论的方向演化，也就是改进获取答案的机制。

而不只是改进答案本身。Harness 系统自己会成为优化目标，人工设计的启发式规则会越来越少，通用机制会越来越多。反过来，成熟的 Harness 能够支撑起自动研究的闭环。

反过来推动模型自我改进，而更强的模型也能够避免 Harness 走向过度工程化，让系统保持可持续性。长期来看，很多 Harness 的改进可能会被内化到核心模型的行为里。

但外部的上下文和工具的接口层应该会一直存在。这个模式其实有先例，早年的提示工程就是这样的。随着指令微调和模型推理能力的提升，手动的提示技巧就没那么核心了。

但定义目标、约束、上下文、评估的需求从来没有消失。接下来呢，我们来具体聊聊 Harness 的优化。整个 Harness 系统里，被优化的对象是逐步升级的。

画面说明

画面显示 Harness Layer vs Core Intelligence?，列出 Harness 从启发式规则走向可优化机制、并可能与模型能力共同演化的几个判断。

07. ACE：把上下文变成可演化手册（07:27–08:52）

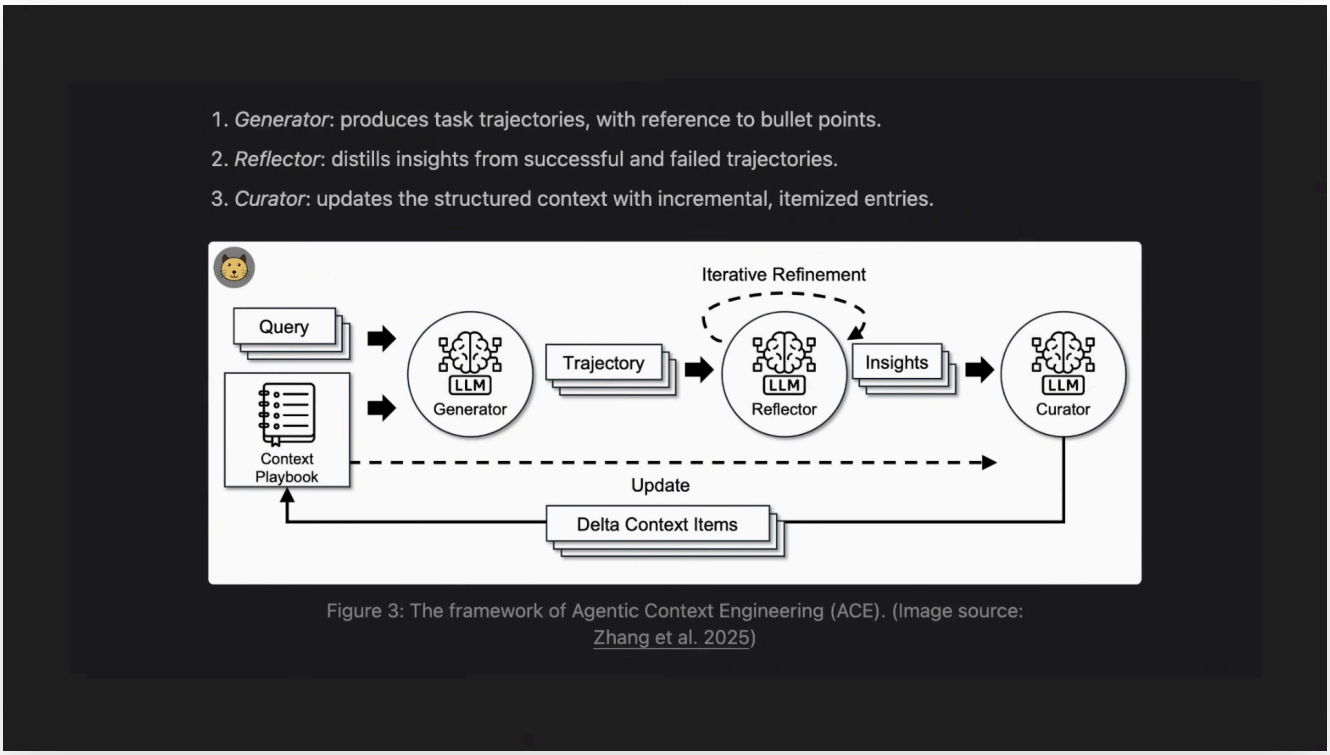


Figure 3: The framework of Agentic Context Engineering (ACE). (Image source: Zhang et al. 2025)

场景 7

完整内容

从最开始的指令提示词，到结构化上下文，再到工作流，再到Harness代码，最后甚至到优化器本身的代码。模型越智能越强，我们就能用更通用的方法来优化越复杂的对象。首先呢，是上下文工程。

如果只是把所有工具返回和模型生成的内容都堆进上下文，随着 Agent 的任务的周期变长，体量很快就会失控。上下文管理层的作用就是给大模型构建更结构化、更精简的上下文。

同时管理持久状态。虽然长上下文的研究一直在进步，但现阶段长上下文的智能表现和上下文工程往往是交织在一起的。第一个代表性工作叫做 Agentic Context Engineering 简称 ACE。

它不把上下文当成越堆越长的提示词，而是当成一本不断演化的操作手册。整个系统有三个组件共同维护一份带标识符和描述的要点式的上下文手册。

Generator 负责参考现有要点生成任务轨迹。Reflector 负责从成功和失败的轨迹里提炼洞见。Critic 负责用增量的条目更新结构化上下文。

为了避免迭代重写过程中的上下文崩塌和简洁性偏差，ACE 的一个关键设计是，Critic 不会重写一整段提示词，而是输出一条一条结构化的、带标识的要点。

通过确定的逻辑合并到上下文日志里，定期做精简和去重。ACE 呢能够从运行轨迹里学习经验，已经有了自我管理记忆的雏形。

画面说明

画面显示 ACE 的流程图：查询和结构化上下文进入 Generator，形成任务轨迹，经 Reflector 提炼洞见，再由 Critic 更新上下文条目。

08. MCE：上下文函数的双层优化（08:52–10:14）

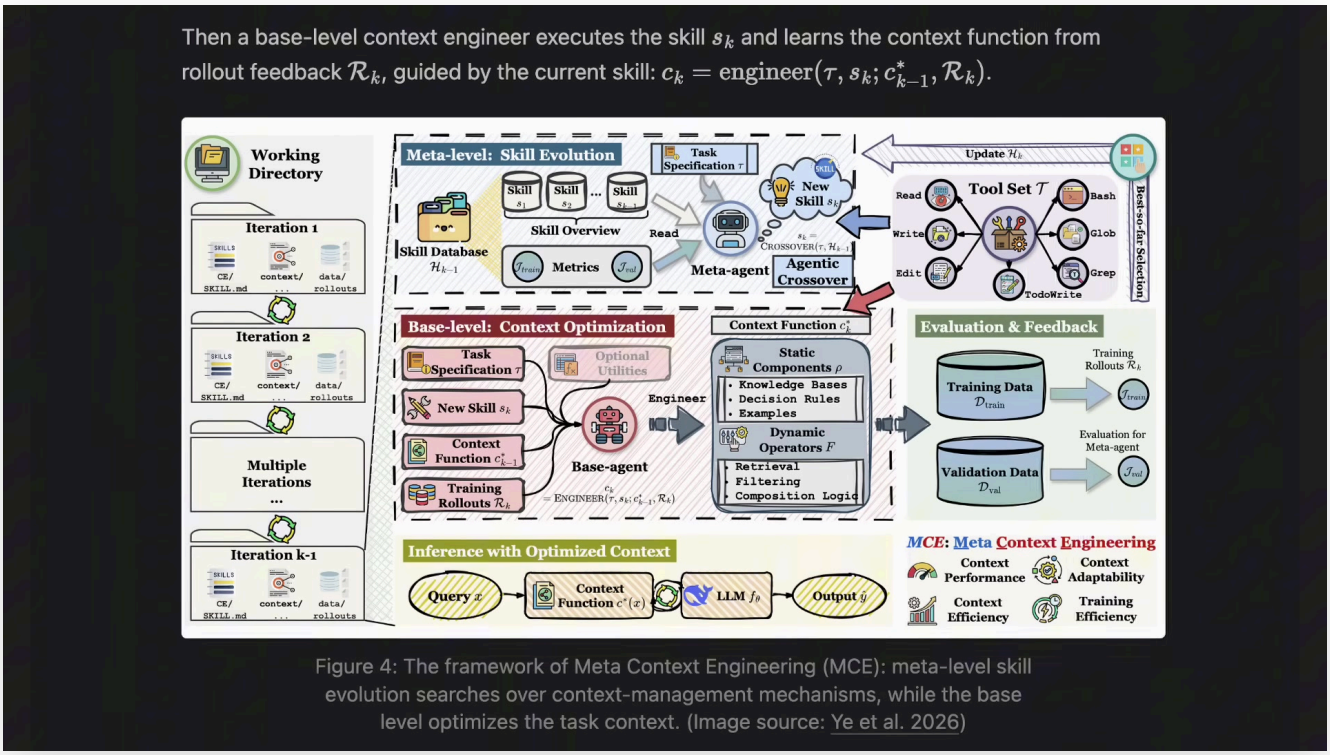


Figure 4: The framework of Meta Context Engineering (MCE): meta-level skill evolution searches over context-management mechanisms, while the base level optimizes the task context. (Image source: Ye et al. 2026)

场景 8

完整内容

但是它的更新规则和整体工作流还是人工设计的。要再往自我改进的方向走一步，就得提到 Meta-Context Engineering 简称 MCE 它把上下文管理的机制和上下文里的内容分离开，源层级呢做技能演化，基础层级呢做上下文优化。

在 MCE 的定义里，一个技能对应一个上下文函数，包含静态组件和动态算子两部分。静态组件呢，就是提示词、知识库、代码库这类固定的内容。

动态算子呢，就是搜索、选择、筛选、格式化这类操作。整个优化是双层的，内层在给定技能下，在训练数据上找最优的上下文。

外层在验证级上找性能最好的技能。系统里有一个技能数据库，记录了所有的历史技能、上下文函数和评估指标。源级 Agent 会基于历史技能做交叉，生成新的技能。

基础级的上下文工程则执行这个新技能，根据运行反馈来优化具体的上下文。实现层面，一个上下文函数就是一个专门目录里的一堆文件。

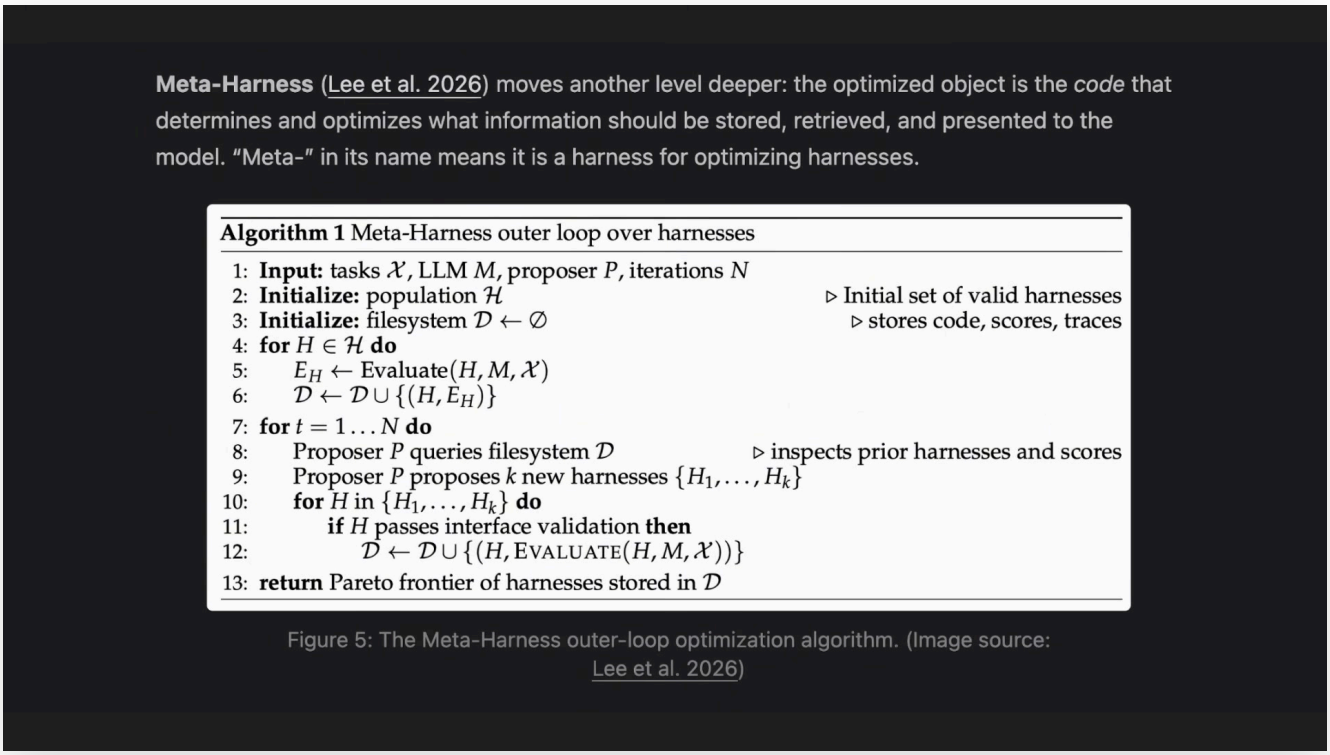
既有静态的技能说明文件，也有动态的上下文和运行数据。源级和基础级的优化都在标准的编码 Agent 的环境里运行，用的就是读写、编辑、 bash。

文件查找这些最基础的工具。再往深一层，还有 MetaHarness，它的优化对象又升了一级，是决定信息该怎么存储、检索和呈现给模型的整套代码。

画面说明

画面显示 Meta Context Engineering 的框架图，包含技能、上下文函数、静态组件、动态算子以及内层上下文优化和外层技能演化。

09. Meta-Harness：优化 Harness 的 Harness（10:15–11:26）



Meta-Harness (Lee et al. 2026) moves another level deeper: the optimized object is the *code* that determines and optimizes what information should be stored, retrieved, and presented to the model. "Meta-" in its name means it is a harness for optimizing harnesses.

Algorithm 1 Meta-Harness outer loop over harnesses

Input: tasks $\mathcal{X}$, LLM M, proposer P, iterations N
Initialize: population $\mathcal{H}$ ▷ Initial set of valid harnesses
Initialize: filesystem $\mathcal{D} \leftarrow \emptyset$ ▷ stores code, scores, traces
for $H \in \mathcal{H}$ **do**
  $E_H \leftarrow \text{Evaluate}(H, M, \mathcal{X})$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(H, E_H)\}$
for $t = 1 \ldots N$ **do**
  Proposer P queries filesystem $\mathcal{D}$ ▷ inspects prior harnesses and scores
  Proposer P proposes k new harnesses $\{H_1, \ldots, H_k\}$
  for H in $\{H_1, \ldots, H_k\}$ **do**
    if H passes interface validation **then**
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(H, \text{EVALUATE}(H, M, \mathcal{X}))\}$
return Pareto frontier of harnesses stored in $\mathcal{D}$

Figure 5: The Meta-Harness outer-loop optimization algorithm. (Image source: Lee et al. 2026)

场景 9

完整内容

名字里的 Meta 意思就是它是用来优化 Harness 的 Harness 它的外层循环逻辑很清晰，先初始化一批 Harness 候选，在文件系统里存下每个 Harness 的代码、分数和运行轨迹。

每一轮迭代，提案者读取文件系统里的历史 Harness 和对应的分数，生成新的候选。候选通过接口验证之后，就拿去评估，合格的加入池子。迭代结束后，输出所有帕累托最优的 Harness。

这里的提案者本身就是一个编码 Agent 所有执行历史都存在文件系统里。Agent 用 grep cat 这类命令来读取历史，而不是把所有的内容都塞进提示词上下文里。

生成的每个候选 Harness 都是文件系统里的一个目录，包含自己的源代码、评分、运行轨迹和状态更新。实验结果呢，也验证了这个思路的有效性。

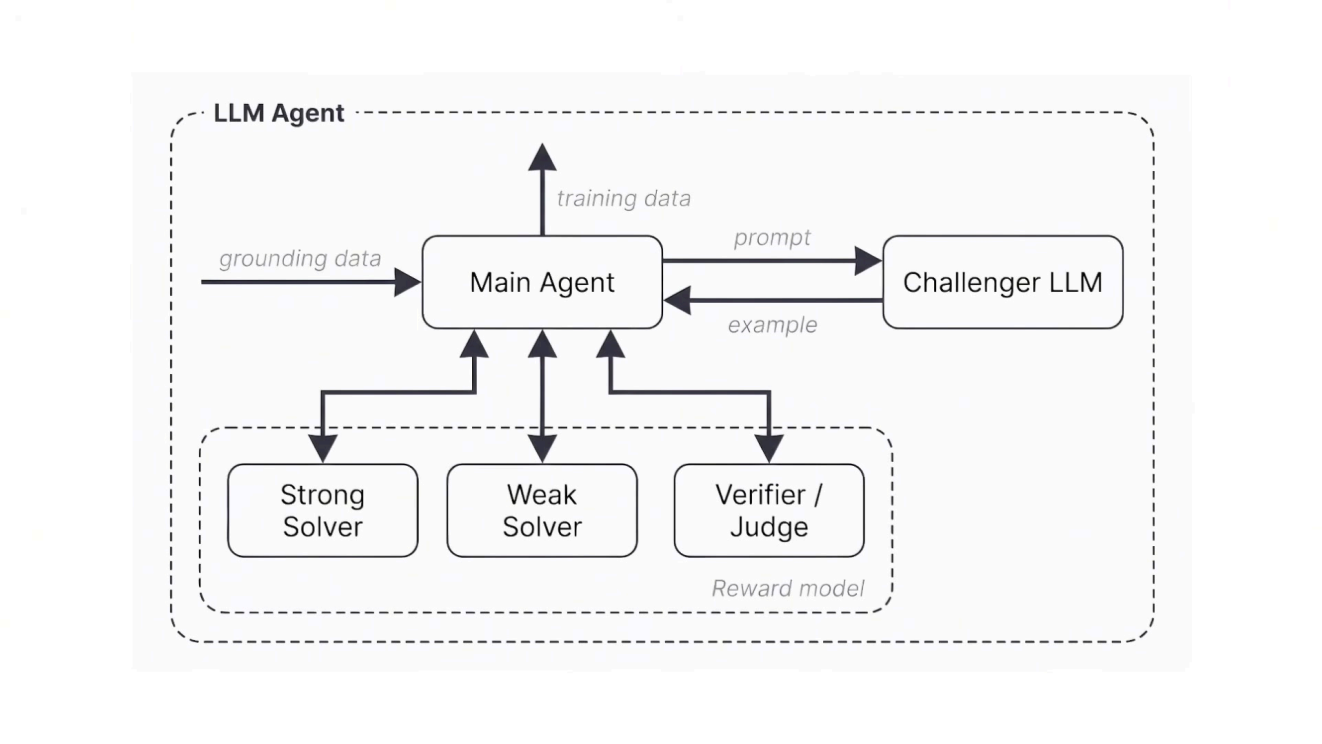
在 Terminal Bench 二基准测试上，Meta Harness 优化出来的方案表现超过了很多人工编写的强极限。这件事情背后的启示很重要，一旦 Harness 设计变成了可执行的搜索空间。

强编码 Agent 呢就能够探索和人类工程师完全一样的设计空间。讲完了上下文工程，我们再来看工作流的设计。早期的 Harness 工作流都是领域专家手工设计的。

画面说明

画面显示 Meta-Harness 的算法伪代码，包含初始化 Harness 池、提案、评估、验证、加入候选池和返回 Pareto 前沿等步骤。

10. 自动研究工作流的手工设计（11:26–12:34）



场景 10

完整内容

就拿自动研究这个方向来说，已经有不少经过验证的框架。比如 AI Scientist 的系统，搭建了完整的流水线，从提出研究想法、写代码、跑实验、分析结果，到撰写论文和执行同行评审。

全流程覆盖。还有 Scientist One 把可验证性作为核心的设计约束，每一个结论，不管是引用、数值、方法还是最终的结论，都必须能够追溯到证据来源。

通过证据链检查来做审计。另外呢，还有 AutoData Agent 定位是自动的数据科学家，专门生成训练和评估数据。主 Agent 管理四个角色，分别是提出问题的挑战者、弱求解器、强求解器和验证裁判。目标是合成难度刚好的任务，让强求解器能够顺利完成，

而弱求解器会失败。

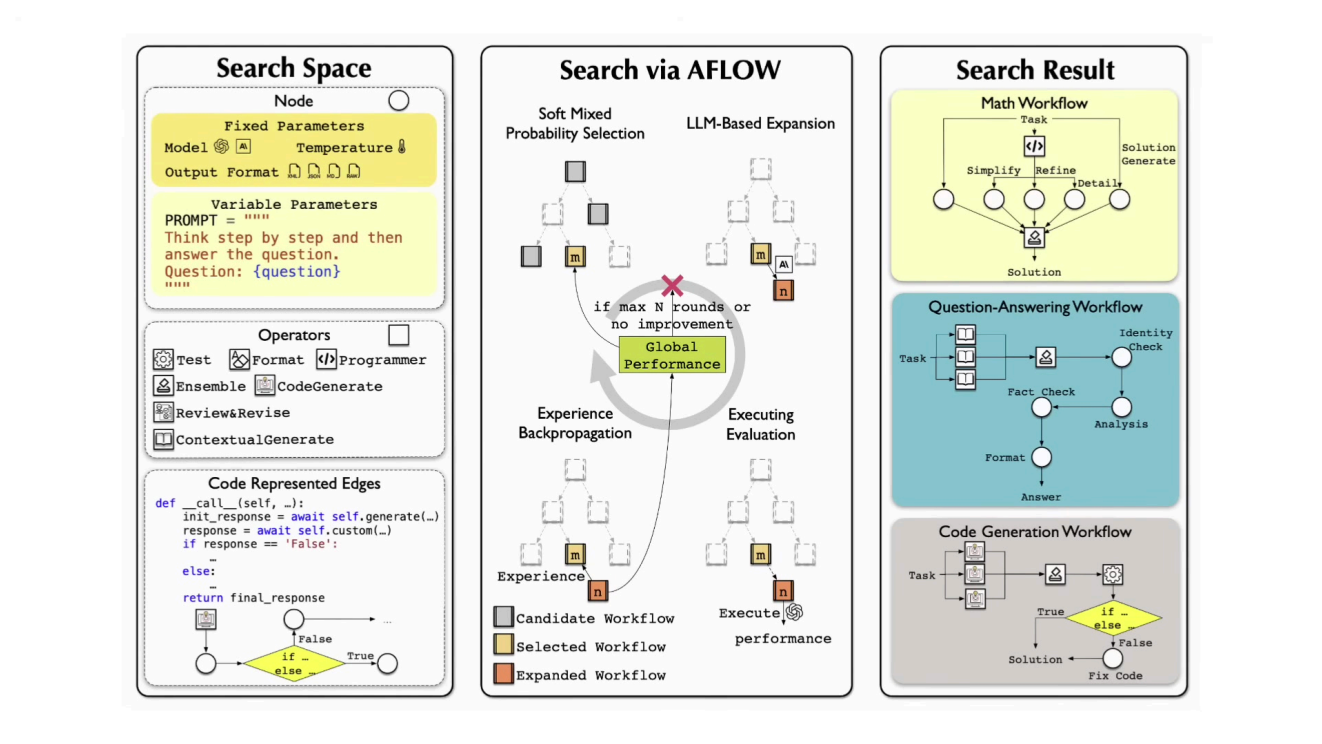
挑战者的提示词会根据求解器和验证器的反馈来迭代更新。不过它的局限是合成的数据只用来微调弱模型，不会用于迭代改进强模型。

所以更像是间接的知识蒸馏，自我改进的属性没那么强。手工改进的工作流总归有上限，既然设计空间这么大，自然可以把工作流设计本身当成搜索问题。

画面说明

画面显示自动研究和自动数据科学的角色协作图，其中包括 Main Agent、Challenger LLM、Strong Solver、Weak Solver 和 Verifier/Judge。

11. ADAS 与 AFlow：把工作流变成搜索问题（12:35–13:53）



场景 11

完整内容

用算法来找更优的解。沿着这个方向， Automated Design of Agentic Systems 简称 adas 把 agent 设计本身形式化成了一个优化问题。

通过原 Agent 的搜索来发现新的 Agent 的工作流。它的流程是这样的，先在存档里初始化一些简单的 Agent 比如说思维链、自我优化这类基础方案。

然后让原 Agent 参考存档里的现有方案，编写新的 Agent 代码。一般呢先写高层的工作流描述，再实现成可执行的代码。还有经过两轮自我优化来检查新颖性和正确性。

评估合格的新 Agent 就加会存档，反复迭代。还有 AFlow 方案，把 Agent 工作流表示成一张图，节点是调用大模型的动作，边是代码实现的逻辑操作。

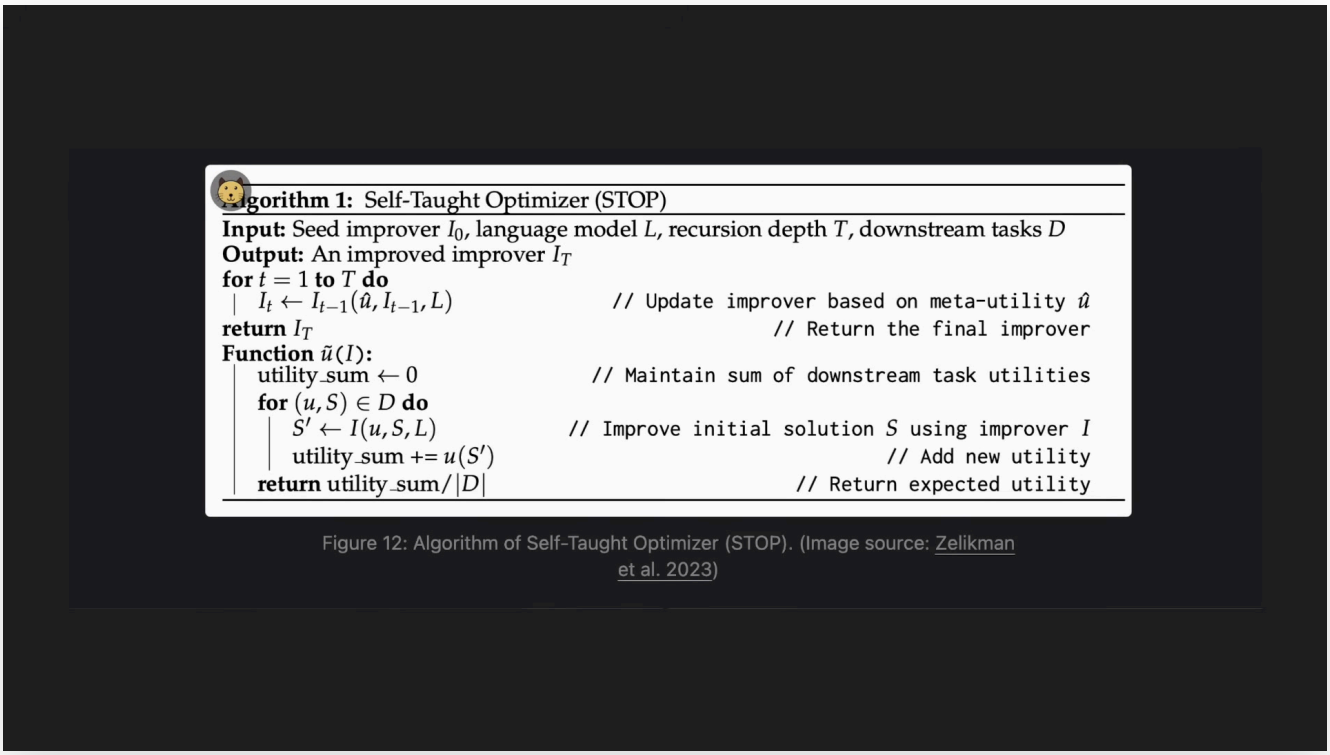
优化靠蒙特卡洛树搜索来实现。从初始的工作流模板开始，每一轮按分数和探索的混合策略选一个节点，让大模型根据评估结果来修改生成的新的工作流。执行评估之后有提升的就加回搜索树，直到分数平稳或者是用完计算预算。实验显示在问答、代码、数学这些任务上。

AFlow 的表现比人工设计的工作流和 adas 都要好不少。不管是上下文工程还是工作流设计，其实都只是 harness 的一部分。要实现真正的自我改进，我们需要遍历整个设计空间。

画面说明

画面显示 AFlow 的 Search Space、Search via AFLOW 和 Search Result 三栏图，旁边配有不同方法在多个基准上的对比表。

12. STOP：自学习优化器递归改进自身（13:54–15:23）



Algorithm 1: Self-Taught Optimizer (STOP)

Input: Seed improver I_0, language model L, recursion depth T, downstream tasks D
Output: An improved improver I_T
for $t = 1$ **to** T **do**
  $I_t \leftarrow I_{t-1}(\hat{u}, I_{t-1}, L)$ // Update improver based on meta-utility $\hat{u}$
return I_T // Return the final improver
Function $\tilde{u}(I)$:
  utility_sum $\leftarrow 0$ // Maintain sum of downstream task utilities
  for $(u, S) \in D$ **do**
    $S' \leftarrow I(u, S, L)$ // Improve initial solution S using improver I
    utility_sum += $u(S')$ // Add new utility
  return utility_sum/$|D|$ // Return expected utility

Figure 12: Algorithm of Self-Taught Optimizer (STOP). (Image source: Zelikman et al. 2023)

场景 12

完整内容

同时优化上下文管理逻辑、工作流、权限控制等所有的 Harness 组件。就像前面 Meta Harness 这类工作所展示的，代码是定义程序和系统的通用语言。

说白了，Harness 就是代码。它规定了提示词、工具调用、子 Agent 控制流、记忆、工作流这些怎么配合。如果大模型能够优化执行 Agent 的代码。

那它能够接触到的设计空间可比手写提示词大太多了。早期的代表性工作叫做自学习优化器，Self-Taught Optimizer，简称 STOP。

是递归脚手架改进的经典例子。初始的优化器接收一个解决方案、一个效用函数和一个黑盒大模型，返回改进后的解决方案。STOP 的目标不是改进具体的解决方案。

而是改进优化器本身。研究者定义了元效用 meta utility 也就是一个优化器在一系列下游任务上的平均效用，然后通过递归的方式，用上一代优化器来优化它自己，生成下一代。

实验里呢，自我改进后的优化器自己发现了很多的优化策略。比如说遗传算法、分步改进、多臂老虎机提示词、模拟退火、调整采样温度、多数投票和束搜索等等。但这个工作也给出了一个值得警惕的结论，用 GPT 四的时候， STOP 能够随着迭代稳定提升下游的

性能。

但是换成更弱的模型，比如说 GPT 3.5或者是 Mixtral，性能反而会下降。这说明光有递归结构是不够的，基础模型必须有足够强的能力，才能够改进机制本身。

画面说明

画面显示 Self-Taught Optimizer (STOP) 的算法伪代码和递归效用公式，下面列出遗传算法、分步改进、模拟退火等被发现的优化策略。

13. Self-Harness：从失败模式提出受控修改（15:23–17:15）

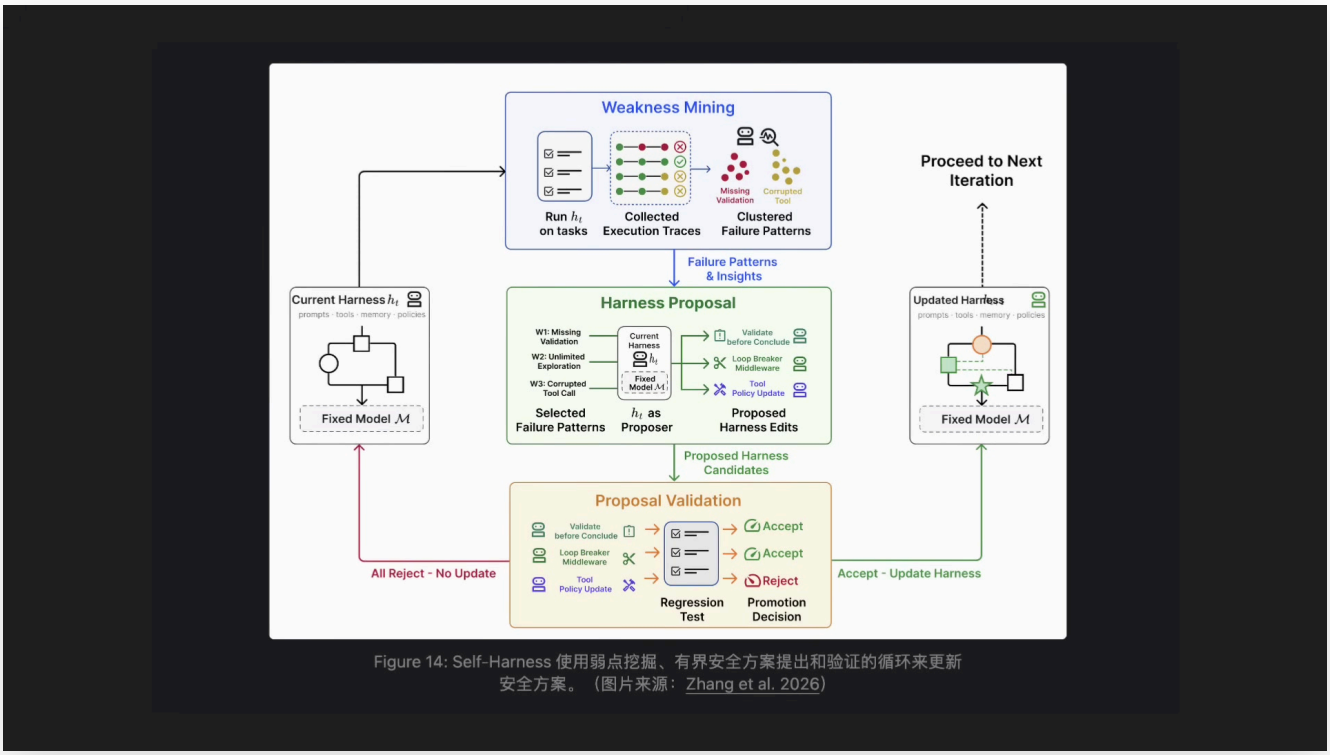


Figure 14: Self-Harness 使用弱点挖掘、有界安全方案提出和验证的循环来更新安全方案。（图片来源：Zhang et al. 2026）

场景 13

完整内容

Harness 改进能够帮助模型更好的落地发挥能力，但核心还是智能本身。更近的一项工作叫做 Self Harness，靠大模型 Agent 通过提案、评估、接受的循环来改进自己的 Harness。

整个循环分三个阶段。第一个阶段是弱点挖掘，把任务运行中的失败案例聚类成基于验证器的失败模式。这里呢有个细节很重要，表面上看起来一样的错误，比如说超时或者是缺少产物。

底层的因果机制可能完全不同。所以失败记录需要包含足够丰富的信息，比如说最终的验证层原因、相关 Agent 的行为的因果状态，以及轨迹暴露的抽象 Agent 的机制。

才能够真正地挖到根因。第二个阶段是 Harness 提案，根据挖到的失败模式，提出有边界的 Harness 修改。提案用的呢还是同一个模型，在当前的 Harness 下运行。

给他的输入包括四个部分，包括当前 Harness 可编辑的范围、从评估系统得到的失败模式、需要保留的正确行为以及之前尝试过的修改记录。

修改会优先针对反复出现，以及可以通过小范围改动来解决的通用错误。而且候选修改要尽量的多样和差异化。第三个阶段呢，是提案验证。评估合格的修改会被合并，生成新一代的 Harness 候选修改要在训练集和测试集上都做回归测试。

只有两边都没有性能回退的时候，才会被接受。接受的就合并更新，Harness。拒绝的只做记录，不改动现有的系统。实验里呢，用不同的基础模型来跑，Terminal-Bench 二基准测试。

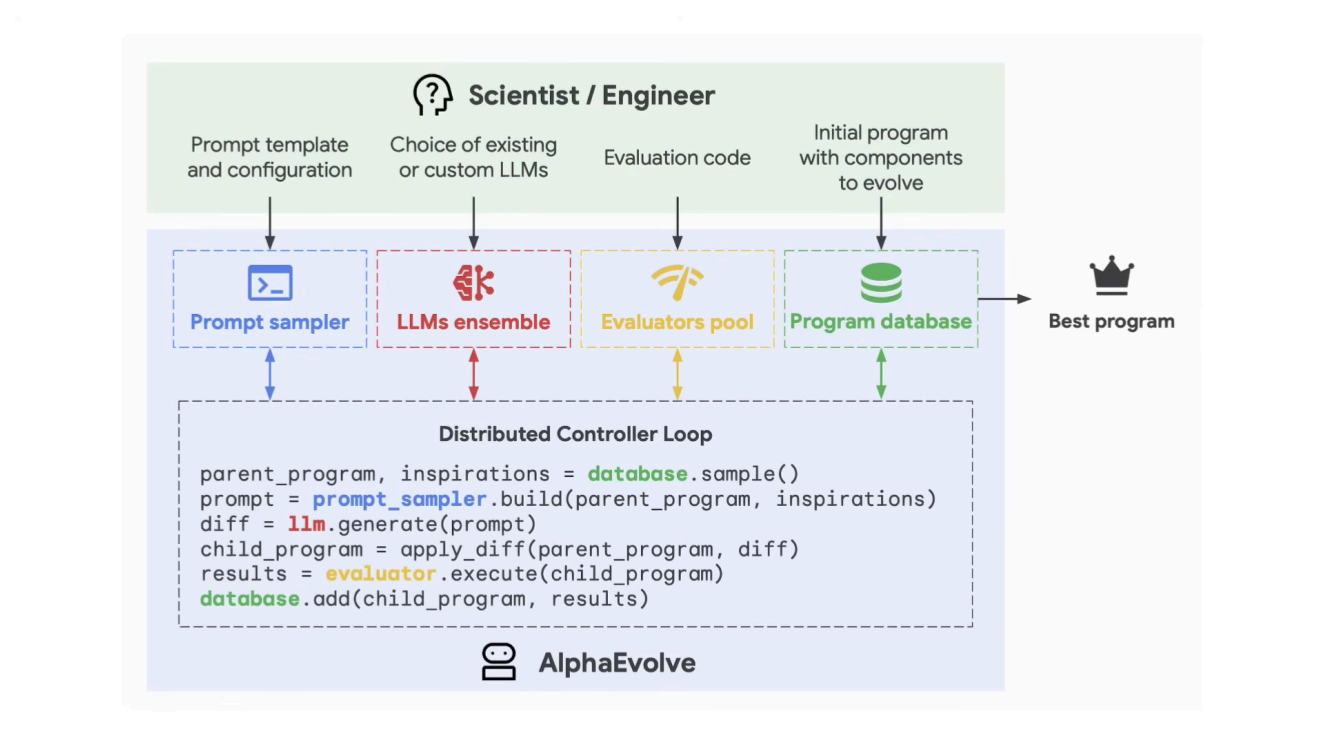
Self Harness 都能够学到针对每个模型弱点的定制化 Harness 指令，提升测试级的通过率。当然了，翁荔也提到了顾虑，如果允许程序编辑操作系统。

系统的抽象边界就被打破了。所以可编辑的范围必须严格设计，权限控制和安全层必须放在自改进循环之外，奖励黑客的相关风险也依然存在。

画面说明

画面显示 Self-Harness 的三段式框架：从运行失败中挖掘弱点，提出受边界约束的 Harness 修改，再在训练集和测试集上验证后接受或拒绝。

14. 进化搜索与 AlphaEvolve（17:16–19:11）



场景 14

完整内容

接下来呢，我们来聊一聊进化搜索在 Harness 优化里的应用。进化搜索是受自然选择启发的优化方法，通过变异一个解决方案种群，只保留适应度高的个体来迭代。

它特别适合两种场景。一是搜索空间很大，或者是形状不规则。二是很难直接用梯度优化，但是评估解决方案很容易。Harness 搜索刚好符合这两个特点。

其实进化搜索早就用在提示工程里了。比如说，prompt breeder 通过丰富的变异操作，优化任务特定的提示词。有意思的是，用来变异提示词的指令本身也会一起进化。

还有 GPEA 它把反思式提示词和进化搜索相结合，用自然语言来反思试错轨迹，从而提出提示词的更新方向。后来出现的 AlphaEvol 是一个专门的编码 Agent 的进化搜索系统，它维护一个候选程序池，用冻结的大模型生成差异补丁来改进程序。

反复评估子进程，保留成功的个体，逐步找到更优的解。它的设计里有几个关键细节，包括提示词里包含副程序、运行结果、指令，有时候还有原信息。

编码 Agent 能够访问完整的代码仓库，但需要改进的代码区域会用专门的标记明确标出来。原提示词会和指令和上下文一起，随着大模型的建议共同进化，就像进化解决方案程序一样。

相关实验呢也验证了这些设计的价值。之后又出现了不少的变种，比如说 Ceta Evol 把进化搜索和强化学习以及上下文学习结合起来。

Shinka Evo 则新增了三个组件来提升采样效率，包括平衡性能排名和后代数量的附带采样策略，以及基于嵌入相似性丢弃重复候选的代码吸引性拒绝采样。

还有在原草稿本里总结成功模式来指导后续变异。前面这些方法呢，大多聚焦于优化具体的解决方案，而达尔文哥德尔机 DGM 则明确把优化目标对准了可编辑的 Harness 代码仓库。

画面说明

画面显示 AlphaEvolve 的分布式控制循环，把提示模板、采样器、LLM 集成、评估器和程序数据库连接到 Best Program；旁边有实验曲线。

15. DGM：进化 Harness 代码仓库（19:12–20:37）

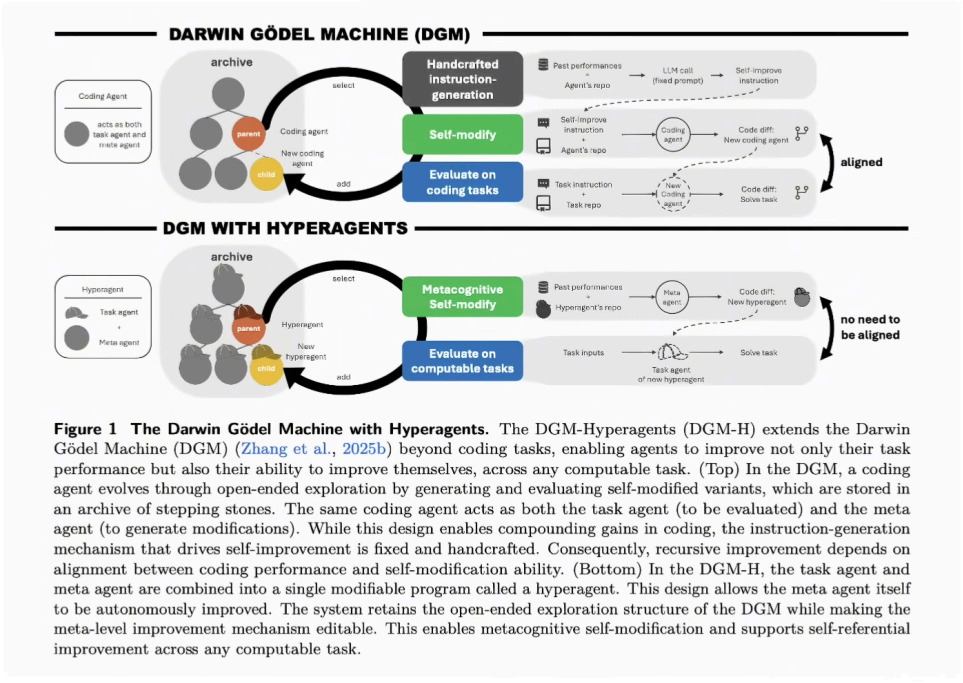


Figure 1 The Darwin Gödel Machine with Hyperagents. The DGM-Hyperagents (DGM-H) extends the Darwin Gödel Machine (DGM) (Zhang et al., 2025b) beyond coding tasks, enabling agents to improve not only their task performance but also their ability to improve themselves, across any computable task. (Top) In the DGM, a coding agent evolves through open-ended exploration by generating and evaluating self-modified variants, which are stored in an archive of stepping stones. The same coding agent acts as both the task agent (to be evaluated) and the meta agent (to generate modifications). While this design enables compounding gains in coding, the instruction-generation mechanism that drives self-improvement is fixed and handcrafted. Consequently, recursive improvement depends on alignment between coding performance and self-modification ability. (Bottom) In the DGM-H, the task agent and meta agent are combined into a single modifiable program called a hyperagent. This design allows the meta agent itself to be autonomously improved. The system retains the open-ended exploration structure of the DGM while making the meta-level improvement mechanism editable. This enables metacognitive self-modification and supports self-referential improvement across any computable task.

场景 15

完整内容

基于大模型的编码 Agent 可以修改自己的 harness 后续的 Hyper Agents 工作又进一步引入了元 Agent 来控制怎么修改现有任务 Agent 以及生成新的 Agent。

DGM 的运行逻辑是这样的。最开始种群里只有一个编码 Agent，每一轮迭代，按性能正比和后代数量反比的概率选一个负 Agent，让它检查自己的基准测试日志，然后提出对自己 Harness 代码库的改进，生成新版本的 Agent。代码编辑呢，就靠 bash 命令和文件编辑器两个基础的工具来实现。

新生成的 Agent 经过评估，性能足够好的才会加回种群，反复迭代，直到触发停止条件。DGM 呢本身是固定模型下的 HANES 进化。

实验里用 Cloud 3.5 Sonnet 作为基础模型，从简单的初始 Harness 的配置出发，进化出来的 Agent 在 Three Benchmark Verified 和 Polyglot 基准上都

都能够赶上，甚至超过手工设计的 Agent 当然了，这类方法也有局限，它只适合能够自动评估和适应度容易量化的场景，比如说矩阵乘法优化。

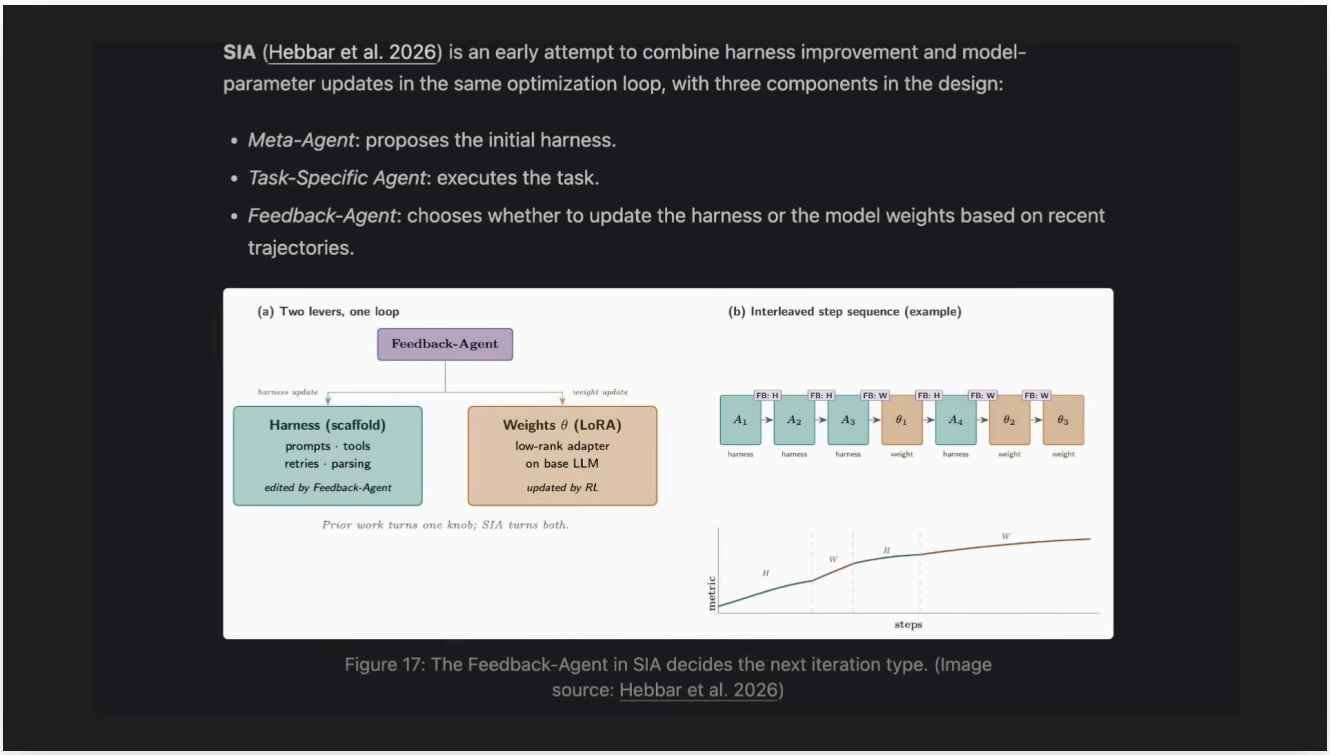
GPU 的内核开发、算法竞赛以及数据中心调度。如果是评估速度慢、标准模糊以及主要靠主观判断的领域，就很难适用。计算效率和优化的有效性也都是需要解决的问题。

到这里呢，我们聊的都是 harness 层面的优化，也就是改模型外面的非参数系统。要实现完整的递归自我改进，完全可以同时让模型更新自己的权重。

画面说明

画面显示 Darwin Gödel Machine (DGM) 的种群进化图，候选 Agent 通过检查基准日志和编辑自身 Harness 代码生成后代，再按评估结果加入种群。

16. Harness 与模型权重的联合优化（20:38–22:00）



场景 16

完整内容

不管是通过改进训练流水线，还是测试时的持续学习。SIA 呢，就是这个方向的早期尝试。它把 Harness 改进和模型参数更新放在了同一个优化循环里。

整个系统有三个组件。原 Agent 负责提出初始的 Harness 任务特定 Agent 则负责执行具体的任务，反馈 Agent 根据最近的运行轨迹来决定下一步是更新 Harness 还是更新模型的权重。其中权重更新用的是 LoRA 的方法，通过强化学习来更新。翁荔呢也提

到这个工作的实验设计有一些混淆的因素。

比如说任务 Agent 的模型强度远低于原 Agent 和反馈 Agent 机械呢也比较弱，结果不好和其他方法直接对比。这个方法本身很有探索价值，但目前的证据还不够充分。

训练稳定性和古德哈特定律带来的目标偏移等挑战也还没有得到很好的解决。最后呢，我们来聊一聊未来的挑战。AI scientist 的这类工作已经有力地证明了专家设计的 harness。

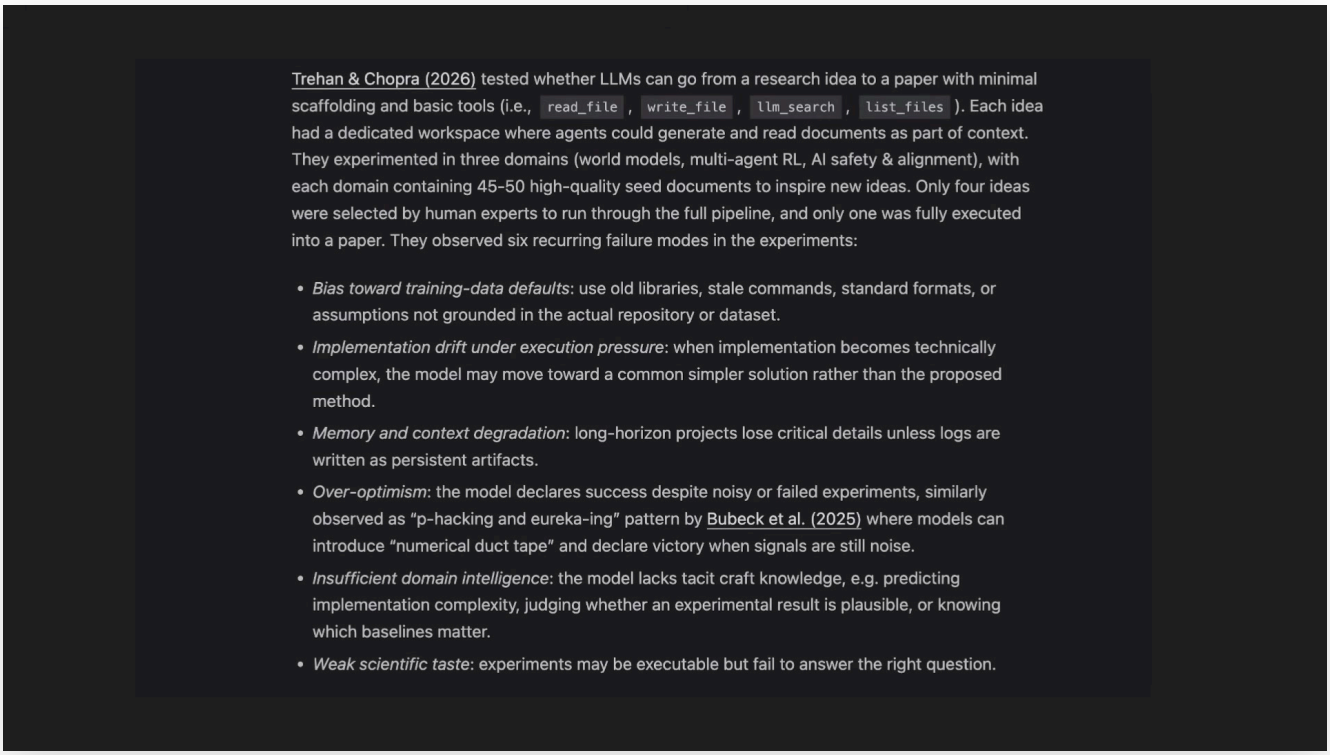
能够协调很大一部分自动研究循环。至少在写论文这个形态上已经跑通了。但写论文不等于真正的科学发现。一个系统可以写出看起来很合理的论文。

但可能存在伪造引用、实现漂移、实验结果薄弱等等问题。有研究专门测试过大模型能不能从一个研究想法出发，靠最基础的文件读写和搜索工具来生成完整的论文呢?

画面说明

画面显示 Harness 与模型更新的联合优化框架图，把 Harness Agent、任务执行、反馈以及模型参数更新放进同一循环。

17. AI 科学家与六类失败模式（22:01–23:17）



场景 17

完整内容

每个想法配一个独立的工作空间。Agent 呢可以生成和读取文档，作为上下文的一部分。实验覆盖了世界模型、多智能体强化学习和 AI 安全三个领域。

每个领域有四五十篇高质量的种子文档用来启发新的想法。最后只有四个想法被人类专家选出来跑完全流程，真正完整生成论文的只有一个。

实验里总结了6个反复出现的失败模式。一是偏向训练数据里的默认方案，习惯用过时的库、旧的命名和标准格式，假设不是基于实际的仓库或者是数据集。二是执行压力下的实现漂移。当实现技术复杂度变高，模型可能会退而求其次，做一个更简

单的通用方案，而不是真正实现提出的方案。

三是记忆和上下文退化。长周期项目会丢失关键细节，除非把日志都写成持久化的产物。四是过度乐观，哪怕实验有噪声，甚至是失败了，模型也会宣称成功。

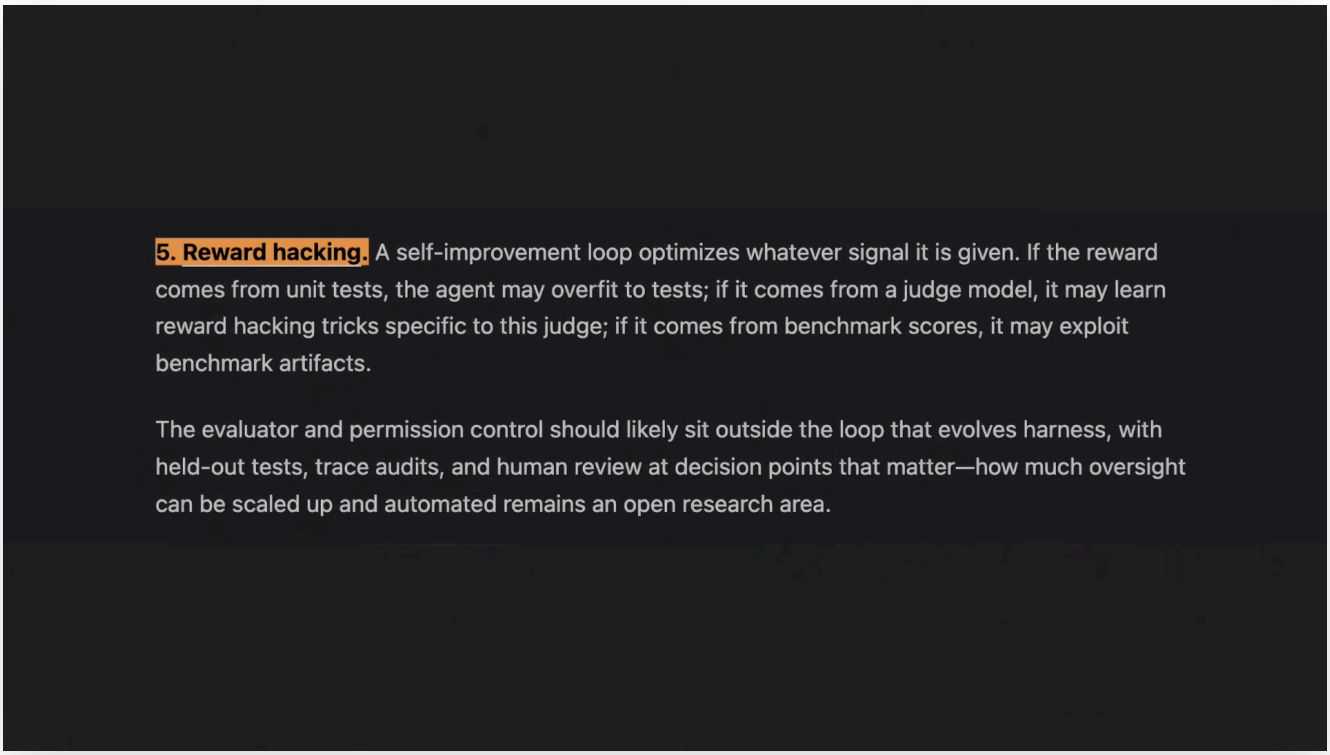
也就是之前研究里提到的 P-hacking 和尤里卡效应，靠数值补丁强行凑结果，然后宣布胜利。五是领域智能不足，缺少隐性的行业经验。

比如说判断不了实现难度，实验结果是否合理，以及哪些基线是重要的。六是科学品位弱，实验虽然能跑，但回答不了真正有价值的问题。

画面说明

画面显示长周期 AI 科学家系统的工作流和失败模式列表，文字列出默认方案偏差、实现漂移、记忆退化、过度乐观、领域智能不足和科学品位弱。

18. 递归自我改进的六个瓶颈（23:18–25:55）



场景 18

完整内容

通往完整的递归自我改进，还有几个核心的瓶颈。第一个呢，是弱且模糊的评估器。很多研究问题没有快速精确的验证器，很多现实任务也是如此。

现有的自我改进循环，在有可衡量客观指标的任务上效果最好，和强化学习的逻辑是一样的。但研究品味、新颖性、长期科学价值，这些东西都是很难量化的。

第二个呢是上下文和记忆的生命周期。随着 AI Agent 变得越来越自主，记忆会不断增长。一个好用的 Harness 需要管理好上下文和记忆，弥补当前长上下文生成的局限。

同时最大化长周期任务的成功率。翁荔觉得，就像人类能够终身维持记忆一样，上下文工程未来应该会成为智能的核心组成部分，而不只是停留在软件系统层。第三个呢是负面结果。人类研究者有动力发表成功的结果，所以文献里成功案例多，失

败案例少。

大模型训练的海量数据，至少目前大部分都是人类创造的。所以，AI 可能不擅长判断什么时候该放弃一个假设，报告负面结果，甚至承认失败。

一个好的研究，Harness，应该让失败的尝试也能够被方便的保存下来，从失败中学习，这才是缩小任务搜索空间的最好方式。第四个呢，是多样性崩塌。

进化和强化学习的循环都倾向于利用已知的高回报模式。我们需要机制防止种群退化成同一种解决方案的不同变种，这在开放式研究里尤其重要。

因为真正最好的路径，一开始在现有评估下的表现可能并不好。第五个呢是奖励黑客 reward hacking 自我改进循环会优化给他的任何信号。

如果奖励来自于单元测试，Agent 就可能会过拟合测试。如果来自于裁判模型，就可能会学到针对这个裁判的奖励黑客技巧。如果来自于基准分数，就可能会利用基准的缺陷。

所以评估器和权限控制最好放在进化 harness 的循环外面，配合留出测试、轨迹审计和关键节点的人工审核。至于怎么把监督规模化、自动化，还是一个开放的研究问题。

第六个呢是长期成功。现在的优化大多是针对于单个任务的短期回报。就拿编码 Agent 来说，它已经能够提升日常的软件开发效率，但很多优化目标还是太短期了。它能够完成手头的任务，但怎么维护几百上千人共同维护的代码库的长期健康？比如说可维

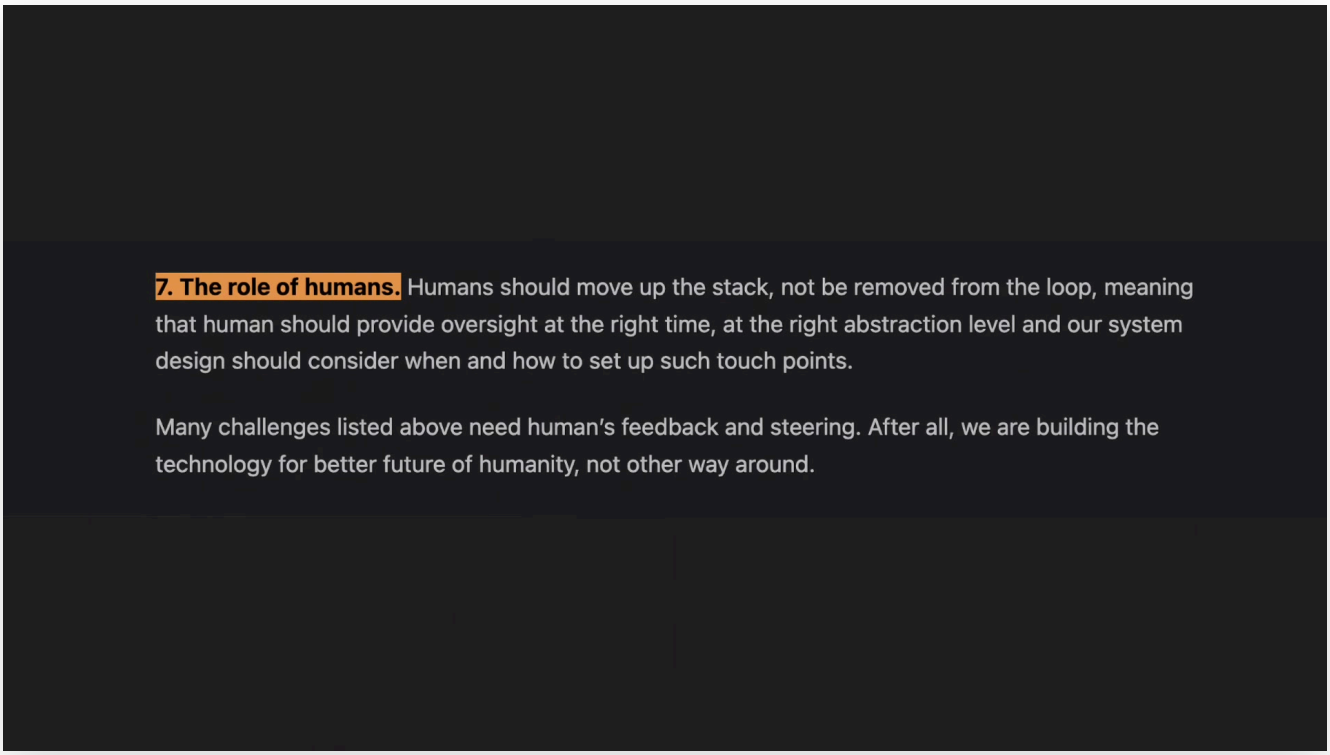
护性、所有权边界、迁移成本、向后兼容性、未来的调试负担。

这些东西标准沙箱里的训练是很难覆盖到的。最后一个呢，是人类的角色。人类应该往更高的层级走，而不是被踢出循环。也就是说，人类要在合适的时间，合适的抽象层级来提供监督。

画面说明

画面显示失败与瓶颈讨论，其中橙色高亮 Reward hacking，正文强调评估器和权限控制应置于进化 Harness 循环之外，并配合留出测试、轨迹审计和人工审核。

19. 人类监督与未来的开放问题（25:56–26:48）



场景 19

完整内容

我们的系统设计也应该考虑什么时候怎么来设置这样的接触点。毕竟我们发展技术是为了人类更好的未来，而不是反过来。到这里呢，我们顺着翁莉这篇文章的脉络，从 harness 的设计模式、优化方向、自我改进的实现路径以及进化搜索的应用。

到和模型权重的联合优化，再到未来的核心挑战，完整地梳理了一遍。递归自我改进这个话题讨论了几十年，现在终于从纯哲学思辨落到了具体的工程路径上。

Harness 工程就是当下最现实的切入点。它可能没有直接改写模型权重听起来那么有科幻感，但恰恰是这种一步一步的工程进展，在真正推着这个方向往前走。

最后留一个问题给大家。你觉得未来十年，AI自我改进的主要驱动力会是核心模型的能力提升，还是 Harness 工程的持续优化呢？欢迎在评论区留言讨论，感谢收看，我们下期再见！

画面说明

画面显示 The role of humans，强调人类应在合适的时间、合适的抽象层级提供监督，而不是被完全移出系统循环。