



北京大学
PEKING UNIVERSITY

内部研讨系列

Harness Engineering——驯服Agent

从理论到落地

AI肖睿团队

20260714@北京



Harness Engineering —— 驯服 Agent



北京大学
PEKING UNIVERSITY

报告概览 · 从模型能力、运行环境到系统治理三个层面重新理解 AI Agent 的构建方式

01

Prompt → Context → Harness → Loop

回顾 Prompt Engineering、Context Engineering、
Harness Engineering 与 Loop Engineering 的演进关系
四阶段层层递进、嵌套包含，工程焦点从模型提示外移到自
驱动化循环系统

02

概念起源 · 核心思想 · 关键问题

分析 Harness Engineering 的概念起源与核心思想
梳理其解决的关键问题：任务漂移、上下文遗忘、
执行失控、结果无法验证

03

OpenAI / Anthropic 代表性实践

结合 OpenAI、Anthropic 等代表性实践
梳理 Harness 的核心架构、设计原则与工程模式
总结 Agent 系统从“能用”走向“可靠”的演进路径

04

Loop Engineering · Agent OS

对 Harness Engineering 的未来发展趋势进行展望
Loop Engineering：长期自主运行与自我驱动
Agent OS 等方向：组件下沉与标准化基础设施





01 AI Agent 工程实践的演进历程

02 Harness Engineering 概念与架构

03 Harness Engineering 核心架构：组件体系

04 Harness 的设计原则

05 Harness Engineering 实践案例分析

06 Agent Harness 系统的实践经验

07 Harness Engineering 的未来演进及前沿模型



当 Agent 从演示走向真实生产环境时，人们很快发现：**模型能力的提升并不意味着系统可靠性的提升。**

主要工程挑战：

- ⚠ **任务漂移与上下文遗忘：** 长周期任务中容易丢失初始目标。
- 🔄 **执行失控与无限循环：** 遇到错误时无法自我纠正，陷入无效重试。
- ✖ **结果无法验证：** 缺乏客观的外部评价标准，导致产出不可靠。
- 💰 **长期运行成本过高：** 效率低下导致 Token 消耗巨大。

核心问题：如何让 Agent **"持续、稳定、可控地完成任务"**成为新的核心问题。





PART 01 ▶

01 AI Agent 工程实践的演进历程

- 1.1 Prompt Engineering
- 1.2 Context Engineering
- 1.3 Harness Engineering
- 1.4 Loop Engineering



P/C

基础阶段：提示与信息

01 Prompt Engineering

核心：如何说对话（优化指令）

02 Context Engineering

核心：模型看到什么（优化输入）

H/L

进阶阶段：驾驭与循环

03 Harness Engineering

核心：系统如何可靠运行（设计环境）

04 Loop Engineering

核心：长期自主运行（自动化循环）

嵌套层级: Prompt Context Harness Loop

- 随着任务从“一次性”到“长期自主”演进，工程焦点外移：模型提示 → 信息环境 → 运行约束 → 自动化循环系统
- 当前顶级实践（OpenAI、Anthropic 等）已深度结合 Harness + Loop，仍依赖前两者打基础
- 挑战与注意：Loop 带来更高自主性，但需警惕 Token 成本、退出条件设计、错误放大风险
- 优秀实践强调清晰 Goal、验证回路和小步安全迭代
- Harness 工程要解决的：执行沙箱、工具协议、上下文压缩、可观测性、验证回路、权限治理



资料整理
火龙果软件



1.1 Prompt Engineering (提示工程)

- **核心关注**: 如何说对话 (Refining Intent / 优化指令) 告诉模型要做什么。
- **做什么**: 设计角色、任务描述、输出格式、Few-shot 示例、Chain-of-Thought 等。
- **适用场景**: 早期聊天机器人、简单问答、代码补全等单轮或短任务。
- **局限**: 模型容易“忘”、幻觉严重、上下文窗口有限, 无法处理复杂任务。
- **对应问题**: 输出质量不稳定, 主要靠“魔法咒语”。

1.2 Context Engineering (上下文工程)

- **核心关注**: 模型看到什么 (Managing Information / 优化输入环境) 给模型正确且足够的信息。
- **做什么**: RAG、记忆系统、会话历史管理、知识库构建、上下文压缩、工具描述注入等。
- **适用场景**: 多轮对话、知识密集型任务、需要外部数据的 Agent。
- **局限**: 长周期任务仍会漂移、累积错误、缺乏全局控制。
- **关系**: 建立在好的 Prompt 之上, Prompt 仍是上下文的一部分。

1.3 Harness Engineering (驾驭工程)

核心关注：整个系统如何可靠运行 (Controlling Execution)

构建 Agent 周围的完整控制系统：

- 架构约束与规则 (Architecture Guardrails)
- 反馈与验证回路 (Evaluator Agent、自动测试)
- 工具权限边界、沙箱、安全机制
- 多 Agent 编排与分工
- 状态管理、记忆持久化、技术债清理

核心理念：

Agent = Model + Harness

模型提供智能，Harness 提供可靠性。人类从“写 Prompt/代码”转向“设计环境与规则”。

适用场景：

生产级、长运行、自主 Agent (如编码 Agent 持续开发项目)。

监控、可观测性、人类干预接口、自我纠正循环等均属于 Harness 的范畴。Harness 包含并超越了前两者，重点转向系统级可靠性。**Harness Harness Prompt Context**

1.4 Loop Engineering (循环工程)



核心关注：如何实现长期自主运行与自我驱动

主要工作：在 Harness 基础上引入时间/调度维度和递归目标达成机制

- 设计自动化的“**循环系统**”，代替人工逐轮 Prompt，由系统按计划/定时触发 Agent。
- 迭代循环：Action → Observe → Reason → Adjust，直到目标完成或阻塞。
- 工作发现与 Triage（自动化发现任务）、Worktrees（并行避免冲突）、子 Agent 生成。
- 持久化状态、退出条件（Stopping Rules）、多循环堆叠（Loop Stacking）。

适用场景：长期自主 Agent（多日/数周编码项目、持续开发、自动化运维、研究循环）

从“人类在循环中”转向“人类设计循环”

核心理念：不再手动 Prompt Agent，而是设计系统（Loop）来 Prompt Agent

Loop 位于 Harness 之上（Harness + Timer + Self-Feeding + Helpers）

一句话总结：构建自主循环系统，让 Agent 针对目标持续迭代，直至完成

关系：嵌套包含前三者。优质 Loop 依赖强 Harness（可靠单次执行）、Context（干净信息）和 Prompt（循环内指令）。

重点从“单 Agent 运行环境”转向“多轮、持久、目标驱动的自动化流程”。



总结 AI Agent 工程实践的四个阶段

- ① Prompt Engineering: 如何说对话, 优化指令, 让模型单次产生更好输出
- ② Context Engineering: 模型看到什么, 优化输入环境, RAG/记忆/知识库
- ③ Harness Engineering: 整个系统如何可靠运行, Agent = Model + Harness
- ④ Loop Engineering: 如何实现长期自主运行, 设计系统来 Prompt Agent

嵌套关系: Prompt $\subset$ Context $\subset$ Harness $\subset$ Loop, 层层递进非替代

工程焦点外移: 从模型提示到信息环境, 再到运行约束, 最后到自动化循环系统

人类从“写 Prompt/代码”转向“设计环境与规则”, 再到“设计循环系统”





北京大学
PEKING UNIVERSITY

PART 02 ▶

02 Harness Engineering 概念与架构

- 2.1 概念起源与贡献
- 2.2 解决了哪些问题



资料整理
火龙果软件



本章从概念起源、核心定义出发，剖析 Harness Engineering 解决的关键问题，理解 Agent 时代的基础设施构建方法。

核心问题

Prompt Engineering 难以支撑长周期、多步骤任务
Agent 执行缺乏外部约束与验证机制
复杂任务中状态、记忆、协作不可控

Harness 价值

为 Agent 提供可复现、可验证的执行环境
将 AI 能力从单次交互升级为生产级流
实现约束、度量、修复的闭环工程化



Harness Engineering 定义

Harness Engineering 是 AI Agent 时代的新兴工程学科，围绕 AI 模型设计和构建完整执行环境 (Harness)

核心公式：Agent = Model + Harness。Model 是智能核心，Harness 是模型之外的一切：系统提示、工具接口、权限控制、沙箱、反馈循环、验证机制等

Harness 像马具：为强大但可能失控的 AI 提供缰绳与约束，确保其高效、可控地完成任务

强调约束 (block)、度量 (measure)、修复 (repair)，超越 Prompt Engineering，适用于长周期自主任务

关键词：约束、度量、修复、外部执行骨架



资料整理
火龙果软件



2.1 Harness Engineering 概念起源与贡献

Mitchell Hashimoto (2026.02): HashiCorp 联合创始人。在《My AI Adoption Journey》中首次提出并命名。强调：发现错误时，不只是重试，而是工程化环境（规则、检查、守卫）让错误永不重现。

OpenAI (2026.02): 发布《Harness Engineering: Leveraging Codex in an Agent-First World》。实证三人团队 5 个月生成约 100 万行生产代码、1500 个 PR，几乎零手动编写代码。证明了高可靠的 Agent-First 开发可行性。

Anthropic: 发布多篇工程文章，聚焦长运行 Agent 的 Harness 设计（有效约束、上下文重置、规划/生成/评估分离等模式）。

意义：标志着从“Prompt 时代”向“Agent 基础设施时代”的转变。

各种 Agent 平台等产品正围绕 Harness 构建能力，Harness Engineering 已成为 AI 开发者应对 Agent 复杂性的主流范式。

从个人实践到行业标杆，Harness Engineering 已成为 AI 工程核心范式

Harness 解决的 7 大问题

- ✓ 1. Agent “会做，但做不完”——缺少任务执行结构
- ✓ 2. AI 自己判断自己正确——执行者与裁判角色重叠
- ✓ 3. 复杂任务失控——边界模糊、子任务遗漏、修改冲突
- ✓ 4. 行为不可预测——同一输入输出不一致
- ✓ 5. 长期任务失忆——上下文丢失、状态难以保持
- ✓ 6. 多 Agent 协作混乱——职责、权限、调度不清晰
- ✓ 7. AI 软件开发无法规模化——缺乏审计、复现、追踪、回滚能力

2.2 解决痛点 (1): 任务闭环与裁判机制

1. 解决 Agent “会做，但做不完” 的问题

典型表现：修改一半忘记目标、上下文过长偏离需求、遇到错误不断循环。Harness 提供“外部执行骨架”（Git、Issue、Code Review、CI/CD、测试体系）保证任务闭环。

普通 Agent (执行者兼裁判)

Agent 写代码



Agent 测试



Agent 判断完成

风险：运动员自己吹哨宣布胜利

Harness 模式 (引入外部验证)

Agent 执行



External Validator 验证



通过才进入下一阶段

转变：从“AI 说完成了”到“系统证明完成了”

验证内容：自动运行测试、检查代码规范、验证接口返回、检查性能指标等。



2.2 解决哪些问题

2. AI 自己判断自己正确

Agent

普通 Agent 中，执行者同时承担裁判角色，
类似运动员自己吹哨宣布胜利。

Harness 引入外部验证器：自动运行测试、
检查代码规范、验证接口返回、检查性能指标。



2.2 解决痛点 (2): 失控管理与确定性结构



3. 解决复杂任务中的失控问题

复杂任务路径: 需求 → 分析 → 拆解 → 多文件修改 → 测试 → 修复 → 部署

复杂任务中 Agent 容易出现边界模糊、子任务遗漏、修改互相覆盖。Harness 通过以下手段让复杂任务变成可管理流程:

- 任务拆解 (Task decomposition) 与 工作流 (Workflow)
- 状态管理 (State management) 与 门控 (Gate)
- 依赖图 (Dependency graph) 明确执行逻辑

核心: 把复杂任务拆成可验证、可回滚的步骤

4. 解决 Agent 行为不可预测的问题

LLM 同一输入不一定产生相同输出。Harness 通过增加确定性结构降低随机性:

- 固定流程与权限限制
- 工具调用规则与状态记录
- 回滚机制降低随机性

让 Agent 从“聪明但不可预测” 变成 “能力强且可管理”



2.2 解决痛点 (3): 长时记忆与协作治理

5. 解决长期任务中的“失忆问题”

企业任务可能持续数小时甚至数天，上下文丢失是核心痛点，导致 Agent 忘记已完成工作和历史决策。Harness 引入：

Memory + State + Event System

- 保存任务状态、已完成步骤、历史决策、错误记录、验证结果。
- 让 Agent 可以：暂停 → 恢复 → 继续执行（类似操作系统保存进程状态）。

长期任务的可靠性依赖状态持久化

6. 解决多 Agent 协作混乱问题

未来开发涉及 Planner, Coder, Tester, Reviewer 等多个角色。Harness 提供统一调度层：

- **Orchestrator (编排器)**、Routing (路由)、Protocol (协议)
- 明确分工、下一步决策及最终权限归属。

多个 Agent → 统一调度层 → 可靠执行



资料整理
火龙果软件



2.2 解决哪些问题

7. 解决 AI 软件开发无法规模化的问题

个人使用 AI 追求“好用”，而**企业使用 AI 需要：可审计、可复现、可追踪、可回滚、可管理。**

Harness 提供完整执行链 (Evidence Chain):



总结： Harness Engineering 是为 AI Agent 构建的一层**执行控制系统**，它通过状态管理、任务编排、工具约束和自动验证，让原本不可预测的 AI 行为变成可持续、可恢复、可验证的软件生产流程。

Harness Engineering 本质

为 AI Agent 构建一层执行控制系统

通过状态管理、任务编排、工具约束、自动验证，让不可预测的 AI 行为变成可持续、可恢复、可验证的软件生产流程

从 “Prompt 时代” 迈向 “Agent 基础设施时代”

核心能力：block（约束）、measure（度量）、repair（修复）

Harness Engineering：让 Agent 成为可靠的生产力伙伴



北京大学
PEKING UNIVERSITY

PART 03 ▶

03 Harness Engineering 核心架构： 组件体系

- 3.1 为什么需要 Harness：Agent 周围的完整控制系统
- 3.2 Context（上下文管理）
- 3.3 Orchestration & State（编排与状态）
- 3.4 Feedback & Verification（反馈与验证）
- 3.5 Observability（可观测性）
- 3.6 Governance（治理）



资料整理
火龙果软件



一级框架：Harness 的核心思想



二级框架：五个工程模块

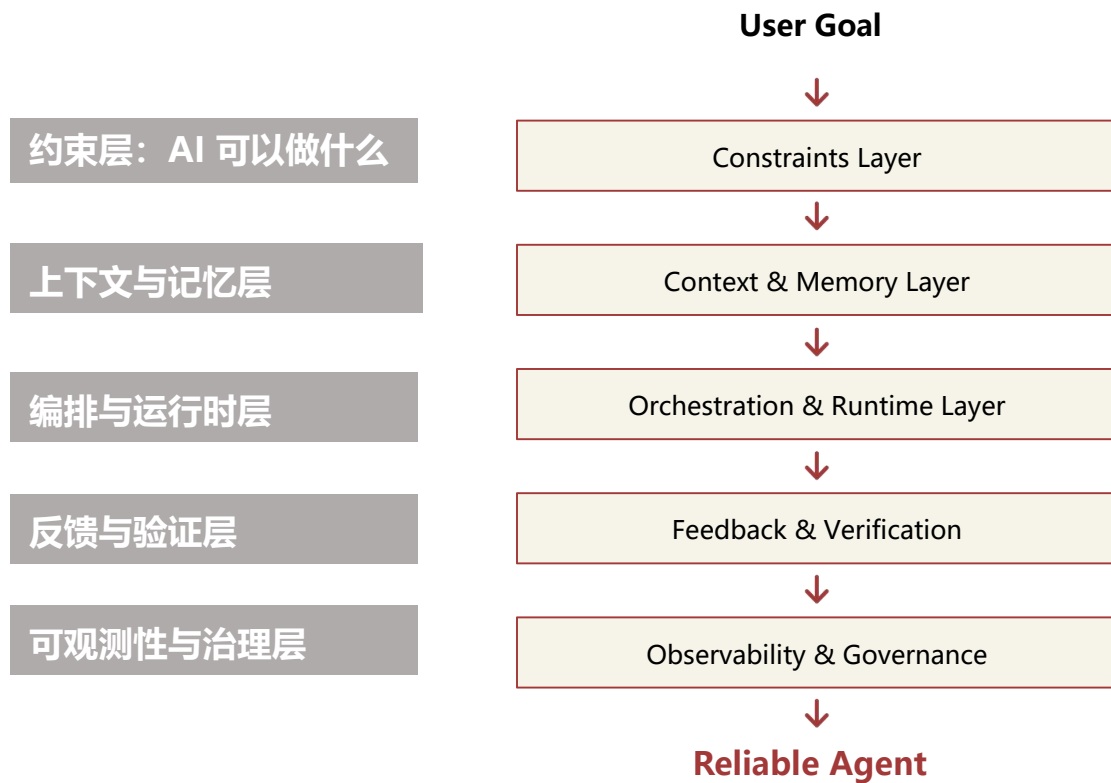
Context	每一步该看到什么信息（相关性过滤、窗口管理）
Memory	跨步骤/跨会话该记住什么（短期 vs 长期）
Orchestration	谁来决定下一步做什么（规划、路由、子任务分解）
State	任务进展到哪了（成功/失败/待重试的状态机）
Feedback	行动结果如何反馈回模型（结构化 vs 自然语言）

三级框架：对应完整工程体系

三级框架：Execution、Tool、Context、Lifecycle、Observability、Verification、Governance 等作为完整工程体系

3.1 为什么需要 Harness: 完整控制系统

Harness 更关注“如何保证 AI 按照预期持续完成任务”。



Harness 更关注 “如何保证 AI 按照预期持续完成任务”。成熟的 Agent 系统需要管理 Agent 生命周期中的每一个环节。

核心问题：AI 可以做什么？（前馈控制，错误发生前避免错误）

(1) 架构约束

定义项目的基本规则：系统架构、Coding Convention、API 设计、仓库结构等。

典型文件：

- AGENTS.md / CLAUDE.md
- ARCHITECTURE.md
- CONTRIBUTING.md

(2) 工具边界

限制 Agent 能够调用的工具：Read/Edit File, Terminal, Browser, MCP Tool。

最小权限原则：

- 是否允许联网/执行 Shell
- 是否允许删除文件

(3) 权限控制

强制执行的机器规则：

- Human Approval
- Sandbox
- Runtime Policy
- Hook / Tool Permission

核心目标：让 Agent 不容易犯错，而不是犯错以后再纠正

这些文档不再只是开发文档，而成为 Agent 的长期行为规范。

Harness 负责建立最小权限原则（Least Privilege）。

Anthropic、OpenAI、Claude Code 等都大量采用 Hook 和 Runtime Policy，把 Prompt 中的规则真正变成机器可以强制执行的规则。

让 Agent 不容易犯错，而不是犯错以后再纠正。



资料整理
火龙果软件



核心问题：AI 应该知道什么？（Harness 的输入管理系统）

Knowledge

- 项目知识库
- 文档管理
- RAG (检索增强生成)

为 Agent 提供完成任务所需的外部知识支撑。

Memory

- 短期 / 长期记忆
- 工作记忆
- 用户偏好
- 历史任务

让 Agent 跨步骤、跨会话保持上下文连贯性。

Compression

- Summary (摘要)
- Compression (压缩)
- Retrieval (检索)
- Sliding Window (滑动窗口)

避免上下文窗口溢出，保持相关信息。

Context Engineering 本质上就是 Harness 的输入管理系统，通过精细化的上下文调度，确保 Agent 在处理长序列任务时的信息有效性。

3.3 Orchestration & State (编排与状态)

核心问题：任务如何推进？现在做到哪里？

Workflow (编排机制)

- Planner / DAG (有向无环图)
- Task Graph / Scheduler

Multi-Agent (多角色协作)

分工独立，职责明确：

Planner → Research → Coding → Reviewer → Evaluator

Runtime (运行时管理)

- Long-running Agent / Checkpoint
- Resume / Retry / Session

State Management (状态管理)

- Event Store / Projection
- Current State / Memory Persistence

核心转变：Agent 不再仅仅依赖有限的上下文，而依赖持续维护的运行状态。通过 Event Kernel 等技术，实现任务的可恢复性与确定性。

3.4 Feedback & Verification (反馈与验证)

核心问题：如何证明任务真正完成？（Harness 最重要的价值）

自动反馈 (Feedback)

- **Evaluator / Reviewer Agent**
- **Reflection** (自我反思)
- **Retry** (重试机制)
- *Agent 可以持续修正自己的输出*

自动验证 (Verification)

- **Unit / Integration Test**
- **Benchmark / Linter**
- **Build** (构建验证)
- *AI 不负责宣布成功，系统负责验证成功*

让 AI 从 “认为完成” → 变成 “证明完成”

3.4 Feedback & Verification (反馈与验证) ——任务契约 (Obligation) 与技术债管理

Obligation (任务契约):

任务不再依赖 Agent 的主观判断，而必须满足预先定义的验收条件：



这类似把一份施工合同和一份验收清单合并成一份文件——既明确“需要完成什么”，也规定“满足哪些条件才算完成”

技术债管理：保证长时运行的健康

- **Garbage Collection** (垃圾回收)：清理冗余状态。
- **Dead Context Cleanup** (死上下文清理)：维持上下文窗口高效。
- **Tech Debt Tracking** (技术债追踪)：记录并解决执行中的潜在问题。

目标：保证 Agent 长时间运行不会不断累积错误，维持系统的长期稳定性。

核心问题：系统如何长期、安全、可维护地运行？

👁️ 3.5 可观测性 (Observability)

- **Logging / Metrics / Trace**: 全量记录。
- **Event / Replay**: 行为回放与问题定位。
- 所有 Agent 行为均可记录、回放、定位与分析性能。

🛡️ 3.6 治理 (Governance)

- **Audit**: 完整审计每一次 Tool Call 和 Decision。
- **Human in the Loop**: Approval / Override。
- **Self-correction**: 发现问题并重新规划恢复。
- 人类拥有最终控制权，确保关键业务安全落地。

治理层最终解决的是企业如何真正放心地把关键业务交给 Agent。



北京大学
PEKING UNIVERSITY

PART 04 ▶

04 Harness 的设计原则



资料整理
火龙果软件



核心目标：让 Agent 在真实环境中稳定、可靠地运行。 设计原则围绕以下三个问题展开：

- ❓ 如何约束 Agent 的行为边界？
- ❓ 如何保证 Agent 的执行过程稳定可控？
- ❓ 如何确认 Agent 的结果值得信任？

核心问题

约束：如何约束 Agent 的行为边界？

循环：如何保证 Agent 的执行过程稳定可控？

质量：如何确认 Agent 的结果值得信任？

Harness 价值

约束要小而明确——最小权限、显式约束、失败收敛

循环要可控可恢复——状态可恢复、反馈结构化、循环有边界

质量要可验证不可假设——验证优于信任、风险决定自主权、可观测性

Harness 的职责是让 Agent 在真实环境中稳定、可靠地运行。归纳为三个层面：约束要小而明确、循环要可控可恢复、质量要可验证不可假设。

Constraints

约束要小而明确

决定 Agent 的行为边界。好的约束不是越多越好，而是越明确越好，让模型始终运行在可预测的范围内。

- 最小权限
- 显式约束
- 失败收敛

Execution Loop

循环要可控可恢复

Agent 的工作本质是不断规划、执行、反馈和修正的循环，Harness 要保证整个循环稳定运行，而非一次完成任务。

- 状态可恢复
- 反馈结构化
- 循环有边界

Quality Gate

质量要可验证不可假设

模型可以生成结果，但不能保证结果一定正确。Harness 需要建立独立于模型之外的质量验证机制。

- 验证优于信任
- 风险决定自主权
- 可观测性

原则 (1) Constraints: 约束要小而明确



北京大学
PEKING UNIVERSITY

核心理念：约束决定边界，明确优于繁多



最小权限

Agent 默认只拥有完成当前任务所需的权限，高风险操作需额外授权。



显式约束

工具能力、使用条件和副作用应明确描述，不依赖模型推断。



失败收敛

错误应限制在局部范围内，避免影响整个任务流程。



让模型始终运行在可预测的范围内



资料整理
火龙果软件



原则 (2) Execution Loop: 循环要可控、可恢复



北京大学
PEKING UNIVERSITY

核心理念：保证循环稳定运行，而非追求一次完成

状态可恢复

任务中断后能够从断点继续，而不是重新开始 (Resumability)。

反馈结构化

工具反馈采用结构化数据，帮助模型准确理解当前状态。

循环有边界

设置明确的完成条件、超时时间和最大迭代次数，避免无限循环。

规划



执行



反馈



修正



资料整理
火龙果软件



原则 (3) Quality Gate: 质量要可验证, 不可假设



北京大学
PEKING UNIVERSITY

核心理念: 建立独立于模型之外的质量验证机制



验证优于信任 (Verifiability over Trust)

任务完成需经过测试、校验或人工确认, 而不是依赖模型自我判断。



风险决定自主权 (Risk-based Autonomy)

根据任务风险决定自动化程度, 高风险操作保留人工介入。



可观测性 (Observability by Design)

记录完整的执行过程, 支持追踪、调试和持续优化。

模型可以生成结果, 但不能保证结果一定正确



资料整理
火龙果软件





PART 05 ▶

Harness Engineering 实践案例分析

- 5.1 案例选择：为什么选择 OpenAI 与 Anthropic?
- 5.2 OpenAI Codex
- 5.3 Anthropic Claude Code
- 5.4 OpenAI vs Anthropic Harness 对比
- 5.5 总结



代表了当前 Agent Engineering 的两种典型路线

OpenAI (Codex)

术语命名: Harness Engineering

由 Ryan Lopopolo (2026.02) 率先提出。倾向于用响亮的新词概括方法论。

核心关注: 工程任务完成

强调让 Agent 进入真实开发流程, 通过环境构建承担复杂任务。

Anthropic (Claude Code)

描述方式: 长任务/状态管理

更倾向于用具体的工程问题描述实践。探讨同类工程问题。

核心关注: 安全自主执行

强调权限控制、上下文管理和人机协作, 确保在真实代码库中可靠工作。

趋势: Agent 的竞争重点正在从“模型能力”转向“围绕模型构建 Harness 能力”的竞争。

核心理念：像工程师一样工作

不再仅仅是代码生成器，而是完整的工程环境执行者。

工程循环 (Engineering Loop)

需求理解



任务拆解



修改代码



运行测试



修复问题



提交结果

内部实证案例 (2025.08 - 2026.01)

100 万行

生产代码产出

1,500 个

Pull Request 合并

0 行

三人团队手写代码量



资料整理
火龙果软件





Constraints: 约束不变量

- 重点是补充**工程基础设施**而非优化 Prompt。
- **约束不变量**而非实现细节（如强制数据结构校验）。
- 让 Agent 快速迭代而不破坏系统基础。

Execution Loop: 任务闭环

- 人定义目标，AI 执行循环。
- 交互几乎完全通过 **prompt** 完成。
- 自主收集上下文（gh、本地脚本），无需人工复制粘贴。

Quality Gate: Agent 互审

- 利用现有体系：编译、测试、CI/CD。
- **Ralph Wiggum Loop**: Agent 相互评审迭代，直到达成共识。
- 绝大部分评审工作从人转移到 Agent。

让系统证明任务完成，而不是让模型声明任务完成。



核心理念：获得能力的同时，严控执行动作风险

🛡️ Constraints: 权限系统

细粒度权限控制体系：

- 文件读取/修改、Shell 执行
- **Allow / Ask / Deny** 策略
- 行动必须经过权限边界

Anthropic 在 Claude Code 中建立了细粒度权限控制体系

文件读取、文件修改、Shell 命令执行等不同操作对应不同授权策略

用户可以通过规则控制不同工具的 Allow / Ask / Deny 权限

对应 Harness 中的 Least Privilege + Explicit Constraints 原则

Agent 可以行动，但行动必须经过权限边界

🔄 Execution Loop: 上下文协同

处理长任务核心：

- **Compaction** (压缩) 维持连贯性
- CLAUDE.md / Memory 机制
- MCP / Skills 工具扩展

为了支持长任务，Claude Code 还需要处理上下文压缩、项目记忆、工具状态、会话恢复等问题。

通过配置文件（如 CLAUDE.md）、Memory 机制和工具扩展机制（MCP、Skills、Subagent），让 Agent 持续理解项目环境。

在上下文窗口填满时，通过压缩（compaction）机制维持长任务的连贯性。

👤 Quality Gate: 风险控制

人机协作控制体系：

- 高风险命令确认 / 修改授权
- 利用 **ML 分类器**减少确认疲劳
- 显著提升自动审批率

人机协作控制体系

高风险命令需要确认

修改文件需要授权

自动模式需要额外的安全机制

Claude Code 重点解决：如何让 Agent 安全地操作真实环境？

OpenAI Codex vs Anthropic Claude Code

OpenAI Codex

核心理念 让 Agent 像工程师一样工作

叙事重点 如何完成复杂工程任务

Constraints 工程基础设施、约束不变量

Execution Loop 工程循环：理解→修改→测试→修复

Quality Gate 测试 + CI/CD + agent 互审

关键机制 Ralph Wiggum Loop

Anthropic Claude Code

核心理念 让 Agent 安全自主执行

叙事重点 如何让行动保持可控

Constraints 细粒度权限系统 Allow/Ask/Deny

Execution Loop 工具协同 + 上下文压缩

Quality Gate 人机协作审批控制

关键机制 CLAUDE.md / MCP / Skills

提醒：以上“叙事重点”反映的是两家公司公开文章的表达侧重，而非两个产品底层能力投入的互斥选择

Harness 是 Agent 能力放大的关键

模型能力只是 Agent 的基础，真正决定 Agent 是否能够进入生产环境的，是围绕模型构建的 Harness

OpenAI 更关注：如何让 Agent 完成越来越复杂的软件任务

Anthropic 更关注：如何让 Agent 在真实环境中安全可靠地行动

两者最终走向相同：通过约束控制行为，通过循环提升能力，通过验证保证质量

这验证了 Harness 三层模型——Constraints → Execution Loop → Quality Gate

——正在成为构建下一代 Agent 系统的基础工程范式



北京大学
PEKING UNIVERSITY

PART 06 ▶

Agent Harness 系统的实践经验

- 阶段一：从口头叮嘱到 CLAUDE.md，规则被记录但是约束力下降
- 阶段二：用 Skill 沉淀方法论，同时警惕它悄悄“夺权”
- 阶段三：引入 Orchestrator，将选择、调度、验收拆开管理
- 阶段四：通过 Hook 实现规则的程序化执行
- 让 Memory + State + Event System 真正落到实处
- 将 Request → Decision → Action → Evidence → Validation → Audit Log 转化为执行链路
- 用 Obligation（任务契约）定义“完成”，不靠 AI 自证
- 回到第一层：约束要小而明确，判断留给 Agent
- Harness 实践建议



资料整理
火龙果软件





控制权逐步迁移到系统机制

一个真实 **Agent** 系统如何在不断暴露问题、修正设计的过程中，逐步建立可靠的控制体系。
整个过程是控制权的迁移：从人的经验和模型自觉，迁移到明确的系统机制。

阶段一：从口头叮嘱到 **CLAUDE.md**，规则被记录但约束力下降

阶段二：用 **Skill** 沉淀方法论，同时警惕它悄悄“夺权”

阶段三：引入 **Orchestrator**，将选择、调度、验收拆开管理

阶段四：通过 **Hook** 实现规则的程序化执行

控制权从口头规则迁移到文本约束

从文本约束迁移到方法沉淀（**Skill**）

从方法沉淀迁移到统一调度（**Orchestrator**）

从调度判断迁移到自动执行（**Hook**）



资料整理
火龙果软件





阶段一：规则被记录，但约束力下降

从口头叮嘱到 CLAUDE.md：零散要求变成显式约束

第一步改进：将经验沉淀到 CLAUDE.md 等规则文件，把零散要求转化为显式约束

规则示例

- 修改代码前先理解现有结构
- 优先采用简单可维护方案
- 修改范围保持精准
- 始终围绕目标执行任务

新的问题

将零散经验沉淀到规则文件，规则文件从几十行增长到数百行后，Agent 对规则的遵循能力明显下降。

所有规则最终仍只是上下文中的文本信息，重要规则并不会自动获得更高优先级。

经验：Prompt 适合表达原则、偏好和工作方式，但不适合作为生命周期级别的强制控制机制。



资料整理
火龙果软件



阶段二：Skill 沉淀方法，但不能“决策”

1. 封装方法与明确分工

Skill 将可复用经验封装为方法模块，例如 brainstorming、systematic-debugging、TDD 等。

- **CLAUDE.md**：负责定义“应该遵守什么规则”。
- **Skill**：负责提供“如何完成某类任务的方法”。

2. “夺权”风险与权责混乱

当 Skill 开始承担任务分配和流程控制时，容易演变成实际上的“决策中心”。

- 抢任务、争夺调用权、缺乏统一裁决标准，都是权责混乱的信号。

设计原则：不能让同一个模块既当运动员又当裁判

决策权（调度者决定流程） → 严格分离 → 执行权（执行者完成任务）

阶段三：Orchestrator 统一选择、调度与验收

系统进一步拆分为三层职责：策略层（规则）、决策层（调度）、执行层（执行）。

(1) 选择执行者

判断当前任务应该交给哪个 Skill 或 Agent。这个过程不能完全依赖模型自我判断，而需要结合规则、任务类型和上下文状态进行决策。

(2) 管理执行流程

当多个任务或 Agent 并行运行时，需要协调依赖关系、执行顺序和资源冲突。类似软件工程中的 DAG 调度或流水线编排。

(3) 管理验收状态

任务完成后，不再简单采用“成功/失败”的二元判断，而是将结果拆分为：

- 放行
- 可恢复阻断
- 硬阻断

Orchestrator 是系统中的调度中心，负责连接任务、能力和验证机制，本身不参与具体执行。

阶段四：Hook 让规则成为自动控制流程

核心关注：将约束转化为程序化执行

如果“这一步是否通过”仍由 AI 自己判断，评审者和执行者依然是同一个角色，系统缺少真正独立的约束能力

- Hook 可在工具调用前、调用后、会话初始化等关键生命周期节点自动触发检查逻辑。
- 规则由“模型需要主动遵守”，转变为“系统自动执行”。
- 早期问题：Hook 虽被触发，但脚本内部仍依赖 AI 自身判断，只是把 Prompt 中的规则迁移到了代码文件里。
- 规则位置发生变化，不等于最终决策权真正交给系统。

设计原则：可靠的 Agent 约束体系不仅需要清晰的规则定义，还需要由机器执行的自动化检查机制。

规则只有具备可验证、可拦截、可执行的能力，才能真正成为系统的一部分。

让 Memory + State + Event System 落地

核心痛点：聊天记录是系统里唯一能还原“发生过什么”的地方，每次复盘都要靠翻聊天记录重建事实。

Harness 应当引入 Memory + State + Event System，主动保存执行过程中的状态和决策记录

📅 状态与步骤

- 任务状态
- 已完成步骤

🧠 决策与错误

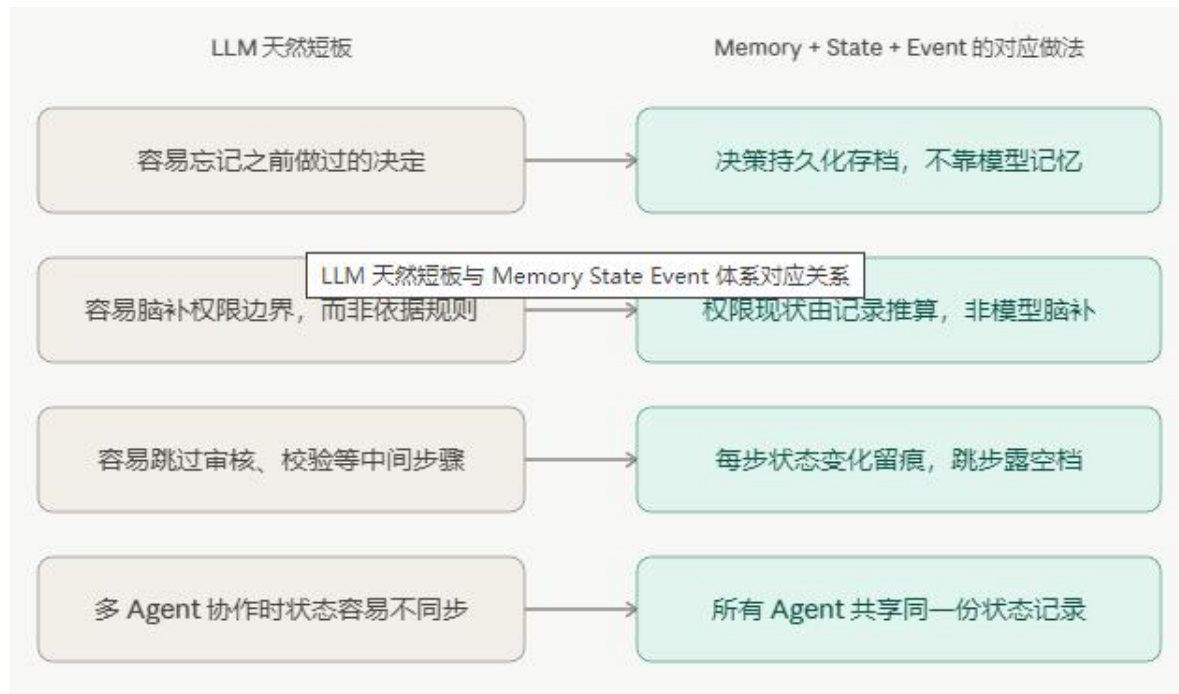
- 历史决策
- 错误记录

🔍 验证与依据

- 验证结果
- 独立存在

核心转变：执行过程中的状态和决策记录，需要由系统主动保存。这些信息必须**独立存在**，并成为系统判断当前进展的依据。

让 Memory + State + Event System 真正落到实处



抽象执行链

将抽象执行链拆解为六个环节，逐步把原本交给 AI 自行判断的部分收归系统：



具体拆解环节

- 1. 请求接入：**接收自然语言请求。
- 2. 路由判定：**用规则加轻量模型联合判定由谁处理、走哪条路径。
- 3. 状态记录：**将实际发生的事情写入 Memory + State + Event System。
- 4. 任务编译：**把请求整理成结构化、范围明确的执行任务。
- 5. 执行准入：**采用放行、可恢复阻断、硬阻断三种状态。
- 6. 验收核对：**核对证据、报告与状态记录是否一致，由系统裁定是否完成。

目标：形成一条可验证、可追溯的完成链路，而不是靠模型一句“我做完了”来结束任务。



资料整理
火龙果软件



Obligation 用任务契约定义“完成”

核心关注：不依赖 Agent 主观判断，用契约定义验收标准

四种可独立校验的身份：

- **请求归属：**谁提出这次任务
- **任务契约：**任务必须满足哪些验收项
- **执行载体：**具体是哪一次执行在处理任务
- **事实记录：**关联 Memory、State、Event System 中的状态变化和事件记录

核心理念：

明确“需要完成什么”与“满足哪些条件”

只有状态记录能回答“发生了什么”，却回答不了“这件事算不算做完”。Obligation 补上的是验收标准。

五步完成链路：

列出任务契约 → 派发执行 → 收集执行产出的证据 → 按契约逐项验证 → 确认完成并归档

五步串联起来，形成一条可验证、可追溯的完成链路，而不是靠模型一句“我做完了”来结束任务。



三类断裂点暴露“完成”机制缺失



北京大学
PEKING UNIVERSITY

核心痛点：三类容易被忽略的断裂点

- **路由完成，但无人接手：**任务已经分配，却没有实际执行者认领，导致任务停留在中间状态；
- **任务派发，但长期无响应：**执行过程陷入沉默，没有超时检测和异常恢复机制；
- **结果返回，但没有验收：**执行产出已经生成，却没有经过契约验证，任务状态被错误推进。

核心结论：很多 AI 系统的问题并非能力不足，而是缺少将“应该完成”转化为“必须经过验证才能完成”的机制。



资料整理
火龙果软件



Constraint 要小而明确，判断留给 Agent



核心原则： Constraint 层只负责机器能够明确判断的事情，把需要探索和推理的问题交给 Agent。

约束层职责： 只处理不依赖模型主观理解、能被系统稳定验证的事务。例如：状态记录、任务契约追踪、证据验证和生命周期规则。

Agent 职责： 涉及方案探索、复杂推理、创造性决策的部分，则交给 Execution Loop 中的 Agent 完成。

设计价值： Constraint 层保持小而稳定，不随任务复杂度膨胀。

系统的智能扩展空间，则留给外围执行循环（Execution Loop）。



资料整理
火龙果软件



总结本章演进实录，提炼出 Harness 实践的 8 条核心建议：



- ✔ **先分类，再动手。** 每当出现新问题，不要条件反射式地全都加入prompt。先想清楚究竟是**约束类问题、状态类问题、契约类问题、验证类问题，还是成本类问题**——问题的性质决定了它该被放进系统的哪一层。
- ✔ **Prompt 的定位是表达意图，而非实施管控。** 它适合承载原则和偏好这类信息，一旦被当作强制执行的控制手段，往往力不从心。
- ✔ **Skill 提供方法与证据，但不该拥有裁决权。** 它可以产出执行结果、给出佐证,但任务的分配权和最终是否“合格”的判定，不应该交给 Skill 自己。
- ✔ 随着工具数量增长，系统真正需要的是一套**统一的约束（Constraint）模型**来收拢这些能力,而不是任其野蛮生长。
- ✔ 一切“必须完成”的事项，都应归到统一的任务契约机制中追踪，不能让它们零散地存在于 prompt 措辞、临时性报告或散落的脚本里，那样既难以核查也容易遗漏。
- ✔ **“完成”不能自证。** 任何完成状态的认定,都必须依赖外部证据和验证结果,而不能仅凭执行者自己的一句声明。
- ✔ **强能力不等于可直接信任。** 即便是能力很强的 Agent,它的输出也需要先经过转化,变成可验证、可审查的形式,才能进入系统的下一环节。
- ✔ **约束层要保持“小而稳”。** 真正的扩展性和智能判断，应该交给 Execution Loop 去承担,而不是不断往 Constraint 层里堆功能。

这段实录也印证了第五章提出的三条设计原则——**约束要小而明确、循环要可控可恢复、质量要可验证不可假设。**



PART 07 ▶

Harness Engineering 的未来演进 及前沿模型

- 7.1 从 Prompt 到 Loop: Agent 工程范式的持续演进
- 7.2 人机协作边界的重新划分
- 7.3 仍有挑战
- 7.4 展望：模型竞争格局下的未来





从 Prompt 到 Loop 的范式演进、人机协作边界的重新划分、仍待解决的挑战，以及御二家竞争格局下的未来展望。

- 7.1 从 Prompt 到 Loop: Agent 工程范式的持续演进
- 7.2 人机协作边界的重新划分
- 7.3 仍有挑战：持久化记忆、多 Agent 组织、长期自主性
- 7.4 展望：前沿模型竞争格局下的未来



从 Prompt 到 Loop: Agent 工程范式的持续演进



北京大学
PEKING UNIVERSITY

核心关注：这一演进呈现出清晰的嵌套与因果关系：前一阶段的局限性成为下一阶段的核心驱动力。

核心演进路径：

- 从控制输出
- 到控制环境
- 再到控制反馈循环

Loop Engineering 的延伸：

Harness 解决“Agent 的操作系统问题”（提供可靠的运行时环境）；Loop Engineering 解决“Agent 的控制理论问题”（通过 feedback、evaluation、correction 与 persistence，实现对长期目标的闭环控制）。



资料整理
火龙果软件



随着 Harness Engineering 走向生产落地，其内部组件正在发生“下沉”趋势——部分能力从定制化工程转变为标准化基础设施。

Context Engineering 成为运行时核心能力：

- 过去：上下文是“一次性输入内容”；
- 如今：成为动态运行时资源，需要被主动检索、压缩、刷新、版本管理和治理。

核心理念：

对传统软件而言，数据是程序处理的对象；而在 Agent 系统中，Context 本身成为需要持续治理的运行时资产。

1. Agent 大规模接管的领域

Harness Engineering 的成熟推动了人机分工的深刻重构，Agent 正在接管：

- **执行 (Execution)**：具体任务的落地与实施。
- **验证 (Verification)**：结果的核对与质量把关。
- **常规决策与优化 (Routine Decision & Iteration)**：重复性的判断与持续改进。

2. 人类聚焦的核心价值

人类的工作重心从“亲力亲为”转向系统级治理：

- **目标设定 (Goal Setting)**：定义任务的终点与愿景。
- **约束定义 (Constraint Definition)**：设定 Agent 运行的边界与规则。
- **价值判断与最终责任 (Value Judgment & Accountability)**：核心决策的裁定。

人类负责定义空间 → Agent 负责在空间内高效搜索

核心关注：人类负责定义空间，Agent 负责在空间内高效搜索。

人机分工的深刻重构：

- **传统软件时代：**人类需亲力亲为完成大部分实现细节。
- **Agent 时代：**人类工作重心转向意图澄清、边界设定与系统级治理。

核心理念：

这种分工有望显著提升整体生产力，同时也对工程师的技能模型提出新要求——从“写代码”转向“设计与驾驭智能系统”。

Agent 演进仍有挑战：从任务型到组织型/持久型



北京大学
PEKING UNIVERSITY

当前局限：主流 Agent 多为任务导向（接受目标 → 执行 → 结束），记忆与经验难以跨任务持久积累。

📅 持久化记忆与改进

如何在长期运行中积累可靠经验，实现可控的自我优化，而非失控漂移？

👥 多 Agent 组织能力

多个专业 Agent 如何形成稳定分工、协调与治理机制，类似人类组织？

🚧 长期自主性边界

在缺乏持续人类监督的情况下，如何保持目标一致性、安全性与可解释性？

核心挑战：Agent 从“任务型”向“组织型/持久型”演进仍存在很大差距，这需要底层架构的深刻变革。

核心背景： Harness Engineering 的演进正处于激烈的行业竞争前沿

截至 **2026 年 7 月 10 日**，短短两周内发生的一系列产品和模型动态，深刻揭示了 Agent 工程范式的未来走向。

- OpenAI 与 Anthropic 的“贴身缠斗”进入白热化阶段。
- 模型发布、产品重构与政府安全审查高度同步。
- Harness 能力正在从“团队手工搭建”转向“平台化能力封装”。

OpenAI: GPT-5.6 三档能力矩阵与产品架构重组



GPT-5.6 家族能力档位 (7月9日全量开放) :

Sol: 旗舰模型, 主打复杂推理和长周期 Agentic 任务, 新增 max 推理强度和 ultra 多子智能体协作模式。

Terra: 性能对齐 GPT-5.5, 价格降低一半, 主打日常均衡场景。

Luna: 最快、最便宜的档位, 服务高频简单任务。

OpenAI: Codex 并入 ChatGPT 桌面应用, Harness 平台化

Codex 并入新版 ChatGPT 桌面应用, 原本独立的 Codex App 变成了 Chat、Work、Codex 三种模式共存的统一客户端。

Harness 设计信号:

这套“旗舰 / 均衡 / 轻量”三分法本身, 意味着不同风险等级、不同复杂度的任务, 理应路由到不同能力和成本的模型上。

核心理念:

Memory、State、Orchestration、Quality Gate 这些原本需要工程团队从零搭建的模块, 正在被逐步封装进标准产品里。

1. 原始推理智能的高光表现

Claude Fable 5 在开发者群体中获得了极高评价：

- **SWE-Bench Pro**：拿到 80.3% 的成绩，显著超过 GPT-5.6 Sol 的 64.6%。
- **开发者评价**：被形容为“需要跨越星系时才动用的曲速引擎”，在原始推理上更胜一筹。

2. 纳入政府安全评估的治理路径

Fable 5 的发布经历了一套典型的“治理剧本”：

- **发布与管制**：6月9日发布，6月12日因出口管制暂停访问，7月1日恢复。
- **行业共性**：与 OpenAI 经历了高度同步的“预览 → 审查 → 放量”路径。

小范围预览 → 政府安全审查 → 分阶段合规放量

视线转回国内：GLM 系列的亮眼表现



国内智谱 AI 的 GLM 系列同样保持了很快的节奏

最新的 GLM-5.2 发布即拿到 LMArena 代码榜开源模型第一、全球模型第二的成绩

主打“规划-实现-迭代”的工程闭环，价格保持明显优势；在中文语境理解、工具集成和本土工程实践适配上展现出独特优势

展望：期待 GLM 在 Harness Engineering 的标准化、开源生态建设和企业级安全治理等方面贡献更多创新实践，推动 Agent 从“可用”走向“可靠”，再走向“自主”。

总结：竞争验证了执行系统的重要性



北京大学
PEKING UNIVERSITY

核心结论：模型能力军备竞赛仍在继续，但真正决定 Agent 生产落地的是围绕模型构建的执行系统。

1. Harness 能力产品化： OpenAI 将 Memory、State、Orchestration 等模块封装进 ChatGPT Work，标志着从手工搭建走向平台化。

2. 工程范式收敛： “通过约束控制行为，通过循环提升能力，通过验证保证质量” 已成为顶尖实验室公认的工程范式。

未



资料整理
火龙果软件





感谢各位老师和同学的批评指导

欢迎沟通交流



资料整理
火龙果软件

