Agent Harness ????????????? 12 ???

> 来源: Akshay Pachaar,《The Anatomy of an Agent Harness》(2026-04-06)
> 推文: https://x.com/akshay_pachaar/status/2041146899319971922 本仓库归档: `Agent/design/harness.pdf`

????? Agent Harness

你已经搭过 chatbot，也许还接了几个工具的 ReAct 循环——demo 跑得很好。可一旦要上生产，问题立刻浮现：模型忘掉三步前做过的事、工具调用静默失败、上下文窗口被垃圾塞满。

> 问题不在模型，而在模型周围的一切。

LangChain 在 TerminalBench 2.0 上做过一个"反事实"实验：模型不变（同权重），只改外围基础设施，排名从 30 名开外飙到第 5。另一项研究让 LLM 自己优化基础设施，pass rate 达到 76.4%，超过人手设计的系统。

这套基础设施有了正式名字：**Agent Harness**（智能体外壳）。

1.1 ??

> **Harness** 是包裹 **LLM** 的完整软件基础设施：编排循环、工具、记忆、上下文管理、状态持久化、错误处理、护栏。

- Anthropic 的 Claude Code 文档：SDK 就是 "the agent harness that powers Claude Code"
- OpenAI 的 Codex 团队明确把 "agent" 与 "harness" 当同义词使用
- LangChain 的 Vivek Trivedy 给出了经典口径：

> **"If you're not the model, you're the harness."**（不是模型的部分，就都是harness。）

1.2 ?????"Agent" vs "Harness"

- **Agent**：用户感知到的"涌现行为"——目标导向、会用工具、会自我纠错的实体
- **Harness**：产生这种行为的机器

> 当有人说"我做了个agent"，本质上是：做了个**harness**，把它指向某个模型。

1.3 ????·??????Beren Millidge, 2023?

裸 LLM = 一颗没有RAM、没有硬盘、没有I/O的CPU：

计算机	**LLM Agent**
RAM（快但小）	Context window
磁盘（大但慢）	外部数据库
设备驱动	Tool 集成
操作系统	**Harness**

??????

围绕模型的三层同心圆：

1. **Prompt Engineering**：雕琢喂给模型的指令
2. **Context Engineering**：管理模型在何时看到什么
3. **Harness Engineering**：包含上面两者，再加上完整的应用基础设施——工具编排、状态持久化、错误恢复、校验回路、安全约束、生命周期管理

> Harness 不是prompt的包装。它是让自主agent行为成为可能的完整系统。

????? Harness ? 12 ???

> 综合 Anthropic、OpenAI、LangChain 与广泛实践者社区的工程经验。

---

1. Orchestration Loop??????

整个心脏。实现 **TAO** 循环（**Thought-Action-Observation**），也叫 ReAct loop:

```
组装 prompt → 调 LLM → 解析输出 → 执行 tool calls → 喂回结果 → 循环至完成
```

机制上往往只是一个 **while** 循环。复杂度全在循环管理的东西，而不在循环本身。

> Anthropic 把自家 runtime 称为 **"dumb loop"**：循环越笨越好，所有智能都留在模型里，harness 只管轮次。

---

2. Tools????

工具是 agent 的 "手"。以 Schema 形式声明（name、description、parameter types），注入到 LLM 上下文中告诉模型有哪些能力可用。Tool 层负责:

- 注册（registration）
- Schema 校验（validation）
- 入参提取（argument extraction）
- 沙箱执行（sandboxed execution）
- 结果捕获（result capture）
- 把结果格式化回模型可读的 observation

典型实现:

- **Claude Code**: 6 大类工具——文件操作、搜索、执行、Web 访问、代码智能、subagent 派生
- **OpenAI Agents SDK**: function tools（`@function_tool`）+ hosted tools（WebSearch / CodeInterpreter / FileSearch）+ MCP server tools

---

3. Memory????

记忆按时间尺度分两层:

- **Short-term memory**: 单会话内的对话历史

- **Long-term memory**: 跨会话持久化
  - Anthropic: `CLAUDE.md` 项目文件 + 自动生成的 `MEMORY.md`
  - LangGraph: namespace 组织的 JSON Stores
  - OpenAI: SQLite/Redis 支持的 Sessions

Claude Code 的三层结构：

1. 轻量索引（每条 ~150 字符，始终常驻）
2. 主题详情文件（按需取入）
3. 原始 transcript（只能搜索访问）

> 关键原则：**agent** 把自己的记忆只当作"提示"，行动前必须先去验证真实状态。

---

4. Context Management???????

很多 agent 在这里静默失败。根本问题是 **Context Rot**:

- 关键内容落在窗口中段时，模型性能下降 **30%+**（Chroma 研究，与 Stanford "Lost in the Middle" 互证）
- 百万 **token** 窗口也照样会随上下文增长发生指令遵循退化

生产策略：

策略	做法
Compaction	临近上限时摘要历史（Claude Code 保留架构决策与未解 bug，丢弃冗余工具输出）
Observation Masking	屏蔽老的工具输出但保留工具调用本身（JetBrains Junie）
Just-in-time Retrieval	只持有轻量标识符，按需加载（Claude Code 用 grep / glob / head / tail，而不是直接读全文件）
Sub-agent Delegation	子 agent 充分探索，但只返回 1k–2k token 的浓缩摘要

> Anthropic context engineering 指南的目标：找到使期望结果概率最大化的"最小高信号 **token** 集合"。

---

5. Prompt Construction?Prompt ???

每一步模型实际看到的东西。层级化拼装：

```
System Prompt → Tool Definitions → Memory Files → Conversation History → Current User Message
```

OpenAI Codex 采用严格优先级栈：

- Server-controlled system message（最高）
- Tool definitions
- Developer instructions
- User instructions（级联的 AGENTS.md，**32 KiB** 上限）
- Conversation history

---

6. Output Parsing??????

现代 harness 普遍依赖原生 **tool calling**：模型直接返回结构化的 `tool_calls` 对象，不再需要解析自由文本。

```
有 tool calls？ → 执行 + 循环
没 tool calls？ → 这就是最终回答
```

结构化输出：OpenAI 与 LangChain 均通过 Pydantic 模型做 schema 约束。

老式的 `RetryWithErrorOutputParser`（把原 prompt + 失败补全 + 解析错误一并喂回模型）作为兜底仍然可用。

---

7. State Management??????

四种代表性流派：

框架	做法
LangGraph	类型化字典在图节点间流动，reducer 合并更新；super-step 边界打 checkpoint，可断点恢复、时间旅行调试
OpenAI	四种互斥策略：应用内存 / SDK Sessions / 服务端 Conversations API / 轻量的 `previous_response_id` 链式拼接
Claude Code	用 **git commit** 当 **checkpoint**，**progress** 文件当结构化草稿本

---

8. Error Handling??????

为什么这件事至关重要：

> 10 步流程，每步成功率 99% → 端到端只剩 **~90.4%**。错误会快速复利。

LangGraph 的四类错误：

类型	处理
Transient (瞬时)	retry + backoff
LLM-recoverable (模型可自纠)	错误以 `ToolMessage` 形式回传，让模型自我调整
User-fixable (需要人介入)	中断 (interrupt) 等待人类输入
Unexpected (意外)	向上冒泡，便于调试

- Anthropic：在 tool handler 内部捕获失败，作为 error result 返回，保持循环不中断
- Stripe 生产 harness：最多重试 **2** 次

---

9. Guardrails and Safety???????

OpenAI SDK 的三层：

- **Input guardrails**：在第一个 agent 上运行
- **Output guardrails**：在最终输出上运行
- **Tool guardrails**：在每次工具调用上运行
- **Tripwire** 机制：触发即立刻停机

Anthropic 的核心理念：把权限执行从模型推理中架构性分离。

- 模型决定想做什么
- 工具系统决定允许做什么

Claude Code 把约 **40** 个独立工具能力分别上闸，分三阶段：

1. 项目加载时建立信任 (trust establishment)
2. 每次调用前检查权限 (permission check)
3. 高风险操作要求显式用户确认

---

10. Verification Loops??????

这是把玩具 **demo** 与生产 **agent** 区分开的关键。

Anthropic推荐三种:

- **Rules-based feedback**: 测试、Linter、类型检查
- **Visual feedback**: UI任务用Playwright截图比对
- **LLM-as-judge**: 另一个subagent来评估输出

> Claude Code作者**Boris Cherny**: 给模型一个能验证自己工作的方式，质量提升**2**到**3**倍。

---

11. Subagent Orchestration?? agent ???

Claude Code的三种执行模型:

模式	含义
Fork	字节级完整副本（复制父上下文）
Teammate	独立终端面板+文件邮箱通信
Worktree	自己的git worktree，每个agent独立分支

- **OpenAI SDK**: agents-as-tools（专家处理有界子任务）+ handoffs（专家完全接管）
- **LangGraph**: subagent实现为嵌套state graph

---

12. Long-Horizon Execution????? / Ralph Loop ?

> 原文标题说12个组件、正文显式编号到11；这里把"跨上下文窗口的长程执行"列为第12项——它在PDF中以独立模式**"Ralph Loop"**详细展开。

问题：单次会话可能跑1–2轮就结束，但复杂重构任务可能跨多个上下文窗口——简单的循环根本撑不到完成。

Anthropic的两阶段**Ralph Loop**:

1. **Initializer Agent**: 搭好环境
    - init脚本
    - progress文件
    - feature列表
    - 初始git commit

2. **Coding Agent**
   - git log + progress
   - feature
   - → commit →

> “

/ tool call / / token / guardrail tripwire / / /

??????? 7 ???

12

1. Prompt Assembly	system prompt + tool schemas + memory files + conversation history + **prompt** Lost in the Middle
2. LLM Inference	API text / tool call requests /
3. Output Classification	tool call → tool call → handoff → agent
4. Tool Execution	→ → →
5. Result Packaging	LLM **error result**
6. Context Update	compaction
7. Loop	1

> “ 1–2 tool call

?????????

Anthropic Claude Agent SDK	单一 `query()` 函数，返回流式异步迭代器	"dumb loop"；Gather-Act-Verify 循环（gather context → take action → verify results → repeat）
OpenAI Agents SDK	`Runner` 类（async/sync/streamed）	**Code-first**：用原生 Python 写工作流，不用 graph DSL
OpenAI Codex Harness	三层架构	Codex Core（agent + runtime）+ App Server（双向 JSON-RPC）+ 客户端（CLI / VS Code / Web）。**"Codex** 模型在 **Codex** 表面感觉更好**"**，因为共享同一 **harness**
LangGraph	显式 state graph：`llm_call` 与 `tool_node` + 条件边	从已废弃的 `AgentExecutor` 进化而来；原生支持多 agent
LangChain Deep Agents	明确使用 "agent harness" 一词	内置工具 + 规划（`write_todos`）+ 文件系统 + subagent + 持久记忆
CrewAI	Agent / Task / Crew + Flows	角色制（role/goal/backstory/tools）；Flows 提供 "intelligence where it matters"的确定性骨架
AutoGen → Microsoft Agent Framework	Core / AgentChat / Extensions 三层	五种编排模式：sequential / concurrent (fan-out/fan-in) / group chat / handoff / **magentic**（manager agent 维护动态任务账本协调专家）

?????? Harness ? 7 ?????

#	决策	关键 **trade-off**
1	单 **agent vs** 多 **agent**	Anthropic & OpenAI 都建议先把单 **agent** 做到极致。多 agent 增加额外的路由 LLM 调用、handoff 时的上下文丢失。只在工具数量超过约 **10** 个重叠工具，或任务领域明显分离时才拆分
2	**ReAct vs Plan-and-Execute**	ReAct 每步都交错推理与动作（灵活但每步成本高）；Plan-and-Execute 把规划与执行分离。**LLMCompiler** 报告比顺序 **ReAct** 快 **3.6×**
3	上下文窗口管理策略	五种：time-based clearing / summarization / observation masking / structured note-taking / sub-agent delegation。**ACON** 研究：**26–54% token** 削减 **+ 95%+** 准确率保留（关键：reasoning trace 优先于 raw tool output）

#	决策	关键 **trade-off**
4	校验回路设计	Computational（测试、 linter）＝确定性 ground truth；Inferential（LLM-as-judge）＝抓语义问题但增加延迟。 Martin Fowler / Thoughtworks 把它们框成 **guides**（feedforward，行动前引导） vs **sensors**（feedback，行动后观察）
5	权限与安全架构	Permissive（快但险，自动批准多数动作） vs Restrictive（安全但慢，每个动作都需批准）。取决于部署场景
6	工具范围策略	更多工具往往意味着更差的性能 。 Vercel 从 v0 删掉 80% 工具，结果反而更好；Claude Code 用懒加载实现 95% 上下文削减。 原则：只暴露当前步骤所需的最小工具集
7	**Harness** 厚度	多少逻辑放在 harness、多少放在模型里。**Anthropic** 押注薄 **harness** 与模型进化；图框架押注显式控制。**Anthropic** 会随新模型版本，定期从 **Claude Code** 的 **harness** 里删掉规划步骤（因为模型自己内化了）

????????????

7.1 ?????????????????

- 建筑脚手架 ＝ 临时基础设施，让工人能去到原本到不了的位置
- 它不做施工本身；但没有它，工人上不到顶层
- 建筑完工时脚手架就被拆除

> 核心洞见：模型变强 → **harness** 复杂度应该下降。

Manus 6 个月内重写 5 次，每次重写都在删复杂度：

- 复杂的工具定义 → 通用 shell 执行
- "Management agents" → 简单的结构化 handoff

7.2 ??????

模型现在是带着特定 **harness** 一起做 **post-train** 的:

- Claude Code 的模型学会了用它训练时配套的那套 harness
- 改工具实现可能因为这种紧耦合而降低性能

7.3 "?????"??

> 如果换用更强的模型时，性能能 **scale up** 而不需要追加 **harness** 复杂度，这个设计就是好的。

??Harness ???

两个用相同模型的产品，可以仅因 harness 设计的差异而性能天差地别。TerminalBench 的证据是清晰的:只改 **harness**，**agent** 名次能跨越 **20+** 位。

Harness 不是已解决问题，也不是商品化层。这里才是真正的硬工程:

- 把上下文当稀缺资源管理
- 设计能在错误复利之前抓住失败的校验回路
- 构建能提供连续性又不幻觉的记忆系统
- 在 "多搭脚手架" 和 "留给模型自己" 之间做架构性押注

> 趋势是 **harness** 在变薄——但它不会消失。
> 即便最强的模型，也需要东西来管它的上下文窗口、执行它的工具、持久化它的状态、校验它的工作。

> 下一次你的 **agent** 出问题时，别怪模型，看 **harness**。

???????AllData????

对照仓库里现有 Agent 设计文档（`一种理想的智能体编排架构.md`、`基于关键词的知识树系统设计方案.md`）以及 WeaveAgent，建议的对齐方向：

Harness 组件	与本仓库的映射
Memory 三层结构	与 MindStore 的多层级摘要（pyramid + MarkdownLogic）高度同构，可以把"轻量索引 / 详情 / 原文"模式直接落到 MindStore 上
Context Management — Compaction & Tool Output Offloading	直接对应 `project_chronicle_grain.md`：WeaveAgent Chronicle 只记"原因 + 结果"，不复述工具返回——这就是 PDF 里说的 **tool output offloading**
Verification Loops（rules / visual / LLM-as-judge）	后续 Agent 输出质量评估的标准范式，Qwen3-Embedding 训练数据增强环节也可以接 LLM-as-judge
Long-Horizon Execution / Ralph Loop	Auto/ 流水线（PDF→MD→Logic→Embedding→MindStore）天然是长程任务，可以引入 progress 文件 + git checkpoint 的 Ralph Loop 模式
Harness 厚度原则	随着 Qwen3-Embedding checkpoint 持续训练增强，本地 Agent 的 harness 应有意识地变薄——删功能比加功能更重要
工具范围 —"少即是多"	Vercel 砍 80% 工具反而更好的经验，对 Auto/ 中各 `gen_*.py` 也成立：合并语义重复步骤优于堆功能

Revision #1
Created 2026-05-18 08:20:55 UTC by Colin
Updated 2026-05-18 08:20:55 UTC by Colin