从 Claude Code 看 Harness Engineering

Bojie Li

Chief Scientist, Pine AI

2026 年 4 月

只有上下文和工具是失控的天才，只有约束是安全的废物

开场：这场泄露真的是巧合吗？

2026/4/1，Claude Code 源码通过 npm 包泄露——源码里藏着一个完整的宠物扭蛋机

Buddy 系统：藏在 CLI 里的宠物

18 种物种 · 5 档稀有度（legendary 仅 1%）· 5 项随机属性 · 3 帧 ASCII 动画

四个耐人寻味的细节：

证据一：SALT = `'friend-2026-401'` ——friend + 泄露日期，精确到天

证据二：Teaser Window 精确到 April 1-7, 2026。注释写 `"Sustained Twitter buzz"` ——更像营销策划用语

证据三：18 个物种名全部 hex 编码，因为 `capybara` 碰撞了下一代模型的内部代号

证据四：统一 hex 编码 "so one doesn’t stand out"——反而让所有逆向工程者都去解码了

三种可能的解读

A. 纯巧合 (10%)：Buddy 是愚人节彩蛋，source map 配置失误，碰巧同一天

B. 技术团队"不小心" (55%)：有人在那次构建中"不小心"开启了 source map。法务发 DMCA 是真实的应激反应，但窗口期已足够代码传遍全球。Buddy 是**提前埋好的引爆物**

C. 其他可能 (35%)：完全意外但事后默许，或公司策划

不管答案是什么：全球开发者免费做了一次深度代码审查和口碑传播。这可能是 2026 年最成功的技术营销。

但真正的价值在于：51 万行 TypeScript 第一次让我们看到，一个日活百万的商业级 AI Agent 在工程层面到底在解决什么问题。

OpenClaw vs. Claude Code：广度与深度的两个极端

同一个模型，不同的 **Harness**，效果天差地别

两种路线对比

	OpenClaw	**Claude Code**
定位	通用 Agent 框架	Coding Agent
代码量	两个月几十万行	51 万行 TypeScript
功能广度	大量功能、将将能用	只做 coding，做到极致
同模型效果	基准	90%+ 场景更优
架构成熟度	概念验证级	生产级

方汉（昆仑万维创始人）测试：春节同一任务、同一模型，90%+ 情况 Claude Code 更好。方汉类比当年的中文 Linux——**Linus** 对社区的治理水平比 **OpenClaw** 创始人高很多。

OpenClaw 是否会成为操作系统？

OpenClaw 的贡献：重新定义了 Agent 的交互范式：

1. 更像和一个“人”持续沟通，不再有“**session**”的概念
2. 所有插件通过自然语言安装和交互，无需通过 GUI 安装和配置
3. 使用 **Skills + CLI**（命令行）取代 **MCP**（工具），实现 Agent 能力的轻松扩展，不懂技术的人也可以用自然语言编写 Skill

但架构层面缺乏深度，例如：

1. 缺少 **Harness** 深度：只有让模型能做事的上下文和工具，缺少让模型办事靠谱的错误恢复、安全机制，导致办事不靠谱
2. 原生记忆系统效果差：原生记忆机制过于简陋，需要搭配更好的第三方记忆系统
3. 浪费 **token**：KV Cache 不友好、上下文压缩机制简陋，导致 token 开销高
4. 与多人交互时混淆身份：与多人交互时分不清是用户说的，还是陌生人说的
5. 缺少异步通知：外部事件触发、通知系统没做成一等公民

Outline：本次分享的五个部分

一、Harness Engineering 是什么

范式演进：Prompt ⊂ Context ⊂ Harness；Agent = Model ✖ Harness

二、让 Agent 能干事（上下文 + 工具）

Prompt Cache 架构约束 · 五层上下文压缩 · Side Query 并行 · Markdown 记忆 + Dream

三、让 Agent 不出错（约束·验证·纠正）

Fail-closed + 五层权限 · 投机执行 · 错误恢复 + 熔断器 · 多 Agent 能力分区

四、用做研究的方法做产品

消融实验 + 双层 Feature Flag · 反蒸馏两层纵深防御

五、从 Claude Code 看 AI 与人的未来

GUI 的黄昏 · Context vs 渠道之争 · AI Native 组织 · 三种人才原型 · Context 是人的护城河 · 一人公司 OPC

结语：Model × Harness = Agent

基座模型公司的飞轮，与“模型即 Agent”是否成立

一、Harness Engineering 是什么

Agent 工程的下一个范式

范式演进：Prompt ⊂ Context ⊂ Harness

Agent 发展的三个阶段

阶段	关注点	说明
Prompt Engineering	问什么	优化输入给模型的指令
Context Engineering	看什么	系统性管理模型能看到的信息
Harness Engineering	整个系统	模型运行的全部基础设施

模型能力趋于商品化，**竞争优势正在转移到模型之外的工程实践**。

Harness 就是**模型之外的一切**：上下文怎么给、工具怎么调、出错怎么恢复、安全怎么保障、缓存怎么共享、并行怎么协调。

关键案例

LangChain Terminal Bench 2.0：不换模型，只改 Harness，准确率 **52.8% → 66.5%**，排行榜从 30 名外直接进入前 5

OpenAI 内部：3 名工程师 + Agent + 合理 Harness，**5** 个月完成约百万行代码、**1500** 个 **PR**

Claude Code：51 万行 TypeScript，其中绝大部分不是在做"让模型调工具"，而是在做工具调用之后的一切。**这就是 Harness Engineering 最好的实战样本**

Agent 的核心公式：Agent = Model × Harness

一个 Agent 要解决三件事

层面	取决于	类比
智力——能不能想明白	模型	大脑
能力——能不能干成事	上下文 + 工具	手脚
靠谱——会不会干错事	约束 + 验证 + 纠正	缰绳

Harness 包含两大部分

手脚：上下文、工具——让 Agent 能干事

缰绳：约束、验证、错误恢复——让 Agent 不出错

只有上下文和工具是失控的天才，只有约束是安全的废物。

Claude Code 中的 Harness 全貌

Environment：40 个内置工具、五层上下文压缩、Prompt Cache 经济学、CLAUDE.md 记忆体系、Dream 离线巩固、Side Query 并行调用、投机执行

约束：Fail-closed 默认值、工具白名单、Shell 语义解析、Undercover Mode

验证：LLM 权限分类器、Hook 系统、五层权限判断

纠正：熔断器、消息扣留、模型降级、死亡螺旋防护

二、让 Agent 能干事

Environment：Prompt Cache、上下文压缩、并行 LLM 调用、记忆系统

Prompt Cache：第一天就要考虑的架构约束，不是优化

如果只能从 **Claude Code** 源码中学一条原则，我选这条

缓存边界写进了系统提示的物理结构

```
[全局稳定内容 - 跨用户可缓存]
--- DYNAMIC BOUNDARY ---
⚠️ WARNING: Do not remove or reorder
   without updating cache logic
[会话特定内容 - 不缓存]
```

系统提示词的组织结构首先由缓存边界决定，其次才是语义逻辑。大部分人习惯按语义分段写 prompt，但在生产系统里，按缓存边界分段更重要。

Fork Agent 的缓存共享

每次 fork 传入 `CacheSafeParams` ——系统提示、用户上下文、工具列表、消息前缀、thinking 配置——必须和父 Agent 字节级一致才能命中同一份 Cache。整个代码库 9 个 fork 调用者都走这套。

每个架构决策都在为缓存让路

设计点	缓存考量
系统提示分段	全局/动态边界分离
工具定义位置	放在系统提示稳定缓存
Fork agent 参数	CacheSafeParams 字节级一致
工具结果截断	冻结替换字符串
Thinking config	maxOutputTokens 影响 cache key
消息序列化	确定性 JSON key 顺序

Cache 命中 vs 未命中 = **成本和延迟的量级差别**。它决定了消息怎么序列化、子 Agent 怎么分叉、工具结果怎么存储。**第一天就画缓存边界图**。

五层上下文压缩管线

不同类型的信息有完全不同的"保质期"，需要完全不同的处理方式

Layer 1 `Tool Result Budget`	巨量输出存磁盘，模型只看预览 替换决策冻结以保护缓存	轻
Layer 2 `HISTORY_SNIP`	最精细裁剪，纯噪声直接删掉 搜索返回 500 行但只用了 3 行，不值得摘要	轻
Layer 3 `Microcompact`	在 API 缓存层面做编辑（ `cache_edits` ） 本地消息不变，压缩完全在 API 层完成	中
Layer 4 `CONTEXT_COLLAPSE`	旧对话轮次归档为摘要 保留结构——哪一轮做了什么、结论是什么	重
Layer 5 `Autocompact`	最后兜底，先 session memory 再全量 熔断器：连续 3 次失败放弃（曾有会话失败 3272 次 / 全球浪费 ~250K 调用/天）	最重

核心原则：前面能搞定就不触发后面。L1-L3 是"不丢信息的压缩"——数据还在磁盘上，只是模型不看全文。L4-L5 是"有损压缩"——信息被摘要替代。大部分时候最重的 Autocompact 根本不需要跑。

Side Query：主 Agent 循环不应该是唯一调用 LLM 的地方

5+ 类并行辅助调用

调用场景	模型	说明
权限分类	小模型	判断工具调用是否安全
记忆检索	小模型	哪些 CLAUDE.md 与当前任务相关
Tool Use Summary	小模型	异步总结工具操作
Agent 摘要	小模型	子 Agent 进度报告
提示建议	小模型	预测用户下一步

Tool Use Summary 的精妙设计

主模型推理时（5-30s），Haiku 同时 总结上一轮的工具操作。1 秒内完成，延迟完全被藏在主模型推理时间里。后续压缩时用这个摘要替代原始输出，大幅节约 token。

关键架构观念

权限判断、摘要生成、记忆检索这些事情可以用更小更快的模型并行处理：

```
主模型推理 (5-30s)
  ├── Haiku 异步总结工具 → 省 token
  ├── 小模型权限分类 → 安全判断
  ├── 小模型记忆检索 → 相关 context
  └── 小模型提示建议 → 预测下一步
```

把"调用 LLM"当成可以到处撒的轻量操作，而不是只有在主循环里才能做的重量级事件。

主模型推理期间，五六个辅助任务同时在跑——权限分类、记忆检索、摘要生成、提示建议全部并行。这是 Claude Code 同等模型下比其他 Agent 快得多的关键。

记忆架构：为什么用 Markdown + 文件系统这种最原始的方法

向量数据库的根本问题：Top-K 检索不完整

一个简单的例子：系统见过 100 只猫，其中 **90 只黑猫 + 10 只白猫**。用向量数据库取 Top-10——

① **决策错误**：Top-10 很可能全是黑猫、一只白猫都没有。基于 Top-K 的样本做判断，结论必然偏颇

② **无法计数**：如果想统计"黑猫 vs 白猫"的比例，向量数据库根本做不到——它没有办法把原始 100 个例子全部匹配出来

不是说向量数据库不好，是说单靠它存原始数据不够。必须对原始数据做结构化整理、压缩、总结——这正是 Markdown 的作用。

为什么选 Markdown 而不是知识图谱

- **自然语言是 LLM 最擅长的格式**——Markdown = 结构化的自然语言
- **知识图谱**在专业领域更精确，但在**通用场景**下，自然语言的灵活性和覆盖面更强
- **可编辑可审计**：打开就能看 AI 记了什么，记错了直接改；Git 可追溯

Claude Code 的 Markdown 记忆体系

文件	用途
`CLAUDE.md`	项目级记忆与约定
`MEMORY.md`	核心事实与用户偏好
`memory/YYYY-MM-DD.md`	按日期归档的交互日志

Dream：睡眠学习机制

人类睡眠时大脑巩固记忆——Agent 空闲时巩固会话记忆：

- 扫描近期会话，提取值得长期保留的信号
- 合并到已有主题文件，删除被推翻的旧事实
- 修剪索引，保持精简

Dream 做的事，向量数据库做不了：它需要理解"这条信息是不是推翻了旧的"、"这 90 只黑猫能不能总结成一条规律"——这些都需要**自然语言层面的推理**，不是向量相似度能解决的。

The Bitter Lesson：最终胜出的方法一定是**能利用更多计算资源的通用方法**。Markdown + 自然语言组织 + 文件系统，再叠加 LLM 做压缩、整理，这是通用、可扩展的路径。

三、让 Agent 不出错

约束 · 验证 · 纠正：Fail-closed、错误恢复、熔断器

安全不是功能，是架构（对比：OpenClaw vs. Claude Code）

Fail-closed 的工具默认值

```
TOOL_DEFAULTS = {
  isConcurrencySafe: false,  // 假定不安全
  isReadOnly: false,         // 假定有写操作
  toAutoClassifierInput: "", // 不参与分类
}
```

每一个默认值都选择更保守的选项。忘了声明只读？当写操作处理。忘了声明并发安全？独占运行。**忘了声明比错误声明安全得多。**

LLM 权限分类器：只看 tool_use

分类器从对话历史中只提取 `tool_use` **block，不包含助手的自由文本。**原因：如果能看到助手的自然语言输出，攻击者可以通过让主模型输出特定文本来误导分类器。

只看结构化的工具调用，攻击面小得多。

五层权限判断

```
模型请求执行工具
    │
    ▼
① Settings 静态规则   ← alwaysDeny/Allow/Ask
    ▼
② PreToolUse Hook     ← 用户自定义脚本
    ▼
③ Tool 自身属性       ← isReadOnly 白名单
    ▼
④ LLM Auto-Classifier ← 只看 tool_use block
    ▼                        不看自然语言（防注入）
⑤ 拒绝熔断器          ← 连续≥3次 → 回退交互
```

设计哲学：每层只负责一个关注点。静态规则快速裁剪、Hook 支持企业定制、LLM 处理无法预定义的模糊场景、熔断器保底防卡死。

错误恢复：在确认无法恢复之前，不暴露中间态

输出触顶的两步恢复

第一步：**静默升级上限——**撞 `max_output_tokens` 时，如果是默认 8K，直接用 64K 重发。**对用户完全透明，只火一次。**

第二步：**多轮接续（最多 3 次）**——64K 还不够，才注入 meta 消息："Resume directly — no apology, no recap. Pick up mid-thought."

消息扣留（Message Withholding）

恢复期间，错误消息被**扣留**，不发给消费者（桌面端、IDE 插件）。因为它们看到 error 就会终止会话。

```
恢复成功 → 消费者永远不知道出过错
全部失败 → 释放扣留的错误
```

错误处理的边界不是"单次 API 请求"，而是整个恢复循环。

模型降级

主模型不可用时，自动剥离旧模型的签名块和 tool_use 格式，用备用模型重发。

死亡螺旋防护

API 出错 → 触发 stop hooks → hooks 也调 API → 也出错 → 又触发 hooks……

规则：API 错误时跳过所有 stop hooks。宁可丢失一次记忆提取，也不能让系统陷入无限循环。

熔断器无处不在

场景	阈值	真实数据
上下文压缩	连续 3 次	曾浪费 ~250K 调用/天
权限分类	连续 3 / 累计 20	防 Agent 被卡死
输出触顶	最多 3 次接续	先升级再续写

这些阈值不是拍脑袋定的——**3 次阈值来自 1279 个会话曾出现 50+ 次连续失败的真实产线数据。**

Agent 安全：限制工具集合、基于语义的命令行安全检查

不同的 Agent 有不同的工具集合

Agent 类型	工具面
主 Agent	完整工具集
子 Agent	禁止递归创建 Agent、退出计划模式
Coordinator	只有 3 个工具：创建/发消息/停止 worker
Async Agent	只有文件读写、搜索、shell
In-process Teammate	Async + 任务管理 + cron

Capability-based 的身份定义比 system prompt 角色提示词更硬、更可审计、更难绕过。

Shell 安全：语义解析 > 关键词黑名单

完整的命令语义解析器，每条 git 子命令的每个 flag 做类型化标注。

真实安全案例：

```
git diff -S -- --output=/tmp/pwned
```

- `-S` 之前被标记为 `none`（不带参数）
- 但 git 的实际行为是：`-S` 强制消费下一个 argv
- 攻击路径：验证器认为 `-S` 不带参数 → 推进 1 个 token → 遇到 `--` 停止检查 → `--output` 未检查 → 任意文件写入
- 修复：把 `-S` 改成 `string` 类型

普遍规律：安全机制的粒度应该和攻击面的粒度匹配。关键词黑名单粒度太粗，语义解析才能覆盖攻击面。

四、用做研究的方法做产品

消融实验 · Feature Flag · 反蒸馏

消融实验：把 ML 的研究方法搬到产品工程

ABLATION_BASELINE：一次性关掉 7 个功能

源码里有一个 `ABLATION_BASELINE` flag，注释标注为 "Harness-science L0 ablation baseline"。启用后一次性关掉：

- Thinking mode
- 上下文压缩
- 自动记忆
- 后台任务
- 简单模式
- 交织思考
- 第二个后台任务 flag

跑对照实验——关掉某个功能后，任务完成率、token 消耗、会话时长有没有显著变化。

做过 ML 研究的人都熟悉消融实验。**把这个方法论搬到产品工程上，在工业代码里是少见的。** 最接近的类比是字节跳动——几乎所有产品决策都经过 A/B 验证。

双层 Feature Flag

层级	机制	用途
编译时	Bun `feature()`	false 分支从二进制物理删除——安全研究员反编译也找不到
运行时	GrowthBook	灰度发布 / A/B / kill switch 三合一

GrowthBook 实验工程

- **服务端分桶**（`remoteEval: true`，客户端拿计算好的结果）
- **用户画像定向**（按 deviceId / organizationUUID / subscriptionType / platform / email）
- **曝光追踪**（每个 feature 的 experimentId + variationId 记到事件系统）

核心理念：不是“感觉有用就上”，是“数据证明有用才上”。压缩熔断器的 3 次阈值、autocompact 的触发条件，全部有真实生产数据支撑。

反蒸馏：两层纵深防御

源码中独立的工程子系统——阻止竞争对手通过 API 输出训练自己的模型

Layer 1：Fake Tools 训练集投毒

```
// Anti-distillation: send fake_tools
// opt-in for 1P CLI only
anti_distillation: ['fake_tools']
```

API 后端在响应中注入虚假工具调用，混在真实输出中。批量抓取训练数据的系统会一并吃进去——相当于在训练集中投毒。

攻击者两难：要么花成本过滤（但很难区分真假工具调用），要么接受训练数据被污染。

Layer 2：Connector Text Summarization

服务端把模型在工具调用之间生成的自由文本（推理过程）缓存起来，替换为摘要 + 加密签名。客户端发回签名摘要，服务端凭签名还原原文。

外部观察者只能看到摘要——模型的推理链条被签名遮蔽，只有 Anthropic 服务端能解密。

两层互补的设计逻辑

Layer 1 针对"大网捞鱼"式的批量抓取——成本低、覆盖广，靠噪声比让训练数据不可用

Layer 2 针对精细逆向工程——即使攻击者过滤掉虚假工具调用，模型推理过程仍被签名遮蔽

反蒸馏的存在本身就证明模型能力还没真正商品化。如果模型已经可替换，为什么要大力防蒸馏？

五、从 Claude Code 看 AI 与人的未来

软件形态 · 商业护城河 · 组织变革 · 个人分化

GUI 的黄昏？软件的价值正在从界面转向数据治理

GUI 是给人有限注意力打的补丁

GUI 存在的前提是**人类注意力稀缺**——屏幕、鼠标、键盘都是为了在有限带宽下让人理解和操作复杂系统。

- **Agent 阅读（prefill）的速度是人类的数百倍**（Agent 每秒可阅读 10K - 100K token，人类只能阅读 10 - 100 token）
- **Agent 思考（decode）的速度是人类的数十倍**（Agent 每秒可输出 50 - 200 token，人类只能思考 5 - 10 token）

但 **Agent 操作 GUI 的速度比人类还慢**，因此 GUI 对 Agent 是不友好的。

Agent 时代，软件的价值将主要集中在三个地方：

- **数据壁垒**：独家数据源和长期积累的业务知识
- **业务逻辑**：领域规则、流程、约束的准确表达（未必是代码形式，可能是 Skill 形式）
- **权限与数据治理**：谁能读、谁能写、什么时候能改

SaaS 分化：没有数据壁垒的 SaaS 大概率被干掉

Claude Code 印证了这一点

51 万行 TypeScript 里：

- **没有一个产品级 GUI**——整个交互是终端 + 文件
- **核心价值全在结构化能力**：Harness、工具系统、权限模型、缓存经济学
- **和用户最显性的接口是** `CLAUDE.md` ——一个 Markdown 文件

Claude Code 就是典型的"**GUI 价值低、业务逻辑与数据治理价值高**"的产品形态。

> **另一种声音**
>
> 昆仑万维创始人方汉：大多数用户还是想看图形界面而不是文字。他提出 **intentware** 概念——software 从“功能系统”变成“意图系统”，只要有想法就能直接转化为执行能力。

Context 是人避免被 AI 取代的护城河

来自 OpenAI 工程师 Jiayi Weng 的洞察

> 我觉得自己在 OpenAI 的工作也没有那么难，并不需要很高的智商。如果换一个人，有我所有的 **context**，也是能干的。

Context 对人和模型都是最重要的：

- 团队合作最大的问题是 **context 的不一致**——一个人写的代码，另一个人接手时 context 不一样，就会出问题
- AI 短期内无法取代人的最大原因也是 **context**——AI 跟人不在同一个环境里，能访问的 context 远低于一个人类员工

为什么能力如此强的 AI 无法直接决策

根本原因：**人和 AI 的 Context 不一样**

- **需求背后的隐性约束**——饭桌上、走廊里聊出来的设计目的，AI 无从获取
- **屎山代码里的坑**——遗留系统看似不合理的设计其实有历史原因
- **未曾表达的内心想法**——就算把人一生的交互全录下来，AI 也没有读心术，不知道人现在正在想什么
    - 《黑镜》第二季《马上回来》（Be Right Back）：**数字分身的局限性**（这里就不剧透了，建议做数字分身的一定要仔细看看）

为什么人还需要学习底层原理

- 理解底层原理也让人知道**哪些 context 对设计取舍最关键**，从而告诉 AI 合适的 context——**这就是人用好 AI 的关键**
- 理解底层原理让人能利用比 AI 多出来的那些 context，做出更全面的设计取舍

Context 是人避免被 AI 取代最关键的护城河——其实 Context 也是**技术专家避免被年轻人取代的护城河**。

另一种声音

某些 Context 会被模型内化到权重中，另外一部分 Context 也会被蒸馏到 Skill 中。真正不可替代的是**未被记录的、与特定场景/人的信任绑定的 context**。

AI Native 组织：用 AI 替代传统的上传下达

Brooks 的《人月神话》有了新答案

Brooks 指出：**沟通开销是软件项目最大的敌人**——给延期项目加人只会更慢。他的解法是"概念完整性"——系统应反映单一架构师的统一心智模型。

软件工程历史上最好的软件，在 MVP 阶段基本都是单个技术专家亲自动手实现的（UNIX、C、Linux、Git 等），“委员会设计”产出的软件几乎一定是平庸的。

AI 可以更好地实现概念完整性——不是因为协调变好了，而是**协调变得不必要了**：

指标	单人 + AI 数据
日均代码产出	**4-5 万行**
日均 Token 消耗	**18 亿**
单日最高 Git Commit	**1,374 次**

AI Native 公司的共同特征

- **扁平结构**：Anthropic、Kimi 等公司的中层大幅压缩，上传下达被 AI 替代
- **Context 共享**：Jiayi Weng（OpenAI）——"人类组织的千古难题：难以保持组织架构的 context sharing 一致性"。

“砍掉高层的手脚，砍掉中层的屁股，砍掉基层的脑袋”

这句段子在现在的 AI Native 公司不再成立了。

传统中层的核心职能是上传下达，**这三件事 AI 都能做，且比人做得好**——因为 AI 不会为了公司政治扭曲信息

所以 AI Native 组织不是砍掉一半人，而是把中层这个信息上传下达功能交给 AI，让高层直接看到基层信号、让基层直接看到战略 context。

另一种声音：给传统公司的务实建议

昆仑万维创始人方汉：很多国内公司还没完成信息化——**传统公司不要当先烈，也不要成为后进生**。

- 从上到下考 AI（笔试+机试），让组织人知道 AI 的边界在哪
- 已成熟的（chatbot、AI coding）全力推进，其他场景保持克制

什么样的人会被 AI 替代：头尾安全、腰部危险

软件行业从劳动密集型 → 极端杠杆与认知带宽驱动

当 AI 处理大量执行层任务后，人类开发者将分化为三种专业化原型：

🎬 **电影导演型（Film Directors）— 从 0 到 1 创造者**

定义产品愿景和新功能的"感觉"。端到端：定义体验 → 写 prompt → 构建集成 → 设计 UI → 发布。瓶颈：创造性判断力。

🌆 **城市规划师型（Urban Planners）— 从 1 到 100 架构师**

管理系统规模化后的复杂性。核心基础设施、服务契约、评估框架、可观测性。瓶颈：架构判断力。

🏎️ **F1 赛车手型（F1 Racers）— 极限推进者**

在基础模型的数学和实验前沿战斗。训练、新架构、基准性能。瓶颈：科学洞察力。

更重要的是泛化能力——学习新知识、适应新场景的能力，将成为最关键的能力。

一人公司 OPC 并不是新鲜事

在 Agent 出现之前，“一人公司”就长期存在：

- 张雪峰
- 独立量化交易者
- 独立开发者

AI 是能力的放大器——但它放大的主要是技术能力，非技术能力的相对比重反而在上升

最常见的误解：“一个人能开发 App = 一人公司”

- 大多数独立开发者的瓶颈从来不是“写不出代码”，而是“没人用”
- AI 让更多人跨过技术门槛，但更拥挤的市场里，获客、信任、运营成为更稀缺的能力

Model × Harness = Agent

“模型即 Agent” 到底对不对?

一个常见的误读

一些基座模型公司在推“模型即 Agent”的概念，很多人把它理解成：

> 模型 + 几个简单工具 = **Agent**

但 Claude Code 的 51 万行源码告诉我们——**远远不够**。

真正的 Agent 里有一堆复杂的 Harness：五层压缩、Prompt Cache 经济学、Side Query 并行、投机执行、熔断器、错误恢复……**Harness 的代码量远超工具+提示词本身**。

真实含义：Model × Harness 的协同优化飞轮

模型公司说的不是“光靠模型就行”，而是：**只有模型公司手里握着 Model × Harness 的协同优化飞轮**。

飞轮是一条三层接力

1. 用户 → 应用层：提真实业务场景的 bug 和需求
2. 应用层 → 模型：沉淀成下一代训练信号
3. 模型搞不定的部分 ← **Harness** 兜底：应用层用外部逻辑把模型当前做不好的事补上

应用层既是**模型能力的压力测试场**，也是**兜底工程的承载者**——只有模型公司同时控制两端。

应用层公司的护城河需要在技术之外

观察一：Harness 里的“屎山”是飞轮的化石

为什么模型公司不把全部 bug 都喂给下一代模型，让 Harness 变薄？因为训练不是阿拉丁神灯：

1. 训练需要时间（数月）
2. 模型不能内化训练数据中所有的约束和偏好

所以 Harness 里的“屎山”其实反映了模型公司内部的模型团队和应用团队之间的张力：

- 模型稳定掌握的 → 下一代 Harness 可以**去掉**
- 模型还不稳的 → Harness 里**继续兜底**

Claude Code 泄露源码里布满注释，记录着真实业务场景里模型没扛住、Harness 补上的一次次事件，是**模型当前能力边界的化石**。

观察二：模型会商品化吗?

顶尖模型：没有商品化，差距还会拉大

- 硅谷 AI 人才薪酬 $1.5M-$10M+，**是同级非 AI 工程师的 3-4 倍**
- 反蒸馏工程的存在 = 顶尖模型的推理轨迹还有独占价值
- **人才溢价 + 算力壁垒 + 数据飞轮——差距可能继续扩大而非收敛**

中端模型（普通人类智力，日常工作）：正在商品化

- 开源和闭源前沿差距已在一年以内（Kimi K2.6 编码接近 Claude 4.6 Sonnet）
- 目前前沿模型的智力已经能胜任大多数人的日常工作，只缺 Context 和 Harness
- 同等智力的模型，API 价格一年下降超过一个数量级

关键启示：Harness 是应用层短期的技术杠杆，但**技术优势最终会被模型公司的飞轮吃掉**。应用层公司的长期护城河需要在技术之外：数据、渠道、牌照、用户信任、网络效应等等。

谢谢

只有上下文和工具是失控的天才，只有约束是安全的废物

从 Demo 到产品的真正距离，在模型之外的 Harness

Bojie Li

Chief Scientist, Pine AI

博客：01.me