从Harness Engineering，再到 Trellis 落地

一、 Harness 开源方案对比

1. 三种类型

纯配置方案：最轻量的起点是直接编写 CLAUDE.md / AGENTS.md，配合 superpowers 等 skill 集合。这一方案的核心是把配置层和约束层的实践手动落地。

框架增强方案：openspec 和 ccg-workflow 等框架在配置层之上增加了 spec 系统和 hook 自动注入能力。以 Trellis 为例，它通过三个 hook 时机（session-start、inject-subagent-context、ralph loop）确保 spec 文档在每次对话中都被使用，而不是写完就被遗忘。Trellis 选择将沉淀层系统化：spec 通过 hook 自动注入到每次交互中的强制上下文。

多角色编排方案：oh-my-claudecode、oh-my-opencode 等套件定义了多种 agent 角色（planner、executor、reviewer、debugger 等），实现了角色分工和多 agent 协作。

值得注意的是，Harness 说到底是在模型智能的基础上构建系统，随着模型基础能力的提升，Harness 的部分组件应该能够简化甚至移除。

Anthropic 的研究对此的原话是：

> every component in a harness encodes an assumption about what the model can’ t do on its own, and those assumptions are worth stress testing, both because they may be incorrect, and because they can quickly go stale as models improve.
>
> 每个 harness 组件都包含了关于模型无法独立完成的假设，而这些假设是值得进行压力测试的，原因在于它们可能是错误的，同时随着模型的改进，它们也可能很快过时。

一个可供参考的判断方法：**如果在项目中花了大量时间在维护 Harness 本身（编辑计划文件、更新进度文档、协调 Agent 之间的通信）而实际代码产出很少，说明 Harness 已经变成了负担而非助力。**

2. 每种类型的对比

2.1 Superpowers：工程流程强化型

Superpowers 的核心不只是“给 Agent 加技能”，而是把 Agent 的开发过程方法论化。它官方将自己定义为 **agentic skills framework & software development methodology**，强调通过可组合

skills 和初始指令，把 brainstorming、plan、TDD、review 等流程串成一套较强约束的开发工作方式。它更关注的是：**Agent 应该如何按工程纪律做事**，本质上是在强化执行流程与开发规范。

特点：

- 重点是规范 Agent 的**做事方式**
- 强调流程、纪律、方法论
- 更像是在给 Agent 增加一层“工程流程操作系统”

优点

- “头脑风暴” 对于完善想法和探索方案非常有用，能帮忙找出很多不成熟的问题
- worktree独立工作区，减少 AI 误操作导致不可恢复的后果
- 合并了 “探索、提需求草案、设计实现方案、具化 todo 列表”，直接给到方案，简单方便，没有基础的小白也能快速vibe出原型

缺点

- 哪怕是小任务，也会问很多问题，不够快捷，在 CC/codex 中会不停的去确认思考，方案设计
- 项目框架特别重，每次使用都会注入大量上下文，什么功能都要拆TDD，而且有个大问题是在5.4xhigh的情况下，容易上来就把上下文跑空，然后自动压缩，多来几次之后一些细节全没了，看着好像很好。结果一个下午过去了，终于把过去一小时就搞好的东西搞出来了，一看细节，坏了全错了
- 没法解决项目内各种特化的问题，没有持久化记忆，需要重复说明一些习惯问题，经常不遵守

2.2 OpenSpec：规格沉淀强化型

OpenSpec 的核心不是强化某一种编码流程，而是给 AI 开发增加一层 **lightweight spec layer**。它强调“先对齐，再实现”，并将每次变更组织为 proposal、spec、design、tasks 等结构化产物，且每个 改动 都有独立 文件夹，方便后续追踪、迭代和归档。它更关注的是：**开发依据是否被清晰表达并持续沉淀下来**，本质上是在强化需求、设计与任务这些规格资产。

特点：

- 重点是规范开发的**依据和产物**
- 强调 spec、design、task 的结构化沉淀
- 更像是在给 AI 开发增加一层“规格管理系统”

优点

- 把项目规范写成结构化文档，让 AI 在编码前先读规范。
- 很多原本只存在于老成员经验里的约定，可以被整理成显式规则，例如目录结构、分层边界、命名方式、错误处理原则。

- 更容易让 AI 输出接近团队既有风格的代码。

缺点

- 文档写出来之后，AI 是否每次都读、是否读对、是否真正遵守，仍然不一定。
- 对于任务拆解、执行控制之类的效果不是很好，并且上下文长了之后，容易被遗忘，导致最后效果
- 只关心文档而不太关心开发过程，缺少类似于头脑风暴这类能帮忙完善需求plan的skill

2.3 oh-my-claude（oh-my-claudecode）：多 Agent 编排型

oh-my-claudecode 官方定位是 **Multi-agent orchestration for Claude Code**，主打 teams-first、zero learning curve。它的重点不在规格沉淀，也不主要在流程方法论，而是在 Claude Code 之上封装一层多 Agent 协作机制，让不同角色或不同 agent 分工处理复杂任务。它更关注的是：**复杂任务能否通过角色分工和编排提升执行效率**。

特点：

- 重点是强化 Agent 之间的**协作方式**
- 强调角色分工、任务委派、自动化编排
- 更像是在给 Claude Code 增加一层“团队协作系统”

优点

- 通过多 Agent 分工和阶段式流水线，更适合复杂任务拆解与并行推进。
- 提供 `/autopilot`、`/team`、magic keywords 等入口，上手成本相对较低。
- 支持 Claude、Codex、Gemini 等多模型协同，适合做交叉评审和补充分析。

缺点

- 编排层本身比较重。官方参考文档里已经包含 **29 个 agents、35 个 skills、20 个 hooks、11 个生命周期事件**，能力很强，但也意味着系统复杂度更高，出了问题时排查成本会明显上升。
- 对 Claude Code Plugin 体系依赖较强，接入更深，也更有侵入性。
- 更适合复杂任务，对简单任务可能存在一定“过度编排”。

二、Trellis 框架介绍

> **项目地址**：https://github.com/mindfold-ai/Trellis/blob/main/README_CN.md

1. 为什么我们需要 Trellis?

在日常开发中，我们广泛使用 Claude Code、Cursor 等 AI 编程助手，但也随之面临以下显著痛点：

1. **“会话失忆”（Session Death）**：由于上下文窗口限制，每次开启新对话时，AI 就会遗忘之前的进度和项目背景，开发者需要反复重新解释。
2. **规则文件逐渐失控**：如果将所有代码规范、架构风格塞进单个 `.cursorrules` 或 `CLAUDE.md` 文件，会导致其变得臃肿无比，AI 容易出现 “上下文过载” 并忽略关键细节。
3. **多任务并行冲突**：在同一分支或环境下让 AI 同时处理多个任务时，AI 极其容易相互覆盖代码或造成 Git 分支冲突。

Trellis 是一个高级 AI Agent 工作流与编排框架（Agent Harness）。它的核心理念是：**“教 AI 认识你的项目，只需一次”**。它通过分层的规范体系、任务上下文隔离以及工作区持久记忆，为所有 AI 工具提供了一个统一的 “外接大脑”。

2. 核心概念

要熟练使用 Trellis，我们必须准确理解它定义的几个核心名词，它们与传统开发工具中的概念有本质区别。

2.1 Spec（规范 / 规约）

定义：在 Trellis 中，Spec（存放在 `.trellis/spec/`）指的是 “写给 AI 助手的项目规约与上下文记忆”。它是一系列 Markdown 文件，涵盖了我们团队的编码标准、文件结构规则、错误处理风格以及 Code Review 习惯

作用机制 (按需注入、渐进式披露)：当你让 AI 开发新功能时，Trellis 不会把所有文件丢给 AI，而是像 “人类开发前回忆规范” 一样，将当前任务强相关的 Spec 提取出来，在 AI 开始写代码前精准注入到 Prompt 中

动态演进：它不是静态的。在一次需求开发完毕后，Trellis 会有专门的复盘流程去检查本次编码是否产生了新的最佳实践，并自动更新 Spec 文档 (claude code 专属，其他工具需要手动触发)，真正做到让 AI “越用越顺手”

2.2 Task（任务）与生命周期状态

在 Trellis 中，Task 是一个**具有物理目录、独立记忆和层级关系的开发容器**（存放在 `.trellis/tasks/` 下），由包含元数据的 `task.json` 和相关需求文档组成

它支持父子任务拆解（Subtasks / Parent-Children 树），并且在 Trellis 的底层设计中拥有严密的**生命周期状态 (States)**。熟悉这些状态有助于你利用 Trellis 的自动化钩子 (Hooks) 联动其他工具

- Created (已创建)
  - **含义**：你刚刚通过指令创建了任务及其子任务，但尚未开始编写代码。
  - **行为**：生成 `task.json`，可触发 `after_create` 钩子。
- Started / In Progress (进行中)
  - **含义**：AI 被唤醒并投入该任务，配合 Git Worktrees 与 Work Journals (日志) 进行连续不断的编码与调试。

- **行为**：系统会持续追踪任务进度，并可触发 `after_start` 钩子。此时该任务上下文被标记为活跃 (Active)。

- Finished (已完成)
  - **含义**：功能开发和 AI 自测结束。
  - **行为**：触发 Code Review 和 `after_finish` 钩子。Trellis 会引导 AI 对照 Spec 中的标准检查代码，确保没有引入不符合团队规范的代码异味（Vibe coding 的通病）。
- Archived (已归档)
  - **含义**：工作彻底结束或需要封存。你通过 `/record-session` 等配套指令让 AI 记录最终的成果总结，并结束会话。
  - **行为**：任务被移出当前的高优先记忆区，防止污染未来新任务的上下文，并触发 `after_archive` 钩子。

2.3 json/jsonl 配置文件

描述 task 信息的元数据结构

task.json Schema

代码块

```
{
        "id": "02-27-user-login",
        "name": "user-login",
        "title": "添加用户登录",
        "description": "实现 JWT 登录流程",
        "status": "in_progress",
        "dev_type": "backend",
        "scope": "auth",
        "priority": "P1",
        "creator": "alice",
        "assignee": "alice",
        "createdAt": "2026-03-27T10:00:00Z",
        "completedAt": null,
        "branch": "feature/user-login",
        "base_branch": "main",
        "worktree_path": null,
        "current_phase": 1,
        "next_action": [{
                "phase": 1,
                "action": "implement"
        }, {
                "phase": 2,
                "action": "check"
        }, {
```

```
                "phase": 3,
                "action": "finish"
        }, {
                "phase": 4,
                "action": "create-pr"
        }],
        "commit": null,
        "pr_url": null,
        "subtasks": [],
        "relatedFiles": [],
        "notes": ""
}
```

JSONL（JSON Lines）文件定义了每个 Agent 需要读取哪些文件。每行是一个 JSON 对象：

字段说明：

字段	必填	说明
file	是	文件或目录的相对路径（相对于项目根目录）
reason	是	为什么需要这个文件（同时用于生成 completion markers）
type	否	默认 "file"。设为 "directory" 则读取目录下所有 .md 文件（最多 20 个）

三种 JSONL 文件：

文件	用于	典型内容
implement.jsonl	Implement Agent	workflow.md + 相关 spec + 代码模式示例
check.jsonl	Check Agent	finish-work.md + check 命令 + 相关 spec
debug.jsonl	Debug Agent	相关 spec + check 命令

implement.jsonl

代码块

```
{
        "file": ".trellis/workflow.md",
        "reason": "Project workflow and conventions"
} {
        "file": ".trellis/spec/shared/index.md",
        "reason": "Shared coding standards"
```

```
  } {
        "file": ".trellis/spec/backend/index.md",
        "reason": "Backend development guide"
  } {
        "file": ".trellis/spec/backend/api-module.md",
        "reason": "API module conventions"
  } {
        "file": ".trellis/spec/backend/quality.md",
        "reason": "Code quality requirements"
  }
```

check.jsonl

代码块

```
  {
        "file": ".claude/commands/trellis/finish-work.md",
        "reason": "Finish work checklist"
  } {
        "file": ".trellis/spec/shared/index.md",
        "reason": "Shared coding standards"
  } {
        "file": ".claude/commands/trellis/check-backend.md",
        "reason": "Backend check spec"
  }
```

3. 核心架构与目录解析

Trellis 的核心结构集中在项目根目录的 `.trellis/` 文件夹中。它取代了传统的单体规则文件，将 AI 的认知环境科学地拆分为三个维度：

- **`.trellis/spec/` (规范层)**
  用于存放 Markdown 格式的编码标准、目录结构规则、评审习惯等。Trellis 的机制是**按需加载**，它只会向 AI 注入与当前任务相关的规范，而不是一次性塞入所有内容。
- **`.trellis/tasks/` (任务层)**
  存放结构化的 PRD（产品需求）、实现上下文以及任务当前状态。这能确保 AI 的产出始终紧扣业务目标，不偏离任务轨道。
- **`.trellis/workspace/` (记忆与工作区)**
  这是解决 “会话失忆” 的核心。这里存放**工作日志 (Journals)**。它会记录上一次会话执行了什么、发现了什么 bug、下一步要做什么。每次新会话开始时，AI 会先读取工作区日志，无缝接续上次的思考逻辑。

目录结构全貌

```
代码块trellis/spec/
├── frontend/ # 前端规范
│ ├── index.md # 索引：列出所有规范及状态
│ ├── component-guidelines.md # 组件规范
│ ├── hook-guidelines.md # Hook 规范
│ ├── state-management.md # 状态管理
│ ├── type-safety.md # 类型安全
│ ├── quality-guidelines.md # 质量指南
│ └── directory-structure.md # 目录结构
├── backend/ # 后端规范
│ ├── index.md # 索引：列出所有规范及状态
│ ├── database-guidelines.md # 数据库规范
│ ├── error-handling.md # 错误处理
│ ├── logging-guidelines.md # 日志规范
│ ├── quality-guidelines.md # 安全质量规范
│ └── directory-structure.md # 目录结构
│
└── guides/ # 思维指南
├── index.md
├── cross-layer-thinking-guide.md # 跨层思维指南
└── code-reuse-thinking-guide.md # 代码复用指南
```

4. 团队接入与配置指南

4.1 编写规范 (Specs)

最佳实践：不要人工从零手写所有规范文件！ 在初始化 Trellis 后，会自带一个初始任务：`00-bootstrap-guidelines`，专门用来初始化规范，所以更高效的做法是：利用 AI 读取现有的优质代码模块，让 AI 自动起草规范草案，然后再由资深开发人员使用 /update-spec 命令微调最核心的架构规则（如数据库约定、组件复用逻辑等），保存至 `.trellis/spec/`。

4.2 规范沉淀即资产

如果某个团队成员在开发中摸索出了一套绝佳的工作流，或解决了一个容易让 AI 踩坑的顽疾，只需将这段经验写入 Spec 文件并提交到代码库。团队中其他人的 AI 助手拉取代码后，就能立刻 “学会” 这个新技能。

可以使用 /break-loop 命令触发 Bug 分析流程，AI 会深度分析这个 Bug 出现的原因，最后更新到 spec 约束中，防止下次再犯。

4.3 多开发者协作

> 为了更好的多人协作，.trellis 文件夹需要纳入 git 管理

不冲突的部分（各人独立）：

- `workspace/{name}/` — 每人独立目录

- `.developer` — gitignored，每人独立
- 并行任务的 worktree — 物理隔离

可能冲突的部分（需要协调）：

- `spec/` — 多人可能同时修改同一规范
- `tasks/` — 多人可能操作同一任务

最佳实践：

- Spec 修改通过 PR 审查
- 重要的 spec 变更在团队会议上讨论

1. 使用不支持 Hooks 系统的 AI 编程助手时（非 Claude Code），需要在会话创建后运行 `/start` 注入 Trellis 上下文；在开始写前端 / 后端代码之前，需要运行 `/before-frontend-dev` 或 `/before-backend-dev`，指示 AI 读取 `.trellis/spec/` 下的代码规范。
2. 在代码完成开发后，使用 `/check-frontend` 与 `/check-backend` 检查写的代码是否符合 `.trellis/spec` 下的代码规范
3. 在窗口上下文即将用满时，运行 `/finish-work` 与 `/record-session` 获取当前上下文（分支、任务等），AI 会调用 `add-session.sh` 把会话标题、commit hash、摘要写入 `.trellis/workspace/<developer>/journal-N.md`
4. Task 相关的一切调用都依赖于自然语言（创建 task，更新 task 状态，归档 task，将 task 分配给 `developer-name` 等）。prd.md 与 json 文件在 `/start` 命令成功注入上下文的情况下会自动创建

5. 工作流程

以实际场景为例讲清楚如何使用 Trellis：

项目背景

Spec0：初始化 Trellis（仅初始化一次即可）

初始化命令根据你使用的平台选择：

- Claude Code：`trellis init -u your-name`
- Cursor：`trellis init -u your-name`
- Codex：`trellis init --codex -u your-name`
- OpenCode：`trellis init --opencode -u your-name`

`your-name` 会成为你的开发者身份标识，并创建个人工作区 `.trellis/workspace/your-name/`。

`/trellis-onboard` 也可以使用这个命令

Spec1：设定 “项目规范”

在开发开始前，`.trellis/spec/` 中已经沉淀了 sm-go的核心开发规范。这让 AI 懂得如何写出符合项目规范的代码。

- `index.md` — 规范总索引，按任务类型指向对应文档
- `directory-structure.md` — 模块组织、服务目录结构与命名规范
- `database-guidelines.md` — DAO 模式、分片、缓存、事务、SQL 安全写法
- `error-handling.md` — 错误码定义、响应格式、错误传播链路
- `logging-guidelines.md` — `slog`（新）vs `tlog`（废弃）及日志级别使用
- `grpc-guidelines.md` — gRPC 服务端/客户端、etcd 服务发现、拦截器
- `middleware-routing-guidelines.md` — Gin 中间件链、路由分组、限流、认证
- `config-guidelines.md` — TOML 静态配置与 Apollo 动态配置管理
- `message-queue-guidelines.md` — Kafka / Pulsar / Beanstalkd 选型与使用
- `quality-guidelines.md` — Lint、测试、import 分组、Code Review 规范

效果：当你对 AI 说 “帮我写个读取xxx数据的接口” 时，你不必每次都重复 “要用slog打日志，数据库遍历游标获取数据不要执行任何耗时操作”，Trellis 会自动将这些 Spec 注入 AI 的上下文。

Spec2：拆解并启动任务

现在我们要正式做功能了。

Prompt 示例

```
代码块
1  现在需要改动xxx功能，需求是这样的：xxxx，这是相关的文件 @文件1 @文件2 创建trellis任务，同时补充jsonl配置文件，视情况拆分为subtasks
```

Trellis 动作：

- 自动创建 `.trellis/tasks/任务名/task.json`、`.trellis/tasks/任务名/prd.md`、`.trellis/tasks/任务名/check.jsonl`、`.trellis/tasks/任务名debug.jsonl`、`.trellis/tasks/任务名/implement.jsonl`。

> 如果需求不明确，可以使用 /brainstorm 命令进行头脑风暴

Spec3：遵循规范的编码

在任务细节确定好后，可以新开一个会话重置上下文，**执行 /start 进行初始化（claude code不需要）**，选择要执行的命令

Prompt 示例：

```
代码块
1  现在开始执行xxx任务
```

AI 在写代码之前，会调用 skill 获取对应的 spec，然后才开始写代码。写完代码后，会再次调用工具验证代码是否符合规范。

> 能否在正确的时机调用 skill 取决于你使用的模型，如果模型不够智能，可以在开始写前端 / 后端代码之前，手动运行 `/before-frontend-dev` 或 `/before-backend-dev`，指示 AI 读取 `.trellis/spec/` 下的代码规范。

Spec4：代码检查

在代码完成开发后，使用 `/check-frontend` 与 `/check-backend` 检查写的代码是否符合 `.trellis/spec` 下的代码规范

> 使用比较聪明的模型时，他们在写代码时就会执行自检

Spec5：会话中断与持久化

场景：你写完 Service 层代码后，由于当前窗口上下文达到上限，或者你今天下班了，必须关闭当前对话窗口。

用户 Prompt（断开前）：

```
代码块
1  “我要结束当前会话了。请使用 Trellis 记录当前进度到 Journal。
2  记录重点：Service 层逻辑已写完，但 validate() 方法中关于‘工单状态检查’的逻辑还没通过单元测试，明天需要先修这个。”
```

Trellis 动作：

- 在 `.trellis/workspace/journals/` 下生成一份带时间戳的日志。

Spec6：恢复上下文

场景：第二天，你开启了一个全新的空白对话窗口。

用户 Prompt（开始时）：

```
代码块
1  “加载任务 feat-quality-inspection。读一下昨天的 Journal，告诉我接下来该做什么。”
```

AI 会读取之前的记忆和任务进度，继续执行。

三、总结

从 Superpowers 的流程强化、OpenSpec 的规格沉淀，到 oh-my-claudecode 的多 Agent 编排，再到 Trellis 的全链路落地，各方案侧重不同，没有绝对优劣，选择取决于团队当前最迫切的痛点。

目前 AI Coding 整体仍处于探索阶段，业界尚未形成统一范式——模型能力在快速演进，工具链在持续迭代，今天的最佳实践明天可能就会被更简单的方式替代。

但有一件事不会随范式变化而过时：**从现在开始积累属于自己业务线的数据资产**。无论是代码规范、架构决策、任务拆解记录，还是 AI 与人协作过程中沉淀下来的 spec 文档，这些资产都是迁移到任何新范式时最快的起跑线。模型会换，工具会换，但对业务上下文的深度理解是无法被替代的护城河。