Trellis

面向 Claude Code、Cursor、iFlow 和 Codex 的
一体化 AI 框架与工具集

全面技术报告

版本 0.3.0-rc.2
由 Mindfold LLC 发布
分析日期：2026 年 2 月 17 日

包名：`\@mindfoldhq/trellis`
仓库：`github.com/mindfold-ai/Trellis`
许可证：AGPL-3.0-only

目录

1. 执行摘要

Trellis 是由 Mindfold LLC 开发的开源 AI 辅助开发工作流框架。它为跨多个平台（具体包括 Claude Code、Cursor、iFlow CLI、OpenCode 和 Codex）管理 AI 编程智能体提供了结构化方法，通过精密的基于钩子的注入系统确保一致的代码质量、持久的会话记忆和强制执行的开发标准。

1.1. 问题陈述

现代 AI 编程助手存在若干根本性的局限：

1. **上下文遗忘**：AI 在会话之间丢失上下文，要求开发者反复解释项目规范。
2. **规范偏移**：写在 `CLAUDE.md`、`.cursorrules` 或 `AGENTS.md` 中的指令经常在对话中途被忽略或遗忘。
3. **质量不一致**：在缺乏强制标准的情况下，随着对话变长，AI 生成代码的质量会逐渐下降。
4. **并行隔离**：在同一代码库上同时运行多个 AI 智能体会导致冲突和资源浪费。
5. **知识孤岛**：团队成员的最佳实践停留在个人脑中，未能被编纂和共享。

1.2. 解决方案概述

Trellis 通过五个核心机制来应对这些挑战：

自动注入 所需的规范和工作流通过钩子脚本自动注入到每次 AI 对话中，而非依赖 AI 的记忆。

规范库 最佳实践存储在按领域（前端、后端、指南）组织的自动更新规范文件中，随时间推移价值不断增长。

并行会话 多个 AI 智能体可以同时在隔离的 Git 工作树中工作，每个都有自己的任务上下文。

团队同步 规范提交到代码仓库，确保一个开发者的最佳实践惠及所有人。

会话持久化 工作痕迹保存在仓库内的日志文件中，使 AI 能够跨会话恢复项目上下文。

1.3. 关键指标

指标	数值
仓库文件总数	666
TypeScript 源码 (src/)	50 个文件，共 5,650 行
Python 模板	55 个文件，共 9,395 行
Markdown 模板	321 个文件，共 14,400+ 行
测试代码	20 个测试文件，共 3,479 行
支持的平台	5 个（Claude Code、Cursor、iFlow、OpenCode、Codex）
斜杠命令	17+
智能体定义	6 个（implement、check、debug、research、plan、dispatch）
Git 提交（截至分析时）	205+
npm 预发布版本	20+（beta.0 至 rc.5）

项目存续时间	23 天（2026 年 1 月 26 日 – 2 月 17 日）
主要贡献者	6 人（taosu、kleinhe 等）

2. 架构概述

Trellis 采用分层架构，在 CLI 工具（通过 npm 全局安装）、项目本地配置（在 `trellis init` 期间生成）和在 AI 编程会话期间运行的运行时钩子系统之间实现了清晰的分离。

2.1. 系统层次

该架构由四个不同的层次组成：

层次	技术	职责
CLI 层	TypeScript / Node.js	安装、初始化、更新、迁移
配置层	Markdown / YAML / JSON	平台设置、智能体定义、斜杠命令
钩子层	Python 3.10+	会话启动注入、子智能体上下文、质量控制循环
脚本层	Python 3.10+	任务管理、会话记录、开发者身份、多智能体编排

Table 1: 系统层次及其使用的技术

2.2. 目录结构

当 Trellis 在项目中初始化时，会创建以下目录结构：

```
.trellis/                          # 工作流根目录
├── .version                       # 版本跟踪
├── .template-hashes.json          # 修改跟踪（SHA-256）
├── .developer                     # 开发者身份（已 gitignore）
├── .current-task                  # 当前任务指针
├── .gitignore                     # Git 忽略规则
├── workflow.md                    # 工作流指南（自动注入）
├── worktree.yaml                  # 多智能体工作树配置
├── scripts/                       # Python 工具脚本
│   ├── __init__.py
│   ├── common/                    # 共享工具（10 个模块）
│   ├── multi_agent/               # 并行智能体管道（6 个模块）
│   ├── task.py                    # 任务生命周期管理
│   ├── get_context.py             # 会话上下文检索
│   ├── add_session.py             # 日志会话记录
│   └── init_developer.py          # 开发者初始化
├── workspace/                     # 每个开发者的工作空间
│   ├── index.md                   # 工作空间索引
│   └── {developer}/               # 个人工作空间
│       ├── index.md               # 会话历史索引
│       └── journal-N.md           # 编号日志文件
├── tasks/                         # 任务管理
│   ├── archive/                   # 已归档的任务
│   └── {MM}-{DD}-{name}/          # 活动任务目录
│       ├── task.json              # 任务元数据
│       ├── prd.md                 # 产品需求文档
│       ├── implement.jsonl        # implement 智能体上下文
│       ├── check.jsonl            # check 智能体上下文
```

```
│       └── debug.jsonl            # debug 智能体上下文
└── spec/                          # 项目规范指南
    ├── guides/                    # 思维指南（3 份文档）
    ├── backend/                   # 后端指南（6 份文档）
    └── frontend/                  # 前端指南（7 份文档）

.claude/                           # Claude Code 配置
├── settings.json                  # 钩子配置
├── agents/                        # 智能体定义（6 个智能体）
├── commands/trellis/              # 斜杠命令（14 个命令）
├── hooks/                         # Python 钩子脚本（3 个钩子）
└── skills/                        # Claude 技能

.cursor/                           # Cursor 配置
└── commands/                      # 斜杠命令（13 个命令）

.iflow/                            # iFlow CLI 配置
├── settings.json                  # 钩子配置
├── agents/                        # 智能体定义
├── commands/trellis/              # 斜杠命令
└── hooks/                         # Python 钩子脚本

.opencode/                         # OpenCode 配置
├── commands/trellis/              # 斜杠命令
├── agents/                        # 智能体定义
├── plugin/                        # JavaScript 插件
└── lib/                           # 共享 JavaScript 工具

.agents/                           # Codex 技能
└── skills/                        # 技能定义
```

2.3. 信息流架构

运行时信息流遵循精心编排的顺序：

1. **会话启动**：当开发者打开 AI 编程会话时，`session-start.py` 钩子触发，将工作流上下文、规范索引和当前任务状态注入到对话中。
2. **任务研究**：Research 智能体分析代码库以识别相关的规范和代码模式。
3. **上下文配置**：任务特定的 `.jsonl` 文件被填充为对规范的引用，这些规范将在后续智能体调用中被注入。
4. **智能体调度**：当 Dispatch 智能体通过 Task 工具调用子智能体时，`inject-subagent-context.py` 钩子拦截该调用，并动态注入相关的规范、需求（来自 `prd.md`）和工作流指令到智能体的提示词中。
5. **质量控制**：当 Check 智能体尝试完成时，`ralph-loop.py` 钩子验证所有质量门禁是否已通过，然后才允许智能体停止。如果验证失败，智能体将被强制继续修复问题（最多 5 次迭代）。
6. **会话记录**：会话结束时，工作记录保存在日志文件中，供未来上下文恢复使用。

关键的架构洞察是**规范是被注入的，而非被记忆的**。这消除了对 AI 记忆的依赖，确保在长对话和多次会话中保持一致的质量。

3. CLI 工具设计

Trellis CLI 是一个全局安装的 Node.js 工具，使用 TypeScript 和 Commander.js 框架构建。它提供两个主要命令：`init` 和 `update`。

3.1. 技术栈

组件	技术	用途
CLI 框架	Commander.js 12.x	命令解析和选项处理
终端 UI	Inquirer 9.x	交互式提示和选择
样式	Chalk 5.x	终端彩色输出
横幅	Figlet 1.9.x	ASCII 艺术横幅生成
模板下载	Giget 3.x	GitHub 仓库模板获取
构建	TypeScript 5.7	类型安全编译至 ES2022
测试	Vitest 4.x	单元测试和集成测试
代码检查	ESLint 9.x + typescript-eslint	TypeScript 代码质量
Python 检查	basedpyright 1.37	钩子/脚本的 Python 类型检查
格式化	Prettier 3.4	代码格式化
Git 钩子	Husky 9.x	提交前质量强制执行

3.2. `init` 命令

`init` 命令编排了项目的完整 Trellis 设置。其执行过程遵循精密的多阶段流程：

3.2.1. 阶段 1：环境检测

该命令首先检测开发者的身份和项目特征：

- **开发者姓名**：自动从 `git config user.name` 检测，如不可用则交互式提示。可通过 `-u <name>` 显式设置。
- **项目类型**：通过分析项目文件和依赖，自动分类为 `frontend`、`backend`、`fullstack` 或 `unknown`。

项目类型检测会检查：

- **前端指标**：是否存在 `vite.config.ts`、`next.config.js`、React/Vue/Svelte/Angular 依赖、`src/App.tsx` 等。
- **后端指标**：是否存在 `go.mod`、`Cargo.toml`、`requirements.txt`、`pyproject.toml`、Express/Fastify/NestJS 依赖等。
- **Package.json 分析**：扫描 `dependencies` 和 `devDependencies` 中的框架特定包。

3.2.2. 阶段 2：平台选择

开发者选择要配置的 AI 平台：

```
trellis init                          # 交互式选择
trellis init --claude --cursor        # 显式指定平台
trellis init -y                       # 默认值（Claude + Cursor）
trellis init --codex --iflow          # 其他平台
```

平台选择由 `AI_TOOLS` 注册表驱动，该注册表作为所有平台数据的唯一事实来源。每个平台条目指定其显示名称、配置目录、CLI 标志、模板目录以及是否使用 Python 钩子。

3.2.3. 阶段 3：模板选择

CLI 从 GitHub 上的 Mindfold 文档仓库获取模板索引：

```
https://raw.githubusercontent.com/mindfold-ai/docs/main/marketplace/index.json
```

用户可以选择预构建的规范模板（如 `electron-fullstack`）或使用空白模板。对于已存在的规范目录，有三种处理策略：

- **跳过**：保留现有文件（默认）
- **覆盖**：替换所有文件
- **追加**：仅添加尚不存在的文件

3.2.4. 阶段 4：结构创建

工作流结构通过 `createWorkflowStructure()` 创建，它会生成：

- 所有 Python 脚本（从模板源复制）
- 开发者的工作空间目录
- 任务管理目录
- 基于项目类型的规范模板
- 配置文件（`worktree.yaml`、`.gitignore`、`workflow.md`）

3.2.5. 阶段 5：平台配置

通过将模板目录复制到项目来配置每个选定的平台。配置器处理平台特定的问题：

- **Claude Code**：复制命令、智能体、钩子和 `settings.json`，并解析 `{{PYTHON_CMD}}` 占位符以实现跨平台 Python 检测。
- **Cursor**：复制带 `trellis-` 前缀的命令（Cursor 不支持子目录命名空间）。
- **iFlow**：与 Claude Code 类似，具有相同的钩子架构。
- **OpenCode**：复制命令、智能体、JavaScript 插件和库文件。
- **Codex**：将技能定义以 SKILL.md 格式写入 `.agents/skills/`。

3.2.6. 阶段 6：后置设置

配置完成后：

1. 计算模板哈希并存储在 `.template-hashes.json` 中，用于未来的修改跟踪。
2. 通过 Python `init_developer.py` 脚本初始化开发者身份。
3. 创建引导任务（`00-bootstrap-guidelines`），指导开发者填写规范文件。
4. 显示激励性的“我们解决了什么”部分，将常见的开发者痛点与 Trellis 解决方案相联系。

3.3. `update` 命令

`update` 命令是代码库中最复杂的组件，共 1,684 行。它对 Trellis 配置文件执行智能的非破坏性更新。

3.3.1. 版本管理

更新过程从三方版本比较开始：

1. **项目版本**：从 `.trellis/.version` 读取（项目中安装的版本）。
2. **CLI 版本**：全局安装的 Trellis CLI 的版本。
3. **npm 版本**：npm 注册表上发布的最新版本。

基于比较结果：

- 如果 CLI 比项目新：可用升级
- 如果 CLI 比项目旧：降级保护（需要 `--allow-downgrade`）
- 如果 CLI 比 npm 旧：建议更新 CLI

3.3.2. 基于哈希的修改跟踪

这是一个关键创新。Trellis 在安装时将所有模板文件的 SHA-256 哈希存储在 `.trellis/.template-hashes.json` 中。更新时，系统比较：

1. **存储的哈希 vs 当前文件**：如果匹配，说明用户未修改该文件——可以安全地自动更新。
2. **当前文件 vs 新模板**：如果不同，说明模板已更新。

这产生了三种类别：

- **自动更新**：模板已更改 + 用户未修改 = 可以安全地自动替换。
- **用户已修改**：用户已修改 + 模板已更改 = 需要用户确认。
- **未更改**：两者匹配 = 跳过（已是最新）。

3.3.3. 迁移系统

迁移系统处理版本之间的文件组织变更。迁移存储为 `src/migrations/manifests/{version}.json` 中的 JSON 清单：

```
{
  "version": "0.3.0-beta.14",
  "description": "迁移描述",
  "breaking": true,
  "changelog": "变更内容",
  "recommendMigrate": true,
  "migrationGuide": "详细的迁移指南文本",
  "migrations": [
    { "type": "rename", "from": "old/path.md", "to": "new/path.md" },
    { "type": "rename-dir", "from": "old/dir/", "to": "new/dir/" },
    { "type": "delete", "from": "obsolete/file.md" }
  ]
}
```

迁移分为四类：

- **自动**：源文件存在，用户未修改，目标不存在——可安全自动执行。
- **确认**：源文件存在，用户已修改——需要交互确认。
- **冲突**：源和目标都存在——需要手动解决。

- **跳过**：源文件不存在——无需操作。

该系统还会检测"孤立"迁移，即源文件仍然存在但版本信息表明迁移应该已经完成。

3.3.4. 备份系统

在任何破坏性操作之前，update 命令会创建带时间戳的完整备份：

```
.trellis/.backup-2026-02-17T10-30-45/
├── .trellis/                # 完整的 trellis 目录快照
├── .claude/                 # 完整的 Claude 配置快照
├── .cursor/                 # 完整的 Cursor 配置快照
└── .iflow/                  # 完整的 iFlow 配置快照
```

用户数据目录（`workspace/`、`tasks/`）和之前的备份不包含在快照中。

3.3.5. 冲突解决

对于用户已修改的文件，update 命令提供逐文件或批量解决方案：

1. **覆盖**：替换为新版本
2. **跳过**：保留当前版本
3. **创建 .new**：将新版本保存为 `.new` 文件以供手动合并
4. **全部应用**：对所有剩余冲突应用相同的决定

4. 平台注册表与配置器系统

平台系统遵循类似插件的架构，其中所有平台特定的数据和行为都派生自中央注册表。

4.1. 唯一事实来源：AI_TOOLS

`src/types/ai-tools.ts` 中的 `AI_TOOLS` 常量定义了每个受支持的平台：

```
export const AI_TOOLS: Record<AITool, AIToolConfig> = {
  "claude-code": {
    name: "Claude Code",
    templateDirs: ["common", "claude"],
    configDir: ".claude",
    cliFlag: "claude",
    defaultChecked: true,
    hasPythonHooks: true,
  },
  cursor: {
    name: "Cursor",
    templateDirs: ["common", "cursor"],
    configDir: ".cursor",
    cliFlag: "cursor",
    defaultChecked: true,
    hasPythonHooks: false,
  },
  // ... opencode, iflow, codex
};
```

所有派生数据都从这个注册表计算，而非单独维护：

- `PLATFORM_IDS`：所有平台标识符的数组
- `CONFIG_DIRS`：所有配置目录的数组（`.claude`、`.cursor` 等）
- `ALL_MANAGED_DIRS`：`.trellis` 与所有配置目录的并集
- `getConfiguredPlatforms()`：通过目录存在性检测已安装的平台
- `getPlatformsWithPythonHooks()`：筛选需要 Python 的平台（用于 Windows 编码）
- `resolveCliFlag()`：将 CLI 标志映射到平台标识符

4.2. 平台函数注册表

`src/configurators/index.ts` 中的 `PLATFORM_FUNCTIONS` 对象将每个平台映射到其行为实现：

```
const PLATFORM_FUNCTIONS: Record<AITool, PlatformFunctions> = {
  "claude-code": {
    configure: configureClaude,         // 初始化：复制模板
    collectTemplates: () => { ... },    // 更新：收集模板
  },
  cursor: {
    configure: configureCursor,
    collectTemplates: () => { ... },
  },
```

```
  // ...
};
```

数据（在 AI_TOOLS 中）和行为（在 PLATFORM_FUNCTIONS 中）的分离使得添加新平台只需：

1. 在 AI_TOOLS 中添加条目（数据）
2. 创建配置器函数（行为）
3. 在 PLATFORM_FUNCTIONS 中注册（关联）
4. 创建模板目录
5. 在 cli/index.ts 中添加 CLI 标志

4.3. 编译时安全性

代码库使用 TypeScript 编译时断言来确保一致性：

```typescript
// 确保每个 CliFlag 都是 InitOptions 中的有效键
type _AssertCliFlagsInOptions = [CliFlag] extends [keyof InitOptions]
  ? true
  : "ERROR: CliFlag has values not present in InitOptions";
const _cliFlagCheck: _AssertCliFlagsInOptions = true;
```

这意味着添加新平台而不更新所有必需位置会导致编译时错误，而非运行时故障。

4.4. 跨平台 Python 检测

具有 Python 钩子的平台需要正确的 Python 命令检测。shared.ts 模块提供：

```typescript
function getPythonCommand(): string {
  // 首先尝试 python3（macOS/Linux 上首选）
  // 回退到 python（Windows 上常见）
  // 如果都不可用则默认为 python3
}

function resolvePlaceholders(content: string): string {
  return content.replace(/\{\{PYTHON_CMD\}\}/g, getPythonCommand());
}
```

设置文件中的 {{PYTHON_CMD}} 占位符在安装时解析，确保钩子使用宿主平台正确的 Python 命令。

5. 钩子系统

钩子系统是 Trellis 的运行时骨干，实现了“规范是被注入的，而非被记忆的”这一核心原则。三个 Python 钩子拦截不同的生命周期事件。

5.1. 会话启动钩子（session-start.py）

触发器：`SessionStart` 事件，匹配器为 `startup`

此钩子在 AI 编程会话开始时触发。它执行以下操作：

1. 检查非交互模式（`CLAUDE_NON_INTERACTIVE=1` 或 `OPENCODE_NON_INTERACTIVE=1`），并在子智能体上下文中跳过注入。
2. 运行 `get_context.py` 获取当前会话状态（开发者身份、git 状态、活动任务、当前任务）。
3. 读取 `workflow.md` 获取开发工作流指南。
4. 从 `spec/frontend/index.md`、`spec/backend/index.md` 和 `spec/guides/index.md` 读取规范索引。
5. 读取 `start.md` 命令模板获取任务启动指令。
6. 返回包含 `hookSpecificOutput` 的 JSON 响应，其中所有上下文包裹在 XML 标签中：

```
<session-context>会话初始化上下文</session-context>
<current-state>开发者身份、git 状态、任务</current-state>
<workflow>完整的工作流指南</workflow>
<guidelines>规范索引</guidelines>
<instructions>任务启动指令</instructions>
<ready>上下文已加载。等待用户的第一条消息。</ready>
```

这确保每次 AI 会话都以全面的项目知识开始。

5.2. 子智能体上下文注入钩子（inject-subagent-context.py）

触发器：`PreToolUse` 事件，匹配器为 `Task`（在 Task 工具调用之前拦截）

这是最复杂的钩子，大约 24,000 字节。其核心设计理念是：“调度智能体是纯粹的路由器；钩子负责注入所有上下文。”

5.2.1. 智能体特定的上下文加载

每种类型的智能体接收定制的上下文：

Implement 智能体：
1. 从 `implement.jsonl` 加载规范（如果不存在则回退到 `spec.jsonl`）
2. 注入产品需求文档（`prd.md`）
3. 包含实现说明（`info.md`）如果存在
4. 添加工作流约束：“遵循所有规范，完成前运行 lint 和类型检查”

Check 智能体：
1. 从 `check.jsonl` 加载规范（如果不存在则回退到 `spec.jsonl`）
2. 包含质量指南、跨层思维指南和完工检查清单
3. 注入硬编码的检查文件：`quality-guidelines.md`、`cross-layer-thinking-guide.md`、`finish-work.md`
4. 添加工作流：“对照规范审查所有代码更改，直接修复问题”

Debug 智能体：
1. 从 `debug.jsonl` 加载规范（如果不存在则回退到 `spec.jsonl`）
2. 包含检查文件作为上下文
3. 如果可用则添加 codex 审查输出
4. 工作流："理解问题，按照规范修复，验证修复效果"

Research 智能体：
1. 加载可选的 `research.jsonl`
2. 注入项目结构概述
3. 工作流："分析代码库，找到相关的规范和模式"

5.2.2. 上下文源文件（`.jsonl`）

每个任务目录可以包含 `.jsonl` 文件，定义哪些规范应注入到每个智能体的上下文中：

```
{"file": ".trellis/spec/backend/quality-guidelines.md", "reason": "代码质量标准"}
{"file": ".trellis/spec/guides/cross-layer-thinking-guide.md", "reason": "跨层验证"}
{"file": "src/utils/", "type": "directory", "reason": "工具函数模式"}
```

此机制允许任务特定的上下文配置：前端任务注入前端规范，而后端任务注入后端规范。

5.2.3. 阶段管理

钩子根据被调用的子智能体类型自动更新 `task.json` 中的 `current_phase`。这使调度智能体能够跟踪多阶段工作流的进度，无需手动记录。

从子智能体类型到动作的映射：
- `implement` → 查找动作为 "implement" 的阶段
- `check` → 查找动作为 "check" 或 "finish" 的阶段
- `debug` → 不更新阶段（可随时调用）
- `research` → 不更新阶段

5.3. Ralph 循环钩子（`ralph-loop.py`）

触发器：`SubagentStop` 事件，匹配器为 `check`（在 Check 智能体尝试停止时拦截）

以"Ralph Wiggum 技术"命名（指像走廊巡视员一样强制执行质量的引用），此钩子实现了质量控制循环。

5.3.1. 验证方法

钩子支持两种验证方式：

编程验证：如果 `worktree.yaml` 定义了 `verify` 命令，则执行它们：

```
verify:
  - pnpm lint
  - pnpm typecheck
```

如果任何命令失败，智能体将被阻止停止，并收到错误输出作为反馈。

基于标记的验证：如果未配置验证命令，钩子会读取 `check.jsonl` 并从条目原因生成完成标记：

对于如"确保 TypeScript 类型检查通过"这样的原因，标记会变成 `TYPECHECK_FINISH`。Check 智能体必须输出这些标记以证明它完成了每个验证步骤。

5.3.2. 安全机制

- **最大迭代次数**：5 次迭代的硬限制防止无限循环。
- **状态超时**：如果状态文件超过 30 分钟，会自动重置。
- **任务变更检测**：如果当前任务发生变化，循环状态会重置。
- **状态跟踪**：`.trellis/.ralph-state.json` 中的持久状态跟踪跨智能体轮次的迭代计数和完成状态。

5.3.3. 循环控制流

```
Check 智能体尝试停止
        ↓
ralph-loop.py 拦截
        ↓
检查验证条件
  ├── 所有标记都存在? → 允许停止
  ├── 所有验证命令通过? → 允许停止
  ├── 达到最大迭代次数? → 允许停止（带警告）
  └── 验证失败 → 阻止停止，提供反馈
        ↓（如果被阻止）
Check 智能体继续工作
        ↓
再次尝试停止……
```

6. 智能体系统

Trellis 定义了六个专门化的智能体，每个都有独特的职责、工具访问权限和工作流约束。所有智能体都使用 Opus 模型以获得最大能力。

6.1. Implement 智能体

用途：代码实现专家，理解规范和需求后实现功能。

工具：Read、Write、Edit、Bash、Glob、Grep、网络搜索、代码上下文搜索

工作流：
1. 阅读所有注入的规范和需求
2. 理解现有的代码模式和架构
3. 实现请求的更改
4. 运行 lint 和类型检查以验证正确性

约束：
- 不进行 git commit、push 或 merge 操作
- 必须严格遵循注入的规范
- 不过度工程化，超出需求范围
- 完成前运行验证

6.2. Check 智能体

用途：代码质量验证专家，对照规范审查更改并自行修复问题。

工具：与 Implement 智能体相同

工作流：
1. 获取所有代码更改（通过 `git diff`）
2. 对照注入的规范进行审查
3. 直接修复发现的任何问题（而非仅仅报告）
4. 运行验证命令
5. 为 Ralph 循环输出完成标记

关键原则：“自行修复问题，不要仅仅报告。”Check 智能体应当主动纠正问题，而非被动地记录它们。

Ralph 循环集成：Check 智能体必须输出从 `check.jsonl` 条目派生的特定标记（如 `TYPECHECK_FINISH`、`LINT_FINISH`）以证明验证已完成。

6.3. Debug 智能体

用途：问题修复专家，进行有针对性的缺陷修复，修复精确且最小化。

工具：与 Implement 智能体相同

工作流：
1. 理解报告的问题
2. 如有需要进行研究（搜索代码库，阅读相关文件）
3. 按优先级分类：P1（关键）、P2（重要）、P3（次要）
4. 一次修复一个问题，验证每次修复

5. 所有修复完成后运行类型检查

指导原则：
- 应该做：进行精确、最小化的修复
- 不应该做：重构周围代码、添加功能、使用非空断言

6.4. Research 智能体

用途：纯粹的信息收集和文档化，不修改任何文件。

工具：Read、Glob、Grep、网络搜索、代码上下文搜索（无 Write/Edit）

核心原则："查找并解释信息"——这个智能体是记录者，不是审查者。

禁止的操作：
- 建议改进
- 批评代码质量
- 推荐重构
- 修改任何文件

输出格式：包含文件路径、代码模式和相关规范的结构化研究报告。

6.5. Plan 智能体

用途：多智能体管道规划器，评估需求并配置任务目录。

工具：Read、Bash、Glob、Grep、Task（可以生成子智能体）

工作流：
1. 评估需求的清晰度、完整性和范围
2. 接受或拒绝任务（附带拒绝理由）
3. 初始化上下文文件（`implement.jsonl`、`check.jsonl`、`debug.jsonl`）
4. 生成 Research 智能体来分析代码库
5. 编写包含明确需求和验收标准的 `prd.md`
6. 在 `task.json` 中配置任务元数据
7. 验证完整性

拒绝标准：Plan 智能体可以拒绝不清晰、不完整、超出范围、可能有害或对单次实现周期来说过大的任务。

6.6. Dispatch 智能体

用途：纯粹的编排器，按正确的阶段顺序将工作路由到其他智能体。

工具：Read、Bash（有限——无文件修改工具）

核心原则："纯调度器"——它从不直接读取规范。相反，钩子系统将上下文注入到每个子智能体调用中。

阶段处理：
1. 阶段：implement → 调用 Implement 智能体（30 分钟超时）
2. 阶段：check → 调用 Check 智能体（15 分钟超时）
3. 阶段：debug → 调用 Debug 智能体（20 分钟超时，仅在检查失败时）

4. 阶段：finish → 调用 Check 智能体并带 `[finish]` 标记
5. 阶段：create-pr → 执行 `multi_agent/create_pr.py` 进行 git 提交

Dispatch 智能体是**唯一**执行 git 提交的智能体，且仅在最终阶段通过 Python 脚本而非直接 git 命令来完成。

7. 任务管理系统

任务是 Trellis 中工作的基本单位。每个任务是一个包含元数据、需求和上下文配置文件的目录。

7.1. 任务生命周期

```
创建 → 规划 → 进行中 → 审查 → 完成 → 归档
```

任务通过 `task.py` 脚本管理，该脚本提供全面的 CLI：

```
python3 .trellis/scripts/task.py create "标题" --slug name
python3 .trellis/scripts/task.py init-context <dir> <type>
python3 .trellis/scripts/task.py add-context <dir> implement <path> "原因"
python3 .trellis/scripts/task.py start <dir>
python3 .trellis/scripts/task.py finish
python3 .trellis/scripts/task.py archive <name>
python3 .trellis/scripts/task.py list
```

7.2. 任务目录结构

每个任务目录包含：

文件	用途
`task.json`	元数据：标题、状态、优先级、负责人、阶段、时间戳
`prd.md`	产品需求文档，包含目标、标准和技术说明
`implement.jsonl`	Implement 智能体的上下文文件
`check.jsonl`	Check 智能体的上下文文件
`debug.jsonl`	Debug 智能体的上下文文件
`spec.jsonl`	任何智能体的回退上下文（如果特定文件不存在）
`research.jsonl`	Research 智能体的可选上下文
`info.md`	额外的实现说明（可选）

7.3. 任务元数据（`task.json`）

```
{
  "title": "实现用户认证",
  "description": "为 API 添加基于 JWT 的认证",
  "status": "in_progress",
  "dev_type": "backend",
  "scope": "auth",
  "priority": "P1",
```

```
  "creator": "claude-agent",
  "assignee": "claude-agent",
  "createdAt": "2026-02-17",
  "current_phase": 2,
  "next_action": [
    { "phase": 1, "action": "implement" },
    { "phase": 2, "action": "check" },
    { "phase": 3, "action": "debug" },
    { "phase": 4, "action": "finish" }
  ],
  "branch": "feat/auth",
  "worktree_path": null,
  "commit": null,
  "pr_url": null
}
```

`next_action` 数组定义了多阶段工作流。`current_phase` 字段由钩子系统在智能体完成工作时自动更新。

7.4. 上下文配置（`.jsonl` 文件）

这些文件定义了哪些规范应注入到每个智能体的上下文中：

```
{"file": ".trellis/spec/backend/quality-guidelines.md", "reason": "代码质量"}
{"file": ".trellis/spec/backend/error-handling.md", "reason": "错误处理模式"}
{"file": "src/models/user.ts", "reason": "现有的用户模型模式"}
```

钩子系统在运行时读取这些文件，并将引用的文件内容注入到智能体的提示词中。这意味着每次智能体调用都会重新加载规范，消除了上下文漂移。

7.5. 引导任务

当 Trellis 首次初始化时，它会自动创建一个引导任务（`00-bootstrap-guidelines`），指导开发者填写规范文件。此任务：

- 根据项目类型识别哪些规范需要填写
- 提供记录现有代码模式的分步指令
- 包含完成检查清单
- 被设置为当前任务，因此会出现在会话启动上下文中

8. 规范系统

规范系统是 Trellis 的知识管理层。它提供结构化的、特定领域的指南，自动注入到 AI 智能体的上下文中。

8.1. 组织方式

规范按三个领域组织：

8.1.1. 后端指南（`spec/backend/`）

六份涵盖服务端开发的文档：

1. **index.md**：入口，包含指南索引
2. **directory-structure.md**：模块组织、文件布局、设计决策
3. **quality-guidelines.md**：代码标准、禁止的模式、CLI 设计模式
4. **error-handling.md**：错误类型、处理策略、错误传播
5. **logging-guidelines.md**：结构化日志、日志级别、应该记录什么
6. **script-conventions.md**：`.trellis/scripts/` 的 Python 脚本标准

8.1.2. 前端指南（`spec/frontend/`）

七份涵盖客户端开发的文档：

1. **index.md**：入口，包含指南索引
2. **directory-structure.md**：组件/页面/钩子组织
3. **component-guidelines.md**：组件模式、属性、组合
4. **hook-guidelines.md**：自定义钩子、数据获取模式
5. **state-management.md**：本地状态、全局状态、服务端状态
6. **quality-guidelines.md**：代码标准、禁止的模式
7. **type-safety.md**：TypeScript 类型模式、验证

8.1.3. 思维指南（`spec/guides/`）

三份扩展开发者思维的文档：

1. **index.md**：指南索引，包含思维触发器
2. **cross-layer-thinking-guide.md**：跨系统层次的数据流验证
3. **code-reuse-thinking-guide.md**：模式识别和减少重复
4. **cross-platform-thinking-guide.md**：平台特定假设和兼容性

8.2. 设计理念

规范系统体现了几个关键原则：

记录现实，而非理想 指南应描述代码库**实际做了什么**，而非开发者希望它做什么。AI 需要匹配现有模式。

分层架构 与单体的 `CLAUDE.md` 或 `.cursorrules` 文件不同，Trellis 使用分层方式，只为当前任务加载相关规范。这减少了上下文开销并提高了 AI 的专注度。

活的文档 规范随代码库一起演进。`/trellis:update-spec` 命令提供结构化模板，用于添加新模式、设计决策和经验教训。

团队知识沉淀 因为规范提交到仓库，一个开发者的宝贵经验惠及整个团队。

8.3. 远程规范模板

Trellis 支持从 Mindfold 文档仓库下载预构建的规范模板。这允许团队从针对特定技术栈（如 `electron-fullstack`）的成熟规范开始，而非空白模板。

模板市场索引从以下地址获取：

```
https://github.com/mindfold-ai/docs/tree/main/marketplace
```

目前仅支持 `spec` 类型的模板，计划支持 `skill`、`command` 和 `full` 模板类型。

9. 多智能体管道

多智能体管道支持在不同功能上同时运行多个 AI 编程智能体，每个都在隔离的 Git 工作树中。

9.1. 工作树架构

当调用 `/trellis:parallel` 时，Trellis 为每个并行任务创建一个单独的 Git 工作树：

```
project/                              # 主工作树
├── .trellis/
│   └── tasks/
│       └── 02-17-feature-x/         # 设置了 worktree_path 的任务
│
trellis-worktrees/                    # 工作树根目录（可配置）
├── feature-x/                       # 隔离的工作树
│   ├── .trellis/.developer          # 从主工作树复制
│   └── (完整的项目检出)             # 独立分支
└── feature-y/                       # 另一个隔离的工作树
```

9.2. 配置（`worktree.yaml`）

```yaml
worktree_dir: ../trellis-worktrees  # 创建工作树的位置

copy:                               # 复制到每个工作树的文件
  - .trellis/.developer

post_create:                        # 工作树创建后执行的命令
  # - npm install
  # - pnpm install --frozen-lockfile

verify:                             # 质量门禁命令
  # - pnpm lint
  # - pnpm typecheck
```

9.3. 管道脚本

`scripts/multi_agent/` 中的六个 Python 脚本管理管道：

1. **start.py**：创建新工作树，设置分支，复制所需文件，运行创建后命令，并在工作树中启动 AI 智能体。
2. **plan.py**：评估任务需求并使用适当的上下文文件配置任务目录。
3. **status.py**：报告所有活动工作树智能体的状态（运行中、已停止、错误）。
4. **create_pr.py**：在工作树中提交更改并创建 GitHub 拉取请求。
5. **cleanup.py**：删除工作树并清理智能体注册表条目。

9.4. 智能体注册表

`registry.py` 模块维护活动智能体的 JSON 注册表：

```
{
  "agents": [
    {
      "id": "agent-uuid",
      "worktree_path": "/path/to/worktree",
      "pid": 12345,
      "started_at": "2026-02-17T10:30:00",
      "task_dir": ".trellis/tasks/02-17-feature-x",
      "platform": "claude"
    }
  ]
}
```

这使得能够监控和管理跨不同工作树的多个并发智能体。

10. 会话持久化

Trellis 提供日志系统，使 AI 能够跨会话保持上下文。

10.1. 日志系统

每个开发者在 `.trellis/workspace/{name}/` 有一个个人工作空间，包含：

- **index.md**：带有日志文件链接的会话历史表
- **journal-N.md**：编号的日志文件（每个最多 2,000 行）

10.2. 会话记录

`/trellis:record-session` 命令触发 `add_session.py`，它会：

1. 检测开发者的当前日志文件
2. 检查文件是否超过 2,000 行限制
3. 如果需要，创建新的编号日志文件
4. 追加包含时间戳、提交哈希和描述的会话摘要
5. 用新的会话条目更新 `index.md`

10.3. 上下文恢复

当新会话启动时，`session-start.py` 钩子读取：

1. 开发者最近的日志条目
2. 当前任务（如果有）
3. Git 状态和最近的提交

这为 AI 提供了之前工作的摘要，使其能够从上次会话停止的地方继续，无需开发者显式重新解释。

10.4. 开发者身份

多开发者支持确保每个团队成员有隔离的工作空间：

```
python3 .trellis/scripts/init_developer.py john-doe
python3 .trellis/scripts/init_developer.py cursor-agent
python3 .trellis/scripts/init_developer.py claude-agent
```

每个开发者有自己的日志文件，`.developer` 文件被 gitignore 以防止身份冲突。

11. 斜杠命令

Trellis 提供 17+ 个斜杠命令，为 AI 准备特定任务和上下文。这些命令以 Markdown 文件实现，在调用时展开为完整的提示词。

11.1. 命令类别

11.1.1. 会话管理

命令	用途
`/trellis:start`	开始会话：初始化上下文，阅读指南，分类任务
`/trellis:record-session`	提交后将会话摘要记录到日志
`/trellis:finish-work`	提交前检查清单：lint、类型、测试、API 变更、跨层
`/trellis:parallel`	在隔离的工作树中生成并行智能体

11.1.2. 开发工作流

命令	用途
`/trellis:brainstorm`	复杂任务的需求发现
`/trellis:before-frontend-dev`	编码前加载前端指南
`/trellis:before-backend-dev`	编码前加载后端指南
`/trellis:break-loop`	调试后的深度缺陷分析（5 维框架）

11.1.3. 质量保证

命令	用途
`/trellis:check-frontend`	对照前端指南验证代码
`/trellis:check-backend`	对照后端指南验证代码
`/trellis:check-cross-layer`	验证跨系统层次的数据流

11.1.4. 知识管理

命令	用途
`/trellis:update-spec`	将新知识沉淀到规范文件中
`/trellis:integrate-skill`	将 Claude 技能整合到项目指南中
`/trellis:onboard`	面向新团队成员的三段式入职培训

`/trellis:create-command`	创建新的自定义斜杠命令
`/trellis:improve-ut`	提升单元测试质量和覆盖率

11.2. 值得注意的命令设计

11.2.1. Brainstorm 命令

brainstorm 命令实现了结构化的需求发现流程：

1. 确认并分类任务
2. 立即创建任务目录
3. **一次问一个**问题（不要同时抛出多个问题让人应接不暇）
4. 每次回答后更新 PRD
5. 为决策提出架构方案
6. 在继续之前确认最终需求

11.2.2. Break-Loop 命令

在困难的调试会话后，此命令触发 5 维分析：

1. **根本原因类别**：规范缺失、跨层契约、变更传播失败、测试覆盖缺口或隐式假设
2. **修复失败原因**：分析为什么初始修复尝试未能奏效
3. **预防机制**：哪些规范更新或检查可以防止再次发生
4. **系统性扩展**：代码库中其他哪些地方可能存在相同问题
5. **知识沉淀**：将学到的经验更新到规范中

12. 测试策略

Trellis 维护着包含 20 个测试文件的全面测试套件，涵盖单元测试、集成测试和回归测试。

12.1. 测试基础设施

方面	配置
框架	Vitest 4.x 搭配 TypeScript ESM
测试超时	每个测试 10 秒
覆盖率提供者	vitest/coverage-v8
覆盖率报告器	text、html、json-summary
模块系统	ES 模块 (ESM)
CI 集成	GitHub Actions (Node.js 20)

12.2. 测试理念

Trellis 的测试理念强调：

1. **最小化模拟**：只模拟外部依赖（figlet、inquirer、child_process、fetch）。内部模块通过完整代码路径执行。
2. **真实文件系统**：集成测试使用临时目录进行实际文件操作，而非模拟文件系统。
3. **注册表不变式**：测试验证跨模块的一致性，灵感来自 SQLite 的完整性检查。
4. **回归预防**：记录历史缺陷并编写验证测试以防止复发。

12.3. 测试类别

12.3.1. 单元测试

隔离测试各个模块：

- **配置器**：平台注册表一致性、已配置平台检测、平台模板创建
- **类型**：AI_TOOLS 注册表字段验证、模板目录包含
- **模板**：模板常量非空、Python 语法有效性、settings.json 有效性、命令/智能体/钩子集合
- **工具**：项目检测、文件写入、模板哈希、版本比较、模板获取
- **常量**：路径常量、目录名称、分隔符约定
- **迁移**：版本过滤、迁移摘要、元数据聚合

12.3.2. 集成测试

使用真实文件系统测试完整的命令流程：

Init 集成（10 个场景）：

1. 使用默认值创建预期的目录结构
2. 单平台仅创建该平台的文件
3. 多平台创建所有选定平台的文件

4. Codex 平台创建 `.agents/skills/` 结构
5. 强制模式覆盖已修改的文件
6. 跳过模式保留已修改的文件
7. 使用强制模式重新初始化产生相同的文件集
8. 将开发者名称传递给初始化脚本
9. 写入正确的版本文件
10. 为所有领域创建规范模板

Update 集成（11 个场景）：

1. 相同版本更新是真正的无操作（零更改，无备份）
2. 试运行不进行任何文件更改
3. 重新创建被删除的模板文件
4. 自动更新未修改的模板
5. 强制覆盖用户已修改的文件
6. SkipAll 保留用户已修改的文件
7. CreateNew 创建 `.new` 副本而不覆盖
8. 成功更新后更新版本文件
9. 在更改前创建备份目录
10. 降级保护阻止不安全的降级
11. AllowDowngrade 允许有意的降级

12.3.3. 回归测试

40+ 个回归测试按历史缺陷类别组织：

Windows/编码回归：

- Windows 的 UTF-8 stdout 配置
- 子进程的 `CREATE_NEW_PROCESS_GROUP` 标志
- 入口脚本中的内联编码修复

路径问题：

- 任务路径推导（不在 workspace 下）
- 模板中无硬编码用户名
- 正确的路径分隔符使用

语义化版本/迁移引擎：

- 预发布版本排序
- 数字标识符比较
- 迁移字段完整性验证

模板完整性：

- 已迁移到 Python 的模板中不残留 shell 脚本
- multi_agent 目录使用下划线命名
- 设置模板中的 JSON 有效性
- 钩子命令中的占位符使用

平台注册表：

- 所有平台注册了必需字段
- CLI 适配器支持所有平台
- 配置目录命名一致

12.4. 模拟策略

项目保持严格的模拟纪律：

依赖	模拟原因	方法
`figlet`	ASCII 横幅，不可测试	`vi.mock("figlet")`
`inquirer`	交互式提示，CI 中无 TTY	`vi.mock("inquirer")`
`child_process`	Git 配置、Python 调用	`vi.mock("node:child_process")`
`fetch`	npm 注册表网络调用	`vi.stubGlobal("fetch")`
`process.cwd()`	重定向到临时目录	`vi.spyOn(process, "cwd")`
`console.log`	静默测试输出	`vi.spyOn(console, "log")`

从不模拟：`fs`、`path`、内部模块（`configurators/`、`utils/`、`templates/`）、`chalk`。

13. CI/CD 管道

项目使用 GitHub Actions 进行持续集成和部署。

13.1. 构建验证（ci.yml）

每次推送到 `main` 和所有拉取请求时触发：

1. 检出仓库
2. 设置 Node.js 20 和 pnpm 9
3. 安装依赖（`--frozen-lockfile`）
4. 构建项目（`tsc` + 模板复制）
5. 验证构建输出（`dist/` 存在且包含预期产物）

13.2. npm 发布（publish.yml）

在 GitHub 发布和版本标签（`v*`）时触发：

1. 检出完整 git 历史
2. 设置 Node.js 20 和 npm 注册表配置
3. 构建项目
4. 根据版本确定 npm 标签：
   - `beta` → `beta` 标签
   - `alpha` → `alpha` 标签
   - `rc` → `rc` 标签
   - （其他）→ `latest` 标签
5. 使用适当的标签发布到 npm

这种智能标签确保预发布版本在 npm 上被正确标记，防止通过 `npm install @mindfoldhq/trellis` 意外安装不稳定版本。

13.3. 提交前钩子

Husky 在每次提交时运行 `lint-staged`，它会：

- 对暂存的 `.ts` 文件应用 ESLint 自动修复
- 对暂存的 `.ts` 文件应用 Prettier 格式化

这确保在提交级别维护代码质量，独立于 CI。

14. 版本历史与演进

Trellis 自 2026 年 1 月 26 日创建以来发展迅速，在 23 天内完成了 205+ 次提交。

14.1. 版本时间线

版本	日期	关键变更
0.2.9	1 月 26 日	初始项目创建和公开发布
0.3.0-beta.0–2	1 月 28–29 日	模板重组、命令命名空间化
0.3.0-beta.3–5	1 月 30 日	Shell 到 Python 迁移开始、跨平台修复
0.3.0-beta.6–9	1 月 31 日	Brainstorm 增强、工具选择重构
0.3.0-beta.10–12	2 月 2 日	注册表重构、数据驱动方法
0.3.0-beta.13–14	2 月 3 日	OpenCode 支持、迁移清单
0.3.0-beta.15–16	2 月 3–4 日	iFlow 支持、Windows 修复
0.3.0-rc.0–1	2 月 6 日	候选发布版、测试套件、单元测试规范
0.3.0-rc.2	2 月 9 日	Codex 支持、测试强化

14.2. 重大演进里程碑

14.2.1. Shell 到 Python 迁移（beta.3–beta.5）

最初所有脚本都用 Bash 实现。之后迁移到 Python 3.10+，原因如下：

- 跨平台兼容性（Windows 没有原生 Bash）
- 更好的错误处理和数据解析（JSON、YAML）
- 通过 basedpyright 实现类型安全
- 更清晰的子进程管理

14.2.2. 平台注册表重构（beta.10–beta.12）

平台系统从分散的条件判断重构为集中式注册表：

- `AI_TOOLS` 成为唯一事实来源
- 所有派生列表自动计算
- 添加新平台变成 5 步清单
- 编译时断言防止不完整的添加

14.2.3. 多平台扩展（beta.13–rc.2）

快速的平台添加：

- **OpenCode**（beta.13）：基于 JavaScript 插件的架构
- **iFlow**（beta.15）：克隆 Claude Code 钩子架构
- **Codex**（rc.2）：基于技能的架构，使用 SKILL.md 格式

每次添加都得益于注册表模式，只需对核心代码进行最少的修改。

14.2.4. 测试套件添加（rc.0）

添加了全面的测试套件，包括：

- 所有模块的单元测试
- 两个命令的集成测试
- 所有历史缺陷的回归测试
- 覆盖率跟踪和 CI 集成
- 在规范中记录的测试约定

14.3. 提交模式

项目严格遵循约定式提交：

```
feat(scope): description    # 新功能
fix(scope): description     # 缺陷修复
docs(scope): description    # 文档
refactor(scope): description # 代码重构
test(scope): description    # 测试添加
chore(scope): description   # 维护
```

开发速度非常高，最活跃的日子提交了 30–35 次。

15. 设计模式与原则

从 Trellis 代码库中涌现出几个值得注意的设计模式。

15.1. 自举使用

Trellis 使用自己的配置来开发自身。项目包含带有活动规范和工作流的 `.trellis/`、`.claude/`、`.cursor/`、`.iflow/` 和 `.opencode/` 目录。然而，在初始化用户项目时，模板来源于 `src/templates/` 而非项目自身的配置文件，确保了自举使用内容和分发内容之间的清晰分离。

15.2. 唯一事实来源（SSOT）

`AI_TOOLS` 注册表展示了 SSOT：

- 平台数据只定义一次
- 所有列表、检查和行为都从中派生
- 添加/删除平台只需在一个数据位置加一个行为位置进行修改
- 编译时断言捕获不完整的更改

15.3. 基于哈希的变更跟踪

Trellis 不使用简单的文件比较，而是使用 SHA-256 哈希来区分：

- 用户修改（需要保留的有意更改）
- 模板漂移（可安全自动应用的 Trellis 更新）
- 未更改的文件（无需操作）

这使得“智能更新”成为可能，在尊重用户自定义的同时仍能应用框架改进。

15.4. 规范注入优于记忆

核心架构决策：与其寄希望于 AI 记住指令，不如在每次机会都重新注入。这通过以下方式实现：

- 会话启动钩子（全局上下文）
- 子智能体上下文钩子（任务特定上下文）
- Ralph 循环钩子（质量强制执行）

每次智能体调用都是自包含的，包含所有必要的上下文，消除了“AI 忘记了我的指令”这一失败模式。

15.5. 基于阶段的编排

多阶段工作流（research → implement → check → debug → finish）是数据驱动的：

- 阶段在 `task.json` 中定义为数组
- 当前阶段由钩子跟踪并自动更新
- Dispatch 智能体按顺序读取阶段
- 阶段完成触发推进

这允许通过修改阶段数组来为每个任务自定义工作流。

15.6. 渐进式增强

Trellis 在不移除现有工具的情况下添加能力：

- 与现有的 `CLAUDE.md`、`.cursorrules` 和 `AGENTS.md` 文件并行工作
- 规范在平台原生配置之上叠加
- 命令是对平台能力的扩展而非替代
- 开发者可以部分使用 Trellis（如仅使用规范而不使用多智能体）

16. 跨平台考量

Trellis 运行于 macOS、Linux 和 Windows 上，需要仔细关注平台差异。

16.1. Python 命令检测

不同平台使用不同的 Python 命令：

- macOS/Linux：`python3`（首选）
- Windows：`python`（常见，通过 Microsoft Store 或安装程序）

`getPythonCommand()` 函数在安装时探测可用性，结果通过 `{{PYTHON_CMD}}` 占位符写入 `settings.json`。

16.2. Windows 编码问题

大量缺陷修复涉及 Windows 特定的编码问题：

1. **stdout UTF-8**：Python 钩子在 Windows 上显式重新配置 stdout/stderr 以使用 UTF-8 编码：

```
if sys.platform == "win32":
    sys.stdout.reconfigure(encoding="utf-8", errors="replace")
```

2. **Git UTF-8**：Git 命令使用 `-c i18n.logOutputEncoding=UTF-8` 以获得一致的输出。
3. **子进程创建**：Windows 在多智能体场景中需要 `CREATE_NEW_PROCESS_GROUP` 标志以进行正确的子进程管理。

16.3. 路径处理

- 所有路径内部使用正斜杠，即使在 Windows 上也是如此
- Python 脚本中使用 `pathlib.Path` 进行跨平台路径解析
- `isManagedPath()` 中的反斜杠规范化确保 Windows 兼容性：

```
const normalized = dirPath.replace(/\\/g, "/");
```

17. 未来路线图

根据 README 和代码库分析，以下功能已在规划中：

17.1. 已确认的路线图项目

1. **更好的代码审查**：具有增强 Check 智能体能力的更全面的自动化审查工作流。
2. **技能包**：可作为插件安装的预构建工作流包（针对特定框架的即插即用模式）。
3. **更广泛的工具支持**：持续扩展到更多 AI 编程平台。
4. **更强的会话连续性**：无需显式调用 `/trellis:record-session` 即可自动保存全会话历史。
5. **可视化并行会话**：每个并行智能体的实时进度可视化。

17.2. 从代码库推断

1. **模板市场扩展**：模板获取器支持 `skill`、`command` 和 `full` 模板类型，但目前仅实现了 `spec`。
2. **增强的迁移指南**：迁移系统支持带有 AI 指令的详细逐版本指南，暗示预期会有更多破坏性变更。
3. **Python 类型覆盖**：basedpyright 集成表明对钩子脚本的 Python 代码质量有持续投入。

18. 结论

Trellis 代表了一种解决 AI 辅助软件开发问题的精密方法。它不将 AI 编程助手视为需要不断提醒的无状态工具，而是提供了一个框架：

1. **结构化地编码知识**——通过持久存储在仓库中的规范文件
2. **自动强制执行标准**——通过在每次智能体交互时基于钩子的注入
3. **支持并行工作**——通过带有协调任务管理的 Git 工作树隔离
4. **保留上下文**——通过基于日志的会话持久化
5. **跨团队扩展**——通过每个开发者的工作空间和共享规范

该架构的关键洞察——规范应该被注入而非被记忆——解决了当前 AI 编程助手的根本局限：上下文漂移。通过在每次智能体调用时重新加载相关规范，Trellis 将 AI 代码生成的随机性转化为更确定性的、质量可控的过程。

从 v0.2.9（1 月 26 日）到 v0.3.0-rc.2（2 月 9 日）的快速演进——包括 Shell 到 Python 迁移、五个平台集成、全面的测试套件和迁移系统——既展示了项目的雄心，也展示了其架构的灵活性。类似插件的平台注册表模式被证明特别有效，使新平台支持能在数小时而非数天内添加。

随着 AI 编程助手变得更加强大，Trellis 的价值从“让 AI 能够工作”转向“让 AI 一致地、大规模地、跨团队地工作”。规范注入模式与平台无关，有可能应用于当前五个受支持平台之外的未来 AI 编程工具。

报告结束
由 Claude Code 生成 — 2026 年 2 月 17 日
总页数：40