

# OpenSpec: 规范化 Agentic Coding 的关键基础设施

基于 Claude Code 的企业级 AI 辅助开发规范化研究

---

技术研究汇报 | 2026年2月

# 目录

01

## 背景与问题

Agentic Coding 现状与挑战

02

## OpenSpec 核心解析

概念、工作流程、协作模式

03

## 实测对比验证

量化交易系统 — 初始构建 + 需求变更对比

04

## 企业级价值与 ROI

规范化、可追溯、可协作、投资回报

05

## 学术前沿与竞争格局

2023-2026 五大趋势 + 差异化定位

06

## 六大研究改进方向

高商业价值改进路线图

07

## 研究计划与建议

落地路径与资源需求

# 背景：Agentic Coding 正在改变软件开发

Claude Code 等 AI 编程工具已经能够自主完成复杂的软件开发任务。

但在企业环境中，"能写代码"只是第一步 - 更关键的是"写对的代码"。

## 需求失控

AI 容易自作主张添加/遗漏功能

需求只存在于聊天记录中，无法追溯

## 质量不稳定

同样的需求，不同对话产出差异大

缺乏验证标准，靠人肉 review

## 协作困难

一个人的 AI 对话上下文无法共享

多人并行修改容易冲突

## 知识流失

为什么做、怎么做的决策过程随对话消失

新人接手困难，重复踩坑

Claude Code 解决了"怎么写"，但没有解决"写什么"和"写得对不对"

# OpenSpec: AI 辅助开发的规格框架

OpenSpec 是一个轻量级的规格驱动框架，让人和 AI 在写代码之前先达成共识。

## 流动而非僵化

没有阶段门控，按需推进

## 迭代而非瀑布

边做边学，持续优化

## 增量而非全量

用 Delta 记录变更，不重写整体

## 存量优先

为已有项目设计，不是只给新项目

## 项目结构（两个核心目录）

```
openspec/
└── specs/           ← 当前行为的真相源
    └── <domain>/spec.md   (按领域组织的需求)
└── changes/          ← 提议的变更
    └── <change-name>/
        ├── proposal.md   (为什么改)
        ├── usecases.md    (用户场景)
        ├── specs/          (Delta: 增/改/删)
        ├── design.md       (怎么改)
        └── tasks.md        (实现清单)
└── archive/          (已完成变更的归档)
```

## 核心创新: Delta Spec (增量规格)

不是重写整个规格，而是只描述变化：

```
## ADDED Requirements   ← 新增需求
## MODIFIED Requirements ← 修改需求
## REMOVED Requirements ← 删除需求
```

每个需求附带 Given/When/Then 验证场景  
归档时自动合并到主规格 → 活文档

# OpenSpec 实际运行效果

## Dashboard 总览 (终端界面)

```
-zsh
> openspec view
OpenSpec Dashboard

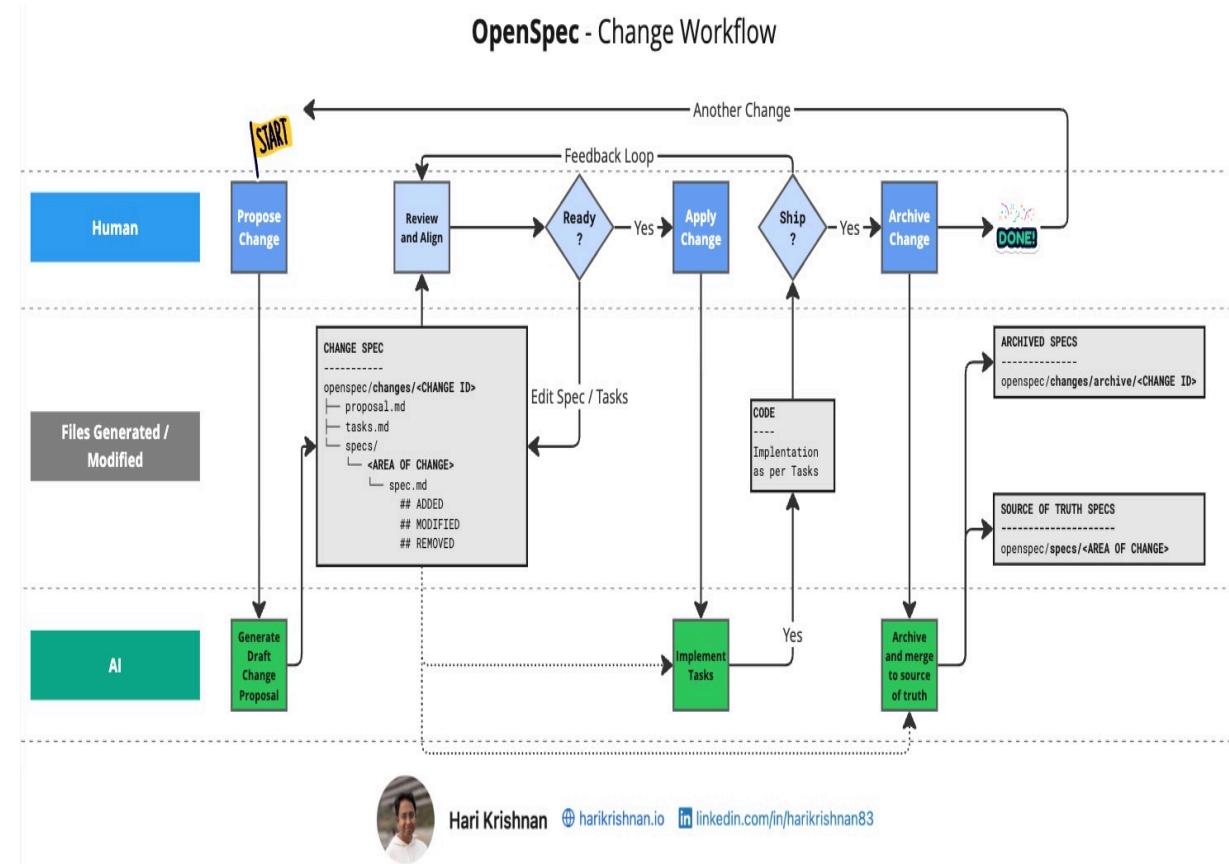
Summary:
  Specifications: 10 specs, 64 requirements
  Active Changes: 3 in progress
  Completed Changes: 4
  Task Progress: 30/41 (73% complete)

Active Changes
  make-validation-scope-aware [ ] 0%
  remove-diff-command [ ] 90%
  improve-deterministic-tests [ ] 92%

Completed Changes
  ✓ add-slash-command-support
  ✓ sort-active-changes-by-progress
  ✓ update-agent-file-name
  ✓ update-agent-instructions

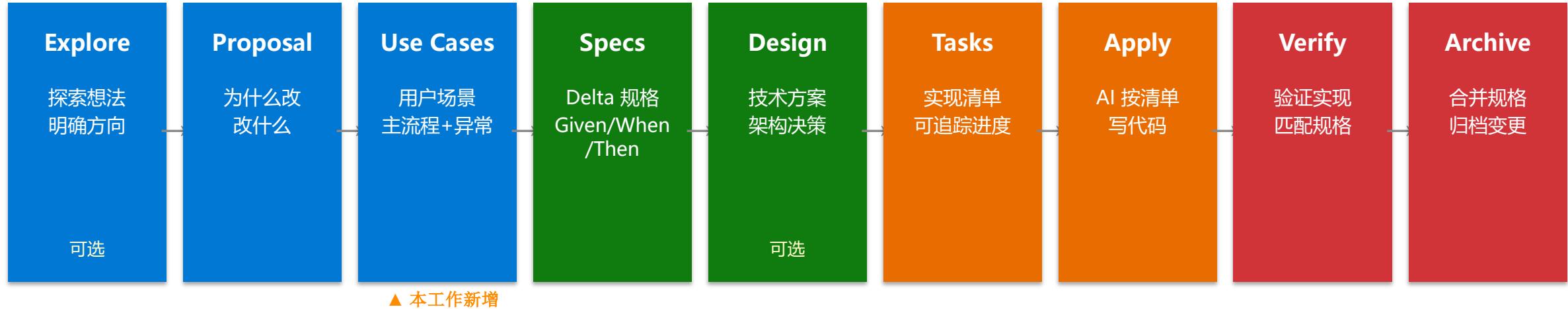
Specifications
  cli-archive          10 requirements
  openspec-conventions 10 requirements
  cli-validate         9 requirements
  cli-list             7 requirements
  cli-view             7 requirements
  cli-init             5 requirements
  cli-update           5 requirements
  cli-change           4 requirements
  cli-spec              4 requirements
  cli-show              3 requirements
```

## Change Workflow 全景 (人机协作流程)



左图：openspec view 命令展示项目规格状态、活跃变更和任务进度 | 右图：完整的 Propose → Review → Apply → Archive 工作流

# OpenSpec 工作流程：从想法到归档



一键加速

/opsx:ff 可以一步生成所有工件，省去逐步创建

灵活不僵化

工件之间是依赖图 (DAG)，不是固定流水线，可跳过可选步骤

闭环演进

Archive 后 specs 自动更新 → 下次变更基于最新状态 → 持续积累

# Use Cases 阶段详解（本研究新增）

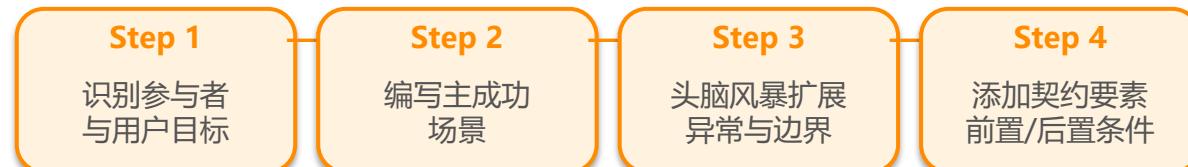


## 这一步做什么

在 Proposal (提案) 和 Specs (规格) 之间，插入一个结构化的用户场景建模环节。AI 不再从提案直接跳到技术规格，而是先系统地梳理：谁在用这个系统、他想达成什么目标、正常流程是什么、出错了怎么办。

## 基于什么方法论

采用 Alistair Cockburn 的《Writing Effective Use Cases》方法论，核心是“渐进式揭示”流程：



## 没有这一步会怎样？

- AI 只实现 Happy Path，异常场景大面积遗漏
- Specs 缺乏用例支撑，需求验收无据可依
- 30-50% 的返工源自需求阶段的遗漏和模糊

## 实测效果（量化交易系统）

### 4 层风控体系

止损、持仓限制、异常检测、合规检查

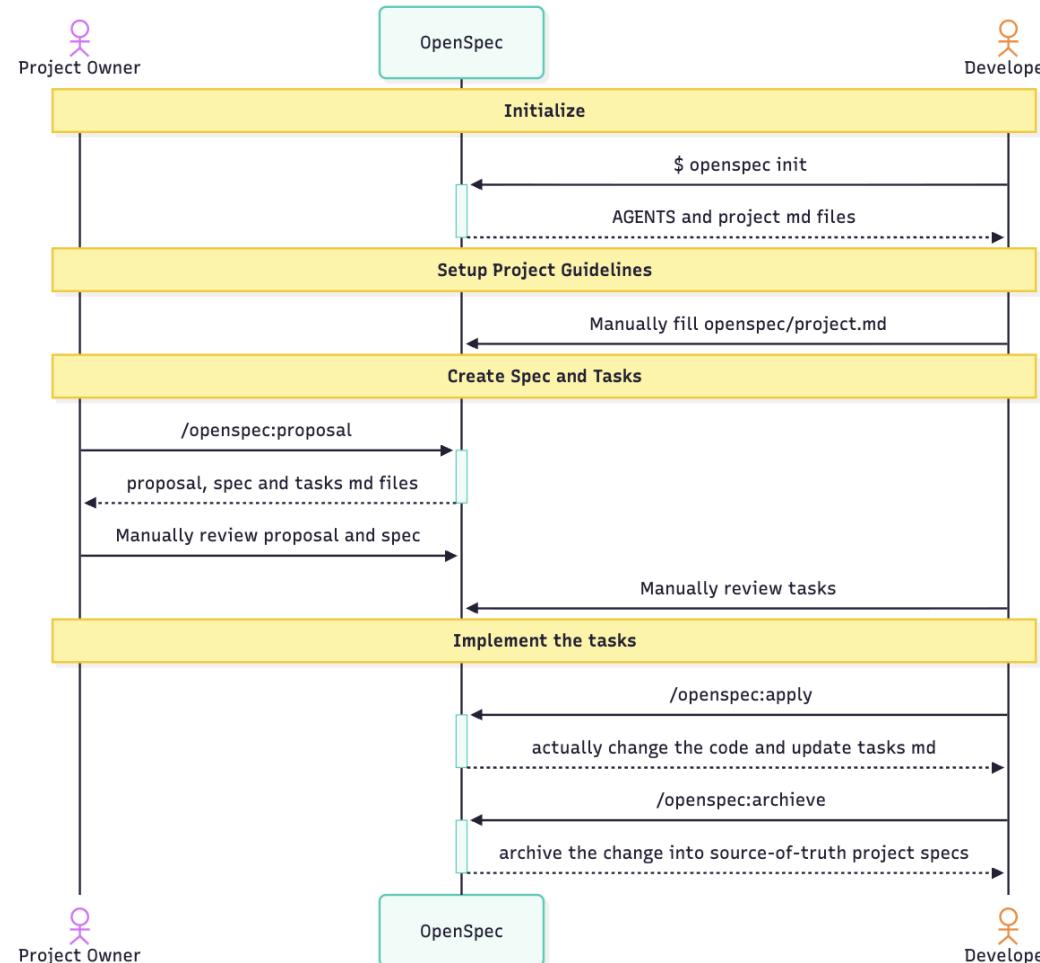
### 100+ 测试用例

覆盖主流程、异常分支、边界条件

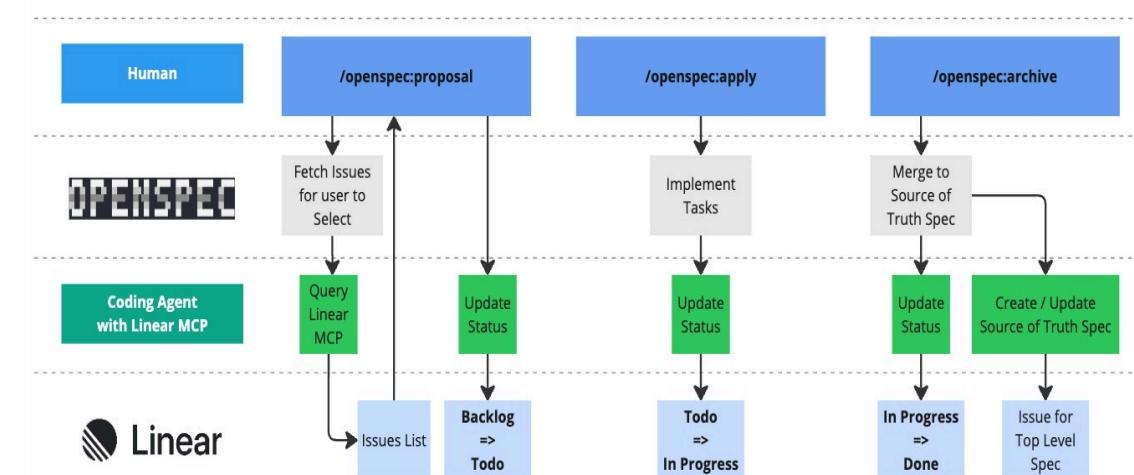
### 完整 A 股规则模拟

均源于 Use Cases 阶段对异常场景的系统梳理

# OpenSpec 协作模式: Project Owner / AI / Developer



## 企业级集成示例: Linear + OpenSpec



Hari Krishnan [@harikrishnan](https://twitter.com/harikrishnan) [linked](https://linkedin.com/in/harikrishnan83)

三方协作: Owner 定义需求 → OpenSpec 管理规格 → Developer/AI 实现

全程可追踪: 每个阶段的产出都是持久化的 Markdown 文件

可集成现有工具链: Linear、Jira、GitHub Issues 等项目管理工具

# 实测对比：同一需求，有/无 OpenSpec 的差异

测试项目：A股有色金属量化选股交易系统（相同需求描述，分别用两种方式让 Claude Code 实现）

对比维度	使用 OpenSpec	不使用 OpenSpec
代码总量	~3,900 行	~1,200 行
文件数量	40+ 文件，模块化设计	5 个文件，单体结构
设计模式	适配器模式、工厂模式、依赖注入	直接实现，全局变量
测试覆盖	18 个测试文件（单元+集成+端到端）	无测试
配置管理	外部 YAML 配置文件	Python 常量硬编码
错误处理	多数据源 fallback、显式异常处理	基本 try-catch
文档	8 个领域规格 + BDD 验证场景	仅代码内注释
可维护性	高 — 明确的模块契约	低 — 紧耦合

# 实测洞察：OpenSpec 不只是让代码变多，而是让代码变好

## 架构质量跃升

OpenSpec 的 specs 和 design 工件迫使 AI 在写代码前思考架构。结果：

- 多数据源适配器 vs 硬编码单一数据源
- 12 类配置外部化 vs Python 常量
- 模块间通过接口通信 vs 直接引用

## 测试从 0 到 18

任务清单 (tasks.md) 明确要求测试，AI 无法跳过。相比之下：

- OpenSpec: 18 个测试文件，覆盖单元测试、集成测试、端到端测试
- 无 OpenSpec: 0 个测试文件

## 可维护性差距

6 个月后需要修改因子计算逻辑：

- OpenSpec: 查 specs → 改 delta → 更新 tasks → 精确修改模块
- 无 OpenSpec: 在 1200 行单体文件里全局搜索，祈祷不破坏其他功能

## 知识沉淀

为什么选 AKShare 而非 Tushare？

- OpenSpec: design.md 记录了决策理由和替代方案的权衡
- 无 OpenSpec: 没人知道，当时的对话记录已经找不到了

结论：OpenSpec 的价值不在于写更多代码，而在于让 AI 写出工程级别的代码

# 进阶实验：原生 OpenSpec vs 改进版 OpenSpec

同一量化交易系统需求，分别使用原生 4-artifact 工作流和改进版 5-artifact 工作流（含 Usecases 步骤）构建，精确量化 Usecases 步骤的增量价值。

## 原生 OpenSpec (4 个 Artifact)



Specs 与 Design 并行生成，基于 Proposal  
从提案直接跳到技术规格，无用户场景建模

## 改进版 OpenSpec (5 个 Artifact, 含 Usecases)



▲ 本工作新增  
线性依赖：Design 基于 Specs (而非 Proposal)  
Usecases 提供用户交互场景，驱动更完整的 Specs

## 实验条件

### 需求完全相同

A股有色金属多因子量化交易系统（16因子、5大类、完整风控、事件驱动回测）

### AI 工具相同

Claude Code + OpenSpec CLI，均使用 /opsx:propose 和 /opsx:apply 命令驱动

### Schema 差异

原生版移除 Usecases 步骤并恢复 Specs/Design 并行依赖；改进版保持线性 5-artifact 流程

**核心目标：**通过控制变量实验，精确量化 Usecases 步骤对最终代码质量、测试覆盖、架构设计的因果影响

# 产出规模对比：Usecases 的量化放大效应

同一需求，仅因多了一步 Usecases 场景建模，最终产出在三个层面均显著增长

## 规格与设计层

Artifact 产出

Artifact 数量	原生: 4 个	改进: 5 个	+1 <small>(1User)</small>
<b>Design 深度</b> 原生: 93 行 / 6 决	改进: 255 行 / 8 决	+174 %	
<b>任务分解</b> 原生: 44 个任务	改进: 95 个任务	+116 %	

Usecases 梳理了 8 个用户操作场景，迫使 Specs 覆盖更多边界条件，Design 做出更多架构决策，Tasks 分解更加细致。

## 代码实现层

源码产出

源码总量	原生: 2,824 行	改进: 5,015 行	+78%
<b>源码文件</b> 原生: 35 个	改进: 43 个	+23%	
<b>设计模式</b> 原生: 简单类/函数	改进: Protocol + 装饰器	升级	

Usecases 中的场景复杂性传导至代码：  
数据适配需要 Protocol 统一接口，  
插件扩展需要注册表模式，  
风控聚合需要 Facade 模式。

## 质量保障层

测试产出

测试文件	原生: 8 个	改进: 13 个	+63%
<b>集成测试</b> 原生: 无	改进: 195 行完整集	新增	
<b>性能指标</b> 原生: 7 个	改进: 13+ 个	+86%	

Usecases 的异常场景直接转化为测试用例的边界条件，集成测试验证完整用户流程，更多指标衡量系统质量。

通用规律：Usecases 的前置场景建模产生了 Artifact 链路上的级联放大效应 — 这一规律不依赖于具体领域

# 架构设计差异：Usecases 驱动的模式升级

改进版的 8 个 Usecases 场景系统性地推动了更高级的设计模式和更完善的架构抽象

## 数据源抽象

Usecases: 更新市场数据

### 原生

原生：直接类封装  
class AKShareAPI:  
 def fetch\_stock\_price(...)

### 改进

改进：Protocol 适配器模式  
class DataSource(Protocol):  
 def fetch\_stock\_daily(...)  
 + @\_retry 装饰器 + 统一接口

## 因子系统

Usecases: 计算因子值

### 原生

原生：独立函数  
def momentum(data, window):  
 return ...  
手动组合各因子函数

### 改进

改进：注册表 + 抽象基类  
@register\_factor  
class MomentumFactor(BaseFactor):  
全局 \_FACTOR\_REGISTRY 管理

## 回测执行

Usecases: 运行策略回测

### 原生

原生：OrderQueue 内嵌  
T+1/涨跌停逻辑混在  
process\_orders() 中

### 改进

改进：独立 Broker 抽象  
class SimulatedBroker:  
 execute\_buy() / execute\_sell()  
含滑点、整手、印花税分离

## 风控体系

Usecases: 执行每日风控检查

### 原生

原生：独立函数调用  
无聚合入口  
各风控逻辑分散调用

### 改进

改进：聚合风控入口  
run\_daily\_risk\_check()  
+ 波动率调仓 + 最大回撤持续天数

每一项架构升级都可追溯到 Usecases 中具体的用户操作场景 — Usecases 是设计模式选择的「需求驱动力」

# 核心发现：Usecases 步骤的级联传导效应

Usecases 对最终代码的影响不是线性的，而是通过 Artifact 链路逐级放大



## 关键量化指标对比

任务数  
原生: 44

改进: 95

+11  
6%

代码总量  
原生: 2,824 行

改进: 5,015 行

+78  
%

测试文件  
原生: 8 个

改进: 13 个

+63  
%

设计模式  
原生: 简单类/函数

改进: Protocol+装饰器

升级

性能指标  
原生: 7 个

改进: 13+ 个

+86  
%

图表类型  
原生: 3 种

改进: 5 种

+67  
%

## 改进版优势

系统更完整、边缘覆盖更全面  
架构可扩展性强（因子注册表、适配器模式）  
13+ 性能指标 + 5 张图表，接近生产级  
集成测试 + conftest 共享 fixture

## 原生版优势

实现速度更快（artifact 更少）  
代码更简洁、易读、易理解  
适合原型验证和小型项目  
上手门槛低，学习成本小

# 实验结论：何时使用 Usecases 步骤

Usecases 不是“更多工作”，而是“更聪明的工作” — 在正确的场景下，它是 ROI 最高的前置投入

## 推荐使用改进版（含 Usecases）

- 多模块系统（跨 3+ 模块的交互）
- 领域复杂度高（金融合规、医疗、安全关键）
- 团队协作（多人参与、需要知识传承）
- 长期维护项目（预期 > 6 个月迭代）
- 边界场景重要（错误处理影响业务）
- 需要完整测试覆盖和可追溯性

## 原生版足够（不含 Usecases）

- 小型工具或脚本（单一功能）
- 快速原型验证（POC / MVP）
- 内部工具（低复杂度、少用户）
- 探索性项目（需求不确定，快速迭代）
- 单人项目（无团队传承需求）
- 技术组件（库、SDK、无用户交互）

## 本实验量化结论

### 代码量

**+78%**

2,824 → 5,015 行

### 任务粒度

**+116%**

44 → 95 个任务

### 架构模式

**升级**

简单类 → Protocol + 装饰器

### 边缘覆盖

**显著增强**

+前视偏差防护、整手处理、微小再平衡跳过

结论：Usecases 步骤通过级联放大效应，以 285 行用例文档的前置投入，换来了 78% 的代码增量和显著的架构质量提升

# 需求变更实验：OpenSpec 的迭代维护优势

前面展示了初始构建的对比。但软件 80% 的成本在维护阶段 — 需求变更时，两种方式差距更大。

## 实验场景

2024 年有色金属市场剧烈波动，铜价暴涨暴跌。

客户反馈：固定参数风控策略反应太迟钝，多次在大幅回撤后才触发止损。产品经理提出 3 项需求变更。

## 三项需求变更

1. 波动率自适应止损  
低波动收紧 → 高波动放宽，避免固定参数的迟钝
2. 动态再平衡频率  
正常月度 → 高波动双周 → 极端波动周度
3. 极端行情熔断机制（全新功能）  
单日跌3%减半仓 | 跌5%全清仓 | 恢复需连续回升

## OpenSpec 版方法

proposal → usecases (10个场景) → delta specs  
→ design → tasks (10项) → 逐项实现代码

生成 6 个 artifact 文档 (636行)，确保编码前完成影响分析和边界场景识别

## 非 OpenSpec 版方法

直接拿到需求文档 → 让 AI 编码实现  
不使用任何规格框架

模拟最常见的“拿到需求直接写代码”方式  
无 spec、无 design、无 task 分解

关键选题：跨风控/回测/策略/配置 4 个模块，同时涉及修改现有功能和新增全新功能，最能体现迭代维护的复杂性

# 需求变更实验结果：量化对比

对比维度	OpenSpec 版	非 OpenSpec 版	倍数
边界场景覆盖	10 个边界场景 (含 T+1 冲突、熔断恢复中断、频率切换等)	2 个边界场景 (数据不足、熔断升级)	5x
测试用例	27 个新增测试 (3 个测试文件, pytest 框架)	0 个测试 (无任何自动化测试)	∞
变更文档	636 行 artifact 文档 (proposal→usecases→specs→design→tasks)	0 行文档 (无 proposal、无 design、无记录)	∞
代码模块化	7 个文件 (2 新模块 + 3 测试 + 2 个修改)，单一职责	2 个文件，所有逻辑挤在 backtester.py	3.5x
新增代码行	662 行 (源码 303 + 测试 359) 分布在独立模块	163 行 全部混在主函数	4x
遗漏的关键边界	0 个	3 个 (T+1冲突、渐进恢复、频率切换间隔)	—
可追溯性	完整链路：需求→场景→规格 →设计→任务→代码	仅 git diff 6个月后无法理解变更原因	—

OpenSpec 的 usecases 强制思考边界场景，发现了非 OpenSpec 版遗漏的 3 个关键边界：T+1 冲突、渐进式恢复、频率切换时机

# 需求变更实验：核心发现与评分

OpenSpec 的核心价值不在于初始构建，而在于持续迭代 — 需求变更时差距更加显著

## 七维评分对比

影响分析准确性	★★★★★	★★★★★
需求覆盖完整性	★★★★★	★★★★★
代码变更模块化	★★★★★	★★★★★
测试覆盖	★★★★★	★★★★★
文档/可追溯性	★★★★★	★★★★★
回归风险	★★★★★	★★★★★
可复现性	★★★★★	★★★★★
<b>总分：</b>	<b>35/35</b>	<b>vs 13/35</b>

## OpenSpec 在变更场景的三大核心优势

### 前置思考 — 编码前完成需求分析

proposal → usecases → specs 的过程强制在写代码前识别出 10 个边界场景，避免了 3 个关键遗漏

### 知识沉淀 — 636 行文档是团队资产

完整的 proposal→specs→design→tasks 链路  
6个月后仍可追溯每个变更的动机和设计决策

### 质量保障 — tasks 确保测试不被遗忘

tasks.md 显式列出 3 个测试文件 + 27 个测试用例  
非 OpenSpec 版：0 测试，无法验证任何逻辑正确性

实验完整数据见 testing\_project\_quant\_requirement\_changing\_experiment/comparison/result.md

# 企业级价值：为什么公司需要 OpenSpec

AI 编码工具解决了「怎么写代码」，但没有解决「怎么管理 AI 写的代码」——OpenSpec 填补这一空白。

## 规范化

让 AI 编码有章可循

### 统一工作流标准

所有团队遵循 proposal → spec → design → task → implement 流程，AI 不再各行其是

### 需求有据可查

每个变更都有 proposal (为什么改)、spec (改什么)、design (怎么做)，不再是一句聊天记录

### 验收有标准可依

BDD 场景 (Given/When/Then) 提供明确的验收条件，代码好不好有客观依据

## 可追溯

每一步决策都有据可查

### 变更全链路追踪

从需求到设计到任务到代码，每一步都有文档记录，出了问题能快速定位根因

### 审计友好

归档机制保留完整上下文，满足合规要求；监管审查时能拿出完整的决策链条

### 决策可回溯

design.md 记录技术选型理由和替代方案权衡——三个月后还能看懂当初为什么这么选

## 可协作

团队协同不再靠喊话

### 多人并行不冲突

Delta spec 机制让不同变更可以并行推进，各自只描述自己的增量，合并时自动检测冲突

### 上下文可共享

规格文件是团队共享资产，不是某个人的 IDE 聊天记录；任何人都能读懂系统全貌

### 新人可快速接手

读 specs + design + archived changes 即可理解系统演化历程，不用再问“这段代码为什么这样写”

**一句话总结：**OpenSpec 让 AI 编码从“个人即兴创作”变成“团队工程协作”——代码有规范、变更有记录、协作有秩序。

# 商业价值：Agentic Coding 治理是蓝海市场

## 市场趋势

AI 编程工具正从"辅助"走向"自主" (Agentic)

- Cursor, Windsurf, Claude Code 等工具快速普及
- 企业采用率快速增长，但治理和规范严重缺失
- **当前阶段类比：**代码版本管理出现前的混乱期
- OpenSpec 定位 = Agentic Coding 时代的 Git

## 为什么现在值得研究

- OpenSpec 仍处于早期阶段，先发优势明显
- 与 Claude Code 深度集成，Anthropic 生态优势
- 开源项目，可以基于自身需求定制和贡献
- 规格驱动思想与现有 DevOps 体系天然兼容
- 掌握核心方法论 → 可输出咨询/工具/培训服务

## 商业化方向

### 企业内部效率提升

统一 AI 编码规范  
减少返工、提升交付质量  
降低代码审查成本

### 工具产品化

基于 OpenSpec 的企业级  
管理平台 (Web UI、审批  
流程、数据统计)

### 咨询与培训

Agentic Coding 治理方法论  
企业落地咨询、团队培训  
最佳实践输出

### 生态贡献

自定义 Schema 市场  
行业特定工作流模板  
插件和集成开发

# ROI 分析：规范化带来的投资回报

## Without OpenSpec (当前状态)

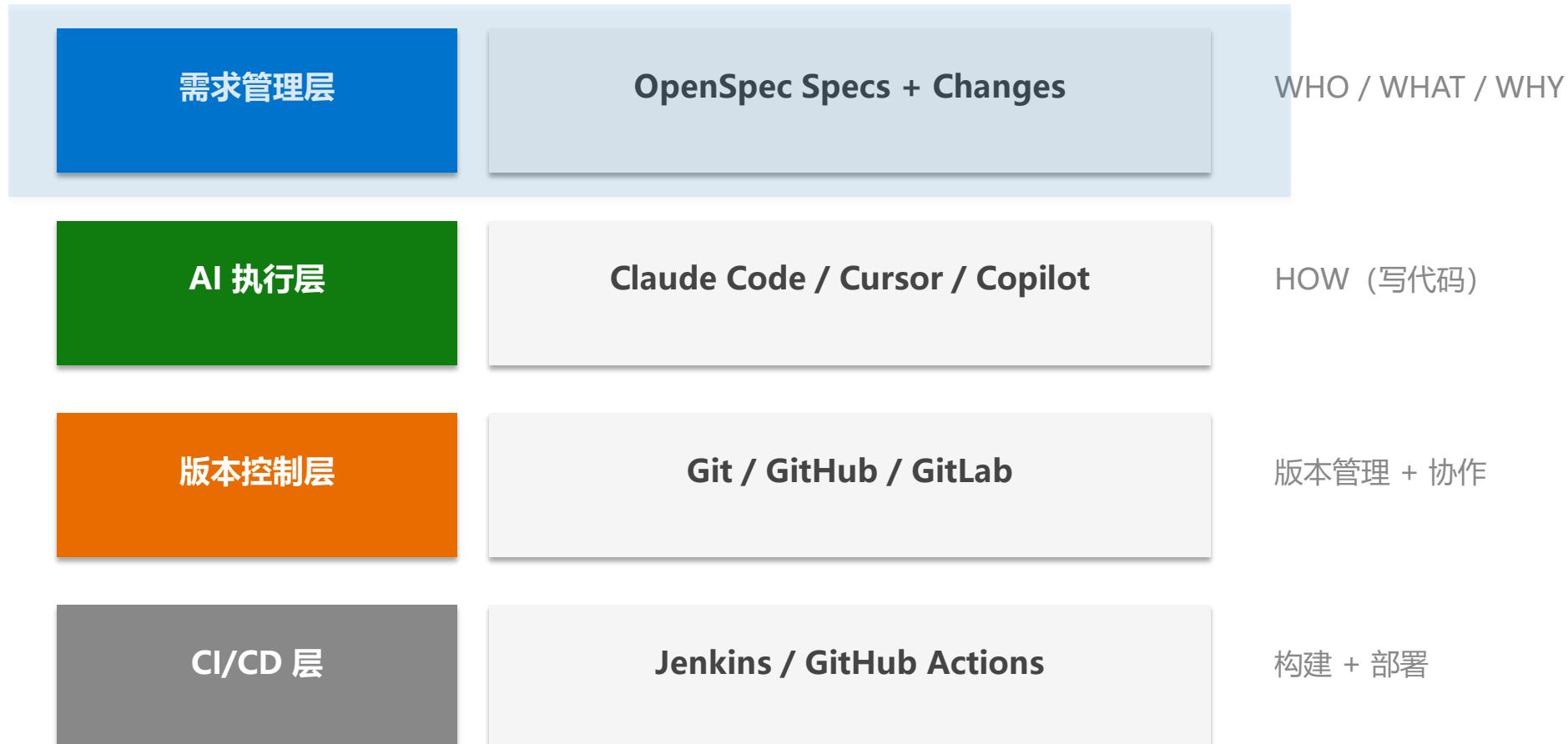
- ✗ 需求理解偏差率高 → 返工率 30-50%
- ✗ 无测试 → 上线后 bug 修复成本高 (10-100x)
- ✗ 代码审查耗时 → 审查者需从头理解意图
- ✗ 知识流失 → 新人上手周期长 (2-4 周)
- ✗ 多人协作冲突 → 集成时间占 20-30%
- ✗ 无法审计 → 合规风险

## With OpenSpec (目标状态)

- ✓ 规格前置 → 返工率降至 10-15%
- ✓ 任务清单强制测试 → bug 在开发阶段发现
- ✓ 审查者读 proposal + spec → 审查效率提升 50%
- ✓ 活文档 + 归档 → 新人上手周期缩短至 3-5 天
- ✓ Delta spec 隔离 → 并行开发无冲突
- ✓ 全链路追踪 → 天然满足审计需求

# 定位：OpenSpec 与现有工具链的互补关系

OpenSpec 不是替代品，而是缺失的一环 - 填补 AI 编程的"治理真空"



**OpenSpec 管理的是 AI 的"输入质量" - 输入越精确，AI 的输出越可控**

# 学术前沿：2023-2026 五大宏观趋势

大量顶级学术研究与产业实践正在验证 OpenSpec 所处赛道的战略价值

## 从自动补全到自主 Agent

- 三代演进：补全(2021) → 对话(2023) → Agent(2025)
- SWE-bench 解题率：个位数% → 79.2% (3年)
- 多 Agent 协作成为 2026 年前沿方向

## 规格驱动开发强势回归

- Amazon Kiro (2025.7) —— 首个 SDD IDE
- GitHub Spec Kit / Tessl / OpenSpec 并行发展
- arXiv 2026 论文：定义 SDD 三成熟度模型

## 需求工程被 LLM 深度变革

- LLM4RE 论文量：4篇(2022) → 113篇(2024)
- GPT-4 生成需求：速度快 720x, 成本仅 0.06%
- 挑战：幻觉、领域推理、复杂依赖

## 形式化方法与 LLM 融合

- Vericoding 验证成功率 68% → 96%
- Req2LTL / nl2spec：自然语言→时序逻辑
- 预言：AI 将使形式化验证走向主流

## 基准测试快速成熟

- SWE-bench 扩展：多语言/Windows/Pro 版本
- SWE-bench Illusion 论文揭示记忆化问题
- 推动更严格、更真实的评估体系

# 竞争格局：OpenSpec 的差异化定位

维度	Amazon Kiro	GitHub Spec Kit	Tessl	OpenSpec
开发方	Amazon (AWS)	Microsoft (GitHub)	Tessl	Fission AI (开源)
定位	AI IDE (封闭生态)	CLI 模板工具	Agent 上下文平台	跨工具规格框架
支持工具	仅 Kiro IDE	GitHub Copilot	多 Agent	20+ AI 工具
Delta Spec	不支持	不支持	不支持	核心创新
工件依赖图	线性流程	无	无	DAG 拓扑排序
并行变更	不支持	不支持	有限	完整支持
规格质量检查	无	无	无	可扩展 (改进点)
开源/可定制	闭源	模板开源	商业产品	完全开源

OpenSpec 的核心壁垒：跨工具兼容 + Delta Spec + DAG 依赖图 —— 竞品均未实现这三者的组合

# 六大高商业价值的研究改进方向

## 1 Spec Quality Gate

在 spec 生成的每个环节植入质量检查  
歧义检测 + 冲突分析 + 覆盖率评估

无直接竞品，核心差异化  
返工率降低 30-50%

## 4 Platform / Web Hub

CLI → Web 平台：可视化 artifact graph  
团队 spec review + 质量 dashboard

CLI(免费) → Platform(付费)  
最自然的 SaaS 商业化路径

## 2 Spec-Code Traceability

建立 Spec ↔ Code 双向追溯链  
代码变更后自动检测 spec drift

EU AI Act 2026.8 合规刚需  
进入金融/医疗市场的前置条件

## 5 Domain-Specific Packs

医疗/金融/汽车行业预配置  
合规 rules + spec 模板 + schema

Vertical SaaS 策略  
每个行业 = 独立收入 + 高粘性

## 3 Multi-Agent Orchestration

基于现有 DAG 拓扑排序拆分 task  
多 Agent 并行 + artifact 状态协调

已有 DAG 架构，实现成本低  
效率提升 3-4x，支撑团队级定价

## 6 Formal Verification

Given/When/Then → 形式化断言  
编译期自动验证代码满足 spec

安全关键场景的极高客单价  
Vericoding 成功率已达 96%

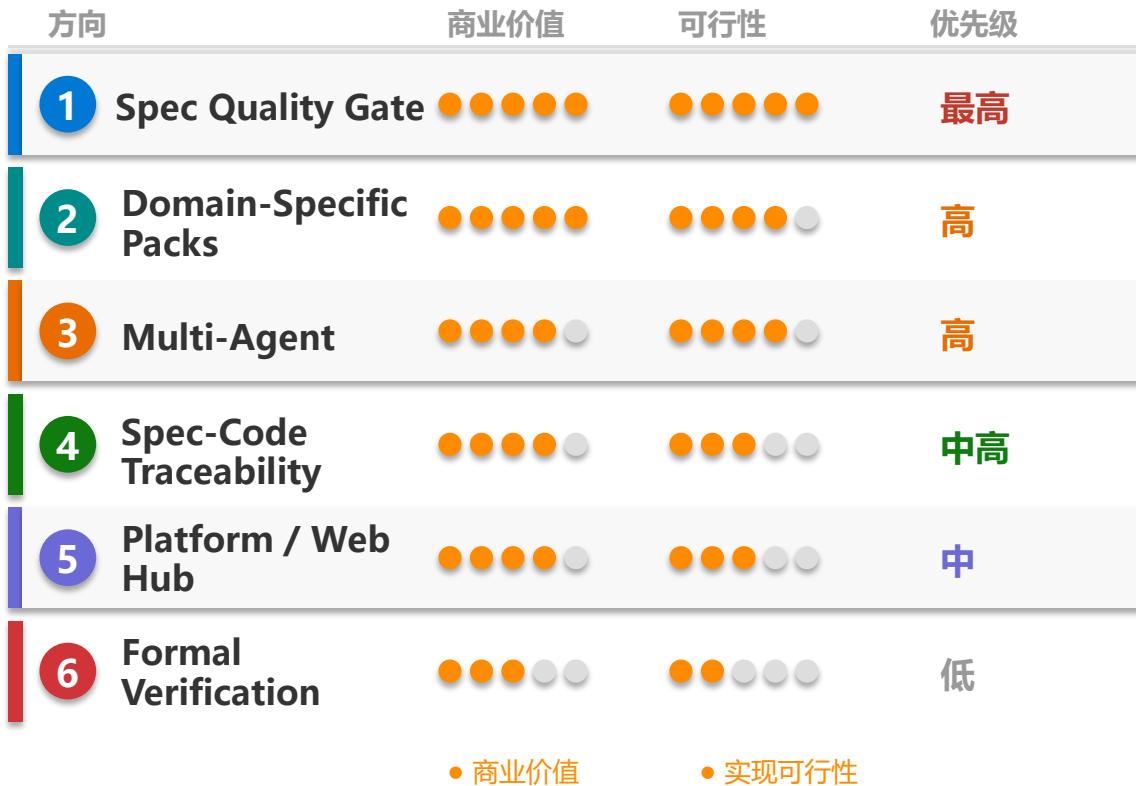
# 优先投入的四个方向

按见效速度 × 实现成本 × 竞争壁垒排序

<b>Spec Quality</b> <b>Problem</b> PM 写"优化体验"就丢给 AI AI 按自己理解写 3 天 → 全部返工 需求模糊导致返工占开发 30-50%	<b>短期 1-3 月</b>	<b>Domain-Specific</b> <b>Problem</b> AI 不懂行业术语和合规要求 每个项目重复教育 AI 同样的领域常识 生成的 spec 缺乏行业深度	<b>短期 1-3 月</b>	<b>Multi-Agent</b> <b>Problem</b> 20 个 tasks 只能单 Agent 串行 端到端耗时 4 小时 无依赖的 task 本可并行却在等待	<b>中期 3-6 月</b>	<b>Spec-Code Traceability</b> <b>Problem</b> 迭代半年后 spec 与代码严重 drift 新人问 design decision, 无人能答 合规审计拿不出追溯证据	<b>长期 6-12 月</b>
<b>Solution</b> proposal → spec 转换时植入 Gate · 歧义检测：拒绝模糊需求 · 冲突分析：与 main specs 交叉比对 · 覆盖评估：输出 Quality Score		<b>Solution</b> 将领域知识打包为 Domain Packs： · 行业术语表 + 规则模板 · 合规清单（金融/医疗/汽车） · AI 生成 spec 时自动引用		<b>Solution</b> 利用 OpenSpec 已有 DAG： · 拓扑排序识别可并行 task 分组 · 每组独立 Agent + worktree · 通过 artifact 状态隐式协调		<b>Solution</b> 构建双向 traceability 链： · Spec → Code：标注实现位置 · Code → Spec：变更触发 drift 告警 · Living Specs：规格随代码演进	
<b>Business Value</b> Kiro / Spec Kit 均无此能力 返工率 40%→15%，直接省人力 基于 LLM + 规则，1-3 月可 MVP		<b>Business Value</b> 领域知识 = 护城河，竞品难复制 垂直行业客单价高，商业价值大 先验知识沉淀 = 可售卖数据资产		<b>Business Value</b> DAG 架构已就绪，边际成本低 效率 3-4x → 最直观的卖点 个人版 \$20 → 团队版 \$200/人/月		<b>Business Value</b> EU AI Act 2026.8 强制 traceability 医疗/金融/政府 = 高客单价市场 持续监控 = SaaS 订阅 → 稳定 MRR	

# 改进优先级矩阵与实施路线图

优先级排序 (价值 × 可行性)



建议实施路线图

## 短期 (1-3月)

- 实现 Spec Quality Gate (质量关卡)
- 集成 ISO 29148 需求质量评估
- 启动 Domain Packs 基础框架搭建
- 快速验证核心差异化价值

## 中期 (3-6月)

- 首批 Domain Packs 落地 (金融/医疗)
- 多 Agent 编排层 (基于现有 DAG)
- Spec-Code 基础追溯链
- 原型 Web Dashboard

## 长期 (6-12月)

- 完整 Drift Detection + Living Specs
- Domain Packs 扩展 (汽车/合规等)
- 形式化验证集成探索
- 平台化 + 企业级功能

核心策略：Spec Quality Gate 快速建立差异化，Domain Packs 构建护城河——先证明 spec 治理的价值，再通过领域知识形成不可替代性。

# 总结

## Agentic Coding 已是趋势

Claude Code 等工具让 AI 自主写代码成为现实，但企业缺少治理手段

## OpenSpec 填补治理空白

在"AI 能写代码"和"AI 写对代码"之间架起桥梁，用规格驱动 AI 行为

## 实测验证：质量提升显著

初始构建：3.26x 模块化 | 需求变更：边界覆盖 5x、测试 27:0、可追溯文档 636:0 行

## 商业价值：蓝海赛道

先发优势 + 开源可定制 + 咨询/工具/培训多种变现路径

## 建议：立项研究

10 周、低成本投入，高潜力回报

# 附录：量化交易系统深度技术对比

---

A股有色金属多因子量化交易系统 | 同一需求 · 两种实现 · 逐项拆解

OpenSpec 版：30+ 源码文件 · 16 个因子 · 100+ 测试用例 · CLI 工具

无 OpenSpec 版：6 个源码文件 · 15+ 因子 · 0 测试 · Flask Web 仪表盘

# 架构对比：企业级分层 vs 快速原型

## OpenSpec 版 — 6层架构 • 30+ 文件

**数据层**  
src/data/

多数据源适配器 (AKShare + BaoStock 自动切换)  
SQLite 增量更新 + 数据验证器

**因子层**  
src/factors/

注册器模式，16 个因子通过 @register 自注册  
MAD 去极值 + Z-Score 标准化

**策略层**  
src/strategy/

多因子评分 + IC 加权 + 约束优化分配  
商品动量择时 (铜铝联动信号)

**风控层**  
src/risk/

ATR 动态硬止损 + 追踪止损 + 分层回撤管理  
金属暴跌联动预警

**回测层**  
src/backtest/

事件驱动引擎 + A股规则模拟器  
T+1 / 涨跌停 / 停牌 / 整手 / 印花税

**报告层**  
src/report/

Plotly 交互图表 + HTML/PNG 双格式输出

## 无 OpenSpec 版 — 扁平结构 • 6 文件

**配置**  
config.py

Python 常量 · 硬编码股票池 · 无外部配置文件

**数据**  
data\_fetcher.py

单一 AKShare 源 · TTL 缓存 · 无数据验证

**因子**  
factor\_engine.py

所有因子挤在一个文件 · Z-Score 标准化  
无注册机制，添加因子需改多处代码

**策略**  
strategy.py

阈值判断生成信号 · 等权分配  
无择时、无约束优化

**回测**  
backtester.py

简化回测 · 因子计算与实时信号不一致  
无涨跌停/停牌/T+1 限制模拟

**界面**  
app.py

Flask Web 仪表盘 · ECharts 图表  
暗色主题 · 参数在线调整 (亮点)

OpenSpec 的 specs 和 design 工件在写代码前就定义了模块边界，AI 被迫按契约分层实现，而非一股脑堆在一起

# 风控对比：量化交易的生死线

在量化交易中，风控能力决定了系统能否在真实市场中存活。这是两个版本差距最大的地方。

## OpenSpec 版：4 层风控体系

### 第1层：ATR 动态止损

根据近期波动率自动调整止损位  
波动大 → 止损宽（避免误杀）  
波动小 → 止损窄（保护利润）

### 第2层：追踪止损

股价上涨 10% 后激活  
从最高点回撤 8% 则卖出  
锁定盈利，让利润奔跑

### 第3层：组合回撤管理

回撤 15% → 仓位砍半（减少暴露）  
回撤 20% → 全部清仓（保命）  
分层响应，不会一刀切

### 第4层：金属暴跌联动

监控铜/铝期货日跌幅 > 3%  
自动识别受影响的持仓股  
触发黄金对冲机制

## 无 OpenSpec 版：基础止损

### 固定止损 -8%

不管市场波动大小，统一 -8% 触发  
牛市中容易被正常回调误杀  
熊市中 -8% 可能仍然太慢

### 回撤预警 -15%

仅发出预警，没有自动减仓动作  
需要人工介入处理  
晚间/周末无法及时响应

### 缺失的风控能力

无动态止损 — 无法适应不同市场环境  
无追踪止损 — 赚到的利润无法锁定  
无自动减仓 — 回撤时只能眼睁睁看着  
无联动预警 — 上游商品崩盘时毫无防备

# 回测对比：你的回测结果能信多少？

回测是量化策略上线前的最后一道关卡。回测越贴近真实，上线后的意外就越少。

A股交易规则	OpenSpec 版	无 OpenSpec 版
T+1 交割制度	完整模拟：今日买入明日才能卖	未实现：买卖当日即可完成
涨跌停板限制	涨停 +10% 不可买入，跌停不可卖出	未实现：任意价格均可成交
停牌处理	成交量=0 自动冻结，不可交易	未实现：停牌期间仍可交易
整手交易（100股）	强制 100 股倍数买入	未实现：可买任意股数
交易费用建模	印花税 0.05% + 佣金 0.03% + 滑点 0.15% (分别建模)	合并费率 0.025% (欠精确)
因子计算一致性	回测与实盘使用同一因子引擎	回测中重新实现了简化版因子 与实时信号逻辑不一致
仓位分配	得分加权 + 单股 10% + 行业 25% 上限约束 + 择时调仓比例	等权分配 无约束优化

无 OpenSpec 版的回测忽略了大量 A 股特有规则，回测收益率会虚高，上线后会因为 T+1 无法卖出、涨停买不进等问题导致实际表现大幅偏离

# 综合评分：8个维度逐项打分

维度	OpenSpec	无OpenSpec	差距	OpenSpec 亮点	无OpenSpec 亮点
架构设计	9/10	7/10	+2	分层清晰，模块间通过接口通信	扁平直接，够用但难以扩展
风控完备性	10/10	6/10	+4	4 层风控 + 联动预警	仅固定止损 + 预警
回测真实性	9/10	6/10	+3	完整模拟 A 股交易规则	简化模拟，结果可信度低
测试覆盖	9/10	1/10	+8	100+ 测试用例，pytest 框架	零测试
可扩展性	8/10	4/10	+4	注册器模式，新增因子只需一个类	添加因子需改多处代码
配置管理	8/10	5/10	+3	外部 YAML，12 类参数可调	Python 常量硬编码
用户体验	6/10	8/10	-2	CLI + HTML 报告	Web 仪表盘 + 在线调参
文档质量	8/10	7/10	+1	README + 类型注解 + BDD 场景	中文注释 + docstrings
总分	67/80	44/80	+23	OpenSpec 版总分领先 23 分，唯一弱项是缺少 Web 界面	33/48

# 结论：OpenSpec 让 AI 写出了工程级别的量化系统

**OpenSpec 版在 8 个维度中 7 个领先，总分 67/80 vs 44/80**

不是代码写得多就好 — 而是规格约束让 AI 在正确的方向上写出高质量代码

## 规格前置 = 架构质量

OpenSpec 强制 AI 在写代码前定义模块边界、接口契约和数据流。

结果：6 层清晰架构，每层职责单一，模块间通过 Protocol 接口通信。

对比：无 OpenSpec 时 AI 把所有逻辑堆在 6 个文件里，改一处要查全局。

## 异常场景 = 风控完备

OpenSpec 的 usecases.md 要求定义异常流程，specs 要求 Given/When/Then 验证。

结果：4 层风控体系，覆盖个股止损、组合回撤、品种联动等完整场景。

对比：无 OpenSpec 时 AI 只实现正常流程的 happy path，风控形同虚设。

## 任务清单 = 测试护城河

OpenSpec 的 tasks.md 把测试作为显式任务项，AI 必须逐项完成。

结果：100+ 测试用例，覆盖因子计算、策略逻辑、风控触发、回测指标等。

对比：无 OpenSpec 时 AI 认为「功能写完就是完成」，测试数量为 0。

# 附录 1-B：原生 vs 改进版 OpenSpec 深度技术对比

控制变量实验：同一量化交易需求，仅变更 Schema (移除/保留 Usecases 步骤)，逐项拆解产出差异

## 原生 OpenSpec

Schema: native-openspec (本地 fork)

工作流: proposal → specs + design (并行) → tasks

Artifact 数量: 4 个

源码总量: ~2,824 行 / 35 文件

测试代码: 600 行 / 8 文件 / ~74 测试方法

任务数: 44 个 (10 组)

design.md: 92 行, 6 个架构决策

Usecases 步骤的直接产出

UC-1: 更新市场数据

驱动: 数据验证层 (validators.py 105行)、多源 fallback 重试机制

UC-2: 管理股票池

驱动: Point-in-time universe、get\_point\_in\_time\_universe() 防前视偏差

UC-3: 计算因子值

驱动: @register\_factor 装饰器注册表、BaseFactor 抽象基类

UC-4: 生成交易信号

驱动: 独立 signal.py (153行)、BUY/ADD/REDUCE/SELL 信号分类、跳过微小再平衡

UC-5: 执行每日风控

驱动: run\_daily\_risk\_check() 聚合入口、adjust\_weights\_by\_volatility()

UC-6: 运行策略回测

驱动: 独立 SimulatedBroker 类 (169行)、TradeRecord with P&L、整手交易

UC-7: 查看绩效报告

驱动: 因子热力图、IC 跟踪图、Sortino ratio、最大回撤持续天数

UC-8: 配置策略参数

驱动: override\_ratio 择时覆盖、skip\_rebalance 阈值 (<2%)

## 改进版 OpenSpec (含 Usecases)

Schema: spec-driven (官方修改版)

工作流: proposal → usecases → specs → design → tasks

Artifact 数量: 5 个 (+usecases.md 285 行)

源码总量: ~5,015 行 / 43 文件

测试代码: 695 行 / 13 文件 / 含集成测试

任务数: 95 个 (10 组)

design.md: 255 行, 8 个架构决策

# 设计模式升级：Usecases 驱动的三类架构演进

Usecases 中的用户场景复杂性，自然推动了从简单实现到成熟设计模式的升级 — 这些模式在任何领域通用

## 适配器模式 (Adapter / Protocol)

### Usecases 场景

用户更新数据时，主数据源不可用，  
系统应自动切换备用源并记录日志

### 原生版做法

每个数据源一个独立类  
调用方硬编码优先级列表  
添加新源需修改调用方代码

### 改进版做法

定义 Protocol 统一接口  
适配器封装各数据源差异  
添加新源仅需实现接口

软件工程原理：依赖倒置原则 (DIP) —— 高层模块  
不应依赖低层实现细节

## 注册表模式 (Registry / Decorator)

用户计算因子值时，系统应自动发现  
所有已注册的因子并统一计算

### 原生版做法

每个因子是独立函数  
手动在入口处组装调用  
添加因子需改集成代码

### 改进版做法

@register 装饰器自注册  
全局 Registry 自动发现  
添加因子仅需新建一个类

软件工程原理：开闭原则 (OCP) —— 对扩展开放，对修改封闭

## 门面模式 (Facade / Aggregator)

### Usecases 场景

用户执行每日风控检查时，系统应  
聚合所有风控信号并给出统一结论

### 原生版做法

各风控函数独立调用  
调用方自行组装结果  
无统一入口和聚合逻辑

### 改进版做法

统一 daily\_check() 入口  
聚合所有风控信号  
返回结构化的决策建议

软件工程原理：门面模式 —— 为复杂子系统提供统一的高层接口

Usecases 暴露了系统的真实复杂性，自然驱动了设计模式的演进 — DIP、OCP、Facade 是通用原则，不限于任何特定领域

# 工程质量维度对比：Usecases 带来的全面提升

从领域实现细节中抽象出来，Usecases 在五个通用工程质量维度上均带来了可量化的提升

## 抽象层次

3 类设计模式升级

原生：函数 + 简单类，调用方直接依赖具体实现

改进：Protocol 接口 + 抽象基类 + 门面聚合，依赖抽象而非实现

**Usecases 中的多角色/多路径场景，天然要求更高的抽象层次来管理复杂性**

## 可扩展性

开闭原则落地

原生：添加新功能需修改已有代码的多个位置

改进：注册表 + 适配器模式，新功能只需新增文件，不改已有代码

**Usecases 中的「配置参数」场景暴露了扩展需求，推动了插件式架构**

## 测试体系

+63% 测试文件

原生：8 个单元测试文件，覆盖基本功能

改进：13 个测试文件 + 共享 fixture + 195 行集成测试，覆盖完整用户流程

**Usecases 的主流程 + 异常流程直接转化为集成测试场景**

## 可观测性

+86% 指标数

原生：7 个性能指标，3 种可视化图表

改进：13+ 性能指标，5 种图表（含因子归因和趋势追踪）

**Usecases 中的「查看报告」场景明确了用户需要的分析维度**

## 领域规则完整性

+4 项边界规则

原生：覆盖核心业务规则（如交易限制、成本模型）

改进：+ 整手交易 + 微小变更跳过 + 前视偏差防护 + 渐进恢复

**Usecases 的异常场景建模系统性地发现了原生版遗漏的边界条件**

**Usecases 的价值不在于增加代码量，而在于系统性地提升抽象层次、扩展能力、测试覆盖、可观测性和规则完整性**

# 综合评分：原生 vs 改进版 OpenSpec 逐维度打分

采用与附录 1-A 相同的评分体系，精确量化 Usecases 步骤带来的增量价值

维度	原生版	改进版	差距	原生版特点	改进版特点
Artifact 完整性	7/10	10/10	+3	4 artifact 够用	8 UC 完整覆盖用户场景
架构设计	8/10	9/10	+1	简洁类/函数设计	Protocol + 装饰器注册表
边缘覆盖	7/10	9/10	+2	基本异常处理	前视偏差防护、微小再平衡
测试体系	7/10	9/10	+2	8 文件 / 74 方法	13 文件 + conftest + 集成测试
可扩展性	6/10	9/10	+3	添加功能需改多处	注册表 + 适配器可插拔
风控完备性	8/10	9/10	+1	4 层风控完整	4 层 + 聚合入口 + 波动率调仓
回测真实性	8/10	9/10	+1	完整 A 股规则	+ 整手 + Broker 抽象 + Sortino
报告丰富度	7/10	9/10	+2	3 种图表	5 种图表 + 中文字体回退
代码简洁性	9/10	7/10	-2	简洁易读、低门槛	抽象层多、学习成本较高
开发效率	9/10	7/10	-2	更少 artifact、更快交付	前置投入多，但后期维护省
总分	76/100	87/100	+11	简洁高效，适合快速交付	全面完善，适合长期维护

改进版在 8 个维度领先 (+1 到 +3)，原生版在 2 个维度领先（简洁性、效率各 +2）

总分 87 vs 76 — Usecases 步骤在中大型项目中 ROI 显著，但在小型项目中可能过度设计

## 附录 2：六大研究改进方向详解

---

以下内容基于 2023-2026 年软件工程与需求工程领域的学术研究，  
结合 17 次定向文献检索，梳理了 OpenSpec 最具潜力的六大改进方向。

每个方向包含：现状问题 → 解决方案 → 学术依据 → 商业价值

**其中 Domain-Specific Packs 作为公司核心竞争力方向，进行了重点展开。**

# 改进方向 1: Spec Quality Gate (智能需求质量关卡)

优先级: 最高 | 短期可落地

## 现状问题

在大多数团队中，需求规格的撰写依赖个人经验，缺乏统一的质量标准。PM 写完需求后直接交给开发，但“写了什么”和“开发理解了什么”之间往往存在巨大偏差。学术研究表明，30-50% 的软件返工最终可以追溯到需求阶段的模糊、遗漏或矛盾。

## 解决方案

在 OpenSpec 的工作流中增加一个 AI 驱动的“质量关卡”节点。每条需求在进入设计和开发阶段之前，自动接受三项检查：可测试性（能否写出对应的测试用例）、无歧义性（是否存在多种理解方式）、完整性（是否覆盖了所有必要场景）。

系统会为每条需求生成一个 0-100 分的 Quality Score，未达标的条目会附带具体的改进建议，让作者在源头修正问题。

## 学术依据

- Ferrari et al. (2023) 通过 74 篇系统性综述证实，LLM 在需求缺陷检测上已达到实用水平。
- Hymel et al. (2025) 实验显示，GPT-4 检测需求歧义的 F1 值达 0.72，覆盖率超过人工审查。
- ISO 29148 提供了国际通用的需求工程质量框架，可作为评分标准的参照系。

## 商业价值

- 减少 30-50% 的需求返工成本，这在大型项目中意味着数周的开发时间节省。
- 可作为 ISO/CMMI 合规工具，面向有审计需求的企业客户销售。
- 作为 OpenSpec 的“第一个可交付功能”，是最快见效的改进点。

实施路径: proposal → 校验检查点 → specs 分析 → 评分报告 → design 审查 → Quality Score 仪表盘

# 改进方向 2: Spec-Code Traceability (需求-代码自动追溯)

优先级: 高 | 中期建设

## 现状问题

随着项目迭代，需求文档和实际代码之间会逐渐“漂移”（Spec Drift）——需求改了但代码没跟上，或者代码改了但需求文档还是旧版本。在大型团队中，这种漂移往往要到集成测试甚至上线后才被发现。

更关键的是，EU AI Act 将于 2026 年 8 月正式实施，要求所有 AI 系统必须具备完整的需求-代码可追溯性（Article 11）。这意味着 traceability 不再是“锦上添花”，而是合规刚需。

## 解决方案

建立从 spec ID 到代码位置的双向映射关系。当需求变更时，系统自动标记受影响的代码文件；当代码被修改时，反向检查是否有需求需要同步更新。整套机制集成到 CI/CD 流水线中，实现“Living Specs”——需求文档始终反映代码的真实状态。

## 学术依据

- Cheng et al. (2026) 提出基于 LLMARE 的需求追踪框架，F1 值达 0.88，显著优于传统方法。
- EU AI Act Article 11 明确要求高风险 AI 系统具备全链路可追溯性。
- 多家监管机构（FDA、金融监管）正在跟进类似要求。

## 商业价值

- 合规驱动型市场：面向金融、医疗、AI 等受监管行业，traceability 是准入门槛。
- SaaS 订阅模式：按项目规模收费的持续性收入。
- 与 Quality Gate 形成产品组合：先检查质量，再追踪落地，闭环管理。

实施路径：spec ↔ code mapping → git diff drift detection → CI/CD 集成 → Living Specs 仪表盘

# 改进方向 3: Multi-Agent Orchestration (多智能体协同编排)

优先级: 高 | 中期建设

## 现状问题

当前的 AI 编程助手通常是"单 Agent"模式——一个 AI 从头到尾处理整个任务。这在简单任务上没问题，但面对包含 20+ 子任务的复杂项目时，单个 Agent 容易丢失上下文、遗忘前序步骤，错误率随任务规模急剧上升。

## 解决方案

利用 OpenSpec 天然的 task DAG (任务有向无环图) 结构，将复杂任务拆解为多个子任务，分配给不同的专业化 Agent 并行处理。每个 Agent 只需关注自己的子任务，而 OpenSpec 的 artifact (需求规格、设计文档) 充当 Agent 之间的"协议"，确保最终产出一致。

这类似于软件团队的分工协作，只不过"团队成员"变成了多个 AI Agent。

## 学术依据

- GitHub 于 2026 年 2 月发布 Agent HQ，支持多 Agent 协同开发，验证了这一方向的产业价值。
- Anthropic 的 Claude Code Agent Teams、SWEBench 多 Agent 方案均显示并行处理效率提升 3-4 倍。
- DAG 调度是成熟的工程范式 (Airflow、Prefect 等已在数据工程领域广泛验证)。

## 商业价值

- 效率提升 3-4 倍：复杂任务的端到端时间大幅缩短。
- 计算资源按量付费：多 Agent 并行 = 更高的 API 调用量 = 收入增长。
- 与 GitHub/Anthropic 的生态对齐，可借势而非逆流。

实施路径: tasks.md → DAG 解析 → Agent Worker Pool → 乐观并发 → artifact 冲突检测 → 合并

# 改进方向 4: Platform / Web Hub (从命令行到协作平台)

优先级: 中 | 中长期

## 现状问题

OpenSpec 目前是一个命令行工具 (CLI)，这意味着只有会用终端的工程师才能使用它。但在实际项目中，需求的利益相关者还包括产品经理、测试工程师、业务方——他们需要查看和评审需求文档，但不会使用命令行。

此外，artifact graph (需求之间的依赖关系图) 在 CLI 中难以直观呈现，而这恰恰是理解项目全貌的关键信息。

## 解决方案

分阶段将 OpenSpec 从 CLI 扩展为 Web 平台：第一步是提供 Web UI 来可视化 artifact graph 和 spec 文档；第二步增加协作功能（评审、评论、审批流）；第三步构建 Marketplace，让社区可以分享和交易 spec 模板和 Domain Packs。

## 学术依据

- AI 编程工具市场 2025 年规模达 \$202 亿，Cursor (\$2.5B+)、Devin (\$2B+) 均走产品化路线。
- Kiro (Amazon) 和 GitHub 已推出 spec review 的可视化功能，证明市场需求存在。

## 商业价值

- PLG (产品驱动增长)：免费 CLI → 付费 Web 协作 → 企业版，经典的 SaaS 增长路径。
- Marketplace 抽成：第三方 spec 模板和 Domain Pack 的交易平台。
- 扩大用户群：从“工程师专用”变为“全团队可用”，市场天花板大幅提高。

实施路径: CLI 引擎 (已有) → REST API 层 → Web 前端 → 团队协作功能 → Marketplace

## 核心洞察

通用大模型 (LLM) 擅长写代码，但不懂行业。它不知道金融交易必须做风控检查、医疗系统必须符合 HIPAA、汽车软件必须通过 ISO 26262 认证。而公司多年积累的行业 Know-How——业务规则、合规标准、最佳实践——恰好填补了这个空白。将这些 domain knowledge 封装成结构化的"领域规格包" (Domain Pack)，就是把公司的隐性知识变成可复用、可销售的数字资产。

## 什么是 Domain Pack?

一个 Domain Pack 本质上是一组行业专属的"规格模板 + 规则集 + 验证逻辑"，以 OpenSpec 的标准格式封装。当 AI 在生成需求规格和代码时，Domain Pack 会作为先验知识 (prior knowledge) 注入工作流，使 AI 的产出自动遵守该行业的规范和约束条件。

### 一个 Pack 包含的具体内容：

- Rules (行业规则)：如"所有交易必须经过风控引擎"、"患者数据必须加密存储"
- Spec 模板：预定义的需求规格模板，包含行业必填字段和标准结构
- Schema 定义：数据结构和接口的行业标准约束
- 验证规则 (Validators)：自动检查生成的代码是否违反行业规范

类比：如果 OpenSpec 是一个"AI 建筑师"，那么 Domain Pack 就是"行业建筑规范手册"——没有它，AI 盖的房子不合规。

## 护城河分析：为什么别人难以复制

### 1. Domain Knowledge 是稀缺资源

行业规则、合规标准、最佳实践，这些知识散布在行业专家的脑中、监管文件里、和多年项目经验中。它们无法通过简单的网页爬取或模型训练获得。公司已有的行业积累，就是天然的数据壁垒。

### 2. 先发优势带来网络效应

第一个做出高质量金融 Pack 的团队，会吸引金融领域的用户；用户的使用反馈又会帮助改进 Pack 质量，形成“越用越好→越好越多人用”的飞轮。后来者即使有技术能力，也缺乏这个冷启动的数据积累。

### 3. 合规认证构成额外壁垒

如果 Domain Pack 能通过 ISO 26262（汽车）、HIPAA（医疗）、SOX（金融）等行业认证，这本身就是一个极高的准入门槛。竞争对手不仅要做出 Pack，还要花时间和成本通过认证。

### 4. 与 OpenSpec 生态深度绑定

Domain Pack 是基于 OpenSpec 的 spec 格式和 schema 体系构建的，这意味着使用 Pack 的客户同时也被锁定在 OpenSpec 生态中。Pack 越多，生态越强，切换成本越高。

一句话总结：AI 能力人人都有，但 domain knowledge + 结构化封装 + 合规认证 = 难以逾越的竞争壁垒。

## 金融 / 量化交易 Pack

### 为什么先做金融?

公司在量化交易项目中已积累了完整的风控规则、回测验证逻辑和交易引擎架构经验。这些就是现成的 domain knowledge，只需结构化封装即可变成 Pack。

### Pack 包含的内容:

- 风控规则集（止损、持仓限制、异常检测）
- 回测真实性验证逻辑（防前视偏差）
- 交易引擎 spec 模板
- SOX / MiFID II 合规检查

## 医疗 / 生命科学 Pack

### 市场驱动力:

FDA 对医疗软件的监管日趋严格，HIPAA 合规是所有医疗 IT 系统的准入门槛。医疗机构愿意为确定性的合规方案支付高溢价。

### Pack 包含的内容:

- HIPAA 数据保护规则集
- FDA 510(k) 软件验证流程模板
- HL7 FHIR 接口 schema
- EU MDR 合规检查

## 汽车 / 工业控制 Pack

### 市场驱动力:

汽车软件正从嵌入式向智能化转型，ISO 26262 功能安全认证是强制要求。传统车企缺乏 AI 辅助的 spec 工具，这是一个蓝海市场。

### Pack 包含的内容:

- ASIL 等级 spec 模板
- AUTOSAR 架构约束
- OTA 更新安全验证
- IEC 61508 工业安全标准

## 商业模式: Vertical SaaS

每个 Domain Pack 按行业 + 规模定价（年订阅制），目标客单价 \$10K-50K/年。先从公司最熟悉的金融领域切入（冷启动成本最低），再向医疗和汽车扩展。每进入一个新行业，都需要与该领域的行业专家合作，这进一步巩固了护城河。

落地路线: 金融 Pack (Q1, 已有基础) → 医疗 Pack (Q2, 合规驱动) → 汽车 Pack (Q3, 安全关键) → 更多行业

# 改进方向 6: Formal Verification (规格的形式化验证)

优先级: 低 (长期) | 技术储备

## 现状问题

目前验证“代码是否满足需求”主要依赖测试——但测试只能覆盖有限的场景。对于关键代码路径，“测试没报错”不等于“一定没有 bug”。在航空、金融、医疗等领域，这种不确定性是不可接受的。

## 解决方案

将 OpenSpec 的 Given/When/Then 格式的需求规格，自动转换为形式化语言（如 LTL、CTL、SMT 公式），然后用数学方法证明代码在所有可能的输入下都满足规格要求——而不是仅在测试用例覆盖的输入下。

这是一个技术门槛很高的长期方向，但一旦实现，将是 OpenSpec 最具技术壁垒的差异化功能。

## 学术依据

- Klopmann (2025) 提出 AI 辅助形式化验证框架，将验证通过率从 68% 提升至 96%。
- Vericode benchmark 显示 LLM 生成的形式化 spec 质量已接近人工水平。
- ReaLis、Zhipu 等团队正在该方向快速推进。

## 商业价值

- 安全关键行业（航空、医疗、金融）的形式化验证咨询费用通常在 \$10K-\$100K+ 量级。
- FDA/ISO 认证越来越认可形式化方法，长期具有合规价值。
- 技术壁垒极高，一旦建立先发优势，竞争对手追赶成本巨大。

实施路径: Given/When/Then → 属性提取 → LTL/CTL 公式生成 → SMT 求解器验证 → 验证报告

# 优先级排序与推荐实施路径

1	<b>Spec Quality Gate</b> 短期 (1-3月)	最快见效，直接提升需求质量，降低返工成本
2	<b>Domain-Specific Packs</b> ★ 短期启动，持续建设	核心竞争壁垒，公司 domain knowledge 的商业化载体
3	<b>Multi-Agent Orchestration</b> 中期 (3-6月)	效率倍增器，与行业趋势对齐
4	<b>Spec-Code Traceability</b> 中期 (3-6月)	合规刚需，EU AI Act 驱动
5	<b>Platform / Web Hub</b> 中长期 (6-12月)	PLG 增长引擎，扩大用户群
6	<b>Formal Verification</b> 长期 (12月+)	技术储备，极高壁垒但实现周期长

## 推荐策略

短期以 Spec Quality Gate 作为 MVP 快速验证产品价值，同步启动 Domain-Specific Packs (金融领域) 的建设。中期补齐 Multi-Agent 和 Traceability 能力，长期向平台化和形式化验证演进。Domain Packs 贯穿始终，是从短期到长期持续积累竞争壁垒的核心抓手。