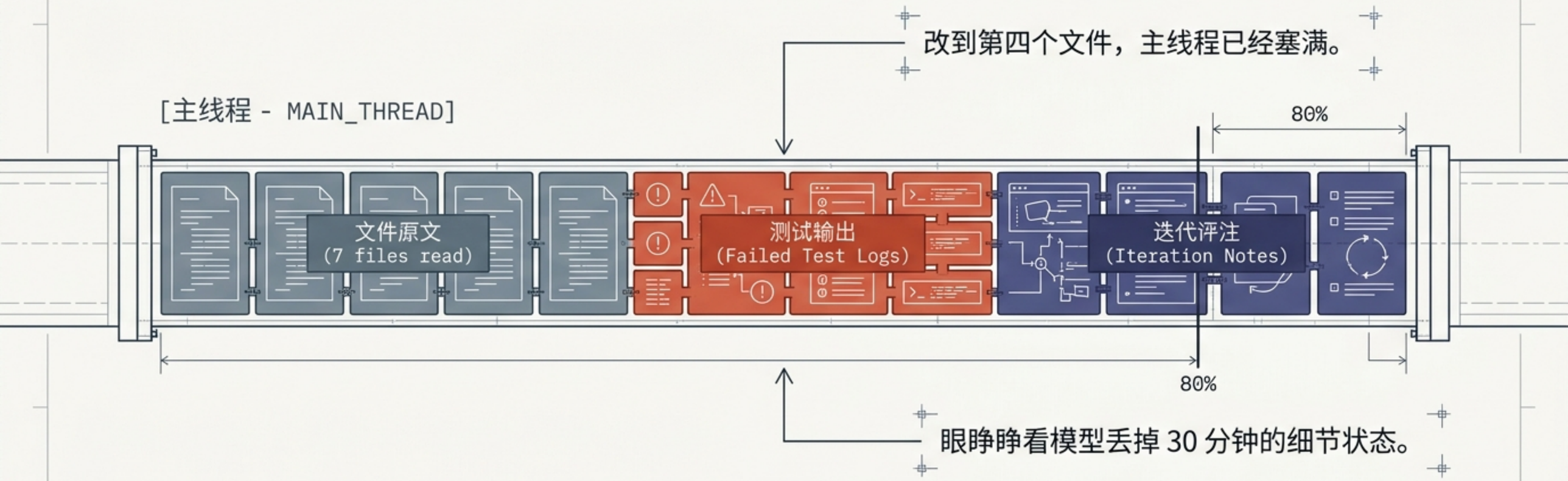
两个月，推倒重来四次：我的 AI 工作流演进史

为什么 Solo Dev 的底层框架不同于团队，
以及如何构建基于“工作形状”的动态路由。

```
~ % execute ./orchestrator --analyze
```

STATUS: Published | SYSTEM: Claude Code

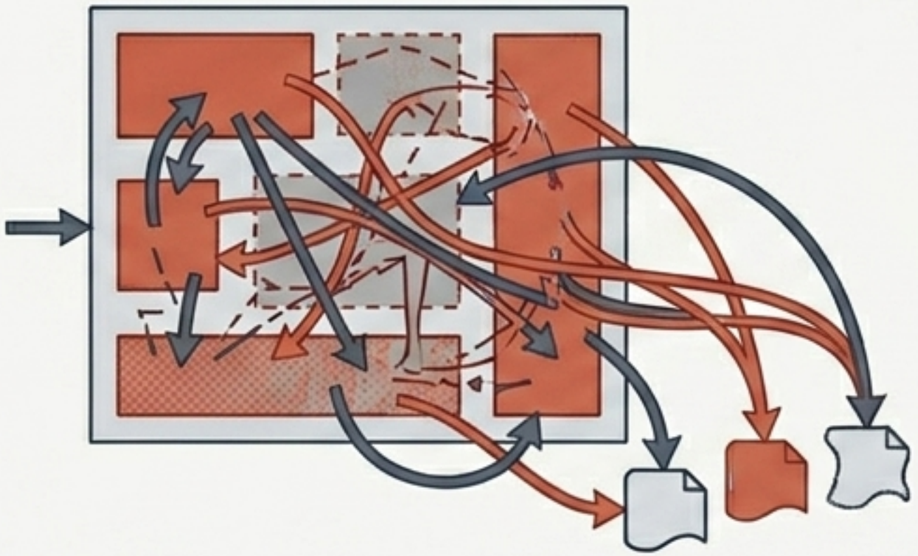
那个让 Vanilla 崩盘的 Session



裸 Claude Code 是完美的 Tier-1/2 工具（单文件/单问题）。再大一点，主线程就会沦为 Context 垃圾场。

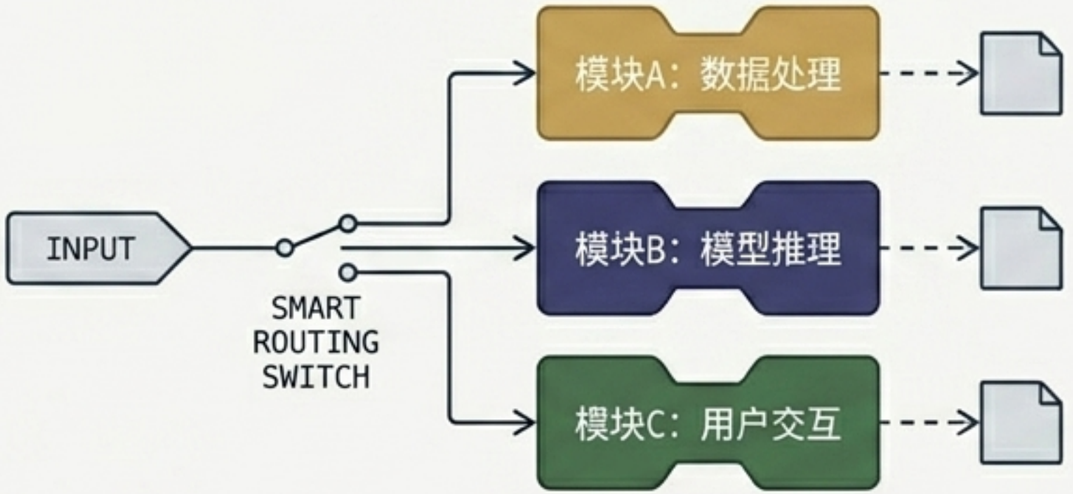
核心顿悟：工作流的形态必须匹配任务的形状

神话：单线程通吃



没有一个统一的默认引擎能处理所有事情。

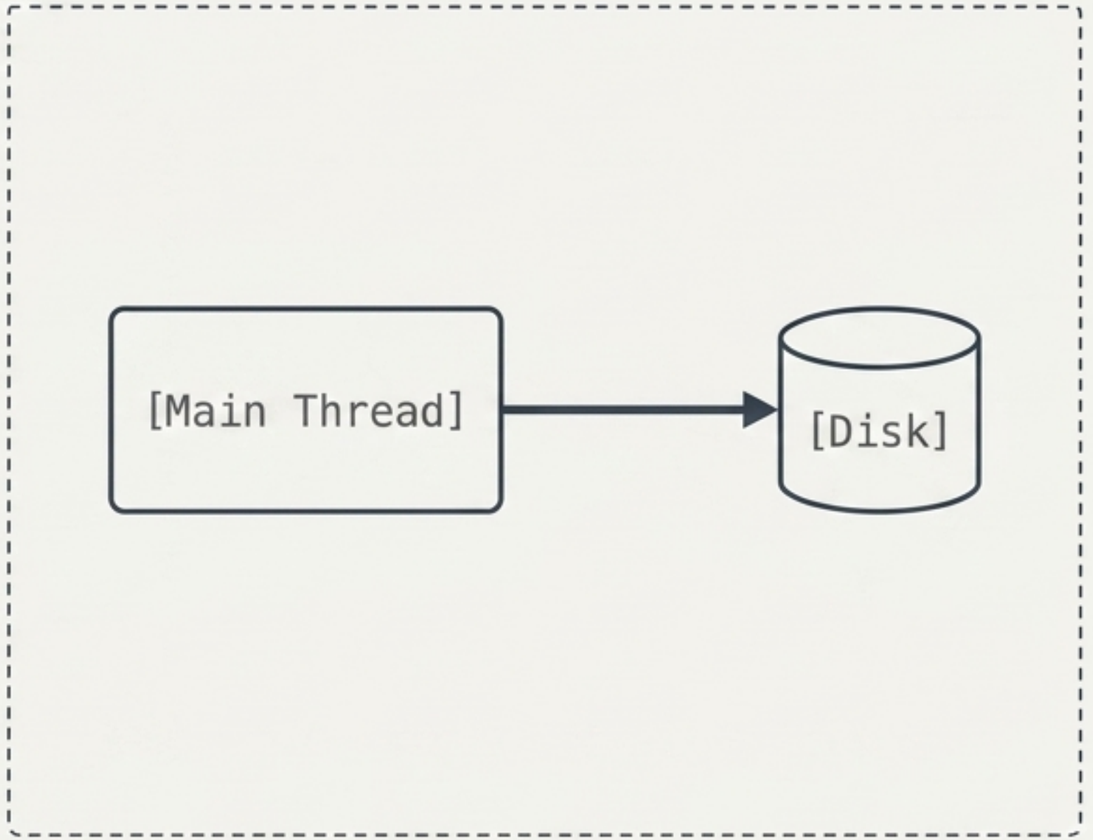
现实：多态路由



解决 context 上限的终极方案不是更好的模型，
而是更好的架构纪律。

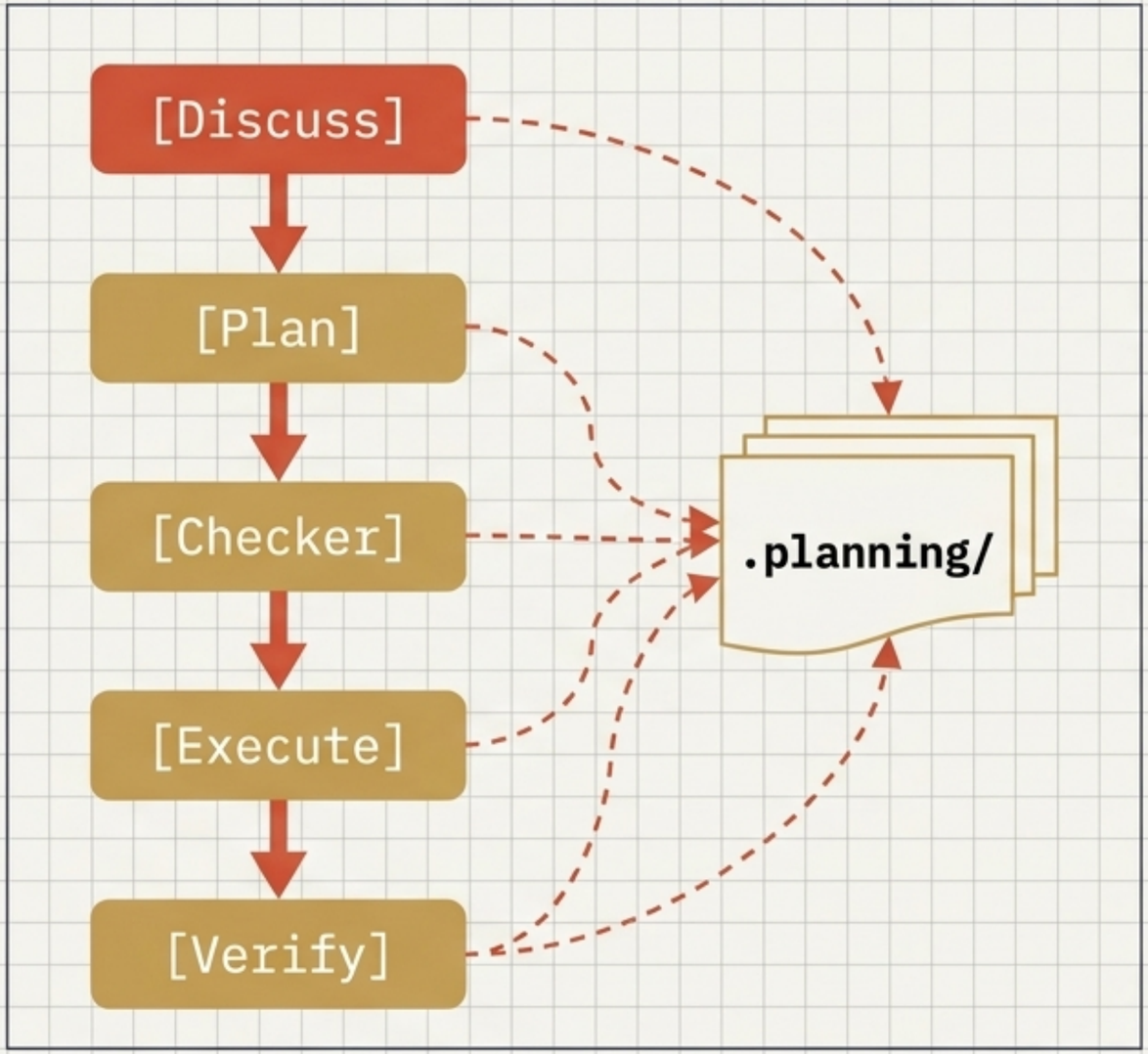
Solo Dev 的项目终将达到团队规模的复杂性。

架构一：Vanilla（裸奔模式）



最佳场景	Tier-1/Tier-2（Hotfix，打字错，单函数）。高信噪比。
崩盘点	>2-3 个文件 + 持续 TDD 测试。
致命缺陷	一个线程干所有事，10 个文件的小功能会在 1 小时内吞噬 80% 的 Context。

架构二：GSD（多 Agent 仪式）



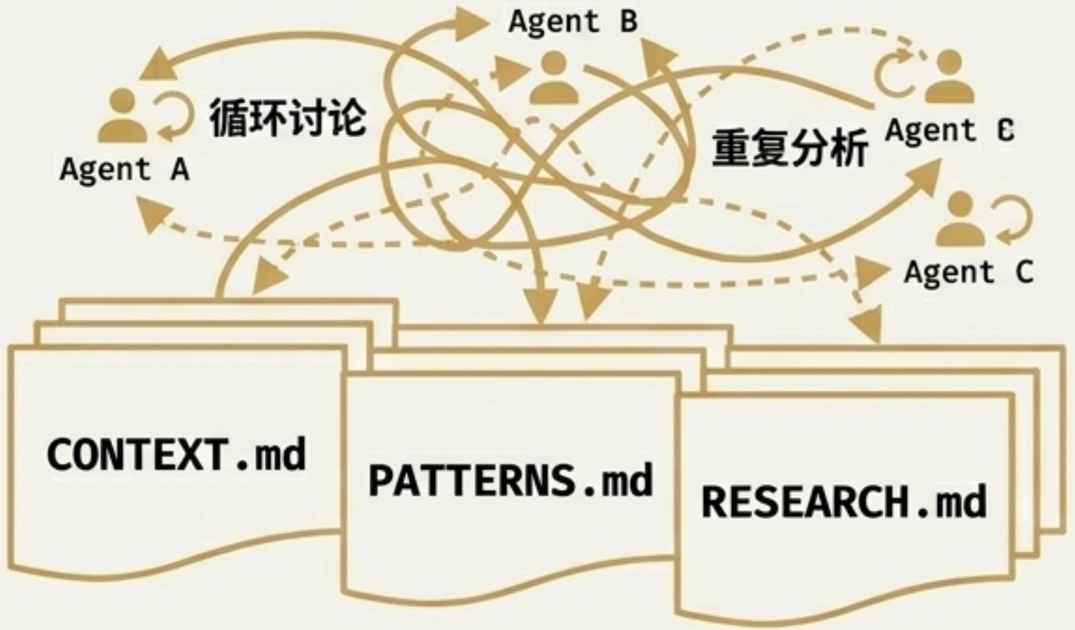
Specification		
1	最佳场景	真正的重型架构迁移（Schema 跨 phase 依赖，完整重设）。
2	崩盘点	Solo Dev 的中型任务（Tier-3）。每次改动都会生成六份独立文档。

.planning/ 目录代码行数统计
> 识川项目：~21,000 行
> ÉLAN 项目：~24,000 行

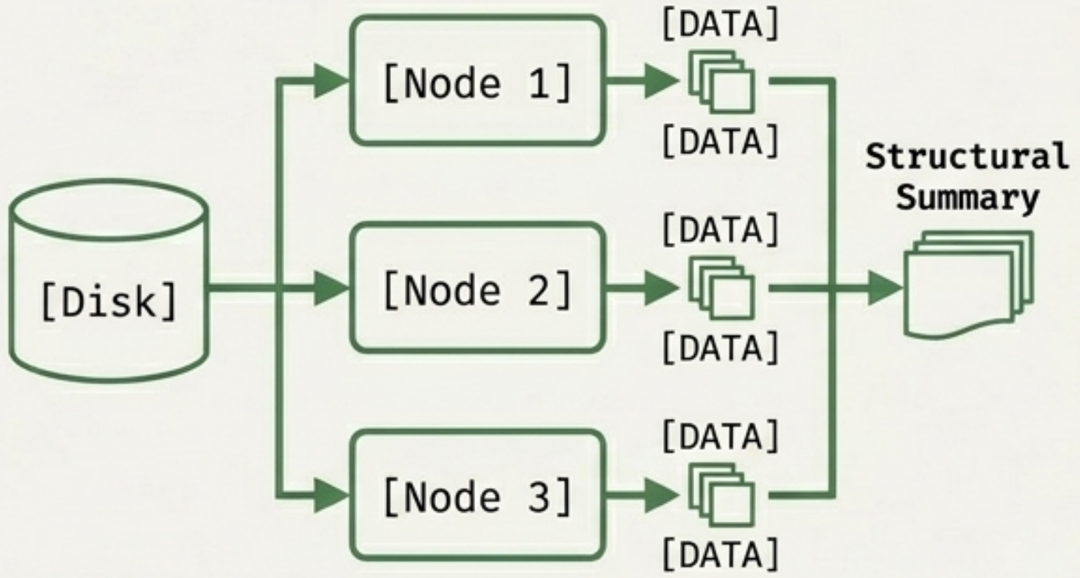
结论：把规划文档读完的时间比写代码还长。

两个月才看懂的 GSD 悖论

规划仪式（过度工程）

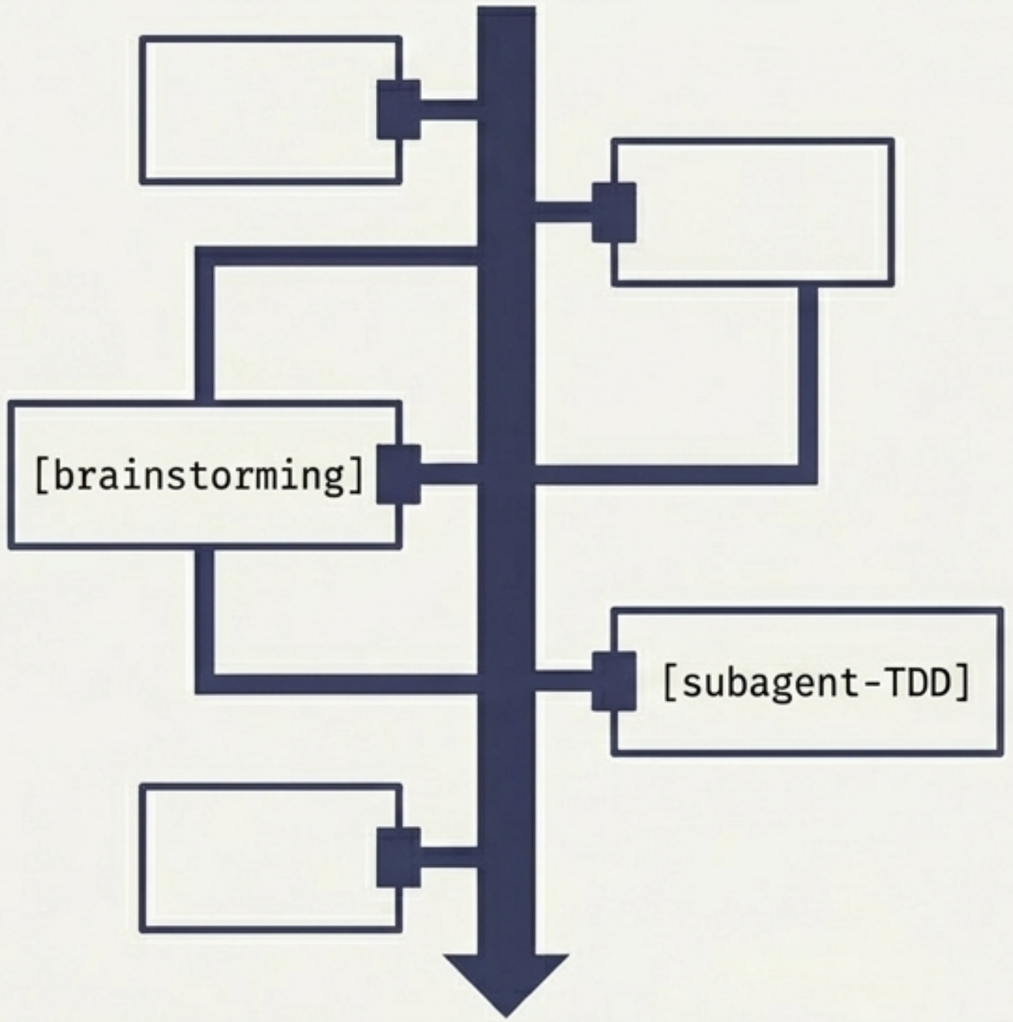


执行引擎（极其精简）



GSD 的 Execute 引擎在主线程只占 ~15% 的 budget！Subagent 自己从磁盘加载 Plan，只回传结构化 Summary 而非原文。这套长时自动跑的纪律无人能敌。

架构三：Superpowers（Skill 库模式）



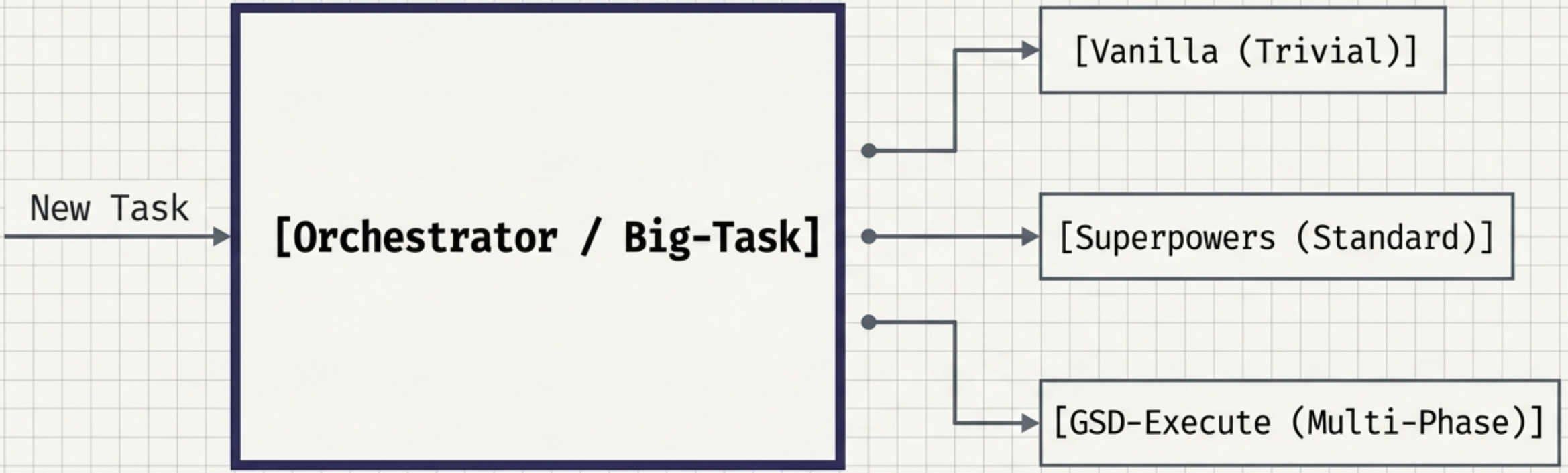
Specification		
1	最佳场景	标准 Feature，设计已锁定。强大的两段 Review 机制。
2	优势	技能可组合，通过 session-start hook 强势注入，绝不漏调。
3	致命缺陷	1. **Auto-injection Tax**：每次 Session 固定消耗 ~5,000 Token。 2. **明文串行**：为了避免冲突，强制串行执行任务。20 个 Task 的计划会让主线程迅速臃肿，无法实现真正的并行。

演进总结：为什么前三次都失败了？

	Vanilla	GSD（Execute）	Superpowers
主线程占用	极高	最低（~15%）	中高
任务并行度	无	完美并行	无（串行瓶颈）
启动与规划开销	零	灾难级（6 份文档）	5k Token 税

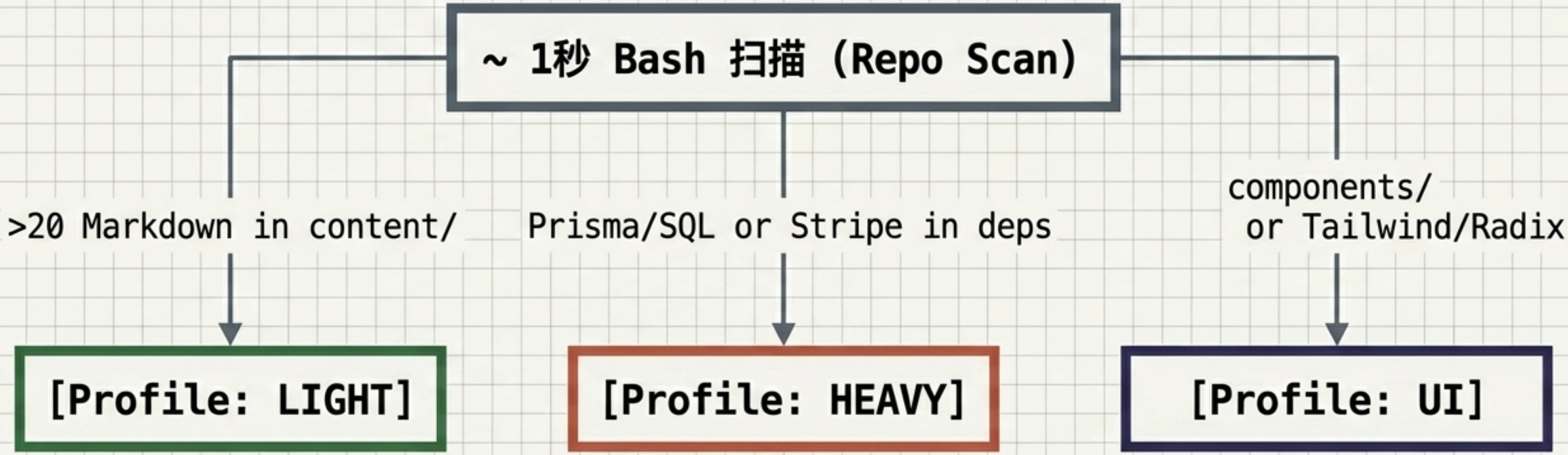
我手里拿着两个“对了一半”的框架。真正的答案不是造一个更好的轮子，而是造一个智能路由器。

终极答案：架构四 Big-Task（不造轮子，只做路由）



- 它唯一的活儿是“选对”。它不重新实现 TDD，不重新实现 subagent 派发。
- 当任务涉及 3+ 文件时自动触发，根据 工作形状 动态委托底层引擎。

Phase 0.0: 第一步：基于代码库特征的启发式路由



快速扫描回答了“这个 codebase 是什么形状”，但这还不够。博客仓库是 Light，但在博客里加 Stripe 支付网关绝不是轻量活。

Phase 0.0: 第 2.5 步：超越关键词，识别真实意图

愚蠢的关键词匹配

```
> REGEX SCANNER
Scanning: "I looked at Stripe API yesterday"
Match Found: "Stripe" -> [HEAVY]
```

在 LLM Context 里拿 Regex 过滤自然语言是反模式。

基于系统影响的意图分类

- **本质重活（Heavy）**：持久化变更，Trust boundary，Revenue 路径。
- **本质轻活（Light）**：书面文字，纯视觉微调。
- **UI 性质（UI）**：已知模式的第 N+1 次应用。

爆炸半径与可逆性决定真实层级，不受 Repo 基础属性限制。

第三步：按工作性质派发 Subagent

模式（Mode）	何时用	主线程占比
`parallel-worktree`	独立文件上的实现	<20% (独立 worktree)
`parallel-readonly`	调查、审计、审查	<20% (每目标一 agent)
`serial-subagent`	共享文件实现	~30%
`inline`	单文件, hotfix, 琐碎决策	**100%**

钢铁底线：当 Tier ≥ 3 且独立任务 ≥ 3 时，绝不 Inline。 再大还 inline 主线程必臃肿！

Orchestrator 的四大钢铁纪律

纪律一：磁盘加载

Subagent 必须自己从磁盘读取输入，主线程不把全文塞进 Prompt。

纪律二：摘要限制

Subagent 仅回传结构化 Summary，强制 ≤ 200 Token，拒收原始输出。

纪律三：状态落盘

运行状态保存在本地磁盘，绝不留在 Chat History。

纪律四：并行隔离

并行任务强制使用 git worktree 隔离，绕过共享状态导致的写入冲突。

大师蓝图：给 Solo Dev 的终极路由指南

任务形态 (Task Shape)	推荐引擎 (Recommended Harness)	核心理由 (Rationale)
琐碎改动	[Vanilla]	极高信噪比，单线程足矣。
内容与视觉	[Vanilla] 或 [Big-Task (Light)]	无爆炸半径，轻量快速。
标准功能，设计锁定	[Superpowers (Serial)]	强大的两段 Review 防止偏离规格。
多 Phase 并行	[GSD Execute 引擎]	极低主线程开销（~15%），适合长时无人值守运行。
规格不清，探索性架构	[Superpowers (Brainstorm -> Plan)]	强制思考，架构决策重于执行吞吐量。

永远不要把某一个 Harness 当作默认。

> 我前三次犯的错都一样：先挑 harness，再逼工作去匹配。Harness 必须跟工作形状匹配。上面的 Orchestrator 只是个路由，真正的核心是你得清楚：这次到底是什么活。

```
> BIG-TASK: github.com/xingfanxia/claude-config
  > GSD: github.com/gsd-build/get-shit-done
  > SUPERPOWERS: github.com/obra/superpowers
```