VIBE CODING:

从“想法”到“产品”还有多远？

AI原生软件开发范式探索

2025-10-15

目录（AGENDA）

1. 关于输入输出的思考
2. 关于AI辅助编程的思考
3. AI时代的软件架构与软件工程
4. AI原生程序员？
5. 结语

一、从输入输出的角度来理解世界

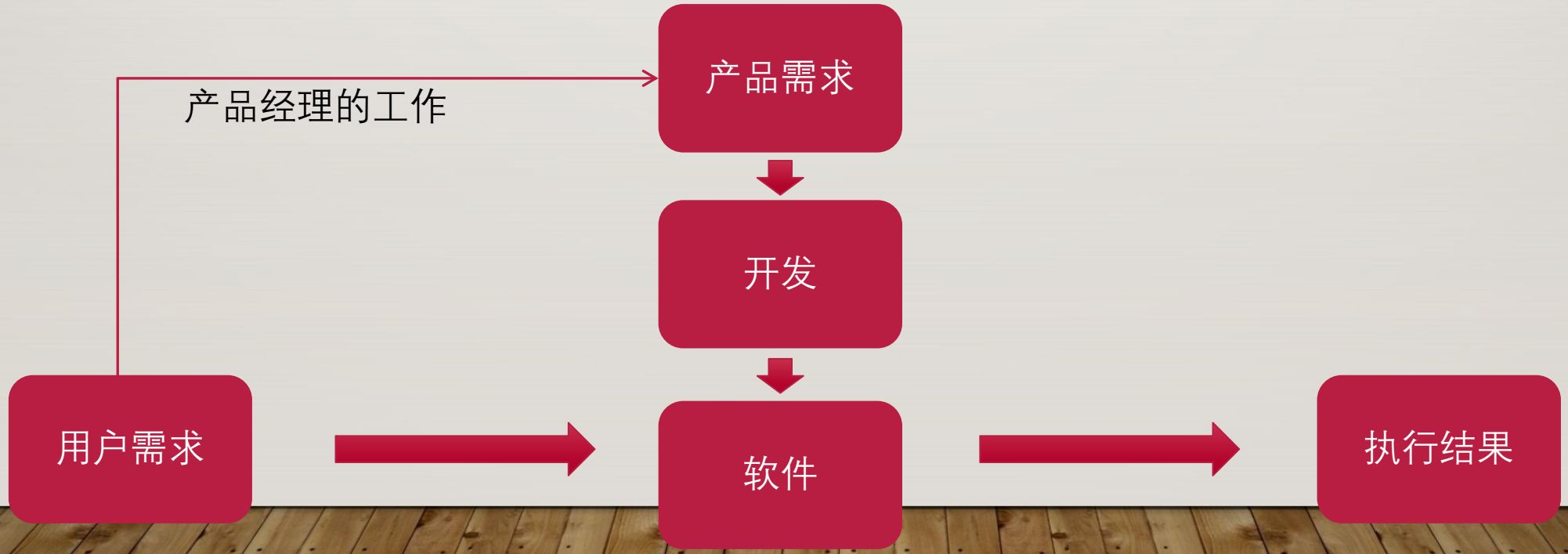
- 用 I/O 抽象看系统：任何系统都是信息的“输入→处理→输出”。
- 关注可观测性：输入可度量、处理可解释、输出可验证。
- 关键在于：处理的环节是如何实现的?

处理的环节从人变成了机器（硬件+软件）

- 自动化的本质：把可重复的认知步骤固化到机器。
- 硬件提供算力与IO通道；软件提供规则与状态机。
- 人从主力执行者转变为意图表达者与监督者。
- 下一个问题：处理阶段所需的软件，如何生产？

开发仍是：输入（需求）→处理（开发）→输出（软件）

- 需求清晰度决定开发效率；验证机制决定输出质量。
- 需求 = 目标 + 约束 + 验收标准（tests/specs）。



VIBE CODING 的期望：用 AI 替代大部分开发流程

- 把“想法/意图”直接转译为可运行系统的骨架。
- 开发者从“写代码”转向“编排与验收”。
- 关键是形成一套可复用的“意图→工件”流水线。
- 人类主要贡献：目标清晰化、风险边界、价值判断。

输入信息量与输出信息量：如何提高 O/I 比？

我如何说一句话，让那个小伙连干三天？

产量增加的三种情况

- 重复劳动：抄写1000遍
- 目标明确：把这个算法的效率，提高20%
- 创意发挥：约定俗成+探索式生成+遵守约束

任务目标类型：真、善、美（软件常常三者兼有）

- 真（Correctness）：功能正确
- 善（Goodness）：性能可靠、体验顺滑
- 美（Aesthetics）：视觉美观、结构优雅、可读可维护。

创意带来的不稳定，如何收敛？

RAILS 的创新：约定大于配置

- 通过“合理默认 + 目录结构 + 脚手架”固化最佳实践。
- 减少自由度 = 减少出错面 = 提升生产率与一致性。
- 把“专家经验”沉到框架里，让新人即刻可产出。
- Vibe Coding 借鉴：以约定化组件/蓝图驱动生成。

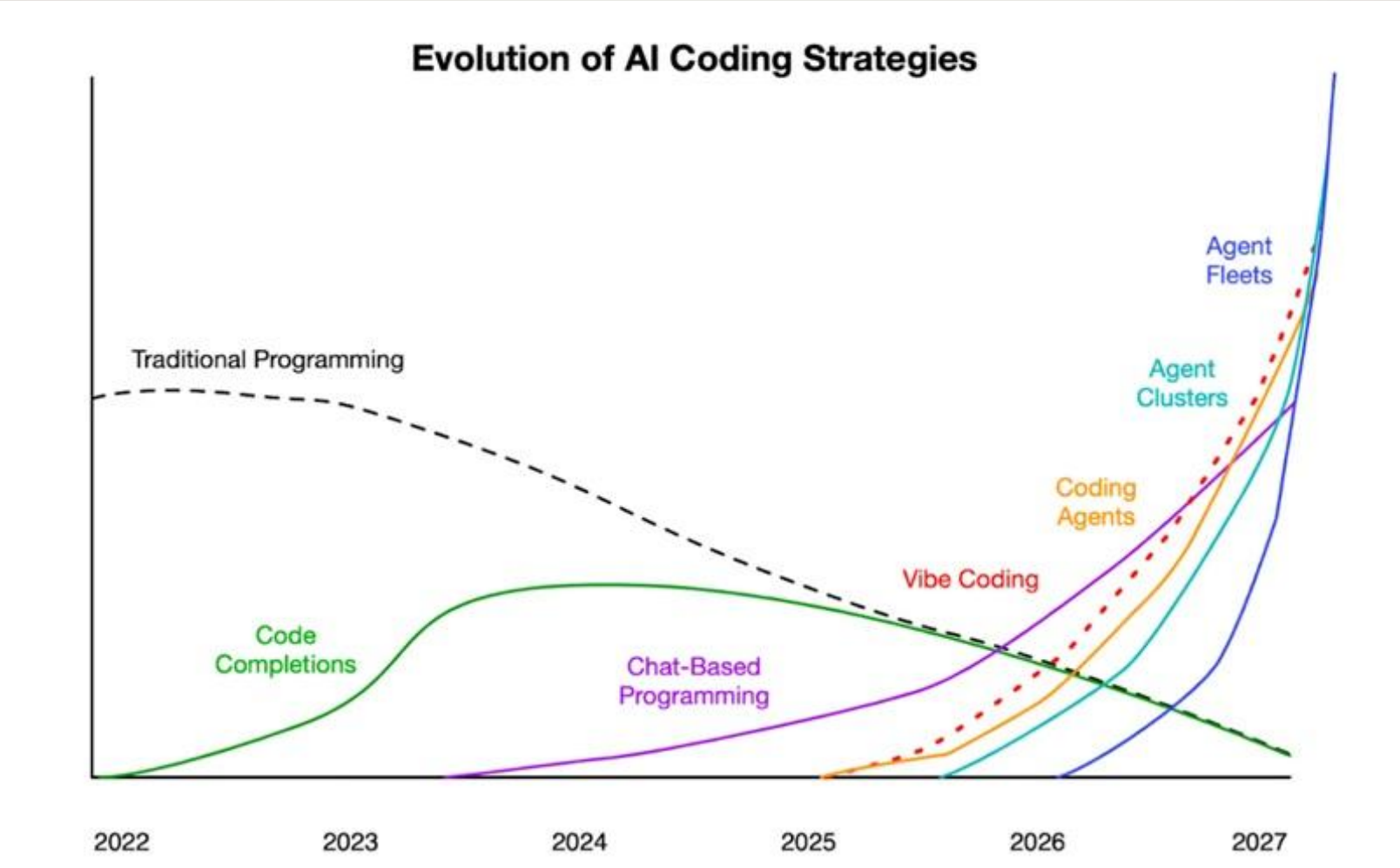
VIBE CODING =

LLM + 领域知识 + 设计模式 + 最佳实践 + 测试驱动

我们仍处于一个飞速变动的时期

- 模型/工具更新极快：关注“稳定能力”，少绑死具体工具。
- 以“过程资产”沉淀而非“工具清单”。
- 建立评测与回归基线，稳步升级。
- 目标：在变化中保持可控与可复现。

二、关于AI辅助编程的思考



来源：Revenge of the junior developer

工具体验：ROO CODE；CLACKY.AI

- Roo Code：VS Code插件
- Clacky.ai：从IDE到CDE

最近在试用的两个工具

- Claude Code + DeepSeek 3.2
  - 比较好用，能够完成大部分工作
- OpenAI CodeX
  - 适合用于管理GitHub的开源代码仓库

新玩具：GITHUB SPEC-KIT

- 地址：https://github.com/github/spec-kit
- 意图→规格→工件：以规范化描述驱动生成与校验。
- 提示：把“需求规格”做成可执行/可测试的第一等公民。
- 与 Vibe Coding 高度契合：先规格、后生成、再验收。

我们的 AI CODING 表演赛：6 个工具 × 3 类题

- 三类题：Issue 级、复刻级、创意级。
- 说实话，6种工具的差距并不大
- 作为开发者，不需要有任何忠诚度

我们的《AI编程宣言》

1. 目标达成 胜于 机器炫技

聚焦解决实际问题和创造价值，而非炫耀AI模型或工具的能力。

2. 人机共创 胜于 彼此对立

人类掌控方向，人机协作达成目标。

3. 拥抱变化 胜于 墨守陈规

主动拥抱 AI，实现编程从传统到智能的跃迁。

4. 明晰洞察 胜于 魔法崇拜

理解 AI 编程的原理、能力和边界，建立理性认知体系。

5. 过程可控 胜于 盲盒祈愿

确保 AI 编程开发过程可控。

6. 安全持续 胜于 贪功冒进

在 AI 编程中坚守代码质量和系统可维护性。

遗憾一：为何不是《AI软件工程宣言》？

遗憾二：

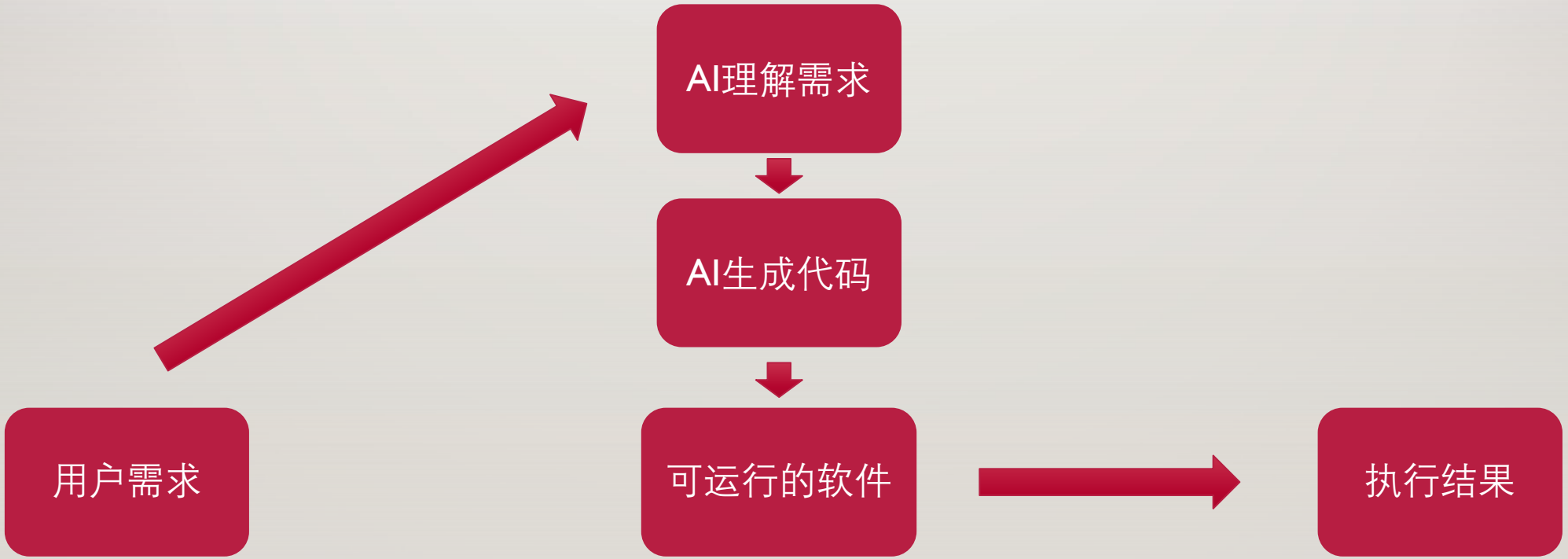
一个朋友的批评，如果这个时代最大的哲学是不要编程呢，你怎么说？

三、关于AI时代的软件架构与软件工程的思考

架构中如何容纳不确定性？

能不能在需要用到的时候再生成代码？随用随丢？

压缩层级：需求→代码→解答



AI时代的程序员画像与技能

- 意图建模：把模糊愿景转为可执行规格与样例。
- 系统思维：边界、耦合、复杂度与成本意识。
- 数据与评测：构造数据集、建立自动化验收/回归。
- 工具链：代理/编排、CI/CD、云原生与安全。

过去的演进路径

- 软件开发 → 工程化开发 → 组织与管理 → 工具化。
- 工具沉淀最佳实践，标准使规模化成为可能。
- 经验：每一步都伴随度量与反馈机制的建立。

未来：开发/架构/协作方式高度不确定

- 多范式并存：代理式、规格驱动、生成-审阅混合。
- 团队规模与边界漂移：小而精团队与超大规模编排并存。
- 合规与安全左移，隐私/版权/供给链成为硬约束。
- 工程化将更多关注“约束满足”而非“代码行数”。

结论：我们仍难完全定义“AI时代的软件工程”

四、AI原生程序员?

AI 原生学习路径将完全不同

- 任务驱动与反向学习：先做项目，再补空白。
- 以规范/测试/示例为起点，而非教科书章节。
- 多模态：代码+文档+对话+流程图一体化。
- 有效反馈比“时长”更重要。

基础知识不“全盘”重要，但关键词必须掌握

- 高频概念词表+心智图，定位问题空间。
- 足够的计算机原理/网络/安全“骨干”知识仍关键。
- 会查证与溯源：官方文档、标准、权威实现。
- 知道“不知道”的边界：何时请专家、何时请工具。

如何把需求说清楚?

- 四要素：目标、约束、示例、验收（Given-When-Then）。
- 使用最小可行规格（MSS）+ 扩展测试。
- 多轮澄清：让 AI 复述你的意图以校验一致性。
- 把决策记录到“变更日志/设计记录”。

如何判断 AI 输出的结果是否合格?

- 自动化测试优先：单测/集成/契约/回归。
- 客观指标：性能、安全、可维护性、复杂度。
- 对齐目标：是否满足业务 KRs/用户故事?
- 人工审查：关键路径代码与架构决策需“二签”。

新最佳实践可能出现

- “对话即工件”：把对话与生成过程纳入版本控制。
- 数据与提示的可观测性：Prompt/Spec/Dataset 版本化。
- “一次性工装”模式：按需生成/销毁的脚本与服务。
- 评测驱动发布：每次改动通过统一基线。

未来社区会有哪些特征?

- 意图社交：围绕“想解决什么”组织协作。
- 小而精社群：高密度产出与快速复盘。
- 代理作为成员：不断进化的Agents

从 OPEN SOURCE → OPEN RESOURCE → OPEN INTELLIGENCE

结语

保持探索、保持思考、保持开放