AI Coding 实战：10年祖传系统，54万行代码，2周重构结束

书接上文：**《万字：AI Coding 到了什么程度了？》**，我们得到的结论是：

> AI Coding 已经不只是适合写 Demo、补函数、起页面了。
>
> 在约束清楚、边界明确、验收方式可执行的前提下，它已经可以参与相当一部分真实交付工作**

然后，用过 AI Coding 的同学也会很清楚，在新项目、规则清晰的项目上，AI 的威力很大，那么对应的问题也就来了：**年老失修的屎山代码 AI Coding 的情况怎么样呢？**比如：

> 运行十余年的系统，文档缺失、逻辑混乱、维护成本高到离谱，想重构又怕出问题，不重构又制约业务迭代，这种地狱模式 AI 是否立得住？

今天就和大家分享我们团队的 AI Coding 实战经历：**仅用2周，成功重构一套运行10年、累计54万行PHP代码的核心电商系统，平稳迁移至Java版本，**全程依靠AI，成功做到了了按时平稳地交付。

地狱开局

10年老系统、54万行 PHP、两周交付窗口、核心电商链路、PHP 迁到 Java，这类项目，哪怕纯人工来做，很多团队都得沉默。

因为这种事最难的地方，从来都不是代码怎么写，而是：你根本不知道自己到底在动一个什么东西。

老系统不像新项目，新项目业务逻辑清晰（至少文档清晰），很多时候你只要把需求拆清楚，AI 就能一路往前推，但屎山代码不是这样。

它最大的问题，不是烂，而是**它烂得很复杂、很真实、还已经在线上跑了很多年。**

很多逻辑，文档里没有；很多分支，设计稿里没有；很多兼容，甚至连原开发自己都未必记得为什么要这么写。

你今天看到的一段坏代码，很可能不是谁当年水平不行，而是因为三年前有个线上事故，临时用这种方式先兜住；你今天觉得某个字段命名奇怪，也许背后已经挂着三个旧客户端和五个外围系统...

所以，面对这种系统，**最大的风险不是写不出来，而是你以为自己理解了，实际上根本没理解。**

而这次项目最值钱的地方，也恰恰在这里。它不是在证明 AI 已经能独立重构屎山系统了，而是在证明：

当人把边界守住之后，AI 已经足以吞掉重构里大量最脏、最累、最重复的工作。

这才是我觉得这个案例真正值得拿出来讲的原因。接下来我们还是要正面难点：

老系统重构

很多人提到老系统重构，第一反应一般都是代码量大、历史包袱重、技术债多。

这些当然都对，但如果你真的进过这种项目，你会发现，真正让项目死掉的，往往不是代码烂这件事本身，而是下面这三件事叠在一起：

一、量太大，看不完

54万行总代码量，不会有疯子真的准备自己去读完的...

这不是说多看几天总能看完的问题，而是你压根没有办法在短时间内建立对整个系统足够可靠的理解。

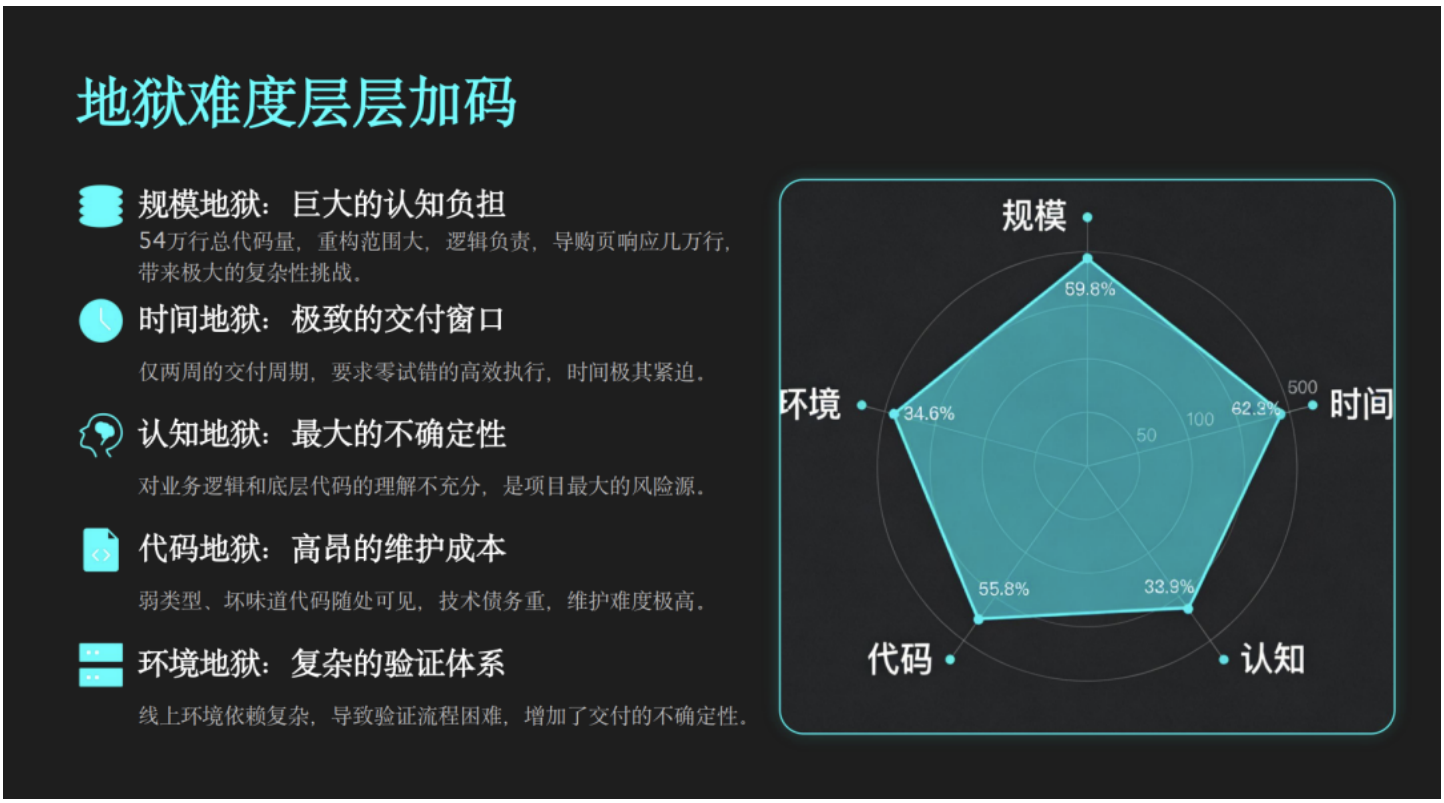
尤其是核心电商系统，导购、商详、库存、价格、促销、异常处理、兼容分支，往往全是互相牵扯的。

一个看起来不起眼的接口，后面可能挂着一串很长的调用链；一个响应对象里不起眼的字段，可能被前端、运营后台、外围服务同时依赖。

更麻烦的是，这种系统里通常还伴随着几个典型问题：

1. 文档不全，甚至很多地方根本没文档
2. 弱类型代码很多，字段真实类型并不稳定
3. 大量隐式逻辑和历史兼容藏在细节里
4. 代码坏味道明显，但又不敢随便动

所以它最大的难点，并不是读不懂某一段代码，而是**永远看不完**，也看不全。



二、时间太紧

如果时间够长，系统再复杂，也还有慢慢磨的可能。

但这次没有这个条件。两周，**首先意味着，公司层面压根不想听到重构两个字，他们会天然认为你们在浪费资源。我这边历史上的技术基建（技术债），都是加班完成**...

其次，他意味着你没有时间把旧系统完整读透，没有时间先补完文档，没有时间去做一轮从容的抽象设计，更没有时间走那条很多人习惯的老路：先全部理解清楚，再慢慢重构。

真实场景下没有尽量试试，而是**必须按时交付**。没有太多试错空间，也没有资格用还在理解中当借口。

这种行为，每一次重构，都相当于高速公路换轮胎，刺激的一逼！

三、风险极高

如果这是个边缘系统，一个内部后台，一个报表工具，那出问题还有修正空间。

但如果面对的是核心系统。这类系统最可怕的地方，不是它会不会直接挂，而是它很可能出现一种更麻烦的情况：**看起来能跑，但行为已经悄悄变了：**

金额少算一点，库存判断偏一次，导购页少走一个分支，看起来都是小事，但一旦放在线上流量里，就是真实损失。

所以，这里真实的难点是：

> 怎么在极短时间里，把一个巨大的未知系统，压缩成一个可分析、可验证、可灰度、可回滚的工程对象？

如果做不到这一点，AI 越能写，风险只会越大。

建立基线

很多团队做老系统重构，一开始都会掉进一个很自然的坑：先把旧系统彻底看懂，再开始改。

听上去很合理，但在高压交付里，这条路非常容易把自己耗死，**因为遗留系统很多时候根本没有彻底看懂这回事。**

尤其是跑了十年的老系统，它的真实行为和它当年的设计意图，往往早就不是一回事了。很多逻辑、很多分支、很多兼容，都是线上一点点演化出来的，你去追它本来应该怎么设计，经常会越追越偏。

所以我们这次真正改变战局的第一步，不是继续沿着先理解全貌的思路走，而是把策略切成了另一条路：**不先追求完整理解，而是先还原系统的真实行为。**

换句话说，我们不先问它本来该怎么设计，而是先问：

1. 这个接口现在实际收什么参数
2. 会经过哪些逻辑分支
3. 会返回什么结果
4. 会访问哪些数据库
5. 会产生哪些副作用
6. 异常情况下到底怎么表现

我们最后是以 URL 接口为最小单元，去做行为还原的。

这是一个非常关键的转向，因为一旦行为基线立住了，后面很多事情就都有了判断标准：

1. AI 生成的新代码到底对不对，有标准了
2. 测试用例该怎么写，有依据了
3. 差异分析该怎么看，有锚点了
4. 哪些是优化，哪些其实已经改行为了，也能区分出来了

没有这一步，重构就很容易从迁移变成偷偷重写。

而老系统重构最怕的，不是代码写不出来，而是你在不知不觉中把原有行为改掉了，还以为自己是在做优化。

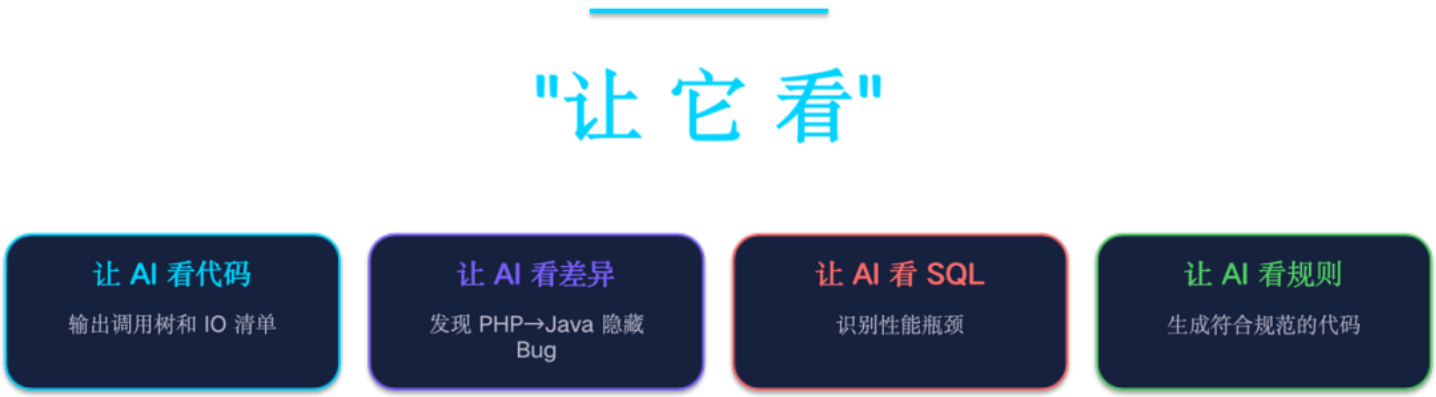
所以我们最后的核心策略其实很朴素：**任务驱动，先完成，再优化。**

先把接口真实行为摸清楚，先把一致性做出来，先把结果对齐，再去谈结构优化、性能提升、代码优雅（这个想想算了吧，我又不读）。

几个核心点

其实有个问题：**AI 在这次项目里，真正改变的是哪几件事？**

很多人一说 AI Coding，第一反应还是 AI 帮忙写代码。但说实话，这次项目让我更强烈地感受到，**AI 在复杂重构里最先改变的，其实不是编码速度，而是另外三件更底层的事。**



AI → 大规模阅读、结构化分析、规范化输出

当这两者配合好了，重构效率的提升不是 10%、20%——而是数倍的量级。

一、它先帮我们把旧系统变成可以操作的对象

面对 54 万行 PHP，AI 在这里最大的价值，不是替我们理解系统的伟大设计，而是帮助我们把一堆庞杂、散乱、无文档的代码，快速压缩成一些可读、可讨论、可验证的结构化结果。

比如：

1. 批量梳理接口调用关系
2. 提炼参数逻辑和异常分支
3. 生成架构图、逻辑图、说明文档
4. 提取入参、出参、数据库操作、副作用行为
5. 辅助识别弱类型字段的真实数据分布

这些事情，如果全靠人工，一样能做，但速度会非常慢，而且很容易因为人脑负荷过高而失去整体感。

所以我现在越来越倾向于一个判断：

> **AI 在遗留系统改造里最大的价值，不是先替你写代码，而是先替你压缩认知成本**

它先把看不完、理不清、没法下手的系统，变成一个至少可以被分段推进、分段验证的工程对象。

二、大规模机械翻译和重复改造

第二个真正改变效率的地方，才是大家最容易感知到的，编码提速。

这次有大量 PHP 到 Java 的迁移工作。如果完全靠人工来做，纯机械翻译都足够把人拖死。更别说老系统里还有那么多坏味道、重复结构、弱类型转换和混乱的数据组织方式。

在这种场景下，AI 非常适合承担认知明确之后的大批量执行。

比如：

1. PHP 转 Java 初稿生成
2. 函数拆解与重构
3. 工具类提取
4. DTO 化改造
5. 强类型替换
6. 冗余逻辑清理

举个很具体的例子，在处理 getListProductInfo 这类函数时，AI 会先拆解原 PHP 逻辑，把调用链和分支整理出来，再自动翻译成 Java，顺手提取独立工具类，优化嵌套逻辑，最后生成差异清单供人工审核。

这样一来，人工的注意力就不用再被机械搬运拖死，而是能集中在真正值钱的地方：

1. 业务逻辑有没有偏
2. 新旧行为有没有变
3. 边界情况有没有漏
4. 有没有把不该动的兼容逻辑顺手动掉

这里有个边界特别重要：**AI 可以吞掉翻译成本，但不能替代业务判断。**

它很适合做高效执行者，不适合做最终决策者。

三、验证、比对、排障

很多人以为写代码是交付里最费时间的环节，但真正做过项目的人都知道，很多时候更费时间的是后面的验证、比对、调试和收口。

尤其是老系统迁移，最大的挑战不是把新代码写出来，而是证明它和旧系统在关键行为上是一致的。

这次项目里，AI 在这一块的价值，甚至不比编码低：

1. 自动生成测试用例，覆盖正常、异常、边界场景
2. 辅助白盒测试，把原来依赖人工经验的验证转成可执行脚本
3. 辅助日志分析和 Debug，帮忙复现问题、定位 Bug

4. 自动生成新旧差异清单，提示哪些差异是优化，哪些可能是行为变化

如果说编码只是把东西做出来，那验证和排障才是真正决定它能不能进生产环境的地方。AI 在这里的价值，比自动补全几段代码要硬得多。

人的价值更多了

这次项目里，我们一直坚持一个很明确的原则：**AI 扛效率，人控边界。**

因为 AI 最大的风险，从来不是它不会写，而是它看起来写得很像那么回事。

它最危险的时候，往往不是报错的时候，而是它交给你一段局部看起来完全合理的代码，但从系统角度看，已经埋了坑。

所以人的职责在这类项目里反而更清楚了。

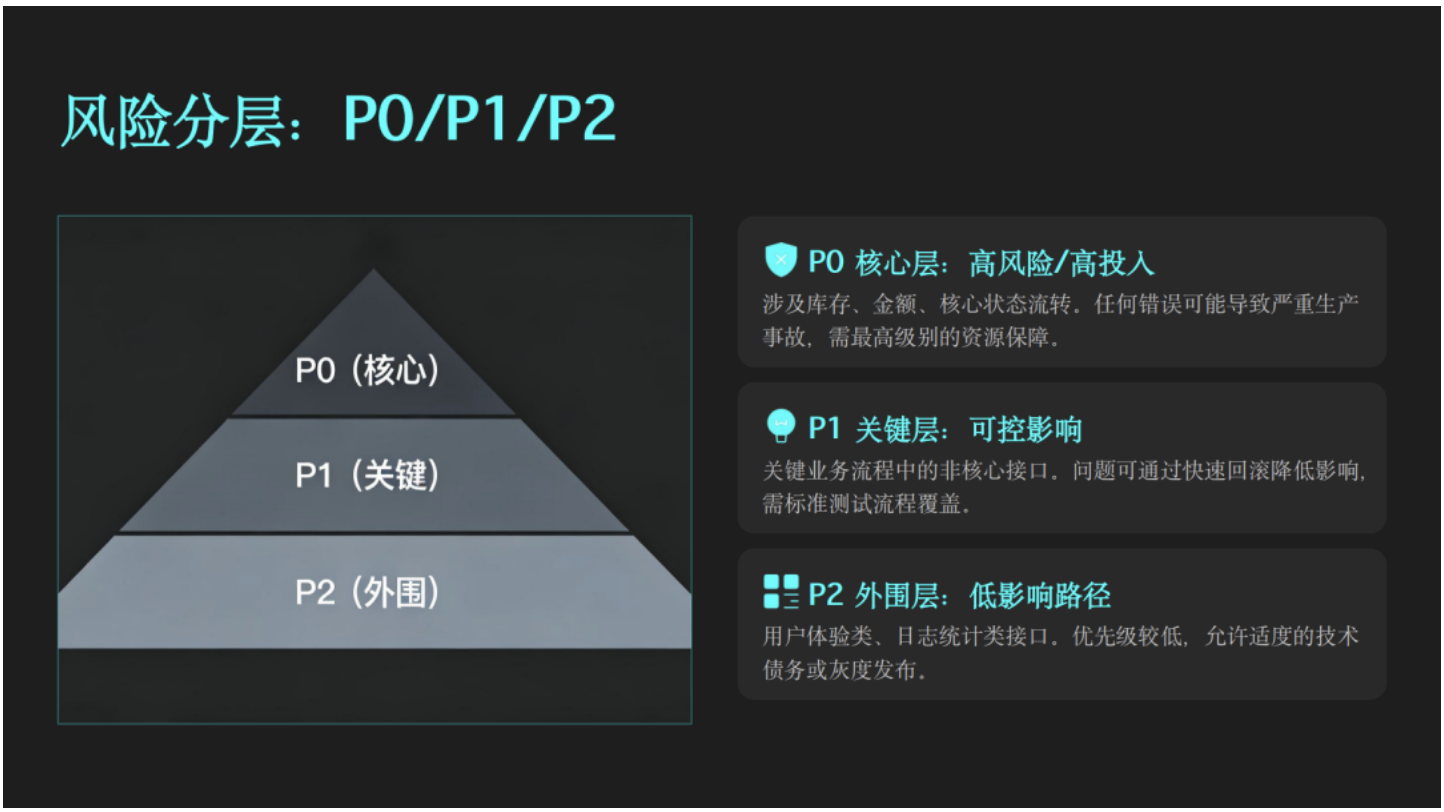
第一，边界和标准必须由人来定

什么能改，什么不能改；重构范围在哪；哪些接口必须 100% 行为一致；哪些地方允许后续再优化；哪些兼容逻辑绝不能碰，这些事都不能交给模型自己猜。

我们当时就明确按 P0、P1、P2 做了风险分层：

- P0 核心层：人工主导，AI 辅助
- P1 关键层：AI 主导实现，人工严格审核
- P2 外围层：AI 自主度更高，允许一定技术债务

不是所有代码都值得同样的谨慎，也不是所有代码都适合同样的自动化程度。



第二，核心决策不能交出去

所以我们在整个编码过程里都强制要求 Plan 模式。

不是让 AI 看完需求就直接改，而是先让它输出完整实施计划：改哪些文件、分几步做、风险点在哪、影响范围是什么、哪些地方需要确认。

人工先审计划，再允许它执行。

Plan 模式最大的价值，不是形式完整，而是把 AI 先开干改成了 AI 先暴露思路。

很多风险，其实在计划阶段就能看出来：**它有没有偷偷扩大改动范围？有没有擅自做抽象？有没有把平迁理解成顺便优化架构？**

这些东西等代码都改完了再发现，成本就高很多。

负责到底

AI 很容易在局部看上去很聪明，但从系统角度看却埋下隐患。

最典型的一类问题，就是接口通了，代码也跑了，但全链路行为已经悄悄变了。

比如我们遇到过一个特别典型的坑：AI 把 Result<T> 替换成 BaseResponseDTO<T>。从抽象视角看，两者差不多，都是返回对象。但一细看就会发现：

1. message 变成了 msg
2. costTime 字段丢了
3. 调用方原本依赖 message 字段拿值，现在直接拿不到

这种问题最危险，因为后端看起来服务正常，但前端和调用链已经悄悄坏了。

所以后来我们明确加了一条规则：**改动前必须先对比两个类的完整结构。**

字段名有没有变化、字段有没有缺失、默认值有没有变化、调用方依赖有没有受影响，都要做检查清单。

这类坑特别能说明一件事：**重构里真正致命的，往往不是大崩溃，而是这种悄悄改变行为的小错。**

工程化

这次项目如果一定要提炼方法论，我觉得最重要的不是用了哪个模型，而是我们逐渐沉淀出了一套能让 AI 在复杂工程里真正起作用的做法：

- 第一，**先按风险分层，不平均用力。**

20% 的接口承载了 80% 的风险，不把这件事拆清楚，后面所有资源投入都会失真。

- 第二，**强制 Plan 模式，先出方案，再动代码。**

AI 先暴露思路，人再决定让不让它往下走。

- 第三，**把 Prompt 从提问升级成规则系统。**

比如：

1. 全部按照原代码实现，不要擅自简化
2. 翻译 PHP 到 Java 时不要应用反射
3. PHP 中是 API 访问的，统一封装到 Feign
4. 在进行任何改动前，必须确认不会影响其他逻辑
5. 有疑问就问，不允许 AI 自作主张
6. Bug 分析必须端到端追踪完整数据流

这些看起来不像什么神奇提示词，但它们比花哨提示语有价值得多，因为它们本质上是在把团队经验沉淀成 AI 可执行的规则。

- 第四，**把能跑拆成三层：能跑、能对、能稳。**

能跑：代码能编译、能启动、能走通

能对：关键行为和旧系统一致

能稳：性能、并发、可维护性、灰度发布、回滚能力都达标

只有这样，重构才不只是代码搬过去了，而是真正能进生产环境。

上线要点：灰度机制

很多重构项目做到后面最容易产生一种错觉：代码写差不多了，测试也跑过几轮，应该可以上了。

但核心系统不是靠感觉差不多上线的。你不能因为 AI 说逻辑一致，就相信它一致；也不能因为几个页面看起来正常，就认为它稳定。

你必须用一整套可追溯的证据，证明新系统和旧系统在关键行为上是对齐的。所以在验证阶段，我们采用的是 AI 自动化加人工校验的组合：

1. 自动生成测试用例
2. 差异分析生成清单
3. 白盒测试脚本辅助
4. 日志分析与排障支持

而上线策略也绝不是一键切全量。我们最后采用的是：

1. 影子验证
2. 小流量灰度
3. 分批切流
4. 全量替换
5. 观察复盘

同时，回滚预案提前准备好，确保真出问题时能分钟级回滚。

因为真正出线上问题的时候，现场是没有时间慢慢思考的。**回滚方案提前写好，本质上是在为团队争取第二次修正机会。**

结语

做完这次项目之后，我对 AI Coding 有一个更明确的判断：

过去大家很喜欢用一些轻松场景去证明 AI 厉害，比如写页面、补功能、起 Demo。这些当然有价值，但很多还停留在能用、好用、有点惊喜的层面。

而这次不一样。

这是一个运行 10 年的老系统，是 54 万行遗留 PHP 代码，是两周交付窗口，是核心电商业务，是必须稳定上线的真实工程任务。这样的场景，不再是展示场景，而是非常硬的交付场景。

它至少说明了三件事。

遗留系统改造被AI覆盖了

遗留系统最痛的，他理解成本高、重复劳动多、验证链条长，这些恰恰都是 AI 最擅长提效的地方。

我们原来觉得恼火的部分，反而是 AI 很擅长的部分。

边界很重要

会写提示词，不等于会做交付；会调模型，不等于会做工程。真正的门槛，是风险拆解、边界定义、验证设计、灰度发布和回滚兜底。

有人会失业

以后工程师更重要的能力，可能不再只是我能写出一段多漂亮的代码，而是我能不能快速理解系统、拆解问题、识别边界、设计工作流，并对最终结果负责。

AI 不会替你为系统承担责任，但它会把那些原本压得人喘不过气的脏活、累活、重复活，成片地吞掉。

谁先学会和它协同，谁就更有机会接住那些过去根本不敢接的硬仗；至于学不会的同学，那就危险咯...

最后补一句：**从去年下半年我们开始验证，AI Coding 现在真的很强了，大家积极拥抱吧...**