

LEARNING NOTE 03

AI 产品经理学习笔记 03

从 Agent 到完整 AI 产品闭环

这是我的 AI 产品经理学习记录，也是“第一层：保底必学”的最后一部分。前两篇分别整理了传统产品基本功，以及大模型、Prompt 和 RAG。这一篇继续往后，把 Agent、API 与工程基础、数据指标、Bad Case 和技术方案选择串起来，最后再用一个完整的 AI 游戏内容助手案例，把前面学习过的知识重新走一遍。

到这里，我希望自己至少能够从产品角度解释清楚：一个 AI 功能为什么要做、为什么需要 AI、模型如何接入产品、失败以后怎么办、怎样判断效果，以及下一版本应该优化哪里。

本篇属于：第一层·保底必学 | 第五至第十章·阶段性完成

学习记录·公开整理版

2026 · AI Product Management

第五章：Agent 基础

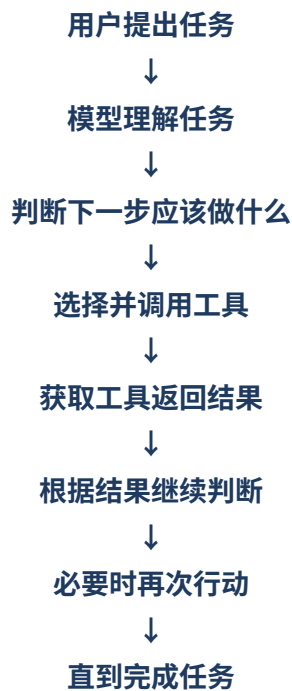
1. Agent 到底是什么？

如果暂时不讨论复杂的 Agent 架构，可以先从最简单的区别开始理解。普通的大模型交互通常更接近：



用户提出一个问题，模型根据当前上下文生成回答。

而 Agent 更接近：



相比单纯“生成一个回答”，Agent 更强调：判断 → 行动 → 获取结果 → 再判断。

模型不再只是回答问题，而是参与任务执行过程。比如用户提出：“帮我看看东京明天的天气，并根据天气判断要不要带伞。”模型本身不需要“记住”明天天气，而是需要判断这个任务需要实时信息，调用天气工具，获取结果，再根据工具返回的数据形成建议。

这和直接让模型凭已有知识回答，是完全不同的产品逻辑。

2. Tool 是什么？

Tool 可以理解成模型能够调用的外部能力。例如：

- Search：搜索网页；
- Calendar：读取或创建日历事件；
- Database：查询数据库；
- Email：读取或发送邮件；
- Weather：查询天气；
- Calculator：执行计算；
- Internal API：调用企业内部系统。

大模型本身并不拥有这些外部系统里的实时信息，也不能天然完成真实世界中的操作。但是，如果产品把这些能力以 Tool 的形式提供给模型，模型就可以根据任务判断：“我现在应该调用哪个工具？”

Tool 的价值，是让模型从“只会生成内容”进一步走向“能够访问外部信息或执行动作”。

3. 什么是 Function Calling？

Function Calling 可以先理解成：告诉模型当前有哪些函数或工具可以调用，以及调用这些工具需要什么参数。

```
get_weather(city)
```

如果用户问：“东京天气怎么样？”模型可以判断需要调用 get_weather，并生成所需参数：

```
function: get_weather  
city: "Tokyo"
```

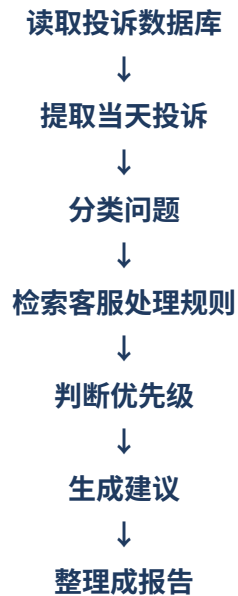
然后真正的天气服务完成查询并返回数据，模型再基于结果组织成自然语言答案。

模型负责决定是否调用工具以及如何提供参数；真正的数据查询或动作执行，是外部工具完成的。

4. 为什么 Agent 很重要？

因为真实世界里的很多任务，并不是“输入一句话 → 输出一句话”，而是需要理解任务、拆解步骤、寻找信息、调用系统、观察结果，再决定下一步。

例如：“帮我整理今天的客户投诉，并生成对应的处理建议。”这个任务可能需要：



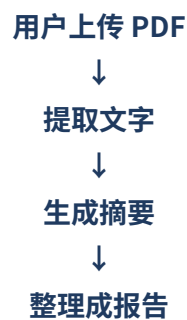
这里已经不是单纯的一次文本生成任务。模型需要和多个数据源、工具以及业务流程发生交互。

Agent 的价值，就是让 AI 有机会参与这种更加复杂的任务执行。

5. Workflow 和 Agent 有什么区别？

两者都可以处理多步骤任务，但最大的区别之一在于：任务执行路径是不是提前确定的。

Workflow：流程提前确定



无论用户上传什么 PDF，系统基本都会按照这条流程执行。产品设计阶段已经知道第一步、第二步、第三步分别是什么。模型可能参与其中某一步，但整体流程是预先编排好的。

Agent：执行路径需要动态判断

例如用户说：“帮我研究一下这个竞争对手。”这个任务并没有唯一固定流程。Agent 可能需要判断：

- 是否搜索官网？

- 是否查看产品价格？
- 是否搜索用户评论？
- 是否查看应用商店？
- 当前信息是否已经足够？
- 还需不需要继续搜索？什么时候停止？

流程基本固定 → 优先考虑 *Workflow*；需要根据任务动态决定下一步 → 才进一步考虑 *Agent*。

这里最需要避免的是：为了使用 Agent，而把本来可以简单解决的问题做复杂。如果三步固定 Workflow 已经能够稳定完成任务，就没有必要因为“Agent 更像 AI”而强行改成 Agent。

第六章：API 与工程基础

1. API 是什么？

API 可以简单理解成：两个软件系统之间按照约定规则进行通信的接口。一个 AI 产品可能是：



作为 AI 产品经理，不一定需要自己完成所有后端开发，但至少应该能够理解产品怎样把用户输入发送给模型，又怎样拿到模型返回结果。否则讨论 Prompt、Token、Streaming、Timeout、Retry 等问题时，很容易只停留在概念层。

2. Request 与 Response

Request | 请求

Request 是你的系统发送出去的信息，例如：

```
{
  "model": "xxx",
  "input": "你好"
}
```

请求里还可能包含 Prompt、上下文、参数、工具定义等信息。

Response | 响应

服务器处理完成以后返回结果，例如：

```
{
  "output": "你好，有什么可以帮助你的？"
}
```

真实 API 的结构可能比这个复杂很多。但作为 AI PM，至少应该能够看懂：请求发了什么、服务返回什么，以及产品应该怎样继续处理。

3. JSON

JSON 是 AI 产品开发过程中非常常见的数据格式。例如：

```
{
  "name": "Alice",
  "age": 20,
  "skills": ["Python", "Figma"]
}
```

第一阶段至少需要能够理解：

- Object | 对象；
- Array | 数组；
- Field | 字段；
- String | 字符串；
- Number | 数字；
- Boolean | 布尔值。

我并不认为 AI 产品经理一定要成为专业程序员。但如果连一个基本 JSON 都无法阅读，那么讨论 API、Structured Output、Tool Calling、Workflow 和 Agent 都会变得非常困难。

不一定要能从零写复杂后端，但至少应该能够看懂产品和模型之间正在交换什么数据。

4. HTTP 状态码

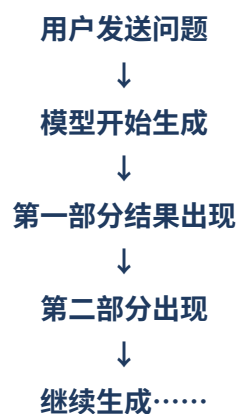
调用 API 时，经常会遇到一些状态码。第一阶段至少需要认识：

- 200 请求成功
- 400 请求本身存在问题
- 401 身份认证失败或缺少认证
- 403 当前身份没有访问权限
- 404 请求的资源不存在
- 429 请求超过当前服务允许的限制
- 500 服务器端发生异常

这些状态码背后的具体含义还会受到 API 服务实现方式影响，但至少要知道：模型没有返回结果，并不代表所有失败都是同一种失败。不同错误，需要不同处理策略。

5. Streaming

普通模式可能是：用户发送问题后等待一段时间，最后一次性看到完整答案。Streaming 则更接近：模型一边生成，用户一边看到结果逐渐出现。



系统实际总延迟和用户感受到的延迟，并不是完全相同的概念。

有时总生成时间并没有大幅缩短，但因为用户很快看到了第一个字，主观等待体验已经明显不同。

6. Token

Token 可以先粗略理解成模型处理文本时使用的基本单位。它并不严格等于一个汉字、一个英文单词或者一个字符。

在产品层面，最重要的是理解：模型的调用成本、上下文容量等问题通常都会受到 token 数量影响。比如 Prompt 很长、历史对话很多、RAG 一次塞入大量文档、Tool 返回内容很长，都可能导致：

- 成本增加；
- 响应变慢；
- 上下文更加拥挤。

所以 Token 并不是只有开发人员才需要关注的工程指标，它也会影响产品设计。

7. Context Window

Context Window 可以粗略理解成模型在一次任务中能够处理的上下文容量范围。一个 AI 请求里可能同时包含：

```
System Prompt
+
用户当前问题
+
历史聊天
+
RAG 检索结果
+
Tool 返回结果
```

这些内容都需要占用 Context。所以 Context 不是无限空间。产品设计不仅需要考虑“能不能放进去”，还应该考虑“有没有必要放进去”。

Context Window 是模型能力边界之一，而 Context Management 是产品设计问题。

8. Temperature

Temperature 可以先理解成影响模型输出分布的一类生成参数。在一些模型和接口中，较低的设置通常更偏稳定，较高的设置可能带来更大的输出变化。

不要把 Temperature 简化成一个万能的“创造力按钮”。

产品最终需要的不是“理论上这个参数应该更有创造力”，而是在我的具体任务和评测集上，哪个配置实际表现更好。所以最后还是要回到测试和 Evaluation。

9. Timeout

如果模型一直不返回结果，产品不能让用户无限等待。可以设计预设超时、终止当前请求、提示用户当前请求未完成，并提供重新尝试入口。具体多少秒应该根据业务、模型和产品体验测试决定。

所有外部模型调用都有失败和超时的可能，产品必须提前设计异常状态。

10. Retry

有些失败属于暂时性问题，因此第一次请求失败以后，系统可以尝试重新发起请求，这就是 Retry。

但 Retry 不能无限进行，否则可能带来请求数量增加、Token 成本增加、服务压力放大，甚至把局部故障进一步放大。Retry 需要次数限制、间隔策略以及最终失败处理。

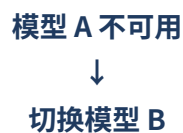
11. Rate Limit

模型服务通常会对一定时间内的请求次数、Token 使用量或并发量等做限制。超过限制以后，请求可能被拒绝或者延后。

这会直接带来产品问题：高峰流量怎么办？核心任务如何保证？是否需要排队？是否应该限制频率？是否需要切换其他服务？这些已经不只是模型问题，而是产品稳定性问题。

12. Fallback / Degradation | 降级

如果主要模型出现故障，产品是不是就完全不可用了？一种思路是设计 Fallback。例如：



或者在高性能模型持续超时，让部分任务切换到其他可用模型。但降级并不只是“换一个模型”这么简单，因为不同模型之间可能存在能力差异、输出格式差异、工具支持差异和成本差异。真正的降级策略需要提前测试。

AI 产品不能只设计“最佳情况”，还要设计服务能力下降时如何继续运行。

13. 一个 AI 请求的完整链路

学习到这里以后，我认为脑中应该逐渐形成一条比较完整的链路：

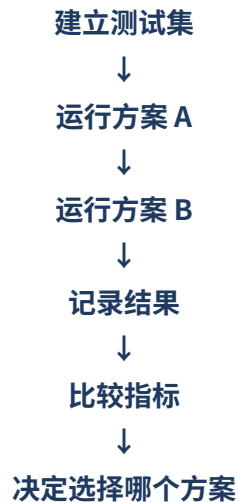


Prompt、Context、RAG、Tool Calling、Agent、Evaluation、异常处理、Latency、Cost 和 Bad Case，并不是彼此孤立的概念，而是在这条 AI 请求链路的不同位置解决不同问题。

第七章：基础数据能力

1. 为什么 AI 产品经理必须懂数据？

如果一个 Prompt 修改以后，我们只是说：“我感觉现在回答好多了。”其实很难证明它真的变好了。更合理的方式应该逐渐变成：



尽可能把“感觉更好”变成“有证据说明哪里更好”。

因为模型具有概率性，Evaluation 对 AI 产品尤其重要。

2. 一些基础指标

Count | 数量

例如：本周一共生成了 10,000 次内容。

Ratio | 比例

例如生成成功率：成功生成次数 ÷ 总生成次数。

Average | 平均值

例如平均响应时间。但平均值有时容易受到极端数据影响。

Median | 中位数

如果大部分请求只需要 2 秒，但少数异常请求需要几十秒，那么只看平均值可能无法完整描述用户体验。这时中位数等指标也会提供不同角度的信息。

现阶段最重要的不是背统计公式，而是知道不同指标描述的是数据不同侧面。

3. 产品漏斗



通过这条漏斗就可以继续问：为什么一半用户进入以后没有输入？为什么有请求没有成功生成？为什么用户看到结果以后没有采用？采用以后为什么没有回来？

漏斗真正的价值不是得到几个百分比，而是帮助产品定位用户主要在哪一个环节流失。

4. AI 产品常见指标

产品指标

- DAU;
- 留存;
- 使用次数;
- 完成率;
- 转化率。

AI 效果指标

- 正确率;
- 任务成功率;
- 幻觉相关指标;
- 格式正确率;
- 指令遵循情况。

具体使用哪个指标，应该根据任务类型设计。

用户行为指标

- 重新生成率；
- 编辑率；
- 采纳率；
- 点赞 / 点踩；
- 人工接管率。

这些行为可以帮助判断用户到底怎样对待 AI 输出。例如高编辑率可能意味着结果有一定价值，但距离直接使用仍然存在差距。

工程指标

- Latency；
- Error Rate；
- Timeout Rate。

即使模型回答质量很好，如果经常超时或者接口失败，仍然不是一个稳定产品。

成本指标

- Token 使用量；
- 单次请求成本；
- 单用户成本。

AI 产品最终也必须考虑经济性。一个效果提高非常有限、但调用成本翻数倍的方案，不一定就是更好的产品方案。

第八章：Bad Case

1. 什么是 Bad Case？

Bad Case 可以简单理解成：AI 没有按照产品预期完成任务的案例。

例如用户要求“推荐三个角色”，模型却输出了四个。哪怕这四个角色本身推荐得都很好，这依然是一个 Bad Case，因为模型没有正确遵循任务要求。

2. Bad Case 不能只收藏

如果只是建立一个文件夹，保存“错误案例 1、错误案例 2、错误案例 3……”，价值其实很有限。更重要的是给错误分类。

例如统计 100 个 Bad Case：

- 35% 事实错误
- 25% 格式错误
- 20% 信息遗漏
- 10% RAG 检索问题
- 5% 工具调用失败
- 5% 其他问题

这时就能开始回答：下一版本究竟应该优先优化哪里？如果最大问题是事实错误，就没有必要把大量时间都花在 UI 改版上；如果绝大多数错误来自 RAG Retrieval，也不能简单通过无限增加 Prompt 规则解决。

3. Bad Case 的基本处理流程



发现

从线上数据、测试集、用户反馈等地方发现异常。

记录

不能只记一句“模型回答不好”。需要尽量保留用户输入、模型输出、当时使用的 Prompt、Context、模型版本、是否调用 RAG / Tool，以及具体失败现象。

分类

判断属于事实问题、格式问题、Instruction Following、Retrieval、Tool Calling，还是其他问题。

统计

看哪类问题占比最高，避免被少数印象深刻的案例带偏。

找根因

这是最重要的一步。例如“答案错了”背后，可能是 RAG 没找到资料，也可能是资料找对了但模型理解错了，甚至可能是产品一开始就没有给用户提供足够输入。表面现象相同，解决方案却完全不同。

提出方案并重新测试

优化以后重新运行测试集。不能因为原来的一个 Bad Case 修好了，就认为问题解决，还需要观察会不会修改 A 以后，把 B 又弄坏了。这也就是回归测试的意义。

第九章：规则、Prompt、RAG、Tool、Agent 到底怎么选？

学到这里以后，一个非常重要的问题是：面对一个需求，我到底应该使用哪种方案？并不是看到所有问题都先想“这里能不能加一个大模型？”

景一：能定解

例如判断用户年龄是否 ≥ 18 。这个任务已经拥有明确规则：

```
age >= 18
```

使用普通程序判断更加简单、稳定、便宜，没有必要让大模型进行概率判断。

场景二：需要自然语言理解或生成

例如把一段文字改成更加正式的表达。这类任务没有唯一固定规则，又属于大模型比较擅长的自然语言转换，可以考虑：

LLM + Prompt

场景三：需要外部知识

例如回答公司内部的报销制度。模型本身不知道企业内部文件，这时可以考虑：

Retrieval + LLM

即 RAG 类方案

场景四：需要获取实时外部信息

例如查询今天的天气。如果产品已经提供天气 API，可以考虑：

LLM + Tool Calling

模型负责判断应该查询天气并构造工具参数，真正的数据由天气服务返回。

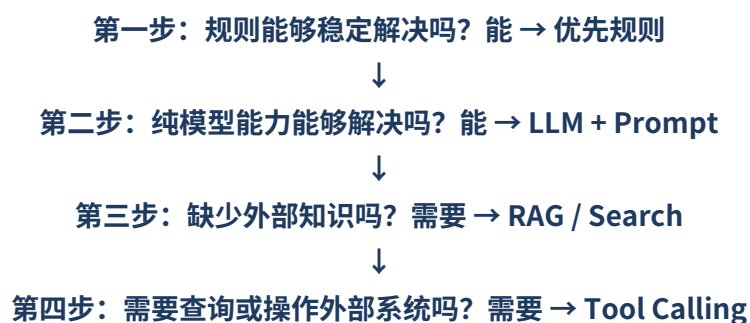
场景五：需要操作外部系统

例如：“帮我读取邮件，找出需要回复的内容，然后看看我的日历什么时候有时间安排会议。”这个任务可能同时涉及 Email Tool 和 Calendar Tool，已经不只是生成文字，而是在调用真实外部能力。

场景六：需要动态规划多步骤任务

如果任务还需要根据中间结果动态决定下一步搜索什么、调用哪个工具、是否已经完成、是否还需要继续，这时候才进一步考虑 Agent。

一个简单的选型顺序





第五步：任务执行路径需要动态规划吗？需要 → Agent

这并不是一个绝对技术公式，真实产品可能组合多种方案。但它提醒我一个很重要的原则：

优先使用能够稳定解决问题的最简单方案。

所以真正应该问的问题不是“这里能不能用 AI？”，而是“这里使用 AI，是否真的比传统方案更加合适？”

第十章：把前面的内容串起来——AI 游戏内容助手

最后尝试用一个简单案例，把第一层学到的内容重新串起来。假设现在需要设计一个 AI 游戏内容助手，目标用户是游戏内容运营。

Step 1：用户需求

运营人员在角色活动上线前，需要准备多版营销文案。目前可能需要：

查看角色设定



寻找角色卖点



撰写初稿



反复修改



审核

当需要快速准备多个版本时，人工重复工作量会增加。所以要解决的问题是：如何更高效地生成符合角色设定的营销文案初稿。

Step 2：为什么考虑 AI？

这个任务主要涉及自然语言理解、信息提取、文本生成和风格控制，这些属于 LLM 比较适合参与的任务。

引入 AI 的原因不是“游戏行业应该用 AI”，而是任务本身与大模型的能力存在匹配。

Step 3 : Input

产品首先需要明确模型需要什么信息，例如：

- 角色名；
- 角色设定；
- 活动主题；
- 目标用户；
- 文案长度。

如果角色信息完全没有提供，却希望模型保证角色一致性，本身就是不合理的产品设计。

Step 4 : Prompt

例如 System Prompt 可以定义：“你是一名游戏内容运营助手。” Task 是根据提供的角色信息和活动主题生成角色宣传文案。Constraints 可以包括：

- 不得修改已有角色设定；
- 不得捏造未提供的剧情；
- 输出长度符合要求。

Prompt 的目标不是堆很多规则，而是让任务边界更加明确。

Step 5 : Output

如果后续产品需要结构化展示，可以要求：

```
{  
  "title": "",  
  "selling_point": "",  
  "copy": ""  
}
```

这样前端可以分别展示标题、卖点和正文，而不是把所有内容混成一整段自然语言。

Step 6 : 用户流程

填写角色信息





如果满意，可以进入编辑并采用；如果不满意，可以调整要求后重新生成。AI 生成并不一定就是最终结果，人工编辑本身可以是合理流程的一部分。

Step 7：异常设计

模型超时

提示用户当前生成未完成，并提供重新尝试。

角色信息不足

不要让模型随便补全，可以提示用户补充角色性格或关键设定。

输出格式错误

系统进行格式检查，并根据产品策略重新生成或者提示异常。

内容违反安全或业务规则

执行相应的拦截或人工审核流程。

不能默认模型每一次都按照预期返回正确结果。

Step 8：指标

产品指标

- 使用人数；
- 使用频率；
- 任务完成率。

AI 效果指标

- 生成成功率；

- 格式正确率；
- 角色一致性。

用户行为指标

- 重新生成率；
- 采用率；
- 人工编辑率。

工程指标

- 平均响应时间；
- Error Rate；
- Timeout Rate。

成本指标

- Token 使用；
- 单次生成成本。

Step 9 : Bad Case

人物性格错误

可能属于角色一致性问题，需要继续判断角色信息是否完整、模型是否正确使用 Context。

文案过长

可能属于 Instruction Following 问题，需要检查长度要求是否明确、模型能否稳定遵循，以及是否需要程序层进一步校验。

捏造剧情

可能属于事实性 / Grounding 问题，需要判断模型是不是缺少角色知识，是否应该增加可靠的角色设定来源，或者引入 RAG。

Step 10 : 下一版本

这里不能简单说“继续优化模型”，而应该根据 Bad Case 分别处理。例如：

角色资料不足 → 优化输入和 Context

大量事实错误 → 考虑增加可靠知识来源

格式经常错误 → 加强 Structured Output 和程序校验

Latency 太高 → 检查模型、Prompt、Context 和调用链路

重新生成率很高 → 分析用户为什么不满意

到这里，一个相对完整的 AI 产品闭环才逐渐形成：



在第一束之后

整理完第 0 章到第 10 章以后，我重新回头看了一遍最开始的问题：AI 产品经理到底应该先学什么？

最开始，我很容易把 AI 产品经理理解成 Prompt、RAG、Agent、大模型，因为这些词看起来最有“AI 感”。但把第一层完整整理下来以后，我现在更愿意把它理解成一条完整链路：





其中任何一个环节单独拿出来，都不能代表完整的 AI 产品能力。只会写 Prompt，不等于会做 AI 产品；会搭 RAG，也不代表需求一定成立；用了 Agent，更不意味着产品就更加先进；模型效果很好，如果延迟太高、成本不可接受、错误无法处理，同样可能不是一个能够真正使用的产品。

第一，先问问题，再问 AI

不要看到一个业务场景以后第一反应就是“能不能加 AI”。先问用户到底有什么问题。如果普通规则能够更稳定、更便宜地解决，就没有必要强行使用大模型。

第二，选择最合适的方案，而不是最复杂的方案

如果 Prompt 能解决，就不一定需要 RAG；如果固定 Workflow 能解决，就不一定需要 Agent。复杂并不自动等于先进。产品最终需要的是稳定解决用户问题。

第三，AI 产品必须默认模型会失败

幻觉、格式错误、超时、工具失败、RAG 召回错误、Rate Limit.....这些都不是只存在于极端情况的理论风险。

异常处理不是补充功能，而是 AI 产品设计的一部分。

第四，没有 Evaluation，就很难真正迭代 AI

如果所有优化最终都停留在“这个版本感觉比之前聪明”，产品很难形成稳定迭代。需要逐渐建立：



第五，AI 产品不是一个模型

这是我整理完第一层以后最明确的一点。一个真正的 AI 产品更接近：



模型非常重要，但模型并不是整个产品。

第一层：阶段性完成

到这里，我的“AI 产品经理学习笔记 | 第一层：保底必学”暂时整理完成。

- 第 0 章 AI 产品岗的基础门槛
- 第 1 章 传统产品基本功
- 第 2 章 大模型基础认知
- 第 3 章 Prompt Engineering
- 第 4 章 RAG 基础
- 第 5 章 Agent 基础
- 第 6 章 API 与工程基础
- 第 7 章 基础数据能力
- 第 8 章 Bad Case

第 9 章 规则、Prompt、RAG、Tool、Agent 的方案选择

第 10 章 基础 AI 产品案例

它仍然只是一套基础框架。很多内容现在只是建立了第一层认知：Agent 还需要继续学习更复杂的任务规划与工具协作；RAG 后面还会涉及 Retrieval、Reranking 和 Evaluation；数据能力还需要真正进入测试集和实验；Evaluation 也需要建立更加系统的方法；真正的 AI 产品最终还需要在实际项目中不断验证。

这里的“完成”，不是“我已经学会了 AI 产品经理”，而是“我终于建立了一张可以继续往下学习的基础地图”。

后面的学习，就不再只是不断收集 AI 名词，而是开始沿着这张地图，继续把每一块能力做深、做实，并放进真正的产品和项目里验证。

学习 AI 产品，不是为了证明自己知道多少技术名词。最终还是要回到一个问题——能不能用合适的方法，把真实的问题解决得更好。

说明：本文为个人学习记录的公开整理版，内容用于记录当前阶段对 AI 产品基础能力的理解，后续会随着学习和实践继续修正。