AI 产品经理学习笔记 04

从“会做 AI Demo”到“能完成 AI 产品闭环”

第二层：核心能力｜第 0～3 章

个人学习记录 · 持续整理

> **第一层结束以后，我已经能够大致解释一个 AI 产品为什么要做、什么时候需要 Prompt / RAG / Agent，以及一个基础 AI 请求链路是怎样运行的。**

但继续往下学习以后，我开始意识到：“知道这些东西是什么”和“真正能够把 AI 产品做好”之间，还有很长一段距离。

第二层开始关注的，不再只是“能不能把功能做出来”，而是当系统真正出现问题时，我能不能定位问题、设计评测、验证方案，并最终形成一套可以持续迭代的产品闭环。

这一部分，我先从 RAG 开始。

第 0 章：第二层到底在学什么？

第一层主要解决的是：

> **我是否理解一个 AI 产品是怎样工作的？**

例如：

- 什么是 Prompt；
- 什么是 RAG；
- 什么是 Agent；
- 模型怎样通过 API 接入产品；
- AI 为什么存在概率性；
- 什么是 Bad Case；
- 产品为什么需要 Evaluation。

到了第二层，仅仅“知道这些概念”已经不够了。

这一层更希望解决的是：

> **当 AI 产品真的出现问题以后，我能不能找到问题、验证问题、解决问题，并证明下一版本确实比上一版本更好？**

所以第二层不再满足于：

- 会写 Prompt；
- 会搭一个知识库；
- 会调用 Agent；
- 会做一个 AI Demo；
- 知道几个 AI 指标。

而是需要进一步回答：

1. 为什么选择这个 AI 方案？
2. 系统完整链路是什么？
3. 哪一个环节最容易出错？
4. 出错以后应该从哪里开始定位？
5. 怎样建立 Evaluation？
6. 怎样证明 V2 比 V1 更好？
7. 怎样避免优化一个指标，却伤害了另一个指标？
8. 上线以后应该监控什么？
9. Agent 执行失败以后怎么办？
10. 怎样形成下一轮迭代？

整理到这里，我对第二层的理解逐渐变成了三个关键词：

> **AI 产品诊断能力 + Evaluation 能力 + 系统设计能力**

第一层让我知道：“系统里有哪些东西。”

第二层则开始要求我理解：“这些东西为什么会出问题，以及出了问题以后应该怎么办。”

第一章：第二层整体框架

我目前把第二层大致拆成五部分。

1. RAG 全链路

需要解决的问题是：模型怎样可靠地获得外部知识？

第一层学习 RAG 时，我更多理解的是：

```
用户问题
↓
检索资料
↓
把资料放进 Context
↓
模型回答
```

但真正进入第二层以后，会发现这里面的每一个环节都可能影响结果。

- 数据本身有没有问题?
- 文档有没有解析正确?
- Chunk 是怎么切的?
- Query 有没有表达完整?
- Retriever 有没有找到正确资料?
- 正确资料是不是排得太后?
- 最后送给模型的 Context 有没有组织好?
- 模型有没有根据资料回答?
- Citation 有没有对应正确来源?

所以 RAG 不再只是“搭一个向量知识库”，而是一整条需要分别设计和评测的链路。

2. Agent 与 Workflow

需要解决的问题是：模型怎样从“回答问题”进一步走向“完成任务”？

第一层已经区分过：固定流程更适合 Workflow；执行路径需要动态判断时，才进一步考虑 Agent。

- Agent 如何判断下一步?
- 工具为什么会调用失败?
- 任务执行到什么程度应该停止?
- 如果某一步失败怎么办?
- 什么时候应该重新规划?
- 什么时候应该让用户接管?

Agent 真正困难的地方，并不是“让模型调用一个 Tool”，而是怎样让整个任务执行过程稳定、可控，并且能够恢复。

3. Evaluation

Evaluation 解决的是：怎样证明 AI 真的变好了？

比如修改了一个 Prompt，不能只说：“现在感觉回答更自然了。”

```
测试集
↓
运行 V1
↓
运行 V2
↓
定义指标
↓
记录结果
↓
分析 Bad Case
↓
比较差异
```

最后才能回答：V2 到底有没有比 V1 更好？

这也是我认为从“做 AI Demo”走向“做 AI 产品”的一个重要分界。

4. 数据与实验

数据与实验关注：从结果中发现哪里有问题，以及下一步应该改什么。

例如：重新生成率为什么突然升高？采用率下降是不是因为生成质量变差？Latency 下降以后，是否牺牲了效果？模型升级以后，成本增加了多少？某一类用户的任务成功率是不是明显更低？

如果没有数据，只靠感觉，很难稳定判断 AI 产品应该往哪里迭代。

5. 工程化与异常处理

真实 AI 产品一定会遇到：

- 模型超时；

- API 失败；
- Rate Limit；
- Tool 调用异常；
- RAG 检索失败；
- 输出格式错误；
- 成本过高；
- 第三方服务不可用。

因此一个真正能够使用的 AI 产品，必须回答：当 AI 不稳定的时候，产品怎样继续工作?

所以第二层最后会逐渐形成这样一条生命周期：

```
Design
↓
Build
↓
Evaluate
↓
Diagnose
↓
Iterate
↓
Release
↓
Monitor
↓
再次迭代
```

AI 产品生命周期

真正的 AI 产品工作并不是“做完 Demo → 结束”，而应该是：

```
设计
↓
实现
↓
测试
↓
发现问题
↓
定位原因
↓
优化
↓
```

```
重新验证
↓
上线 / 发布
↓
持续监控
↓
进入下一轮
```

第二章：RAG 全链路

1. 什么是 RAG?

RAG：Retrieval-Augmented Generation，中文通常称为“检索增强生成”。

最简单的理解仍然是：

```
用户提出问题
↓
系统从外部资料中寻找相关内容
↓
把相关内容作为 Context 提供给大模型
↓
模型基于资料生成回答
```

> **不要让模型只依赖自身参数中已有的知识，而是在需要时先找到相关资料，再回答。**

2. 为什么需要 RAG?

大模型本身存在一些知识边界。

私有知识

- 公司内部制度；
- 企业知识库；
- 游戏内部策划文档；
- 未公开角色设定；
- 用户自己的文件。

这些内容可能根本没有进入模型训练数据。如果产品需要回答这些问题，就需要给模型提供外部知识。

最新知识

- 最新产品公告；
- 最新业务制度；
- 当前项目文档；
- 最新价格；
- 最新内部规范。

模型参数中的知识存在时间边界，而现实世界的信息一直在变化。如果产品要求回答动态变化的信息，就不能只依赖模型已有记忆。

专业知识

- 大量专业资料；
- 专业术语；
- 特定规则；
- 内部规范。

通用模型未必能够稳定掌握这些细节。这时通过外部资料为模型提供 Grounding，通常会比要求模型只依赖自身知识更加可控。

可追溯性

在很多场景下，用户不会满足于：“AI 说是这样的。”还希望知道：“AI 是根据什么资料得到这个答案的？”

例如：根据《员工手册》第 17 页……

因此，一个 RAG 产品往往还需要考虑：Grounding + Citation。

不仅回答问题，还要尽量让答案能够回到对应资料中进行核验。

3. RAG 的完整链路

```
Data
↓
```

```
Parse
↓
Clean
↓
Chunk
↓
Embedding
↓
Index
↓
Query
↓
Retrieve
↓
Rerank
↓
Context
↓
LLM
↓
Citation
↓
Evaluation
```

这条链路中的任何一个步骤都有可能让最终答案变差。

所以一个 RAG Bad Case 出现以后，不能第一反应就是：“模型不好。”真正的问题可能早在模型回答之前就已经发生了。

4. Data Source｜数据源

一个非常重要的认识是：RAG 的第一步并不是 Embedding。

- PDF；
- Word；
- Excel；
- 网页；
- 数据库；
- FAQ；
- API；
- 企业内部文档；
- 用户上传内容。

而是：先判断数据本身是否可靠。

如果原始数据已经错误、过期、互相冲突或者存在大量垃圾内容，那么后面即使使用再好的模型和检索方法，也很难得到可靠答案。

> **Garbage In, Garbage Out.**

5. Parse｜文档解析

真实文档往往不仅包含正文。

- 标题；
- 表格；
- 图片；
- 页眉；
- 页脚；
- 脚注；
- 多栏排版；
- 页码。

例如原始文件中写的是：“差旅住宿标准为 500 元 / 晚。”但如果解析工具处理失败，最后可能得到语义结构被破坏的碎片。

所以 RAG 最终回答错误，并不一定是模型导致的，也可能是知识在进入检索系统之前，就已经被 Parse 阶段破坏了。

```
原文：差旅住宿标准为 500 元 / 晚。


解析异常后可能变成：
差旅住宿 / 页码 12 / XX 公司 / 标准 / 500 / 元 / 晚
```

6. Clean｜数据清洗

解析完成以后，还需要清理无效内容。

- 重复页眉；

- 页脚；
- 广告；
- 无意义导航；
- OCR 噪声；
- 重复文档；
- 无效符号。

但是清洗不能简单理解成“删除得越多越好”。

因为标题、章节关系、表格字段等结构本身也可能是知识的一部分。

7. Chunking｜切块

Chunking 可以理解成：把较长文档拆成若干较小的信息块。

例如一份 100 页的员工手册。用户只问：“上海出差住宿标准是多少？”显然没有必要每一次都把完整 100 页发送给模型。

```
Chunk 1
Chunk 2
Chunk 3
……
```

当用户提问时，系统只寻找与问题相关的 Chunk。因此 Chunking 实际上会直接影响：系统以后到底能够检索到什么。

8. Chunk Size｜Chunk 应该多大?

Chunk Size 并不存在一个所有场景通用的固定答案，需要在不同目标之间取舍。

Chunk 太大

- 优势：上下文通常更加完整。
- 无关信息更多；
- Token 使用增加；
- 信息密度下降；
- 检索结果不够聚焦。

Chunk 太小

- 优势：每一个 Chunk 的主题更加集中。
- 上下文被切断；
- 指代关系丢失；
- 完整语义被破坏。

```
Chunk A:
上海地区员工出差时……


Chunk B:
住宿标准为 500 元 / 晚。
```

如果只召回 Chunk B，“500 元到底是谁的标准？”信息已经不完整。这就是一种 Context Fragmentation（上下文碎片化）。

9. Chunk Overlap

为了减少切块边界造成的信息丢失，可以让相邻 Chunk 保留一部分重复内容。

```
Chunk A: A B C D
Chunk B: C D E F


Overlap: C D
```

它的目标是减少重要语义刚好被切在两个 Chunk 中间的风险。

- 索引数据增加；
- 存储成本增加；
- 检索结果重复；
- Token 浪费。

所以 Overlap 也需要通过真实任务进行测试。

10. Structure-aware Chunking

> **不是所有文档都适合按照固定字符数机械切块。**

更加合理的方式可能是根据原始结构切分，例如按照：

- 标题；
- 章节；
- 段落；
- 表格；
- FAQ；
- Markdown Heading。

```
《员工差旅制度》


国内住宿
  上海
  北京
  广州
```

如果“国内住宿”本身就是一个完整逻辑单元，那么保留这种结构，通常会比“每 500 字强制切一次”更符合文档本身的语义。

11. Metadata

Metadata 可以理解成：描述文档或 Chunk 的附加信息。

```
{
  "department": "HR",
  "year": 2026,
  "document": "员工手册",
  "page": 13,
  "section": "差旅制度"
}
```

这些信息不是正文内容本身，但对于检索非常有价值。

例如用户问：“2026 年上海差旅标准是多少？”系统可以先过滤 year = 2026，然后再在这些资料中进行检索。

所以 Metadata 不只是“给文档打标签”，它还可以参与 Filter、Retrieval 和 Citation。

12. Embedding

Embedding 可以先继续理解成：把文本转换成一种可以进行语义比较的向量表示。

```
北京住宿报销标准
→ [0.12, -0.44, 0.81, ...]


北京出差酒店报销限额
```

虽然使用的文字不同，但语义比较接近。在理想情况下，它们对应的向量也会比较接近。这样系统就可以进行 Semantic Search（语义搜索）。

13. Keyword Search｜关键词检索

关键词搜索更加关注：文字本身有没有出现。

例如用户搜索 RTX5090，关键词检索往往很适合寻找：

- 专有名词；
- 编号；
- 产品型号；
- 文件编号；
- 精确字段。

因为这些内容本身就具有很强的字面特征。

14. Vector Search｜向量检索

向量搜索更加适合：表达方式不同，但意思相近。

> **“员工住酒店最多能报多少钱？” ↔ “住宿费用报销上限。”**

两句话没有使用完全相同的词，但语义相近。向量检索的价值，就是帮助系统发现这种语义关系。

15. Hybrid Search｜混合检索

Keyword Search 和 Vector Search 各有优势。

```
Keyword Search
+
Vector Search
=
Hybrid Search
```

例如明确型号“RTX5090”可能更适合关键词检索；而“当前最高性能的显卡型号是什么？”这样的表达，语义检索可能更有价值。

> **不要执着于“哪个方法更高级”，而是判断不同方法解决什么问题。**

16. Query

Query 就是用户用于检索的查询。但真实用户的输入并不总是完整。

```
第一轮：上海住宿标准是多少？
第二轮：那北京呢？
```

如果直接拿“那北京呢？”去检索知识库，信息量非常有限，因为这个问题依赖上一轮上下文。于是就会需要进一步处理 Query。

17. Query Rewrite

> **“那北京呢？” → “北京地区员工差旅住宿报销标准是多少？”**

Query Rewrite 的目标不是改变用户问题，而是把一个依赖上下文、表达不完整的问题，转换成更适合检索的 Query。

18. Query Expansion

有时还可以把一个 Query 扩展成多个相关表达。

```
AI 产品经理工资


→ AI PM 薪资
→ 人工智能产品经理待遇
→ AI Product Manager salary
```

这样做的目的是增加找到相关资料的机会。但过度扩展也可能引入不相关内容。

19. Retrieval

Retrieval 就是：根据 Query，从知识集合中找到相关 Chunk。

```
Chunk A：0.93
```

```
Chunk B: 0.90
Chunk C: 0.81
Chunk D: 0.72
Chunk E: 0.61
```

这些分数代表某种检索相关性排序。真正进入产品以后，还需要继续决定到底要把多少内容送给后续环节，这就涉及 Top-K。

20. Top-K

Top-K 可以理解成：只取排名最高的 K 条结果。

> **Top-3 = 只使用排名前三的 Chunk。**

K 太小可能漏掉真正需要的信息；K 太大又可能把大量无关内容一起送给模型。

所以 Top-K 实际上是在进行一种权衡：Recall 与噪声之间的 Trade-off。

21. Recall

Recall（召回率）可以粗略理解成：应该找到的资料中，系统到底找到了多少。

```
真正相关资料：10 条
系统找到：8 条
Recall = 8 / 10 = 80%
```

Recall 更关心的是：有没有漏。

22. Precision

Precision（精确率）可以理解成：系统找出来的资料里，有多少是真的相关。

```
系统检索：10 条
真正相关：7 条
Precision = 7 / 10 = 70%
```

Precision 更关心的是：找出来的东西准不准。

23. Recall 与 Precision 的取舍

```
检索更宽：Recall ↑，Precision 可能 ↓
```

```
检索更严格：Precision ↑，Recall 可能 ↓
```

因此产品设计需要根据场景进行权衡。真正重要的不是追求某个指标永远越高越好，而是当前产品场景到底更害怕“漏掉”，还是更害怕“找错”。

24. Reranking

初次 Retriever 往往需要优先保证速度。例如先快速找出 Top 20，然后再通过 Reranker，对候选结果进行更加精细的相关性判断。

```
Retriever
↓
Top 20

Reranker
↓
真正最相关的 Top 5
```

我目前会简单理解成：Retriever 强调“先找出来”，Reranker 进一步强调“重新排准”。

25. Context Construction

检索到资料以后，并不是把所有 Chunk 随便拼起来就结束了。还需要考虑：

- 顺序；
- 去重；
- Token；
- 来源；
- 文档结构；
- 相关性。

最终形成 Model Context。所以 Context Construction 本身也是 RAG 产品设计的一部分。

如果正确资料都已经召回，但最终 Context 拼接混乱，模型仍然可能回答错误。

26. Grounding

Grounding 可以理解成：让模型的回答尽量建立在给定资料之上。

```
只根据提供的资料回答。如果资料不足，请明确说明当前信息无法支持结论。
```

目标是减少模型在缺乏依据时自行补充内容。这并不能保证模型绝对不会出错，但能够帮助产品进一步明确：回答应该依据什么。

27. Citation

如果模型回答：“上海住宿标准为 500 元 / 晚。”一个更加完整的 RAG 产品可能还应该展示：

```
来源：《员工手册》
页码：P13
```

Citation 的价值包括：

- 用户验证；
- 提升可追溯性；
- 合规检查；
- Debug。

尤其对产品团队来说，Citation 还有一个很实际的用途：答案出了问题以后，我可以追溯模型到底用了哪份资料。

28. No Answer

这是我认为 RAG 产品非常重要、但很容易被忽略的一种能力。

如果知识库中没有答案，错误的处理方式是：模型自己编一个看起来合理的回答。

> **更加合理的设计应该允许系统说：当前知识库中没有足够信息支持这个回答。**

这可以被理解成 Abstention / Refusal / No-answer Design。

一个可靠的 AI 产品不仅需要学会什么时候回答，还要学会什么时候不要回答。某种程度上，能够明确说“我不知道”，本身就是系统可靠性的一部分。

第三章：RAG Bad Case 分析

进入第二层以后，我觉得 RAG 最需要改变的一个思维方式就是：如果系统回答错了，不能只说“RAG 效果不好”。

这句话几乎没有办法指导下一步优化。必须继续往下拆：到底是哪一个环节出了问题?

1. Data Error｜数据错误

最前面的原始资料本身就是错的。

例如企业制度已经更新，但知识库中仍然使用旧版本。这种情况下，模型即使完全按照资料回答，也可能得到错误结果。

解决方向首先应该是：修数据，而不是先调 Prompt。

2. Parsing Error｜解析错误

原始文件没有问题，但是解析以后内容被破坏。

例如：表格错位、标题丢失、内容顺序混乱、OCR 识别错误、页眉混入正文。

解决方向可能是：检查 Parser 或文档格式。

3. Chunking Error｜切块错误

正确答案本来存在于文档中，但因为 Chunk Strategy 不合理，语义被拆开了。

例如“上海员工出差时……”和“住宿标准为 500 元。”被切到两个彼此缺乏语义的块中。

解决方向：调整 Chunking Strategy。

4. Query Error｜查询问题

用户的问题本身过于模糊。

例如：“那北京呢？”如果直接检索，很可能无法找到正确内容。

解决方向可能包括 Query Rewrite、Clarification，以及结合对话上下文重新构造 Query。

5. Retrieval Miss｜正确资料没有被召回

知识库里明明存在正确答案，但是检索阶段根本没有找到。

可能需要检查 Embedding、Keyword Search、Hybrid Search、Query Rewrite、Top-K、Metadata Filter。

这时换模型并不能直接解决问题，因为模型从一开始就没有拿到正确资料。

6. Ranking Error｜排序错误

正确资料其实已经被召回了，但是它排得非常后。

例如系统召回 Top 20，正确答案排在第 18 位，但产品最终只使用 Top 5。

这种情况需要检查 Ranking / Reranking。

7. Context Error｜上下文构建错误

正确资料已经找到了，但最终送给模型的 Context 有问题。

例如内容顺序混乱、重复片段太多、关键信息被截断、不相关资料占据大量 Context、多份资料互相冲突。

这时 Retrieval 可能没有问题，真正的问题发生在 Context Construction。

8. Generation Error｜生成错误

正确资料找到了，Context 也没有明显问题，但模型最终还是答错。

这才真正进入 Generation Error。

可能需要继续检查 Prompt、模型能力、Structured Output、Grounding。

9. Citation Error｜引用错误

答案本身是正确的，但是引用来源错了。

例如答案来自文档 A，系统却展示文档 B。

因此 Citation 也需要单独评测，不能因为“答案正确”就认为整个 RAG 系统已经完全正确。

RAG Bad Case 标准分析表

我目前可以先用一个最基础的分析表来判断问题发生在哪里：

Case	Retrieval	Context	Answer	可能根因
A	错	—	错	Retrieval / Recall
B	对	错	错	Context
C	对	对	错	Generation
D	对	对	对	正常

> **不要一看到最终答案错误，就立刻去修改 Prompt。先往前检查。**

如果以后被问：“RAG 回答错误怎么办？”

以前可能很容易回答：“调 Prompt。”或者：“换一个更强的模型。”

但整理完这一章以后，我觉得第一句话应该先变成：

> **我会先判断这是 Retrieval 问题，还是 Generation 问题。**

然后继续往下拆：

```
Data
↓
Parsing
↓
Chunking
↓
Query
↓
Retrieval
↓
Ranking
↓
Context
↓
Generation
↓
Citation
```

只有把问题定位到真正的环节以后，才能选择对应的优化方案。

写在这一部分学习之后

第一层学习 RAG 时，我对它最简单的理解是：先找资料，再让模型根据资料回答。这个理解没有错。

但进入第二层以后，我开始发现：真正决定 RAG 产品效果的，并不是“有没有接向量数据库”。

而是一整条链路：

```
数据可靠吗?
↓
解析正确吗?
↓
Chunk 合理吗?
↓
Query 完整吗?
↓
Retriever 找到了吗?
↓
正确资料排得够前吗?
↓
Context 组织合理吗?
↓
模型有没有 Grounding?
↓
Citation 对吗?
↓
不知道的时候会不会拒答?
```

因此，我现在更愿意把 RAG 理解成：一个由数据、检索、上下文构建、生成和评测共同组成的系统。

它不是一个单独的“AI 功能”，更不是“上传 PDF → Embedding → 向量库 → 接大模型”就算完成。

而第二层学习真正开始训练的，也并不是“搭系统”本身，而是一种新的思考方式：系统出了问题以后，我能不能先拆链路，再找根因。

这和第一层最大的区别可能就在这里。第一层更像是在回答：“AI 产品由什么组成？”第二层则开始回答：“这些组成部分为什么会失败，以及我应该怎样证明自己真的把它修好了？”

接下来再继续学习 Evaluation、Agent、实验和工程化时，我也希望沿用同一个原则：

> **不要只优化最终输出。先找到问题真正发生在哪一层。**

只有这样，AI 产品的迭代才不会一直停留在“感觉这个版本好像更好了”，而是逐渐变成一套可以定位、可以验证、可以比较、也可以持续迭代的产品方法。

笔记说明

本文为个人学习记录，内容基于当前阶段对 AI 产品方法与 RAG 全链路的理解整理。示例主要用于帮助理解概念，不代表真实线上业务数据。后续随着 Evaluation、Agent、实验与工程化学习继续深入，其中部分认识也会继续补充和修正。