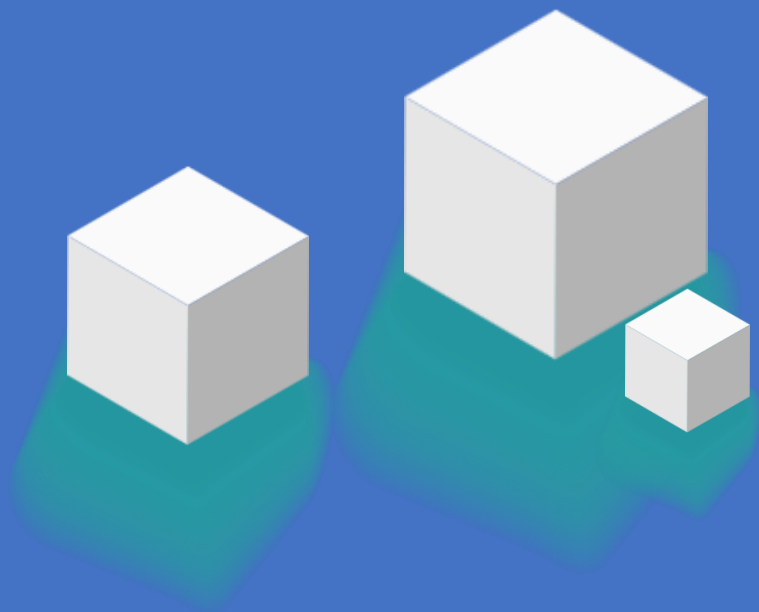


项目实战：企业知识库



>> 今天的学习目标

项目实战：企业知识库

- CASE：RAG冠军方案（多路由+动态知识库）
- RAG比赛任务说明
- 基础RAG系统流程
- 解析模块、Docling优化、表格序列化
- 内容提取（ingestion）
- 检索（Retrieval）
- LLM 重排序 (LLM reranking)
- 父页面检索、整合后的检索器
- 增强 (Augmentation)
- 生成 (Generation)
- 思维链、结构化输出、思维链+结构化输出
- 指令细化 (Instruction Refinement)
- 提示词创建、Prompt.py 实现
- RAG系统调参
- CASE：搭建自己的RAG系统

CASE：RAG冠军方案（多路由+动态知识库）

TO DO：搭建企业知识库RAG系统

- 参考项目：RAG-Challenge-2
<https://github.com/IlyaRice/RAG-Challenge-2>
- 实现知识库RAG系统，在企业RAG挑战赛中取得优异成绩。RAG-Challenge系统在比赛中赢得了所有类别的奖项，展示了在处理公司年度报告问题回答方面的强大能力。

系统特点：

- **定制PDF解析**：使用Docling进行PDF文档解析。
- **向量搜索与父文档检索**：通过向量搜索实现文档检索。
- **LLM重排序**：利用LLM对检索结果进行重排序，提高上下文相关性。
- **结构化输出提示**：采用链式推理（chain-of-thought reasoning）进行结构化输出。
- **查询路由**：支持多公司比较的查询路由功能。

这是竞赛代码，虽然功能完善但存在临时解决方案，比如IBM Watson集成部分是竞赛特定的，无法正常工作。

CASE：RAG冠军方案（多路由+动态知识库）

企业RAG比赛：<https://github.com/trustbit/enterprise-rag-challenge/tree/main>

基于公司年度报告构建一个问答系统，简单来说，比赛当天的流程如下：

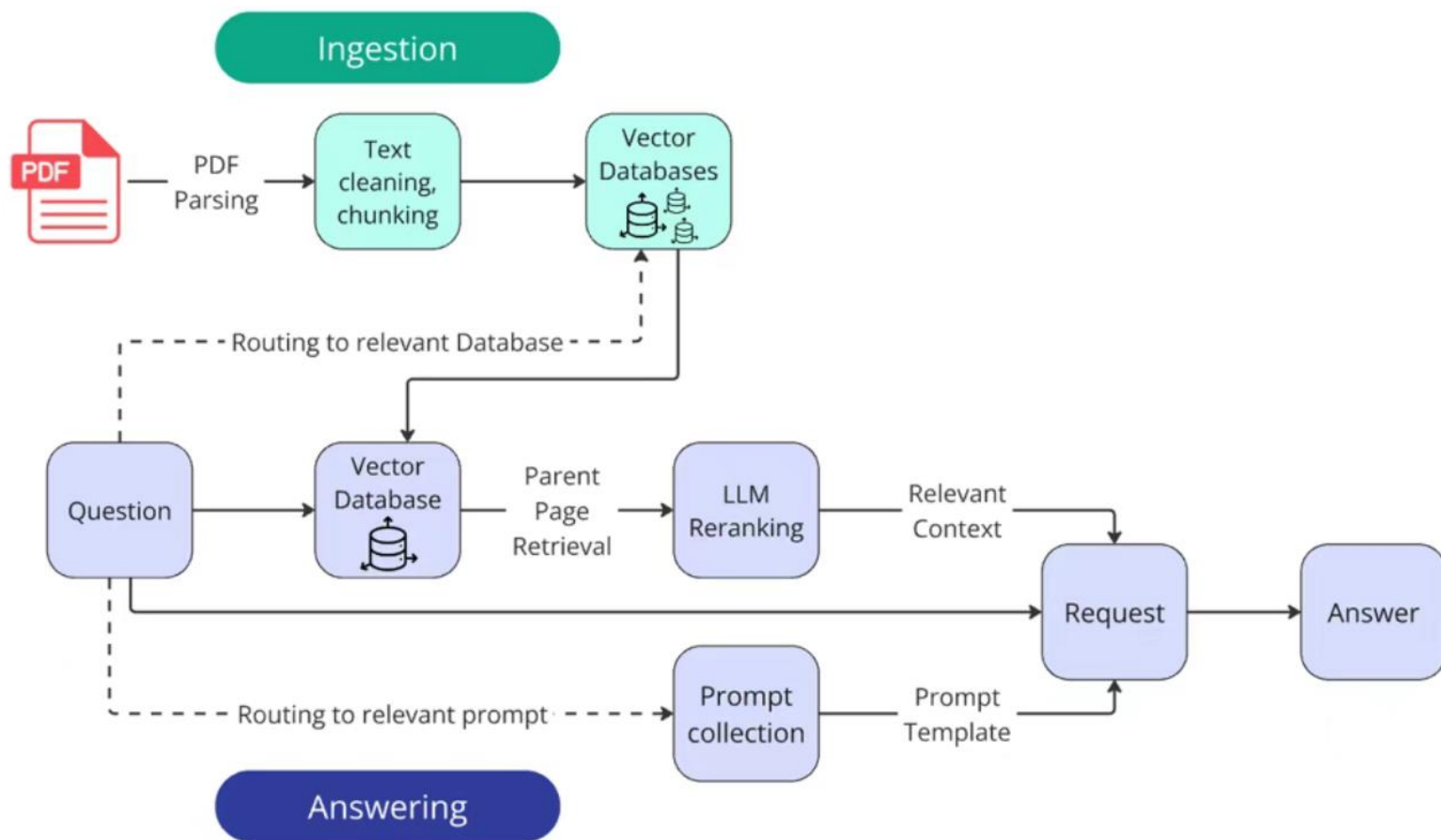
- 我们会收到来自随机挑选公司的 100 份年度报告，并需要在 2.5 小时内解析这些报告并构建一个数据库。这些报告是 PDF 格式，每份最长可达 1000 页。
- 然后，系统会生成 100 个随机问题（基于预设模板），我们的RAG系统必须尽快回答这些问题。

所有问题都有确定的答案，例如：

- 是/否；
- 公司名称（某些情况下是多个公司名称）；
- 领导职位头衔、推出的产品名称；
- 数值指标：营收、商店数量等。

每个答案都必须注明引用的页码，确保系统是从原文中得出答案，而不是输出虚假信息（hallucinate）。

CASE: RAG冠军方案（多路由+动态知识库）



获胜方案在native RAG的基础上，加入了两个路由器（router）和 LLM 重排序模块（LLM reranking）

CASE：RAG冠军方案（多路由+动态知识库）

Team / Experiment		Time	R/G	Score
1. ▶ Ilia Ris	🥇	49 min	83/81	123.7
2. ▶ Emil Shagiev	🥈	55 min	86/78	121.6
3. ▶ Dmitry Buykin	🥉	8 hours	81/76	117.5
4. ▶ Sergey Nikonov	🥈	30 hours	85/73	116.4
5. ▶ ScrapeNinja.net	🥈	23 hours	82/71	112.5
6. ▶ xsl777	🥈	16 hours	79/71	110.9
7. ▶ nikolay_sheyko(grably.tech)	🥈	25 hours	81/69	110.4
8. ▶ Felix-TAT	🥈	7 days	80/69	109.4
9. ▶ A.Rasskazov/V.Kalesnikau		30 hours	84/67	109.3
10. ▶ Dany the creator	🥈	3 hours	82/67	108.4



Ilya Rice

- **Best experiment:** Dense Retrieval; Router; LLM reranking; o3-mini
- **Signature:** f1d79f
- **Summary:** Dense retrieval combined with LLM reranking and SO CoT.
- **Article:** [How I Won the Enterprise RAG Challenge](#)
- **Source code:** [Github](#)

Models used:

- o3-mini-2025-01-31

Architecture

Ilya Rice solved the problem by making it easy to run numerous experiments before the competition has even started. He created an evaluation pipeline that let him quickly evaluate different architectural solutions. The best solution was also among the fastest ones.

冠军方案在 retrieval和generation，以及score上都取得了不错的成绩，采用 o3-mini 进行推理

<https://abdullin.com/erc/>

基础RAG系统流程

Thinking: 基础 RAG系统的开发流程是什么?

- 解析 (Parsing): 为知识库准备数据, 包括收集文档、将其转换为文本格式, 并清理无关的噪点信息。
- 内容提取 (Ingestion): 创建并载入知识库。
- 检索 (Retrieval): 构建一个工具, 根据用户查询查找并返回相关数据, 通常在向量数据库中进行语义搜索。
- 回答 (Answering): 将检索到的数据 + 用户的提示词 (prompt), 发送给 LLM, 返回最终答案。

解析模块 (Parsing)

Thinking：解析模块的挑战？

需要将 PDF 文档转换为纯文本。PDF 解析是一项远非易事的工作，其中充满了无数细微的难题：

- 保留表格结构；
- 保留关键的格式元素（例如标题和项目符号列表）；
- 识别多栏文本；
- 处理图表、图片、公式、页眉/页脚等等。

CAIXA GERAL DE DEPÓSITOS, S.A.

CONSOLIDATED STATEMENTS OF CHANGES IN SHAREHOLDERS' EQUITY

(EUR Thousand)

	Share capital	Other equity instruments	Reserves	Other reserves and retained earnings	Net income for the exercise	Subtotal	Non-controlling interests	Total
Balance at December 31, 2020	3,844,144	500,000	267,266	3,362,753	491,592	8,468,754	204,076	8,700,830
Other entries directly recorded in equity	-	-	-	-	-	-	-	-
Changes in the valuation reserves	-	-	(12,187)	-	-	(12,187)	(157)	(13,344)
Appropriation of comprehensive income from associate and jointly controlled entities	-	-	-	(30,893)	-	(30,893)	-	(30,893)
Employee benefits - actuarial gains and losses	-	-	-	345,437	-	345,437	337	345,774
Foreign currency differences	-	-	-	102,230	-	102,230	37,786	143,026
Changes in the fair value of equity instruments	-	-	-	42	-	42	-	42
Net income for the exercise	-	-	-	-	583,381	583,381	50,334	633,695
Other	-	-	-	-	(412)	(412)	(38,224)	(38,696)
Total gains and losses for the exercise recognised in equity	-	-	(12,187)	414,404	583,381	965,578	50,038	1,040,814
Appropriation of net income for 2020	-	-	-	-	-	-	-	-
Transfer to reserves and retained earnings	-	-	-	407,953	(407,953)	-	-	-
Dividends paid	-	-	-	(300,000)	(83,639)	(383,639)	-	(383,639)
Dividends and other expenses related with the issue of other equity instruments	-	-	-	(53,750)	-	(53,750)	-	(53,750)
Equity transactions with non-controlling interests	-	-	-	(787)	-	(787)	-	(787)
Dividends paid to non-controlling interests	-	-	-	-	-	-	(16,305)	(16,305)
Balance at December 31, 2021	3,844,144	500,000	255,079	3,866,572	583,381	9,049,156	237,807	9,286,963
Other entries directly recorded in equity	-	-	-	-	-	-	-	-
Changes in the valuation reserves	-	-	(125,416)	-	-	(125,416)	(24)	(125,440)
Appropriation of comprehensive income from associate and jointly controlled entities	-	-	-	(84,373)	-	(84,373)	-	(84,373)
Employee benefits - actuarial gains and losses	-	-	-	310,333	-	310,333	(430)	309,906
Foreign currency differences	-	-	-	71,697	-	71,697	16,975	88,672
Changes in the fair value of equity instruments	-	-	-	9,432	-	9,432	-	9,432
Net income for the exercise	-	-	-	-	842,786	842,786	80,208	923,075
Other	-	-	-	227	-	227	923	1,150
Total gains and losses for the exercise recognised in equity	-	-	(125,416)	307,316	842,786	1,024,686	97,736	1,122,422
Appropriation of net income for 2021	-	-	-	-	-	-	-	-
Transfer to reserves and retained earnings	-	-	-	342,290	(342,290)	-	-	-
Dividends paid	-	-	-	(137,160)	(241,071)	(378,231)	-	(378,231)
Dividends and other expenses related with the issue of other equity instruments	-	-	-	(13,139)	-	(13,139)	-	(13,139)
Reimbursement of other equity instruments (AT1)	-	(500,000)	-	-	-	(500,000)	-	(500,000)
Equity transactions with non-controlling interests	-	-	-	26	-	26	2,267	2,293
Dividends paid to non-controlling interests	-	-	-	-	-	-	(37,647)	(37,647)
Balance at December 31, 2022	3,844,144	-	129,663	4,365,905	842,786	9,182,497	300,163	9,482,661

大型表格有时会旋转 90 度，导致解析器产生乱码

解析模块 (Parsing)

Thinking：如何选择解析器？

作者尝试了20多种解析器：

- 小众解析器；
- 知名的解析器；
- 基于前沿机器学习算法训练的解析器；
- 支持 API 访问的商业解析器。

最终作者选择了 Docling作为PDF解析器

没有任何解析器可以处理所有细微之处，并在不丢失部分重要信息的情况下，将 PDF 内容完全还原为文本。

CAIXA GERAL DE DEPÓSITOS, S.A.									
CONSOLIDATED STATEMENTS OF CHANGES IN SHAREHOLDERS' EQUITY									
(EUR Thousand)									
	Share capital	Other equity instruments	Reserves	Other reserves and retained earnings	Net income for the exercise	Subtotal	Non-controlling interests	Total	
Balance at December 31, 2020	3,844,144	500,000	267,266	3,392,753	491,592	8,498,754	204,076	8,700,830	
Other entries directly recorded in equity	-	-	(12,187)	-	-	(12,187)	(157)	(13,344)	
Charges in the valuation reserves	-	-	-	(30,890)	-	(30,890)	-	(30,890)	
Appropriation of comprehensive income from associate and jointly controlled entities	-	-	-	345,437	-	345,437	337	345,774	
Employee benefits - actual gains and losses	-	-	-	102,230	-	102,230	37,786	143,026	
Foreign currency differences	-	-	-	42	-	42	-	42	
Changes in the fair value of equity instruments	-	-	-	-	583,381	583,381	50,334	633,695	
Net income for the exercise	-	-	-	-	(412)	(412)	(38,221)	(38,696)	
Other	-	-	(12,187)	419,404	583,381	969,578	50,008	1,040,814	
Total gains and losses for the exercise recognised in equity	-	-	-	-	-	-	-	-	
Appropriation of net income for 2020	-	-	-	407,953	(407,953)	-	-	-	
Transfer to reserves and retained earnings	-	-	-	(300,000)	(83,639)	(383,639)	-	(383,639)	
Dividends paid	-	-	-	(53,750)	-	(53,750)	-	(53,750)	
Dividends and other expenses related with the issue of other equity instruments	-	-	-	(787)	-	(787)	-	(787)	
Equity transactions with non-controlling interests	-	-	-	-	-	-	(16,305)	(16,305)	
Dividends paid to non-controlling interests	-	-	-	-	-	-	237,807	237,807	
Balance at December 31, 2021	3,844,144	500,000	255,079	3,886,572	583,381	9,049,156	-	9,298,963	
Other entries directly recorded in equity	-	-	(125,416)	-	-	(125,416)	(24)	(125,440)	
Charges in the valuation reserves	-	-	-	(84,373)	-	(84,373)	-	(84,373)	
Appropriation of comprehensive income from associate and jointly controlled entities	-	-	-	310,333	-	310,333	(430)	309,906	
Employee benefits - actual gains and losses	-	-	-	71,697	-	71,697	16,975	88,672	
Foreign currency differences	-	-	-	9,432	-	9,432	-	9,432	
Changes in the fair value of equity instruments	-	-	-	-	842,786	842,786	80,280	923,076	
Net income for the exercise	-	-	-	-	227	227	923	1,150	
Other	-	-	(125,416)	307,316	842,786	1,024,686	97,736	1,122,422	
Total gains and losses for the exercise recognised in equity	-	-	-	-	-	-	-	-	
Appropriation of net income for 2021	-	-	-	342,290	(342,290)	-	-	-	
Transfer to reserves and retained earnings	-	-	-	(137,160)	(241,071)	(378,231)	-	(378,231)	
Dividends paid	-	-	-	-	-	-	-	-	
Dividends and other expenses related with the issue of other equity instruments	-	-	-	(13,139)	-	(13,139)	-	(13,139)	
Reimbursement of other equity instruments (AT1)	-	(500,000)	-	-	-	(500,000)	-	(500,000)	
Equity transactions with non-controlling interests	-	-	-	26	-	26	2,267	2,293	
Dividends paid to non-controlling interests	-	-	-	-	-	-	(37,647)	(37,647)	
Balance at December 31, 2022	3,844,144	-	129,663	4,365,905	842,786	9,182,497	300,163	9,482,661	

Docling优化

尽管 Docling 的结果非常优秀，但它缺乏一些基本能力。作者研究了docling源代码，并重写了几个方法以满足需求：

- 在解析后得到了一个包含所有必要元数据的 JSON 文件。
- 利用这个 JSON，可以构建了 Markdown 文档，表格结构从 PDF 转换为 Markdown 甚至 HTML。
- 有时部分文本从 PDF 中解析出来时会出错，包含特定的语法，降低了可读性和意义。我们使用了十几个正则表达式来处理这个问题。

Operating sustainably⁴

CARBON EMISSIONS⁵ UP
(tCO₂-e/ct)

0.139 +10%

```
["Operating sustainably 4",  
"/C.cap /A.cap/R.cap/B.cap/O.cap/N.cap /E.cap/M.cap/I.cap/S.cap/S.  
cap/I.cap/O.cap/N.cap/S.cap /five.pl /U.cap/P.cap",  
"(t/C.cap/O.cap/two.pl-e/ct)",  
"/zero.pl.tnum./one.tnum.pl /three.pl.tnum/nine.pl.tnum /space.  
tnum /plus.tnum/one.tnum.pl /zero.pl.tnum /percent.pl"]
```

解析不佳的文本示例

使用Docling，可以将报告解析为 JSON 格式

表格序列化

Thinking： 是否需要进行表格序列化？

在大型表格中， 度量名称（横向表头） 通常离纵向表头太远， 削弱了语义连贯性。

	GEAR UP! MOL		RISE 2013		STEER FOR 2020		ROLLING PLAN				
	2012/3	2013/3	2014/3	2015/3	2016/3	2017/3	2018/3	2019/3	2020/3	2021/3	2022/3
For the year											
Shipping and other revenues	¥1,435,220	¥1,509,194	¥1,729,452	¥1,817,069	¥1,712,222	¥1,504,373	¥1,652,393	¥1,234,077	¥1,155,404	¥ 991,426	¥1,269,310
Shipping and other expenses	1,368,794	1,432,014	1,587,902	1,683,795	1,594,568	1,388,264	1,513,736	1,094,915	1,035,771	911,055	1,117,405
Selling, general and administrative expenses	90,885	92,946	100,458	116,024	115,330	113,551	115,972	101,442	95,852	85,674	96,899
Operating profit (loss)	(24,459)	(15,766)	41,092	17,249	2,323	2,558	22,684	37,718	23,779	(5,303)	55,005
Ordinary profit (loss)	(24,320)	(28,568)	54,985	51,330	36,267	25,426	31,473	38,574	55,090	133,604	721,779
Income (loss) before income taxes and non-controlling interests	(33,514)	(137,938)	71,710	58,332	(154,385)	23,303	(28,709)	46,778	47,130	100,313	732,993
Profit (loss) attributable to owners of parent	(26,009)	(178,846)	57,393	42,356	(170,447)	5,257	(47,380)	26,875	32,623	90,052	708,819
Free cash flow [(a) + (b)]	(129,298)	(25,285)	(25,615)	(66,656)	182,508	(56,318)	(2,471)	(143,093)	(6,527)	44,238	200,187
Cash flows from operating activities (a)	5,014	78,955	94,255	92,494	209,189	17,623	98,380	55,248	100,723	98,898	307,637
Cash flows from investing activities (b)	(134,312)	(104,240)	(119,870)	(159,150)	(26,681)	(73,941)	(100,851)	(198,341)	(107,250)	(54,640)	(107,450)
Depreciation and amortization	85,624	94,685	83,983	87,803	92,771	87,190	86,629	90,138	87,765	85,798	86,399
At year-end											
Total assets	¥1,946,161	¥2,164,611	¥2,364,695	¥2,624,049	¥2,219,587	¥2,217,528	¥2,225,096	¥2,134,477	¥2,098,717	¥2,095,559	¥2,686,701
Total tangible fixed assets	1,293,802	1,303,967	1,379,244	1,498,028	1,376,431	1,323,665	1,290,929	1,193,910	1,201,698	1,099,458	1,111,152
Total investments and other assets	249,228	323,468	422,426	577,157	353,197	381,097	425,300	524,411	533,320	637,736	1,187,472
Interest-bearing debt	869,619	1,046,865	1,094,081	1,183,401	1,044,980	1,122,400	1,118,089	1,105,873	1,096,685	1,026,994	1,000,697
Net assets	717,909	619,492	783,549	892,435	646,924	683,621	628,044	651,407	641,235	699,150	1,334,866
Shareholders' equity	637,422	535,422	679,140	782,556	540,951	571,983	511,242	525,064	513,335	577,782	1,274,570
Amounts per share of common stock**											
Profit (loss) attributable to owners of parent (Yen)	¥ (72.53)	¥ (498.57)	¥ 159.97	¥ 118.07	¥ (475.00)	¥ 14.65	¥ (132.05)	¥ 74.91	¥ 90.93	¥ 250.99	¥ 1,970.16
Net assets (Yen)	1,777.57	1,492.53	1,893.00	2,180.87	1,507.60	1,594.08	1,424.94	1,463.46	1,430.77	1,610.04	3,532.32
Cash dividends applicable to the year (Yen)	16.67	0.00	16.67	23.33	16.67	6.67	6.67	15.00	21.67	50.00	400.00
Management indicators											
Gearing ratio (Times)	1.36	1.96	1.61	1.51	1.93	1.96	2.19	2.11	2.14	1.78	0.78
Net gearing ratio (Times)	1.23	1.58	1.35	1.35	1.64	1.44	1.82	1.88	1.94	1.63	0.71
Equity ratio (%)	32.8	24.7	28.7	29.8	24.4	25.8	23.0	24.6	24.5	27.6	47.4
ROA (%)	(1.3)	(1.4)	2.4	2.1	1.5	1.1	1.4	1.8	2.6	6.4	30.2
ROE (%)	(4.0)	(30.5)	9.5	5.8	(25.8)	0.9	(8.7)	5.2	6.3	16.5	76.5
Dividend payout ratio (%)	—	—	10.4	19.8	—	45.5	—	20.0	23.8	19.9	20.3
CO ₂ emissions of MOL Group** fleet (Thousand tons)	19,660	18,876	17,810	18,803	18,674	18,203	17,774	12,199	11,137	9,831	10,112
Number of MOL Group** employees	9,431	9,465	10,289	10,508	10,500	10,794	10,828	8,941	8,931	8,571	8,547

有 1500 个不相关的 token将纵向和横向表头隔开

表格序列化

Thinking: 大型表格理解的挑战?

理论上, 大型表格会降低在向量搜索中 chunk 的相关性 (同时, 也存在表格不能完全装进一个 chunk 的情况)。

此外, LLM 在处理大型表格时也很难将度量名称与表头对应起来, 可能会返回错误的值。

表格序列化 (Serialization of tables) 成为了解决方案。

关于这个主题的研究较少 => 需要独立探索, 我们可以搜索 Row-wise Serialization (行式序列化)、Attribute-Value Pairing (属性-值对匹配), 参考阅读 <https://arxiv.org/pdf/2402.17944>

结论: 作者在测试不同配置时, 惊讶地发现我们对它寄予厚望的表格序列化功能不仅没有改进系统, 反而略微降低了其有效性 => 所以实际使用上, 并没有采用 表格序列化的方法

表格序列化

Thinking: 让LLM对表格数据进行理解, 是采用Markdown格式, 还是html格式?

我们以Markdown 格式向 LLM 输入表格, 但后来改用 HTML 格式。

语言模型对 HTML 的理解程度要高得多, 而且 HTML 可以描述包含合并单元格、子标题和其他复杂结构的表格。

序列化代码见: `tables_serialization.py`

内容提取 (ingestion)：分块

报告已从 PDF 转换为 => Markdown 文本 => 接下来，需要用它们来创建数据库。

根据比赛规则，必须指明包含相关信息的页码。企业系统也采用同样的方法：引用允许验证模型的答案是否是虚假信息 (hallucinated)。这不仅使系统对用户更透明，也简化了开发过程中的调试。

Thinking: 如何对文档进行切分？使用一页作为一个chunk，还是300个token作为一个chunk？

最简单的选择是将文档的整页作为一个块，因为页面很少超过几千个 token。

但让我们再次思考查询和文档文本块之间的语义连贯性。通常情况下，能回答问题的信息片段不超过十个句子。因此，从逻辑上讲，一个包含目标语句的小段落将比同样语句稀释在一整页相关性较弱的文本中获得更高的相似度得分。

=> 将每页的文本分割成 300 个 token (约 15 个句子) 的块。

每个块都存储其 ID 及其在元数据 (metadata) 中的父页面编号。

内容提取 (ingestion)：向量化

Thinking: 如果有100家公司的文档, 是创建一个Faiss数据库, 还是100个Faiss数据库?

100 个数据库, 其中 1 个数据库 = 1 个文档。

为什么要 将所有公司的信息混合在一起, 之后再试图将一个公司的收入与另一个公司分开呢?

=> 基于公司为单位创建的 Faiss数据库, 结构更清晰

=> 想看哪个公司的情况, 就直接从该faiss库中进行检索

Faiss数据库使用IndexFlatIP方法创建。

Flat 索引的优点是所有向量都“原样”存储, 没有压缩或量化。搜索使用暴力搜索, 精度更高。

缺点是这种搜索计算和内存消耗更高。

为了将块和查询嵌入到向量表示中, 作者使用了text-embedding-3-large (这个根据需求, 进行替换)

检索 (Retrieval)

RAG 系统中的R (检索) 是一个通用的搜索系统, 它接收查询作为输入, 返回回答所需的相关文本。

在基础实现中, 它只是对向量数据库发起查询, 提取 Top N 个结果。

Thinking: Retrieval是 RAG 系统中的关键部分, 是否需要采用混合搜索?

如果 LLM 在查询上下文中没有接收到必要信息, 它就无法提供正确答案 (无论我们如何精心设置提示词)

垃圾进 → 垃圾出 (Garbage in → Garbage out) 。

常用的策略是: 采用混合搜索

=> vDB + BM25混合搜索 (Hybrid search), 结合了基于向量语义和传统关键词的搜索 (BM25 算法) 。

理论上, 它通过不仅考虑文本语义, 还考虑精确关键词匹配来提高检索准确性。

=> 将两种方法的搜索结果合并, 再进行重排序。

混合搜索理论上不错, 但在它的基础实现中, 它通常会降低检索质量而不是提高。

重排序 (Reranking)

Jina Reranker 是一款专为提升信息检索性能设计的神经网络重排序模型，尤其适用于RAG：

- **细粒度语义分析**：采用交叉编码器架构，对查询和文档进行联合编码，捕捉令牌级别的交互细节，弥补传统向量检索（如余弦相似度）在语义理解上的不足。
- **多阶段检索优化**：在向量检索初步召回候选文档后，对结果进行二次重排，提升Top-K结果的准确性。例如，在LlamaIndex测试中，命中率提升7.9%，均值倒排率（MRR）提升33.7%。
- **多语言支持**：支持100多种语言，在MKQA基准测试中表现优于同类模型（如bge-reranker-v2-m3）²⁴。
- **Agentic RAG增强**：集成函数调用和Text-to-SQL能力，可识别结构化数据查询意图（如MySQL/MongoDB），并为API调用生成相关性评分，扩展了RAG的应用场景。
- **超快推理速度**：通过Flash Attention 2技术优化，v2版本比前代快6倍，比竞品bge-reranker-v2-m3快15倍，参数量仅278M，兼顾效率与性能

Jina Reranker和bge-m3都是常用的Reranking模型，但在比赛中作者使用功了LLM重排序

LLM 重排序 (LLM reranking)

Thinking: 如何实现LLM重排序?

```
system_prompt_rerank_single_block = """  
你是一个RAG检索重排专家。  
你将收到一个查询和一个检索到的文本块，请根据其与查询的相关性进行评分。
```

评分说明:

1. 推理: 分析文本块与查询的关系, 简要说明理由。

2. 相关性分数 (0-1, 步长0.1) :

0 = 完全无关

0.1 = 极弱相关

0.2 = 很弱相关

0.3 = 略有相关

0.4 = 部分相关

0.5 = 一般相关

0.6 = 较为相关

0.7 = 相关

0.8 = 很相关

0.9 = 高度相关

1 = 完全匹配

3. 只基于内容客观评价, 不做假设。

LLM 查询结果进行格式化输出, 包含两个字段:

reasoning (让模型解释其判断) 和

relevance_score, 可以直接从 JSON 中提取。

修正后的相关性得分, 使用加权平均计算:

vector_weight = 0.3, llm_weight = 0.7

理论上, 我们可以直接跳过向量搜索, 将每一页都直接传递给 LLM。然而, 使用embedding进行更便宜、更快速的过滤仍然是必要的。

对于一份 1000 页的文档, 仅仅回答一个问题就可能花费约 25 美分——太昂贵了。

父页面检索 (Parent Page Retrieval)

Thinking: 我们之前将文本分割成小块chunk, 这里是否还需要引入其父节点?

虽然回答问题的核心信息通常集中在某个小块中 (这也是分块能提升检索效果的原因), 但同一页面的其他部分可能仍包含次要却重要的细节。

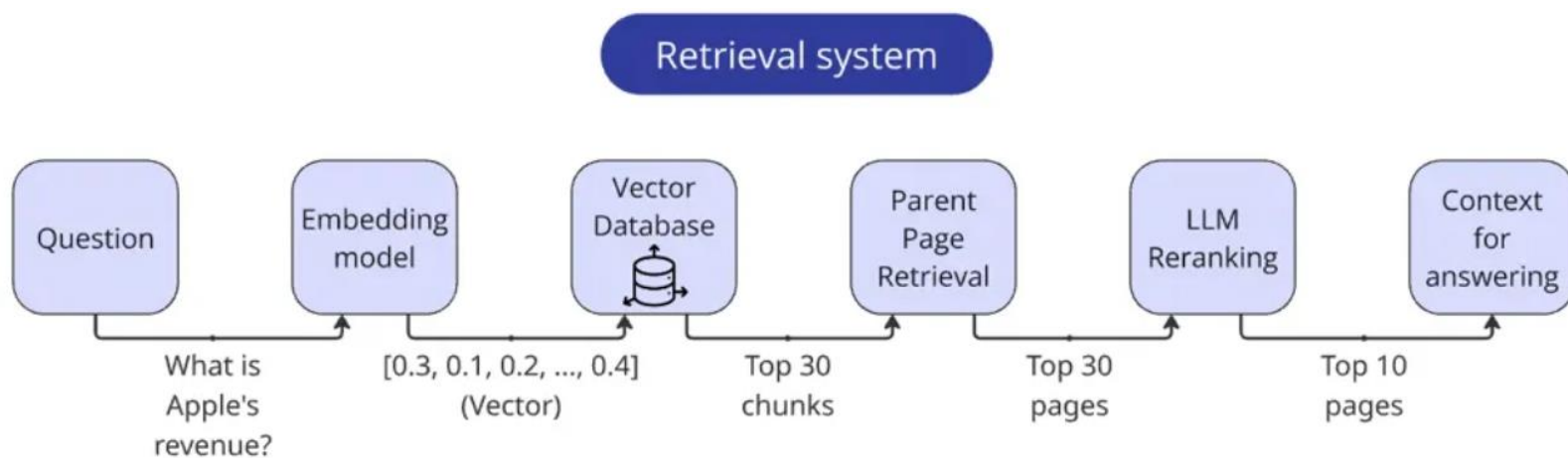
因此, 在实际应用中, 我们会先检索出 Top N 个最相关的文本块

=> 这些块仅作为“指针”来定位对应的完整页面。

=> 随后, 我们会将整个页面的内容纳入上下文 (context) 中进行分析。

这就是为什么我们在每个块的元数据中记录其所属的页面编号——以便快速回溯原始内容。

整合后的检索器 (Assembled Retriever)



Assembled Retriever diagram

最终检索器的步骤：

- 1) 对查询进行向量化。
- 2) 根据查询向量找到 Top 30 个相关块。
- 3) 通过块的元数据提取对应的页面（记得去重）。
- 4) 通过 LLM 重排序器处理这些页面。
- 5) 调整页面的相关性得分。
- 6) 返回得分最高的 Top 10 页，在每一页前面加上页码，并将它们合并成一个字符串。

增强 (Augmentation)

目前为止，向量数据库已建立，RAG 系统中的“R”（检索）已完成，我们现在进入“A”（增强）部分。

这部分相当直接，主要是 f-string 字符串拼接操作。

Thinking: 如何更有效的管理提示词存储的方式？

在项目中，尝试了不同方法后，最终确定：将提示词存储在一个专门的prompts.py文件中，并将提示词分割成逻辑块：

- 核心系统指令；
- 定义 LLM 返回响应格式的 Pydantic schema；
- 用于创建单次示例（one-shot）/少次示例（few-shot）提示词的问答对示例；
- 用于插入上下文和查询的模板。

增强 (Augmentation)

1. 核心系统指令

告诉大模型“你是谁、你要做什么、你要遵循哪些规则”的主提示词。

```
class AnswerWithRAGContextSharedPrompt:
```

```
    instruction = """
```

```
    你是一个RAG（检索增强生成）问答系统。
```

```
    你的任务是仅基于公司年报中RAG检索到的相关页面内容，回答给定问题。
```

```
    ...
```

```
    """
```

增强 (Augmentation)

2. 定义 LLM 返回响应格式的 Pydantic schema

用 Pydantic 的 BaseModel 定义 LLM 输出的 JSON 格式，强制 LLM 输出结构化内容，便于后续解析和校验。

```
class ComparativeAnswerPrompt:
    class AnswerSchema(BaseModel):
        step_by_step_analysis: str = Field(description="详细分步推理过程，至少5步，150字以上。")
        reasoning_summary: str = Field(description="简要总结推理过程，约50字。")
        relevant_pages: List[int] = Field(description="保持为空列表。")
        final_answer: Union[str, Literal["N/A"]] = Field(description="公司名称需与问题中完全一致。答案只能是单个公司名或'N/A'。")
```

增强 (Augmentation)

3. 用于创建单次示例 (one-shot) / 少次示例 (few-shot) 提示词的问答对示例

给 LLM 一个或几个“问题-答案”示例，帮助它学会输出你想要的格式和风格。

```
example = r"""示例:
```

```
问题:
```

```
"下列公司中，哪家2022年总资产最低: "A公司", "B公司", "C公司"? 若无数据则排除。"
```

```
答案:
```

```
{
```

```
"step_by_step_analysis": "...",
```

```
"reasoning_summary": "...",
```

```
"relevant_pages": [],
```

```
"final_answer": "C公司"
```

```
}
```

```
"""
```


增强 (Augmentation)

4. 用于插入上下文和查询的模板

让 LLM 能够“看到”相关内容和问题，基于这些内容作答。

```
user_prompt = """
```

```
以下是上下文:
```

```
\\"\\"
```

```
{context}
```

```
\\"\\"
```

```
---
```

```
以下是问题:
```

```
"{question}"
```

```
"""
```

prompts.py 文件就是把所有和 LLM 交互相关的提示词、格式、示例、模板都集中管理，并且分成了四个逻辑块：

- **系统指令**：告诉 LLM 你是谁、你要干什么。
- **Pydantic schema**：告诉 LLM 输出什么格式。
- **示例**：给 LLM 看标准答案，学会怎么答。
- **模板**：动态插入上下文和问题，驱动实际问答。

这样做的好处是结构清晰、易于维护、可复用、易于扩展，也是**当前主流 LLM/RAG 项目的最佳实践**。

生成 (Generation)

RAG 中的“G”（生成）是最耗费精力的。要在这一阶段实现高质量，需要巧妙地使用几种技术。

比赛场景：

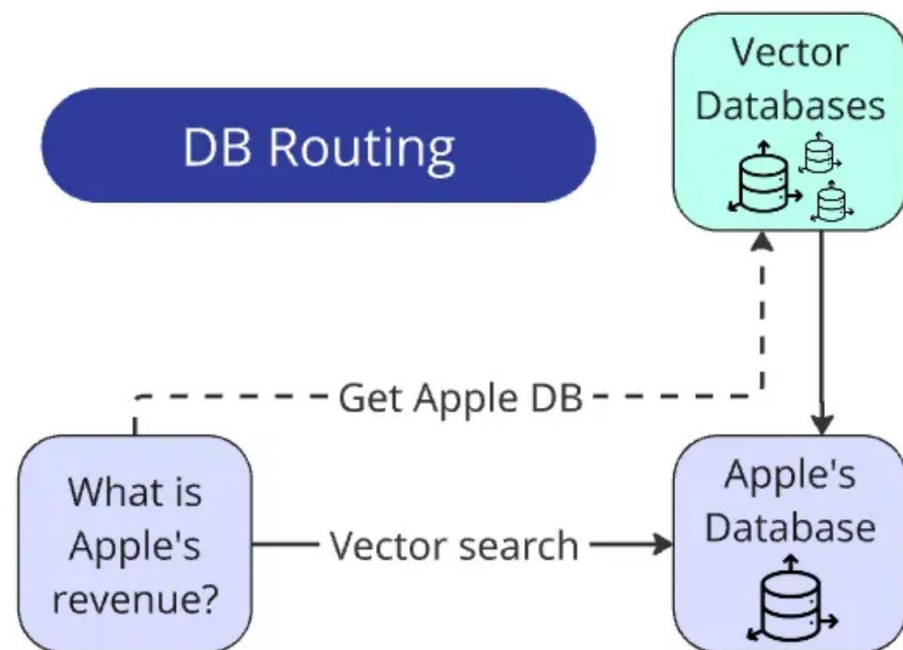
- 每家公司都设置了独立的向量数据库。
- 问题中明确包含公司名，直接匹配即可定位数据库。

实际应用：可能需要 LLM 提取实体或打标签来匹配数据库。

比赛场景与实际应用中的处理理念是一样的，其核心逻辑：

1. 提取公司名 → 2. 匹配对应数据库 → 3. 仅搜索该库。

这样可将搜索范围缩小 100 倍，提升效率。



技术1：将查询路由到数据库

(RAG 系统中最简单却最有用的部分)

生成 (Generation)

比赛要求：答案必须简洁且严格匹配指定数据类型（如 int/float/bool/str/list[str]），类似数据库存储格式。

挑战：

- 每种类型有 3-6 个细节需处理（如去除货币符号、单位换算等）。
- 直接让 LLM 遵守所有规则不可靠：规则越多，出错概率越高。

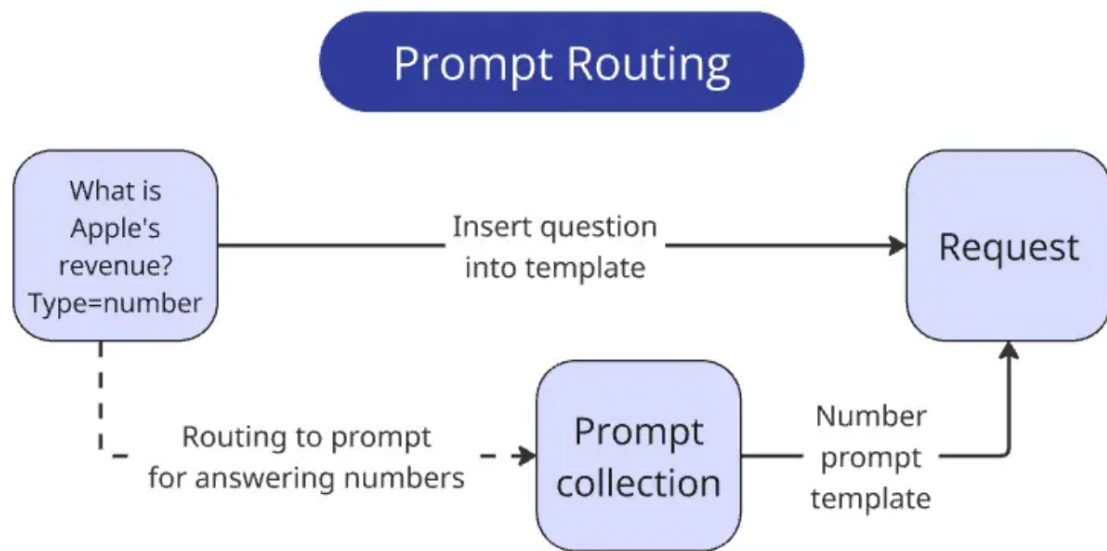
解决方案：

将复杂查询拆解为多个简单步骤，减少单次请求的规则数量。例如：

- 先提取原始数值；
- 再单独处理货币单位转换。

分步操作能显著提升准确性。

核心思路：简化任务 → 降低 LLM 认知负荷 → 确保格式零错误。



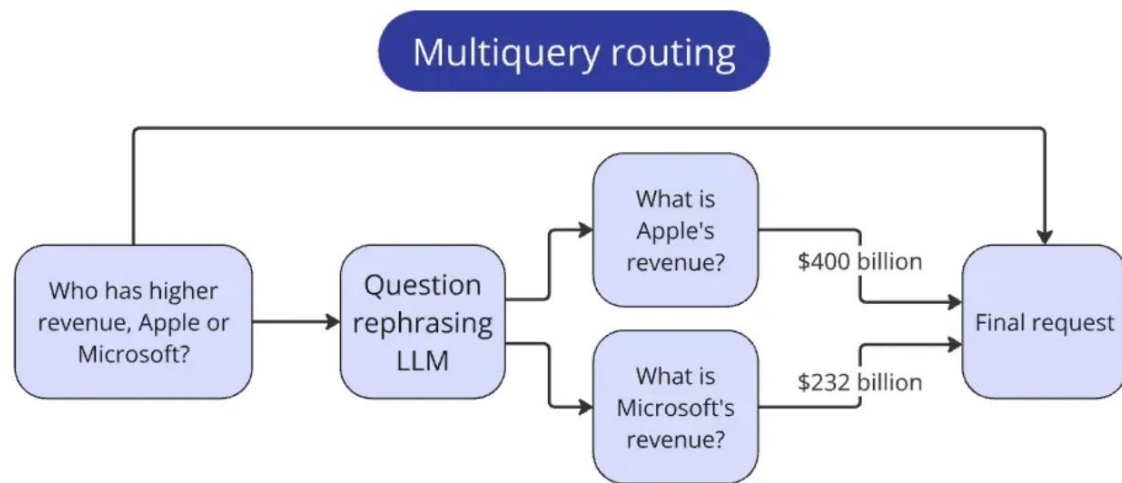
技术2：将查询路由到提示词

生成 (Generation)

在比赛中，由于每个问题的预期响应类型（如 int/float/bool/str/list[str]）都是明确给定的，我们可以采用更直接的解决方案：

为每种数据类型预先设计了专门的提示词模板，然后通过简单的if-else逻辑来自动选择对应的提示词版本。

这种方法既保证了回答格式的精确性，又实现了高效的自动化处理。



技术3：复合查询路由

生成 (Generation)

在比赛中，处理涉及多家公司比较的复杂问题时，采用分步拆解。

比如：苹果和微软谁的营收更高？

1. 问题拆解阶段

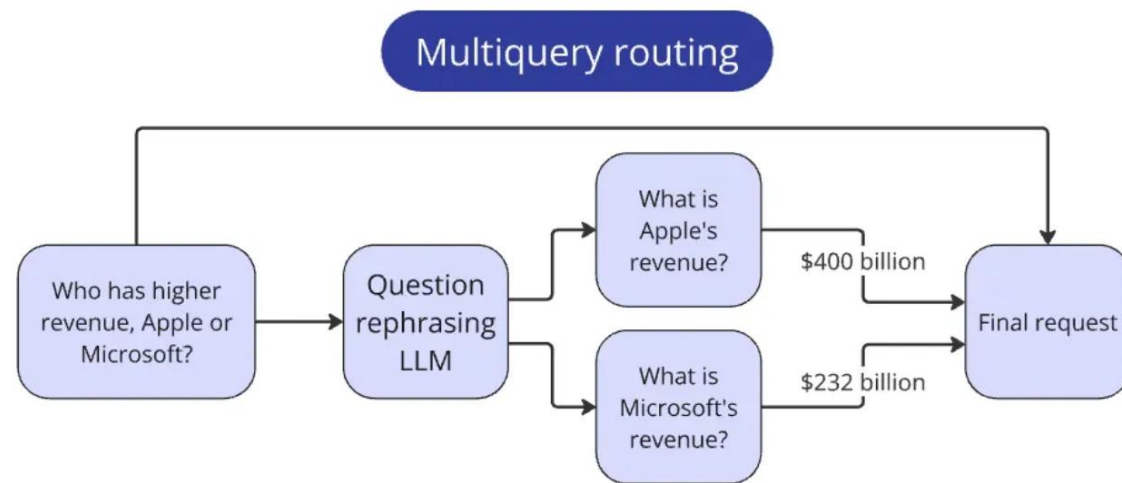
- 首先由LLM将原始比较问题自动拆解为多个独立子问题
- 示例生成："苹果的营收是多少？"和"微软的营收是多少？"

2. 并行查询阶段

- 对每个子问题使用标准查询流程独立处理
- 分别获取各家公司的精确指标数据

3. 综合判断阶段

- 将收集到的各公司数据作为上下文
- 由LLM进行最终比较并生成答案



技术3：复合查询路由

这种模块化处理模式具有高度可扩展性，能够灵活应对各类复杂比较查询

思维链 (Chain of Thoughts, CoT)

思维链 (CoT) 通过让模型在给出最终答案之前“出声思考”，显著提高了答案质量。

早期提示词工程中，“一步步思考” (Think step by step) 这类通用指令能提升答案质量，但对复杂任务仍不足。

尤其较弱模型（如GPT-4o-mini、Llama 8b）常出现“虚假推理”

——先给答案再倒推理由，甚至捏造事实。

为了确保CoT的作用，我们必须清晰地引导模型如何进行推理。解释推理步骤、目标、并提供示例。

思维链 (Chain of Thoughts, CoT)

带有模糊上下文的推理示例：

问题：Ritter Pharmaceuticals Inc. 的研发设备（按成本计）是多少？

模型的推理步骤：

1. 问题询问 Ritter Pharmaceuticals Inc. 的“研发设备，按成本计”。这表明需要从资产负债表中提取一个特定数值，代表专门用于研发的设备的**原始购置成本**，**不包含**任何累计折旧。
2. 上下文（第 35 页）显示“不动产和设备，净额”为 12,500 美元。这是一个**净值**（已扣除折旧），而且是一个**更广泛**的类别，包含所有不动产和设备，而不仅仅是研发设备。
3. 上下文（第 37 页）还提到“机械和设备”的“累计折旧”为 110,000 美元。这代表了总的**折旧**，而不是原始成本，更重要的是，它没有说明这些设备**专门**用于研发。
4. 这两个指标都**不完全**符合请求的指标。“不动产和设备，净额”范围太广，且是折旧后的价值。“累计折旧”仅显示折旧，而非成本，并且缺乏研发的针对性。
5. **由于上下文没有提供仅用于研发设备的原始成本**，且我们无法进行假设、计算，因此答案是“N/A”。

结构化输出 (Structured outputs)

结构化输出 (Structured Output, SO) 是一种强制模型返回标准化格式 (如JSON/Pydantic schema) 的方法。

通过API传递schema, 确保输出始终符合预定结构。

比如, 这里是用于 LLM 重排序的 Pydantic schema



有了这个 schema, LLM 总是会返回一个包含两个字段的JSON——第一个是字符串, 第二个是数字。

```
class RetrievalRankingSingleBlock(BaseModel):  
    """Rank retrieved text block relevance to a query."""  
    reasoning: str = Field(  
        description=(  
            "Analysis of the block, identifying key information and how it"  
            "relates to the query"  
        )  
    )  
    relevance_score: float = Field(  
        description=(  
            "Relevance score from 0 to 1, where 0 is Completely Irrelevant "  
            "and 1 is Perfectly Relevant"  
        )  
    )
```


CoT+SO (思维链+结构化输出)

思维链 + 结构化输出 可以理想地结合使用：

在生成阶段，模型有一个专门用于推理的字段，还有一个单独的字段用于最终答案 => 这样我们可以直接提取答案，而无需从冗长的推理步骤中进行解析。

在用于回答比赛问题的主要schema 中，设置了四个字段：

- **step_by_step_analysis**：初步推理（思维链本身）。
- **reasoning_summary**：前一个字段的精简摘要（用于更轻松地跟踪模型的逻辑）。
- **relevant_pages**：答案引用的报告页码。
- **final_answer**：根据比赛要求格式化后的简洁答案

指令细化 (Instruction Refinement)

Thinking: 什么是指令细化?

当AI回答用户问题时, 需要判断"答案的灵活范围"——比如问"CEO是谁", 实际要包括哪些类似职位? (如总裁、董事总经理等)

这种问题, 在现实中经常遇到, 而且现实数据不完美, 即不同公司用不同头衔称呼老大 (美国叫CEO, 英国可能叫MD)

用户真正想知道的可能比字面意思更广 => 所以需要对 instruction 进行细化

Thinking: 指令细化的挑战是什么?

1. 解释自由度: 该把哪些边缘情况算作答案?

- 硬标准: 只认"CEO"字眼
- 软标准: 包括相似职位 (如总裁/MD), 但要划定界限

2. 如果没有找到答案, 该如何处理:

比如: 问"股息政策有变吗?", 若报告未提及, 该回答"无变更"还是"无信息"?

指令细化 (Instruction Refinement)

解决方案:

- 提前与客户确定规则 (如"接受MD作为CEO的答案吗? ")
- 收集大量边缘案例测试系统

真实案例对比:

- 灵活回答 (适合开放问答):

"Ethan Caldwell是董事总经理 (最接近CEO的角色), 但目前因调查暂停职务..."

- 严格回答 (适合简答比赛):

需预先定义是否允许"MD \approx CEO", 否则可能输出错误简答

指令细化, 这一部分的工作量与整个数据准备阶段不相上下, 因为需要进行无止境的迭代调试、校对答案以及手动分析模型的推理过程。

提示词创建 (Prompt Creation)

在比赛前一周，作者团队利用公开的问题生成器快速创建了100个问题的验证集，并通过手动回答这些问题获得了两个重要收益：

- **量化改进：**验证集成为客观衡量系统性能的标准，通过统计正确率和错误模式，能精准优化提示词和流程。
- **发现隐性规则：**手动分析暴露了问题中隐藏的细节和歧义，促使团队与专家（Rinat）确认回答规范，并将这些规则明确写入提示词指令中。

最终，这些洞察被系统化地整合到提示词设计中，形成更严谨的指令框架。

Prompt.py 实现

在Prompt.py中有多种 prompt，代表不同的任务说明、结构化输出要求和示例，比如：

- boolean：要求输出是/否，并分步推理
- number：要求输出数值，并分步推理
- name：要求输出实体名
- names：要求输出实体名列表

这些 prompt 直接决定了 LLM 的回答风格和结构。

在主流程（如 api_requests.py）中，处理每个问题时会读取 kind 字段，然后根据 kind 选择不同的 prompt 模板，比如：

kind: "boolean" → 用 AnswerWithRAGContextBooleanPrompt

kind: "number" → 用 AnswerWithRAGContextNumberPrompt

kind: "name" → 用 AnswerWithRAGContextNamePrompt

kind: "names" → 用 AnswerWithRAGContextNamesPrompt

Prompt.py 实现

```
class AnswerWithRAGContextBooleanPrompt:
```

```
    instruction = AnswerWithRAGContextSharedPrompt.instruction
```

```
    user_prompt = AnswerWithRAGContextSharedPrompt.user_prompt
```

```
    class AnswerSchema(BaseModel):
```

```
        step_by_step_analysis: str = Field(description="""
```

详细分步推理过程，至少5步，150字以上。特别注意问题措辞，避免被迷惑。有时上下文中看似有答案，但可能并非所问内容，仅为相似项。

```
""")
```

```
        reasoning_summary: str = Field(description="简要总结分步推理过程，约50字。")
```

```
        relevant_pages: List[int] = Field(description="""
```

仅包含直接用于回答问题的信息页面编号。只包括：

- 直接包含答案或明确陈述的页面

- 强有力支持答案的关键信息页面

不要包含仅与答案弱相关或间接相关的页面。

列表中至少应有一个页面。

```
""")
```

```
        final_answer: Union[bool] = Field(description="""
```

一个从上下文中精确提取的布尔值（True或False），直接回答问题。

如果问题问某事是否发生，且上下文有相关信息但未发生，则返回

False。

```
""")
```

```
    pydantic_schema = re.sub(r"^\{4}", "",  
inspect.getsource(AnswerSchema), flags=re.MULTILINE)
```

```
    example = r"""
```

问题：

"万科企业股份有限公司'年报是否宣布了分红政策变更？"

答案：

```
...  
  
{
```

```
    "step_by_step_analysis": "1. 问题询问是否有分红政策变更。\\n2. 年  
报12、18页提到年度分红金额增加，但政策未变。\\n3. 45页有分红细  
节。\\n4. 持续小幅增长，符合既定政策。\\n5. 问题问的是政策变更，  
非金额变化。",
```

```
    "reasoning_summary": "年报显示分红金额变化但政策未变，答案为  
False。",
```

```
    "relevant_pages": [12, 18, 45],
```

```
    "final_answer": false
```

```
}  
...  
"""
```

对应 kind = “boolean” 的prompt

Prompt.py 实现

```
class AnswerWithRAGContextNumberPrompt:
    instruction = AnswerWithRAGContextSharedPrompt.instruction
    user_prompt = AnswerWithRAGContextSharedPrompt.user_prompt
    class AnswerSchema(BaseModel):
        step_by_step_analysis: str = Field(description="""
详细分步推理过程，至少5步，150字以上。
**严格的指标匹配要求：**

1. 明确问题中指标的精确定义，它实际衡量什么？
2. 检查上下文中的所有可能指标。不要只看名称，要关注其实际衡量内容。
3. 仅当上下文指标的含义与目标指标*完全一致*时才接受。可接受同义词，但概念不同则不可。
4. 拒绝（并返回'N/A'）的情况：
    - 上下文指标范围大于或小于问题指标。
    - 上下文指标为相关但非*完全等价*的概念（如代理指标或更宽泛类别）。
    - 需要计算、推导或推断才能作答。
    - 聚合不匹配：问题要求单一值，但上下文仅有总计。
5. 不允许猜测：如对指标等价性有任何疑问，默认返回`N/A`。
""")
```

```
reasoning_summary: str = Field(description="简要总结分步推理过程，约50字。")
```

```
relevant_pages: List[int] = Field(description="""
仅包含直接用于回答问题的信息页面编号。只包括：
- 直接包含答案或明确陈述的页面
- 强有力支持答案的关键信息页面
不要包含仅与答案弱相关或间接相关的页面。
列表中至少应有一个页面。
""")
```

```
final_answer: Union[float, int, Literal['N/A']] = Field(description="""
答案应为精确的数值型指标。
- 百分比示例：
    上下文值：58,3%
    最终答案：58.3
.....
""")
```

对应 kind = “number” 的prompt

Prompt.py 实现

```
class AnswerWithRAGContextNamePrompt:
```

```
    instruction = AnswerWithRAGContextSharedPrompt.instruction
```

```
    user_prompt = AnswerWithRAGContextSharedPrompt.user_prompt
```

```
    class AnswerSchema(BaseModel):
```

```
        step_by_step_analysis: str = Field(description="详细分步推理过程，至少5步，150字以上。特别注意问题措辞，避免被迷惑。有时上下文中看似有答案，但可能并非所问内容，仅为相似项。")
```

```
        reasoning_summary: str = Field(description="简要总结分步推理过程，约50字。")
```

```
        relevant_pages: List[int] = Field(description=""""
```

仅包含直接用于回答问题的信息页面编号。只包括：

- 直接包含答案或明确陈述的页面
- 强有力支持答案的关键信息页面

不要包含仅与答案弱相关或间接相关的页面。

列表中至少应有一个页面。

```
""")
```

```
    final_answer: Union[str, Literal["N/A"]] = Field(description="""
```

如为公司名，需与问题中完全一致。

如为人名，需为全名。

如为产品名，需与上下文完全一致。

不得包含多余信息、词语或注释。

如上下文无相关信息，返回'N/A'。

```
""")
```

对应 kind = “name” 的prompt

Prompt.py 实现

```
class AnswerWithRAGContextNamesPrompt:
```

```
    instruction = AnswerWithRAGContextSharedPrompt.instruction
```

```
    user_prompt = AnswerWithRAGContextSharedPrompt.user_prompt
```

```
    class AnswerSchema(BaseModel):
```

```
        """RAG上下文下多实体/名单类答案的结构定义。"""
```

```
        step_by_step_analysis: str = Field(description="详细分步推理过程，至少5步，150字以上。注意区分实体类型，避免被迷惑。")
```

```
        reasoning_summary: str = Field(description="简要总结推理过程，约50字。")
```

```
        relevant_pages: List[int] = Field(description="""
```

仅包含直接用于回答问题的页面编号。只包括：

- 直接包含答案或明确陈述的页面
- 强有力支持答案的关键信息页面

不要包含仅与答案弱相关或间接相关的页面。

列表中至少应有一个页面。

```
""")
```

```
    final_answer: Union[List[str], Literal["N/A"]] = Field(description="""
```

每个条目需与上下文完全一致。

如问题问职位（如职位变动），仅返回职位名称，不含姓名或其他信息。新任高管也算作职位变动。若同一职位有多次变动，仅返回一次，且职位名称用单数。

示例：['首席技术官', '董事', '首席执行官']

如问题问姓名，仅返回上下文中的全名。

示例：['张三', '李四']

如问题问新产品，仅返回上下文中的产品名。候选产品或测试阶段产品不算新产品。

示例：['生态智能2000', '绿能Pro']

如无信息，返回'N/A'。

```
""")
```

对应 kind = “names” 的prompt

RAG系统调参

拥有一个验证集不仅帮助改进了提示词，也使整个系统受益。

我们将所有关键功能配置化，以便衡量它们的实际效果并微调超参数。以下是一些示例配置字段：

```
class RunConfig:
    # 运行流程参数配置
    use_serialized_tables: bool = False
    parent_document_retrieval: bool = False
    use_vector_dbs: bool = True
    use_bm25_db: bool = False
    llm_reranking: bool = False
    llm_reranking_sample_size: int = 30
    top_n_retrieval: int = 10
    parallel_requests: int = 1 # 并行的数量，需要限制，否则qwen-turbo会超出阈值
    pipeline_details: str = ""
    submission_file: bool = True
    full_context: bool = False
    api_provider: str = "dashscope" # openai
    answering_model: str = "qwen-turbo-latest"
    config_suffix: str = ""
```

通过配置不同的RAG超参数，惊讶地发现：原来被寄予厚望的表格序列化，不仅没有改进系统，反而降低了有效性。

Summary

系统化方法胜于“神奇方案”：成功并非依赖单一突破性技术，而是通过系统化的流程优化，结合并精细调整多种技术。

这些关键因素包括：

- **高质量解析**：确保数据预处理精准、结构化。
- **高效检索**：优化检索效率与准确性。
- **智能路由**：动态分配查询到最合适的处理模块。
- **LLM重排序**：利用大语言模型对检索结果重新排序，显著提升相关性。
- **提示词设计**：精心设计的提示词（prompt engineering）使紧凑模型也能发挥出色性能。

RAG的优化是一个精细化工程问题，RAG问答结果高度依赖对任务细节的深入理解，通过精准微调每个环节（解析、检索、路由、排序等），即使简单技术也能实现显著效果。

打卡：RAG-Challenge



跑通RAG-Challenge，并整理阅读笔记，包括：










- 系统性理解RAG_Challenge中的模块：高质量解析，高效检索，智能路由，LLM重排序，提示词设计
- 对代码实现进行理解，包括src文件夹下面的核心实现：
api_request_parallel_processor.py, api_requests.py, ingestion.py,
parsed_reports_merging.py, pdf_parsing.py, pipeline.py, prompts.py,
questions_processing.py, reranking.py, retrieval.py, tables_serialization.py,
text_splitter.py

阅读笔记的字数任意（可以是200-500字），找到你认为有价值的部分进行理解和整理

CASE：打造自己的RAG系统

TO DO：运行RAG-Challenge-2，并打造自己的RAG系统

打造知识库，将多份投研报告存放到
data/stock_data/pdf_reports中

-  【财报】中芯国际：中芯国际2024年年度报告
-  【东方证券】产能利用率提升，持续推进工艺迭代和产品性能升级
-  【光大证券】中芯国际2025年一季度业绩点评：1Q突发生产问题，2Q业绩有望筑底，自主可控趋势不改
-  【国信证券】工业与汽车触底反弹，良率影响短期营收
-  【华泰证券】中芯国际（688981）：上调港股目标价到63港币，看好DeepSeek推动代工需求强劲增长
-  【上海证券】中芯国际深度研究报告：晶圆制造龙头，领航国产芯片新征程
-  【兴证国际】季度盈利低于预期，看好国产芯片长期空间
-  【中原证券】产能利用率显著提升，持续推进工艺迭代升级——中芯国际(688981)季报点评
-  中芯国际机构调研纪要

针对知识库进行提问，比如：

- 中芯国际在晶圆制造行业中的地位如何？其服务范围和全球布局是怎样的？
- 半导体行业有哪些关键特性，这些特性如何助力中芯国际发展？
- 中芯国际的营收和利润情况近期有何变化？影响因素是什么？
- 中芯国际的收入结构有何变化？尤其是在中国大陆和北美市场的表现如何？
- 美国对中国半导体产业的限制政策对中芯国际有何影响？中芯国际如何应对？

存放到 data/stock_data/questions.json 中

CASE：打造自己的RAG系统

Step1, 跑通RAG-Challenge-2

```
python -m src.pipeline
```

Step2, 简化Markdown生成

原有的pipeline中的处理过程复杂，对其进行简化

1. 解析PDF报告为结构化JSON

```
pipeline.parse_pdf_reports_sequential()
```

2. 序列化表格

```
pipeline.serialize_tables(max_workers=5)
```

3. 将解析后的JSON规整为更简单的每页markdown结构

```
pipeline.merge_reports()
```

4. 导出规整后报告为纯markdown文本

```
pipeline.export_reports_to_markdown()
```



```
改写 pipeline.export_reports_to_markdown()
```

编写pdf_mineru.py, 通过mineru进行文档解析

```
from src import pdf_mineru
```

```
task_id = pdf_mineru.get_task_id(file_name)
```

```
print(f"task_id: {task_id}")
```

```
pdf_mineru.get_result(task_id)
```

CASE：打造自己的RAG系统（mineru解析）

```
import requests
import time
import zipfile
api_key = '你的api_key'

def get_task_id(file_name):
    url='https://mineru.net/api/v4/extract/task'
    header = {
        'Content-Type':'application/json',
        "Authorization":f"Bearer {api_key}".format(api_key)
    }
    pdf_url = 'https://vl-image.oss-cn-shanghai.aliyuncs.com/pdf/' + file_name
    data = {
        'url':pdf_url,
        'is_ocr':True,
        'enable_formula': False,
    }
```

```
res = requests.post(url,headers=header,json=data)
print(res.status_code)
print(res.json())
print(res.json()["data"])
task_id = res.json()["data"]['task_id']
return task_id
```

替换自己的API KEY

将文件上传到url中，或使用本地的mineru进行解析

CASE：打造自己的RAG系统（mineru解析）

```
def get_result(task_id):
    url = f'https://mineru.net/api/v4/extract/task/{task_id}'
    header = {
        'Content-Type': 'application/json',
        "Authorization": f"Bearer {api_key}".format(api_key)
    }
    while True:
        res = requests.get(url, headers=header)
        result = res.json()["data"]
        print(result)
        state = result.get('state')
        err_msg = result.get('err_msg', '')
        # 如果任务还在进行中，等待后重试
        if state in ['pending', 'running']:
            print("任务未完成，等待5秒后重试...")
            time.sleep(5)
            continue
```

```
# 如果有错误，输出错误信息
if err_msg:
    print(f"任务出错: {err_msg}")
    return
# 如果任务完成，下载文件
if state == 'done':
    full_zip_url = result.get('full_zip_url')
    if full_zip_url:
        local_filename = f"{task_id}.zip"
        print(f"开始下载: {full_zip_url}")
        r = requests.get(full_zip_url, stream=True)
        with open(local_filename, 'wb') as f:
            for chunk in r.iter_content(chunk_size=8192):
                if chunk:
                    f.write(chunk)
        print(f"下载完成，已保存到: {local_filename}")
```


CASE：打造自己的RAG系统（mineru解析）

下载完成后自动解压

```
    unzip_file(local_filename)
```

```
else:
```

```
    print("未找到 full_zip_url, 无法下载。")
```

```
    return
```

其他未知状态

```
print(f"未知状态: {state}")
```

```
return
```

解压zip文件的函数

```
def unzip_file(zip_path, extract_dir=None):
```

```
    """
```

解压指定的zip文件到目标文件夹。

:param zip_path: zip文件路径

:param extract_dir: 解压目标文件夹，默认为zip同名目录

```
    """
```

```
if extract_dir is None:
```

```
    extract_dir = zip_path.rstrip('.zip')
```

```
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
```

```
    zip_ref.extractall(extract_dir)
```

```
print(f"已解压到: {extract_dir}")
```

```
if __name__ == "__main__":
```

```
    file_name = '【财报】中芯国际：中芯国际2024年年度报告.pdf'
```

```
    task_id = get_task_id(file_name)
```

```
    print('task_id:', task_id)
```

```
    get_result(task_id)
```

CASE：打造自己的RAG系统 (text_splitter)

Step3, 对Markdown进行切分

原代码是对json进行切分，保存到

test_set/databases/chunked_reports中

#对 json 文件分块，输出还是 json

```
def split_all_reports(self, all_report_dir: Path, output_dir: Path,
serialized_tables_dir: Optional[Path] = None):
```

```
    """
```

批量处理目录下所有报告（json文件），对每个报告进行文本分块，并输出到目标目录。

如果提供了序列化表格目录，会尝试将表格内容插入到对应页面的分块中。

主要用于后续向量化和检索的预处理。

参数：

all_report_dir: 存放待处理报告json的目录

output_dir: 分块后输出的目标目录

serialized_tables_dir: （可选）存放序列化表格的目录

```
    """
```



```
"content": {
  "chunks": [
    {
      "page": 1,
      "length_tokens": 6,
      "text": "# Annual Report\n\n2022",
      "id": 0,
      "type": "content"
    },
    {
      "page": 2,
      "length_tokens": 129,
      "text": "# Table of Contents\n\n\n| Management's Discussion and Analy",
      "id": 1,
      "type": "content"
    },
    {
      "page": 3,
      "length_tokens": 10,
      "text": "# Message to Shareholders\n\nDear Shareholders,",
      "id": 2,
```

CASE：打造自己的RAG系统（text_splitter）

对text_splitter.py进行改写，添加 split_markdown_reports 和 split_markdown_file 函数

```
split_markdown_reports(all_md_dir: Path, output_dir: Path,  
chunk_size: int = 30, chunk_overlap: int = 5, subset_csv: Path = None)`
```

- 作用：批量处理目录下所有markdown文件，分块并输出为json文件到目标目录。支持通过subset.csv补充公司名、sha1等元信息。
- 参数：
 - all_md_dir——md文件目录；
 - output_dir——输出目录；
 - chunk_size——每块最大行数；
 - chunk_overlap——分块重叠行数；
 - subset_csv——可选，文件名到公司名映射表。
- 返回：无，直接输出分块结果到文件。

```
split_markdown_file(md_path: Path, chunk_size: int = 30,  
chunk_overlap: int = 5) -> List[Dict]`
```

- 作用：按行分割markdown文件，每个分块记录起止行号和内容，适合处理结构化较强的md文本。
- 参数：
 - md_path——markdown文件路径；
 - chunk_size——每块最大行数；
 - chunk_overlap——分块重叠行数。
- 返回：分块内容列表。

CASE：打造自己的RAG系统（text_splitter）

```
"metainfo": {
  "sha1": "stock_10001",
  "company_name": "中芯国际",
  "file_name": "【财报】中芯国际：中芯国际2024年年度报告.md"
},
"content": {
  "chunks": [
    {
      "lines": [
        1,
        30
      ],
      "text": "# 中芯国际集成电路制造有限公司2024 年年度报告 \n\n# 重要提示"
    },
    {
      "lines": [
        26,
        55
      ],
      "text": "\n\n适用□不适用 \n公司治理特殊安排情况： \n\n本公司为红筹企业"
    }
  ]
}
```

```
{
  "lines": [
    51,
    80
  ],
  "text": "第三节 公司简介和主要财务指标。> \n\n第四节 管理层讨论与分析。12"
},
{
  "lines": [
    76,
    105
  ],
  "text": "尊敬的各位股东、投资人： \n\n2024年是中芯国际推动高质量发展取得新"
},
{
  "lines": [
    101,
    130
  ],
  "text": "# 三、信息披露及备置地点 \n\n<html><body><table><tr><td>公司选"
}
```

在pipeline.chunk_reports() 中调用了text_splitter.split_markdown_reports，对文档进行了切分
保存到 stock_data/databases/chunked_reports

CASE：打造自己的RAG系统（kind=string）

Thinking：针对kind=string的问题，如何实现？添加 AnswerWithRAGContextStringPrompt 功能

帮我查找，程序中实现了

AnswerWithRAGContextNumberPrompt

AnswerWithRAGContextBooleanPrompt

AnswerWithRAGContextNamesPrompt

ComparativeAnswerPrompt

===

在理解这些实现的基础上，我想实现一个

AnswerWithRAGContextStringPrompt，即开放性的问题，回答的是一段文本

> Searched and read prompts.py

```
prompts.py +42 ✓
```

```
class AnswerWithRAGContextStringPrompt:
    instruction = AnswerWithRAGContextSharedPrompt.instruction
    user_prompt = AnswerWithRAGContextSharedPrompt.user_prompt

    class AnswerSchema(BaseModel):
        step_by_step_analysis: str = Field(description="""
            详细分步推理过程，至少5步，150字以上。请结合上下文信息，逐步分析并归纳答案。
            """)
        reasoning_summary: str = Field(description="简要总结分步推理过程，约")
        relevant_pages: List[int] = Field(description="")
```

仅包含直接用于回答问题的信息页面编号。只包括：

- 直接包含答案或明确陈述的页面
- 强有力支持答案的关键信息页面

已为你新增 AnswerWithRAGContextStringPrompt，实现方式与现有的 Number/Boolean/NamesPrompt 保持一致，支持开放性文本问题，输出为一段完整文本。注释、字段说明和示例均为中文，便于直接用于 RAG 场景下的自由文本问答。如需在 API 或调用链中集成此 Prompt，请告知具体调用点！

CASE：打造自己的RAG系统（kind=string）

```
class AnswerWithRAGContextStringPrompt:
    instruction = AnswerWithRAGContextSharedPrompt.instruction
    user_prompt = AnswerWithRAGContextSharedPrompt.user_prompt

    class AnswerSchema(BaseModel):
        step_by_step_analysis: str = Field(description="""
详细分步推理过程，至少5步，150字以上。请结合上下文信息，逐步分析并归纳答案。
""")
        reasoning_summary: str = Field(description="简要总结分步推理过程，约50字。")
        relevant_pages: List[int] = Field(description="""
仅包含直接用于回答问题的信息页面编号。只包括：
- 直接包含答案或明确陈述的页面
- 强有力支持答案的关键信息页面
不要包含仅与答案弱相关或间接相关的页面。
列表中至少应有一个页面。
""")
        final_answer: str = Field(description="""
最终答案为一段完整、连贯的文本，需基于上下文内容作答。
如上下文无相关信息，可简要说明未找到答案。
""")

pydantic_schema = re.sub(r"^\s{4}", "", inspect.getsource(AnswerSchema), flags=re.MULTILINE)
```

```
example = r'''
```

示例：

问题：

"请简要总结'万科企业股份有限公司'2022年主营业务的主要内容。"

答案：

```
'''
```

```
{
```

```
    "step_by_step_analysis": "1. 问题要求总结2022年万科企业股份有限公司的主营业务。\\n2. 年报第10-12页
```

```
    "reasoning_summary": "年报10-12页明确列出主营业务，答案基于原文归纳。",
```

```
    "relevant_pages": [10, 11, 12],
```

```
    "final_answer": "万科企业股份有限公司2022年主营业务包括房地产开发、物业服务、租赁住房、物流仓储等，
```

```
}
```

```
'''
```

```
'''
```

```
system_prompt = build_system_prompt(instruction, example)
```

```
system_prompt_with_schema = build_system_prompt(instruction, example, pydantic_schema)
```

prompts.py 中添加 AnswerWithRAGContextStringPrompt

CASE：打造自己的RAG系统（kind=string）

这里对应的kind为 string，类似

AnswerWithRAGContextNumberPrompt 对应的kind=number

Q Searched 4 times and read api_requests.py

api_requests.py +6 ✓

```
        if use_schema_prompt else prompts.ComparativeAnswerPrompt.system_prompt)
    response_format = prompts.ComparativeAnswerPrompt.AnswerSchema
    user_prompt = prompts.ComparativeAnswerPrompt.user_prompt

    elif schema == "string":
        # 新增：支持开放性文本问题
        system_prompt = (prompts.AnswerWithRAGContextStringPrompt.system_prompt_with_schema
                        if use_schema_prompt else prompts.AnswerWithRAGContextStringPrompt.system_prompt)
        response_format = prompts.AnswerWithRAGContextStringPrompt.AnswerSchema
        user_prompt = prompts.AnswerWithRAGContextStringPrompt.user_prompt

    else:
        raise ValueError(f"Unsupported schema: {schema}")
    return system_prompt, response_format, user_prompt
```

已在 `_build_rag_context_prompts` 方法中增加了对 `schema == "string"` 的分支，自动调用 `AnswerWithRAGContextStringPrompt`，实现方式与 `number/boolean/names` 完全一致。这样只要 `kind` 或 `schema` 为 `string`，即可支持开放性文本问题的 RAG 问答。

如需前端或调用链适配 `string` 类型，或有其他自定义需求，请继续说明！

```
elif schema == "string":
    # 新增：支持开放性文本问题
    system_prompt = (prompts.AnswerWithRAGContextStringPrompt.system_prompt_with_schema
                    if use_schema_prompt else prompts.AnswerWithRAGContextStringPrompt.system_prompt)
    response_format = prompts.AnswerWithRAGContextStringPrompt.AnswerSchema
    user_prompt = prompts.AnswerWithRAGContextStringPrompt.user_prompt
else:
    raise ValueError(f"Unsupported schema: {schema}")
return system_prompt, response_format, user_prompt
```

api_request.py 中添加 `schema=="string"`

CASE：打造自己的RAG系统（界面）

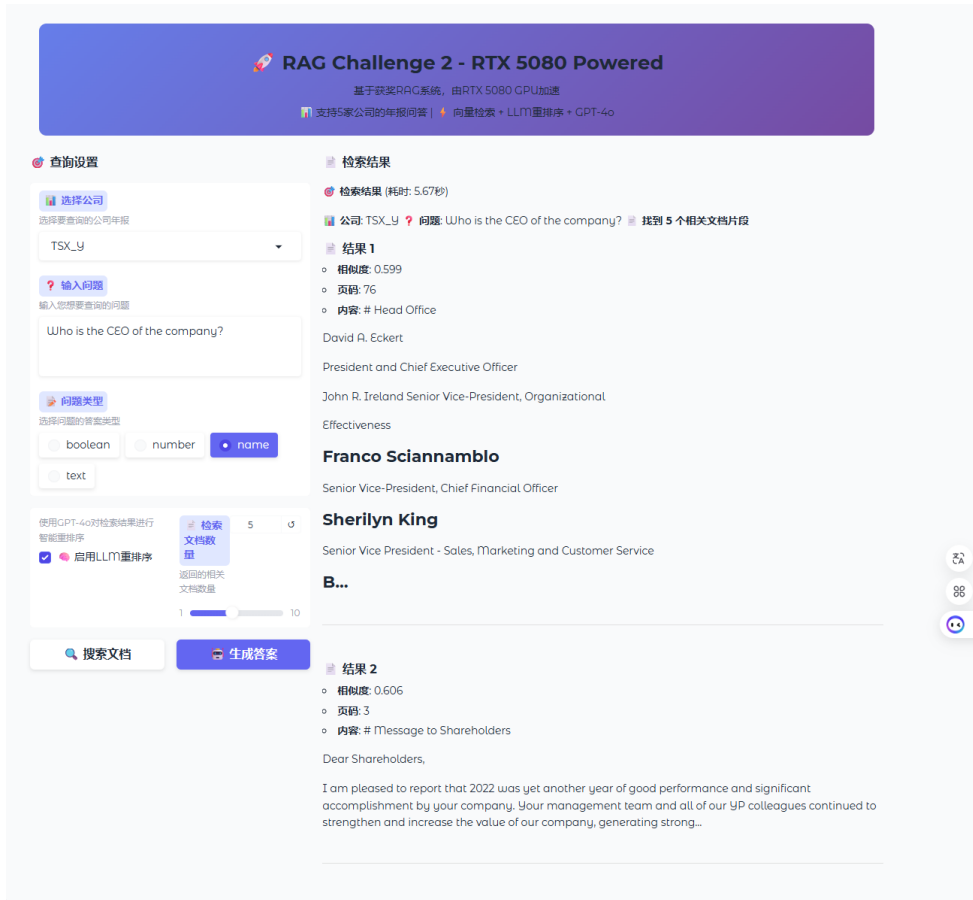
Thinking：如何搭建自己的RAG系统界面？

可以找到一个还不错的RAG系统界面，截屏下来，让Cursor进行参考，使用streamlit 或者 gradio

帮我使用streamlit搭建可视化界面，用户提问了一个问题，后台调用类似 `pipeline.process_questions()`

默认kind=string，这里只需要单个问题即可，然后回答用户，在界面上显示。

参考给你的这个界面风格



CASE：打造自己的RAG系统（界面）

> Listed directories

app_streamlit.py +51 -1 ✓

```
import streamlit as st
from src.pipeline import Pipeline

# 初始化 pipeline（假设有默认配置，或可根据需要调整）
pipeline = Pipeline()

st.set_page_config(page_title="RAG Challenge 2 - RTX 5080 Pow

# 页面标题
st.markdown("""
<div style='background: linear-gradient(90deg, #7b2ff2 0%, #f
    <h2 style='color: white; margin: 0;'>🚀 RAG Challenge 2
    <div style='color: #fff; font-size: 16px;'>基于深度RAG系统
</div>
```

解决方案：为 Pipeline 添加 answer_single_question 方法

你可以在 `src/pipeline.py` 里添加如下方法，复用已有的 `QuestionsProcessor` 逻辑，支持单问题、kind=string 的即时推理。

推荐实现（直接加到 Pipeline 类里）：

app_streamlit.py +2 ✓

```
import streamlit as st
from pathlib import Path
from src.pipeline import Pipeline, max_config
from src.questions_processing import QuestionsProcessor

# 你可以让 root_path 固定，也可以让用户输入
root_path = Path("data/stock_data")
|         |         st.error(f"生成答案时出错: {e}")
else:
    st.info("请在左侧输入问题并点击【生成答案】")
```

CASE：打造自己的RAG系统（界面）

查询设置

输入问题

中芯国际的主营业务收入如何？

生成答案

RAG Challenge 2

基于深度RAG系统，由RTX 5080 GPU加速 | 支持多公司年报问答 | 向量检索+LLM推理+GPT-4o

检索结果

分步推理：

推理摘要：

相关页面：

[0 : 0]

最终答案：

{ "step_by_step_analysis": "1. 问题询问中芯国际的主营业务收入。\\n2. 年报第5页提到，公司2024年销售收入为人民币578亿元。\\n3. 该数据直接反映了主营业务收入。\\n4. 需要确认是否还有其他相关页面提供更详细的信息，但根据现有内容，已足够回答问题。\\n5. 形成最终答案。", "reasoning_summary": "年报第5页明确列出了2024年的销售收入，即主营业务收入。", "relevant_pages": [5], "final_answer": "中芯国际2024年的主营业务收入为人民币578亿元。" }

经过一系列的沟通调试，搭建出来了streamlit界面，但是输出的结果没有正确解析，还需要完善

CASE：打造自己的RAG系统（界面）

<

查询设置

输入问题

中芯国际的2024年主营业务是多少?

生成答案

 RAG Challenge 2

基于深度RAG系统，由RTX 5080 GPU加速 | 支持多公司年报问答 | 向量检索+LLM推理+GPT-4o

检索结果

分步推理：

1. 问题要求确定中芯国际2024年的主营业务金额。

2. 年报中提到，2024年中芯国际的销售收入为人民币578亿元。

3. 该数据直接反映了公司的主营业务收入。

4. 需要确认是否还有其他相关数据或补充说明。

5. 根据年报内容，没有其他与主营业务收入相关的额外信息，因此可以得出结论。

推理摘要：

年报明确指出2024年中芯国际的销售收入为578亿元，这是主营业务的核心数据。

相关页面：

▼

[

0 : 0

]

最终答案：

中芯国际2024年的主营业务收入为人民币578亿元。

结果解析正确了，但是检索时间过长，还需要进一步优化

打卡：打造自己的RAG系统



结合RAG-Challenge，改成自己的RAG系统，包括：

- 使用DashScope API KEY，替换原有的openai API

可以使用qwen-turbo-latest作为LLM， text_embedding_v1作为embedding模型

- 使用 MinerU替换现有的docling解析
- 更新中文知识库，比如选用多份研报报告作为知识库 data/stock_data/pdf_reports
- 设置相关的问题清单 data/stock_data/question.json
- 针对开放式的问题，进行 kind=string的类型设置
- 搭建前端页面，比如使用 streamlit

查询设置

输入问题

中芯国际的2024年主营业务是多少？

生成答案

 RAG Challenge 2

基于深度RAG系统，由RTX 5080 GPU加速 | 支持多公司年报问答 | 向量检索+LLM推理+GPT-4o

检索结果

分步推理：

1. 问题要求确定中芯国际2024年的主营业务金额。
2. 年报中提到，2024年中芯国际的销售收入为人民币578亿元。
3. 该数据直接反映了公司的主营业务收入。
4. 需要确认是否还有其他相关数据或补充说明。
5. 根据年报内容，没有其他与主营业务收入相关的额外信息，因此可以得出结论。

推理摘要：




年报明确指出2024年中芯国际的销售收入为578亿元，这是主营业务的核心数据。

相关页面：

[0 : 0]

最终答案：

中芯国际2024年的主营业务收入为人民币578亿元。



Thank You
Using data to solve problems