Policy KB Assistant 面试项目问答

这份文档用于面试前准备。它不是对外 README，而是把面试官可能追问的问题整理成可回答的口径。

推荐总开场：

> *这个项目是一个企业 IT 服务台场景的 RAG + Agent 系统。它支持制度问答、一句话建单、工单查询/评论/催办/取消确认、多轮记忆、审计追溯、MCP 工具暴露、Redis 问答缓存、运行监控和 Docker Compose 云部署。我重点解决的问题是：如何把 LLM 从“能回答”变成“能在受约束系统里安全办事”。*

1. 项目总体

Q1：这个项目一句话是什么？

这是一个面向企业内部 IT 服务台的 RAG + Agent 工单助手。

用户可以用自然语言问制度问题，也可以说“我的电脑连不上公司内网，帮我提交工单”。系统会自动判断是问答还是工单动作，执行后返回结果，并把问答、路由、工具调用和工单操作写入审计日志。

Q2：这个项目和普通 RAG demo 有什么区别？

普通 RAG demo 通常只做“提问 -> 检索 -> 回答”。我这个项目做了更完整的业务闭环：

- 有登录和当前用户。
- 有工单创建、草稿续办、查单、评论、催办、取消。
- 有 Agent 路由和工具执行。
- 有高风险动作二次确认。
- 有审计追溯。
- 有 Redis 缓存和运行监控。
- 有 Docker Compose 部署。

所以它更接近一个可运行的 LLM 应用系统，而不是单点模型调用。

Q3：你在项目里主要负责什么？

这个项目是我独立开发的。我从最初的知识库问答开始，逐步加入：

- RAG 检索和评测。
- 文档切块优化。
- FastAPI 服务层。
- ITSM 工单模型。
- Agent 路由、Planner、Validator。
- LangGraph 编排。
- MCP 工具服务。
- 登录鉴权。
- Redis 缓存。
- 监控和成本统计。
- Docker Compose 云部署。

2. 系统流程

Q4：用户问一句话后，系统怎么走？

以 `/agent` 为例：

```text
用户登录
-> UI 调 POST /agent
-> FastAPI 校验 JWT
-> Redis 限流
-> Agent 判断意图
-> 走 ASK / CREATE_TICKET / NEED_MORE_INFO / LOOKUP_TICKET 等分支
-> 如果是问答，进入 RAG 链路
-> 如果是工单，进入工单服务层
-> 写数据库和审计日志
-> 返回结构化响应
```

Q5：问答链路怎么走？

问答链路是：

```text
问题标准化
-> Redis ASK 缓存查询
-> Dense 向量检索
-> BM25 关键词检索
-> RRF 融合排序
-> 可选 rerank
-> LLM 基于证据生成 JSON
-> JSON 校验/修复/fallback
-> 写 kb_queries 和 audit_logs
-> 返回 answer/citations/meta
```

缓存命中时不会调用模型，但仍会生成新的 request_id 并落库，保证审计完整。

Q6：建单链路怎么走？

建单链路是：

```text
用户描述故障
-> Agent 识别 create_ticket
-> 抽取 title/category/priority/location/contact
-> 字段完整：创建 tickets
-> 字段缺失：创建 ticket_drafts
-> 关联 kb_request_id
```

-> 写 audit_logs
-> 返回 route 和 ticket/draft
```

如果用户补充“地点是北京总部，联系方式是 138...”，系统会通过 draft_id 继续草稿，字段齐全后转成正式工单。

### Q7：取消工单为什么要二次确认？

取消工单是高风险动作。如果用户一句话里说“取消上一个工单”，系统不能直接取消，因为可能误识别。

我的设计是：

```text
第一次请求：返回 NEED_CONFIRMATION + confirm_token
第二次确认：带 confirm_token 才执行 CANCEL_TICKET
```

后端会校验 token 是否存在、是否属于当前用户、是否过期、是否已消费。这样可以防止误操作。

3. RAG 相关

Q8：为什么不用纯向量检索？

纯向量检索对语义相近的问题比较好，但企业制度里有很多专有名词、系统名、部门名、工单编号、条款编号。这些内容有时靠关键词更稳定。

所以我做了 Dense + BM25 + RRF：

- Dense 负责语义泛化。
- BM25 负责关键词和专有名词。
- RRF 负责低成本融合排序。

Q9：RRF 是什么？为什么选它？

RRF 是 Reciprocal Rank Fusion。它不直接依赖不同检索器的原始 score，而是根据排名融合：

```text
score = sum(1 / (k + rank_i))
```

我选择它的原因：

- 简单稳定。
- 不需要训练。
- 能融合 Dense 和 BM25 这种分数尺度不同的检索器。
- 实测延迟增加很小。

Q10：为什么不默认开 rerank？
```

Rerank 质量更高，但 CPU 环境下延迟太大。

我的评测里 Hybrid RRF + Rerank 的质量最好，但 p50 检索延迟明显高于 Hybrid RRF。项目默认实时模式选择 Hybrid RRF，是因为它在质量和响应速度之间更均衡。

可以这样回答：

> *Rerank 适合高质量模式或离线评测，不适合作为小云服务器上的默认实时路径。*

Q11：你的 RAG 做了什么量化评测？

我构建了 130 条制度问答评测集，对比 Dense、Hybrid RRF、Hybrid RRF + Rerank。

关键结果：

- Dense Top5 召回：90.00%
- Hybrid RRF Top5 召回：96.15%
- Dense MRR：0.7955
- Hybrid RRF MRR：0.8574
- Hybrid RRF p50 检索延迟只增加约 5ms

切块方面，structured_hybrid 相比固定窗口：

- 自动答案要点覆盖率 APC 提升。
- Citation 输出率提升。
- 拒答率下降。

Q12：为什么设计 structured_hybrid 切块？

企业制度不是普通文章，而是“章-条-款”的层级结构。固定窗口切块容易把一个条款切断，导致引用不完整。

structured_hybrid 会尽量保留标题、条款和上下文，必要时做短块补充和跨页窗口。这让检索结果更像完整制度片段，而不是碎片。

Q13：如果检索结果不准，你怎么排查？

我会按链路排查：

1. 看问题是否在评测集中。
2. 看 Dense top-k 是否召回正确文档。
3. 看 BM25 是否命中关键词。
4. 看 RRF 排名是否把正确文档排上来。
5. 看 chunk 是否切得太碎或丢标题。
6. 看 rerank 是否改善。
7. 看最终 LLM 是否正确引用证据。

项目里 `retrieve_topk`、citations、`kb_queries` 和评测脚本都能辅助排查。

4. Agent 相关

Q14：你怎么做 Agent 路由？

我采用规则路由 + LLM Planner 的混合方式。

规则路由处理高频、确定性强的场景：

- 建单。
- 补充信息。
- 确认操作。
- 查单。

LLM Planner 处理长尾表达，把用户输入转成 `ToolPlan`。

这么做的原因是：不是所有场景都需要 LLM 决策。高频固定意图用规则更稳定、更便宜、更快。

Q15：为什么不让 LLM 直接调用数据库？

因为风险太高。LLM 输出可能：

- 参数缺失。
- 工具名幻觉。
- 错误识别工单号。
- 直接执行高风险动作。

所以我让 LLM 只生成计划，后端负责：

- Pydantic schema 校验。
- 工具白名单。
- 用户权限/所有权校验。
- 高风险确认。
- 审计落库。

Q16：ToolPlan 长什么样？

核心字段是：

```json
{
  "tool": "cancel_ticket",
  "args": {
    "ticket_id": "TCK-2026-XXXX",
    "reason": "问题已恢复"
  },
  "need_confirmation": true,
  "missing_fields": []
}
```

但这个计划不会直接执行，必须通过 Validator。

Q17：LangGraph 在项目里解决了什么？

LangGraph 主要解决 Agent 流程编排问题。

当系统只有一个 if/else 路由时，不需要 LangGraph；但当流程变成多节点、多状态、多分支后，图结构更清晰。

项目里 LangGraph 负责：

- 状态管理。
- 条件路由。
- ASK 节点。
- 工具节点。
- 工作记忆更新。

Q18：规则路由和 LangGraph 是否共用底层能力？

是的，后来我专门抽象了 `ask_pipeline.run_cached_ask_steps()`，让传统 ASK 和 LangGraph ASK 共享同一套底层问答和 Redis 缓存逻辑。

这样避免出现两个问答入口行为不一致。

5. 工单系统

Q19：工单有哪些能力？

主要包括：

- 创建工单。
- 缺字段草稿。
- 续办草稿。
- 查询工单。
- 追加评论。
- 催办。
- 取消。
- 状态更新。

Q20：为什么需要草稿？

用户自然语言经常信息不完整，比如：

```text
帮我提交网络故障工单
```

缺地点和联系方式时，如果直接失败，体验不好；如果胡乱建单，数据质量差。

所以系统会创建 draft，保存已抽取字段和缺失字段，用户补充后继续提交。

Q21：怎么防止重复建单？

项目里有几层措施：

- 直接 `/tickets` 建单要求 `Idempotency-Key`。
- Agent 自动生成幂等窗口。
- 草稿被消费后记录 `source_draft_id`。
- 重复续办同一个 draft 时返回同一个 ticket_id。

Q22：怎么保证用户不能操作别人的草稿？

草稿有 `owner_user_id`。继续草稿时会校验当前用户是否是草稿 owner。不是 owner 时返回 `draft_not_found`，避免暴露草稿存在性。

6. 记忆

Q23：项目里记忆怎么分层？

层级	作用
L0 Working Memory	单次请求内临时状态
L1 Session Memory	当前会话最近轮次和待办
L2 Task Memory	草稿、待确认动作
L3 User Memory	默认地点、联系方式等偏好
L4 Episodic Memory	从审计和历史记录回放事件

Q24：记忆会不会带来隐私或误操作风险？

会，所以我做了边界：

- 高风险确认 token 不完整暴露，只保存前缀或状态。
- 用户偏好只保存低风险字段。
- 草稿和确认动作有 owner 和过期时间。
- 操作历史通过审计表回放，不靠模型自由发挥。

7. MCP

Q25：为什么要接 MCP？

MCP 的意义是把项目里的业务工具能力暴露给外部 Agent/IDE/Host，而不是只能通过 Web UI 调用。

但我没有重写一套逻辑，而是让 MCP 工具复用服务层：

```text
MCP tool
-> mcp_wrapper executor
```

-> services 工具函数
-> DB / audit
```

### Q26：MCP 和 Web Agent 怎么保持一致？

核心是复用：

- 同一套工具定义。
- 同一套服务函数。
- 同一套确认和审计逻辑。

这样 Web 调用和 MCP 调用不会变成两套行为。

## 8. 登录、安全与审计

### Q27：项目怎么做登录？

项目有：

- `/auth/register`
- `/auth/login`
- `/auth/me`

登录成功后返回 JWT。UI 保存 token，后续请求带：

```text
Authorization: Bearer <token>
```

Q28：项目的安全边界是什么？

这是 demo 级安全，不是完整生产级 RBAC。

已经做的：

- JWT 当前用户。
- 写接口登录态校验。
- Redis 限流。
- 工具白名单。
- Pydantic schema 校验。
- 草稿 owner 校验。
- 高风险动作 confirm_token。
- 审计日志。

未做的：

- 企业 SSO。
- 完整 RBAC。
```

- 多租户隔离。
- WAF / HTTPS / Nginx。

Q29：审计日志有什么价值？

审计日志能回答：

- 用户输入了什么。
- Agent 判定了什么 route。
- Planner 生成了什么计划。
- 执行了哪个工具。
- 操作了哪个工单。
- 为什么失败或拒绝。

这对企业 Agent 很重要，因为 Agent 不只是在回答，还会操作业务数据。

9. Redis 缓存与性能优化

Q30：Redis 缓存缓存了什么？

缓存的是非流式 ASK 的标准化结果：

- answer
- citations
- trace_hits
- output_meta
- 原始 retrieve/answer 延迟

不缓存 request_id 和 query_id，因为它们属于单次请求。

Q31：缓存 key 怎么设计？

缓存 key 基于：

- 标准化问题。
- department。
- collection。
- embedding 模型。
- LLM 模型。
- retrieval mode。
- rerank 配置。
- namespace version。

这样可以避免模型或知识库配置变更后误用旧缓存。

Q32：为什么缓存命中还要落库？

因为审计链路不能断。

即使命中缓存，这次用户请求也是真实发生的，所以仍然要：

- 生成新的 request_id。
- 写入 kb_queries。
- 写入 audit_logs。
- 在 meta 里标记 `cache_hit`。

Q33：缓存效果如何？

本地实测：

```text
第一次同问：约 28.6s，cache stored
第二次同问：约 24ms，attempt_stage=cache_hit
```

这可以明显减少重复政策问答的延迟和模型成本。

10. 监控与成本

Q34：运行监控看什么？

`/ops/metrics` 和 UI 监控页主要看：

- ASK 总数。
- 成功/失败。
- JSON 有效率。
- citation 输出率。
- repair/fallback 比例。
- 平均延迟、p50/p95。
- token 总量。
- 估算成本。
- route/action 分布。
- 拒绝原因。

Q35：token 成本怎么统计？

优先使用模型 API 返回的 usage：

- prompt_tokens
- completion_tokens
- total_tokens

如果 API 没返回 usage，就用字符数做粗估，并标记 `token_usage_estimated=true`。

成本按环境变量配置的单价估算，主要用于趋势观察，不作为严格账单。

Q36：为什么监控页普通用户也能看到？

这是 demo 取舍。为了面试展示，登录用户可以直接看到运行监控。

真实生产应该按角色控制：

- 普通用户看不到成本和系统指标。
- 管理员/运维角色可以查看。

11. 部署

Q37：项目怎么部署？

用 Docker Compose 单机部署：

```text
ui
api
mcp
kb-init
postgres
redis
qdrant
```

`kb-init` 是一次性初始化任务，负责检查 Qdrant collection 是否存在，不存在就自动 ingest。

Q38：没有 GPU 怎么做向量检索？

没有 GPU 也可以做：

- embedding 模型用 CPU 跑。
- 小云服务器用 `BAAI/bge-small-zh-v1.5`。
- Qdrant 是 CPU 向量数据库。
- 生产回答调用外部 LLM API，不在本地跑大模型。

GPU 不是必须，代价是首次入库和 embedding 慢一些。

Q39：2 核 4G 小服务器怎么适配？

我做了这些取舍：

- 使用 `bge-small` 而不是 `bge-large`。
- 加 4G swap。
- 关闭默认 rerank。
- 使用外部 LLM API。
- Redis 缓存减少重复模型调用。
- 只暴露 8501，不暴露数据库和中间件。

Q40：云部署遇到过什么问题？

主要问题：

1. GitHub clone 中断：用浅克隆或 zip 上传解决。
2. Docker Hub 拉取慢：配置镜像源。
3. pip install 慢：使用阿里云 PyPI 源和 build cache。
4. HuggingFace 下载慢：配置 `HF_ENDPOINT`。
5. PDF 上传后容器看不到：通过 `compose.override.yaml` 挂载 `data/raw`。
6. 安全组端口：只开放 22 和 8501。

12. 技术取舍

Q41：为什么前端用 Streamlit？

项目重点是后端 RAG/Agent 工程链路，不是前端复杂交互。

Streamlit 能快速展示：

- 登录。
- Agent 输入。
- 工单管理。
- 审计追溯。
- 运行监控。

如果生产化，可以替换成 React/TypeScript，但当前求职项目阶段，Streamlit 更适合快速交付。

Q42：为什么用 FastAPI？

FastAPI 适合这个项目：

- Pydantic schema 强。
- 异步接口友好。
- OpenAPI 文档自动生成。
- 路由和依赖注入清晰。
- Python AI 生态兼容好。

Q43：为什么用 PostgreSQL？

工单、用户、审计、记忆都是关系型数据，需要事务、索引和结构化查询。PostgreSQL 比单纯 JSON 文件或 SQLite 更适合作为云部署演示。

SQLite 保留给最小本地模式。

Q44：为什么用 Redis？

Redis 用在三类场景：

- 登录/Agent 限流。
- 幂等控制。
- ASK 结果缓存。

这些都是高频、轻量、适合 TTL 的状态。

Q45：为什么用 Qdrant？

Qdrant 是专门的向量数据库，适合：

- 存储 embedding。
- top-k 向量检索。
- payload 保存 doc_id/page/snippet。
- Docker Compose 部署简单。

13. 可能被质疑的问题

Q46：这个项目是不是没有真实落地？

可以这样答：

> *它不是接入真实企业工单系统的生产项目，但我已经把上线前的核心工程链路做完整了：登录、RAG、Agent、工具执行、数据库、缓存、审计、监控、Docker Compose 云部署。真实落地时主要是把 ITSM-lite 的 tickets 表替换成企业已有工单系统 API，Agent 的工具层和校验层可以复用。*

Q47：为什么不直接用 LangChain Agent？

因为这个项目有工单等真实业务动作，不能让通用 Agent 自由调用工具。

我更关注可控性：

- 计划结构可校验。
- 工具范围可控。
- 风险动作可确认。
- 每步可审计。

LangGraph/Planner 是辅助编排，最终执行权在后端。

Q48：为什么不用本地大模型？

云服务器没有 GPU，本地跑大模型不现实。项目选择外部 OpenAI-compatible API，把有限资源用于：

- API 服务。
- Qdrant。
- Postgres。
- Redis。
- embedding 入库。

这是成本和性能取舍。

Q49：为什么不做完整权限系统？

项目目标是展示 RAG + Agent 工程链路。完整 RBAC、组织架构、SSO 是生产化方向，但不是当前最小闭环的核心。

当前已经有 JWT 用户、草稿 owner、高风险确认和审计，足够支撑 demo 的安全边界。

Q50：如果模型回答错了怎么办？

项目有几层兜底：

- RAG 只基于检索证据回答。
- 输出必须是结构化 JSON。
- JSON 失败会 repair 或 fallback。
- 无证据时拒答。
- citations 和 retrieve_topk 可追溯。
- 错误原因写入 `failure_reason`。
- 后续可通过评测集和 CI 发现回归。

14. 可以主动讲的亮点

亮点 1：默认策略不是质量最高，而是综合最优

Hybrid RRF + Rerank 指标最好，但 CPU 延迟高，所以默认选择 Hybrid RRF。

这体现了工程取舍：

> *不是一味追求指标，而是在实时性、成本和质量之间选可上线方案。*

亮点 2：缓存没有破坏审计

很多缓存实现只追求快，但会丢失请求历史。我这里缓存命中也落库，既快又保留追溯。

亮点 3：Agent 有安全边界

模型不直接执行数据库操作，而是走 Planner -> Validator -> Executor。

亮点 4：MCP 和 Web 共用底层工具

这说明工具层抽象比较干净，不是为每个入口重复写业务逻辑。

亮点 5：项目真的部署到了云服务器

能讲清楚：

- 小服务器资源约束。
- 为什么用 bge-small。
- 为什么用 swap。
- 为什么只开放 8501。
- 为什么数据目录要 volume 挂载。

15. 面试时推荐的 2 分钟介绍

可以这样说：

> *我这个项目是一个企业 IT 服务台 RAG + Agent 系统。最开始只是政策知识库问答，后来我把它扩展成能办事的系统：用户可以问制度，也可以一句话提交 IT 工单，系统会自动抽取字段，缺信息时生成草稿，补充后继续建单。对于已有工单，还支持查单、评论、催办和取消，取消这类高风险动作要二次确认。*
>
> *RAG 侧我做了 Dense + BM25 + RRF 混合检索，并用 130 条问答评测集比较了 Dense、Hybrid RRF 和 Rerank。最终默认选 Hybrid RRF，因为 Top5 召回从 90% 提升到 96.15%，但 p50 延迟只增加约 5ms。Agent 侧我没有让 LLM 直接操作数据库，而是让它输出 ToolPlan，再由后端做 schema 校验、工具白名单、权限校验和确认态。*
>
> *工程化方面，我做了 FastAPI、PostgreSQL、Redis、Qdrant、Docker Compose，补了登录、审计追溯、Redis 问答缓存、运行监控和 token/cost 统计。项目已经在 2 核 4G 云服务器上部署，能通过网页完整演示问答、建单、追溯和监控。*

16. 面试时不要夸大的点

不要说：

- “这是生产级系统。”
- “微调了大模型并大幅提升。”
- “完全解决了幻觉。”
- “支持企业级权限系统。”
- “MCP 已经是多用户生产级。”

应该说：

- “这是一个完整工程 demo，覆盖上线前关键链路。”
- “通过 RAG 证据、JSON 校验、拒答、审计和评测降低幻觉风险。”
- “权限是 demo 级 JWT，生产可扩展 RBAC/SSO。”
- “MCP 目前展示工具暴露和复用，生产还要补身份映射。”

17. 最后记忆口诀

项目主线：

```text
先问答
再评测
再建单
再工具
再确认
再记忆
再 MCP
再登录
再缓存
再监控
```

再部署
```

核心价值:

```text
LLM 负责理解和计划
系统负责校验和执行
所有动作可追溯
所有质量可评测
所有部署有取舍
```
```