Policy KB Assistant 个人理解归档

这份文档回答三个问题：
1. 这个项目到底从哪里开始，为什么一步步变复杂。
2. 每次改造解决了什么真实工程问题。
3. 面试前应该怎样把这些模块串成一条清楚的成长线。

1. 一句话总览

这个项目最初只是一个企业制度知识库问答 demo，后来逐步演进成了一个面向企业 IT 服务台的 RAG + Agent 工单系统：

政策 PDF
-> 文档解析与切块
-> Dense / BM25 / RRF 混合检索
-> LLM 结构化回答
-> 自然语言 Agent 路由
-> 工单创建/查单/评论/催办/取消
-> 多轮记忆、审计追溯、MCP 工具服务
-> 登录鉴权、Redis 缓存、监控成本统计
-> Docker Compose 云部署

它的核心不是“套壳调模型”，而是把 LLM 放进一个受约束的业务系统里：模型可以理解语言、提出计划，但真正执行动作必须经过后端的 schema 校验、权限校验、工具白名单、高风险确认和审计落库。

2. 项目的阶段演进

阶段 0：单问答原型

最开始的目标很简单：用户问一句制度问题，系统能从本地政策文档里找证据，然后让大模型生成答案。

这一阶段的重点是跑通最小 RAG：

- PDF 文档读取。
- 文本切块。
- embedding 向量化。
- Qdrant 存储和召回。
- LLM 基于 top-k 片段回答。
- 返回 answer 和 citations。

这个阶段解决的问题是“能不能问起来”。它的局限也明显：

- 固定窗口切块容易打断制度条款。
- 纯向量检索对企业内部专有名词、系统名、编号不稳定。
- 答案格式不稳定，引用和拒答不可控。
- 没有业务动作，只是问答。

阶段 1：RAG 质量优化

当单问答能跑以后，项目开始进入质量评测阶段。这里做了几件关键事情：

- 构建 130 条制度问答评测集。
- 比较 Dense、Hybrid RRF、Hybrid RRF + Rerank。
- 设计 structured_hybrid 切块策略，保留“章-条-款”层级。
- 引入 GoldDoc Recall@K、MRR、APC、Citation Output、Refusal 等指标。
- 把质量评测接入 CI/脚本，避免只凭肉眼感觉调参。

这一阶段最大的价值是：RAG 不再是“感觉还行”，而是有了量化指标。

关键结果：

策略	效果
Dense -> Hybrid RRF	Top5 召回从 90.00% 提升到 96.15%，MRR 从 0.7955 提升到 0.8574
Hybrid RRF -> Rerank	质量继续提升，但 CPU 下延迟显著增加
fixed chunk -> structured_hybrid	自动答案要点覆盖率、引用输出率提升，拒答率下降

最后项目默认选择 Hybrid RRF，因为它在质量和延迟之间更适合实时服务；Rerank 更适合高质量模式或离线评测。

阶段 2：从问答到工单

接下来项目不再只回答问题，而是开始处理 IT 服务台场景：

- 一句话创建工单。
- 抽取标题、描述、类别、优先级、地点、联系方式。
- 字段完整时直接建单。
- 字段缺失时生成草稿，等待用户补充。
- 工单写入数据库。
- 问答和工单通过 `request_id` / `ticket_id` 关联。

这一阶段的核心变化是：系统从“信息查询”变成“能办事”。

技术上引入了：

- FastAPI 路由。
- SQLAlchemy ORM。
- PostgreSQL / SQLite 兼容运行。
- tickets、ticket_drafts、audit_logs、kb_queries 等表。
- Streamlit UI 调 HTTP API，而不是直接调用 Python 函数。

阶段 3：审计追溯与可回放

业务系统不能只有结果，还要能解释过程。因此项目加入审计追溯：

- 每次 ASK 写入 `kb_queries`。
- 每个 Agent 路由和工具动作写入 `audit_logs`。
- 工单 context 记录关联的 `kb_request_id`。
- UI 支持按工单号回放工单详情、关联问答和审计时间线。

这一步让项目从 demo 更接近企业系统。面试时可以强调：

> *我没有只关注模型回答，而是把每次模型决策、工具调用和业务动作都落到了可审计记录里，这样出现误操作或回答争议时可以回放链路。*

阶段 4：Agent 工具化

最初的 Agent 只是规则判断：像“报修”“查单”这类关键词触发固定分支。后来项目进一步抽象了工具能力：

- `lookup_ticket`
- `add_ticket_comment`
- `escalate_ticket`
- `cancel_ticket`
- `continue_ticket_draft`
- `kb_answer`
- `create_ticket`

工具不只是函数，而是有统一契约：

- 工具名。
- 参数 schema。
- 是否需要确认。
- 允许的 planner scope。
- 审计动作类型。
- 执行函数。

这一步的意义是：让 `/agent`、LLM Planner、MCP tools 可以共享同一套工具定义，减少重复实现。

阶段 5：Planner、Validator 与高风险确认

如果让 LLM 直接操作数据库，风险很高。所以项目采用了“模型提议，系统执行”的架构：

```text
用户输入
-> Planner 生成 ToolPlan
-> Pydantic 校验参数
-> 工具白名单过滤
-> 权限/所有权校验
-> 高风险动作二次确认
-> 服务层执行
-> 审计落库
```

取消工单是典型高风险动作：

1. 用户说“帮我取消这个工单”。
2. 系统不直接取消，而是返回 `NEED_CONFIRMATION` 和 `confirm_token`。
3. 用户再次确认并带回 token。

4. 后端校验 token、用户、有效期、状态。
5. 才执行取消并写审计。

这体现了 Agent 工程化的一个关键原则：

> *LLM 可以理解意图，但不能拥有最终执行权。*

阶段 6：LangGraph 编排

为了让 Agent 流程更清晰，项目引入了 LangGraph。

LangGraph 主要承担：

- 状态对象统一传递。
- 节点拆分：路由、ASK、工具调用、记忆更新等。
- 条件边控制流程走向。
- 工作记忆保存单次请求里的临时状态。

这里不是为了“强行上框架”，而是因为 Agent 流程开始有多个分支、多种状态和多步动作。用图式编排比在一个函数里写大量 if/else 更容易解释和维护。

项目同时保留 legacy/hybrid 路径，便于灰度切换和测试。

阶段 7：多层记忆

项目逐步补了 L0-L4 记忆边界：

层级	名称	在项目中的作用
L0	Working Memory	单次请求内保存 route、抽取字段、工具结果、confirm_token 等
L1	Session Memory	保存当前会话的最近轮次、未完成目标、会话摘要
L2	Task Memory	工单草稿、待确认动作、任务状态
L3	User Memory	默认地点、联系方式、部门等低风险偏好
L4	Episodic Memory	从问答、工单、审计中回放历史事件

典型能力：

- “继续刚才那个工单”
- “上一单怎么样了”
- 草稿过期或不存在时给出明确响应
- 用户默认地点/联系方式可被低风险复用

这里要注意面试表达：不是说项目实现了“人类级长期记忆”，而是实现了业务场景里可控、可审计、可过期的分层记忆。

阶段 8：MCP 工具服务

项目后来把工单工具能力暴露为 MCP 服务，包括 stdio 和 HTTP 两种形态。

MCP 的价值不是“又写一套工具”，而是：

- Web `/agent` 和 MCP 调用复用同一套服务层。
- 工具动作仍然走权限、确认和审计。
- 外部 Host/IDE/Agent 可以用统一协议调用受约束工具。

这可以讲成：

> *我把业务动作抽象成统一工具层，然后让 Web Agent 和 MCP 工具服务共享底层 executor，避免两套入口出现行为不一致。*

阶段 9：登录鉴权与用户边界

项目从早期 API Key 演示，升级为登录/注册 + JWT Bearer Token：

- `/auth/register`
- `/auth/login`
- `/auth/me`
- UI 登录后保存 token。
- `/agent`、`/ask`、写工单接口都需要当前用户。
- 草稿、确认 token、工单操作绑定用户身份。

这一步解决的是多用户 demo 的基本边界：

- 不是所有人共用一个 actor。
- 用户 A 不能继续用户 B 的草稿。
- 审计能记录真实 actor。

阶段 10：Redis 缓存、监控与成本统计

项目最后补了两个很适合面试讲工程取舍的能力。

Redis ASK 缓存

非流式 ASK 路径接入 Redis 结果缓存：

- 规则 ASK 和 LangGraph ASK 共用 `ask_pipeline.run_cached_ask_steps()`。
- 精确标准化问题 + department + 配置指纹生成 key。
- 只缓存成功且无 failure_reason 的结构化答案。
- 缓存命中时仍然生成新的 request_id，并写入 `kb_queries` 和 `audit_logs`。
- Redis 异常 fail-open，不影响主链路。

本地实测：

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

这个结果非常适合讲：

> *静态制度问答存在大量重复问题，用 Redis 缓存可以减少模型调用成本和延迟，但必须保证审计链路不丢，所以命中缓存也会落库和生成新的 request_id。*

监控与成本统计

项目新增 `/ops/metrics` 和 UI 的“运行监控”页，统计：

- ASK 请求量、成功数、失败数。
- JSON 有效率。
- citation 输出率。
- repair/fallback 比例。
- 平均延迟、p50/p95。
- token 总量和估算成本。
- 工单审计事件、失败/拒绝事件、route/action 分布。

这里不是完整 APM，而是项目级轻量 observability，用于展示“上线后怎么知道系统跑得怎么样”。

阶段 11：Docker Compose 与云部署

最终项目用 Docker Compose 拉起：

- API
- UI
- Postgres
- Redis
- Qdrant
- MCP
- kb-init 一次性知识库初始化任务

云部署踩过的关键点：

- 2C4G 小服务器要加 swap。
- 国内云拉 Docker Hub、GitHub、HuggingFace 可能慢，需要镜像源或上传文件。
- `kb-init` 容器看的是容器内 `/app/data/raw`，宿主机上传 PDF 后需要 volume 挂载。
- 安全组只开放 22 和 8501，不暴露 Postgres/Redis/Qdrant/API/MCP。
- `kb-init` 成功后 API/UI 才会启动。

这一步让项目从“本地能跑”变成“可被公网访问的演示系统”。

3. 当前系统运行链路

3.1 问答链路

```text
UI 登录
-> POST /agent 或 /ask
-> JWT 当前用户
```

-> Redis cache lookup
-> Qdrant Dense 检索
-> BM25 关键词检索
-> RRF 融合
-> 可选 rerank
-> LLM 结构化 JSON 答案
-> JSON 校验/修复/fallback
-> 写 kb_queries
-> 写 audit_logs
-> 返回 answer/citations/meta
```

### 3.2 建单链路

```text
用户一句话描述故障
-> /agent
-> 当前用户与限流
-> 规则路由或 Planner
-> 工单字段抽取
-> 字段完整：create_ticket_workflow
-> 字段缺失：ticket_drafts 草稿
-> 写 tickets / ticket_drafts
-> 写 audit_logs
-> 返回 route/ticket/extraction
```

3.3 工单工具链路

```text
用户自然语言操作已有工单
-> Agent 识别工具意图
-> ToolPlan
-> Validator
-> service 层执行
-> 高风险动作 confirm_token
-> 审计落库
```

3.4 MCP 链路

```text
MCP Host
-> MCP HTTP / stdio server
-> mcp_wrapper executor
-> services 工具层
-> DB / audit
-> 返回工具结果
```
```

4. 我在项目里真正掌握的知识点

RAG

- 为什么纯向量检索不够。
- BM25 对内部名词、编号、系统名称更稳。
- RRF 是低成本融合策略。
- Rerank 提升质量但 CPU 延迟高。
- 切块策略对引用和答案质量影响很大。
- 评测指标要区分检索命中、答案覆盖、引用输出和拒答率。

Agent

- Agent 不是让 LLM 直接执行，而是让 LLM 生成受约束计划。
- 工具必须有 schema。
- 工具执行必须有白名单。
- 高风险动作必须二次确认。
- 所有动作都要可审计。
- 规则路由和 LLM Planner 可以混合使用。

后端工程

- FastAPI 负责 API 边界。
- Pydantic 负责输入输出 schema。
- SQLAlchemy ORM 管理业务数据。
- Alembic 管理数据库迁移。
- Redis 用于限流、幂等、缓存。
- Docker Compose 负责交付环境。

可观测与成本

- 每次 ASK 要记录延迟、模型、JSON 状态、失败原因。
- token 和 cost 即使是估算，也能用于趋势观察。
- 缓存命中不能破坏审计链路。
- 监控页对 demo 很有价值，生产要加权限控制。

部署

- 小服务器优先使用 bge-small。
- 无 GPU 也可以做检索，embedding 和 Qdrant 都能 CPU 跑，只是慢。
- 不要把数据库、中间件端口暴露公网。
- 数据文件要么 build 前进入镜像，要么通过 volume 挂载。

5. 项目里的关键文件地图

文件/目录	作用
`src/kb/ingest.py`	PDF 扫描、切块、embedding、写入 Qdrant

`src/kb/retrieve.py`	Dense/BM25/RRF/Rerank 检索
`src/kb/answer.py`	LLM 回答、JSON 修复、fallback、token/cost meta
`src/api/ask_pipeline.py`	ASK 统一流水线、缓存、公共响应
`src/api/ask_cache.py`	Redis ASK 结果缓存
`src/api/services.py`	核心业务服务层：ASK、Agent、工单、草稿、确认
`src/api/skills.py`	Agent 工具 registry
`src/agent_graph/`	LangGraph 编排与工作记忆
`src/mcp_wrapper/`	MCP 工具执行器
`src/api/ops_metrics.py`	运行监控聚合
`src/ui/app.py`	Streamlit 演示页面
`alembic/versions/`	数据库迁移
`compose.yaml`	部署版服务编排

6. 我应该怎样讲这个项目

推荐讲法：

> *我做的是一个企业 IT 服务台 RAG + Agent 系统。最开始只是政策问答，后来我发现单纯问答不能体现业务闭环，于是加入了工单创建、草稿续办、查单、催办和取消确认。为了让 Agent 更安全，我把模型限制在 Planner 角色，真正执行前要经过 schema 校验、工具白名单、用户权限和高风险二次确认。RAG 侧我做了 Dense + BM25 + RRF 的混合检索，并用 130 条评测集比较不同策略。最后我补了 Redis 问答缓存、运行监控、token/cost 统计和 Docker Compose 云部署，让它从 demo 变成一个可运行、可观测、能解释成本和性能取舍的项目。*

7. 项目边界与不足

这个项目已经适合作为求职项目，但不是生产系统。当前边界包括：

- UI 是 Streamlit，不是正式企业前端。
- 权限是 demo 级用户和角色，不是完整 RBAC。
- 没有接真实企业工单系统，只是 ITSM-lite。
- 监控是轻量聚合，不是 Prometheus/Grafana 级 APM。
- MCP 身份映射仍偏 demo。
- 云部署是单机 Docker Compose，不是 Kubernetes。

这些不足不需要隐藏，面试时可以讲成取舍：

> *我把重点放在 RAG/Agent 工程链路和可演示闭环上，没有过早引入 K8s、复杂 RBAC 或企业 SSO。因为项目目标是证明我能把 LLM 应用从检索、Agent、工具、安全、缓存、监控到部署完整跑通。*

8. 下一步如果继续扩展

如果这个项目继续做，优先级应该是：

1. 接入正式 RBAC，管理员才能看监控。
2. 增加 Prometheus/Grafana 或 OpenTelemetry。
3. MCP 多用户 token 映射。
4. 引入真实工单系统 webhook 或 API。
5. 做独立 RAG 质量测试工具，把当前评测能力产品化。

当前阶段不建议继续无限加功能。更好的策略是：部署稳定、录屏、写简历和面试话术，然后开一个小而锋利的新项目，比如 RAG 质量评测工具。

优先级判断

方向	概念	项目现状	两天推荐度
监控/成本统计	记录延迟、成功率、JSON 修复率、token/cost	已有 ASK 延迟落库，但没有汇总面板和 token 成本	强烈推荐
Redis 问答缓存	静态政策问答命中缓存，少调模型	只有模型/BM25 进程缓存，没有结果缓存	强烈推荐
超时降级	LLM 超时后返回规则/检索摘要/证据不足	有 timeout、fallback model，但策略可讲述性还不够	推荐
JSON 稳定性	schema 校验、修复、拒答兜底	已经做得不错，不宜重复造	补文档/测试即可
路由策略	简单请求走规则，复杂请求走 Planner	已有 rules/llm/hybrid	推荐整理成亮点
提示词精简	减 token、降成本、降漂移	可以做，但收益不如监控缓存明显	中等推荐
模糊输入理解	“上一单”“刚才那个”等引用恢复	项目已有记忆和 Planner 规则	中等推荐
消息队列	异步处理日志、通知、评测等非主链路	当前不需要，DB 审计够用	不推荐周末做
多 Agent 协作	多角色 Agent 分工通信	对当前项目偏炫技，风险大	不推荐
K8s	容器编排、伸缩、滚动发布	你已有 Docker Compose，求职够讲	不推荐周末做
SaaS	多租户、账号、套餐、计费、线上可用服务	项目是 demo 级安全模型	不推荐硬做