1. 这个项目解决什么问题?

这个项目是一个面向企业内部 IT 服务台的 RAG + Agent 系统。用户可以用自然语言问制度、报障建单、查单、评论、催办和取消。系统的核心设计思路是把 LLM 限制在理解、路由和计划生成上，真正的业务执行由后端服务层完成，并通过 schema 校验、权限校验、幂等控制、高风险确认和审计日志保证安全可控。

RAG 部分我做了 Dense + BM25 + RRF 的混合检索，并用实验对比了 Dense、Hybrid RRF 和 Hybrid RRF + Rerank 的效果与延迟，最终选择 Hybrid RRF 作为实时默认模式。Agent 部分用 LangGraph 把记忆加载、引用恢复、规则路由、Planner、工具校验、业务执行和 finalize 串起来。工单部分支持缺字段草稿续办和 confirm_token 二次确认。工程上用 Docker Compose 拉起 API、UI、Postgres、Redis、Qdrant 和 MCP，并通过 CI 做确定性测试和轻量质量门禁。

当前不足是业务系统还没有接入真实 ITSM，自动评测还不能替代人工准确率，长尾 Agent 场景还需要更多测试。后续我会优先接真实工单系统、扩充 workflow eval、增强权限过滤和可观测性，而不是盲目堆更多 Agent 概念。

2. 请求主链路怎么走?

Streamlit 聊天页（Streamlit 是纯 Python 写网页的库，不需要你懂 HTML、CSS 或 JavaScript）
-> _render_chat_page()
-> _chat_execution_path(mode, text)

问答路径:
-> PolicyAPIClient.ask_stream_events()
-> POST /ask/stream
-> JWT 鉴权
-> run_ask_workflow_stream_async()
-> retrieve_async()
-> LLM 流式生成
-> 写 kb_queries / audit_logs
-> SSE 返回 token/final
-> 前端流式展示答案和 citations

工单路径:
-> PolicyAPIClient.agent()
-> POST /agent
-> JWT 鉴权
-> Redis 限流
-> run_agent_workflow 或 run_agent_graph
-> 记忆加载 / 引用恢复
-> rules / planner / validator
-> ticket service 或确认态
-> 写 tickets / drafts / comments / pending_actions / audit_logs
-> 返回 AgentResponse
-> 前端展示工单结果

前置步骤：登录

Streamlit 页面
-> _render_auth_page()
-> PolicyAPIClient.login()
-> PolicyAPIClient._request()
-> POST /auth/login
-> routes/auth.py::login()
-> crud.get_user_by_login_identifier()

-> verify_password()
-> create_access_token()
-> 返回 AuthTokenResponse
-> 前端写入 st.session_state
-> st.rerun()
-> 进入登录后的 Agent 页面

登录详细

POST /auth/login
-> login(payload, request, response, db, redis)
-> 提取 identifier 和 client_ip
-> consume_rate_limit(scope="auth:login:ip", subject=ip)
    -> Redis INCR 固定窗口计数
    -> 超限返回 429
-> consume_rate_limit(scope="auth:login:principal", subject=ip+identifier)
    -> Redis INCR 固定窗口计数
    -> 超限返回 429
-> crud.get_user_by_login_identifier()
-> verify_password()
-> 检查 user.is_active
-> crud.update_user_last_login()
-> _build_auth_response()
-> 返回 AuthTokenResponse
-> 前端保存 token/profile
-> st.rerun()
-> 进入聊天页

前置步骤：注册

POST /auth/register
-> register(payload, db)
-> 提取 username/email/phone
-> 检查 username/email/phone 是否重复
-> hash_password(password)
    -> PBKDF2-HMAC-SHA256
    -> 16 字节随机 salt
    -> 310000 次迭代
    -> 生成 password_hash
-> crud.create_user()
-> crud.update_user_last_login()
-> _build_auth_response()
    -> create_access_token()
    -> 生成 JWT access token
    -> 序列化 user profile
-> 返回 AuthTokenResponse
-> 前端保存 auth_access_token 和 auth_user_profile
-> st.rerun()
-> 进入聊天页

前端页面：

→页面未登录，进入登录页面，_render_auth_page 函数渲染登录/注册页

→登录表单使用 st.form 包裹，以避免每次输入都触发脚本重跑，只有在点击提交按钮时才执行内部逻辑。

→点击提交按钮，先前端检验用户名和密码不能为空，然后触发 client.login 函数（函数内发出"POST" "/auth/login"请求），或者触发 client.registe 函数，（函数内发出 "POST", "/auth/register"请求）

→这里进入后端，返回 ExecutionState

→接收上一级函数返回 AuthTokenResponse（返回体）格式的数据

→获取 access Token 和 User 信息，然后存入 st.session_state（前端持久化数据结构）

后端路由和路由函数：

注册路径：

→FastAPI 路由到 POST "/auth/ register"函数（会做请求体+返回体 schema 校验，传入信息提取和 json 构建，FastAPI 做的事）

→register 函数传入参数：（AuthRegisterRequest（请求体）、 db Session（依赖注入：传入参数时执行函数获取返回结果作为参数））

→register 函数返回参数：AuthTokenResponse（返回体）

→先从 payload 中提取 username、email、phone，然后查询数据库中是否已存在重复数据，返回 409

→如果未重复，就调用 hash_password 函数编码用户输入的密码，得到写入数据库的密码

→hash_password 函数传入参数（password），校验非空，然后生成随机盐（使用 Python 标准库 secrets 模块，**16 字节（128 位）**），

→调用 hash 算法（*通常是 sha256*）生成哈希摘要 digest（字节串）

→**编码 salt 和 digest 为文本格式 ASCII 字符串，**拼接然后返回结果

→获取返回的结果，作为存入数据库的密码，连同 username、email、phone 等信息写入数据库用户表

→调用_build_auth_response 函数，构建最终返回数据

→调用 create_access_token 函数，签发 access Token（基于 JWT 方法，将一个 payload json 格式数据编码成一个字符串）

→生成 AuthTokenResponse（返回体）格式的数据（access_token，expires_in，user（User 表格式））

→返回 AuthTokenResponse（返回体）格式的数据

→直接将上一级函数返回 AuthTokenResponse（返回体）格式的数据

登录路径：

→FastAPI 路由到 POST "/auth/login"函数（会做请求体+返回体 schema 校验，传入信息提取和 json 构建，FastAPI 做的事）

→login 函数传入参数：（AuthLoginRequest（请求体）、 Request（请求对象，主要是为了获取客户端 IP（用于限流））、 Response（原始响应对象，用来在返回前设置自定义响应头（比如限流信息））、 后面是（依赖注入：传入参数时执行函数获取返回结果作为参数）db Session（数据库会话）、）

→login 函数返回参数：AuthTokenResponse（返回体）

→先从 payload 中提取 identifier（登录标识，可以是 username / email / phone），从 request 中提取 ip 信息（用于 ip 限流）

→调用 consume_rate_limit 函数进行限流判定

→传入参数：redis（Redis 客户端）、scope（"auth:login:ip"，表示这是登录 IP 限流）、subject（"ip:客户端 IP"，表示按 IP 维度计数）、limit（窗口内最大请求次数）、window_seconds（窗口秒数）

→构造限流 key，然后读取 Redis 客户端信息，判定是否达到窗口限流上限，达到就计算 retry_after 值，否则当前幂等键的限流+1

→返回参数：判定结果，json 格式（allowed（布尔值）、limit（总限流次数）、current（当前次数）、remaining（剩余次数）、retry_after_seconds（剩余窗口时长）、window_seconds（总时长）、redis_key（幂等键））

→如果 IP 幂等判定就返回：429 重试过与频繁

→ip 幂等允许就进行账号幂等判定，不允许就返回 429

→更新用户最新登录时间，然后返回 AuthTokenResponse（返回体，（access_token），expires_in，User）给

前端

第一步：Agent 路线请求处理流程

Streamlit 编写前端页面：

→页面被点击（run_agent = st.button("调用 /agent", ...)）
→先进行登录校验：
→ **渲染（画按钮）**未登录不可点击，**返回状态（检测点击）：**函数会返回一个布尔值 True 或 False，表示“在脚本这次运行中，这个按钮是不是刚刚被用户点击的”
→st.session_state 是一个字典（生命周期与**用户的浏览器会话**绑定，用户关闭浏览器或长时间无活动后 session 会失效），保存持久化信息（
只有两个键：
→auth_access_token：JWT 或类似的访问令牌（字符串）
→auth_user_profile：用户信息字典，包含 username、department 等字段））
→调用 Streamlit 内部的_handle_agent_submit 函数（传入 client, text, user_name, department 参数）
→调用后端 agent 接口，然后运行结果（也就是一路返回来的 ExecutionState）传入 result

→这里进入后端，返回 ExecutionState

→然后把 ExecutionState 的字段提出来，更新进前端的 st.session_state 里面，然后再处理并选择性返回结果给前端
→client 是自定义类型 PolicyAPIClient（一个 API 客户端对象，有一些方法 agent/ask/create_ticket/...）
→client.agent(text=..., user=..., department=...)方法向后端发送 POST 请求

后端路由和路由函数：

→FastAPI 路由到 POST "/agent"函数（会做请求体+返回体 schema 校验，传入信息提取和 json 构建，FastAPI 做的事）
→agent 函数传入参数：（AgentRequest（请求体）、 Request（请求对象，主要是为了获取客户端 IP（用于限流））、 Response（原始响应对象，用来在返回前设置自定义响应头（比如限流信息））、 后面是（依赖注入：传入参数时执行函数获取返回结果作为参数）db Session（数据库会话）、current_user（当前登录用户信息）、redis（Redis 连接））
→agent 函数返回参数：（AgentResponse，返回体）
→先进行限流检查函数（传入参数：redis、scope（用于区分不同的限流策略）、subject（限流对象标识，user+ip）、limit（最大请求数）、window_seconds（窗口大小））
→返回参数：RateLimitDecision 对象（自定义类，属性：allowed（是否允许限流）、limit、current、remaining、retry_after_seconds、window_seconds、redis_key）
调用 run_agent_graph/run_agent_workflow（视 mode 而定）函数
→run_agent_graph 函数传入参数：db 会话对象、text（用户输入的一句话描述）、user、department、draft_id、fields（补全的额外字段）、confirm_token、actor_user_id、actor_role
函数功能：
→调用_build_initial_state，传入参数（text（用户输入的一句话描述）、user、department、draft_id、fields（补全的额外字段）、confirm_token、actor_user_id、actor_role）
→返回参数：（request（json 对象）（text、user、department、actor_user_id、actor_role、draft_id、confirm_token、fields）、 planner、memory、working、draft、confirm、execution、error、audit、meta（元数据，started_at、finished、graph_version、engine））
→然后 run_graph 直接执行图到底，返回 final_state（就是 State 所有字段）

→这里进入图构建和运行步骤

→返回 response（将 final_state 的 execution（**是一个结构体（字典），真正要返回给前端的业务结果**）提取出来，然后字典化），

→返回调用 normalize_graph_response（**确保响应里一定有 route 字段，没有就返回错误**）函数的结果 execution（route（Agent 最终走的业务路径），message（给前端展示的提示信息）、missing_fields（建单信息不完整，这里放缺失字段）、kb（问答结果，包括答案、引用、检索信息、request_id）、extraction（建单字段抽取结果）、ticket（新建工单后的简要信息，比如工单号、状态）、ticket_detail（查询已有工单时的完整工单详情）、draft（工单草稿信息）、tool_result（MCP/tool 执行结果摘要））

→函数的结果 execution 一路向前返回给前端

第二步：Agent 内部的 LangGraph 流程

→run_graph 函数输入参数：（db 会话，AgentState（集合了各种子 State（RequestContext、PlannerState、MemoryState、WorkingMemoryState、TicketDraftState、ConfirmState、ExecutionState、AskState、ErrorState、AuditState、MetaState，方便清晰查看和分析、维护）））

State（核心）：

request：用户原始输入和身份信息
planner：路由结果 / 工具计划
memory：短期记忆和长期记忆
working：本轮过程记录，用于审计和调试
execution：最终返回给前端的结果
audit：节点执行事件
meta：图执行元信息

→添加节点（NODE_PREPARE（前置准备，**标准化输入**）、NODE_RESOLVE_REFERENCES（**解析工单/草稿引用**）、NODE_LOAD_MEMORY（记忆加载）、NODE_REFERENCE_CLARIFY（需要工单号但是没解析出来，询问用户）、NODE_RULES_ROUTER（规则路由）、NODE_ASK（问答）、NODE_TICKET_CREATE（建单）、NODE_DRAFT（草稿）、NODE_CONFIRM（确认）、NODE_TICKET_TOOL（工单工具（催单、评论、取消、处理））、NODE_GLOBAL_PLAN（全局计划）、NODE_GLOBAL_VALIDATE（全局验证）、NODE_EXECUTE_MCP_TOOL（执行 MCP 工具）、NODE_FINALIZE（最终处理））

→添加直连边（START→NODE_PREPARE→NODE_LOAD_MEMORY→NODE_RESOLVE_REFERENCES）

→条件边

```
|
→NODE_REFERENCE_CLARIFY，需要补充工单号但没解析出来
|
→NODE_RULES_ROUTER，如果有 confirm_token → 规则路由（
|                                          |
|                                          →有确认 token -> confirm
|                                          有草稿 -> draft
|                                          有工单号 -> ticket_tool
|                                          像报修/故障/无法/异常 -> ticket_create
|                                          否则 -> ask）
|
→NODE_GLOBAL_PLAN，否则进入全局规划
    |
    →NODE_RULES_ROUTER，规则路由
    |
    →NODE_GLOBAL_VALIDATE，全局路由参数验证
    |                          |→NODE_RULES_ROUTER，规则路由
    |                          |→NODE_EXECUTE_MCP_TOOL，执行工具
    |                          |→NODE_FINALIZE，最终结果
    |
```

→NODE_FINALIZE，最终处理

哪些节点直接调用 service，哪些节点调用 MCP

Adapters 是将 service 层工作流中的逻辑，拆分为工具函数方便节点调用（然后这些节点调用 adapter 函数间接调用 service）

节点	文件	实现方式	调用对象
prepare	nodes_prepare.py	不调业务服务，只整理 state 简单逻辑不调用	无
load_memory	memory.py	通过 adapters 读记忆（MCP 包装），调 service 工具	services._load_short_term_memory / _load_user_memory
resolve_references	nodes_routing.py	通过 adapters 解析显式与隐式 ticket/draft 调 service 工具	services._extract_ticket_public_id 等
reference_clarify	nodes_routing.py	缺失字段，构造追问响应 调 service 工具	services._build_missing_ticket_reference_response
rules_router	nodes_routing.py	规则判断进入哪个业务节点 简单逻辑不调用	services._should_route_to_ticket 等
ask	nodes_ask.py	直接走 RAG pipeline（自己实现逻辑，未调用 service）	ask_pipeline.run_retrieve_step / run_answer_step
ticket_create	nodes_ticket_create.py	直接调用 service	services._handle_create_ticket_intent
draft	nodes_draft.py	直接调用 service	services._resume_ticket_draft_workflow
ticket_tool	nodes_ticket_tool.py	直接调用 service	services._handle_ticket_tool_route
confirm	nodes_confirm.py	直接调用 service	services._handle_confirmed_pending_action
global_plan	nodes_global.py	调 Planner，不执行工具 调 service 工具	planner.run_global_planner
global_validate	nodes_global.py	调 service 工具	services._validate_global_plan
execute_mcp_tool	nodes_execute_mcp.py	真正调用 MCP wrapper / 远程 MCP	invoke_tool 或 call_remote_tool
finalize	nodes_finalize.py	更新记忆、审计、兜底响应 简单逻辑不调用	services._update_short_term_memory_from_response

rules 路径：基本直接调 services，不走 MCP。
global planner 路径：global_plan + global_validate 后，execute_mcp_tool 才走 MCP。

MCP 调用原理：

入口：run_execute_mcp_tool_node 函数传入参数（）
→读状态和 planner_state，获取 request（请求信息（User-id、原始文本、用户名、部门、role））、tool_request（工具名、参数、请求 id、模式）、original_args（参数）
→判定工具名合法性，然后调用_inject_auto_idempotency_if_needed 对高风险工具（如创建工单、确认操作，具体查询需要幂等键工具名单）自动生成一个基于内容指纹的幂等键，防止重复提交。
→组建参数，上下文信息封装成标准请求对象 ToolCallRequest（tool、args、request_id、actor、actor_user_id、actor_role、department、idempotency_key、mode、raw_text）
→读取 MCP client 状态，可用就调用 call_remote_tool 传入参数（server_url、tool、args、raw_text、timeout_seconds（超时时长）、circuit_breaker_enabled（是否启用熔断器（防止反复调用失败的服务））、circuit_fail_threshold（默认 3，熔断器触发的连续失败次数阈值）、circuit_open_seconds（熔断器打开后拒绝新请求的持续时间（秒））
→MCP 不可用就执行**本地 MCP 工具调用函数**，其作用是在当前进程内（或同一部署环境）直接执行工具逻辑，而不是通过网络调用远程 MCP 服务
→结合 MCP 调用结果，构建 execution 执行结果，写入 State
→调用 update_working_memory 更新记忆（将运行结果更新进记忆，state 和 tool_result_summary）

MCP 准备工作

启动服务：调用 build_mcp_server 函数
→加载环境变量、确保数据库 schema ready、读取 MCP_ACTOR_USER_ID
→构造 MCP server（调用 build_mcp_server 函数，设置参数（启动 HTTP 服务器，监听指定端口和路径（如 http://0.0.0.0:9000/mcp））、通过 streamable-http 启动
→build_mcp_server 里面注册工具名（装饰器）、描述、inputschema，调用对应的函数名实现工具业务（复用 service）
→返回一个 FastMCP 对象（里面有各种工具方法）

MCP Client 的构建与使用

构造 ToolCallRequestToolCallRequest 参数
调用：mcp_wrapper.invoke_tool()（这里调用函数是用自定义工具名，中间要经过 dispatch_tool 函数匹配源函数名，比如（_handle_ask_policy））
→规范化工具名，提取 idempotency_key，校验参数是否合法
→禁止伪造 actor / role / user_id 等敏感字段
→对写操作做幂等检查
→dispatch_tool 找到对应 handler
→handler 调 service 层函数
→把结果包装成统一 ToolCallResult
还有：ToolCallRequest 内部准备参数（MCP wrapper 包装输入）

```
(
    tool="create_ticket",
    args={
        "text": "VPN 连不上，地点 3 楼，手机号 13812345678",
        "fields": {},
        "idempotency_key": "auto-create_ticket-xxx"
    },
    actor="当前绑定用户",
    actor_user_id="当前绑定用户",
    actor_role="user",
    department="IT",
    raw_text="VPN 连不上，地点 3 楼，手机号 13812345678"
)
```

包装输出：

```
{
    "contract_version": "v1",
    "success": True,
    "message": "已创建工单。",
    "data": {
        "route": "CREATE_TICKET",
        "ticket": {
            "ticket_id": "TCK-2026-000001",
            "status": "open"
        },
        "message": "已创建工单。"
    },
    "error": None,
    "route": "CREATE_TICKET",
    "_tool_meta": {
        "raw_tool": "create_ticket",
        "normalized_tool": "create_ticket",
        "deprecated_alias": False
    }
}
```

整理进 state["execution"]

作用：

- 统一输入输出
- 统一工具名
- 统一错误格式
- 统一幂等
- 统一权限上下文
- 禁止参数伪造身份
- 隐藏 DB session
- 把异常转成 PLAN_REJECTED

一个工具被调用后的完整流程

```
Agent execute_mcp_tool 节点
-> call_remote_tool(
        tool="create_ticket",
        args={...}
    )
-> streamablehttp_client 连接 MCP server
-> ClientSession.initialize()
-> session.call_tool("create_ticket", arguments={...})
-> MCP server 找到 @app.tool() 注册的 create_ticket 函数
-> create_ticket() 打开 DB session
-> _invoke_tool()
-> invoke_tool()
-> dispatch_tool()
-> _handle_create_ticket()
-> services._handle_create_ticket_intent()
-> 返回业务 dict
```

-> invoke_tool 包装成 contract
-> MCP server 返回 structuredContent / JSON
-> call_remote_tool 解析成 ToolCallResult
-> execute_mcp_tool 写入 state["execution"]
-> finalize
-> 前端显示

阶段一：调用前准备（Server 侧）

1. **启动 MCP Server**（HTTP 模式）：
   - 读取环境变量 MCP_HOST, MCP_PORT, MCP_ACTOR_USER_ID 等。
   - 初始化数据库连接池，确保 schema 就绪。
   - 创建 FastMCP 实例，注册工具。
   - 启动 HTTP 服务器（uvicorn 风格），在 /mcp 端点接受 POST 请求。
2. **Server 空闲等待**：Server 进程常驻，每次收到请求会动态创建数据库会话（_db_session），处理完即关闭。

阶段二：调用时（Client 侧 + Server 侧）

Client（Agent Graph 节点）：

1. **构造请求**：从 AgentState 中提取 tool, args, raw_text, actor_user_id。
2. **幂等键自动注入**（若工具在 AUTO_IDEMPOTENT_TOOLS 集合中）：
   - 计算基于 tool + actor_user_id + raw_text + args 的哈希。
   - 加上时间窗口桶（默认 60 秒），生成 auto-{tool}-{bucket}-{digest} 作为 idempotency_key。
3. **检查熔断器**：若熔断打开，直接返回 ToolCallResult(ok=False, retryable=True)。
4. **映射工具名**：kb_answer → ask_policy 等。
5. **建立 HTTP 连接**：调用 streamablehttp_client 与 MCP Server 握手。
6. **发送** call_tool：参数为映射后的工具名和远程参数。
7. **解析结果**：_payload_from_call_result 提取 structuredContent 或 text，并标准化为 ToolCallResult。

Server 侧：

1. **接收 HTTP 请求**：MCP SDK 将 JSON-RPC 消息解析为工具调用。
2. **调用对应的工具函数**（如 create_ticket）：
   - 创建独立数据库会话。
   - 调用 _invoke_tool → invoke_tool(db, ToolCallRequest(...))。
3. invoke_tool **内部流程**（本地执行，但通过 MCP Server 封装）：
   - **参数校验**：禁止 department 等顶层键，禁止 is_admin 等参数路径。
   - **幂等性检查**：若工具在 IDEMPOTENT_TOOLS 中且提供了 idempotency_key，则查询 Redis 是否已执行过：若已存在则直接返回缓存结果；若冲突则拒绝。
   - **分发到注册器**：dispatch_tool → TOOL_REGISTRY[normalized_tool]。
   - **执行实际业务逻辑**（例如 _handle_create_ticket 调用 services._handle_create_ticket_intent）。
   - **构造标准响应**：_build_success_contract 包装 data，添加 contract_version, success, _tool_meta 等。
   - **完成幂等记录**：成功后将响应写入 Redis 缓存。
4. **返回结果**：工具函数返回 dict，FastMCP 自动转换为 MCP 协议的 CallToolResult 对象。
5. **关闭数据库会话**。

阶段三：调用后（Client 侧）

- 更新 AgentState["execution"] 存储工具结果。
- 若失败且错误码为 remote_mcp_error，则记录失败到熔断器。
- 记录审计事件 NODE_EXECUTED。
- 更新工作记忆（update_working_memory）。

A. 总体架构类

1. 这个项目一句话是什么？
2. 用户从 UI 输入一句话后，完整链路怎么走？
3. /agent、/ask、/tickets 三个入口有什么区别？
4. 为什么不用一个大 service 函数全写完，而要分 route / service / graph / tool？
5. 这个项目最核心的技术难点是什么？

B. Agent 类

1. 规则路由和 LLM Planner 怎么分工？
2. 为什么高频场景优先规则？
3. Planner 输出错了怎么办？
4. ToolPlan 怎么校验？
5. LLM 为什么不能直接调用数据库？
6. 取消工单为什么要二次确认？
7. “上一单”“刚才那个草稿”怎么恢复？

C. RAG 类

1. 为什么 Dense 不够？
2. BM25 补充了什么？
3. RRF 为什么比简单拼接好？
4. rerank 为什么不默认全开？
5. citation 是怎么生成和校验的？
6. GoldDoc Recall 和答案正确率有什么区别？
7. Auto APC 为什么不能等同于人工准确率？

D. 工单业务类

1. 你的工单 schema 为什么这样设计？
2. 为什么 location/contact 是必填？
3. 缺字段为什么不让 LLM 猜？
4. 草稿如何避免过期和重复提交？
5. 工单状态如何流转？
6. 普通用户能不能取消别人的工单？
7. 前端重复点击提交怎么办？

E. 记忆类

1. L0-L4 分别解决什么？
2. 为什么 L1 不保存完整任务 payload？
3. 为什么 L3 只保存低风险默认资料？
4. 记忆什么时候更新？
5. 错误记忆怎么办？
6. 多轮对话和数据库状态如何对齐？

F. 工程化类

1. Docker Compose 启动了哪些服务？API、UI、Postgres、Redis、Qdrant 和 MCP
2. Redis 在项目里承担什么？
3. Postgres 存什么？
4. Qdrant 存什么？
5. MCP 在项目里解决什么问题？
6. CI 为什么要有质量门禁？
7. 线上部署时最大风险是什么？

项目的不足

不足 1：业务场景还偏模拟
当前 IT 服务台工单是自建 schema 和模拟知识库，还没有接入真实 ITSM 系统，比如 Jira、ServiceNow、飞书审批或企业微信工单。因此它更像一个可运行原型，下一步可以对接真实工单系统 API。
不足 2：RAG 评测还不等于真实人工准确率
当前评测有 GoldDoc Recall、MRR、Auto APC、Citation Output Rate，但这些是自动指标。它们能做质量门禁，不等于人工最终满意度。后续应该增加人工标注、答案事实一致性评估和引用正确性评估。
这个说法和你 README 里对指标边界的说明一致：GoldDoc 是文档级命中，不等于严格证据条款命中；Auto APC 不等同人工最终正确率；Citation Output 不代表引用一定完全正确。
不足 3：Agent 长尾意图还需要更多评测样本
目前 Agent 小样本评测能覆盖核心路径，但长尾表达、含糊指令、多意图混合、恶意输入还不够。后续应该扩充 agent eval set，并加入对工具误调用、越权尝试、确认态绕过的测试。
不足 4：记忆策略还需要更严格的生命周期管理
L0-L4 已经分层，但长期用户记忆还需要更严格的写入策略、用户可见可删除机制、冲突处理和隐私控制。
不足 5：线上可观测性还不够完整
现在有 audit_logs 和 CI，但真正生产环境还需要 metrics、tracing、日志聚合、模型调用成本统计、慢查询分析、告警机制。
不足 6：MCP 当前更多是协议化封装，不是生态级工具平台
MCP 的价值是统一工具描述、调用和返回格式，但当前项目主要封装工单工具。后续可以扩展到邮件、日程、权限系统、资产管理系统等真实企业工具。

未来拓展应该怎么讲?

第一阶段：真实 ITSM 对接
第二阶段：RAG 质量增强
第三阶段：Agent 安全与评测增强
第四阶段：企业工具生态扩展
第五阶段：可观测性与运维化
方向 1：接入真实 ITSM 系统
把 tickets 表从内部模拟工单，升级为外部 ITSM adapter。
支持 Jira / ServiceNow / 飞书 / 企业微信。
本系统保留 Agent、记忆、审计和 RAG，实际工单由外部系统承载。
方向 2：RAG 升级
增加结构化文档解析
按制度版本过滤
加入权限过滤
提升引用片段准确率
加入 query rewrite
加入多轮检索
必要时探索 GraphRAG
GraphRAG 可以说，但不要作为当前核心。你可以说：
GraphRAG 更适合制度之间存在强关系、审批流程依赖复杂、跨文档推理明显的场景。当前项目先用 Hybrid RRF 保证实时性，后续在制度关系稳定后再构建文档实体图和流程图谱。
方向 3：Agent 评测增强
构建更多 workflow eval case
覆盖误路由、越权、重复提交、确认绕过
加入 adversarial prompts
统计 tool success rate / wrong tool rate / confirmation bypass rate
方向 4：多工具扩展

邮件通知
日程预约
资产管理
账号权限系统
VPN 账号开通
打印机资产查询

方向 5：可观测性增强

request tracing
模型调用成本
retrieval latency
tool latency
失败原因聚合
高频问题挖掘
L5 Skill 沉淀

优化项

1. GitHub About 补全

现在 About 显示没有 description、website、topics。
建议 description：

> RAG + Agent assistant for internal IT service desk: policy QA, ITSM-lite tickets, memory, audit logs, MCP tools, and CI eval.

Topics 加：

- rag
- ai-agent
- langgraph
- fastapi
- qdrant
- bm25
- rrf
- mcp
- tool-calling
- docker-compose
- postgresql
- redis
- llmops

2. README 顶部加一个 30 秒 demo GIF 或截图

面试官打开项目第一眼应该看到：

- 登录
- Agent 输入
- 缺字段草稿
- 补充字段
- 创建工单
- 审计追溯

3. 加一个 docs/interview_guide.md

里面放：

- 项目一句话
- 核心架构
- 请求链路
- RAG 设计

Agent 安全设计
工单草稿设计
记忆设计
评测结果
不足与未来

这不是给用户看的，是给面试官看的。

4. 加一个 docs/design_decisions.md

列出关键取舍：

为什么 Hybrid RRF 默认，Rerank 非默认
为什么 LLM 不直接执行工具
为什么缺字段生成草稿
为什么取消需要 confirm_token
为什么 L1 只保存引用
为什么 CI 分 ci-test 和 ci-eval

5. 准备一个 3 分钟演示脚本

脚本就讲一个例子：

我无法登录统一身份认证，帮我提交工单
-> 缺字段，生成草稿
-> 我在图书馆三楼，电话 138xxxx
-> 创建工单
-> 查看审计日志

这个比讲 20 个技术名词有效。

---

9. 你应该如何定位这个项目？

不要把它定位为：

一个很火的开源项目
一个通用 Agent 框架
一个领先业界的多智能体系统

应该定位为：

一个面向国内 AI 应用开发求职的工程型 Agent 项目。

它体现的是：

FastAPI 后端能力
数据库建模能力
RAG 检索增强能力
Agent 编排能力
工具安全执行能力
多轮状态管理能力
Docker 部署能力
CI 评测意识

这正好对齐国内 AI 应用开发 / AI Agent 开发岗位。

项目介绍，技架构设计，技术决策

请用 5-8 分钟介绍一下你的项目。重点讲清楚整体的架构设计，以及在其中做了哪些关键的技术决策。

我做的是一个企业制度问答和 IT 服务台工单助手。它是一个业务 Agent。最重要的目的就是让用户可以用自然语言问制度问题，也可以用自然语言方式完成创建工单、补充草稿、查询工单、催办、追加评论或者取消工单，这样就不用去自己费心考虑工单的每个字段是什么意思，应该怎么填，Agent 自动从用户自然语言描述里面抽取用的上的工单相关信息，然后依据用户登录信息来补充用户信息（项目的价值）。

系统实现的核心目标是：让 LLM 参与理解和规划，但不能让 LLM 直接执行高风险业务动作，所有动作都必须经过后端校验、权限控制和审计。

整体架构上，最外层是 Streamlit 前端，负责登录、自然语言输入、工单管理和追溯展示。后端是 FastAPI，主要暴露 /agent、/ask、/tickets、/audit_logs 等接口。自然语言入口统一走 /agent，路由层做鉴权、限流、Pydantic 请求校验，然后把请求交给 Agent 编排层。

Agent 层有两种执行模式，一种是 legacy service workflow，一种是 LangGraph 编排。LangGraph 里我把流程拆成 prepare、load memory、resolve references、rules/global planner、validate、execute、finalize 等节点。这样做的原因是 Agent 流程里有很多状态，比如当前用户、短期记忆、草稿 ID、上一张工单、planner 输出、确认态 token。如果全部写在一个 service 函数里会越来越难维护，所以我用状态机方式把“路由、校验、执行、收尾”拆开。

业务执行层主要在 service 和 MCP wrapper。比如问答走 RAG 链路，先检索再生成带 citation 的回答；建单会先抽取工单字段，如果缺少地点或联系方式，就生成 ticket draft，等用户补充后再续办；工单工具包括查单、评论、催办、取消。取消属于高风险动作，不会直接执行，而是先写入 pending action，返回 confirm token，用户二次确认后才真正取消。

数据层用 SQLAlchemy ORM，主要表包括 tickets、ticket_comments、ticket_drafts、pending_actions、kb_queries、audit_logs、agent_conversation_memory 和 user_memory。这里我把评论单独做成 append-only 表，而不是放在 ticket 的 JSON 里，是为了避免并发修改时覆盖历史评论。审计日志会记录 ASK、CREATE_TICKET、PLAN_PROPOSED、PLAN_REJECTED、PLAN_EXECUTED、NEED_CONFIRMATION 等事件，方便回放一次请求到底经过了哪些决策。

我个人做的关键技术决策主要有几个。第一个是 RAG 检索策略。我没有只用向量检索，而是支持 dense 向量检索、BM25、RRF 融合和可选 rerank。原因是企业制度里有很多编号、部门名、VPN、IT-03 这类精确词，纯向量检索容易语义相近但漏掉关键编号；BM25 可以补精确匹配，RRF 可以避免不同分数尺度难以直接相加的问题。切块上我对比过 fixed、overlap 和 structured hybrid，最后选择 structured hybrid 加 bge-small 作为默认方案，因为在召回和延迟之间更平衡。

第二个关键决策是把“模型规划”和“后端执行”分离。LLM 只负责判断用户想做什么、选择工具、抽取参数；但它输出的 plan 不可信，必须经过程序侧 validator。比如工具名必须在 registry 白名单里，参数必须通过 Pydantic schema，目标工单必须存在，用户必须是 owner 或 admin，高风险动作必须确认。这样可以避免模型幻觉或 prompt injection 直接造成业务副作用。

第三个决策是引入草稿和记忆。建单时用户经常只说“电脑连不上网，帮我报修”，但缺少地点和联系方式。如果直接失败体验不好，如果直接建单又信息不完整。所以我设计了 ticket draft，保存已抽取字段和 missing fields。短期记忆负责恢复“上一单、刚才那个草稿”，长期记忆只保存相对安全的默认地点和联系方式，用来减少重复输入。

所以这个项目里我的整体设计思路：LLM 负责理解，后端负责约束；工具能力集中注册，执行前强校验；高风险动作二次确认；全链路落库和审计。这样系统既能有自然语言交互的灵活性，也能满足业务系统对安全、可追溯和稳定性的要求。

如果 LLM 在计划里把工具名写错了，或者传了一个不存在的工单 ID，你的 validate 节点具体是怎么拦截的？拦截之后系统是怎么让 LLM 纠正的，还是直接给用户报错？

重点是区分用户问题还是系统的问题

“这个问题正好能说明我为什么要单独设计一个 validate 节点。

LLM 生成的工具计划一共有三类典型错误：工具名不存在、参数不符合 Schema、或者引用的资源不存在（比如工单 ID 是伪造的）。我的 validate 节点会依次做三层校验：

第一层是工具白名单。我用 skills registry 里注册的工具名去匹配 LLM 输出的 tool_name，匹配不上就直接拦截，记录'工具 X 不在白名单'。

第二层是参数 Schema 校验，用 Pydantic 严格验证类型、必填字段和取值范围。比如 LLM 把 priority 写成了字符串 'high' 而不是枚举值 1，这一层就能拦下来。

第三层是资源存在性和权限校验。如果参数里引用了工单 ID，我会去数据库确认这个工单真实存在，并且当前用户必须是工单的 owner 或者 admin，否则也拦截。

三个校验只要有一个没通过，validate 节点就会往共享 State 里写错误信息和原始计划内容，并把执行状态标为 failed。

接下来才是关键——拦截之后怎么处理。

我最开始的做法是直接给用户报错，说'系统内部错误，请重试'，但很快就发现体验很差。用户只是表达

不够精确，结果就被冷冰冰地拒绝了，挫败感很强。
所以我改成了现在的两段式处理。

第一步，不是立刻降级到规则，而是先做一次提示词层面的纠正重试。我把校验失败的具体信息，比如'priority 参数缺失'或者'工具名 xxx 不在可用工具列表中'，直接拼回 prompt，让 LLM 带着错误反馈重新生成一次工具计划。这一步其实能解决大部分问题，因为很多时候 LLM 不是能力不够，就是第一次粗心或理解偏差，你给它指出来，它第二次就能改对。

第二步，如果第二次还是校验失败，我才切到规则模式。

这里要特别说明一点——我的规则模式并不是 LLM 的全面兜底，它只解决一类非常窄的问题：当用户意图极其简单明确的时候，比如就说'查我的工单'、'催办一下'，LLM 偶尔反而会在这种简单场景下产生幻觉，输出一个格式正确但根本不存在的工具名。这种简单意图如果用关键词匹配来做路由，反而比 LLM 更稳定。

所以我的规则模式本质上是基于关键词的硬匹配——'查'走查询、'催'走催办、'取消'走取消——只覆盖这种极简单的场景。它不处理复杂意图，也处理不了。

如果规则模式也匹配失败，说明用户的输入要么太模糊，要么就是复杂意图超出了规则的能力边界。这时候我不会再兜圈子重试，而是直接给用户返回一条人类可读的提示，比如'我理解您想操作工单，但不太确定具体想做什么。您可以这样说：查看我的工单、或者催办一下工单#45'。同时会设置一个重试上限，避免陷入死循环。

那用户体验是怎样的？对用户来说，如果第一次 LLM 计划就通过了校验，整个过程是无感的。如果走到了纠正重试，用户会稍微多等一秒钟左右，但依然不会看到任何错误信息。只有当纠正重试和规则模式都失败时，用户才会收到一条清晰的引导提示。

所以整体上，这个设计有三个特点：一是对 LLM 的小错误有容错能力，不会一次失败就卡住；二是有针对性的降级兜底，规则模式只做它最擅长的那一小部分；三是底线兜底——如果所有路径都走不通，系统会诚实告诉用户'我需要更多信息'，而不是默默失败或者胡乱执行。”

你提到 structured_hybrid 是"按标题切块 + 重合窗口"，这个方案里，你是怎么处理标题层级嵌套的？比如一篇文档有 H1 标题、H2 子标题、H3 小标题，你切块的时候是只在 H1 边界切，还是所有标题层级都做切分？如果 H2 下面的内容很短，你会把它和上一个同级块合并还是单独保留？

我在做 structured_hybrid 切块的时候，首先会识别文档的标题层级。对于企业制度文档来说，最小的语义单元一般是'条'——比如《IT 设备管理办法》第三条讲的是'设备借用流程'，这一条就是一个完整的意思，你没法把它拆成两半。所以我会先定位到每个子标题，把它的完整正文作为一个独立的段落单元。

然后我会逐个处理这些段落。如果内容很短——比如某一条只有 100 字，我不会让它单独变成一个 chunk，因为碎片化太严重的话，向量库里会多出很多几乎没信息量的条目，检索效率低，元数据开销也大。我会把它和下一个同级段落合并，一直合并到接近窗口上限，比如我的窗口设的是 800 字符，那就合并到七八百左右再断开。

反过来，如果一条本身就超长——比如有些政策条款写了上千字——我也不会硬塞成一个 chunk。这种超长块我会在句子边界处切分，尽量保证每个子块仍然是一个相对完整的语义片段。切完后子块之间会保留一定的上下文重叠，尽量减少因为硬切导致的语义断裂。

这个方案当然有缺点。合并小块确实可能引入不相关内容——一个 chunk 里前 200 字是'设备借用流程'，后 600 字是'违规处罚标准'，对借用流程的问题来说，那 600 字就是噪声。但这里有一个权衡：如果不合并，碎片化会让检索效率明显下降，而且 Qdrant 的存储压力也会变大。在我的实验里，这种上下文污染对生成质量的影响，比因为切太碎导致检索召回失败的影响要小。所以综合考虑，我接受了这个 trade-off。

如果一个句子本身就超过了窗口长度怎么办？比如某条政策规定里有一个特别长的句子，可能有 1000 多字，单句已经超了你设定的窗口上限。这种极端情况下，你是强制截断，还是有别的处理方式？

目前我的处理方式是我用的不是硬截断丢弃内容，而是字符级硬切分——意思是超过窗口上限的块，我会在字符边界处切开，内容全部保留，但可能会破坏语义边界。比如一个超长句子可能被切成两半，前后半句分属两个 chunk，这会影响检索时的语义完整性。

为了解决这个问题，我在入库时给每个 chunk 做了编号，并记录了它在文档中的层级路径——比如‘第三章 > 第二节 > 第 5 条’，这个路径我叫 section_path，存在 Qdrant 的 payload 里。这样在检索时，如果命中的 chunk 的前后相邻块属于同一个 section_path，我会在构建 prompt 时把它们拼接在一起，恢复被切断的上下文。这个优化我目前还没实现，但我确认了它的可行性。

关于短块合并，我设的默认合并目标是 520 字符。合并逻辑分两步。第一步是判断两个块是否属于同一

层级——同一条内的两段、同一章内的相邻条，都可以合并。第二步是判断主题是否相近。具体做法是取 section_path 的最后一级标题，如果标题为空就取正文开头，去掉标点和空白后截取前 24 个字符作为主题标签。然后做三层判断：两个标签完全相同、或一个包含另一个、或双方长度都大于等于 4 且共享至少一个二字 gram。满足任一条件就认为主题相近，可以合并。

这个方案当然有风险。如果两个块的标题都为空，取正文开头 24 字符做标签，可能会因为高频模板化开头导致误合并——比如两段不相关的内容都以'本公司全体员工'开头。不过在实际企业制度文档里，这种模板化开头通常出现在章级别的大标题下，而我的合并已经前置了层级判断，跨层级的误合并概率很低，所以目前是可以接受的。

所以我也想过一个改进方案，就是给同一个文件里的相邻 chunk 做编号。检索的时候，如果命中了某个 chunk，我会检查它的前一个和后一个 chunk 是否属于同一个文件。如果是同文件相邻的，就在构建 prompt 时把它们拼接在一起传给 LLM。这样相当于把检索命中的 chunk 当作锚点，自动扩展上下文窗口，能有效缓解因为硬切块导致的语义断裂。

这个方案我还没有实现，主要是两个考量。一个是 LLM 的上下文窗口限制——如果拼接后 token 超限，需要做截断，这个截断逻辑比我目前的方案更复杂。另一个是 Qdrant 的 payload 里需要维护 chunk 的顺序编号，入库逻辑要额外处理。我评估下来，这个优化有明确价值，但目前的强制截断在实际评测中对端到端质量的影响并不大，因为超长单句在企业制度文档里出现概率很低，所以在当前阶段我接受了这个 trade-off。

如果 MCP 客户端的调用量突然很大，比如同时有几十个 MCP 客户端在并发调用你的工单工具，你现有的架构有哪些环节可能会出现瓶颈？你会怎么应对？

核心结论：

1. 对于 FastAPI + MCP HTTP Server 这种典型场景（I/O 密集，比如调用 LLM API、查数据库、读写 Redis）CPU 通常不是第一瓶颈。内存、网络带宽、外部服务（数据库/LLM）的限流往往先于 CPU 耗尽。
2. Worker 数量可以远大于 CPU 核心数，尤其是在异步模式下。
但并非越多越好，超过某个点后收益递减，甚至因上下文切换而下降。
3. 你不需要企业级服务器就能支撑"几十个 MCP 客户端并发"
一台普通的 2~4 核开发机或云服务器（比如 2 核 4G）就足够了。
4. 只用一个异步 worker（即默认的 uvicorn 单进程），就能撑起几百个并发请求。

基于核心结论，进行并发分析：

请求进来

入口层：FastAPI / MCP HTTP Server

→首先 FastAPI 的并发能力取决于 uvicorn 服务器的 worker 数（由启动服务时的设定参数决定）

→MCP HTTP server 可以服务多个客户端，但能力也取决于进程数和 worker

路由层：鉴权、限流、请求校验

→Pydantic 请求体校验（解析 JSON → 校验字段类型、长度、正则等，纯 Python 运算）：单次校验通常在 0.1ms ~ 1ms 级别。即使 1000 并发，总 CPU 时间也就 1 秒，分散到多核上可承受。

→JWT 验证：CPU 密集型（签名验证，尤其 RS256 非对称算法），HS256 对称验证 ≈0.05ms，RS256 可能 1~2ms；

→API Key 查表：I/O 密集型，Redis 查询 0.5~2ms，数据库查询 5~20ms，如果 API Key 存在数据库且无缓存，高并发下 DB 连接可能打满。

→Redis 限流（滑动窗口 / 令牌桶）：I/O 密集型，耗时约 0.5~2ms，因为**每个请求都要同步等待 Redis 返回**，而 Redis 是单线程，高并发下命令排队延迟增加。

→创建 DB session（最常见的第一瓶颈）：从连接池获取一个连接，通常 <1ms（如果池中有空闲连接）。若连接池耗尽，请求会阻塞等待，直到有连接释放

Agent 编排层：LangGraph / legacy service workflow

→同一个用户同时发多个请求，可能出现最后写入覆盖前一次状态的问题。

→同一个草稿被两个请求同时续办，需要锁控制。

→同一个工单被多个请求同时催办/取消，需要行级锁

Planner / LLM 层

→大量请求同时进来时，LLM API 会限流

→响应时间长，会造成后端请求堆积。

MCP Wrapper 层

→几十个客户端同时调用写工具，DB 和 Redis 会有压力

→并发闸门

回答：

这个问题我之前确实没有系统性地考虑过，因为项目目前是 demo 阶段。但如果您让我来分析的话，我会沿着 MCP 工具调用的完整链路从外到内一层一层看。

先说一个重要的前提区分。MCP tools 暴露的是查单、催办、评论、取消这些确定性操作，它们不需要走 LLM 路由，直接进 MCP wrapper 做参数校验后调 service 层执行。所以 LLM 和 RAG 不是 MCP 高并发的瓶颈——这一点和 /agent 自然语言入口不一样。

接下来按链路分析。

入口层，MCP HTTP server 配合 FastAPI 的多 worker，承接几十个并发请求是没问题的，入口层我不太担心。

往下一层是 MCP wrapper 和 service 层。这里有两个需要注意的点。一个是幂等检查——每次写操作都要查 Redis 做幂等判断，如果 Redis 响应变慢或者挂了，幂等保护就会失效，可能导致重复创建工单。另一个是工单状态变更的并发冲突——比如两个 MCP 客户端同时催办同一张工单，或者一个人取消、一个人催办，操作同时到达。这种情况需要数据库层面做并发控制。我会用条件更新来处理，比如催办时 WHERE status IN 可催办状态集，更新失败就说明状态已经被其他操作改变，返回冲突即可。

再往下是数据库层，我觉得这是真正的瓶颈所在。每个工单操作都会访问 tickets 表，写 audit_logs 表。高并发下连接池最容易被占满。我在开发环境用的是默认配置，连接池 20 个。一个建单请求因为涉及多次数据库读写，连接持有时间会比其他操作更长。如果同时有十几个建单请求，连接池就可能被占满，后续请求就得排队。应对策略上，短期是调大连接池、设好 pool timeout 和 max overflow；中长期可以考虑把建单链路做一次精简——比如把审计日志从同步写改成异步写，减少连接占用时间。

Redis 这边，它的单线程吞吐能力其实很高，主要风险不在性能，而在可靠性。限流、幂等、确认态都存在 Redis 里，如果 Redis 不可用，限流可以降级放行，但幂等绝对不能丢。我的思路是在创建工单前加一层数据库级别的唯一约束兜底——比如同一用户短时间内相同标题和描述的工单视为重复——这样即使 Redis 挂掉，数据库层面也能拦住重复建单。

所以整体思路是：先区分调用路径，明确 MCP 工具不走 LLM；然后重点保护数据库连接池和幂等逻辑；Redis 做高并发缓存，但关键写保护要在数据库层兜底；监控连接池使用率，提前预警。

审计日志异步化之后，如果异步写入失败（比如消息队列挂了或者落库失败），那这次审计记录就丢了。对于你这个强调“全链路可审计”的系统来说，这是不是违背了设计初衷？你怎么平衡异步化的性能收益和审计完整性？

异步含义

1. 更新 ticket.status = cancelled
2. 把一条审计事件放进队列
3. 立即返回用户：已取消
4. 后台 worker 从队列取事件
5. 后台 worker 写 audit_logs 表

回答

"异步化确实会带来审计丢失的风险，所以我的做法不是全部异步，而是按事件性质分层处理。

分层原则很简单：涉及业务状态变更或安全决策的事件，必须同步写入审计日志，不容忍丢失。具体来说，取消工单、权限拒绝、高风险动作的确认、工单状态的变更——这些操作如果审计日志丢失，事后就无法追溯谁在什么时间做了什么决定，这是审计完整性的底线。

但是像模型 planner 输出的原始计划、API 访问日志、working memory 的定期 summary、调试用的 RAG trace 和问答链路的中间结果，这些数据更偏向观测性和调试用途，不涉及业务状态变更。异步写入丢失一条，影响的是调试便利性而不是审计完整性，这个代价我可以接受。

关于异步写入失败的处理，我也不会完全静默丢弃。我会把异步事件先写进 Redis Stream，如果写入失败就降级落到本地日志文件，至少保证数据有迹可循，后续可以通过离线脚本回放补录。

这个方案的局限性我也很坦诚——关键业务操作的审计仍然是同步的，性能收益有限。但我的判断是，在高并发场景下，如果一定要在审计完整性和响应延迟之间做选择，我宁可牺牲一点延迟，也不能让审计出现漏洞。如果要进一步优化性能，我会优先考虑数据库读写分离、连接池调优这些方案，而不是在审计可靠性上妥协。

所以整体上是一个分层的取舍——底线守住的同步写，观测性的异步化，丢失有兜底，性能不够想别的办法而不是砍审计。"

你项目里有一个很有意思的设计：同一套 skills registry 同时驱动 /agent 和 MCP tools。我想问的是——你在做这个设计的时候，遇到的最大挑战是什么？是技术实现上的，还是设计理念上的？

Agent skills registry：src/api/skills.py

MCP wrapper registry：src/mcp_wrapper/registry.py

MCP wrapper 最终复用的是同一批 service 工作流和同一套业务校验理念；并且部分工具名通过 alias 对齐，比如 lookup_ticket 映射到 get_ticket_detail。

坦白说，目前并不是完全由同一套 skills registry 同时驱动两边。这个状态是我架构演进的中间态，我可以说一下它怎么变成这样的，以及我理想中它应该是什么样。

最开始的时候只有 Agent 这一个入口。我设计了 skills.py 来定义所有工具的 tool_name、description 和 input schema，Agent 的 planner 用这些定义来做工具选择和参数抽取。

后来我加入了 MCP。我的设计初衷是让 MCP 客户端也能调用同一套工具，避免业务逻辑重复。但实际落地的时候遇到一个问题——skills.py 的字段定义是为 LLM function calling 优化的，跟 MCP 协议要求的 JSON Schema 格式不完全一致。当时为了快速跑通 MCP 链路，我给 MCP wrapper 单独写了一个 registry，手动把工具名映射到 handler 函数。

这就导致了现在的一个尴尬状态：业务执行和校验是复用的，但工具暴露层是分叉的。Agent 侧用 skills.py 定义工具，MCP 侧用自己的 registry。虽然两边底层的 service 代码是同一份，但如果我在 skills.py 里改了一个工具的参数定义，MCP 侧的映射可能不会同步更新。这种漂移实际上已经发生过一次——我改过催单工具在 skills.py 里的 description，但 MCP 侧的描述忘了同步，好在发现得早，没影响使用。

所以对我来说，最大的挑战不是技术实现，而是设计理念上的——怎么在系统多了一个入口之后，防止两套定义慢慢漂移。我的判断是，必须做到 Single Source of Truth。skills.py 应该是唯一的工具定义源，MCP 的 tool list、description 和 input schema 都应该从它自动生成，MCP wrapper 只负责协议适配和工具名的简单 alias。这个改动需要把 skills.py 的结构做一点调整，让它支持生成两种格式——LLM function calling 格式和 MCP JSON Schema 格式。但一旦改完，以后任何工具的变更都只需要改一个地方，两边自动同步，彻底杜绝漂移。

你提到 skills.py 目前是为 LLM function calling 优化的，跟 MCP 要求的 JSON Schema 格式不完全一致。能具体讲一下两者的格式差异在哪里吗？如果要改造成自动生成，你认为最棘手的技术点是什么？

两者的差异不在格式层面，而在语义层面。我举一个具体的例子——Agent 侧的工单取消工具定义，除了 name、description 和 inputSchema 这些 MCP 也需要的信息之外，还包含了很多内部治理字段：risk_level 用来标记高危、auth_rule 标记需要 owner 或 admin 权限、audit_event_type 指定触发哪种审计事件、route_name 是内部路由用的。

如果把这些字段原样暴露给 MCP 客户端，有两个问题。一个是安全——外部知道了哪些操作是 HIGH 风险、哪些路由路径存在，等于给了攻击者一份系统内部的导航地图。另一个是耦合——MCP 客户端不应该关心我的内部路由名称和审计事件编码，这些是后端自己的事。

更深层的问题是，skills.py 里不是所有条目都适合作为 MCP 工具暴露出去。比如 ticket_tool_planner 和 create_ticket，在 Agent 侧它们是给 Global Planner 做分支决策和编排使用的。create_ticket 内部涉及字段抽取、缺失判断、草稿暂存还是直接建单的分支，它是一个多步骤状态机，不是一次原子调用。如果把它直接映射成一个 MCP 工具，外部调用者拿到的是一个需要自己管理草稿生命周期、处理字段补全的函数，这和 Agent 内部的设计逻辑完全不一致。

所以我认为最棘手的技术点，不是写一个 JSON Schema 转换函数，而是重新思考 Skills 的职责边界。我的想法是把 Skills 拆成三层。

第一层是 ToolSpec——只包含 LLM 和 MCP 客户端需要看到的信息，比如 name、description、

inputSchema。这一层是安全的、可以被公开暴露的。

第二层是 ToolPolicy——包含 risk_level、auth_rule、audit_event_type 这些后端治理字段。这一层对 MCP 客户端透明，但后端 validator 和审计模块需要读它。这些信息应该从当前登录上下文中注入，而不是由调用方传入。

第三层是 ToolBinding——包含 handler、handler_semantics、route_name 这些执行调度信息。这一层只和内部 dispatch 逻辑相关，外部完全不需要感知。

从单一数据源的角度，skills.py 仍然是所有三层定义的总源。MCP 的工具列表通过从 skills.py 提取 ToolSpec 子集自动生成，Agent 侧还是使用完整的三层定义。

这里面最难的不是技术实现，而是分类策略——哪些字段入 ToolSpec、哪些入 ToolPolicy、哪些入 ToolBinding，不同工具可能需要不同的划分方式。比如工单查询的风险很低，ToolPolicy 可以很轻量；取消工单是 HIGH 风险，ToolPolicy 必须完整。我的设想是先对现有工具做一次梳理，按工具类型——原子查询、状态变更、内部编排——分组，每组定一套默认的分层模板，再对个别特殊工具做覆盖。

这个方案我目前还在设计阶段，但我确认方向是对的，因为它解决的不是格式问题，而是职责分离问题。

个人问题：

1.描述方式像"代码目录结构"，不像"设计思路"

你原话是：

"后端架构是分为多层，包括 FastAPI 路由层……LangGraph 流程编排层……MCP 工具调用层……service 业务逻辑实现层……scheme 输入输出结构体定义层……crud 层数据库持久化层……"

这种"层层罗列"式的表达，在面试中非常吃亏。面试官听到的是一堆技术名词的堆叠，而不是你在解决什么问题、做什么设计取舍。

暴露出的深层问题是：**你可能习惯用代码分层的思维来描述系统，但这套语言在面试中难以传递"思考深度"**。

面试官关心的不是你的代码目录怎么组织的，而是：为什么要有这一层？它解决了什么业务或技术痛点？没有这一层会怎样？

2.讲决策时只有结论，没有推导过程

你原始回答里的关键决策部分：

"关键决策就是 RAG 的文档切块方式选择和混合检索。还有模型工具选择推理和参数验证的分离……"

只有"做了什么"，没有"为什么这样做"、"对比过什么方案"、"踩过什么坑"。面试官听到这儿会觉得你说的可能是对的，但没有证据证明**这些决策是你思考出来的，还是你从某个教程里搬的**。

修改后：

1. 把"层层罗列"改成"问题驱动"的叙述结构

错误的讲法："我的架构分五层，第一层是……第二层是……"

正确的讲法：

"我做这个项目时遇到一个核心问题：Agent 流程里状态太多了，当前用户、草稿 ID、确认态 token、短期记忆……如果全写在一个函数里会越来越难维护。所以我用 LangGraph 把流程拆成了几个节点，每个节点只管一件事，状态靠 Graph 的 State 来传递和更新。"

规则：每介绍一个设计，先说"我遇到了什么问题"，再说"所以我做了什么"，最后说"效果是什么"。

2. 主动定义自己的贡献边界，不要等面试官追问

先规划清楚要实现什么功能，实现后是什么样子，然后基于这个需求进行实现，实现后验证是否符合最初的规划，最好设计测试方案，实现后进行测试并验收

你在 AI 辅助开发这件事上不用心虚，但需要主动界定清楚：

"整个项目的架构设计、模块划分、关键决策（像 RAG 的混合检索策略、Agent 的安全约束设计）都是我主导的。具体代码实现上我用了 Codex 辅助，但所有生成的代码我都会过一遍，修改不符合我设计意图的部分。像 RRF 融合函数、校验节点的逻辑，是我自己写的，因为这些地方需要精确控制。"

这样说，面试官会觉得你"能驾驭 AI"，而不是"被 AI 替代"。

3.每个关键技术决策，都准备"对比 → 选择 → 验证"三段论

决策点	对比了什么方案	为什么选这个	怎么验证的

RAG 混合检索	纯向量 / 纯关键词 / 混合	企业文档精确词多，纯向量漏召回	评测集 R@3 从 86%到 97%
草稿机制	直接报错 / 建一个不完整工单 / 暂存草稿	不完整工单会污染业务数据，报错体验差	草稿补全成功率 / 用户操作步骤减少
取消工单两步确认	直接执行 / 软删除 / 两步确认	不可逆操作，一旦误删无法恢复	防止模型幻觉误操作

规则模式具体怎么工作的？关键词匹配的规则是谁维护的？
规则模式能覆盖所有工具吗？遇到复杂意图怎么办？
你是在一开始就设计了双模式，还是后来发现 LLM 老出错才补的？

4. 你说的校验流程是对的，但少了一个关键环节：用户看到什么

以后回答既要有系统内部流程，也要有用户体验角度的流程。
如果连续重试都失败，有没有终止机制防止死循环？
5.如果 LLM 模式校验失败，回退到规则模式。规则模式成功了还好，如果规则模式也失败了呢？你的回答没有覆盖这个场景。面试官会立刻想到：如果规则模式也匹配不出来，最后的兜底是什么？会不会变成一个死循环？
设计理念：为什么选"重试+降级"而不是直接报错？
边界处理：重试几次？降级也失败怎么办？
用户体验：用户在这个过程中看到了什么？
对系统弱点的认知不够主动——规则模式兜底有局限、多重试可能带来的延迟、用户连续失败体验差，这些你没主动提。面试官会更喜欢那种"能坦诚讲清楚自己系统局限性的候选人"，而不是只讲优点的人。
6.对全局规划流程不清楚，不求代码级了解，流程了解也行，需要回头读一读流程
7.“最小子标题”这个定位很好，但没有展开讲你为什么这样定义
你说“找到最小子标题进行切分（一般政策最小是按照条进行划分的）”，这句话本身是精准的——企业制度文档的最小语义单元就是“条”。但你没展开说**你是怎么识别“条”的**。面试官可能会追问：你的标题识别依赖 Markdown 的 # 语法，还是解析了 DOCX 的样式层级？如果文档格式不规范怎么办？这个细节你没有铺垫。
8. 合并小块的策略说清楚了，但边界条件没提
你说“一直合并小的连续内容块，直到即将超过窗口大小”，这个逻辑是对的。但有一个边界情况你会被追问：**如果一个小标题下的内容本身就超过了窗口大小怎么办？** 比如一条政策规定写了一千多字，已经超了你的 800 字符窗口。你说“超长的 chunk 会进行切分”，但怎么切？是在句子边界切，还是只能硬截断？切完之后子块之间怎么保持语义关联？这个问题你暂时没展开。
9. 对缺点的分析方向对了，但深度不够
你主动提到“可能块里面只有 20% 有用信息，其他有无关信息造成上下文污染”，这个反思方向很好。但你没说你有没有做过什么来缓解这个问题——比如 chunk 级别的 metadata 标记？或者在检索时对匹配的 chunk 做上下文扩展（上下文窗口动态拼接）？面试官听到你指出问题，会接着看你有没有想过解决方案。
10. "强制截断"四个字太轻了，后果没展开
你直接说"我是强制截断的"，逻辑上没错——超过窗口就截，这是一种处理方式。但你没说强制截断会带来什么问题。面试官听到这儿，心里会有疑问：截断之后，被砍掉的那半句话会不会包含关键信息？比如"以下行为将受到严重处罚，包括但不限于开除"被截成了"以下行为将受到"，那这个 chunk 的语义就完全变了。
你应该在承认强制截断之后，立刻跟上对这个问题的分析——"我知道这样会丢信息，但...”，这样面试官会觉得你对缺陷有预判，而不是没想过。
11. 你提了一个很好的替代方案，但没评估它的代价
你说"想过为同一个文件相邻 chunk 进行编号，构建提示词之前判定如果相邻就连接再传入模型"。这个想法本身是正确的，很多生产级 RAG 系统就是这么做的——检索时定位到一个 chunk，然后把它的前驱和后继也一并传给 LLM。
但你没说你为什么没实现，除了"时间关系"。面试官会追问：这个方案有什么技术上的挑战让你搁置了？比如

拼接之后可能超过 LLM 的上下文窗口？或者判断"相邻"的逻辑在 Qdrant 的 payload 里需要额外维护？如果你能点出一个具体的工程难点，会比"时间关系"更有说服力。

12. 没有把"强制截断"和"相邻拼接"串成一个有优先级的优化路线

你现在是两个独立的陈述："目前强制截断"+"想过相邻拼接但没做"。面试官想听到的是一条有优先级的优化路径：

- 短期方案：强制截断，接受极少数边缘文档的检索质量下降（因为超长单句在企业制度里出现概率极低）
- 中期方案：chunk 编号加相邻扩展，检索时把前一个和后一个 chunk 也传给 LLM
- 长期方案：如果文档质量提升，可以考虑在切块前对超长句做预处理拆分

你如果能把这个演进路线讲出来，面试官会觉得你不仅有方案，还有工程排期的思维。

13. 分类标准"关键 vs 非关键"讲清楚了，但没有给面试官一个快速判断的原则，是看这个事件是否涉及业务状态变更？还是看是否可逆？还是看是否会影响合规审计？

14. 异步写入失败后的降级策略没有提：是静默丢弃？还是落本地日志文件做兜底？

1.FastAPI 在如今的项目总还能够负责什么功能，和之前的工作流阶段相比负责的功能有哪些变化