tools.tools=

[Tool(name='ragflow\_retrieval', description='Retrieve relevant chunks from the RAGFlow retrieve interface based on the question, using the specified dataset\_ids and optionally document\_ids. Below is the list of all available datasets, including their descriptions and IDs. If you\'re unsure which datasets are relevant to the question, simply pass all dataset IDs to the function.

{"description": "第一个演示，用deepseek api", "id": "7061d6142b2611f092eeee3c71c2c890"}', inputSchema={'type': 'object', 'properties': {'dataset\_ids': {'type': 'array', 'items': {'type': 'string'}}, 'document\_ids': {'type': 'array', 'items': {'type': 'string'}}, 'question': {'type': 'string'}},

'required': ['dataset\_ids', 'question']}, annotations=None)]

根据现有信息（包括你提供的代码和问题描述），​**​MCP 协议本身并未明确约定工具描述（Tool Schema）在大模型提示词中的具体格式​**​。以下是关键分析：

**​​MCP 协议的核心定义​​**

1. ​**​工具描述（Schema 定义）​**​
   * 通过 list\_tools() 接口返回 Tool 对象的 description 和 inputSchema 字段。
   * inputSchema 使用 JSON Schema 格式（如 {"type": "object", "properties": {...}}）定义工具输入参数的结构。
2. ​**​通信机制​**​
   * 客户端通过 SSE（Server-Sent Events）与服务端交互。
   * 客户端调用工具时传递符合 inputSchema 的 JSON 参数。

**​​未明确约定的部分​​**

1. ​**​如何将工具描述整合到大模型提示词​**​
   * ​**​问题示例​**​：是否应将 inputSchema 转换为自然语言描述？是否直接拼接 JSON 到提示词？
   * ​**​现状​**​：在 list\_tools() 的实现中，description 字段已经动态拼接了数据集列表（如 {"description": "...", "id": "..."}），但未规定具体的自然语言模板。
2. ​**​提示词格式的标准化​**​
   * ​**​示例差异​**​：
     + ​**​自然语言描述​**​：

text

复制

可用工具：ragflow\_retrieval。功能：基于问题从 RAGFlow 检索相关片段。参数：dataset\_ids（数据集ID列表）、question（问题）。

* + - ​**​结构化文本​**​：

text

复制

[TOOL] ragflow\_retrieval

- 输入: {"dataset\_ids": ["id1"], "question": "..."}

- 输出: 文本片段列表

* + ​**​MCP 协议未强制要求​**​：上述格式由客户端自行决定。

**​​实际实现中的常见做法​​**

1. ​**​自然语言拼接​**​
   * 将 description 和 inputSchema 转换为自然语言，例如：

python

复制

tool\_desc = f"工具名：{tool.name}\n描述：{tool.description}\n输入参数：{json.dumps(tool.inputSchema)}"

* + ​**​优点​**​：人类可读性强，适合通用大模型（如 GPT-4）。
  + ​**​缺点​**​：需客户端自行处理格式，可能引入不一致性。

1. ​**​结构化提示（JSON/YAML）​**​
   * 直接将 Tool 对象的 JSON 结构嵌入提示词：

text

复制

可用工具：{tools\_json}

* + ​**​优点​**​：机器可解析，适合需要结构化输入的 Agent 框架（如 AutoGPT）。
  + ​**​缺点​**​：对大模型的 prompt engineering 要求较高。

**​​建议​​**

1. ​**​遵循最佳实践​**​：
   * 若目标是大模型（如 GPT、Claude），将 description 和 inputSchema 转换为自然语言描述。
   * 示例代码：

python

复制

def format\_tool\_prompt(tool: Tool) -> str:

props = tool.inputSchema["properties"]

params = "\n".join([f"- {name}: {schema.get('type', 'string')}" for name, schema in props.items()])

return f"工具名：{tool.name}\n功能：{tool.description}\n参数：\n{params}"

1. ​**​在协议扩展中补充约定​**​：
   * 如果需多客户端一致性，可自定义 description 字段的模板，例如：

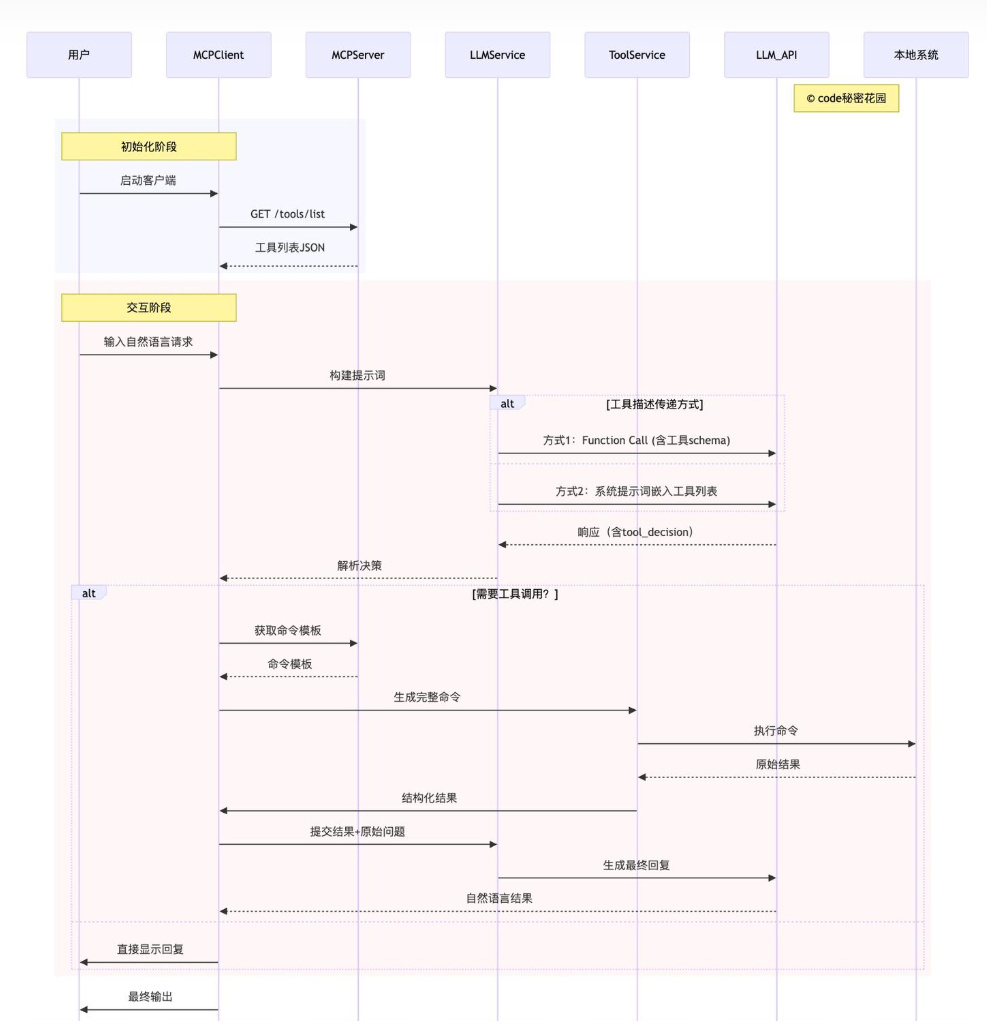
text

复制

"基于问题 `${question}` 从数据集 `${dataset\_ids}` 中检索相关片段。可选参数：document\_ids=${document\_ids}。"

**​​总结​​**

MCP 协议定义了工具的存在性和输入结构（inputSchema），但将 ​**​工具描述的提示词格式化逻辑交给客户端实现​**​。实际开发中需根据目标大模型的特点设计提示词模板。



MCP 协议与 ​**​Function Call​**​（如 OpenAI 的 Function Calling 机制）在核心目标上高度相关，均旨在让大模型（LLM）与外部工具/服务交互。以下是它们的核心关系及差异分析：

**​​1. 核心关系：功能互补​​**

| **​​维度​​** | **​​MCP 协议​​** | **​​Function Call​​（以 OpenAI 为例）** |
| --- | --- | --- |
| ​**​目标​**​ | 标准化大模型与工具服务的通信流程（工具发现、调用、结果返回） | 让大模型理解如何调用外部工具，并结构化返回调用参数 |
| ​**​功能定位​**​ | ​**​协议层​**​：定义工具描述、通信接口、数据格式 | ​**​应用层​**​：描述工具参数，供大模型生成符合预期的调用请求 |
| ​**​依赖关系​**​ | 可独立存在，需客户端和服务端实现协议逻辑 | 依赖大模型原生支持（如 GPT-4 的 function\_call 字段） |

**​​协作流程示例​​：**

1. ​**​工具定义​**​：MCP Server 通过 list\_tools() 返回工具列表（含 inputSchema）。
2. ​**​模型提示​**​：MCP Client 将工具 inputSchema 转换为类似 OpenAI Function Calling 的 JSON 描述，嵌入大模型提示词。
3. ​**​模型决策​**​：大模型生成符合 inputSchema 的 JSON 参数（如 {"dataset\_ids": ["id1"], "question": "..."}）。
4. ​**​调用执行​**​：MCP Client 通过 call\_tool() 触发服务端工具，返回结果。

**​​2. 核心差异：协议 vs 模型能力​​**

| **​​特性​​** | **​​MCP 协议​​** | **​​Function Call​​** |
| --- | --- | --- |
| ​**​工具描述格式​**​ | 使用 JSON Schema（inputSchema 字段） | 使用类似 JSON Schema 的格式（如 OpenAI 的 parameters 字段） |
| ​**​通信方式​**​ | 基于 SSE（Server-Sent Events）的异步通信 | 通过 API 请求/响应同步调用（如 OpenAI 的 /chat/completions 接口） |
| ​**​标准化程度​**​ | 定义完整协议规范（工具发现、调用、错误处理） | 仅定义工具参数结构，不约束通信实现 |
| ​**​适用场景​**​ | 需要多模型、多工具协作的复杂 Agent 系统 | 单一模型与简单工具交互（如单次函数调用） |

**​​关键互补点​​：**

* ​**​MCP 协议扩展了 Function Call 的边界​**​：  
  在 OpenAI 的 Function Calling 中，模型仅生成调用参数，而 MCP 协议进一步定义了：
  + 工具的动态发现（list\_tools()）。
  + 多工具协同调用的生命周期管理。
  + 流式结果返回（通过 SSE）。

**​​3. 实际应用场景​​**

**​​场景 1：基于 MCP 实现多模型 Agent 系统​​**

* ​**​步骤​**​：
  1. MCP Server 注册多个工具（如检索、绘图、计算）。
  2. MCP Client 将工具 inputSchema 转换为 Function Call 描述，嵌入提示词。
  3. 大模型（如 GPT-4）生成工具调用序列。
  4. MCP Client 按协议调用工具并整合结果。

**​​场景 2：兼容 OpenAI Function Calling​​**

* ​**​适配方案​**​：

python

复制

*# 将 MCP 的 Tool 对象转换为 OpenAI Function 格式*

def to\_openai\_function(tool: Tool) -> dict:

return {

"name": tool.name,

"description": tool.description,

"parameters": tool.inputSchema *# 直接复用 MCP 的 JSON Schema*

}

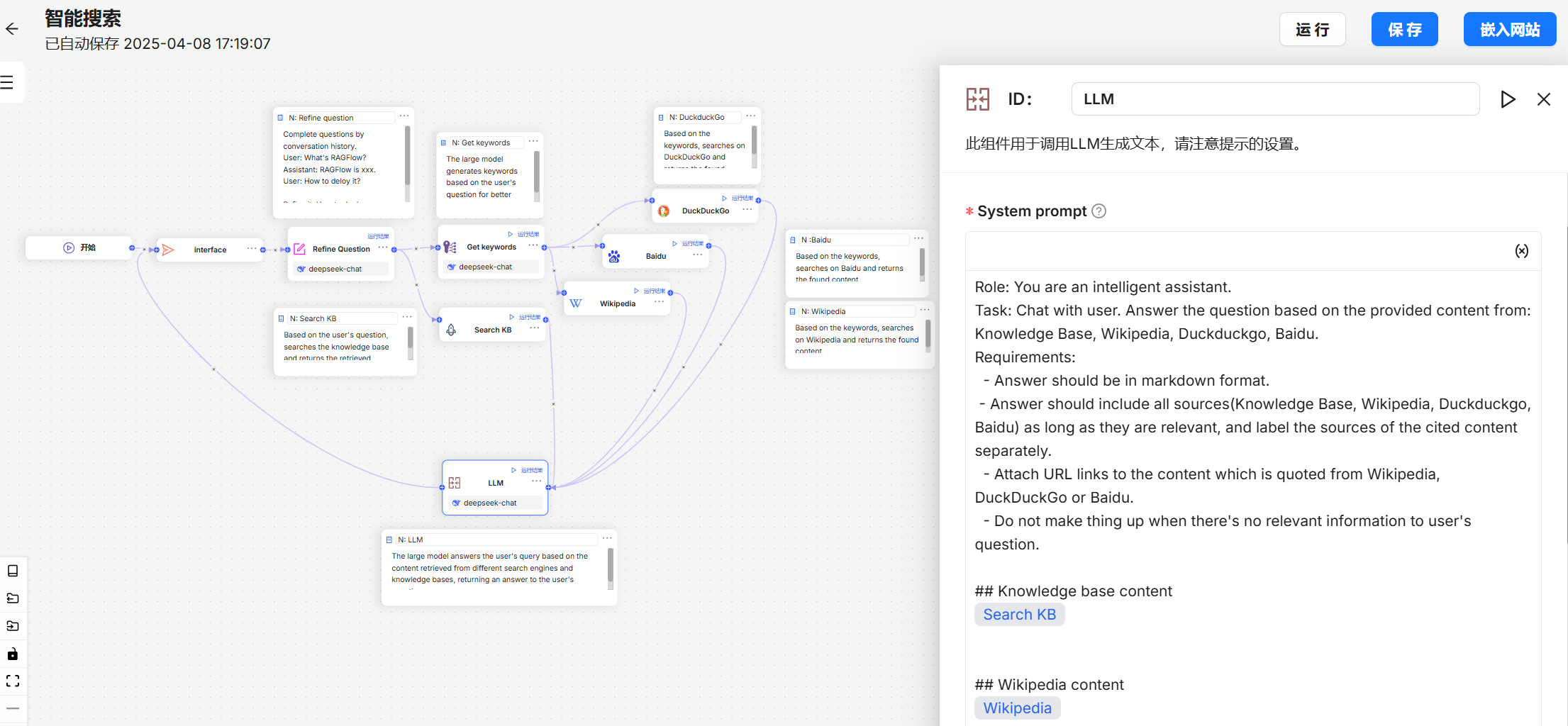
*# 在提示词中传入 functions=[to\_openai\_function(tool1), ...]*

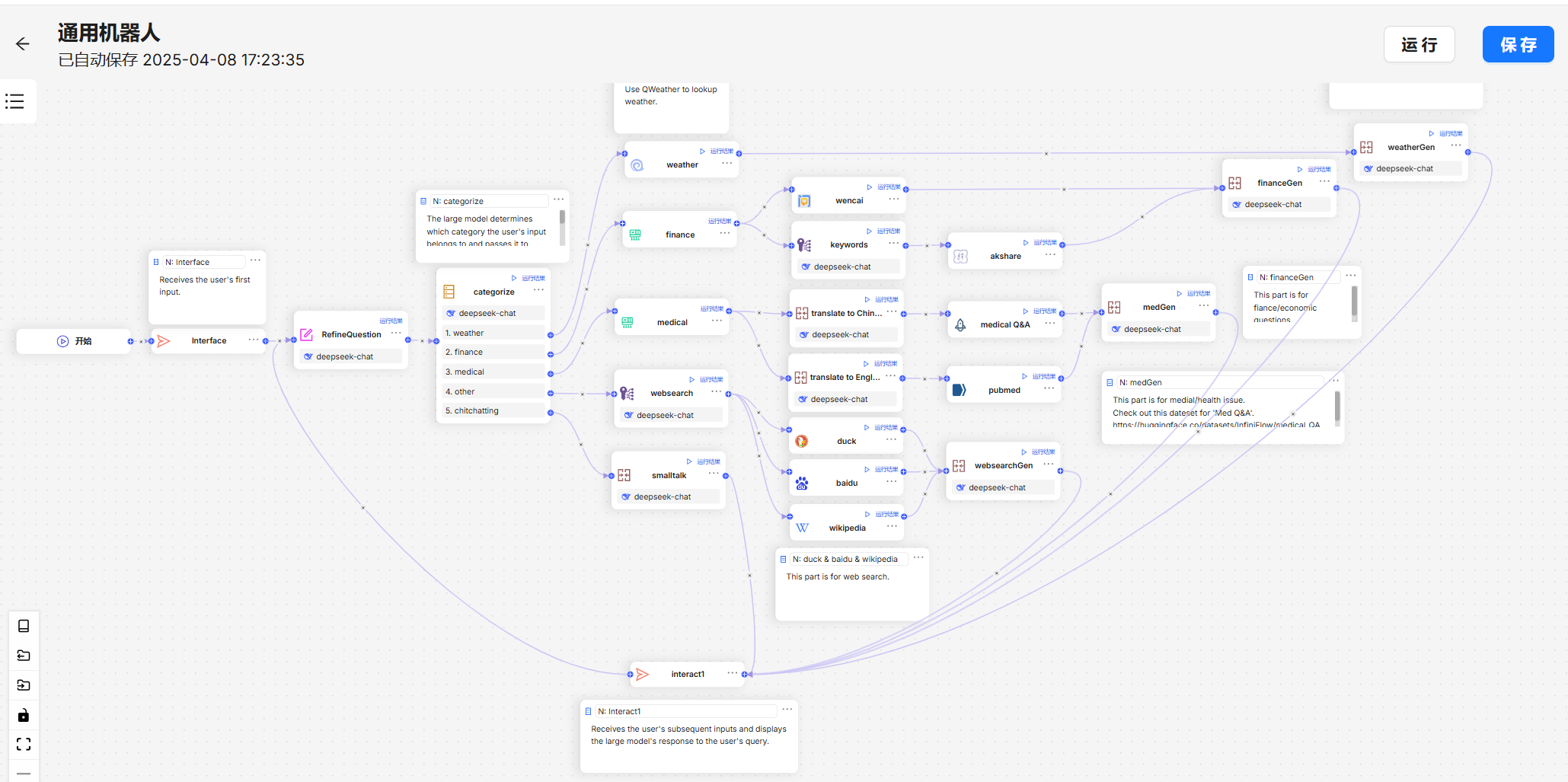
**​​4. 总结​​**

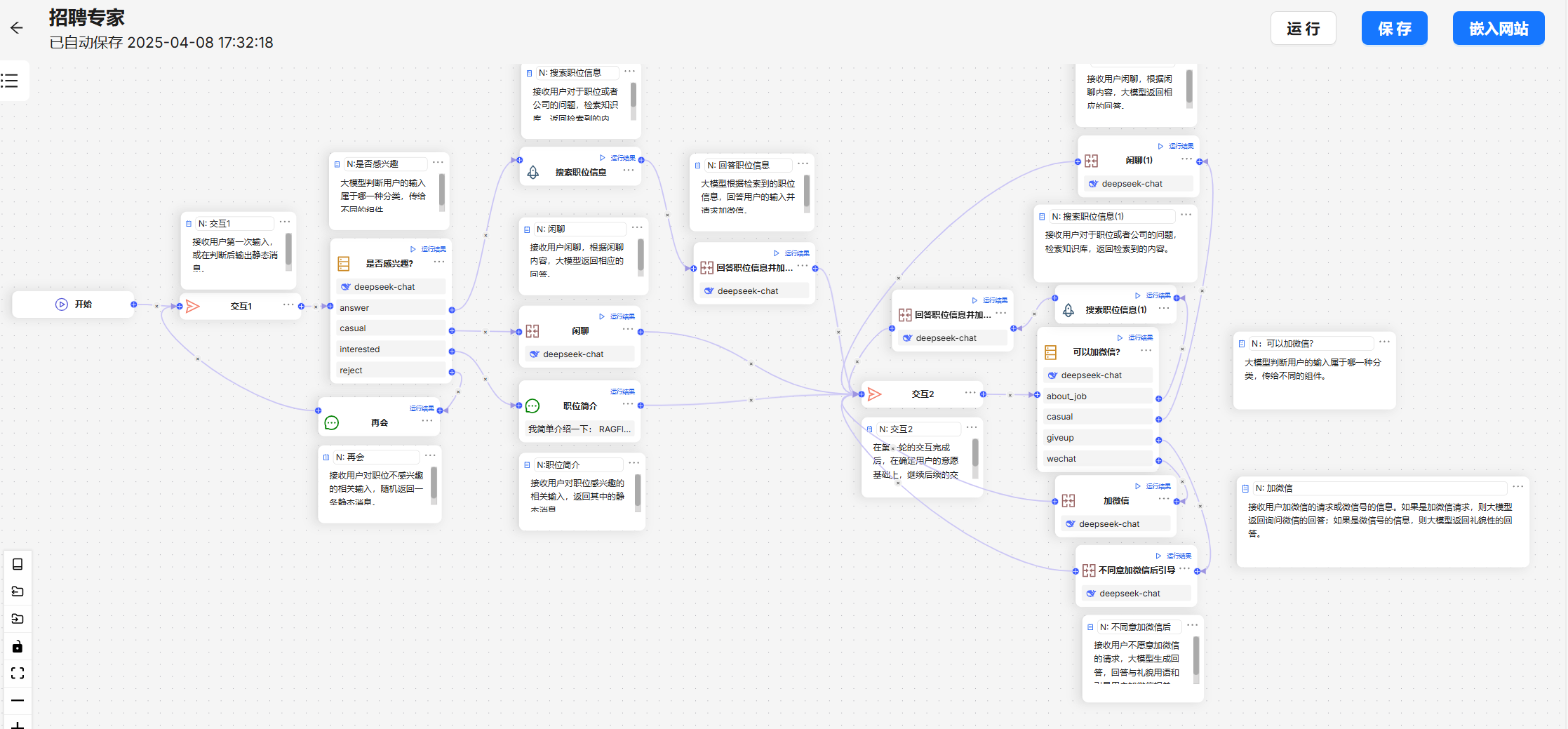
* ​**​MCP 协议​**​ 是 ​**​Function Call 的超集​**​：  
  提供工具发现、多步调用、流式返回等标准化流程，适用于复杂 Agent 系统。
* ​**​Function Call​**​ 是 ​**​MCP 的输入生成器​**​：  
  大模型通过 Function Call 生成符合 MCP 协议要求的参数，二者协同完成工具调用。

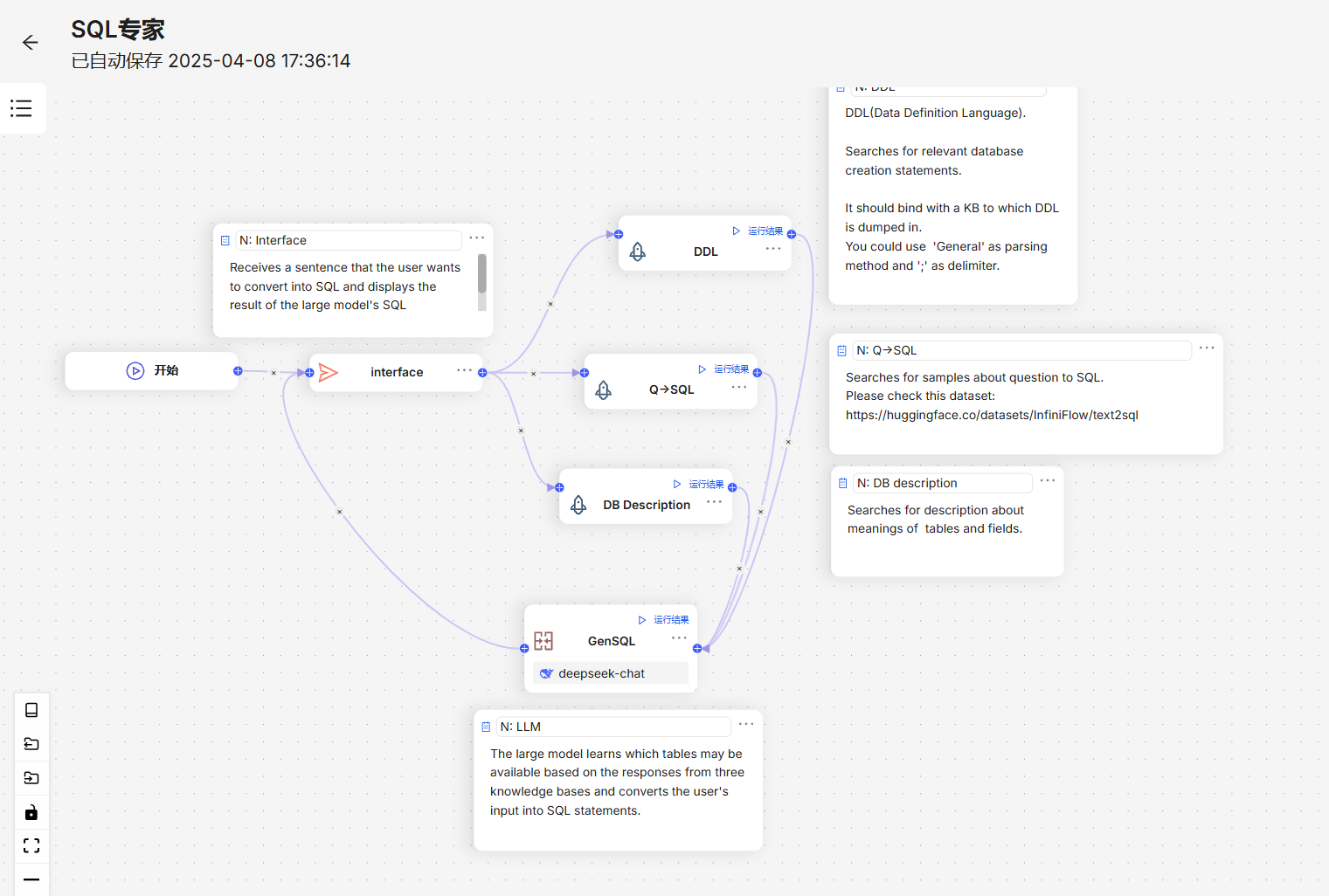
若需进一步优化 MCP 与特定模型（如 GPT、Claude）的 Function Call 兼容性，可设计适配层（如自动转换 inputSchema 到模型所需格式）。

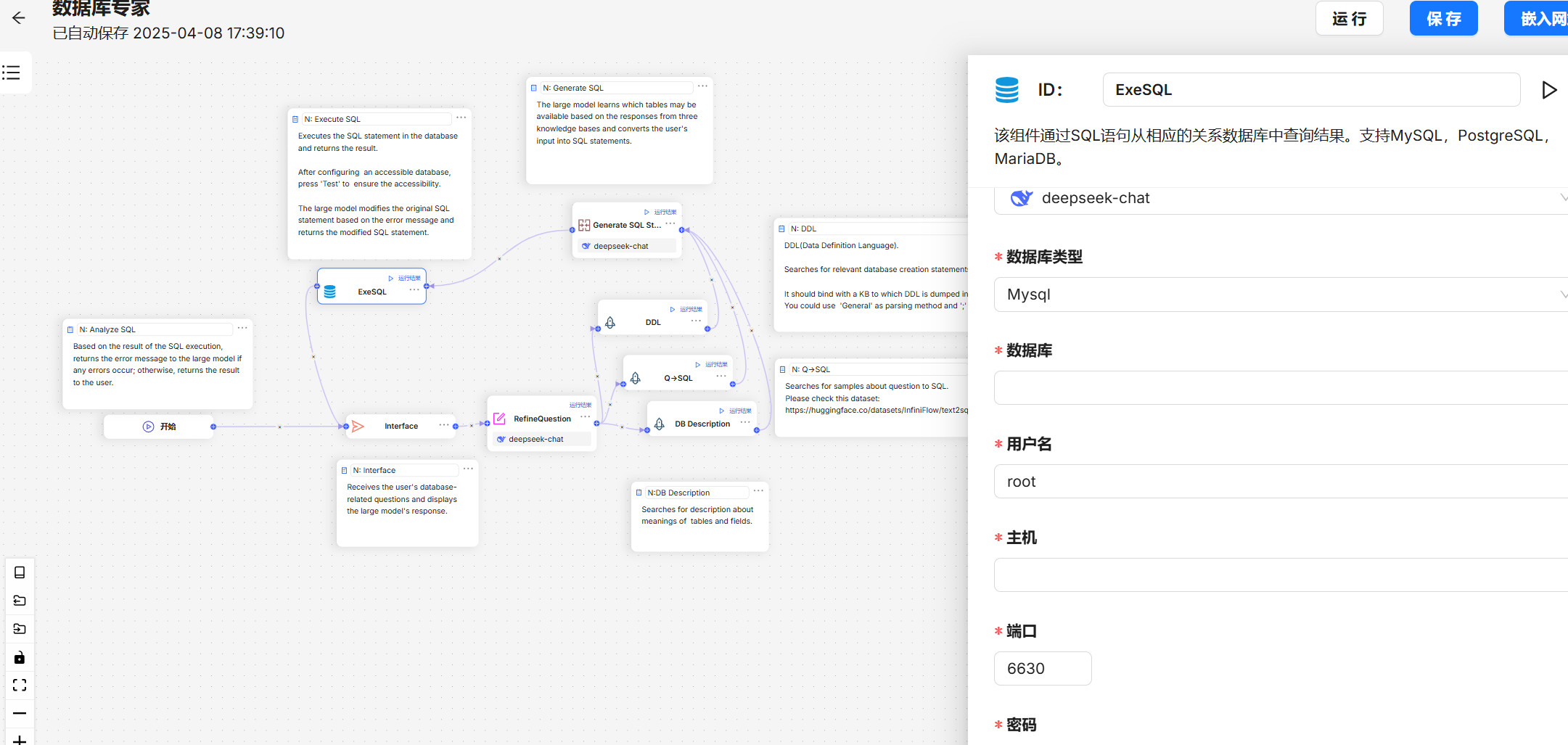












<https://huggingface.co/datasets/InfiniFlow/text2sql> 示例数据集