范围判定

1. 知识图谱构建

system:

```
You are a helpful assistant. You can help me by answering my questions.
```

user:

```
这是我刚才跟你谈论记录，请你先了解我们的任务
```

assistant:

```
好的，我已经仔细阅读了您提供的这份名为“改进建议.md”的文档。

根据文档内容，我理解我们当前的任务是围绕**构建一个关于特定管理规定（以《广铁集团动车组

具体来说，这个任务包含以下几个核心方面：

1.   **目标文档分析与知识图谱设计**：
     *    分析类似“广铁集团动车组列车票价管理实施细则”这样的规章制度文档。
     *    设计并优化知识图谱的模式（Schema），包括定义更精确、更全面的**实体类型**（

2.   **信息抽取方法**：
     *    利用强大的**大型语言模型（LLM）**，结合文档已有的良好结构（章、节、条），从
     *    探讨并确定具体的技术实施细节，包括：
          *    **文本分块策略**：如何将长文档切分成适合LLM处理的小块（建议按“节”或“条
          *    **提示词（Prompt）设计**：如何编写清晰、有效的指令来引导LLM准确地抽取
          *    **结果审核机制**：如何设计自动（如一致性、格式、逻辑校验）和半自动（如

3.   **知识图谱的应用目标**：
     *    最终构建的知识图谱旨在提高对管理规定信息的**检索效率**，增强**信息完整性**，
     *    关注组织层级、文档制定与执行单位、文档结构与内容细节（主题、事件、人物、单位）

总而言之，我们的任务是**设计一个优化的知识图谱模型来表示管理规定文档中的复杂信息，并探

我的理解是否准确？接下来您希望我们聚焦于哪个具体环节进行讨论或操作呢？
```

user:

```
现在我给你一份对markdown格式的管理文件的解析结果（json）格式，请你先理解这个json文档
```

assistant:

```
好的，我已经加载并查看了您提供的 `demo3.json` 文件。

这是一个 JSON 数组，其中每个元素（对象）代表从一个 Markdown 文件（`tests/demo3.m

根据其结构和字段，我理解这个 JSON 文件主要包含以下信息：

1.  **分块（Chunking）**：
    *   原始的 Markdown 文档被分割成了多个块（chunks）。
    *   每个块由一个 JSON 对象表示。

2.  **块内容与元数据**：
    *   `content`：每个块包含其对应的文本内容。值得注意的是，内容似乎包含了从根节点
    *   `heading`：明确提取了该块对应的 Markdown 标题。
    *   `tokens`：该块内容的 token 数量（可能是用于衡量文本长度或 LLM 处理成本）。
    *   `file_path`：指向原始 Markdown 文件的路径。
    *   `full_doc_id`：整个文档的唯一标识符。
    *   `page_numbers`：包含一个数组，指示该块内容在原始文档（可能是 PDF 转换而来

3.  **层级结构（Hierarchy）**：
    *   `chunk_id`：每个块的唯一标识符。
    *   `parent_id`：指向其父块的 `chunk_id`。根块的 `parent_id` 为 `null`。
    *   `child_ids`：包含一个数组，列出了直接属于该块的子块的 `chunk_id`。叶子节
    *   `chunk_order_index`：块在其父块下的顺序索引。
    *   这种父子关系（`parent_id`，`child_ids`）清晰地表达了原始 Markdown 文档

4.  **文档内容概要**：
    *   从 `heading` 和 `content` 字段的内容来看，这份 JSON 解析的是一份名为《广
    *   文件包含了总则、组织管理、监测预警、应急处置、恢复重建、调查评估、应急保障、附
    *   内容详细规定了票务系统突发事件的分类、预警级别、组织职责、响应流程、处置措施等

**总结**：这份 JSON 文件是对一份结构化的应急预案 Markdown 文档进行层级化分块后的结

我的理解是否准确？您可以基于这份解析结果提出下一步的要求了。
```

user:

```
现在你用python写一个从这个json文件中提取知识图谱实体和实体关系的程序：
1、提取实体和实体关系建模和方法已经在之前我给你的改进建议.md里；
2、用LLM进行实体和实体关系提取；
请你先理解我的要求，提出你的思路，我们讨论后再开始具体写出程序
```

assistant:

```
好的，我理解您的要求。

您希望我基于以下条件，为您编写一个 Python 程序，用于从 `demo3.json` 文件中提取知识

1.  **输入数据**: `demo3.json` 文件，它包含了从 Markdown 管理文件解析出的、带有层
2.  **知识图谱模式（Schema）**: 实体和关系的类型已经在您之前提供的 `改进建议.md` 文
3.  **提取方法**: 利用大型语言模型（LLM）进行实体和关系的抽取。
4.  **输出**: 提取出的实体和关系列表。
5.  **当前步骤**: 在编写具体代码之前，先提出我的思路，供您审阅和讨论。

---

**我的思路如下：**

1.  **环境准备与配置**:
    *   需要安装必要的 Python 库: `json` (内置库，用于加载 JSON 文件)，以及一个
    *   配置 LLM API 密钥: 通常通过环境变量或配置文件安全地管理 API 密钥。
    *   定义知识图谱模式: 在代码中明确定义 `改进建议.md` 中确定的实体类型列表和关系

2.  **数据加载**:
    *   编写函数读取 `demo3.json` 文件，将其内容加载为一个 Python 列表（其中每个

3.  **选择处理单元（Chunk Selection）**:
    *   遍历 JSON 文件中的每一个 chunk 对象。
    *   **策略**: 初步考虑处理**每一个** chunk。因为 `demo3.json` 中的 `conten

4.  **构建 LLM 提示词（Prompt Engineering）**:
    *   根据 `改进建议.md` 中的指导，设计两个核心的提示词模板:
        *   **实体提取提示词**:
            *   输入: 单个 chunk 的 `content` 文本。
            *   指令: 要求 LLM 从文本中识别并抽取出所有预定义的实体类型（文档、章节
            *   附加信息: 在提示词中包含每种实体类型的简要定义或示例，以提高准确性。
            *   输出格式: 明确要求 LLM 以 JSON 格式输出，例如 `{"entities": [{
        *   **关系提取提示词**:
            *   输入: 单个 chunk 的 `content` 文本。(*或者，为了更好地捕捉关系，
            *   指令: 要求 LLM 从文本中识别并抽取出所有预定义的关系类型（隶属、版本
            *   附加信息: 在提示词中包含每种关系类型的简要定义或示例。
            *   输出格式: 明确要求 LLM 以 JSON 格式输出，例如 `{"relations": [
    *   将 chunk 的 `content` 填入提示词模板，生成最终发送给 LLM 的具体指令。

5.  **LLM API 调用与结果解析**:
    *   编写一个函数，负责调用配置好的 LLM API。该函数接收生成的提示词作为输入。
    *   发送请求并获取 LLM 的响应。
    *   处理 API 可能返回的错误（如超时、速率限制、认证失败等）。
    *   解析 LLM 返回的 JSON 字符串。需要健壮地处理可能出现的 JSON 解析错误（例如
    *   从解析后的 Python 对象中提取出实体列表和关系列表。

6.  **结果聚合与去重**:
    *   初始化两个集合（Set）：一个用于存储所有提取到的实体（元组形式，例如 `(name,
    *   在处理完每个 chunk 后，将从 LLM 获取并解析出的实体和关系添加到相应的集合中。

7.  **输出**:
    *   处理完所有 chunk 后，将集合中的实体和关系转换回列表或字典的格式，方便查看或
    *   打印或保存最终提取到的、去重后的实体和关系列表。

8.  **代码结构**:
    *   主函数/脚本：协调整个流程（加载数据 -> 遍历 chunk -> 调用提取 -> 聚合结果
    *   LLM 调用模块/函数：封装与 LLM API 的交互。
    *   提示词生成模块/函数：根据 chunk 内容和模式生成具体的提示词。
    *   配置文件（可选）：存储 API 密钥、模型名称等。

**待讨论/确认的点：**

*   **LLM 选择**：您计划使用哪个具体的 LLM 服务或模型？不同的模型可能在遵循指令、输
*   **API 密钥管理**：确认 API 密钥的安全存储方式。
*   **性能与成本**：`demo3.json` 文件包含 61 个 chunk。对每个 chunk 分别调用 L
*   **错误处理**：对于 LLM 返回无效 JSON 或提取结果不符合预期的情况，需要定义处理策
*   **关系提取的上下文**：仅使用单个 chunk 的 `content` 提取关系可能无法捕捉跨 ch
*   **实体/关系规范化**：LLM 提取的实体名称可能存在细微差异（如“集团公司客运部” vs

请您审阅以上思路，特别是待讨论的点。如果您觉得方向正确，或者有任何修改意见，请告诉我，
```

user:

```
你提出下面的两点非常有意义，请你考虑进去
1、关系提取的上下文：仅使用单个 chunk 的 content 提取关系可能无法捕捉跨 chunk 的关
2、实体/关系规范化：LLM 提取的实体名称可能存在细微差异（如“集团公司客运部” vs “客运部
```

assistant:

```
好的，这确实是两个非常关键的点，直接关系到知识图谱的完整性和准确性。我们来把这两点考虑

---

**Refined 思路：**

1.  **环境准备与配置**：(同前)
    *   安装所需库 (`json`, LLM 库)。
    *   配置 LLM API 密钥。
    *   定义知识图谱模式（实体类型、关系类型）。
    *   **新增**：定义一个简单的**实体名称规范化映射表 (Canonical Map)**，例如：

        CANONICAL_MAP = {
            "客运部": "集团公司客运部",
            "信息技术所": "集团公司信息技术所", # 假设需要统一
            "广铁集团": "中国铁路广州局集团有限公司",
            "集团公司": "中国铁路广州局集团有限公司", # 根据上下文判断是否合适
            "本预案": "《广州局集团公司客票发售和预订系统（含互联网售票部分）应急预
            # ... 更多可能的别名、简称 -> 规范名
        }


2.  **数据加载**: (同前)
    *   读取 `demo3.json`。

3.  **处理单元选择与上下文增强**:
    *   遍历 JSON 文件中的每一个 chunk 对象。
    *   **策略**: 仍然以单个 chunk 作为 LLM 处理的基本单元。
    *   **上下文增强**: 虽然 `content` 字段包含了标题路径，这有助于理解当前 chunk
        *   **阶段一 (LLM 抽取)**: LLM 专注于从**当前 chunk 的 `content` 文本
        *   **阶段二 (结构化抽取)**: 在 LLM 处理完所有 chunk 后，利用 `demo3.j

4.  **构建 LLM 提示词**: (基本同前)
    *   **实体提取提示词**: 保持不变，要求提取预定义实体并以 JSON 输出。包含类型定
    *   **关系提取提示词**: 明确指示 LLM **仅提取在所提供的文本段落内能直接找到依据

5.  **LLM API 调用与结果解析**: (同前)
    *   调用 LLM API，处理错误，解析返回的 JSON。

6.  **结果聚合、规范化与去重**: (**重点改进**)
    *   初始化三个集合: `entities_set` (存储规范化后的 `(name, type)` 元组), `
    *   **定义规范化函数**:

        def normalize_entity_name(raw_name):
            cleaned_name = raw_name.strip() # 去除首尾空格
            # 查找映射表，如果找到则返回规范名，否则返回清理后的原始名
            return CANONICAL_MAP.get(cleaned_name, cleaned_name)

    *   **处理 LLM 结果**:
        *   对于从 LLM 提取的每个实体 `(raw_name, type)`:
            *   `normalized_name = normalize_entity_name(raw_name)`
            *   将 `(normalized_name, type)` 添加到 `entities_set`。
            *   将规范化后的实体信息存入 `chunk_entities[current_chunk_id]`。
        *   对于从 LLM 提取的每个关系 `(raw_source, raw_target, type)`:
            *   `normalized_source = normalize_entity_name(raw_source)`
            *   `normalized_target = normalize_entity_name(raw_target)`
            *   将 `(normalized_source, normalized_target, type)` 添加到 `

7.  **添加结构化关系（BelongsTo）**: (**新增步骤**)
    *   在所有 chunk 都经过 LLM 处理后，再次遍历 `demo3.json` 数据。
    *   对于每一个 chunk `C`:
        *   获取其 `chunk_id` 和 `parent_id`。
        *   如果 `parent_id` 不为 `null`:
            *   获取当前 chunk `C` 的规范化实体名称（可能需要从 `C['heading']`
            *   获取父 chunk `P` 的规范化实体名称（同理，从 `P['heading']` 规
            *   **如果能成功找到代表这两个 chunk 的实体名称**（例如，`child_ent
            *   *注意：* 此处需要仔细处理如何准确识别代表 chunk 本身的实体。一个

8.  **输出**：(同前)
    *   将 `entities_set` 和 `relations_set` 中的内容转换为列表或字典格式进行输

**改进后的优势：**

*   **关系上下文**：通过结合 LLM 抽取（处理文本内关系）和结构化抽取（处理层级关系），
*   **实体规范化**：通过引入简单的规范化步骤，减少了因名称不一致导致的数据冗余和后续处

**待确认/进一步细化：**

*   **规范化映射表 `CANONICAL_MAP` 的构建**：这个表需要根据文档内容和常见缩写预先
*   **结构化关系中实体的识别**：在步骤 7 中，如何精确地将 chunk ID 映射到其代表的实
*   **LLM 能力**：即使有规范化，LLM 提取的实体名可能仍有预期外的变化。规范化映射表需

这个细化后的思路解决了您提出的两个关键问题，使得程序的目标更加清晰和健壮。如果您同意这
```

user:

```
可以开始构思具体的 Python 代码实现了
```

assistant:

```
Okay, let's start building the Python code based on our refined思路.

I'll structure the code with clear functions for each step: loading data

**Assumptions:**

1.  You have a Python environment (>= 3.7 for type hints).
2.  You will replace the placeholder `call_llm` function with your actua
3.  You have the `demo3.json` file in the same directory or provide the
4.  You'll set the `LLM_API_KEY` environment variable or modify the code


import json
import os
import logging
import time # For potential rate limiting
from typing import List, Dict, Set, Tuple, Optional, Any

# --- Configuration ---
# 1. LLM Configuration
# !! IMPORTANT: Replace with your actual LLM provider's method !!
# Example for OpenAI (requires `pip install openai`)
# from openai import OpenAI
# client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))
# LLM_MODEL = "gpt-3.5-turbo" # Or your preferred model

# For this example, we use a placeholder
LLM_API_KEY = os.environ.get("YOUR_LLM_API_KEY") # Replace YOUR_LLM_API_
LLM_MODEL = "your-chosen-llm-model" # Replace with your model identifier

# 2. File Path
INPUT_JSON_PATH = "demo3.json"

# 3. Knowledge Graph Schema (Based on 改进建议.md)
ENTITY_TYPES = [
    "文档", "章节", "主题", "关键词", "人员", "角色",
    "组织", "时间", "事件", "法规"
]

RELATION_TYPES = [
    "隶属关系", "版本关系", "引用", "依据", "责任",
    "审批", "时间", "生效", "关联"
]

# 4. Entity Normalization Map (Example - expand as needed)
CANONICAL_MAP = {
    "客运部": "集团公司客运部",
    "信息技术所": "集团公司信息技术所",
    "科信部": "集团公司科信部",
    "财务部": "集团公司财务部",
    "计统部": "集团公司计统部",
    "电务部": "集团公司电务部",
    "供电部": "集团公司供电部",
    "宣传部": "集团公司宣传部",
    "调度所": "集团公司调度所",
    "集团公司应急领导小组办公室": "集团公司应急领导小组办公室", # Already specifi
    "集团公司应急领导小组": "集团公司应急领导小组", # Already specific
    "国铁集团应急领导小组办公室": "国铁集团应急领导小组办公室",
    "国铁集团应急领导小组": "国铁集团应急领导小组",
    "国铁集团客运部": "国铁集团客运部",
    "12306科创中心": "12306科创中心",
    "广铁集团": "中国铁路广州局集团有限公司",
    "集团公司": "中国铁路广州局集团有限公司", # Be careful with context, might
    "本预案": "《广州局集团公司客票发售和预订系统（含互联网售票部分）应急预案》", #
    "《铁路客票发售和预订系统(含互联网售票部分)应急预案》": "《铁路客票发售和预订系统
    "《广州局集团公司网络安全事件应急预案》": "《广州局集团公司网络安全事件应急预案》
    "《广州局集团公司信息系统故障应急处置和调查处理办法》": "《广州局集团公司信息系统
    "客票系统": "客票发售和预订系统", # Normalize system name
    # Add station names if needed, e.g., "广州站": "广州站"
}

# 5. Logging Configuration
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelnam

# --- Helper Functions ---

def load_json_data(file_path: str) -> Optional[List[Dict[str, Any]]]:
    """Loads data from a JSON file."""
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            data = json.load(f)
        logging.info(f"Successfully loaded data from {file_path}")
        return data
    except FileNotFoundError:
        logging.error(f"Error: Input file not found at {file_path}")
        return None
    except json.JSONDecodeError:
        logging.error(f"Error: Could not decode JSON from {file_path}")
        return None

def normalize_entity_name(raw_name: str) -> str:
    """Normalizes entity names using the CANONICAL_MAP."""
    if not isinstance(raw_name, str):
        logging.warning(f"Attempted to normalize non-string value: {raw_
        return str(raw_name) # Attempt to convert to string if not alrea
    cleaned_name = raw_name.strip().replace('\n', ' ') # Clean whitespac
    # Simple lookup, case-sensitive for now
    return CANONICAL_MAP.get(cleaned_name, cleaned_name)

def create_entity_prompt(chunk_content: str) -> str:
    """Creates the prompt for entity extraction."""
    # Provide definitions to help the LLM
    definitions = """
    实体类型定义：
    - 文档：管理规定的文件名称，如《应急预案》。
    - 章节：文档中的具体章节标题，如“1 总则”。
    - 主题：文档或章节的核心议题，如“应急组织机构”。
    - 关键词：文本中重要的名词或术语，如“客票系统”、“应急响应”、“电子客票”。
    - 人员：具体的人名（此文档中可能较少）。
    - 角色：指代具有特定职责的职位或岗位，如“客运部主任”、“售票员”。
    - 组织：涉及的单位、部门或公司，如“中国铁路广州局集团有限公司”、“集团公司客运部”
    - 时间：具体的日期、时间点或时间段，如“2021年”、“4小时及以上”、“每年3月”。
    - 事件：文档中描述的具体活动或状况，如“系统突发事件”、“启动应急预案”、“应急演练”
    - 法规：引用的其他法规或文件名称及其编号，如“《铁路客票发售和预订系统(含互联网售
    """
    prompt = f"""
请从以下文本中提取定义的实体类型。

{definitions}

预定义的实体类型列表：{', '.join(ENTITY_TYPES)}

文本：
\"\"\"
{chunk_content}
\"\"\"

请以严格的 JSON 格式输出，包含一个名为 "entities" 的列表，其中每个对象包含 "name"
例如：
{{
  "entities": [
    {{"name": "集团公司客运部", "type": "组织"}},
    {{"name": "售票故障", "type": "事件"}},
    {{"name": "《铁路客票发售和预订系统(含互联网售票部分)应急预案》（铁办客〔2021〕
  ]
}}
"""
    return prompt

def create_relation_prompt(chunk_content: str) -> str:
    """Creates the prompt for relation extraction (within the chunk)."""
    # Provide definitions
    definitions = """
    关系类型定义 (请仅提取文本段落内明确描述的关系):
    - 隶属关系 (BelongsTo): 通常是结构化的，此提示词主要关注文本内描述，如“办公室设
    - 版本关系 (HasVersion): 指明文档的版本信息或与其他版本的关系 (如“修订版”、“废
    - 引用 (References): 一个实体提到了另一个实体或文件，如“详见附件5”。
    - 依据 (BasedOn): 指出制定某文件或采取某行动所依据的法规或原则，如“根据...制定
    - 责任 (ResponsibleFor): 指明某个角色或组织负责某项任务或职责，如“客运部负责协
    - 审批 (ApprovedBy): 指出某事项需要经过哪个组织或角色批准，如“经...同意后”。
    - 时间 (OccursAt): 事件发生的时间，或规定适用的时间点/段，如“事件影响4小时”、“
    - 生效 (EffectiveFrom): 规定或文件的生效日期，如“自发布之日起实施”。
    - 关联 (RelatedTo): 实体间的其他关联，如“与...不一致时，以此为准”。
    """
    prompt = f"""
请从以下文本中提取实体之间的关系。请专注于在文本段落中**直接陈述**的关系。

{definitions}

预定义的关系类型列表：{', '.join(RELATION_TYPES)}

文本：
\"\"\"
```

```
{chunk_content}
\"\"\"

请以严格的 JSON 格式输出，包含一个名为 "relations" 的列表，其中每个对象包含 "sourc
例如：
{{
  "relations": [
    {{"source": "集团公司应急领导小组办公室", "target": "集团公司客运部", "type
    {{"source": "本预案", "target": "《铁路客票发售和预订系统(含互联网售票部分)应
    {{"source": "客运部", "target": "协调各相关部门", "type": "责任"}}
  ]
}}
"""
    return prompt

# --- LLM Interaction (Placeholder) ---

def call_llm(prompt: str) -> Optional[str]:
    """
    Placeholder function to call the LLM API.
    Replace this with your actual LLM API call logic.
    """
    logging.info(f"--- Sending Prompt to LLM (Length: {len(prompt)}) ---
    # logging.debug(prompt) # Uncomment to see the full prompt
    logging.info("--- Waiting for LLM Response ---")

    # !! --- REPLACE THIS SECTION --- !!
    # Example using OpenAI client (ensure client is initialized above)
    # try:
    #     response = client.chat.completions.create(
    #         model=LLM_MODEL,
    #         messages=[{"role": "user", "content": prompt}],
    #         temperature=0.1, # Lower temperature for more deterministi
    #         response_format={ "type": "json_object" } # If supported b
    #     )
    #     llm_output = response.choices[0].message.content
    #     logging.info("--- Received LLM Response ---")
    #     # time.sleep(1) # Add delay if needed for rate limiting
    #     return llm_output
    # except Exception as e:
    #     logging.error(f"LLM API call failed: {e}")
    #     return None

    # --- Placeholder Response (REMOVE WHEN USING REAL LLM) ---
    # Simulate a response for testing parsing logic
    if "提取定义的实体类型" in prompt:
         # Simulate entity extraction response
        logging.warning("Using PLACEHOLDER response for entity extractio
        # Simulate based on a known chunk, e.g., chunk_id: e50d6d8c_chun
        if "集团公司成立客票系统应急领导小组" in prompt:
             return json.dumps({
                 "entities": [
                     {"name": "集团公司", "type": "组织"},
                     {"name": "客票系统应急领导小组", "type": "组织"},
                     {"name": "集团公司应急领导小组", "type": "组织"},
                     {"name": "科信部", "type": "组织"},
                     {"name": "客运部", "type": "组织"},
                     {"name": "信息技术所", "type": "组织"},
                     {"name": "财务部", "type": "组织"},
                     {"name": "计统部", "type": "组织"},
                     {"name": "电务部", "type": "组织"},
                     {"name": "供电部", "type": "组织"},
                     {"name": "宣传部", "type": "组织"},
                     {"name": "调度所", "type": "组织"},
                     {"name": "广州铁路公安局", "type": "组织"},
                     {"name": "集团公司应急领导小组办公室", "type": "组织"},
                     {"name": "客运部主任", "type": "角色"},
                     {"name": "057-22632", "type": "关键词"}, # Example, 
                     {"name": "各车务站段", "type": "组织"}
                 ]
             })
        else:
            return json.dumps({"entities": []}) # Default empty for plac
    elif "提取实体之间的关系" in prompt:
        # Simulate relation extraction response
        logging.warning("Using PLACEHOLDER response for relation extract
        if "集团公司成立客票系统应急领导小组" in prompt:
            return json.dumps({
                "relations": [
                    {"source": "集团公司应急领导小组办公室", "target": "集团公
                    {"source": "集团公司应急领导小组办公室", "target": "客运
                    {"source": "各车务站段", "target": "集团公司", "type": 
                ]
            })
        else:
             return json.dumps({"relations": []}) # Default empty
    else:
        return None
    # !! --- END REPLACE/REMOVE --- !!


def parse_llm_response(response_text: Optional[str]) -> Optional[Dict[st
    """Safely parses the LLM's JSON response."""
    if not response_text:
        return None
    try:
        # Sometimes LLMs wrap the JSON in  ...
        if response_text.strip().startswith(""):
            response_text = response_text.strip()[7:-3].strip()
        elif response_text.strip().startswith("`"):
             response_text = response_text.strip()[1:-1].strip()

        parsed_data = json.loads(response_text)
        # Basic validation
        if isinstance(parsed_data, dict) and \
           (('entities' in parsed_data and isinstance(parsed_data['entit
            ('relations' in parsed_data and isinstance(parsed_data['rela
            return parsed_data
        else:
            logging.warning(f"LLM response is valid JSON but not the exp
            return None
    except json.JSONDecodeError as e:
        logging.error(f"Failed to decode LLM JSON response: {e}\nRespons
        return None
    except Exception as e:
        logging.error(f"An unexpected error occurred during LLM response
        return None


# --- Main Processing Logic ---

def main():
    """Main function to orchestrate the KG extraction process."""
    data = load_json_data(INPUT_JSON_PATH)
    if not data:
        return

    # Use sets for automatic deduplication
    entities_set: Set[Tuple[str, str]] = set()
    relations_set: Set[Tuple[str, str, str]] = set()
    chunk_map: Dict[str, Dict[str, Any]] = {chunk['chunk_id']: chunk for
    # Store normalized entities per chunk if needed for complex structur
    # chunk_entities: Dict[str, List[Tuple[str, str]]] = {}

    # --- Phase 1: LLM Extraction and Normalization ---
    for i, chunk in enumerate(data):
        chunk_id = chunk.get("chunk_id", f"unknown_{i}")
        content = chunk.get("content")
        heading = chunk.get("heading", "")

        if not content:
            logging.warning(f"Chunk {chunk_id} has no content. Skipping.
            continue
        logging.info(f"Processing chunk {i+1}/{len(data)}: ID {chunk_id}

        # 1. Extract Entities
        entity_prompt = create_entity_prompt(content)
        entity_response_text = call_llm(entity_prompt)
        parsed_entities = parse_llm_response(entity_response_text)

        current_chunk_norm_entities = []
        if parsed_entities and 'entities' in parsed_entities:
            for entity in parsed_entities['entities']:
                raw_name = entity.get('name')
                raw_type = entity.get('type')
                if raw_name and raw_type and raw_type in ENTITY_TYPES:
                    normalized_name = normalize_entity_name(raw_name)
                    entity_tuple = (normalized_name, raw_type)
                    entities_set.add(entity_tuple)
                    current_chunk_norm_entities.append(entity_tuple)
                else:
                    logging.warning(f"Invalid entity format or type in c
            # Store if needed later: chunk_entities[chunk_id] = current_

        # 2. Extract Relations (within chunk)
        relation_prompt = create_relation_prompt(content)
        relation_response_text = call_llm(relation_prompt)
        parsed_relations = parse_llm_response(relation_response_text)

        if parsed_relations and 'relations' in parsed_relations:
            for relation in parsed_relations['relations']:
                raw_source = relation.get('source')
                raw_target = relation.get('target')
                raw_type = relation.get('type')
                if raw_source and raw_target and raw_type and raw_type i
                    normalized_source = normalize_entity_name(raw_source
                    normalized_target = normalize_entity_name(raw_target
                    relation_tuple = (normalized_source, normalized_targ
                    relations_set.add(relation_tuple)
                else:
                     logging.warning(f"Invalid relation format or type i

    logging.info(f"Finished LLM extraction. Found {len(entities_set)} un

    # --- Phase 2: Add Structural Relations (BelongsTo) ---
    logging.info("Adding structural 'BelongsTo' relations...")
    structural_relations_added = 0
    for chunk in data:
        chunk_id = chunk.get("chunk_id")
        parent_id = chunk.get("parent_id")
        heading = chunk.get("heading")
        if parent_id and chunk_id and heading:
            parent_chunk = chunk_map.get(parent_id)
            if parent_chunk and parent_chunk.get("heading"):
                # Use normalized heading as the entity name for the chun
                # Assume chunks with headings map to '章节' or '文档' enti
                # A more robust method might check if these entities wer
                child_entity_name = normalize_entity_name(heading)
                parent_entity_name = normalize_entity_name(parent_chunk[

                # Add the BelongsTo relation
                # Add the entities themselves if not already present (as
                # Infer type crudely - root is Document, others are Sect
                child_type = "章节" # Default assumption for non-root hea
                parent_type = "章节" if parent_chunk.get("parent_id") els

                entities_set.add((child_entity_name, child_type))
                entities_set.add((parent_entity_name, parent_type))

                relation_tuple = (child_entity_name, parent_entity_name,
                if relation_tuple not in relations_set:
                    relations_set.add(relation_tuple)
                    structural_relations_added += 1

    logging.info(f"Added {structural_relations_added} structural 'Belong

    # --- Output Results ---
    print("\n--- Final Extracted Entities ---")
    # Sort for consistency
    sorted_entities = sorted(list(entities_set))
    for name, type in sorted_entities:
        print(f"- ({type}) {name}")
    print(f"\nTotal Unique Entities: {len(sorted_entities)}")

    print("\n--- Final Extracted Relations ---")
    sorted_relations = sorted(list(relations_set))
    for source, target, type in sorted_relations:
        print(f"- {source} --[{type}]--> {target}")
    print(f"\nTotal Unique Relations: {len(sorted_relations)}")


if __name__ == "__main__":
    main()


**How to Use:**

1.  **Save:** Save the code as a Python file (e.g., `kg_extractor.py`).
```

```
2.  **Install Libraries:** If you choose a specific LLM library (like `o
3.  **Configure:**
    *   Set the environment variable `YOUR_LLM_API_KEY` (or rename it in
    *   Update `LLM_MODEL` to the specific model you want to use.
    *   Ensure `INPUT_JSON_PATH` points to your `demo3.json` file.
    *   Review and expand the `CANONICAL_MAP` based on your document's s
4.  **Replace Placeholder:** Modify the `call_llm` function to use your
5.  **Run:** Execute the script from your terminal: `python kg_extractor
6.  **Review Output:** Check the printed lists of entities and relations

**Next Steps & Potential Improvements:**

1.  **Implement Real `call_llm`:** This is the most critical step.
2.  **Refine `CANONICAL_MAP`:** Add more mappings based on observed LLM
3.  **Improve Structural Relation Logic:** Make the mapping between chun
4.  **Error Handling:** Add more specific error handling around LLM call
5.  **Confidence Scores:** If your LLM provides confidence scores, you c
6.  **Advanced Normalization/Entity Linking:** Use libraries like `spaCy
7.  **Output Format:** Save results to CSV, JSONL, or directly into a gr
8.  **Batching/Async:** For large documents, consider batching chunks or

This code provides a solid starting point for extracting the knowledge g
```

user:

```
增加输出格式：
memgraph Cypher statements
```

assistant:

```
Okay, let's add the functionality to output the extracted knowledge grap

We'll add a new function `generate_cypher_statements` and call it at the


import json
import os
import logging
import time # For potential rate limiting
from typing import List, Dict, Set, Tuple, Optional, Any

# --- Configuration ---

# [ Existing configurations remain the same: LLM_API_KEY, LLM_MODEL, INP
# 1. LLM Configuration
LLM_API_KEY = os.environ.get("YOUR_LLM_API_KEY")
LLM_MODEL = "your-chosen-llm-model"
```

```
# 2. File Path
INPUT_JSON_PATH = "demo3.json"
OUTPUT_CYPHER_PATH = "output_graph.cypher" # Optional: File to save Cyph

# 3. Knowledge Graph Schema (Based on 改进建议.md)
ENTITY_TYPES = [
    "文档", "章节", "主题", "关键词", "人员", "角色",
    "组织", "时间", "事件", "法规"
]

# Map Chinese types to English for potentially better compatibility or p
ENTITY_TYPE_MAP_CYPHER = {
    "文档": "Document",
    "章节": "Section",
    "主题": "Topic",
    "关键词": "Keyword",
    "人员": "Person",
    "角色": "Role",
    "组织": "Organization",
    "时间": "Time",
    "事件": "Event",
    "法规": "Regulation"
}


RELATION_TYPES = [
    "隶属关系", "版本关系", "引用", "依据", "责任",
    "审批", "时间", "生效", "关联"
]

# Map Chinese types to English (often uppercase) for Cypher relationship
RELATION_TYPE_MAP_CYPHER = {
    "隶属关系": "BELONGS_TO",
    "版本关系": "HAS_VERSION",
    "引用": "REFERENCES",
    "依据": "BASED_ON",
    "责任": "RESPONSIBLE_FOR",
    "审批": "APPROVED_BY",
    "时间": "OCCURS_AT", # Consider if Time should be a node instead
    "生效": "EFFECTIVE_FROM", # Consider if Time should be a node instead
    "关联": "RELATED_TO"
}


# 4. Entity Normalization Map (Example - expand as needed)
CANONICAL_MAP = {
    "客运部": "集团公司客运部",
    "信息技术所": "集团公司信息技术所",
    "科信部": "集团公司科信部",
    "财务部": "集团公司财务部",
    "计统部": "集团公司计统部",
    "电务部": "集团公司电务部",
    "供电部": "集团公司供电部",
    "宣传部": "集团公司宣传部",
    "调度所": "集团公司调度所",
    "集团公司应急领导小组办公室": "集团公司应急领导小组办公室", # Already specifi
    "集团公司应急领导小组": "集团公司应急领导小组", # Already specific
    "国铁集团应急领导小组办公室": "国铁集团应急领导小组办公室",
    "国铁集团应急领导小组": "国铁集团应急领导小组",
    "国铁集团客运部": "国铁集团客运部",
    "12306科创中心": "12306科创中心",
    "广铁集团": "中国铁路广州局集团有限公司",
    "集团公司": "中国铁路广州局集团有限公司", # Be careful with context, might
    "本预案": "《广州局集团公司客票发售和预订系统（含互联网售票部分）应急预案》", #
    "《铁路客票发售和预订系统(含互联网售票部分)应急预案》": "《铁路客票发售和预订系统
    "《广州局集团公司网络安全事件应急预案》": "《广州局集团公司网络安全事件应急预案》
    "《广州局集团公司信息系统故障应急处置和调查处理办法》": "《广州局集团公司信息系统
    "客票系统": "客票发售和预订系统", # Normalize system name
}

# 5. Logging Configuration
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname


# --- Helper Functions ---

def load_json_data(file_path: str) -> Optional[List[Dict[str, Any]]]:
    """Loads data from a JSON file."""
    # [ Function code remains the same ]
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            data = json.load(f)
        logging.info(f"Successfully loaded data from {file_path}")
        return data
    except FileNotFoundError:
        logging.error(f"Error: Input file not found at {file_path}")
        return None
    except json.JSONDecodeError:
        logging.error(f"Error: Could not decode JSON from {file_path}")
        return None
    except Exception as e:
        logging.error(f"An unexpected error occurred during JSON loading
        return None
```

```
def normalize_entity_name(raw_name: str) -> str:
    """Normalizes entity names using the CANONICAL_MAP."""
    # [ Function code remains the same ]
    if not isinstance(raw_name, str):
        logging.warning(f"Attempted to normalize non-string value: {raw_
        return str(raw_name)
    cleaned_name = raw_name.strip().replace('\n', ' ')
    return CANONICAL_MAP.get(cleaned_name, cleaned_name)

def escape_cypher_string(value: str) -> str:
    """Escapes single quotes and backslashes for Cypher strings."""
    if not isinstance(value, str):
        return str(value) # Return as string if not already
    return value.replace('\\', '\\\\').replace("'", "\\'")

def create_entity_prompt(chunk_content: str) -> str:
    """Creates the prompt for entity extraction."""
    # [ Function code remains the same ]
    definitions = """
    实体类型定义：
    - 文档：管理规定的文件名称，如《应急预案》。
    - 章节：文档中的具体章节标题，如“1 总则”。
    - 主题：文档或章节的核心议题，如“应急组织机构”。
    - 关键词：文本中重要的名词或术语，如“客票系统”、“应急响应”、“电子客票”。
    - 人员：具体的人名（此文档中可能较少）。
    - 角色：指代具有特定职责的职位或岗位，如“客运部主任”、“售票员”。
    - 组织：涉及的单位、部门或公司，如“中国铁路广州局集团有限公司”、“集团公司客运部”、
    - 时间：具体的日期、时间点或时间段，如“2021年”、“4小时及以上”、“每年3月”。
    - 事件：文档中描述的具体活动或状况，如“系统突发事件”、“启动应急预案”、“应急演练”
    - 法规：引用的其他法规或文件名称及其编号，如“《铁路客票发售和预订系统(含互联网售票
    """
    prompt = f"""
请从以下文本中提取定义的实体类型。

{definitions}

预定义的实体类型列表：{', '.join(ENTITY_TYPES)}

文本：
\"\"\"
{chunk_content}
\"\"\"

请以严格的 JSON 格式输出，包含一个名为 "entities" 的列表，其中每个对象包含 "name"
例如：
{{
  "entities": [
    {{"name": "集团公司客运部", "type": "组织"}},
    {{"name": "售票故障", "type": "事件"}},
    {{"name": "《铁路客票发售和预订系统(含互联网售票部分)应急预案》（铁办客〔2021〕
  ]
}}
"""
    return prompt

def create_relation_prompt(chunk_content: str) -> str:
    """Creates the prompt for relation extraction (within the chunk)."""
    # [ Function code remains the same ]
    definitions = """
    关系类型定义 (请仅提取文本段落内明确描述的关系):
    - 隶属关系 (BelongsTo): 通常是结构化的，此提示词主要关注文本内描述，如“办公室设
    - 版本关系 (HasVersion): 指明文档的版本信息或与其他版本的关系 (如“修订版”、“废
    - 引用 (References): 一个实体提到了另一个实体或文件，如“详见附件5”。
    - 依据 (BasedOn): 指出制定某文件或采取某行动所依据的法规或原则，如“根据...制定
    - 责任 (ResponsibleFor): 指明某个角色或组织负责某项任务或职责，如“客运部负责协
    - 审批 (ApprovedBy): 指出某事项需要经过哪个组织或角色批准，如“经...同意后”。
    - 时间 (OccursAt): 事件发生的时间，或规定适用的时间点/段，如“事件影响4小时”、“
    - 生效 (EffectiveFrom): 规定或文件的生效日期，如“自发布之日起实施”。
    - 关联 (RelatedTo): 实体间的其他关联，如“与...不一致时，以此为准”。
    """
    prompt = f"""
请从以下文本中提取实体之间的关系。请专注于在文本段落中**直接陈述**的关系。

{definitions}

预定义的关系类型列表：{', '.join(RELATION_TYPES)}

文本:
\"\"\"
{chunk_content}
\"\"\"

请以严格的 JSON 格式输出，包含一个名为 "relations" 的列表，其中每个对象包含 "sourc
例如：
{{
  "relations": [
    {{"source": "集团公司应急领导小组办公室", "target": "集团公司客运部", "type
    {{"source": "本预案", "target": "《铁路客票发售和预订系统(含互联网售票部分)应
    {{"source": "客运部", "target": "协调各相关部门", "type": "责任"}}
  ]
}}
"""
    return prompt



# --- LLM Interaction (Placeholder) ---
```

```
def call_llm(prompt: str) -> Optional[str]:
    """
    Placeholder function to call the LLM API.
    Replace this with your actual LLM API call logic.
    """
    # [ Function code remains largely the same, use your actual LLM call
    logging.info(f"--- Sending Prompt to LLM (Length: {len(prompt)}) ---
    logging.info("--- Waiting for LLM Response ---")

    # !! --- REPLACE THIS SECTION with your actual LLM call --- !!
    # Simulate a response for testing
    if "提取定义的实体类型" in prompt:
         logging.warning("Using PLACEHOLDER response for entity extracti
         if "集团公司成立客票系统应急领导小组" in prompt:
             return json.dumps({ "entities": [ {"name": "集团公司", "type"
         else: return json.dumps({"entities": []})
    elif "提取实体之间的关系" in prompt:
        logging.warning("Using PLACEHOLDER response for relation extract
        if "集团公司成立客票系统应急领导小组" in prompt:
            return json.dumps({ "relations": [ {"source": "集团公司应急领导
        else: return json.dumps({"relations": []})
    else: return None
    # !! --- END REPLACE/REMOVE --- !!


def parse_llm_response(response_text: Optional[str]) -> Optional[Dict[st
    """Safely parses the LLM's JSON response."""
    # [ Function code remains the same ]
    if not response_text: return None
    try:
        if response_text.strip().startswith(""):
            response_text = response_text.strip()[7:-3].strip()
        elif response_text.strip().startswith("`"):
             response_text = response_text.strip()[1:-1].strip()
        parsed_data = json.loads(response_text)
        if isinstance(parsed_data, dict) and \
           (('entities' in parsed_data and isinstance(parsed_data['entit
            ('relations' in parsed_data and isinstance(parsed_data['rela
            return parsed_data
        else:
            logging.warning(f"LLM response is valid JSON but not the exp
            return None
    except json.JSONDecodeError as e:
        logging.error(f"Failed to decode LLM JSON response: {e}\nRespons
        return None
    except Exception as e:
        logging.error(f"An unexpected error occurred during LLM response
        return None

# --- Cypher Generation ---

def generate_cypher_statements(entities: Set[Tuple[str, str]], relations
    """Generates Memgraph/Neo4j Cypher MERGE statements."""
    cypher_statements = []

    # Add constraint for uniqueness if desired (recommended)
    # You might want to run these manually once or ensure they exist
    # for entity_type_cn in ENTITY_TYPES:
    #     entity_type_cypher = ENTITY_TYPE_MAP_CYPHER.get(entity_type_cn
    #     cypher_statements.append(f"CREATE CONSTRAINT ON (n:{entity_typ

    cypher_statements.append("\n// --- Entity Creation ---")
    sorted_entities = sorted(list(entities))
    for name, type_cn in sorted_entities:
        if not name: # Skip empty names
            continue
        entity_type_cypher = ENTITY_TYPE_MAP_CYPHER.get(type_cn, type_cn
        escaped_name = escape_cypher_string(name)
        # Use MERGE to avoid duplicates
        cypher_statements.append(f"MERGE (:`{entity_type_cypher}` {{name

    cypher_statements.append("\n// --- Relationship Creation ---")
    sorted_relations = sorted(list(relations))
    for source, target, type_cn in sorted_relations:
        if not source or not target: # Skip if source or target is missi
            continue
        relation_type_cypher = RELATION_TYPE_MAP_CYPHER.get(type_cn, typ
        escaped_source = escape_cypher_string(source)
        escaped_target = escape_cypher_string(target)

        # Find the types of source and target from the entities set for
        source_type_cn = next((t for n, t in sorted_entities if n == sou
        target_type_cn = next((t for n, t in sorted_entities if n == tar

        if source_type_cn and target_type_cn:
            source_type_cypher = ENTITY_TYPE_MAP_CYPHER.get(source_type_
            target_type_cypher = ENTITY_TYPE_MAP_CYPHER.get(target_type_
            # Use MERGE for relationships too
            cypher_statements.append(
                f"MATCH (a:`{source_type_cypher}` {{name: '{escaped_sour
                f"MERGE (a)-[:`{relation_type_cypher}`]->(b);"
            )
        else:
             # Fallback if type not found (less safe, might match wrong
             logging.warning(f"Could not determine types for relationshi
             cypher_statements.append(
                 f"MATCH (a {{name: '{escaped_source}'}}), (b {{name: '{
                 f"MERGE (a)-[:`{relation_type_cypher}`]->(b);"
             )

    return cypher_statements


# --- Main Processing Logic ---

def main():
    """Main function to orchestrate the KG extraction process."""
    data = load_json_data(INPUT_JSON_PATH)
    if not data:
        return

    entities_set: Set[Tuple[str, str]] = set()
    relations_set: Set[Tuple[str, str, str]] = set()
    chunk_map: Dict[str, Dict[str, Any]] = {chunk['chunk_id']: chunk for

    # --- Phase 1: LLM Extraction and Normalization ---
    # [ This loop remains the same as before ]
    for i, chunk in enumerate(data):
        chunk_id = chunk.get("chunk_id", f"unknown_{i}")
        content = chunk.get("content")
        heading = chunk.get("heading", "")
        if not content: continue
        logging.info(f"Processing chunk {i+1}/{len(data)}: ID {chunk_id}

        # Extract Entities
        entity_prompt = create_entity_prompt(content)
        entity_response_text = call_llm(entity_prompt)
        parsed_entities = parse_llm_response(entity_response_text)
        if parsed_entities and 'entities' in parsed_entities:
            for entity in parsed_entities['entities']:
                raw_name = entity.get('name')
                raw_type = entity.get('type')
                if raw_name and raw_type and raw_type in ENTITY_TYPES:
                    normalized_name = normalize_entity_name(raw_name)
                    entities_set.add((normalized_name, raw_type))
                else: logging.warning(f"Invalid entity format or type in

        # Extract Relations
        relation_prompt = create_relation_prompt(content)
        relation_response_text = call_llm(relation_prompt)
        parsed_relations = parse_llm_response(relation_response_text)
        if parsed_relations and 'relations' in parsed_relations:
            for relation in parsed_relations['relations']:
```

```
            raw_source = relation.get('source')
            raw_target = relation.get('target')
            raw_type = relation.get('type')
            if raw_source and raw_target and raw_type and raw_type i
                normalized_source = normalize_entity_name(raw_source
                normalized_target = normalize_entity_name(raw_target
                relations_set.add((normalized_source, normalized_tar
            else: logging.warning(f"Invalid relation format or type 

logging.info(f"Finished LLM extraction. Found {len(entities_set)} un

# --- Phase 2: Add Structural Relations (BelongsTo) ---
# [ This loop remains the same as before ]
logging.info("Adding structural 'BelongsTo' relations...")
structural_relations_added = 0
for chunk in data:
    chunk_id = chunk.get("chunk_id")
    parent_id = chunk.get("parent_id")
    heading = chunk.get("heading")
    if parent_id and chunk_id and heading:
        parent_chunk = chunk_map.get(parent_id)
        if parent_chunk and parent_chunk.get("heading"):
            child_entity_name = normalize_entity_name(heading)
            parent_entity_name = normalize_entity_name(parent_chunk[
            if child_entity_name and parent_entity_name: # Ensure na
                child_type = "章节"
                parent_type = "章节" if parent_chunk.get("parent_id")
                entities_set.add((child_entity_name, child_type))
                entities_set.add((parent_entity_name, parent_type))
                relation_tuple = (child_entity_name, parent_entity_n
                if relation_tuple not in relations_set:
                    relations_set.add(relation_tuple)
                    structural_relations_added += 1
logging.info(f"Added {structural_relations_added} structural 'Belong


# --- Output Results ---

# 1. Console Output (Original)
print("\n--- Final Extracted Entities (Console) ---")
sorted_entities = sorted(list(entities_set))
for name, type in sorted_entities:
    print(f"- ({type}) {name}")
print(f"\nTotal Unique Entities: {len(sorted_entities)}")

print("\n--- Final Extracted Relations (Console) ---")
sorted_relations = sorted(list(relations_set))
for source, target, type in sorted_relations:
        print(f"- {source} --[{type}]--> {target}")
    print(f"\nTotal Unique Relations: {len(sorted_relations)}")

    # 2. Cypher Statement Generation and Output
    print(f"\n--- Generating Cypher Statements (Memgraph/Neo4j) ---")
    cypher_statements = generate_cypher_statements(entities_set, relatio

    # Option A: Print to console
    # print("\n".join(cypher_statements))

    # Option B: Save to file
    try:
        with open(OUTPUT_CYPHER_PATH, 'w', encoding='utf-8') as f:
            f.write(";\n".join(cypher_statements) + ";\n") # Add semicol
        print(f"\nCypher statements saved to: {OUTPUT_CYPHER_PATH}")
    except IOError as e:
        print(f"\nError writing Cypher statements to file {OUTPUT_CYPHER_
        print("\nCypher Statements:\n")
        print(";\n".join(cypher_statements) + ";\n") # Print to console 


if __name__ == "__main__":
    main()



**Changes Made:**

1.  **Cypher Configuration:** Added `OUTPUT_CYPHER_PATH`, `ENTITY_TYPE_M
2.  **`escape_cypher_string` Function:** Added a helper to escape single
3.  **`generate_cypher_statements` Function:**
    *   Takes the final `entities_set` and `relations_set` as input.
    *   Iterates through entities, generating `MERGE (:<EntityType> {name
    *   Iterates through relations, generating `MATCH (a:<SourceType> {n
    *   Uses the `escape_cypher_string` function for all names.
    *   Uses backticks (`) around labels and relationship types just in 
4.  **`main` Function Integration:**
    *   Calls `generate_cypher_statements` after all entities and relati
    *   Includes logic to either print the Cypher statements to the cons

Now, when you run the script (after replacing the placeholder LLM call),
```

Chatbox AI