**HyperEventGraph: 一个用于构建自我进化事理图谱的系统化方法**

**HyperEventGraph: A System for Constructing Self-Evolving, Domain-Specific Event Knowledge Graphs**

**版本**: 1.0 (Initial Draft for Academic Review) **负责人**: Gemini Architect

**摘要 (Abstract)**

在信息爆炸的背景下，从海量的、非结构化的文本数据中自动抽取出结构化知识，并揭示事件之间潜在的深层逻辑关系，是前沿研究和应用领域中的一个核心挑战。传统方法往往局限于静态的知识抽取，难以形成能够自我演化的智能系统。本文介绍了一个名为 HyperEventGraph 的新型事理图谱构建系统。该系统旨在构建一个能够“自我完善、持续迭代”的“活”的知识系统。为实现此目标，HyperEventGraph 采用了一系列核心设计原则，包括\*\*状态驱动 (State-Driven)**的架构、人机协同 (Human-in-the-Loop) 的工作流、以及**分层存储 (Layered Storage)**策略。系统的核心创新在于其**知识闭环 (Knowledge Loop)\*\*机制，该机制通过 **Cortex 引擎**进行上下文重建，并利用 **HybridRetrieverAgent** 进行知识增强的分析，从而使系统能够持续学习与进化。本文将详细阐述该系统的设计哲学、分层架构和核心方法论，展示其在自动化知识构建和深度关联分析方面的潜力。

In the information era, automatically extracting structured knowledge and latent relationships from massive, unstructured text is a primary challenge. This is particularly true within specific, knowledge-intensive domains. This paper introduces HyperEventGraph, a novel system for constructing domain-specific event knowledge graphs designed to be "self-improving and continuously iterative". To achieve this, HyperEventGraph employs a state-driven architecture, a human-in-the-loop workflow, and a layered storage strategy. The system's core innovation is its "Knowledge Loop", which uses a **Cortex Engine** for context reconstruction and a **HybridRetrieverAgent** for knowledge-enhanced analysis, enabling continuous learning. While the framework is domain-agnostic, this paper uses the **integrated circuit (IC) industry** as a running example to illustrate its capabilities in automated knowledge construction and deep relational analysis.

**1. 引言 (Introduction)**

**1.1 背景与动机 (Background and Motivation)**

当今时代，决策者和分析师面临着前所未有的信息处理压力，他们需要从新闻资讯、公司公告、研究报告等海量的、非结构化的文本源中获取有价值的洞察。然而，传统的关键词搜索或基础的实体识别技术，虽然能够定位到孤立的信息点，却难以揭示这些信息点之间复杂的、隐藏的逻辑关联，如因果、影响或时序关系。这种“见树不见林”的信息处理模式，已成为智能决策的主要瓶颈。因此，市场迫切需要一种能够超越关键词搜索，实现对事件间深度关联进行分析的先进技术。

**1.2 问题陈述与系统愿景 (Problem Statement and System Vision)**

为应对上述挑战，我们提出需要解决的核心问题是：如何设计并实现一个能够将非结构化文本自动转化为结构化、可分析，并且能够持续学习和进化的知识图谱系统？

针对此问题，我们设计的 HyperEventGraph 系统的核心愿景是构建一个“活”的知识系统。该系统的最终形态是一个智能决策支持引擎，它具备以下三个层面的能力：

* **自动化知识构建 (Automated Knowledge Construction)**: 将海量的、非结构化的行业资讯自动转化为结构化的、可分析的知识图谱。
* **深度关联分析 (Deep Relationship Analysis)**: 超越关键词搜索，揭示事件之间深层的、隐藏的逻辑关系（如因果、影响、时序等）。
* **持续学习与进化 (Continuous Learning and Evolution)**: 通过内置的学习工作流和知识反馈闭环，不断适应新的领域知识和事件模式，实现知识边界的自适应扩张。

**1.3 范围与应用领域 (Scope and Application Domain)**

The HyperEventGraph framework is designed to be domain-agnostic. Its core mechanisms, such as the state-driven workflow and the SchemaLearnerAgent that discovers new event types from unclassified examples, are universally applicable.

However, for practical application and demonstration, the system's power is best illustrated when grounded in a specific, complex domain. The richness of the resulting knowledge graph is directly proportional to the depth and nuances of the chosen field. To provide concrete examples and demonstrate the system's efficacy, this paper will use the integrated circuit (IC) industry as its primary application domain. This domain is ideal due to its complex entity relationships (design houses, foundries, equipment suppliers), high volume of impactful events (e.g., new product launches, patent litigation, factory fires), and the critical need for timely, interconnected intelligence.

**2. 系统架构与设计哲学 (System Architecture and Design Philosophy)**

为实现上述宏大愿景，系统的构建遵循了一套明确的设计哲学，并在此基础上设计了分层化的系统架构。

**2.1 设计哲学 (Design Philosophy)**

* **状态驱动 (State-Driven)**: 系统的所有核心工作流均由一个中央状态数据库驱动。这种设计使得各个功能模块高度解耦、无状态化，从而保证了系统的健壮性、可扩展性和任务的断点续传能力。
* **人机协同 (Human-in-the-Loop)**: 我们承认当前人工智能技术的局限性，因此将人类专家的智慧无缝集成到工作流中。通过高效的审核与交互式学习界面，确保关键知识的准确性，并由专家引导系统的学习方向。
* **分层存储 (Layered Storage)**: 针对不同的知识检索需求，系统采用了图数据库（用于精确的结构化查询）和向量数据库（用于模糊的语义化查询）的混合存储策略，以实现更强大的知识服务能力。
* **模块化与可配置 (Modular & Configurable)**: 所有核心功能被封装为独立的智能代理 (Agents) 和工作流 (Workflows)，所有关键参数由全局配置文件统一管理，极大地提升了系统的可维护性和部署灵活性。

**2.2 分层架构 (Layered Architecture)**

本系统采用自下而上的分层架构设计，确保了模块间的低耦合与流程的清晰性。

* **数据源层 (Data Source Layer)**: 系统的输入端，负责从新闻网站 (HTML)、公司公告 (PDF) 等多样化渠道获取原始、非结构化的文本数据。
* **数据处理与存储层 (Data Processing & Storage Layer)**: 数据的“枢纽站”，负责对原始数据进行解析、清洗、管理和持久化存储。此层的核心是“一个中央状态库，两个核心知识库”：
  + **中央状态数据库 (SQLite)**: 作为系统的“中央神经系统”，统一管理和追踪每一条数据在处理流程中的状态 (master\_state.db)。
  + **图数据库 (Neo4j)**: 存储最终的结构化知识，包括事件节点和它们之间的复杂关系。
  + **向量数据库 (ChromaDB)**: 存储事件的语义向量表示，以支持高效的语义相似度检索。
* **核心技术与分析层 (Core Technology & Analysis Layer)**: 系统的“智能核心”，负责所有复杂的分析、抽取、学习和推理任务。它由智能代理 (Agents)、核心工作流 (Workflows) 和核心技术引擎 (Engines) 构成。
* **应用与服务层 (Application & Service Layer)**: 系统的“价值出口”，直接面向最终用户或外部系统提供服务，如智能化知识问答、行业趋势分析等。

**3. 核心组件与技术实现 (Core Components and Technical Implementation)**

在第2章介绍了系统的高层分层架构后，本章将深入剖析构成系统基石的几个核心基础设施组件的技术实现。这些组件是系统实现状态驱动、模块化和可配置等设计哲学的关键。

**3.1 中央状态数据库管理器 (DatabaseManager)**

如架构文档所述，master\_state.db 是整个系统的“中央神经系统”。DatabaseManager 则是与该数据库交互的唯一、统一的接口，它封装了所有数据库操作，确保了数据访问的一致性和安全性。

* **设计与职责**: DatabaseManager 严格遵循单一职责原则，仅负责数据库的CRUD（创建、读取、更新、删除）操作，不包含任何业务逻辑。这种设计将数据存取与业务流程解耦，提升了代码的可维护性。
* **关键技术实现**:
  + **动态的模式演进 (Dynamic Schema Evolution)**: 这是该模块一个非常关键的健壮性设计。在初始化时，DatabaseManager 不仅会确保核心表 master\_state 的存在，还会通过 PRAGMA table\_info 查询现有列，并与预定义模式进行比对。如果发现有新的字段（例如在系统迭代中增加了 story\_id 列），它会自动执行 ALTER TABLE ... ADD COLUMN ... 来动态地更新表结构。这一机制确保了数据库结构的向后兼容，使得系统升级过程平滑，无需手动干预数据库。
  + **高效的任务认领**: 各个工作流通过调用 get\_records\_by\_status\_as\_df(status: str) 方法来“认领”待处理的任务。该方法利用 pandas.read\_sql\_query 将特定状态的所有记录高效地读取为DataFrame，极大地便利了上层模块进行批处理操作。
  + **高性能的批量更新**: 在处理像“故事单元(Story)”这样涉及多个事件的场景时，update\_story\_info 方法通过 WHERE id IN (...) 的SQL语法，实现对多个记录的批量更新。这种方式避免了逐条更新数据库带来的性能瓶颈，是系统处理高吞吐量数据流的重要保障。

**3.2 提示词管理器 (PromptManager)**

PromptManager 的核心任务是实现提示词工程（Prompt Engineering）的一个核心原则：将提示词（Prompts）作为一种可配置的、与业务逻辑分离的资产进行管理。

* **设计与实现**:
  + **单例模式 (Singleton Pattern)**: 在整个应用程序的生命周期中，只存在一个 PromptManager 实例。这确保了所有的提示词模板只需从磁盘加载一次，有效提升了性能并节约了内存资源。
  + **内置缓存机制**: 该管理器内部实现了一个缓存 (self.template\_cache)。当一个提示词模板被首次请求时，它会从磁盘加载并存入缓存；后续对同一模板的请求将直接从内存中快速返回，避免了不必要的磁盘I/O。
  + **模板化与参数注入**: get\_prompt 方法是外部模块调用的主要接口。它接收模板名称和一组关键字参数，然后安全地将这些参数填充到模板的占位符中（如 {text\_sample}），最终生成一个可供LLM使用的完整提示词。

**3.3 全局配置加载器 (ConfigLoader)**

ConfigLoader 负责解析 config.yaml 文件，并将所有配置项加载为全局可访问的对象。这一组件是实现系统“模块化与可配置”设计哲学的基石。

* **核心价值**: 它将所有可变的参数，如文件路径、模型名称、API密钥路径、以及算法超参数（如DBSCAN的eps值），从业务逻辑代码中完全分离出来。这使得研究人员或运维人员可以在不修改任何Python代码的情况下，通过简单地修改config.yaml文件来调整系统的行为、切换模型或更新参数，极大地提升了实验效率和部署灵活性。

**3.4 LLM 统一接口引擎 (LLMClient)**

LLMClient 是系统与大语言模型（LLM）之间唯一的通信桥梁。它的核心设计目标是将系统内部的业务逻辑（主要在智能代理 Agents 中）与底层具体的LLM供应商和模型完全解耦，从而将复杂的LLM调用抽象为一个简单、可靠的服务。

* **设计原则与核心功能**:
  + **统一入口与配置驱动**: 系统中所有对LLM的调用都必须通过 LLMClient 这个单一入口。该客户端的行为完全由 config.yaml 文件驱动，不硬编码任何模型名称或API地址。这使得研究人员可以仅通过修改配置文件就轻松切换模型或更换供应商，无需改动任何业务代码。
  + **任务路由机制 (Task-based Routing)**: LLMClient 实现了一种精细化的任务路由功能。调用者在请求时需提供一个 task\_type（如 'triage' 或 'extraction'）。客户端会根据此类型从配置文件中查找并使用最适合该任务的模型及其特定参数（如 temperature, max\_tokens）。这使得系统可以为不同任务（如快速初筛和高精度分析）分配不同成本和能力的模型，从而优化整体性能和成本效益。
  + **健壮性与可追溯性**: 为保证系统稳定性，所有LLM API调用都被包裹在 try-except 块中。LLMClient 会对需要JSON输出的任务进行安全的解析，若解析失败或API调用出错，它会记录详细的错误日志（包括LLM返回的原始数据）并返回一个结构合法的默认值，从而防止上游的工作流因API问题而崩溃。
  + **异步高并发支持**: 为了支持高吞吐量的处理工作流（如 run\_extraction\_workflow.py），LLMClient 提供了基于 asyncio 的异步调用方法。这允许系统同时处理多个非阻塞的LLM请求，极大地提升了事件抽取的整体效率。
* **典型应用模式**: 在智能代理（Agent）中，首先通过 PromptManager 获取格式化的提示词，然后将提示词和任务类型传递给 LLMClient 的实例，最后处理返回的（通常是JSON格式的）结果。整个过程清晰地分离了提示词管理、LLM通信和业务逻辑处理三个环节。

**4. 智能代理军团 (The Intelligent Agent Corps)**

系统的“智能”并非由一个单一的、庞大的模型体现，而是由一系列各司其职、协同工作的“智能代理”所构成。每个代理都是一个由LLM驱动的、为特定认知任务而设计的组件。本章将详细介绍系统中四个关键的智能代理，它们分别负责数据处理、知识深化和系统学习等核心环节。

**4.1 前端处理代理 (Frontline Processing Agents)**

这两个代理构成了数据从非结构化到结构化的主要流水线。

**4.1.1 TriageAgent: 初步分类器**

TriageAgent 是数据处理流水线的第一个智能决策点，负责对海量原始文本进行快速、低成本的初步分类。

* **核心职责**: 其唯一职责是判断文本属于“已知事件”还是“未知事件”，并给出相应的置信度。它不负责提取具体信息，这保证了其任务的简单性和高效率。
* **关键设计 - 动态适应性**: 该代理的认知范围（即它认识的事件类型）并非硬编码。其系统提示是**动态构建**的，在初始化时会自动从一个中央的事件模式注册表 (EVENT\_SCHEMA\_REGISTRY) 中加载所有已知的事件类型列表。这意味着，当系统通过 SchemaLearnerAgent 学会一个新事件类型后，TriageAgent 无需任何代码修改，就能在下一次运行时自动获得识别该新事件的能力。
* **输出**: 其输出被严格约束为一个纯净的JSON对象，包含status, event\_type, domain 和 confidence 等字段，便于上游工作流进行可靠的解析和处理。

**4.1.2 ExtractionAgent: Schema驱动的抽取器**

在事件类型被确定后，ExtractionAgent 负责执行精准、深入的信息提取，是将非结构化文本转化为结构化知识的核心执行者。

* **核心设计 - Schema驱动**: 这是该代理最核心的设计思想。它的行为完全由调用时传入的**JSON Schema**来定义和驱动。其系统提示被设计为极度强调对所提供Schema的**严格遵从**，确保输出的JSON对象能够通过该Schema的验证。
* **灵活性与可扩展性**: 正是由于其Schema驱动的特性，ExtractionAgent 具备极高的灵活性。系统无需修改代理的任何代码，只要为其提供一个新的JSON Schema，它就能立即学会如何抽取一种全新的事件类型。如果原始文本中未找到符合Schema的事件，它会稳定地输出一个空列表 []，保证了下游处理的稳定性。

**4.2 高级认知代理 (Advanced Cognitive Agents)**

这两个代理负责更复杂的认知任务，包括知识的深化连接和系统的自我学习。

**4.2.1 RelationshipAnalysisAgent: 知识的织网者**

该代理是实现系统从“点状事件”到“网状知识”跨越的核心，负责挖掘同一个“故事单元”内多个事件之间深层的、隐含的逻辑关联。

* **核心设计 - 上下文依赖**: 关系分析的质量高度依赖于上下文的丰富度。因此，该代理的提示词被精心设计为可以接收三个层次的上下文信息：
  1. **事件层**: 待分析的事件描述列表。
  2. **文档层**: 产生这些事件的原始文本。
  3. **知识库层**: 由 HybridRetrieverAgent 提供的、与当前事件相关的历史背景摘要，这是实现“知识闭环”的关键一环。
* **结构化输出与可扩展性**: 代理的输出被严格定义为一个JSON列表，其中每个对象都清晰地定义了源事件、目标事件、关系类型（如 Causal, Temporal）和解释，可以直接存入图数据库。关系类型的定义明确列在Prompt中，使得未来调整关系体系时，只需修改Prompt模板即可。

**4.2.2 SchemaLearnerAgent: 知识的拓展者**

SchemaLearnerAgent 体现了系统的“成长”能力，它与人类专家协作，从完全未知的数据中发现新的事件模式，并将其形式化为系统可理解的JSON Schema。

* **核心设计 - 人机协同与工具驱动**:
  + **人机协同**: 它的设计完全围绕与人类专家的交互展开，不进行任何全自动决策。代理负责分析（如使用聚类算法对未知事件进行分组）并将结果呈现给专家，由专家进行最终的判断、修正和确认。
  + **工具驱动**: 遵循现代Agent设计范式，其核心能力被封装在 SchemaLearningToolkit 中。其中最关键的工具 induce\_schema 会要求LLM扮演“数据架构师”的角色，从一组相似的事件样本中**归纳**出一个通用的JSON Schema。
* **学习闭环**: 该代理是系统第一个知识闭环（学习回路）的核心。当专家批准一个新Schema后，相关工作流会将此前被标记为 pending\_learning 的数据状态**重置为 pending\_triage**。这使得这些数据能够被系统用新的“知识视角”重新识别和处理，完成了知识的增量学习。

**5. 端到端工作流：一篇新闻稿的生命周期 (End-to-End Workflow: The Lifecycle of a News Article)**

前几章详细介绍了系统的静态组件，本章将通过追踪一条数据在系统中的完整生命周期，将这些组件串联起来，展示它们在一个动态的、端到端的流程中如何协同工作。这些流程由一系列 run\_\*.py 脚本进行编排。

为具象化展示前述章节中各个组件的协同作用，本章将以一篇关于**集成电路**领域的模拟新闻稿为例，完整追踪其数据在系统中的端到端生命周期。

**5.1 场景定义：新闻稿原文**

“《科创板日报》24日讯，芯片巨头‘星辰半导体’今日宣布，受全球供应链持续紧张影响，其旗舰AI芯片‘启明A100’的交付将延迟至第四季度。该公司CEO李明表示，为应对此问题，他们已紧急启动与‘环宇晶圆’的产能扩张合作。分析师认为，此举虽无法立即解决问题，但长期看好。受此消息影响，‘星辰半导体’的主要客户‘未来汽车’的股价应声下跌5%。”

**5.2 阶段一：注入与初步处理**

1. **数据注入**: 系统获取原文后，进行清洗，并计算其哈希值 hash\_xyz 作为唯一ID，存入 master\_state.db，初始状态为 **pending\_triage**。
2. **批量初筛**: run\_batch\_triage.py 工作流启动。TriageAgent (4.1.1) 被调用，其 triage.md 提示词已被动态填充了系统当前所有已知的事件类型。LLM返回分类结果，例如 {"event\_type": "SupplyChainDisruption", "confidence": 0.92}。数据库记录随之更新，状态变为 **pending\_review**。

**5.3 阶段二：人机协同与事件抽取**

1. **质量门**: prepare\_review\_file.py 将待审核记录生成清单，由领域专家进行审核。专家确认分类无误后，process\_review\_results.py 将记录状态更新为 **pending\_extraction**。
2. **并发抽取**: run\_extraction\_workflow.py 工作流启动，高并发地处理所有待抽取记录。ExtractionAgent (4.1.2) 根据记录的事件类型 (SupplyChainDisruption)，获得对应的JSON Schema，并依据 extraction.md 提示词的指令，从原文中抽取出多个结构化事件（如“交付延迟”、“产能扩张合作”、“股价下跌”）。抽取成功后，记录状态更新为 **pending\_clustering**。

**5.4 阶段三：Cortex引擎与上下文重建**

1. **自动触发**: 当 pending\_clustering 状态的事件数达到阈值，run\_cortex\_workflow.py 被自动触发。
2. **上下文重建**: Cortex引擎 (第5.2节详述) 开始执行其两阶段流程。
   * **混合距离聚类**: ClusteringOrchestrator 开始工作。由于从 hash\_xyz 中抽取的3个事件共享“星辰半导体”等核心实体且语义相关，它们极有可能通过**混合距离DBSCAN算法**被分到同一个簇中（例如 cluster\_id: 17）。
   * **LLM精炼**: RefinementAgent 接收该簇，调用LLM将其提炼成一个连贯的“故事摘要”，并为之分配一个唯一的 story\_id（例如 story\_abc）。数据库中这3个事件的记录被批量更新，状态变为 **pending\_relationship\_analysis**。

**5.5 阶段四：知识闭环与最终存储**

1. **知识增强分析**: run\_relationship\_analysis.py 工作流启动，处理 story\_abc。
   * **知识检索 (闭环)**: HybridRetrieverAgent 首先被调用。它提取故事中的核心实体（“星辰半导体”），并并行查询Neo4j和ChromaDB，寻找相关的历史事件（例如，上一季度“启明A100”芯片的发布事件）。
   * **增强分析**: RelationshipAnalysisAgent (4.2.1) 被调用。其提示词（relationship\_analysis.md）中此时注入了三个层次的上下文：当前故事的3个事件、新闻稿原文、以及检索到的历史背景摘要。在这个极其丰富的上下文中，LLM能够轻松推理出事件间的深层关系，如 (交付延迟) --Causal--> (股价下跌)。
2. **知识存储**: StorageAgent 被调用，将新的事件节点、实体节点以及分析出的 :CAUSAL 等关系边存入双数据库。记录的最终状态被更新为 **completed**。

**7. 系统能力与应用场景 (System Capabilities and Application Scenarios)**

HyperEventGraph的最终价值体现在其将非结构化信息转化为可操作智能的能力上。

**7.1 核心能力总结**

* **自动化知识构建**: 实现了从原始文本到结构化、深度关联的知识图谱的端到端自动化处理流程。
* **深度关系发现**: 通过Cortex引擎的上下文重建和知识增强的关系分析，能够发现超越文本表面的因果、影响等深层逻辑。
* **持续学习进化**: 通过人机协同的学习闭环，系统能够不断扩展其知识边界，识别和理解新的事件类型。

**7.2 应用场景示例：智能化知识问答**

该场景直接展示了系统构建完成后的应用价值。

* **用户提问**: 一周后，一位分析师通过系统接口提问：“‘星辰半导体’的供应链问题对它的客户有什么影响？”。
* **系统响应**:
  1. **混合检索**: HybridRetrieverAgent 接收到提问。它不再是简单的文本匹配，而是在Neo4j中进行图遍历。它从“星辰半导体”节点出发，沿着 :INVOLVES 关系找到“交付延迟”事件，再沿着我们刚刚创建的 :CAUSAL 关系链，轻松地找到了“股价下跌”事件，该事件又通过 :INVOLVES 关系关联到了其实体客户“未来汽车”。
  2. **答案生成**: 检索到的这条包含精确因果链的知识被格式化后，连同用户问题一起发送给LLM进行自然语言生成。
  3. **最终输出**: 系统能够返回一个高质量、有理有据的回答：“根据知识库，‘星辰半导体’的‘启明A100’芯片交付延迟，直接导致了其客户‘未来汽车’的股价下跌5%。”。

**8. 结论与展望 (Conclusion and Future Work)**

本文详细阐述了一个用于构建自我进化、领域特定事理图谱的新型系统——HyperEventGraph。我们系统性地介绍了其状态驱动的架构、分层解耦的核心组件、由智能代理驱动的业务流程，并通过一个端到端的示例展示了其从原始文本中提取、关联、并最终应用知识的全过程。本研究的核心贡献在于提供了一套完整的、将上下文重建和知识闭环机制深度融合的系统化方法论，有效地提升了自动化知识图谱构建的深度和准确性。

未来的工作将聚焦于两个方向：一是开发更先进、可视化的Web交互界面，以降低人机协同的门槛；二是在更多复杂领域（如生物医药、法律文书分析）验证和扩展该框架的适用性。

**【作为附录】6. 核心方法论：提示词工程 (Prompt Engineering as a Core Discipline)**

在HyperEventGraph中，提示词被视为一种特殊的、可维护的“源代码”，是确保各个智能代理能够稳定、可靠执行任务的基石。所有提示词由 PromptManager (3.2) 统一管理。

* **triage.md (初筛)**: 设计上追求**简洁、高效、强约束**。通过大写的绝对指令（You MUST output ONLY a valid JSON object）和格式示例来减少LLM输出的随机性。其关键在于通过模板变量注入系统当前已知的所有事件类型，使提示词能随系统学习而“进化”。
* **extraction.md (抽取)**: 设计理念为**Schema驱动、事实优先**。它通过Markdown表格向LLM清晰地展示待抽取的字段，并用一个醒目的警告块（🚫 绝对禁止抽取预测类...）来保证知识库的事实性。其最有效的技术是提供了一个高质量的“输入->输出”Few-shot示例，向LLM示范了如何处理包含预测信息的混合文本。
* **relationship\_analysis.md (关系分析)**: 设计上强调**上下文感知和可解释性**。它将知识库、文档和事件三个层次的上下文全部注入Prompt。它不仅列出了合法的关系类型，还为每一种都提供了清晰的定义和使用条件，并强制要求LLM在输出中包含 reason 字段，以增强结果的可解释性。
* **schema\_generation.md (Schema生成)**: 旨在引导LLM进行**归纳推理**。它为LLM设定了“数据架构师”的角色，并给出了清晰的命名规范（PascalCase, snake\_case），以确保系统内部Schema定义的一致性。

================================

=== 以防万一 保留一下暂时用不上的内容===

**5.1 V3.1 核心流水线：从文本到结构化事件**

这是系统最基础的数据处理流程，负责将原始文本转化为离散的、结构化的事件数据。

1. **批量初筛 (run\_batch\_triage.py)**: 工作流首先查询数据库中所有状态为 pending\_triage 的记录。TriageAgent (4.1.1) 被调用，对每条记录进行快速分类，并将结果（事件类型、置信度）连同新状态 pending\_review 一并更新回数据库。
2. **人机协同审核**: 在此步骤中，人类专家对AI的分类结果进行审核。审核通过后，记录状态被更新为 pending\_extraction。如果专家判断这是一个全新的事件类型，则状态更新为 pending\_learning，进入学习工作流 (5.3)。
3. **并发事件抽取 (run\_extraction\_workflow.py)**: 该工作流处理所有状态为 pending\_extraction 的记录。其核心特点是**高并发**，它利用 asyncio 创建一个异步任务池，并发地调用 ExtractionAgent (4.1.2) 进行信息抽取。为保证写入单个输出文件的线程安全，该流程使用了 asyncio.Lock()。抽取成功后，记录状态更新为 pending\_clustering。值得注意的是，该工作流在结束时会自动检查 pending\_clustering 状态的事件数量，若达到预设阈值，它将**自动以子进程方式触发Cortex上下文重建工作流**。

**5.2 V4.0 核心创新：Cortex引擎与上下文重建**

当积累了足够多的离散事件后，系统进入其最具创新性的阶段。run\_cortex\_workflow.py 启动，负责将独立的“信息点”聚合成连贯的“故事线”。此过程由Cortex引擎通过一个“算法粗聚类 + LLM精炼”的两阶段流程完成。

1. **阶段一：算法粗聚类 (ClusteringOrchestrator)**:
   * **核心算法**: 此阶段采用**带权重的混合距离DBSCAN算法**。选择DBSCAN是因为它无需预设聚类数量，且能有效识别噪声点，非常适合真实世界的复杂数据。
   * **混合距离度量**: 这是该算法的关键创新。它不单纯依赖语义相似度，而是将两种距离加权融合：
     1. **语义距离 (Cosine Distance)**: 捕捉事件在**内容描述**上的相似性。
     2. **实体距离 (Jaccard Distance)**: 计算事件之间**共现实体**的Jaccard距离，捕捉事件在**核心参与者**上的关联性。
   * **输出**: 算法输出一个从 event\_id到 cluster\_id 的映射，并更新数据库状态。
2. **阶段二：LLM精炼 (RefinementAgent)**:
   * **职责**: 该代理负责将算法生成的粗糙“簇”提炼成逻辑清晰的“故事单元”。
   * **核心逻辑**: 对于小簇，代理直接调用LLM生成摘要。对于超出LLM上下文窗口的大簇，它采用一种创新的\*\*“分块-摘要-合并”**策略：先将大簇分割成小块并**并发地\*\*生成初步摘要，然后再次调用LLM扮演“总编辑”角色，将这些初步摘要融合成一个连贯的最终故事摘要。
   * **输出**: 代理为每个故事生成唯一的 story\_id 和摘要，并将故事内所有事件的状态更新为 pending\_relationship\_analysis，正式移交下一步处理。

**5.3 知识的闭环：关系分析与增量学习**

1. **知识增强的关系分析 (run\_relationship\_analysis.py)**:
   * 此工作流处理所有状态为 pending\_relationship\_analysis 的“故事”。
   * **知识闭环**: 这是系统最核心的知识闭环。在进行关系分析前，工作流首先调用 HybridRetrieverAgent 从双数据库（Neo4j, ChromaDB）中检索与当前故事相关的历史背景知识，生成“背景摘要”。随后，RelationshipAnalysisAgent (4.2.1) 在这个被知识增强的、极其丰富的上下文中进行关系推理。最后，StorageAgent 将新的事件节点和分析出的关系边存入知识库，状态更新为 completed。
2. **交互式学习与成长 (run\_learning\_workflow.py)**:
   * 这是一个典型的**人机协同**工作流，负责处理所有 pending\_learning 状态的记录。
   * **学习闭环**: 专家在交互式CLI中，通过 SchemaLearningToolkit 对未知事件进行聚类和归纳，生成新的事件Schema。当专家确认并保存新Schema后，工作流会将这些事件的状态**重置为 pending\_triage**。这使得它们能被系统用新的“知识视角”重新识别和处理，完美闭合了系统的学习与成长循环。