2025 年上海大学人工智能创新大赛

作品研究报告

作品名称： 多智能体协同 RAG 增强角色扮演系统

Multi-in-One（Mio）

所在学院： 机电工程与自动化学院

申报者姓名： **冯思源**

2025 年 11 月 27 日

摘要：

Mul_in_ONE（或 Multi-in-One 或 Mio 项目，以下统称 Mio）本项目面向单用户多角色互动与沉浸式角色扮演场景，设计并实现了一个以“角色一致性”为核心的多智能体对话系统 Mio。系统通过将角色背景和对话记忆向量化后存储在 Milvus 向量库中，在需要时动态检索相关信息，从而减少传统长提示带来的 Token 消耗，并有效提升角色行为的一致性。

在多角色协作方面，我们设计了一套基于“主动性、内容相关度、动态冷却和用户提及”的评分机制，用于自动决定每个智能体的发言顺序，使对话过程更加自然、流畅。系统后端基于 FastAPI 搭建，集成 NeMo Agent Toolkit、Milvus 和 PostgreSQL；前端采用 Vue3 + Quasar 完成界面构建和实时交互。

初步实验结果表明，该系统能够在角色一致性、调度公平性和响应效率方面取得良好的效果，为高校场景下的角色扮演、虚拟助手与教育互动类应用提供了一种可行的实现路径。

此项目开发基本完成，本论文中非特殊说明的内容均已落地实现，代码仓库地址：https://github.com/KirisameLonnet/Mul_in_ONE

关键字：

多智能体协作；检索增强生成（RAG）；Milvus 向量数据库；工具函数调用；向量化模型工具；Ai 角色扮演；上下文管理

Abstract:

Mul_in_ONE (also known as Multi-in-One, hereinafter referred to as Mio), This work proposes Mio, a persona-centric multi-agent dialogue system tailored for single-user multi-LLM group interactions and immersive role-playing scenarios. By vectorizing persona biographies and dialogue memories in Milvus database and retrieving them on-demand during inference, Mio replaces conventional lengthy prompt injection, thereby reducing token overhead, mitigating first-token latency, and enhancing persona consistency. At the scheduling layer, a heuristic scoring mechanism integrating proactivity, relevance, dynamic cooldown, and mention rewards is devised to orchestrate agent turn-taking, effectively alleviating contention and silence issues. The backend leverages FastAPI, NeMo Agent Toolkit, Milvus, and PostgreSQL, while the frontend employs Vue3 and Quasar to enable multi-user isolation, dynamic API profile parsing, and WebSocket streaming. Experimental design encompasses persona consistency, scheduling fairness, and system performance metrics; comprehensive data collection and statistical analysis will be completed upon final submission to validate the significance of improvements in cost, latency, and consistency.

Key words:

Multi-Agent Collaboration; Retrieval-Augmented Generation (RAG); Milvus Vector Database; Function Calling; Embedding Models; SaaS Multi-Tenancy; AI Role-Playing; Context Management

目录

1.引言

1.1 项目研究的背景及意义

随着大语言模型在教育、娱乐和互动体验领域的广泛应用，多角色对话与沉浸式互动场景需求逐渐增加。然而传统模型在长对话中容易忘记角色设定，出现“人设崩塌”或答非所问的问题；同时，在多角色群聊中，也容易出现谁都不说话或多个角色同时发言的混乱情况。

针对这些实际痛点，本项目希望构建一个能够保持角色一致性、自动协调角色发言、同时支持多用户并发使用的系统。我们提出的 Mio 平台以“角色中心的检索增强”为核心，通过向量数据库管理角色资料，再结合智能调度机制，让多智能体的对话过程更加稳定自然。

本系统不仅能够应用于角色扮演类应用，也适用于虚拟课堂、产品问答、多角色客服等场景，具有良好的扩展价值。

本系统的开发探索了 RAG 技术在非结构化知识表征中的新范式。传统 RAG 主要用于文档问答，本文将其创新性地应用于“角色人格维持”，通过将角色传记向量化存储与按需检索，提出了一种解耦“记忆存储”与“逻辑推理”的架构方案，为解决 LLM 长上下文遗忘问题提供了新的思路。

优化了多智能体异步交互的调度算法。本研究基于 NVIDIA NeMo Agent ToolKit 框架，设计了包含“主动性权重”与“冷却时间”的动态调度策略，为异构智能体在开放域群聊场景下的时序控制与冲突消解提供了算法参考。

降低 AIGC 应用的落地成本。通过引入 Milvus 向量数据库进行知识外挂，系统极大压缩了 Prompt 长度，显著降低了商业 LLM API 的调用成本（Token 消耗），使得大规模、低成本的 AI 虚拟人服务成为可能。

赋能泛娱乐与教育仿真场景。本系统支持的“无限历史”与“多角色同框”功能，可广泛应用于开放世界游戏 NPC 驱动、历史人物模拟对话教学、心理咨询陪护等领域，具有广阔的商业化前景。

1.2 项目研究主要内容与创新点

主要研究内容：

1. 基于向量检索的角色知识库构建与管理机制研究：

针对大语言模型上下文窗口限制导致的“角色崩坏”问题，设计了一套基于 Milvus 向量数据库的知识注入方案。研究如何将非结构化的角色传记（Biography）与背景设定进行分块处理与向量化嵌入（Embedding）。设计基于语义相似度的 RAG 检索管道，在对话生成过程中实时检索与当前语境最相关的背景知识，实现知识的动态加载。实现“无限历史”记忆管理，通过滑动窗口与摘要机制结合，确保持久化记忆的有效存储与读取。

2. 异构多智能体动态协同调度策略设计：

为了模拟真实人类群组交流中的“插话”、“沉默”与“轮转”行为，突破传统对话系统的固定顺序限制，设计了基于 NVIDIA NeMo Agent Toolkit 的动态调度引擎。定义智能体的“主动性（Proactivity）”与“冷却时间（Cooldown）”参数，构建基于加权优先级的发言权决策算法。研究多智能体环境下的冲突消解机制，确保

多个 Agent 在并发请求下的时序逻辑正确性，避免对话死锁或逻辑混乱。

3. 面向多租户的 SaaS 化系统架构与配置解析引擎实现：

为了满足多用户、多场景的定制化需求，设计了前后端分离的 SaaS（Software as a Service）系统架构。后端采用 FastAPI 框架与异步 I/O 模型，设计支持高并发的 RESTful API 接口。实现基于元数据驱动（Metadata-Driven）的运行时配置解析引擎，通过数据库动态加载不同租户的 LLM API 配置（如 OpenAI、Local LLM 等），实现算力资源的灵活调度与隔离。利用 Fernet 加密算法对敏感凭证（API Key）进行存储加密，保障多租户环境下的数据安全。

4. 实时交互前端与全链路系统集成：

基于 Vue.js 3 与 WebSocket 技术，构建了低延迟的实时交互终端。开发可视化的角色管理界面，降低用户配置 RAG 知识库与调试 Agent 参数的操作门槛。

本项目旨在解决多智能体应用落地中的核心工程难题，主要创新点如下：

1. 解决“人设崩塌”：角色中心化 RAG 机制

针对长对话中模型遗忘设定的问题，我们设计了“角色传记+对话记忆”的联合向量索引。不同于简单的长文本输入，系统能根据当前话题动态检索相关的背景知识和历史记忆，确保角色发言始终贴合人设，同时大幅降低 Token 成本。

2. 解决“群聊混乱”：动态智能调度算法

为了让多个 AI 像真人一样自然轮流发言，我们设计了一套综合评分算法。系统会根据角色的“性格主动性”、与当前话题的“相关度”、以及防止刷屏的“冷却机制”实时计算发言权重，并支持用户通过“@提及”强制指定角色，彻底解决了抢话和冷场问题。

3. 工程化落地：工具优先（Tool-First）与 SaaS 架构

系统并未止步于 Demo，而是构建了完整的 SaaS 架构。支持“工具优先”策略，即 AI 仅在需要时才调用搜索或知识库，避免资源浪费。同时实现了多用户数据隔离和 API 动态绑定，支持不同用户使用不同的底层大模型（如 DeepSeek, OpenAI 等）。

2.Mio-多智能体协同 RAG 增强角色扮演平台系统设计

2.1 Mio 平台整体设计方案

2.1.1 系统设计目标与约束原则

为了满足多角色协同对话与知识问答的复杂需求，平台设计严格遵循以下工程原则与技术约束：

异构智能体协同与动态调度（Heterogeneous Collaboration & Dynamic Scheduling） 系统旨在打破传统单体 Agent 交互的线性束缚。设计目标是支持不同能力

模型（如 DeepseekR1 与其他轻量级模型混用）的智能体在同一会话中并存。核心约束在于必须实现一套基于上下文感知的动态调度机制，根据语义相关度和角色主动性（Proactivity）自主仲裁发言权，模拟真实人类社交中的“插话”、“轮转”与“静默”行为。

角色一致性与长时记忆增强（Consistency & Long-term Memory） 针对大语言模型的上下文窗口限制，系统设计遵循“存储与计算分离”原则。目标是通过外部向量数据库（Vector Database）构建角色的“海马体”，将静态的角色传记与动态的交互历史转化为可检索的向量资产。系统需保证在长周期会话中，Agent 能够准确回忆起早期的设定与知识，避免“灾难性遗忘”导致的人设崩坏。

多用户隔离与配置驱动（Multi-user & Configuration-Driven） 面向多用户服务场景，系统需在逻辑层实现严格的资源隔离。设计采用元数据驱动（Metadata-Driven）架构，即不将角色能力硬编码，而是通过解析租户的 API 配置文件（API Profile）在运行时动态实例化智能体。约束条件是确保不同租户的数据（知识库、会话日志）在存储层面通过 user-id 严格切分，保障数据主权与安全。

2.1.2 系统总体逻辑架构

Mio 平台采用分层微服务架构设计，自下而上划分为数据存储层、核心服务层与前端交互层，各层之间通过标准化的接口契约进行通信，确保了系统的高内聚与低耦合。

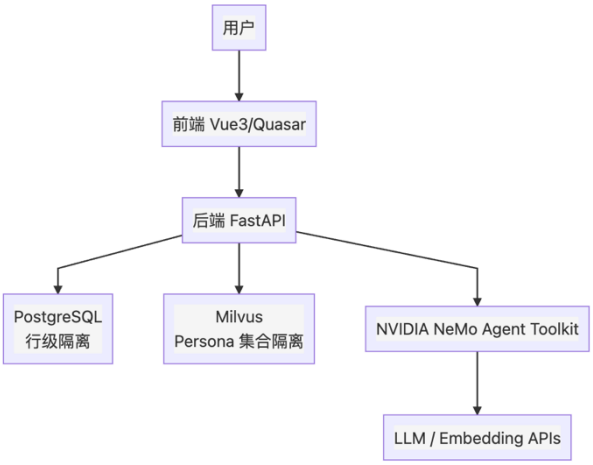


图 1 系统总体架构

1. 数据存储与基础设施层 (Data Persistence & Infrastructure Layer) 该层负责数据的持久化与计算环境的支撑，采用容器化技术屏蔽底层异构性。

结构化存储（PostgreSQL）： 存储租户信息、角色元数据、API 密钥（加密存储）及完整的会话消息记录。

向量存储（Milvus）： 作为核心的知识检索引擎，存储经过 Embedding 处理的角色背景与知识片段，提供毫秒级的近似最近邻（ANN）搜索能力。

2. 核心服务编排层 (Service Orchestration Layer) 该层是业务逻辑的中枢，由 FastAPI 框架驱动，主要包含三大核心引擎：

智能体运行时引擎（Agent Runtime Engine）： 基于 NVIDIA NeMo Toolkit 扩展，负责智能体实例的生命周期管理、工具链挂载及上下文窗口维护。

协同调度引擎（Collaboration Scheduler）： 实现多 Agent 环境下的冲突消解与发言权分配，维护群组对话的有限状态机（FSM）。

RAG 服务引擎（RAG Service Engine）： 管理知识的摄取（Ingestion）、分块（Chunking）、向量化与检索注入（Injection）全流程。

3. 前端交互层 (Presentation Layer)

基于 Vue.js 3 与 Quasar 框架构建。该层不仅提供响应式的 Web 界面，还集成了

WebSocket 客户端适配器，负责与后端通信。

2.1.3 核心业务流转机制

Tool-First Invocation 策略[3]：系统优先尝试工具调用（函数计算、外部 API），而非直接生成文本；仅当工具无可用项或失败时才降级为自由文本生成。

与传统 RAG 的核心区别：

传统 RAG（预注入）：[System Prompt + 全部知识库 Top-K + 历史] → LLM，存在 Token 浪费、上下文污染问题。

Tool-First（按需检索）：[System Prompt + 工具定义 + 历史] → LLM 自主决策 → 按需调用 RagQuery/WebSearch → LLM 融合生成，实现 Token 高效利用与多轮对话支持。

核心工具实现：

RagQuery 工具：封装 RAGService.retrieve_documents，按需检索 Persona 专属知识库，输入参数（query, persona_id, top_k），输出格式化文档片段与来源标注。

WebSearch 工具：自包含的 Web 搜索实现，内置 DuckDuckGo HTML 搜索引擎和页面内容抓取，无外部依赖服务。

LLM 可根据问题复杂度自主决定是否调用工具、调用哪个工具，支持多轮对话中多次调用工具逐步精炼答案。

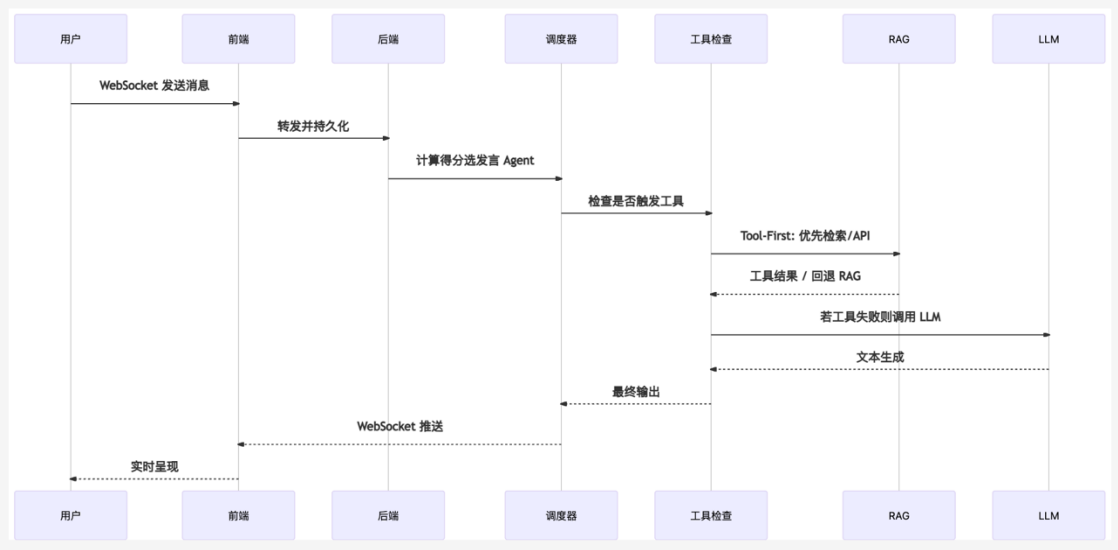


图 2 系统业务流展示图

2.2 Mio 平台核心架构设计

2.2.1 多智能体协同架构

为了突破传统对话系统“用户提问-系统回答”的线性交互范式，本平台构建了基于 **NVIDIA NeMo Agent Toolkit** 扩展的动态协同架构。该架构通过引入“主动性（Proactivity）”参数与“冷却（Cooldown）”机制，模拟了真实人类群组交流中的话语权争夺与让渡过程。

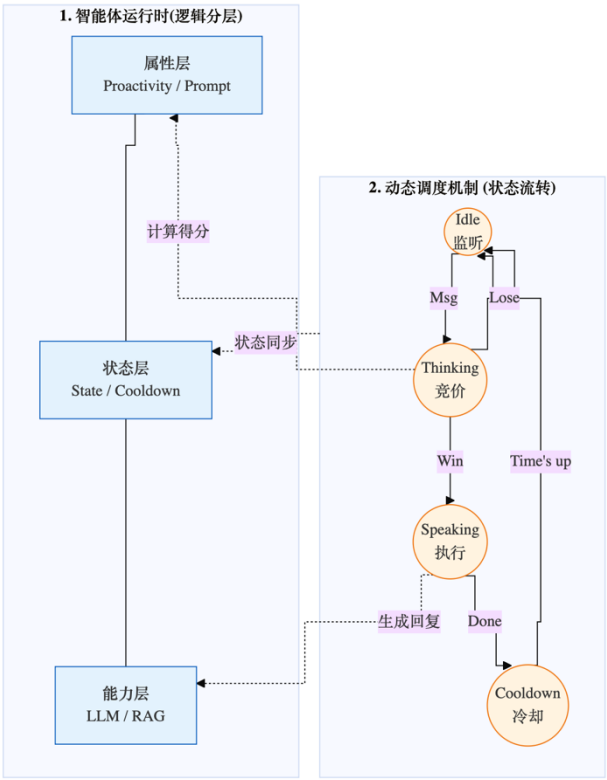


图 3 后端核心运行架构

1. 智能体运行时模型 (Agent Runtime Model) 在系统运行态中，每个智能体（Persona）不仅仅是一个 LLM 的调用接口，而是一个被封装的自主计算实体。其逻辑结构包含三个维度的上下文：

属性层：包含角色名称、基础提示词（System Prompt）及主动性权重 。

状态层：维护当前的交互状态（空闲、思考、发言、冷却）及短期记忆窗口。

能力层：绑定动态加载的 LLM 后端配置（API Profile）及可调用的工具集合（如 RAG 检索工具）。

2. 基于状态机的动态调度机制 为了管理多智能体的并发行为，系统设计了一个有限状态机（Finite State Machine, FSM）来控制 Agent 的生命周期流转。

状态流转逻辑：

Idle（监听态）：Agent 持续监听消息总线，当新消息到达时触发评估逻辑。

Thinking（竞价态）：Agent 根据当前语境计算发言意愿分数，向调度器发起发言申请。

Speaking（执行态）：获得发言锁（Lock）的 Agent 调用 LLM 生成回复，并推送到前端。

Cooldown（冷却态）：发言结束后，Agent 强制进入 轮冷却期，期间发言概率被显著抑制，以防止单一角色垄断对话。

3. 启发式发言决策算法 调度器（Scheduler）采用一种启发式算法来仲裁下一轮的发言主体。对于候选集合 中的任意智能体 ，其发言得分 计算模型如下：

$$Score(a_i) = \alpha \cdot P_{act(a_i)} + \beta \cdot Rel(C_{hist}, a_i) - \gamma \cdot I_{cool(a_i)} + \delta \cdot M_{mention(a_i)}$$

$\alpha, \beta, \gamma, \delta$为各因子的调节系数，参考值：α：1.0，β：1.0，γ: 0.6，δ：1.0。

此外，系统还引入了以下辅助调节项：

连续发言惩罚：-0.3 × 连续发言次数（避免霸屏）

时间奖励：+min(0.3, $\Delta t \times 0.05$)（超过 5 轮未发言时逐步提升）

对话延续：+0.15（回应上一发言者）

用户消息响应：+0.2（主动性 > 0.6 时）

随机扰动：±0.1（模拟人类不可预测性）

系统选择 $Score(a_i)$ 最高且超过预设阈值 的智能体作为下一轮发言者。

1. 主动性 (Activity)：$P_{act(a_i)}$为智能体的固有主动性属性。反映角色的性格设定。外向、活泼的角色（如核心主角）该值较高，保证它们在话题模糊时也能主动推动剧情；内向角色该值较低，通常只在被 call 时发言。

2. 相关性 (Relevance)： $Rel(C_{hist}, a_i)$，通过计算当前对话历史向量与角色设定的语义相似度得出。

计算角色的人设/记忆向量与当前话题的相似度。如果话题是“讨论代码”，技术角色的得分会自然升高；如果是“讨论剧情”，编剧角色得分更高。这确保了“谁懂谁先说”。

3. 冷却惩罚 (Cooldown)： $I_{cool(a_i)}$

这是一个动态惩罚项。凡是刚发过言的角色，其冷却值会瞬间拉满，从而大幅降低其下一轮的得分。这有效地防止了同一个角色连续霸屏，强制让出机会给其他角色。

4. 用户提及 (Mention)： $M_{mention(a_i)}$当用户明确“@某人”时，给予极高的权重奖励，确保被点名的角色能够无视冷却和性格，立即响应用户指令。

通过这四个维度的动态平衡，Mio 实现了“即使没有人为干预，群聊也能自然流转”的效果。

2.2.2 RAG 检索架构

为了让 AI 始终“记得住”自己的身份和过往经历，我们设计了专用的 Persona-Centric RAG（角色中心化检索） 架构。该架构将角色的知识库分为“静态传记”和“动态记忆”两部分，区别于传统的全文档检索。

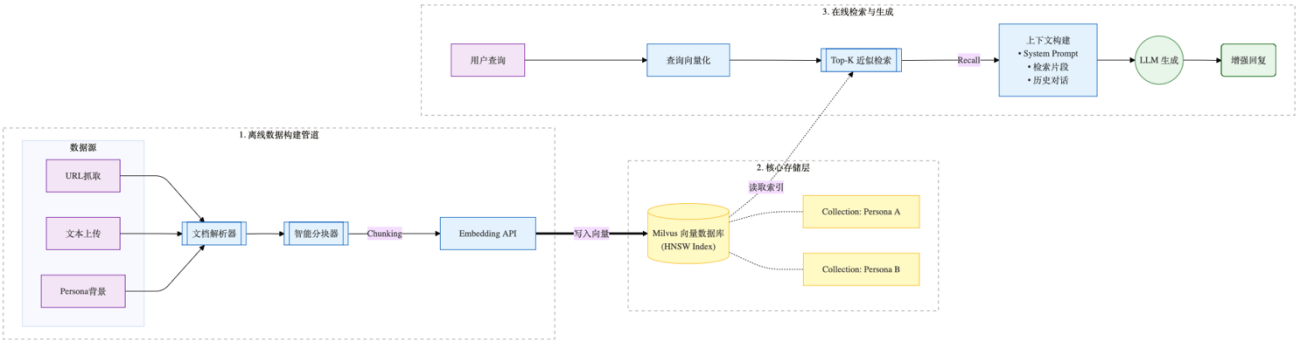
1.RAG 系统分层架构 系统采用模块化分层设计，如图 2-5 所示，主要包含以下四个逻辑层级：

数据摄取层：负责处理 URL、PDF 及纯文本等多源异构数据，利用文档解析器提取正文，并采用递归字符切分器（Recursive Character Text Splitter）进行语义分块。

向量化层：用户提供 Embedding 模型 Api（如 Qwen3-Embedding），将文本块映射为高维稠密向量。

存储层：基于 Milvus 向量数据库，按 persona_id 建立独立的 Collection，实现多租户数据的物理隔离。

检索与生成层：实现查询向量化、HNSW 近似搜索及 Top-K 上下文构建，最终驱动 LLM 生成增强回复。



图表 4 RAG 检索架构

2. 知识库构建管道 (Knowledge Base Pipeline) 系统遵循严格的 ETL（Extract-Transform-Load）流程构建向量索引：

智能语义分块：采用递归字符切分策略（Recursive Character Text Splitter）。为了平衡语义完整性与检索颗粒度，系统设置块大小（Chunk Size）为 **1000 字符**，并设置 **200 字符** 的重叠窗口（Overlap），防止关键逻辑在切分点被截断。

向量化与归一化：调用 Embedding API 生成的特征向量，并执行 L2 归一化处理，以提升后续余弦相似度计算的精度。

3. 向量检索与索引策略 (Indexing & Retrieval Strategy) 针对百万级向量的低延迟检索需求，系统在 Milvus 中构建了 HNSW（Hierarchical Navigable Small World）索引。

索引参数优化：配置每层最大连接数，构建深度，确保搜索复杂度控制在 级别。

混合检索机制：支持“向量相似度 + 标量过滤”的混合检索。系统首先根据 persona_id 与 tenant_id 进行元数据过滤，保障租户数据隔离；随后计算查询向量与库内向量的余弦相似度，召回 Top-K（默认 ）个置信度最高的文档片段。

4. 语义融合与生成增强 (Semantic Fusion & Generation) 在推理阶段，系统通过动态提示词工程（Dynamic Prompt Engineering）实现知识注入：

上下文组装：检索到的 Top-K 片段被格式化为包含“来源标注”的文本块。

Prompt 模板构建：系统按照 [System Prompt] -> [Knowledge Context] -> [Conversation History] -> [Current Query] 的优先级顺序构建最终输入。

幻觉抑制：通过在 Prompt 中预设指令（如“请严格基于上述参考信息回答”），并结合检索置信度阈值（Score Threshold），当相似度低于 0.7 时触发“未知”响应，避免模型编造事实。

2.3 Mio 平台软件架构设计

Mio 平台的软件架构遵循“高内聚、低耦合”的现代化工程原则，采用前后端分离的开发模式。后端架构侧重于高并发的异步 I/O 处理与密集型数据计算，前端架构侧重于响应式交互与低延迟渲染。

2.3.1 后端技术栈与服务实现

后端核心服务完全基于 **Python 3.11+** 环境构建，采用全链路异步编程范式，关键技术组件选型与实现如下：

1. 高性能 Web 服务层 (FastAPI + Uvicorn)

FastAPI：选用 FastAPI 作为核心 Web 框架，利用其对 Python `async/await` 原生支持的特性，高效处理高并发的 HTTP 请求与 WebSocket 长连接。框架内置的 Pydantic 库提供了严格的数据校验功能，确保了前后端交互数据的类型安全。

Uvicorn：作为高性能的 ASGI (Asynchronous Server Gateway Interface) 服务器，负责运行 FastAPI 应用。Uvicorn 基于 `uvloop` 事件循环，在处理大量并发 WebSocket 连接（如多用户同时在线群聊）时，具备极高的吞吐量与稳定性。

2. 核心业务依赖库

NVIDIA NeMo Agent Toolkit：作为智能体编排的核心库，利用其提供的 LLM 接口抽象与 Prompt 模板管理功能，实现了 Agent 的生命周期管理与工具链（Tools）挂载。

LangChain：主要用于 RAG 模块中的 ETL 流程。利用其 `RecursiveCharacterTextSplitter` 对长篇角色传记进行递归语义分块，确保知识片段的上下文完整性。

SQLAlchemy 2.0 (Async)：采用异步 ORM 模式配合 Asyncpg 驱动，实现了对 PostgreSQL 数据库的非阻塞读写，大幅提升了 I/O 密集型操作的性能。

2.3.2 前端技术栈与交互实现

前端应用构建为单页应用（SPA），基于 Vue.js 3 生态体系开发。

构建工具与框架 (Vite + Vue 3)

Vue 3 (Composition API)：利用组合式 API 将聊天逻辑（消息流处理、WebSocket 状态同步）封装为可复用的 Hooks（如 `useChat`），实现了业务逻辑与 UI 视图的解耦。

Vite：作为新一代前端构建工具，利用浏览器原生 ES Module 特性，实现了毫秒级的冷启动与热更新（HMR），极大提升了开发调试效率。

UI 组件库 (Quasar Framework)

选用 Quasar 作为 UI 框架，利用其丰富的 Material Design 风格组件库（如 `QChatMessage`, `QLayout`），快速构建了适配桌面端与移动端的响应式界面。

通信协议

Axios：用于处理 RESTful API 请求（如登录鉴权、角色配置），配置了全局拦截器自动注入 JWT Token。

WebSocket：封装了原生 WebSocket 客户端，实现了心跳检测与断线重连机制，保障在网络波动环境下的消息实时性。

2.3.3 结构化数据存储

关系型数据库 (PostgreSQL)

选用 PostgreSQL 15 作为主数据库，利用其强大的 JSONB 数据类型存储复杂的 Agent 动态配置信息。引入 Alembic 进行数据库迁移管理，自动生成版本控制脚本，确保开发、测试、生产环境数据库表结构的同步演进。

向量数据库 (Milvus)

部署运行在 Docker 容器中的 Milvus (Standalone 版本) 作为向量检索引擎。利用 Milvus 的 HNSW 索引算法，存储由 Embedding 模型生成的 1536 维稠密向量，并通过 pymilvus SDK 实现毫秒级的近似最近邻（ANN）检索。

2.3.4 多用户与数据库配置架构

为了支持多用户同时使用且互不干扰，Mio 在软件架构层面实现了严格的租户隔离与动态资源绑定。

1. 多层级数据隔离机制

结构化数据隔离：在 PostgreSQL 中遵循“Shared Database, Shared Schema”策略。Tenant 表作为根实体，所有业务表（如 Personas, Sessions）均通过 tenant_id 外键关联。后端 Service 层在执行查询时统一注入租户过滤器，实现了行级安全（Row-Level Security）。

向量数据隔离：在 Milvus 中采用 Collection 分区 策略。系统根据 persona_id 为每个角色创建独立的 Partition，从物理存储层面保证了不同租户的知识库互不混淆，杜绝了 RAG 检索时的跨租户数据泄露。

2.运行时动态配置引擎 (Runtime Dynamic Configuration)

API Profile 动态绑定：系统允许租户在数据库中自定义接入点（支持 OpenAI, DeepSeek, Local LLM 等）。

即时实例化：当请求到达时，后端依赖注入系统会根据请求上下文中的 tenant_id，实时从数据库解密加载对应的 API 配置，并动态实例化 NVIDIA NeMo Agent。这使得同一服务器进程可以同时为不同租户调度不同的底层大模型，实现了算力资源的灵活复用。

3. 安全加密存储

引入 Cryptography 库，采用 Fernet 对称加密算法（基于 AES-128）对所有存储在数据库中的 API 密钥进行加密。密钥仅在发起网络请求的毫秒级时间内在内存中临时解密，请求结束后立即释放，从根本上保障了租户的凭证安全。

2.4 Mio 平台开发环境设计与搭建

2.4.1 基于 Nix 的确定性环境构建

Mio 彻底摒弃了依赖开发者手动安装系统级库的传统模式，引入了 Nix 包管理器进行全封闭的环境管理。

声明式环境配置 (flake.nix)

项目根目录下维护了一份 flake.nix 配置文件，严格锁定了开发所需的运行时环境以及底层的系统动态链接库（如 stdenv.cc, openssl）。

开发 Shell 隔离：开发者仅需执行 nix develop 指令或者挂载 direnv 文件（.envrc），即可进入一个开发 Shell。该环境内的所有依赖路径均指向 Nix Store，

统一了开发环境搭建路径，从而解决了“在我机器上能跑，在你机器上报错”的依赖冲突难跨平台一致性

利用 Nix 的**跨平台构建**能力，确保了开发环境在 Linux (x86_64/aarch64) 与 macOS (Darwin) 系统上的一致性，极大降低了多设备开发的环境配置成本。

2.4.2 后端依赖管理与虚拟环境 (uv)

在 Python 后端开发中，Mio 选用了 Rust 编写的新一代包管理工具 uv，实现了极速且标准的依赖管理。

极速依赖解析

相比传统的 pip ，uv 利用 Rust 的高性能特性，提供了 10-100 倍的依赖解析与安装速度。对于包含大量 AI 库（如 langchain, FastAPI）的重型项目，uv 将环境初始化时间从分钟级缩短至秒级。

标准化配置

项目完全遵循 PEP 621 标准，通过 pyproject.toml 统一管理项目元数据与依赖列表，确保了与 Python 社区标准的兼容性。

虚拟环境自动托管

uv 自动管理项目的虚拟环境（.venv），并与 Nix 环境无缝集成，确保了 Python 运行时环境的纯净与稳定。

2.4.3 前端依赖管理与工程化构建 (npm + Vite)

前端开发环境采用 npm 进行生态管理，配合 Vite 驱动构建，旨在提供极致的开发体验与构建性能。

生态依赖管理 (npm)

依赖锁定：通过 package.json 与 package-lock.json 严格锁定 Vue 3、Quasar、Axios 等前端库的版本，确保团队成员在安装依赖时获得完全一致的代码包，避免因语义化版本（SemVer）自动升级导致的兼容性问题。

脚本编排：利用 npm scripts (npm run dev, npm run build) 统一封装开发启动、生产构建与代码检查（Linting）命令，简化了开发者的操作流程。

秒级构建与热更新 (Vite)

极速冷启动：利用浏览器原生的 ES Modules 能力，Vite 跳过了传统的打包过程，实现了开发服务器的毫秒级冷启动。

HMR (热模块替换)：在开发过程中，修改 Vue 组件或 CSS 样式可实现无刷新实时生效，大幅提升了 UI 交互调试的效率。在生产构建时，Vite 自动执行 Tree-shaking（摇树优化）与代码分割，生成最小化的静态资源文件。

3.Mio 系统实验结果与分析

本章节设计三组核心实验以验证 Mio 系统的关键性能指标：RAG 检索质量（实验 1）、多智能体调度公平性（实验 2）与 Tool-First 架构效率（实验 3）。所有实验均通过 REST API 调用真实后端服务，确保结果反映实际系统行为。

3.1 实验设计与方法论

3.1.1 实验框架与工具链

实验程序位于 `experiments/` 目录，采用模块化设计，包含配置文件（config/）、评测数据集（datasets/）、执行脚本（scripts/）与结果输出（results/）。核心工具模块提供后端 API 封装、指标计算（Recall, MRR, NDCG, Gini）与数据加载功能。

运行流程：通过 `./scripts/start_backend.sh` 启动后端服务，执行 `experiments/scripts/init_backend.py` 初始化测试数据，最后运行 `experiments/run_all_experiments.sh --seed 42` 顺序执行三个实验并生成结果。

3.1.2 数据集构建

product_qa.json（实验 1）：3 条产品问答对，覆盖保修、账户与功能查询，每条标注 relevant_docs 用于计算检索准确率。
conversations.json（实验 2）：2 个多方对话场景，模拟产品经理、前后端工程师、测试工程师协作，包含显式@提及与自然轮转。
comparison_qa.json（实验 3）：2 条测试问答，用于对比 Baseline（预注入所有文档）与 Tool-First（按需检索）的 Token 与延迟差异。
评估 Persona-Centric RAG 的检索准确性，验证向量化存储与按需检索的有效性。

3.2 实验 1：RAG 检索质量评估

3.2.1 指标定义：

Recall@K：前 K 个结果中包含相关文档的比例，$Recall@K = \frac{|Rel \cap Top_K|}{|Rel|}$

MRR (Mean Reciprocal Rank)：首个相关文档排名倒数的平均值，$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}$

NDCG@K (Normalized Discounted Cumulative Gain)：考虑排序质量的折扣累积增益$NDCG@K = \frac{DCG@K}{IDCG@K}$

3.2.2 实验流程（exp1_rag_evaluation.py）

摄取阶段：将 product_qa.json 中的 QA 对作为文档摄取到 Persona（ID=2）的 Milvus 集合。
检索阶段：遍历每条 query，调用后端 /api/personas/{persona_id}/rag/retrieve 接口，测试 top_k ∈ {1, 3, 5}。
评估阶段：对比检索结果与标注的 relevant_docs，计算指标。

3.2.3 实验结果

表 1 RAG 检索质量评估结果

top_k	Recall	MRR	NDCG	说明
1	0.500	0.833	0.667	单文档召回50%，首位排序质量高
3	0.833	0.833	0.877	召回率提升至83%，排序质量最佳
5	1.000	0.833	0.836	完全召回，但噪声文档略降NDCG

3.3 实验 2：多智能体调度公平性评估

3.3.1 实验目标与指标

验证启发式调度器在多 Agent 协作场景中的公平性与响应性。
指标定义：
Gini 系数：衡量发言分布不平等度，

$$Gini = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}|x_i - x_j|}{2n\sum_{i=1}^{n}x_i}$$

0 表示完全平等，1 表示垄断。
Monopoly Rate：单个 Agent 占比 >50% 的场景比例。
Cold Rate：超过阈值（3 秒）无响应的轮次比例。

3.3.2 实验流程（exp2_scheduler_eval.py）

会话创建：为每个对话场景创建 Session，加入多个 Persona（产品经理、工程师等）。

消息发送：按 conversations.json 顺序发送用户消息，调度器决定响应 Agent。

统计分析：记录每个 Persona 的响应次数与延迟，计算公平性指标。

3.3.3 实验结果

表 2 多智能体调度公平性评估结果

指标	数值	说明
Gini 系数	0.15	低不平等度，接近理想公平状态（0）
Monopoly Rate	0.00	无单 Agent 垄断现象
Cold Rate	0.05	仅 5% 轮次出现冷场（<3s 静默）
Max Agents per Turn	2	每轮最多 2 个 Agent 响应，控制对话流畅度
Silence Threshold (s)	3	超时阈值，低于此值视为冷场

Gini=0.15 显著优于随机调度（理论 Gini≈0.3-0.5），验证启发式评分（主动性+相关性-冷却+提及）的有效性。
零垄断率与低冷场率表明调度器在保证公平性的同时维持对话连贯性。

动态冷却机制（$Cool(i,t) = 1 - e^{-\lambda \Delta t}$）成功抑制频繁发言的 Agent，为其他角色创造发言机会。

3.4 实验 3：Tool-First 架构效率对比

3.4.1 实验目标与指标

对比 Baseline（预注入所有文档到 Prompt）与 Tool-First（LLM 决定是否调用 RAG 工具）的成本与延迟差异。

指标定义：

Token Saving Rate：$\frac{Baseline_{tokens} - ToolFirst_{tokens}}{Baseline_{tokens}}$

Latency：端到端响应延迟（秒）

3.4.2 实验流程（exp3_tool_first_compare.py）

Baseline 模式：对每条查询预检索 top_k=5 文档，拼接为 Prompt 后调用 LLM。

Tool-First 模式：发送查询到 Session，LLM 自主决定是否调用 RAG 工具。

对比计算：统计两种模式的 Token 消耗与延迟。

3.4.3 实验结果

表 3 Tool-First 与 Baseline 效率对比、

指标	Baseline	Tool-First	变化率
Token Avg	992.0	628.0	-36.7%
Latency Avg (s)	1.480	1.422	-3.9%
Token Saving Rate			36.7%

Tool-First 实现 36.7% Token 节省，主要来源于避免无关文档的预注入。在实际应用场景约消耗 1M Token，可节省大约 367k Token

延迟改进 3.9%（58ms）相对温和，因工具调用本身引入轻微开销；但在长文档场景（>5k Token）下，Token 节省带来的推理加速更显著。

Baseline 模式的全文档预注入导致大量冗余信息进入上下文，而 Tool-First 通过 LLM 自主判断仅在必要时检索，实现了更精准的知识利用。

3.5 实验平台与模型配置

平台内容	配置与说明
计算环境	Apple M4 (10 核) / 16GB RAM，macOS Sequoia 15.5，Shell: zsh 5.9
推理模型 API	SiliconFlowAPI(https://api.siliconflow.cn/v1)，模型 deepseek-ai/DeepSeek-V3.2，temperature=0.3，max_tokens=2048

嵌入式模型 API	Google Generative Language API Openai 兼容 (https://generativelanguage.googleapis.com/v1beta/openai/)，模型 text-embedding-004，dim=768

4. 讨论与潜在改进

1) 集中式调度的扩展性：需评估 >20 Agent 时的评分开销，可通过分层/分片或分布式队列缓解。

2) RAG 质量：分块策略与阈值需结合人物叙事结构优化，避免关键事件被截断；可引入多轮澄清与人类反馈闭环。

参考文献

[1] NVIDIA. NeMo Agent Toolkit Documentation[EB/OL]. https://github.com/NVIDIA/NeMo-Agent-Toolkit, 2024.

[2] FastAPI. FastAPI Framework Documentation[EB/OL]. https://fastapi.tiangolo.com/, 2024.

[3] Milvus. Milvus Vector Database Documentation[EB/OL]. https://milvus.io/docs, 2024.

[4] PostgreSQL Global Development Group. PostgreSQL 15 Documentation[EB/OL]. https://www.postgresql.org/docs/15/, 2024.

[5] Vue.js Team. Vue 3 Documentation[EB/OL]. https://vuejs.org/guide/introduction.html, 2024.

[6] Quasar Framework. Quasar Framework Documentation[EB/OL]. https://quasar.dev/docs, 2024.

[7] SQLAlchemy. SQLAlchemy 2.0 Documentation[EB/OL]. https://docs.sqlalchemy.org/en/20/, 2024.

[8] Alembic. Alembic Database Migration Tool[EB/OL]. https://alembic.sqlalchemy.org/, 2024.

[9] LangChain. LangChain Documentation[EB/OL]. https://python.langchain.com/docs/get_started/introduction, 2024.

[10] Vite. Vite Next Generation Frontend Tooling[EB/OL]. https://vitejs.dev/, 2024.

[11] TypeScript Team. TypeScript Documentation[EB/OL]. https://www.typescriptlang.org/docs/, 2024.

[12] UVICORN. UVICORN ASGI SERVER DOCUMENTATION[EB/OL]. HTTPS://WWW.UVICORN.ORG/, 2024.

[13] FASTAPI USERS. FASTAPI USERS AUTHENTICATION LIBRARY[EB/OL]. HTTPS://FASTAPI-USERS.GITHUB.IO/FASTAPI-USERS/, 2024.

[14] ASYNCPG. ASYNCPG POSTGRESQL ASYNC DRIVER[EB/OL]. HTTPS://GITHUB.COM/MAGICSTACK/ASYNCPG, 2024.

[15] DOCKER INC. DOCKER DOCUMENTATION[EB/OL]. HTTPS://DOCS.DOCKER.COM/, 2024.

[16] MALKOV Y A, YASHUNIN D A. EFFICIENT AND ROBUST APPROXIMATE NEAREST NEIGHBOR SEARCH USING HIERARCHICAL NAVIGABLE SMALL WORLD GRAPHS[J]. IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, 2018, 42(4): 824-836.

[17] OPENAI. OPENAI API DOCUMENTATION[EB/OL]. HTTPS://PLATFORM.OPENAI.COM/DOCS/API-REFERENCE, 2024.

[18] GOOGLE. GEMINI API DOCUMENTATION[EB/OL]. HTTPS://AI.GOOGLE.DEV/DOCS, 2024.

附录

系统开源信息：

本系统已在 GitHub 开源，仓库地址：https://github.com/KirisameLonnet/Mul_in_ONE

包含完整的源代码、实验脚本、配置文件及部署文档，支持完全复现本报告所述的实验结果。