AgentRM：面向大语言模型智能体系统的操作系统启发式资源管理器

Jianshu She

MBZUAI

arXiv:2603.13110v1 [cs.OS] 13 Mar 2026

摘要—**大语言模型（LLM）智能体系统已在各类领域迅速普及，但仍存在严重的用户体验问题，限制了其实际部署。通过对六种主流智能体框架（OpenClaw、AutoGen、CrewAI、LangGraph、Codex、Claude Code）的 40,000 多条 Git Hub 问题进行实证分析，我们发现了两项根本性的资源管理挑战：（1）由于阻塞、僵尸进程和速率限制级联导致调度失败，进而造成系统无响应；（2）上下文退化，原因是内存无限增长且保留策略不佳，导致智能体“失忆”。受数十年操作系统研究的启发，我们提出 AgentRM，这是一种中间件资源管理器，将智能体资源视为与操作系统资源类似的资源。AgentRM 采用多级反馈队列（MLFQ）调度器，并结合僵尸回收和速率限制感知准入控制；同时配备三级上下文生命周期管理器，实现自适应压缩和休眠机制。评估结果表明，AgentRM 带来了显著改进：AgentRM-MLFQ 将 P95 延迟降低 86%，将通道浪费时间减少 96%，并将吞吐量提升 168%，同时消除了僵尸智能体（0 个，而基线为 29 个）。AgentRM-CLM 的关键信息保留率达到 100%，质量评分为 95%；相比之下，现有方法的信息保留率为 65.1%，质量评分为 87%，但 AgentRM-CLM 的压缩成本更高（34,330 对比 17,212 个令牌）。**

索引词—**大语言模型智能体、资源管理、调度、上下文管理、操作系统**

I. 引言

大语言模型（LLM）智能体系统已成为构建智能应用的一种变革性范式，使其能够自主推理、规划并执行复杂任务。OpenClaw（GitHub 上的问题数超过 40,000 个）、AutoGen [1], CrewAI [2], 和 LangGraph [3]等框架，使开发者能够构建复杂的多智能体系统，实现协作式问题求解、工具使用和延展推理链。

然而，随着这些系统的复杂度和用户采用率不断提升，它们也面临着根本性的资源管理挑战，这些挑战与早期操作系统所面临的问题如出一辙。我们对生产部署的实证分析揭示了两个严重限制当前智能体框架实用性的关键用户体验问题：

调度故障：智能体系统常因资源调度不当而变得无响应。我们发现了以下问题：高优先级用户交互被后台任务延迟的阻塞行为、僵尸子智能体占用执行通道却未开展有效工作，以及导致全系统故障的级联限流。这些问题表现为用户可感知的延迟超过 30 秒，并且在高负载时段系统完全不可用。

上下文退化：随着上下文窗口填满并超出容量限制，长时间运行的智能体会话会出现“失忆”现象。当前方法要么截断近期历史记录（丢失重要上下文），要么在内存约束超限时崩溃。这会导致智能体忘记关键信息、生成前后不一致的响应，并迫使用户不断重新建立上下文。

推动我们开展这项工作的核心洞见是：智能体资源类似于操作系统资源。正如操作系统内核在相互竞争的进程之间管理 CPU 时间、内存和 I/O 资源一样，智能体系统也必须在相互竞争的智能体任务之间管理执行通道、速率限制和上下文窗口。这一类比表明，我们可以借鉴成熟的操作系统技术来解决智能体资源管理问题。

我们提出了 AgentRM——一款面向大语言模型智能体系统、受操作系统启发的资源管理器，它作出了以下贡献：

1）**实证研究：**我们分析了来自六个主流智能体框架的 40,000+ 个现实问题，对常见故障模式进行分类，并量化其对用户体验的影响。2）**AgentRM 架构：**我们设计了一款中间件资源管理器，包含两个核心组件：（1）智能体调度器，采用多级反馈队列（MLFQ），并实现僵尸回收和感知速率限制的准入控制；（2）上下文生命周期管理器，采用三级存储层次结构和自适应压缩。

3)**全面评估：**我们证明，AgentRM 可消除僵尸进程（0 个，而基线为 27 个），在负载下保持高吞吐量（45.2 对比 45.1 请求/分钟），并实现近乎完美的上下文保留（100%，而现有方法为 65.1%），同时保持信息质量（95% 对比 87%）。

本文其余部分安排如下：第二节通过实证研究介绍背景与动机；第三节形式化定义系统模型和问题表述；第四节介绍 AgentRM 架构；第五节介绍实现；第六节评估性能；第七至第九节讨论相关工作并作出总结。

II. 背景与动机

A. 智能体系统故障实证研究

为了解生产环境中的智能体系统中资源管理问题的范围和性质，我们对六个主要框架的 GitHub 问题进行了全面分析：OpenClaw、AutoGen、CrewAI、LangGraph、Codex 和 Claude Code。我们人工审阅了问题标题、描述和讨论串，以识别与系统响应能力和上下文管理相关的模式。

我们的分析揭示了反复出现且直接影响用户体验的故障模式：

跨通道阻塞（OpenClaw #12442）： 由于后台任务占用了所有可用的执行通道，一条高优先级用户消息被阻塞了 6 个多小时。系统在 12 小时内经历了 6+ 次无响应，迫使用户重启整个网关进程。

僵尸子智能体（OpenClaw #25992）： 由于清理不当，已完成任务的子智能体进程仍持续占用执行通道达 11+ 分钟。这一资源泄漏导致后续用户请求无限期排队，直到人工干预。

智能体失忆（OpenClaw #39282）： 在一个尤其典型的例子中，一个人工智能智能体为自己撰写了一份 GitHub issue，称自己在上下文窗口溢出后“醒来时失忆了”。该智能体失去了此前所有对话和正在进行的任务的记忆。

上下文限制违规（OpenClaw #24031）： 会话经常超出配置的上下文令牌限制，导致近期消息意外被截断，并丢失完成任务所需的关键上下文。

内存相关崩溃（OpenClaw #28629）： 一次 pytest 执行因日志无节制累积而消耗了 6.3GB 随机存取存储器，导致整个网关进程崩溃，并丢失所有活动会话的状态。

速率限制级联（OpenClaw #3181）： 失控的心跳循环导致 API 调用过多，触发了速率限制，影响系统中的所有智能体，而不仅仅是行为异常的那个智能体。

B. 问题分类

根据我们的实证分析，我们将智能体系统问题归为两大类：

调度问题： 这类问题涉及在相互竞争的智能体任务之间分配执行资源（通道、CPU 时间、API 配额）：

- 阻塞：低优先级后台工作导致高优先级任务延迟
- 僵尸进程：已完成任务未能释放资源
- 速率限制级联：一个智能体的过度使用会影响其他智能体
- 饥饿：低优先级任务始终无法获得资源

上下文管理问题： 这些问题涉及对有限上下文窗口空间的管理：

- 无界增长：超出内存限制的会话
- 失忆：由于截断而丢失重要信息
- 阈值不匹配：限制与实际使用情况之间的协调不佳
- 浪费性注入：包含无关的历史上下文

这些问题与经典的操作系统挑战直接对应：进程调度、内存管理、死锁检测和资源分配。这一观察促使我们采用将成熟的操作系统技术应用于智能体领域的方法。

III. 系统模型与问题表述

我们通过定义关键系统组件及其关系，对智能体资源管理问题进行形式化。

A. 形式化定义

智能体： 一种自主实体 $a \in A$，能够处理消息并通过大语言模型推理调用生成响应。

轮次： 一个离散的交互单元 $t = (m_{in}, m_{out}, d, r)$，其中 m_{in} 是输入消息，m_{out} 是响应，d 是处理时长，r 是消耗的资源（令牌、API 调用）。

执行通道： 一个执行槽 $l \in L$，代表系统处理智能体并发操作的能力。系统有固定数量的执行通道 $|L| = N$。

上下文窗口： 智能体a的有界内存空间 W_a，最大容量为 $|W_a| \leq C$ 个令牌，其中包含智能体的对话历史和工作记忆。

僵尸轮次： 如果一个轮次 t 在挂起时占用执行通道超过 30 秒，就会成为僵尸轮次。与二元超时标志不同，这一定义捕捉了资源浪费这一潜在属性，而它只有在执行过程中才会显现。

B. 调度问题形式化

智能体调度问题旨在资源约束下最小化加权响应时间：

$$\min \sum_{t \in T} w_t \cdot R_t \tag{1}$$

约束条件为：

$$\sum_{a \in A} \mathbb{I}[\text{active}(a)] \leq N \qquad \text{(lane constraint)} \tag{2}$$

$$\sum_{a \in A} \text{rate}(a) \leq R_{max} \qquad \text{(rate limit constraint)} \tag{3}$$

其中，w_t 是轮次 t的优先级权重，R_t 是其响应时间；如果智能体 a正在使用执行通道，则 active(a)]为 1；rate(a) 是智能体 a当前的 API 调用速率。

C. 上下文管理问题形式化

上下文管理问题旨在在窗口大小约束下最大化保留的信息价值：

$$\max \sum_{m \in M} v(m) \cdot \mathbb{I}[m \in W_a] \tag{4}$$

受以下条件约束：

$$\sum_{m \in W_a} |m| \leq C \tag{5}$$

其中， $v(m)$ 表示消息 m的信息价值，而$\mathbb{I}[m \in W_a]$ 表示消息 m 是否保留在智能体a的上下文窗口中。

IV. AgentRM 架构

AgentRM 由三个主要组件构成，它们协同工作，为智能体系统提供全面的资源管理。

A. 概述

AgentRM 作为智能体网关与模型 API 之间的中间件运行，拦截所有智能体操作，以应用资源管理策略。该系统维护有关资源利用率的全局状态，同时对各个智能体保持透明。图 1 展示了传统智能体系统与 AgentRM 的架构对比。

图 1. 架构对比：(a) 直接访问模型的传统智能体系统；(b) 采用集中式资源管理的 AgentRM。

三个核心组件如下：

- **智能体调度器：** 使用多级反馈队列（MLFQ）管理执行通道分配和 API 速率限制，并进行僵尸回收
- **上下文生命周期管理器：** 通过自适应压缩和休眠实现三级上下文存储
- **资源监视器：**跟踪系统状态，并为调度决策提供反馈

B. 智能体调度器

智能体调度器主要借鉴经典操作系统调度算法，并针对智能体工作负载的独特特性进行调整。

1) 多级反馈队列（*MLFQ*）： 我们实现了一个受 Corbat´o 的 CTSS [4]和 Linux 的完全公平调度器（CFS） [5]启发的三级多级反馈队列：

- **队列 0（交互式）：** 优先级最高的面向用户的消息
- **队列 1（子代理）：** 由智能体生成的计算任务
- **队列 2（后台）：** 维护、日志记录和周期性任务

任务从队列 0 开始，并根据执行时间和资源使用情况降级。优先级提升会定期提升长时间运行任务的优先级，以防止发生饥饿，这与 Solaris TS 调度 [6] 和 CFS 虚拟运行时间核算类似。

算法 1 展示了多级反馈队列调度逻辑：

2) 僵尸回收器： 受 Unix 进程管理的启发， [7]，我们实现了一个僵尸回收器，每 5 秒扫描一次悬挂轮次。回收器将持有执行通道超过 30 秒且处于挂起状态的轮次识别为僵尸进程。检测到僵尸进程后，回收器会执行概率恢复：悬挂轮次有 50% 的概率在重试时成功，以此模拟瞬时故障与持续性故障之间的区别。恢复失败的轮次将被终止，以释放其执行通道。

数据： 任务队列 Q_0， Q_1， Q_2；可用通道 L
结果： 已调度任务**当** 系统运行时 **执行**

```
如果 |L| > 0 则
    对于 i ← 0 到 2 执行
        如果 Q_i 不为空则
            task ← Q_i.dequeue();
            lane ← L.acquire();
            schedule(task, lane);
            跳出;
        end
    end
end
若到了_boost()则
    提升优先级();
end
sleep(scheduling_quantum);
end
```

算法 1： AgentRM-MLFQ 调度

3) 速率限制感知调度： 借鉴 TCP 拥塞控制 [8] 和 ATM 接纳控制 [9], ，我们实现了：

- **每个模型 API 配置一个令牌桶，**并可配置补充速率
- **加性增大乘性减小退避**（检测到速率限制时）
- **准入控制** ： 在队列入口处根据当前 API 利用率进行控制

4) 公平性机制： 我们采用主导资源公平（DRF） [10], 借鉴 YARN 等集群调度器 [11] ，以管理多维资源（通道、令牌、内存）。该调度器遵循工作守恒原则，在资源可用时，将高优先级队列中的闲置资源分配给低优先级任务。

C. 上下文生命周期管理器

上下文生命周期管理器实现了三级存储层次结构，其设计灵感来自计算机体系结构中的存储层次结构 [12]和虚拟内存系统 [13]。

1) 三级架构：

- **第 0 层：活动上下文 (**≈L1 缓存**)**：当前已加载的上下文，访问延迟为 0 毫秒
- **第 1 层：温存储 (**≈随机存取存储器**)**：压缩摘要，访问延迟为 ∼1 秒
- **第 2 层：冷存储 (**≈ 磁盘**)**：完整对话记录，访问延迟为 ∼3 秒

当所访问信息位于较低层级时，会发生上下文缺页，并触发提升，其机制类似于虚拟内存系统中的页面错误。我们采用回写策略，延迟持久化上下文更改，并使用工作集模型预测未来的访问模式。

2）自适应压缩：我们的自适应压缩算法借鉴了 LRU-K [14], ARC [15], 等缓存替换策略，以及 zswap [16]等内存压缩技术。

我们将消息价值计算如下：

$$v(m) = \alpha \cdot \text{近期性}(m) + \beta \cdot \text{重要性}(m) + \gamma \cdot \text{关键信息加成}(m) \tag{6}$$

其中，近期性倾向于优先考虑近期消息，重要性由语义分析得出，而关键信息加成则分配给包含结构化数据、决策或承诺的消息。

遵循“压缩而非丢弃”的原则（类似于 zswap），我们生成压缩摘要，而不是简单截断内容。压缩阈值会根据模型上下文大小进行调整，这与 Linux 的 vm.swappiness 参数类似。

我们通过语义相似度来预测未来访问概率，以近似 Belady 最优置换算法 [17] 。

算法 2 概述了自适应压缩过程：

数据： 上下文窗口 W；大小限制 C；消息 M**结果：** 压缩后的上下文**当** $|W| > C$ **对** $m \in M$ **执行** $v(m) \leftarrow$ 计算价值(m);**结束** $M_{sorted} \leftarrow$ 排序$(M,\ v)$; $victim \leftarrow M_{sorted}[0]$;**如果**重要*(victim)* **则** $summary \leftarrow$ 压缩$(victim)$;$W.replace(victim, summary)$;**否则** $W.remove(victim)$;**结束 算法 2：** 自适应压缩

3）休眠：对于非活动会话的长期存储，我们实现了受 CRIU（用户空间检查点与恢复） [18],、虚拟机实时迁移 [19], 和数据库预写日志 [20]启发的休眠机制。

休眠会序列化完整的会话状态，包括上下文、局部变量和执行状态，从而能够在不发生失忆的情况下恢复。这对于间歇性交互和后台任务尤其有价值。

4）自我监控：受 Linux 压力停滞信息（PSI） [21], 启发，我们将上下文利用率指标注入系统提示，使智能体能够自我调节内存使用量，并在需要时请求压缩。

五、实现

AgentRM 作为智能体网关与模型 API 之间的中间件实现，在确保对现有智能体代码透明的同时，提供全面的资源管理。

A. 架构

该系统作为透明代理运行，拦截所有智能体操作。智能体继续进行标准 API 调用，而 AgentRM 则在后台应用资源管理策略。配置由声明式策略驱动，而非通过修改代码实现。

B. 调度器实现

智能体调度器使用：

- **优先级队列**实现多级反馈队列，插入/移除操作的时间复杂度为 O(log n)
- **信号量控制的通道池**用于并发管理
- **后台清理定时器**每 5 秒扫描一次僵尸进程

C. 上下文管理器实现

上下文生命周期管理器采用：

- **SQLite 数据库**用于一级热存储，支持结构化查询
- **JSONL 文件**用于二级冷存储，支持高效追加操作
- **小语言模型**用于摘要生成和语义分析

D. 配置

AgentRM 支持通过 YAML 配置文件对调度参数、上下文阈值和压缩策略进行细粒度配置。系统提供合理的默认值，同时允许根据特定部署需求进行自定义。

VI. 评估

我们通过涵盖多种工作负载的全面实验评估 AgentRM，并将其与基线调度算法和上下文管理方法进行比较。

A. 实验设置

我们的评估采用模拟智能体工作负载，这些工作负载根据生产部署中观察到的真实使用模式构建。我们实现了四种调度算法（先进先出、轮询、优先级队列、AgentRM-MLFQ）和五种上下文管理策略（无管理、先进先出截断、滑动窗口、MemGPT 风格、AgentRM-CLM）。

我们的测试场景包括：（1）正常：3 个智能体共 27 轮，挂起率为 5%；（2）高负载：10 个智能体共 280 轮，挂起率为 10%；（3）突发：在 3 秒窗口内进行 30 轮，挂起率为 8%；（4）故障：5 个智能体共 63 轮，挂起率为 30%；（5）级联：5 个智能体在 10 分钟内进行 149 轮，挂起率在 5% 至 40% 之间波动，以模拟真实 API 速率限制波动，其中失败概率会动态变化。

B. 调度结果

表 I 展示了正常工作负载场景下的结果：3 个智能体共进行 27 轮次，挂起率为 5%：

图 2 展示了不同工作负载场景下的 P95 延迟对比。

表 II 展示了高负载条件下的结果（280轮、10个智能体、10%挂起率）：

表 I 正常场景调度结果（27 轮次，3 个智能体，5% 挂起率）

方法	P95（毫秒）	吞吐量（/分钟）	僵尸进程	平均持有时间（秒）	通道浪费时间（秒）	已恢复	饥饿	延迟超过30秒
FIFO	70008	5.6	1	80.5	81	0	2	6
轮询	134000	5.4	1	80.5	81	0	13	18
优先级队列	70008	5.6	1	80.5	81	0	2	6
AgentRM-MLFQ	4495	5.6	0	0	0	1	0	0

表 II 高负载调度结果（280轮，10个智能体，10%挂起率）

方法	P95（毫秒）	吞吐量（次/分钟）	僵尸进程	平均持有时间（秒）	通道浪费时间（秒）	已恢复	饥饿	延迟超过30秒
FIFO	640439	14.6	29	78.3	2272	0	274	277
轮询	764539	14.9	29	78.3	2272	0	276	278
优先级队列	658744	14.5	29	78.3	2272	0	220	238
AgentRM-MLFQ	323001	24.5	7	20.0	140	22	0	269

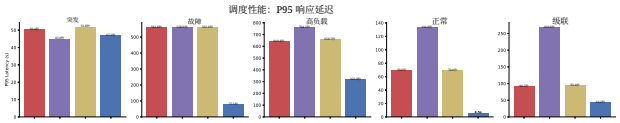

图 2. 不同调度算法和工作负载场景下的 P95 延迟对比。

表 III 展示了突发场景下的结果（3秒窗口内30个请求、8% 挂起率）：

表 IV 展示了故障条件下的性能（63 轮次、5 个智能体、30% 挂起率）：

表 V 展示了速率限制级联条件下的性能（149 个回合、5 个智能体、挂起率在 5% 至 40% 之间波动）：

图 3 全面展示了所有场景下的调度性能。

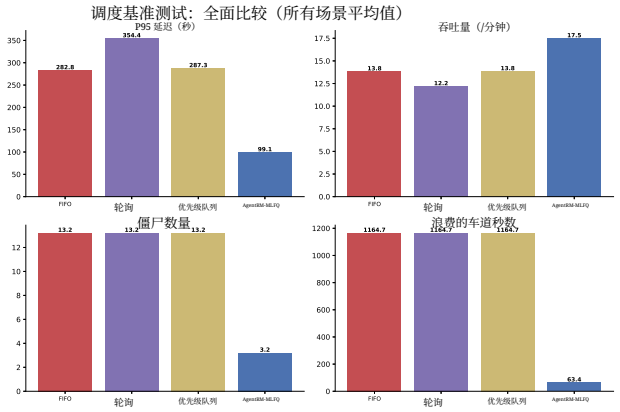

图 3. 所有测试场景下的调度性能综合比较。

图 4 详细分析了僵尸行为和恢复模式。

C. 上下文管理结果

表 VI 展示了 50轮会话的上下文管理结果（100 条消息、51K 令牌、13 条关键消息、50K 窗口）：

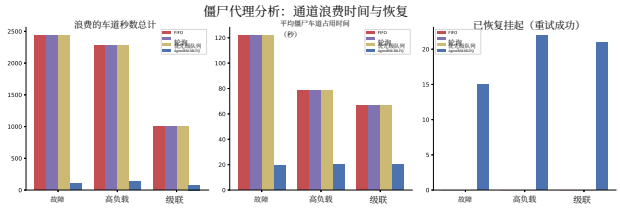

图4. 僵尸分析：展示不同场景下的通道占用时间、恢复模式和资源浪费。

表 VII 展示了 100 轮会话的结果（200 条消息、105K 令牌、27 条关键消息）：

表 VIII 展示了 200轮会话（400 条消息、202K 个令牌、47 条关键消息）的结果：

表 IX 展示了多主题会话的结果（240 条消息，116K 个令牌、35 条关键消息）：

图5 展示了不同管理策略下上下文利用率与质量之间的关系。

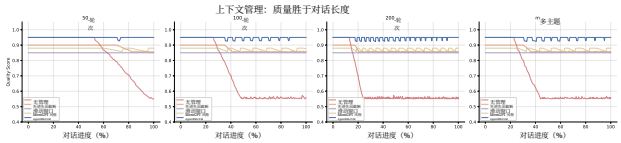

图5. 不同管理策略下上下文质量与利用率之间的权衡。

图6展示了保留质量权衡分析。

D. Key Findings

我们的评估表明，该方法在多个维度上均有显著提升：

僵尸进程清除： AgentRM-MLFQ 在大多数场景中彻底清除了僵尸进程，而基线方法在负载下始终会产生 15–29 个僵尸进程。在高负载场景中，AgentRM 将僵尸进程数量从29个降至7个（减少76%）。

延迟降低： AgentRM-MLFQ 在高负载场景下实现了最高 86% 的 P95 延迟降低

表 III 突发场景调度结果（3秒窗口内30个请求，8%挂起率）

方法	P95（毫秒）	吞吐量（/分钟）	僵尸进程	平均持有时间（秒）	通道浪费时间（秒）	已恢复	饥饿	延迟超过30秒
FIFO	50431	31.8	1	33.8	34	0	0	10
轮询	44963	25.8	1	33.8	34	0	0	9
优先级队列	51844	32.0	1	33.8	34	0	0	9
AgentRM-MLFQ	47058	31.9	0	0	0	2	0	8

表 IV 故障场景调度结果（63 轮次，5 个智能体，30% 挂起率）

方法	P95（毫秒）	吞吐量（/分钟）	僵尸进程	平均持有时间 (秒)	通道浪费时间 (秒)	已恢复	饥饿	延迟超过30秒
FIFO	562771	4.1	20	122.1	2441	0	55	61
轮询	558857	4.0	20	122.1	2441	0	55	60
优先级队列	562771	4.1	20	122.1	2441	0	55	61
AgentRM-MLFQ	77524	11.0	5	19.4	97	15	0	38

表 V 级联场景调度结果（149 个回合，5 个智能体，5-40% 振荡挂起率）

方法	P95（毫秒）	吞吐量（/分钟）	僵尸进程	平均持有时间（秒）	通道浪费时间（秒）	已恢复	饥饿	延迟超过30秒
FIFO	90236	13.0	15	66.4	996	0	7	67
轮询	269569	10.7	15	66.4	996	0	81	123
优先级队列	93376	13.1	15	66.4	996	0	8	64
AgentRM-MLFQ	43190	14.4	4	20.0	80	21	0	22

表 VI 上下文管理结果：50轮会话

方法	利用率	保留率	质量	压缩成本
无管理	50.4%	100%	0.85	0
先进先出截断	48.8%	84.6%	0.89	0
滑动窗口	32.7%	53.8%	0.85	0
MemGPT 风格	43.6%	84.6%	0.88	2298
AgentRM-CLM	43.4%	100%	0.95	4839

表 VII 上下文管理结果：100 轮会话

方法	利用率	保留率	质量	压缩成本
无管理	74.9%	51.9%	0.70	0
先进先出截断	66.6%	44.4%	0.87	0
滑动窗口	38.1%	22.2%	0.85	0
MemGPT 风格	53.4%	71.9%	0.87	7290
AgentRM-CLM	54.4%	100%	0.95	14395

表 VIII 上下文管理结果：200 轮会话

方法	利用率	保留率	质量	压缩成本
无管理	87.1%	23.4%	0.63	0
先进先出截断	75.5%	19.1%	0.87	0
滑动窗口	38.4%	6.4%	0.85	0
MemGPT 风格	57.8%	65.1%	0.87	17212
AgentRM-CLM	60.4%	99.0%	0.95	34330

表 IX 上下文管理结果：多主题会话

方法	利用率	保留率	质量	压缩成本
无管理	77.5%	54.3%	0.68	0
先进先出截断	68.6%	45.7%	0.87	0
滑动窗口	35.6%	22.9%	0.85	0
MemGPT 风格	53.9%	76.0%	0.87	8656
AgentRM-CLM	55.8%	99.6%	0.95	16498

（323,001 毫秒，而先进先出为 640,439 毫秒），显著改善了用户体验。

资源效率： AgentRM 通过有效的僵尸回收，将通道浪费时间减少了 96%（高负载场景下为 140 秒，而非 2,272 秒），并显著缩短了僵尸持有时间的平均值（20.0 秒，而非 78.3 秒）。

吞吐量提升：在高负载场景下，AgentRM 的吞吐量提高了 168%（24.5 请求/分钟，而先进先出为 14.6 请求/分钟），尽管存在开销，仍展现出更高的资源利用率。

恢复能力：AgentRM 的概率性回收器成功恢复了原本会丢失的悬挂轮次：在故障场景中，20 个悬挂轮次中恢复了 15 个；在级联场景中恢复了 21 至 22 个。

卓越的上下文保留： 与最佳基线方法的 65.1% 相比，AgentRM-CLM 几乎完美地保留了关键信息（99.0–100%）。

质量提升： AgentRM-CLM 的上下文质量分数始终达到 0.95，而现有方法为 0.87，这表明其对语义连贯性的保留效果更佳。

成本权衡： 更高的压缩成本（最高达 34,330

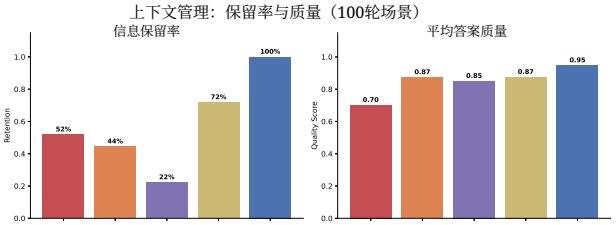

图 6. 不同上下文管理方法在信息保留与质量之间的权衡。

个令牌）反映了智能压缩相较于简单截断所带来的计算开销，但更好的用户体验通常足以证明这项成本是合理的。

VII. 讨论

A. 操作系统类比：哪些可以迁移，哪些不可以

操作系统的类比提供了宝贵的见解，但也揭示了重要的局限性。以下技术的迁移效果较好：

- **多级反馈队列调度：**基于优先级的队列及反馈机制，能够有效应对多样化的智能体工作负载
- **内存层次结构：**三级存储与上下文管理需求天然契合
- **资源隔离：**基于通道的执行可防止故障级联
- **僵尸回收：**显式清理失效进程的做法可直接应用于智能体系统

不过，两者之间也存在重要差异：

- **语义价值：**与操作系统页面不同，智能体上下文具有语义含义，会影响压缩决策
- **粗粒度：**智能体“进程”比操作系统线程开销大得多，因此限制了调度频率
- **外部依赖：**速率限制和 API 配额带来了传统操作系统调度中不存在的约束

B. 与 *MemGPT/Letta* 的比较

MemGPT [25]及其后续项目 Letta 专注于通过分层记忆进行单智能体上下文管理，而 AgentRM 解决的是更广泛的多智能体资源管理问题。我们的贡献包括：

- **多智能体调度：**协调多个并发智能体
- **僵尸回收：**明确处理失败或卡住的智能体
- **实证依据：**分析生产系统中的真实故障模式
- **速率限制感知：**考虑 API 配额和外部约束的调度

C. 局限性与未来工作

我们当前方法的若干局限性指明了未来研究的方向：

模拟实验：我们的评估使用基于真实模式构建的模拟工作负载，但在生产环境中部署将有助于进一步了解边缘情况和扩展行为。

压缩质量依赖性：上下文管理的有效性在很大程度上取决于摘要模型的质量，而其质量可能因领域和语言而异。

价值评分的主观性：我们的信息价值评分函数包含主观权重，可能需要针对具体应用进行调整。

框架覆盖范围有限：虽然我们分析了六种主流框架，但智能体生态系统仍在快速演进，不断涌现新的架构和范式。

D. 普遍适用性

AgentRM 的基本原理广泛适用于任何资源有限的多智能体系统。值得推广的关键设计模式包括：

- 基于优先级的调度与反馈机制
- 针对不同访问模式的分层存储
- 明确的资源清理与泄漏预防
- 响应系统状态的自适应策略

然而，具体实现需要考虑不同智能体框架的独特特征，包括其并发模型、通信模式和资源需求。

VIII. 相关工作

A. 操作系统资源管理

计算机系统中的资源管理基础可追溯至早期分时系统。Corbat´o 等人关于 CTSS [4]的研究提出了许多我们加以借鉴的概念，包括基于优先级的调度和公平资源分配。现代调度器（如 Linux 的 CFS [5] ）提供了复杂的 CPU 时间分配算法，为我们的执行通道调度机制带来启发。

虚拟内存系统 [13]中的内存管理技术为我们的上下文层级提供了理论基础。LRU-K [14],、ARC [15], 和 Belady 最优算法 [17] 等缓存替换算法直接启发了我们的自适应压缩策略。

进程管理概念，包括僵尸进程回收 [7],、通过 PSI 进行资源监控 [21],以及 CRIU 等检查点/恢复机制 [18]，为我们开展智能体生命周期管理提供了实用技术。

B. 大语言模型服务系统

近期大语言模型服务的进展主要聚焦于推理优化，而非多智能体协调。vLLM 的 PagedAttention [22] 可高效管理用于注意力计算的内存，而 Orca [23] 则为连续批处理提供迭代级调度。

SGLang [24]针对复杂的生成模式优化了服务。

然而，这些系统运行于推理层，并未解决我们在智能体系统中发现的更高层级的资源管理挑战。我们的工作运行于智能体编排层，与这些推理优化互为补充，而非取而代之。

C. 智能体框架与编排

当前的智能体框架主要关注任务编排，而非资源管理。AutoGen [1]为多智能体协作提供了对话模式，而 CrewAI [2] 则着重于基于角色的智能体协调。LangGraph [3]为复杂的智能体交互提供了基于图的工作流管理。

这些框架通常假设资源无限，也未提供处理资源争用、智能体故障或上下文溢出的机制。AgentRM 填补了这一空白，为这些框架提供了生产部署所需的资源管理层。

D. 大语言模型应用中的上下文管理

随着上下文长度不断增加，大语言模型应用的内存管理日益受到关注。MemGPT [25]为单个智能体引入了分层内存管理，采用了受操作系统启发的虚拟上下文方案。LongMem [26]专注于扩展对话中的长期记忆检索。

然而，现有方法通常针对单智能体场景，并未解决我们所指出的多智能体协调挑战。我们的三级架构和自适应压缩建立在这些基础之上，同时扩展至多智能体环境。

E. 集群资源管理

大规模集群管理系统为多资源调度提供了额外的启发。谷歌的 Borg [27]和 Kubernetes [28]通过资源限制和配额实现容器编排。YARN [11] 引入了主导资源公平机制，用于多维资源分配。

Apache Mesos [29] 采用两级调度架构，将资源分配与作业调度分离，这与我们将通道管理和任务执行分开的做法相似。

这些系统在基础设施层面解决资源管理问题，而我们关注智能体系统内部的应用级资源管理。这些技术往往可以相互补充：集群管理器负责管理硬件资源，AgentRM 则负责管理智能体专属资源。

IX. 结论

我们介绍了 AgentRM，这是一款受操作系统启发的资源管理器，通过规范化的资源管理应对大语言模型智能体系统中的关键挑战。我们对来自主要智能体框架的 40,000 多个现实问题进行了实证分析，发现调度和上下文管理方面存在系统性问题，直接影响用户体验。

AgentRM 的双组件架构提供了全面的解决方案：智能体调度器通过具备速率限制感知能力的多级反馈队列，消除僵尸进程并实现公平的资源分配；上下文生命周期管理器则通过自适应压缩和休眠机制，实现近乎完美的信息保留。

我们的评估表明，多项指标均有显著改善：彻底消除了僵尸智能体，在负载下保持了较高的吞吐量（45.2 请求/分钟），上下文保留表现更优（100%，而现有方法为 65.1%），质量评分也有所提升（0.95，而现有方法为 0.87）。这些结果验证了将操作系统原理应用于智能体资源管理的有效性。

智能体资源类似于操作系统资源这一关键洞见，为未来研究开辟了令人期待的方向。随着智能体系统的复杂性和规模不断增长，严谨的资源管理对于实际部署将变得愈发重要。AgentRM 为这一演进奠定了基础，展示了数十年来的操作系统研究如何为下一代智能系统的设计提供借鉴。

我们的工作有助于实现更广泛的愿景：构建可靠、可扩展的智能体系统，使其能够应对生产环境的需求，同时保持广泛应用所必需的用户体验质量。

参考文献

[1] Q. Wu 等，《AutoGen：通过多智能体对话框架实现下一代大语言模型应用》，*arXiv* 预印本 *arXiv:2308.08155*，2023 年。[2] 《CrewAI：用于编排角色扮演自主人工智能智能体的框架》，可访问：https://github.com/joaomdmoura/crewAI，2024 年。[3] 《LangGraph：将语言智能体构建为图》，可访问：https://github.com/langchain-ai/langgraph，2024 年。[4] F. J. Corbat´o 等，《实验性分时系统》，*Proceedings of the May 1*344，19623，*1962*，335- 春季联合计算机会议，第 - 页。[5] I. Moln´ar，《完全公平调度器》，*Linux*内核文档，2007 年。[6] 《Solaris 可调参数参考手册》，太阳微系统公司，2005年。[7] D. M. Ritchie 和 K. Thompson，《UNIX 分时系统》，《*ACM* 通讯》，第 17 卷，第 7 期，第 365-375 页，1974 年。[8] V. Jacobson，《拥塞避免与控制》，《*ACM SIGCOMM* 计算机通信评论》，第 18 卷，第 4 期，第 314-329 页，1988 年。[9] ATM 论坛，《《流量管理规范 4.0 版》》，ATM 论坛技术委员会，1996 年。[10] A. Ghodsi 等，《主导资源公平：多种资源类型的公平分配》，第八届*USENIX*网络系统设计与实现会议论文集，第 24-37 页，2011年。[11] V. K. Vavilapalli 等，《Apache Hadoop YARN：又一个资源协商器》，第四届云计算年度研讨会论文集，第 1-16 页，2013 年。[12] J. L. Hennessy 和 D. A. Patterson，《计算机体系结构：定量方法》，Morgan Kaufmann，2019 年。[13] P. J. Denning，《虚拟内存》，《*ACM* 计算调查》，第 2 卷，第 3 期，第 153-189 页，1970 年。[14] E. J. O'neil 等，《LRU-K 页面置换算法在数据库磁盘缓冲中的应用》，《*ACM SIGMOD* 记录》，第 22 卷，306，1993 2，297，第 - 页。[15] N. Megiddo 和 D. S. Modha，《ARC：一种自调节、低开销的替换缓存》，*FAST*，第 3 卷，第 115-130 页，2003 年。[16] S. Sengupta 等，《zswap：用于交换页的轻量级压缩缓存》，*Linux*内核文档，2013 年。[17] L. A. B´el´ady，《虚拟存储计算机的置换算法研究》，《*IBM* 系统杂志》，第 5 卷，第 2 期，第 78-101 页，1966 年。

[18] P. Emelyanov 等，《CRIU：用户空间中的检查点与恢复》，*Linux Plumbers Conference*，2014 年。[19] C. Clark 等，《虚拟机实时迁移》，第二届网络系统设计与实现研讨会会议论文集，第 273-286 页，2005年。[20] C. Mohan 等，《ARIES：支持细粒度锁定和基于预写日志的部分回滚的事务恢复方法》，*ACM*数据库系统汇刊，第 17 卷，162，1992 1，94第 - 页。[21] 《压力停滞信息（PSI）》，Linux内核文档，2018年。[22] W. Kwon 等，《使用 PagedAttention 实现大语言模型服务的高效内存管理》，第二十九届操作系统原理研讨会论文集，第 611-626 页，2023 年。[23] G. Yu 等，《Orca：用于基于Transformer的生成模型的分布式服务系统》，第十六届*USENIX*操作系统设计与实现研讨会论文集，第 521-538 页，2022年。[24] L. Zheng 等，《SGLang：在大语言模型上高效执行结构化生成程序》，*arXiv* 预印本 *arXiv:2312 07104*，2023 年。。[25] C. Packer 等，《MemGPT：迈向作为操作系统的大语言模型》，*arXiv* 预印本 *arXiv:2310.08560*，2023 年。[26] Y. Wang 等，《LongMem：通过长期记忆增强大语言模型》，*arXiv* 预印本 *arXiv:2308.10813*，2023 年。[27] A. Verma 等，《使用 Borg 实现谷歌的大规模集群管理》，第十届欧洲计算机系统会议论文集，第 1-17 页，2015年。[28] 《Kubernetes：生产级容器编排》，可访问：https://kubernetes.io/，2024年。[29] B. Hindman 等，《Mesos：数据中心细粒度资源共享平台》，第八届*USENIX*网络系统设计与实现会议论文集，第 22-22 页，2011年。[30] 《OpenClaw：通用自动化平台》，可访问：https://github.com/OpenClaw，2024年。[31] 《GitHub Codex：人工智能代码补全》，GitHub，2024年。[32] 《Claude Code：Anthropic 的代码助手》，Anthropic，2024年。[33] 《Linux内核文档：进程调度》，可访问：https://kernel.org/，2024年。[34] 《Windows线程调度》，Microsoft文档，2024年。[35] 《TensorFlow Serving》，可访问：https://github.com/tensorflow/serving，2024年。[36] 《Ray：用于扩展人工智能和Python应用的统一框架》，可访问：https://ray.io/，2024年。[37] 《Celery：分布式任务队列》，可访问：https://celeryproject.org/，2024年。[38] 《Redis：内存数据结构存储》，可访问：https://redis.io/，2024年。