arXiv:2601.15335v1 [cs.SE] 20 Jan 2026

ToolCaching：迈向高效的大语言模型工具调用缓存

Yi Zhai、沈典、Junzhou Luo、杨斌东南大学计算机科学与工程学院，中国南京
{zhaiyi,dshen,jluo,binyang}@seu.edu.cn

摘要

大语言模型（LLM）的最新进展革新了 Web 应用，使其能够通过自然语言界面提供智能搜索、推荐和助手服务。工具调用赋予大语言模型与外部 API 交互的能力，大幅提升了其实用性。以往研究通过采用并行执行、异步执行等传统计算机系统技术提升了工具调用性能，但冗余或重复工具调用请求这一挑战仍基本未得到解决。缓存是解决这一问题的经典方案，但由于请求语义各异、工作负载动态变化以及新鲜度要求不同，将其应用于大语言模型工具调用会带来新的困难，使传统缓存策略难以奏效。为解决这些问题，我们提出 ToolCaching，这是一种面向大语言模型工具调用系统的高效、特征驱动的自适应缓存框架。ToolCaching 系统性地整合语义特征与系统级特征，以评估请求可缓存性并估算缓存价值。其核心 VAAC 算法将基于多臂老虎机的准入与价值驱动的多因素驱逐相结合，同时考虑请求频率、近期性和缓存价值。在合成工作负载和公开工具调用工作负载上的大量实验表明，与标准策略相比，采用 VAAC 的 ToolCaching 可将缓存命中率最高提升 11%，并将延迟降低 34%，从而有效加速实际应用中的大语言模型工具调用。

CCS 概念

• **计算方法** → **人工智能**；• **信息系统** → **信息系统应用**。

关键词

缓存、大语言模型系统、工具调用

1 引言

凭借令人印象深刻的能力，大语言模型（LLM）已广泛应用于 Web 生态系统，不仅催生了聊天机器人[4, 11]和代码助手[1, 3], 等新的 Web 应用，还利用先进的人工智能能力赋能搜索、推荐和电子商务等传统 Web 服务。为了进一步扩展大语言模型在这些场景中的功能，工具调用[24, 26, 29]已被引入，使大语言模型能够与外部工具和 API 交互。这一能力显著增强了基于大语言模型的应用的多功能性和实用性，释放了它们执行复杂任务的潜力，例如检索数据、进行计算以及操作外部系统。

随着工具调用的出现，由现代 Web 应用中的大语言模型驱动的、日益复杂的工具链工作流不断涌现，

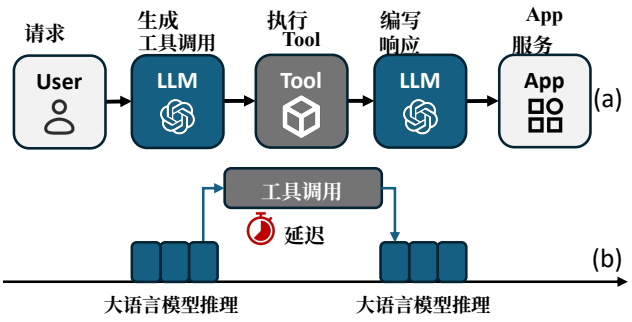

图 1：工具调用工作流及其对应的时间线。

其中，单个模型在多步骤流水线中协调多个外部函数，以执行复杂任务。例如，ReAct[35] 率先将推理与行动相结合，使大语言模型能够在动态推理步骤之间交错执行连续的工具调用。为了优化这些工作流的性能，近期研究致力于提高工具调用执行的效率，例如并行化工具调用[20],、支持异步工具调用[17], ，以及在大语言模型解码[32]的同时并发执行部分工具。

这些技术借鉴了传统计算机系统中的策略，例如并行执行和异步调度，并已展现出性能提升。然而，在大语言模型推理中常见的一种现象——对相似请求进行冗余或重复处理 [21, 25, 36], ——在工具调用工作流中也频繁出现 [28],，而这些方法并未能充分解决这一问题。这促使我们重新审视 Web 应用中广泛采用的另一种经典优化策略：缓存工具调用的结果，以消除冗余执行。尽管如此，大语言模型工具调用的独特特性，例如请求的语义差异、工具成本的异质性，以及与推理系统轻量级集成的需求，使传统缓存策略难以有效应对这些挑战。

首先，工具调用的可缓存性高度不确定：大语言模型会动态生成各种不同的请求，涵盖从信息检索到状态变更操作等类型，因此无法不加区分地进行缓存。其次，新鲜度要求差异很大；单一的生存时间策略要么不安全，要么效率低下。第三，工作负载具有高度动态性：请求模式和热门工具会迅速变化，导致静态策略失效。因此，稳健的缓存需要采用自适应准入与驱逐策略，实时持续响应工作负载特征。

为应对这些复杂问题，我们认识到有两项核心技术挑战：

• **理解大语言模型工具调用请求**。 工具调用具有高度多样性且依赖上下文，因此很难通过传统启发式方法评估其可缓存性。从动态请求中提取语义和系统级特征至关重要，但并非易事。

• **设计自适应缓存策略**。 对于成本和复用模式各异的工作负载而言，命中率或近期性等经典指标并不足够。有效的策略必须整合更丰富的特征，以指导准入与驱逐决策。

这两项挑战构成了我们方法的基础。我们提出 ToolCaching，这是一种缓存框架，将请求语义与系统级信号相结合，以优化大语言模型应用中的工具调用。ToolCaching 将请求的语义分析与轻量级可观测性和缓存内统计信息相结合，从而实现适应工具调用工作负载独特特征的自适应准入与驱逐策略。据我们所知，这是**首个针对大语言模型工具调用缓存问题的研究**。

我们在合成数据集和公开的工具调用工作负载上评估了 ToolCaching。结果表明，与基线策略相比，ToolCaching 将缓存命中率最高提升了 11%，并将端到端延迟最高降低了 34%，展现了其在加速基于大语言模型的 Web 应用方面的有效性。

我们的贡献总结如下：（i）全面分析将缓存应用于大语言模型工具调用所面临的挑战，并提出**ToolCaching**，这是首个整合语义特征与系统特征以实现自适应缓存管理的框架；（ii）识别理解工具调用请求所需的关键特征，并设计轻量级机制来收集这些特征；（iii）开发高效的缓存管理算法 VAAC，利用语义信号和系统级信号指导准入与驱逐决策；（iv）在合成工作负载和公开工作负载上实现并评估 ToolCaching，证明其能够有效提升大语言模型工具调用系统的性能。

2 背景与动机

本节概述我们工作的背景与动机。我们首先介绍大语言模型中的工具调用概念，随后探讨在大语言模型工具调用场景中应用缓存所涉及的固有复杂性，以及缓存准入与驱逐的重要性。

2.1 大语言模型中的工具调用

工具调用是一种让大语言模型与外部工具和 API 交互的机制，使其能够执行超出自身固有能力范围的任务。这一特征已集成到各种大语言模型中，例如 OpenAI 的 GPT-4 [10] 和 Meta 的 Llama 2 [31]，使这些模型能够检索远程数据源、执行计算并操控外部系统。除了简单的单步调用外，工具调用还支持构建复杂的多步骤工作流。例如，ReAct[35] 展示了大语言模型如何将思维链推理与工具使用交替结合，动态规划并执行一系列操作以完成复杂任务。

无论任务复杂程度如何，工具调用工作流都遵循一致的模式：如图 1a 所示，大语言模型会解析用户请求，生成结构化工具调用，并将其发送至工具的 API。返回结果会作为额外上下文反馈，用于组织响应，最终再传递给应用服务层。

如图 1b 所示，工具调用插入大语言模型推理的两个阶段之间，其执行时间可能是端到端延迟的主要来源；尤其是在通过网络调用远程 API 时 [24, 29]。先前研究 [17, 20, 32] 通过调整大语言模型推理与工具执行之间的调度、支持并行处理以及设计更高效的流水线，致力于优化工具调用性能。

除了这些执行级优化外，我们的实证分析揭示了工具调用工作负载中存在大量冗余。在电影推荐数据集 [20, 30],中，我们发现超过 40% 的工具调用会重复，这表明重复利用先前结果具有巨大潜力。冗余也经常出现在单个用户会话中——例如，用户可能先请求“评分最高的 5 部电影”，系统随即检索并描述一份影片列表；随后用户又请求“评分最高的 5 部科幻电影”，由于列表中有重叠影片，系统便会重复调用工具以获取相同影片的详细信息。这类重复调用不仅会延长响应时间，还会增加 API 使用相关的财务成本和资源成本，因为许多工具按调用次数收费或消耗有限配额。表 1 总结了具有代表性的工具任务及其典型延迟和每 1,000 次调用的成本，凸显了通过**缓存工具调用请求的结果**消除冗余调用的潜在优势。

2.2 大语言模型工具调用中的缓存

缓存长期以来一直是传统 Web 应用中用于缓解冗余计算和网络开销的有效策略。近期大语言模型推理领域的 KV-Cache 等进展[21, 25, 36] 进一步展现了其在加速系统性能方面的强大作用。受这些成功经验的启发，我们将这一范式扩展到大语言模型工具调用：通过将工具调用的结果存储在缓存中，并直接从缓存条目中为后续相同的请求提供服务，我们的方法能够大幅减少重复的工具执行，降低外部通信成本，并提升整体端到端效率。

然而，大语言模型及其工具调用机制的独特特性带来了固有的复杂性，使缓存系统的设计与传统 Web 应用中的设计有着根本区别：

表 1：具有典型延迟和成本的代表性工具任务。

任务类型	延迟（毫秒）	成本（$/1000 次调用）
搜索 [5]	700–2000	~5.0
维基百科抓取 [8]	200–1000	0
地图规划 [6]	50–1000	5.0
天气查询 [7]	~200	1.6

工具调用的可缓存性不确定： 大语言模型会动态生成工具调用——从本地函数执行到远程 API 调用，涵盖信息检索或状态变更操作等各种用途。有些调用需要实时数据，另一些则可能触发不可逆副作用（例如发送消息或进行交易）。显然，缓存所有工具调用并不可行：结果可能很快过时，而不加区分地缓存命令可能导致错误或不安全的行为。因此，要准确判断某个工具调用是否适合且值得缓存，需要深入理解每个请求。

结果新鲜度差异显著： 工具调用结果的新鲜度要求差异极大——百科查询可能数周内都有效，而金融数据或传感器读数可能几秒钟后就会过期。采用统一的缓存过期时间（生存时间）要么造成浪费，要么带来风险。要有效管理缓存陈旧问题，需要进行细粒度的自适应生存时间分配，以反映每个结果的具体有效期；在工作负载多样且不断变化的情况下，这尤其具有挑战性。

快速变化且难以预测的工作负载： 大语言模型工具调用系统经常面临快速变化的工作负载——请求频率、热门工具和用户访问模式都会随着用户行为或外部事件而波动。这种波动使静态缓存分区和固定替换策略难以奏效：当前热门的内容很快可能变冷，因此必须动态重新分配缓存资源。因此，稳健的缓存管理必须持续监控工作负载特征，并实时调整缓存策略。

2.3 缓存管理策略

鉴于大语言模型工具调用工作负载的特点，缓存管理策略的设计（包括准入和驱逐机制）对于实现有效缓存至关重要。准入控制决定哪些工具调用结果可以进入缓存，而驱逐策略则规定在容量受限时替换哪些条目。

LRU（最近最少使用）和 LFU（最不经常使用）等传统方法已广泛应用于许多系统中 [15, 16, 39], , 但它们并不适用于大语言模型工具调用场景，因为每次工具调用的执行成本和延迟都不可忽略且差异很大。这些策略主要依赖访问近期性或使用频率，却没有明确考虑工具执行成本高且不均衡这一特点。近期研究表明，自适应、工作负载感知的准入与驱逐策略能够显著提升命中率、资源利用率和端到端性能 [18, 33, 34]。

这凸显了对有效且具备工作负载感知能力的缓存管理策略的需求；该策略能够在准入和驱逐决策中自适应地纳入工具调用的执行成本与延迟特征。

针对这些复杂性和困难，并受到近期缓存领域进展的启发，我们提出了一种用于大语言模型工具调用的新型缓存系统，名为 ToolCaching。

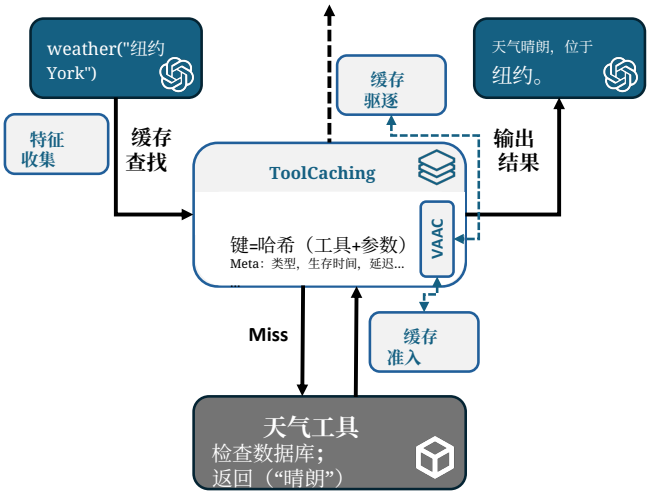

图 2：ToolCaching 的工作流程。

3 设计

本节介绍 ToolCaching 的设计。如第 2 节所述，大语言模型工具调用中的缓存面临两项独特挑战。为解决这些问题，ToolCaching 围绕两个核心组件进行设计：

• **特征收集。** ToolCaching 首先通过提取工具调用请求的语义属性和系统级指标，对其进行全面分析。这些特征通过基于大语言模型的语义理解和轻量级系统监控获得，为制定明智的缓存决策奠定了基础。

• **缓存管理算法。** 基于收集到的特征，ToolCaching 采用了一种名为 VAAC（价值感知自适应缓存）的自适应缓存算法。VAAC 集成了两种关键机制：（i）由基于特征的分组和缓存价值引导的选择性缓存准入（v-CACA）；（ii）综合考虑近期性、使用频率和执行成本的多因子驱逐（v-LRU）。这种设计使 ToolCaching 能够动态适应异构且不断变化的工作负载。

ToolCaching 的工作流程如图 2 所示。ToolCaching 会拦截来自大语言模型的 API请求。如果请求已在缓存中，ToolCaching 会检索缓存结果并将其返回给大语言模型。如果请求是新的，ToolCaching 会将其转发给工具 API，从请求中提取语义特征，并判断其可缓存性。如果判定为可缓存，则会收集系统级特征并评估缓存价值。ToolCaching 根据该价值和其他特征，决定是否将结果存入缓存，或淘汰现有条目。无论结果是否被缓存，都会将其返回给大语言模型以供进一步处理。如果请求不可缓存，ToolCaching 只会将请求转发给工具 API，并将结果返回给大语言模型。

在接下来的章节中，我们将讨论各类别的细节，并说明如何克服第 2 节中提到的复杂性。

4 特征收集

鉴于上述大语言模型工具调用场景中的缓存复杂性，在设计有效的缓存策略之前，全面了解工具调用请求至关重要。本节将说明需要收集哪些特征，以及如何收集这些特征。

4.1 语义特征

判断大语言模型应用中的工具调用请求是否可缓存，本质上是一个语义问题。与传统网页缓存不同，传统网页缓存通常可以根据静态资源类型或 HTTP 元数据推断可缓存性，而大语言模型工具调用工作负载由动态生成、上下文丰富且具有不同特征的请求驱动。我们选择了以下特征，并说明其原因。

请求类型： 工具调用通过支持与外部 API 交互来扩展大语言模型的能力，而这些 API 可以服务于截然不同的目的。单个工具可能同时支持信息检索 和 命令/操作 语义。例如，消息工具可用于发送消息：message(send;Mike;"Hello")，也可用于读取最新消息：message(read;last message)。

这一区别对缓存至关重要：检索类型的请求通常没有副作用，因此适合在重复出现时复用；而命令类型的请求往往会引发不可逆的状态变化，使其结果不适合或无意义地进行缓存。因此，我们将工具调用请求分为两大类：**信息查询**（例如，天气查询、知识库搜索）和**命令**（例如，发送消息、发起事务）。为避免有害副作用，命令请求不纳入缓存，而信息型请求则可作为复用候选项。因此，准确识别请求类型成为判断可缓存性的关键特征，并直接纳入 ToolCaching 的准入控制。

参数类别： 该特征用于根据输入参数对工具调用请求进行分组。例如，在天气查询工具中，可以依据位置或日期等属性对请求进行组织（例如 location=伦敦，date=2024-05-01），进而作为请求分组的依据，并使缓存管理能够灵活适应（详见第 5 节）。

在本研究中，我们采用一种实用的启发式方法：对于多参数工具调用，将第一个参数视为分组的主要类别；对于单参数函数，则不再按参数进行进一步分组。尽管本文采用了简单的启发式方法，但基于参数的分类并非固定不变；在缓存系统中注册每个工具时，可以自定义分组规则，以更好地反映其语义和使用模式。

TTL（生存时间）： 在传统缓存系统中，TTL 是控制数据陈旧性并确保缓存内容有效性的基本机制。我们将这一概念应用于大语言模型工具调用，其中 TTL 指定工具调用结果的建议有效期，并直接决定缓存过期策略。

生存时间较短的工具调用（例如实时传感器读数）会迅速过时，因此不太适合缓存；而生存时间较长的工具调用（例如静态知识查询）则非常适合重复使用。在 ToolCaching 中，我们定义了以下建议默认值：0s COMMAND 请求为 60s，实时数据为 60s，计算结果为 300s，静态知识为 3600s。这些值反映了常见的时间模式，并可针对每种工具进行自定义，或根据工作负载特征动态调整。

综合考虑这些属性后，通过针对每个工具调用请求提示大语言模型，即可获得相应的特征集。附录 A 展示了用于特征收集的提示模板示例。

实践中我们发现，借助清晰的指令和具有代表性的示例进行提示工程，能够有效发挥大语言模型的语义分析能力，并生成准确且一致的特征提取结果。如表 2 所示，其准确率接近 90%，为下游缓存管理奠定了坚实基础。

4.2 系统特征

一旦某个工具调用请求被判定为可缓存，ToolCaching 就会收集一组系统级特征，以评估保留其结果的潜在收益。这些特征有两个用途：（i）估算每个条目的缓存价值；（ii）在动态工作负载下驱动自适应准入与驱逐策略。

我们将系统特征分为两类：

缓存内统计信息，由缓存系统直接维护，且开销极低：

关联用户： 此特征记录访问过每个请求的用户集合。在大语言模型工具调用场景中，不同用户通常会表现出访问特定工具或查询的不同模式。通过跟踪关联用户，ToolCaching 可以根据用户兴趣有效地对请求进行分组，因为用户可能会反复调用相似或相同的工具调用。例如，某位用户可能经常请求个性化电影推荐，而另一位用户则持续查询金融数据。采用基于用户的分组可以显著提高缓存命中率，确保缓存结果与特定用户的请求模式高度匹配。

访问次数： 每个缓存工具调用结果的缓存命中总次数。这一常见的缓存指标有助于 ToolCaching 快速识别经常被请求的条目（例如热门的特定知识查询），从而直接为缓存准入优先级提供依据。

结果大小： 缓存工具调用响应的大小。在大语言模型工具调用场景中，某些请求（例如检索大型数据集）产生的结果会比其他请求大得多。跟踪结果大小有助于 ToolCaching 避免让过大的条目占用不成比例的缓存空间，从而在命中率与整体缓存效率之间实现平衡。

系统延迟： 完成一次工具调用请求所需的端到端时间，包括网络传输和工具执行延迟。ToolCaching 直接在缓存系统中测量此延迟，从而识别出延迟较高的操作，并将其作为优先缓存的对象。优先缓存观测到的延迟较高的结果，可减少重复的慢速调用，从而提升用户感知响应速度。

系统级指标，通过应用程序级测量或底层监控获得：

观测到的资源成本：每个工具调用请求的实测执行成本，可以按本地计算所消耗的中央处理器资源，或外部应用程序接口调用的计费价格来衡量。在大语言模型工具调用中，不同工具的资源成本差异显著；昂贵的本地计算（例如个性化推荐排名）或高成本的第三方应用程序接口（例如法律文档检索），显然值得赋予更高的缓存价值。ToolCaching 利用这一特性，优先保留能够显著节省计算开销或应用程序接口调用费用的结果。

对于难以在应用层收集或收集成本较高的特征，例如本地 API 的中央处理器使用率，我们采用轻量级系统可观测性工具，例如 eBPF（扩展伯克利数据包过滤器）[2],，该工具 [27, 37]广泛用于内核级性能跟踪。这些工具能够以极低的运行时开销，按请求进行资源分析。如有需要，还可以通过类似技术捕获其他低级指标（例如内存使用量、磁盘输入/输出、函数调用），从而进一步为缓存决策提供依据。

5 高效缓存管理

本节将讨论如何基于收集到的特征设计高效的缓存管理策略。

5.1 缓存结构设计

我们首先介绍缓存结构的设计，其中涵盖缓存键构建和值存储。这一基础设计支持特征感知准入和值驱动的优先级排序，为后续章节所讨论的自适应管理策略奠定基础。

缓存键构建：为确保缓存查找高效且准确，我们将工具名称与其输入参数的序列化表示组合起来，为每个工具调用请求构建一个唯一的键。序列化过程经过设计，不受顺序影响，并支持复杂的嵌套参数结构（例如 JSON 对象），从而确保逻辑上等效的请求生成相同的键。随后对这个复合字符串进行哈希处理，生成紧凑且抗碰撞的标识符。例如，检索 2024年5月1日纽约天气的请求会生成一个类似 hash(weather:location=NewYork, date=2024-05-01) 的键。这种方法能够在各种大语言模型工具调用场景中实现快速查找、准确区分请求以及稳健的缓存管理。

缓存值存储： 每个缓存条目都会存储工具调用的结果及其相关元数据。元数据包括第 4 节所述的相关语义特征和系统特征，以及用于指导缓存管理决策的价值评分。该价值评分的计算方式将在后文讨论。

5.2 可缓存性分析

在缓存工具调用请求之前，必须确定该请求是否可缓存。我们利用一组语义特征来评估可缓存性，包括请求类型、参数类别和生存时间。具体而言，我们采用以下规则：(1) COMMAND 类型的请求不可缓存；(2) 生存时间较短（≤ 60 秒）的请求不可缓存。

尽管这些标准基于大语言模型生成的结构化语义特征，但我们刻意不让大语言模型本身来决定可缓存性。基于定义明确的特征进行基于规则的判断，可确保确定性、可解释性和效率；而将这一决策交给大语言模型，则可能引入不确定性和潜在错误。过于复杂或含糊的大语言模型驱动分析，可能导致缓存状态不一致和性能下降。因此，我们的方法将大语言模型的语义提取能力与透明的、基于规则的可缓存性评估相结合，以最大限度地兼顾表达能力与可靠性。

5.3 缓存价值模型

除了缓存系统中的标准指标请求访问次数外，我们还明确考虑在大语言模型工具调用场景中处理每个请求的总体成本。这包括结果大小、系统延迟和资源成本。通过整合这些维度，我们的缓存价值模型能够更准确地反映缓存技术在复杂且异构环境中的实际性能收益。

为确保这些特征具有可比性，且不会因尺度差异而主导缓存价值，我们对每项指标应用最小-最大归一化。归一化过程会重新缩放请求 r_i 的每个特征 j，如下所示：

$$\text{NormFeature}_i^{(j)} = \frac{Feature_i^{(j)} - Feature_{min}^{(j)}}{Feature_{max}^{(j)} - Feature_{min}^{(j)}} \tag{1}$$

其中，$Feature_i^{(j)}$ 表示请求 r_i的特征 j（例如延迟、大小或成本）的取值，$Feature_{(j)min}$和 $Feature_{(j)max}$分别是该特征观测到的最小值和最大值。

然后，我们可以将请求 r_i的缓存价值 v_i定义如下：

$$v_i = \lambda_1 \cdot \text{NormLatency}_i + \lambda_2 \cdot \frac{\text{NormCost}_i}{\text{NormSize}_i} - \lambda_3 \cdot e^{-\frac{\text{TTL}_i}{\tau}} \tag{2}$$

其中，λ_1、λ_2和 λ_3 是可调权重，用于控制延迟、成本与大小之比以及基于生存时间的风险在缓存价值模型中的相对重要性。τ 是一个平滑参数，表示缓存条目的平均生存时间。较小的生存时间$_i$ 会使指数项变大，表明过时风险较高；而较大的生存时间$_i$ 则会使风险值变小，反映出更强的时间稳定性。

直观而言，该模型会优先处理延迟较高、成本较大且结果大小较小的请求（因为它们能节省系统资源），同时降低对生存时间较短、容易迅速失效的请求的优先级。这种对性能提升与时间稳定性的平衡，使 ToolCaching 能够缓存既经常被访问、又能长期持续带来价值的条目。

5.4 缓存准入

缓存准入决定哪些请求进入缓存。在 ToolCaching 中，我们利用第 4 节收集的特征，设计了一种基于强化学习的准入策略，其灵感来自 CACA框架 [18],，能够自适应地调整以应对工作负载变化。

CACA 首先根据内容特征将请求划分为不同组，然后采用一种强化学习方法，具体而言是一种多臂老虎机算法，动态调整每个组的准入策略。通过跟踪各组的使用频率和命中率，CACA 仅选择性地准入表现最佳的

组，将有限的缓存资源集中用于最有可能被重复使用且最有利于系统的请求。这种方法非常适合大语言模型工具调用工作负载高度动态且多样化的特点。

然而，原始的 CACA 仅考虑请求频率和命中率，忽略了执行成本这一大语言模型工具调用场景中的关键因素。我们将缓存价值模型纳入多臂老虎机的奖励函数，将 CACA 扩展为 v-CACA，使请求模式和执行成本都能指导缓存准入。

因此，ToolCaching 中的缓存准入策略可概括如下：

初始状态： 在初始阶段，只要有可用空间，所有可缓存的工具调用请求都会被准入。系统会收集并更新每个请求的特征，尤其是在缓存命中或复用时。

特征分组： 为管理工具调用请求多样且动态的特性，ToolCaching 使用固定的三级特征集，按特征对请求进行分组：（1）工具类型、（2）参数类别和（3）用户身份。请求首先按工具类型划分，随后（在需要时）按参数类别进一步细分，最后按用户身份划分，以捕捉细粒度的局部性模式。该层级以自适应方式构建：仅当某个组的访问频率超过阈值 T_1 且命中率低于 H^r 时，才会将其拆分；否则，该组保持在当前层级。

分组列表 G 中的每个结果组 g_i 都维护自己的聚合统计数据，包括其缓存命中率 H_i 和平均缓存价值 V_i。这种分层特征分组提高了强化学习决策的稳定性，并使缓存能够适应不断变化的工作负载。完整的分组逻辑、相应的层级示意图和伪代码见附录 B。

准入控制： 现在，我们有一组特征组 G，每个特征组都有自己的平均缓存价值 V_i 和命中率 H_i。下一步是确定哪些特征组应准入缓存。

这个问题可以表述为一个多臂老虎机问题，其中每个特征组 g_i 都是一只臂，目标是最大化随时间累积的奖励。与 CACA 类似，我们考虑命中率H_i、节点层级 L_i 以及该组被准入缓存的次数 C_i；但鉴于工具调用请求的特性，我们还考虑组内请求的平均缓存价值 V_i。因此，每个特征组 g_i的奖励函数定义如下：

$$F_i = \frac{\log(H_i + \delta_1) \cdot \log(L_i + \delta_2) \cdot \log(V_i + \delta_3)}{\log(C_i + \delta_4)} \tag{3}$$

其中，δ_1、δ_2、δ_3 和 δ_4 是较小的常数，用于避免除以零以及对零取对数。

为解决这一多臂老虎机问题，我们采用 UCB1 算法 [12], ，该算法通过选择上置信界最高的臂，在探索与利用之间取得平衡。基于 UCB1 的准入控制的详细信息见附录 B。

5.5 缓存驱逐

缓存驱逐决定在需要腾出空间时移除哪些条目。在 ToolCaching 中，我们基于经典的 LRU（最近最少使用）策略，提出了 v-LRU，这是一种多因素淘汰策略

它综合考虑近期性和式（2）中定义的缓存价值。

v-LRU 的流程如下：

生存时间过期： 生存时间已明确到期的条目会立即失效。尽管基于生存时间的过时风险已纳入缓存价值 v_i，显式处理过期仍可确保正确性，并防止访问过时结果。

条目驱逐： 需要驱逐时，我们会找出缓存中近期性最低的 10% 条目（即最近最少使用的条目）。随后，从该子集中移除驱逐分数 e_i最低的条目，其定义如下：

$$e_i = \log(v_i + h_i + \delta_5) \tag{4}$$

其中，v_i 表示缓存价值，h_i 表示命中率，δ_5 是用于确保数值稳定性的小常数。

我们的策略将缓存价值模型与自适应准入（v-CACA）和多因子驱逐（v-LRU）相结合，为大语言模型工具调用提供有效的缓存管理，共同构成<styles id='1'>价值感知自适应缓存（VAAC）算法。

6 评估

本节将评估我们在 ToolCaching 中的特征提取和缓存管理算法的有效性。我们首先评估大语言模型从工具调用请求中提取语义特征的准确率，然后使用模拟工作负载和公开大语言模型工具调用数据集评估我们的缓存管理算法的性能。对于公开数据集实验，我们基于大语言模型编译器框架实现缓存评估。缓存服务器运行在一台配备 16GB 内存和 8 个中央处理器的机器上，用于语义特征提取和大语言模型编译器的大语言模型是基于云端的 DeepSeek V3 [22, 38]。

6.1 语义特征准确率

虽然语义特征是使用大语言模型提取的，但一个关键问题随之而来：大语言模型在提取工具调用请求的语义特征时，准确率有多高?

为了回答这个问题，我们评估了大语言模型对一组工具调用请求进行语义特征提取的能力。具体而言，我们从伯克利函数调用排行榜（BFCL）数据集 [23] 的 BFCL_v3_simple 子集中提取了 50 个工具调用请求。我们对验证集中的每个请求都进行了人工标注，标明其对应的请求类型（命令或信息查询）以及估算的生存时间（TTL）。这些标注作为真实标注，用于评估大语言模型自动分类和 TTL 预测的准确率。为避免因工具调用解析错误或格式错误而造成混淆，我们

表2：大语言模型对工具调用请求进行语义特征提取的准确率。

特征	准确率	精确率	召回率	F1
请求类型	0.980	0.968	1.000	0.984
TTL	0.920	0.864	0.864	0.864
参数类别	1	1	1	1

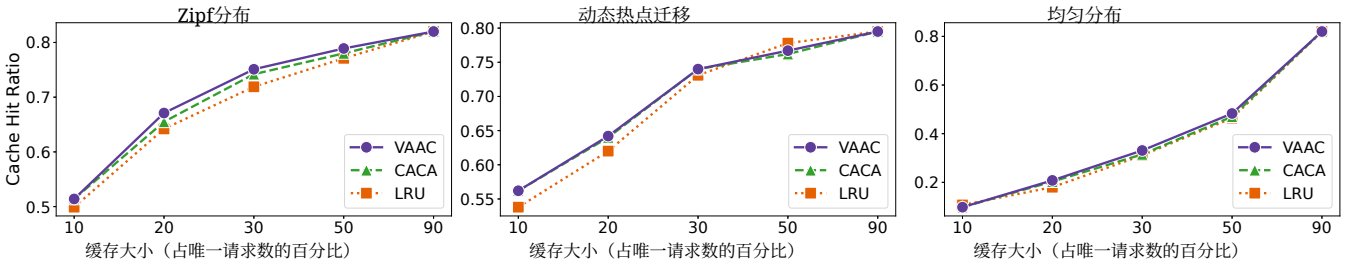

图3：不同缓存大小和工作负载下的缓存命中率

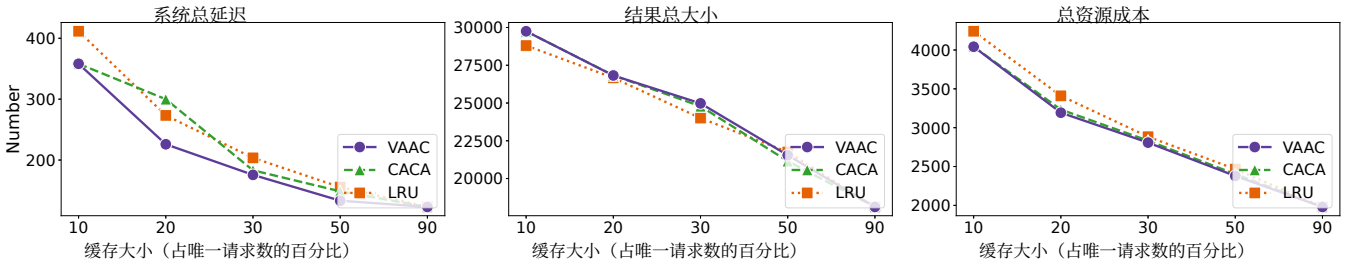

图4：Zipf分布下不同缓存大小对应的请求总延迟、结果大小和资源成本

仅将大语言模型生成了正确工具调用格式的情况纳入评估。因此，由于我们的系统使用每次工具调用的第一个参数作为主要类别，参数分组的正确率始终为 1。

我们使用附录A中展示的相同提示模板，并针对每个语义属性计算标准分类指标，包括准确率、精确率、召回率和F1分数。

如表2所示，大语言模型在从工具调用请求中提取语义特征方面达到了很高的准确率。请求类型分类几乎完美，而生存时间预测也表现出很高的准确率。这些结果表明，大语言模型能够为下游缓存管理提供可靠的语义标注。

6.2 缓存管理

由于缺乏能够准确反映真实世界大语言模型工具调用工作负载模式的公开数据集，我们构建了一个合成数据集，用于评估缓存管理算法的性能。

为了全面评估缓存管理算法的有效性，我们构造了三种类型的合成工作负载，每种包含 1,000 个请求，涵盖 6 种不同的工具，旨在覆盖多种参数

表3：有无基于用户的分组时缓存命中率和延迟的比较。

缓存大小	10%	20%	30%
有用户	18.8 / 563.28	32.2 / 465.60	52.1 / 346.32
无用户	15.5 / 568.88	30.2 / 501.04	51.2 / 363.62

类型、响应大小、生存时间和资源成本，确保广泛覆盖典型的工具调用场景：

- **Zipf分布：**请求按照 Zipf分布 ($\alpha = 1.1$) 进行采样，以模拟真实世界工作负载的局部性 [9, 13, 14, 18]。
- **动态热点迁移：**工作负载会定期在不同的热点区域之间切换。在每个阶段，活动热点内的请求频率遵循 Zipf分布，而其他请求则分布在其余工具之间。
- **均匀分布：**在所有工具和参数组合中均匀随机生成请求，作为基线，用于评估局部性极低情况下的缓存策略性能。

我们使用 Python 实现了用于模拟工作负载的缓存服务器，代码共 ~1400 行，并在 5 种不同的缓存大小下评估缓存命中率：占所有唯一请求的 10%、20%、35%、50% 和 90%。我们设置了 $\lambda_1 = 0.8$、$\lambda_2 = 0.2$、$\lambda_3 = 0.2$、$T_1 = 20$ 和 $H_r = 0.5$。

对于每种缓存大小，我们将 VAAC 算法与两种基线算法进行比较：CACA（接纳请求时不考虑其价值，并采用 LRU 淘汰策略）和 LRU（将所有请求都接纳到缓存中）。结果如图 3 所示。如图所示，在大多数缓存大小和工作负载下，VAAC 的表现始终优于 CACA 和 LRU，最高提升达 11%。这证明了我们的缓存价值模型和多因素淘汰策略在优化缓存性能方面的有效性。

具体而言，图 4 展示了 Zipf 分布工作负载下，不同缓存大小对应的请求总系统延迟、结果大小和资源成本。如图所示，与 CACA 和 LRU 相比，VAAC 的总延迟和资源成本最低，分别降低了

表4：ToolCaching在大语言模型编译器中的有效性

数据集	缓存大小（调用次数）	缓存命中率	延迟（秒）	提升
电影推荐	0%	—	16.2	—
	20%	0.3781	13.81	1.14×
	50%	0.503	11.88	1.26×
	100%	0.514	10.7	1.34×
ParallelQA	0%	—	15.9	—
	20%	0.232	13.08	1.17×
	50%	0.265	12.99	1.18×
	100%	0.2866	12.18	1.23×

最高可达 17.3% 和 6.4%。对于结果总大小，由于我们的缓存价值模型会在 2 中惩罚较大的结果大小，VAAC 达到最高结果大小，因为高价值请求更有可能被拒绝和驱逐，这与我们最大化缓存效用的设计目标一致。

6.3 多用户缓存效果

由于多用户场景在实际的大语言模型应用中十分常见，我们通过模拟一个包含 10 名用户的工作负载，在真实环境下评估 ToolCaching。每位用户都会发出兴趣部分重叠的工具调用请求。我们比较了两种缓存管理策略：（1）全局分组（不区分用户）和（2）基于用户的分组（我们的方法）。表3中的结果显示，基于用户的分组可使缓存命中率最高提升 21.3%，并使平均延迟降低 7.1%。这表明，ToolCaching 能有效适应特定用户的访问模式，从而提升用户感知的性能。

6.4 大语言模型工具调用系统中的端到端评估

工具调用的发展使大语言模型能够根据上下文选择并协调多个函数，从而解决日益复杂的问题。为进一步支持此类应用，LLM Compiler [20] 等框架为多步推理和工具编排提供了优化。

为评估 ToolCaching 在高级大语言模型工具调用场景中的实际效果，我们将缓存管理系统集成到 LLM Compiler 框架中，并进行端到端评估。我们使用 LLM Compiler 提供的以下数据集：

- **电影推荐：** 该数据集包含 500 个样例，每个样例要求从四部候选电影中找出与由四部电影组成的参考集最相似的一部。该任务呈现出 8 路极易并行的执行模式，每个候选项都可以单独评估。
- **ParallelQA：** ParallelQA 是一个专门用于评估复杂依赖关系下工具调用场景的定制基准。它包含 113 个与数学相关、涉及事实属性的问题，每项任务都需要按顺序使用两个工具（例如先搜索，再进行数学计算），且第二个工具的输入取决于第一个工具的输出。所有必需信息都包含在相关维基百科条目的第一段中。

我们将 ToolCaching 与未使用缓存的基线 LLM Compiler 框架进行性能比较。结果汇总于表 4。如表中所示，随着缓存大小增加，ToolCaching 始终能够降低端到端延迟。在电影推荐数据集中，ToolCaching 的延迟最多可降低 34%。同样，在 ParallelQA 中，延迟最多可降低 23%。这些结果表明，ToolCaching 能够有效减少冗余工具调用，并显著提升复杂大语言模型工具调用场景中的系统效率。

6.5 开销

我们还测量了在大语言模型编译器场景下 ToolCaching 的开销。与不使用缓存的基线大语言模型编译器框架相比，ToolCaching 的中央处理器开销为 ~ 15%，内存开销为 ~ 10 %。中央处理器开销主要来自 VAAC 算法的计算，而内存开销则源于缓存条目及其相关元数据的存储。鉴于缓存带来了显著的性能提升，这些开销是可以接受的。

7 结论

本文提出了 ToolCaching，这是一种面向大语言模型工具调用系统的高效缓存框架。ToolCaching 的核心是 VAAC 算法，该算法融合了语义和系统级特征，将请求动态划分为不同组，并结合基于多臂老虎机的准入策略与感知价值的 LRU 淘汰机制。通过广泛的实验，我们证明，与传统的静态策略和基于频率的策略相比，采用 VAAC 的 ToolCaching 实现了更高的缓存命中率和更低的延迟。我们的结果进一步表明，ToolCaching 能够有效适应动态工作负载和不断变化的结果有效性，凸显了缓存对于复杂大语言模型环境的价值。

参考文献

[1] [无日期]. Cursor - 人工智能代码编辑器。https://cursor.com/。[2] [无日期]。eBPF。https://ebpf.io。[3] [无日期]。GitHub Copilot。https://github.com/features/copilot。[4] 2025。ChatGPT。https://chatgpt.com/。[5] 2025。Google自定义搜索JSON API。https://developers.google.com/custom- search/v1/overview?hl=zh-cn。

[6] 2025。Google 地图平台定价。https://developers.google.com/maps/billing-and-pricing/pricing。[7] 2025。定价 - OpenWeatherMap。https://openweathermap.org/price。[8] 2025。维基媒体 API。https://api.wikimedia.org。[9] Bahman Abolhassani、John Tadrous 和 Atilla Eryilmaz。2023。无线边缘动态内容的最优负载拆分与分布式缓存。IEEE/ACM 网络汇刊 31，5（2023年10月），2178–2190。doi:10. 1109/TNET.2023.3244039[10] Josh Achiam、Steven Adler、Sandhini Agarwal、Lama Ahmad、Ilge Akkaya、Florencia Leoni Aleman、Diogo Almeida、Janko Altenschmidt、Sam Altman、Shyamal Anadkat 等。2023。GPT-4 技术报告。arXiv 预印本 arXiv:2303.08774（2023 2303 08774）。arXiv: 。Daniel Adiwardana、Minh-Thang Luong、[11] David R. So、Jamie Hall、Noah Fiedel、Romal Thoppilan、Zi Yang、Apoorv Kulshreshtha、Gaurav Nemade、Yifeng Lu 和 Quoc V. Le。2020。迈向类人开放域聊天机器人。arXiv: 2001 09977 [ ] 10 48550 2001 09977 。cs doi: 。/arXiv. 。Peter Auer、Nicolò Cesa-Bianchi 和 Paul Fischer。2002。多臂老虎机问题的有限时间分析。《机器学习》47，2-3（2002年5月），235–256。doi:10.1023/a:1013689704352[13] L. Breslau、Pei Cao、Li Fan、G. Phillips 和 S. Shenker。1999。Web 缓存与 Zipf 型分布：证据与启示。载于 IEEE INFOCOM '99。《计算机通信会议论文集》。IEEE 计算机与通信学会第十八届年度联合会议。未来已来（目录号 *99 36320* 1 126 134 1 10 1109 1999 749260 CH），第 卷。– 卷。doi: 。/INFCOM. 。Yong Deng 和 Min Dong。 。非均匀需求下编码缓存最优缓存放置的基本结构。IEEE 信息论汇刊 68，10（2022年10月），6528–6547。doi:10.1109/TIT.2022. 3179266[15] Gil Einziger、Roy Friedman 和 Ben Manes。2017。TinyLFU：一种高效的缓存准入策略。ACM 存储汇刊 13，4（2017年11月），35:1–35:31。doi:10.1145/3149371[16] Christine Fricker、Philippe Robert 和 James Roberts。2012。LRU 缓存性能的一种通用且精确的近似方法。载于 2012 年第 24 届国际电信业务大会（ITC 24）。1–8。[17] In Gim、Seung-Seob Lee 和 Lin Zhong。2024。异步大语言模型函数调用。CoRR（2024年1月）。[18] Yu Guan、Xinggong Zhang 和 Zongming Guo。2019。CACA：面向边缘缓存中视频内容的基于学习的内容感知缓存准入。载于第 27 届 ACM 多媒体国际会议论文集（MM '19）。计算机协会，纽约，NY，美国，456–464。doi:10.1145/3343031.3350890[19] Xinyi Hou、Yanjie Zhao、Shenao Wang 和 Haoyu Wang。2025。模型上下文协议（MCP）：现状、安全威胁与未来研究方向。arXiv: 2503 23278 [ ] 10 48550 2503 23278 。cs doi: 。/arXiv. 。Sehoon [20] Kim、Suhong Moon、Ryan Tabrizi、Nicholas Lee、Michael W. Mahoney、Kurt Keutzer 和 Amir Gholami。2024。用于并行函数调用的大语言模型编译器。载于第四十一届国际机器学习会议。[21] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Sheng、Lianmin Zheng、Cody Hao Yu、Joseph Gonzalez、Hao Zhang 和 Ion Stoica。2023。使用 PagedAttention 高效管理大语言模型服务的内存。载于第 29 届操作系统原理研讨会论文集。ACM，德国科布伦茨，611 626 10 1145 3600006 3613165 – 。doi: 。/ 。Aixin Liu、[22] Bei Feng、Bing Xue、Bingxuan Wang、Bochao Wu、Chengda Lu、Chenggang Zhao、Chengqi Deng、Chenyu Zhang、Chong Ruan 等。2024。DeepSeek-V3 技术报告。arXiv 预印本 arXiv: *2412 19437* 2024 2412 19437 。（）。[23] arXiv: 。Shishir G. Patil、Huanzhi Mao、Charlie Cheng-Jie Ji、Fanjia Yan、Vishnu Suresh、Ion Stoica 和 Joseph E. Gonzalez。2025。伯克利函数调用排行榜（BFCL）：从工具使用到大语言模型智能体评测。载于第四十二届国际机器学习会议。[24] Shishir G. Patil、Tianjun Zhang、Xin Wang 和 Joseph E. Gonzalez。2024。Gorilla：连接海量 API 的大语言模型。神经信息处理系统进展 37（2024），126544–126565。[25] Ramya Prabhu、Ajay Nayak、Jayashree Mohan、Ramachandran Ramjee 和 Ashish Panwar。2025。vAttention：无需 PagedAttention 的大语言模型服务动态内存管理。载于第 30 届 ACM 编程语言与操作系统架构支持国际会议论文集，第 1 卷（ASPLOS '25）。计算机协会，纽约，NY，美国，1133 1150 10 1145 3669940 3707256 – 。doi: 。/ 。Timo Schick、[26] Jane Dwivedi-Yu、Roberto Dessi、Roberta Raileanu、Maria Lomeli、Eric Hambro、Luke Zettlemoyer、Nicola Cancedda 和 Thomas Scialom。2023。Toolformer：语言模型能够自学使用工具。神经信息处理系统进展 36（2023年12月），68539–68551。[27] Junxian Shen、Han Zhang、Yang Xiang、Xingang Shi、Xinrui Li、Yunxi Shen、Zijian Zhang、Yongxiang Wu、Xia Yin 和 Jilong Wang。2023。以网络为中心的分布式追踪：借助 DeepFlow 实现微服务零代码故障排查。载于 ACM SIGCOMM 2023 会议论文集。420–437。[28] Simranjit Singh 、 Michael Fore 、 Andreas Karatzas、Chaehong Lee、Yanan Jian、Longfei Shangguan、Fuxun Yu、Iraklis Anagnostopoulos 和 Dimitrios Stamoulis。2024。LLM-dCache：利用 GPT 驱动的本地化数据缓存改进工具增强型大语言模型。在 *2024*年第*31*届*IEEE*国际电子、电路与系统会议（*ICECS*）。1–4。doi:10.1109/ICECS61496.2024.10848749[29] Yifan Song、Weimin Xiong、Dawei Zhu、Wenhao Wu、Han Qian、Mingbo Song、Hailiang Huang、Cheng Li、Ke Wang、Rong Yao、Ye Tian 和 Sujian Li。2023。RestGPT：连接大语言模型与现实世界的 RESTful API。arXiv:2306.06624 [cs] doi:10.48550/arXiv.2306.06624[30] Aarohi Srivastava、Abhinav Rastogi、Abhishek Rao、Abu Awal Shoeb、Abubakar Abid、Adam Fisch、Adam R Brown、Adam Santoro、Aditya Gupta、Adri Garriga-Alonso 等。2023。超越模仿游戏：量化并外推语言模型的能力。机器学习研究汇刊（2023）。[31] Hugo Touvron、Louis Martin、Kevin Stone、Peter Albert、Amjad Almahairi、Yasmine Babaei、Nikolay Bashlykov、Soumya Batra、Prajjwal Bhargava、Shruti Bhosale 等。2023。Llama 2：开放基础模型与微调聊天模型。*arXiv*预印本 *arXiv*: *2307 09288* 2023 2307 09288 。（）。arXiv: 。Yechen Xu、[32] Xinhao Kong、Tingjun Chen 和 Danyang Zhuo。2024。Conveyor：借助工具部分执行实现高效的工具感知型大语言模型服务。（2024年10月）。[33] Juncheng Yang、Ziming Mao、Yao Yue 和 K. V. Rashmi。2023。{GL-Cache}：面向高效、高性能缓存的组级学习。在 第*21*届*USENIX*文件与存储技术会议（*FAST 23*）。115–134。[34] Juncheng Yang、Yao Yue 和 Rashmi Vinayak。2021。Segcache：一种面向小对象、内存高效且可扩展的内存键值缓存。在 第*18*届*USENIX*网络系统设计与实现研讨会（*NSDI 21*）。503–518。[35] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Karthik R. Narasimhan 和 Yuan Cao。2022。ReAct：在语言模型中协同推理与行动。在第十一届国际学习表征会议。[36] Gyeong-In Yu、Joo Seong Jeong、Geon-Woo Kim、Soojeong Kim 和 Byung-Gon Chun。2022。Orca：面向基于{Transformer}的生成模型的分布式服务系统。在 第*16*届*USENIX*操作系统设计与实现研讨会（*OSDI 22*）。521–538。[37] Yi Zhai、Junzhou Luo 和 Jianrui Liu。2025。NRCAC：面向云服务提供商的非侵入式微服务根因分析框架。在 *IEEE INFOCOM 2025*——*IEEE*计算机通信会议。1–10。doi:10.1109/INFOCOM55648.2025.11044716[38] Chenggang Zhao、Chengqi Deng、Chong Ruan、Damai Dai、Huazuo Gao、Jiashi Li、Liyue Zhang、Panpan Huang、Shangyan Zhou、Shirong Ma、Wenfeng Liang、Ying He、Yuqing Wang、Yuxuan Liu 和 Y.X. Wei。2025。深入了解DeepSeek-V3：人工智能架构的扩展挑战与硬件反思。在 第*52*届年度国际计算机体系结构研讨会论文集。ACM，东京，日本，1731–1745。doi:10.1145/3695053.3731412[39] Yikai Zhao、Wenrui Liu、Fenghao Dong、Tong Yang、Yuanpeng Li、Kaicheng Yang、Zirui Liu、Zhengyi Jia 和 Yongqiang Yang。2023。P4LRU：迈向完全运行于可编程数据平面中的 LRU 缓存。在 *ACM SIGCOMM 2023*会议论文集（*ACM SIGCOMM '23*）。计算机协会，纽约州纽约市，美国，967–980。doi:10.1145/3603269.3604813

A 语义特征提取详 ails

图 5 展示了我们用于基于大语言模型的特征提取的详细提示模板。

B 缓存准入详情

B.1 层次化特征分组

我们采用层次化的自适应分组策略。请求首先按工具类型划分。对于每个工具组，我们持续跟踪访问次数和缓存命中率等聚合统计数据。如果某个组的使用频率较高（请求访问次数 $\geq T_1$），但命中率较低（$\leq H^r$），这表明其多样性较大，我们便进一步按参数类别细分。如果基于参数的分组仍不足以捕捉同质性，我们则按用户 ID 细化分组。这在多用户环境中特别有用，因为不同用户的访问模式可能存在显著差异。

通过这一递归过程，请求被组织成树结构（图 6），每一层对应一个特定的特征维度。叶节点表示由缓存管理的同质请求簇。例如，一个叶节点如

```
# 提示类型： 系统# 输入：<其他系统输入> ● 使用工具时，请按以下格式作答：缓存_分析： {{ "请求_类型": "信息查询或命令", "参数_类别": string, "建议的_生存时间_秒": number }} ● 请根据以下原则分析缓存： ● 请求¯类型（请求类型）： - 信息查询：查询或信息检索操作（例如计算、翻译、天气查询）- 命令：状态变更操作（例如发送电子邮件、删除文件、资金转账） ● 参数_类别（参数类别）： - 返回工具的主要参数；对于只有一个参数的工具，返回“none”● 建议的_生存时间_秒（生存时间）： - 根据信息时效性和请求类型确定 - 命令为 0 秒，实时数据为 60 秒，计算为 300 秒，静态知识为 3600 秒。● 请根据你对请求内容的理解，智能分析每次工具调用的缓存特征。
```

图5：用于大语言模型语义特征提取的提示模板。

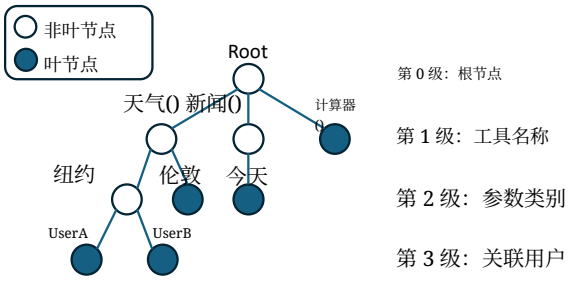

图6：用于工具调用请求的分层特征分组。

root/weather()/New York/UserA 表示一个用于“天气”工具的组，其参数为“纽约”，用户为“UserA”。为防止过度碎片化，请求数量过少的组会与其父组进行合并。

这一由使用频率和命中率引导的动态细化过程，使系统能够通过自适应地将分组粒度与现实世界的工作负载模式相匹配，最大限度地提高缓存效用。同时，这也有助于控制系统复杂度。

经过特征分组后，会形成一个特征组列表：

$$\mathcal{G} = \{g_1, g_2, \ldots, g_n\} \tag{5}$$

其中每个组 g_i 都包含特征相似的请求。对于每个组 g_i，我们都会维护一组统计数据，包括请求的平均缓存价值 $V_i = \frac{1}{|g_i|} \sum_{r_j \in g_i} v_j$ 以及缓存命中率 $\overline{H_i}$。

如果一个组中的请求（$< S_{min}$）过少，我们会将其与父组合并，以减少组的数量。此外，该组会在 B 个请求后定期重置，以适应变化的工作负载。

我们在算法1中给出了周期性层次特征分组的详细算法。

算法1 用于工具调用请求的周期性层次化特征分组请求

输入： 请求流 R、特征维度 F、阈值 T_1（频率），H^r（命中率）、最小组大小 $S_{\min}$、重新分组间隔 B
输出： 叶特征组 G

初始化 G $\leftarrow \emptyset$
初始化请求缓冲区 $R_{all} \leftarrow \emptyset$，计数器 $cnt \leftarrow 0$
过程： 周期性分组(F, B)
  对每个传入请求 r **执行**
    将 r 添加到 R_{all}, $cnt \leftarrow cnt + 1$
    如果 cnt 取模 $B == 0$ **则**
      分组结果 $\leftarrow \emptyset$
      层次分组(R_{all}, F , 1)
      // 现在 G 存储最新的分组结果
过程： 层次分组(R, F , level)
  如果 层级 $> |F|$ **或** $|R| < S_{\min}$ **则**
    将 R 作为一个组添加到 G
    返回
  按特征 F [level] 将 R 划分为子组 $\{R_1, R_2, \ldots\}$
  对于每个子组 R_j **执行**
    计算访问次数 f_j、命中率 H_j
    如果 $f_j \geq T_1$ **且** $H_j \leq H^r$ **则**
      层次分组(R_j, F , 层级 + 1)
    否则，如果 $|R_j| < S_{\min}$ **则**
      将 R_j 添加到父组（跳过拆分）
    else
      将 R_j 作为叶组添加到 G
输出： 最新叶特征组 G（每隔 B 次请求重新-请求）

算法 2基于 UCB1 的算法进行缓存准入

输入： 层次划分得到的叶特征组 G $=\{g_1, g_2, \ldots\ldots, g_n\}$ 划分；每个 g_i的当前统计数据
输出： 每个决策轮次的准入决策

过程： UCB1 准入(G)
  对于 每个决策轮次 t **执行**
    对于所有特征组 $g_i \in$ G **执行**
      更新 g_i的统计信息：命中率 H_i，节点层级 L_i，admission count C_i，平均缓存价值 V_i，选择次数 N_i
      使用公式 (3) 计算奖励 F_i
      使用公式 (6) 计算 $UCB_i = \overline{F_i} + c\sqrt{\frac{\ln t}{N_i}}$
    选择 $g_{i^*} = \arg\max_{g_i} UCB_i$
    将 g_{i^*} 中的请求加入缓存
    更新 $C_{i^*} \leftarrow C_{i^*} + 1$，$N_{i^*} \leftarrow N_{i^*} + 1$
结束过程

B.2 基于 UCB1 的准入控制

我们采用UCB1算法 [12] 来解决缓存准入中的多臂老虎机问题。每个决策轮次中，我们根据公式3计算每个分组的奖励 F_i，并选择UCB值最高的分组：

$$UCB_i = F_i + c\sqrt{\frac{\ln t}{N_i}} \tag{6}$$

其中， N_i 是组 i 被选中的次数， t 是当前轮次， c 是探索参数。

基于 UCB1 的准入控制伪代码见算法 2。

C 关于局限性和未来工作的讨论

C.1 局限性

尽管 ToolCaching 在大语言模型工具调用场景中展现出显著的性能提升，但它主要面向< 스타일 id='1'>信息型工具调用，这类调用的结果不会改变系统状态。然而，随着**模型上下文协议（MCP）** [19], 的日益普及，大语言模型正逐渐具备调用**命令**类型工具的能力，这些工具会执行改变状态或不可逆的操作。

如第 4 节所述，将传统的缓存策略应用于此类调用可能会导致有害的副作用。因此，未来的缓存框架必须明确考虑命令调用的语义和副作用特征，采用与信息型请求所使用的机制根本不同的方法。

C.2 未来工作

未来，我们计划扩展 ToolCaching，以支持更多样化的工具调用模式、多租户场景，以及与多智能体框架的集成，尤其是探索跨智能体工具调用模式中的缓存机会。此外，我们还计划探索如何将缓存安全且有效地应用于**命令**类型的工具调用，从而提升 MCP 场景中的性能。