RePrism：联合请求 与大语言模型智能体服务的表示调度

Xijiang Shen[1] Aoli Zhu[1] Yanbo Tan[2] Xiyuan Hu[1] Huailiang Tan[1],*

[1]湖南大学 [2]上海交通大学

xjshen@hnu.edu.cn zhuaoli@hnu.edu.cn bayernmunich520@sjtu.edu.cn
huxiyuan556@gmail.com *tanhuailiang@hnu.edu.cn（通讯作者）

arXiv:2611.32545v1 [cs.DC] 24 Nov 2026

摘要

智能体工作流通常会多次调用大语言模型。当依赖关系对服务系统可见时，系统便可利用这些依赖关系来调度请求并缓存可复用内容，从而提高工作流执行效率。以往工作主要聚焦于在固定缓存方案下对请求进行排序或管理共享资源。然而，指定模型行为的可复用提示词可能具有多种表示。我们提出了 RePrism，这是一种服务系统，可联合选择要服务的就绪请求以及其提示词所使用的符合条件的表示。其目标是提高吞吐量并缩短会话完成时间（SCT）。调度器通过结合每个请求-表示对的即时价值，以及基于可见工作流对其未来价值的估计，为请求与表示的组合进行排序。资源管理器为 LoRA适配器和缓存的键值状态分配 GPU 内存，并随着内存压力的变化协调二者的准入与驱逐。我们在到达率为每秒 1.8 个会话的动态工作负载下，将 RePrism 与 S-LoRA 进行比较；结果显示，其请求吞吐量达到 1.37×，并将 P95 SCT 降低了 4.10 倍。在受控负载下，相较于 S-LoRA，RePrism 将首个令牌时间缩短了 5.26 倍。在受控工作负载下，RePrism 的联合评分使请求吞吐量比先对请求排序、再选择其表示的策略高出 2.7%，平均 SCT 降低 12.3%。

关键词： 智能体工作流；大语言模型服务；提示词表示；LoRA适配器；KV 缓存；

1 引言

大语言模型（LLM）的应用如今已超越单轮问答。为处理更复杂的任务，越来越多的应用使用智能体工作流来协调多个大语言模型调用 [13, 23, 49, 52]。我们关注这样的工作流：其中各调用之间的数据依赖关系和控制依赖关系在执行前已知，或者智能体框架可以针对每个会话，将其表示为静态或部分静态的有向无环图（DAG）[20, 26, 44]。只有当前置节点完成且其数据依赖关系得到满足后，调用才能运行。用户关心的是整个会话何时完成，因此评估服务系统时，除了请求延迟和首个令牌时间（TTFT）外，还需要使用会话级指标，例如会话完成时间（SCT） [9, 29]。可见工作流揭示了依赖关系和潜在的复用机会，为调度器提供有关未来请求的信息 [12, 31]。

智能体工作负载中的上下文具有不同范围的复用方式。用于指定行为规则和输出格式的系统提示词或角色提示词相对稳定，既可在会话内复用，也可跨会话复用 [10, 42]。此类固定提示词可能包含数百甚至数千个令牌 [31],，而执行期间生成的上下文也可能被后继节点使用。重新处理这些内容会产生重复的预填充成本 [53, 55], ，而保留所有可复用的缓存又会消耗稀缺的 GPU 内存。服务系统必须在重新计算与内存成本之间进行权衡。

指定模型行为的提示词可以采用多种表示方式。*Direct* 会针对每个请求重新处理完整提示词，并且不保留任何状态以供复用。*KV* 会保留完整前缀的键值状态，在相同前缀再次出现时避免重新计算。*Adapter* 使用离线训练的 LoRA 权重，以实现提示词所指定的行为。*Hybrid* 在 Adapter 的基础上，还会保留残余上下文的键值状态。四种表示方式需要在计算、存储以及获取状态的成本之间进行权衡 [3,7,14,21,58]。Direct 和 KV 可视为精确执行路径。Adapter 和 Hybrid 属于参数化近似，需要满足行为质量要求，但无法保证与精确路径逐令牌一致。对于给定请求，四种表示方式中可能只有一部分符合条件，因为提示词指定的某些行为无法内化到适配器参数中。

以往研究主要沿着三个相对独立的方向展开。前缀缓存系统通过优化缓存、驱逐和预取的内容来减少重复的预填充计算，并可利用工作流结构预测未来的复用 [12, 46]。多 LoRA 服务系统则优化批处理、适配器驻留和 GPU 内存管理 [37, 39]。提示词内化方法专注于离线构建固定提示词的参数化表示，使模型在在线推理期间表现出指定行为 [7]。这些方-

法主要解决给定表示的执行与缓存管理问题，或该表示的离线构建问题。当同一请求存在多个表示时，服务系统必须同时决定下一步执行哪个就绪请求，以及选择哪个表示来最有效地推进工作流。

我们的关键观察是，请求调度与表示选择紧密耦合。这种耦合会影响调度器如何比较不同请求之间的执行成本。如果它

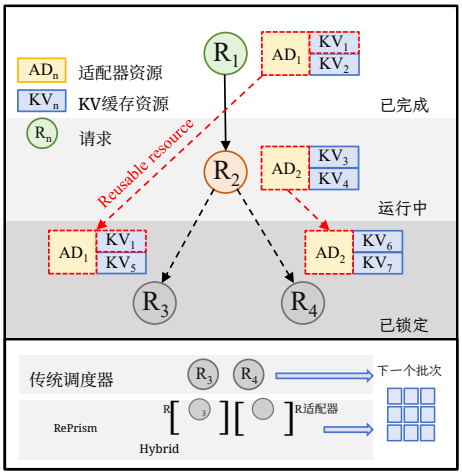

图1：工作流复用与批次选择。有向无环图展示了后续请求可以复用哪些缓存资源。RePrism 将下一批次中的每个请求与一个提示表示配对。

首先根据执行每个就绪请求对其所属会话的价值对请求进行排序，并且只有在确定顺序后才选择表示，那么指定表示所对应的计算和资源成本就无法影响该排序。表示的选择还会影响后续调用可用的资源。保留较大的键值缓存可能会驱逐后继请求即将复用的缓存，而选择适配器表示则可能需要重新处理残余输入。因此，降低当前请求延迟的表示并不总能缩短会话完成时间。为了做出在线决策，调度器必须考虑当前执行成本、预测的复用情况，以及共享 GPU 内存的争用。

我们提出了 RePrism，这是一个为智能体工作流联合调度请求并选择其表示的系统。其目标是在有限的 GPU 内存预算下，保持较高的请求吞吐量并缩短 SCT。如图1所示，RePrism 将每个就绪请求与其每个符合条件的表示进行配对。联合调度器和资源管理器协调候选项选择与共享缓存管理，以推进会话。调度器利用可见的工作流结构对配对进行排序，资源管理器则协调可复用资源的准入与驱逐。

我们在 S-LoRA 上实现了 RePrism，并在搭载 Llama-2-7B-Chat 的 NVIDIA RTX 3090 GPU 上对其进行评估。评估涵盖三种应用模板，并包括动态工作负载和受控工作负载。在动态工作负载下，当到达率为每秒 1.8 个会话时，RePrism 的请求吞吐量达到 S-LoRA 的 1.37× 倍，并将 P95 SCT 降低至原来的 1/4.10。在受控工作负载下，与 S-LoRA 相比，RePrism 将平均 TTFT 降低了 5.26 倍。与在对请求进行排序后选择表示相比，联合评分使请求吞吐量提高了 2.7%，并将平均 SCT 降低了 12.3%。

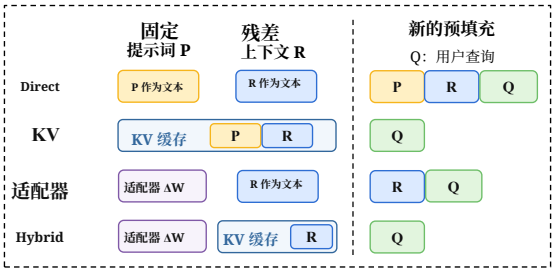

图 2：Direct、KV、适配器和 Hybrid 的缓存资源与预填充输入。

本文作出了三项贡献：

- 我们刻画了不同提示表示之间计算与 GPU 内存的权衡，并展示复用模式和内存压力如何影响表示选择。

- 我们提出 RePrism，它利用工作流信息，联合选择一个就绪请求及其提示词的一种合格表示。RePrism 在共享的 GPU 内存预算下管理可复用资源。

- 我们将 RePrism 与具有代表性的大语言模型服务系统进行比较评估。消融实验和敏感性研究揭示了联合调度的优势，以及它能够发挥作用的条件。

2 背景

智能体工作流与会话目标。传统的在线大语言模型服务侧重于单个请求 [1,21,43,54,60]。智能体应用依赖多个相互依赖的调用来实现一个用户目标 [20, 49, 51],，而用户始终关心的是整个任务是否完成，而非单个请求。我们将从任务到达到最终输出的整个执行过程称为一个会话。对于会话 s，图 $G_s = (V_s, E_s)$ 描述了调度器当前可见的依赖关系。每个节点$v \in V_s$ 表示一次大语言模型调用或一个逻辑执行阶段。一条边 $(u, v) \in E_s$ 表示节点 v 依赖节点 u提供数据或控制信息。可见工作流展示了各节点在图中的位置，并帮助调度器估计未来的复用情况 [12, 26, 29]。不同的就绪节点对会话推进的作用各不相同。完成一个节点可能会解除多个后继节点的阻塞，也可能因条件分支而改变关键路径。当后续调用及其可复用上下文已知时，调度器便可预判复用情况，并决定保留或加载哪些缓存资源。

提示表示的权衡。 智能体工作流中的请求通常包含稳定前缀和随会话或调用变化的输入 [10, 31, 42, 58]。我们根据复用范围将输入分解为 $P \,\|\, R \,\|Q$。 P 是用于指定模型行为和输出格式、可在多个会话间复用的系统提示词或角色提示词。R 是由

当前会话或执行路径生成的上下文。同一会话中的后继节点通常会共享它，但很少会跨会话复用。Q是特定于单次调用的查询，几乎没有复用机会。

图 2 展示了四种运行时表示。Direct 处理 $P \| R \| Q$，并在请求完成后释放其缓存。KV 保留共享前缀在预填充期间计算出的键值张量，以便匹配的请求可以从后缀处继续。适配器使用离线提示内化来训练 LoRA 权重 ΔW_P，使其引发与显式提示 [14],所诱导的相似行为，并且在运行时只需预填充 $R \| Q$。适配器大小取决于张量维度，包括秩和适配层，并且不会随提示长度直接扩展。Hybrid 结合了两种复用生命周期。它用适配器表示 P，并缓存基础模型在加载该适配器时为R 计算出的残余KV张量。当两者均命中时，只需处理Q。与适配器相比，Hybrid 在命中时节省了重新处理 R 的成本，但会产生构建和存储残余KV缓存的成本。因此，必须权衡缓存命中的收益与加载或重建成本，以及留给其他请求的内存。在有限的 GPU 内存预算下提高请求吞吐量并缩短 SCT，需要同时考虑请求顺序及其表示。

3 提示表示的特征分析

本节介绍 Adapter 和 Hybrid 的行为资格认证，衡量四种表示的计算成本和 GPU 内存成本，并评估用于调度实验的代理适配器。这些结果为第 4 节中的候选项构建和在线表示选择提供依据。第 5.1 节介绍实验设置。

3.1 提示内化与行为资格认证

内化目标。借鉴上下文蒸馏 [3,8,59] 和固定输入参数化 [7,41],，我们定义基础模型的两种配置。教师模型将 P 作为显式输入，而学生模型省略P，并使用可训练的 LoRA 权重。如图 3 所示，我们使用数据集的真实答案离线训练适配器，然后检查其是否满足P的行为要求。对于未见过的输入$R \| Q$，学生模型应当在相应的$P \| R \| Q$上近似教师模型的行为。

行为资格认证。 我们将训练好的适配器加载到基础模型中，并在独立测试集上对其进行评估。其行为得分是满足训练前设定的结构和行为需求的输出所占的比例。我们根据 Schema-Guided Dialogue（SGD）数据集 [35] 构造固定提示词。对于 Media_1 提示词（1004 个令牌），秩为 4 和 8 的适配器得分低于 0.85 的预设阈值，而秩为 16 的适配器得分为 0.917，达到资格要求。更高的秩可以编码更多信息，但长度只能部分反映信息量，因此更长的提示词不一定需要更高的秩。基础模型的能力也会限制其响应提示词的方式，因此，同一个提示词在不同模型上可能会 elicite 完全不同的行为。对于后续实验，我们将秩为 4、8、16 和 32 的适配器分别与长度为 378、733、1004 和 1723 个令牌的固定提示词 P 配对。补充材料的 B节给出了训练细节和结果。

3.2 延迟与 GPU内存使用

对于使用合格适配器的提示词P，我们比较使用 Direct、键值、适配器或 Hybrid 为同一请求提供服务时的计算和缓存成本。

图 4 比较了P 长度变化时热路径上的四种表示；报告的P长度包含模型的聊天模板。R、Q和O 的长度分别为 256、256 和 128 个令牌。在全部四种配置中，Direct 均作为基线。键值、适配器和 Hybrid 都避免了$P \| R$的部分重复预填充，在图 4(a) 中 TTFT 加速比峰值超过 6×。随着P 长度增加，Direct 的预填充成本以及表示选择带来的收益都会提高。图 4(b) 表明，在测试的P长度范围内，输出生成将端到端延迟加速比限制在最高约为 1.18×。对于较短的P，适配器和 Hybrid 的端到端延迟高于键值；但随着P 变长，它们会变得更快。使用键值时，每个解码步骤都会关注所有先前的令牌。适配器和 Hybrid 将P 编码进 LoRA 权重中，因此需要关注的令牌更少。

这些计算收益对应着不同的 GPU 内存成本。对于图 4(c) 中最长的提示词，键值占用 1000 MiB 的 GPU 内存，而适配器和 Hybrid 分别占用 64 MiB 和 192 MiB。适配器的内存使用量最低，但其 TTFT 加速比仅约为键值或 Hybrid 的一半。键值实现了最高的 TTFT 加速比，但其内存使用量约为 Hybrid 的五倍，且端到端延迟高于适配器和 Hybrid。因此，Hybrid 比键值使用更少的 GPU 内存，同时比适配器实现更高的 TTFT 加速比。我们测量了每种表示单独注册时所占用的 GPU 内存，但这些数值未计入请求之间共享所带来的节省。例如，当请求共享 P 但具有不同的 R 时，键值和 Hybrid 都可以复用它们为 P 保留的内容，只需重建与 R相关的部分。

Hybrid 并不总是优于适配器。其热路径节省的时间必须与构建和保留残余键值缓存的成本权衡。考虑一个秩16适配器、一个 256 令牌的查询（Q）以及一个 128 令牌的输出（O）。对于长度为 64、128、256 和 512 令牌的残余上下文，首次构建并注册残余键值缓存分别需要 25.28、37.33、65.47 和 140.79 毫秒。缓存驻留期间，相应的缓存资源会额外占用 32、64、128 和 256 MiB 的 GPU 内存。在相同长度下，之后每次命中相较于适配器都可节省 29.77、33.69、70.18 和 120.25 毫秒的

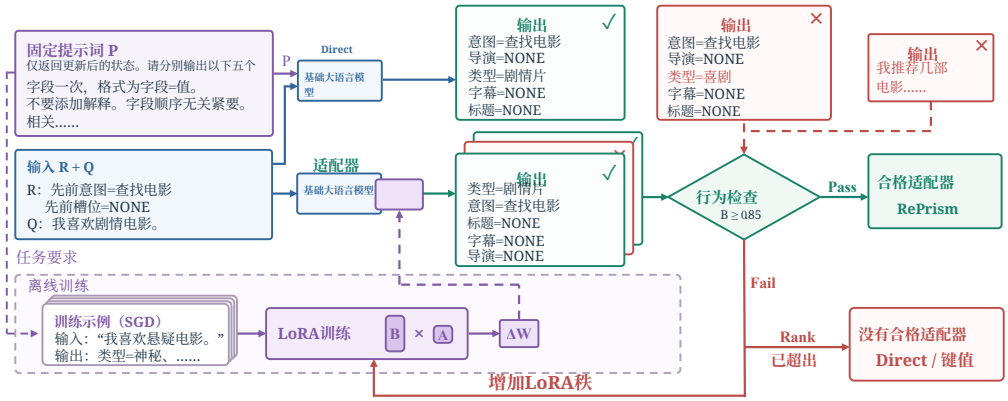

图 3：LoRA适配器的提示内化与行为资格认证。当适配器的行为得分至少为 0.85 时，即可通过其提示词的资格认证。

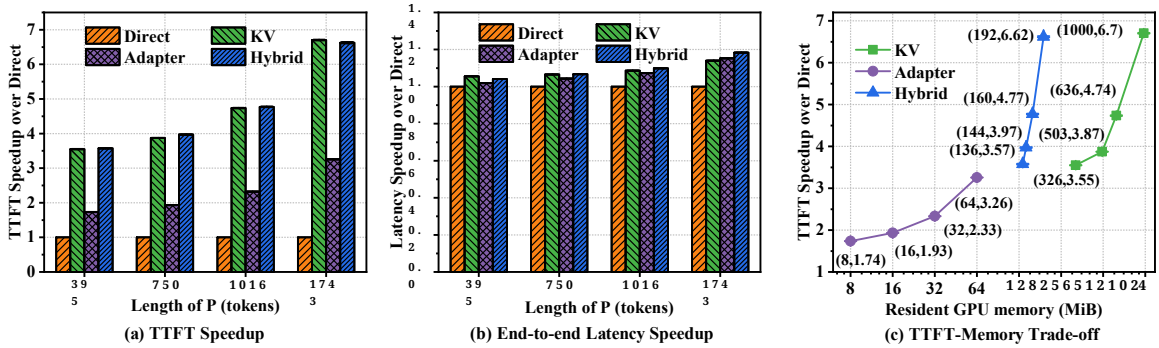

图 4：不同提示表示下的热路径性能和 GPU内存使用。图 (a) 和 (b) 分别展示了 TTFT加速比和端到端延迟加速比随提示长度的变化。图 (c) 展示了 TTFT加速比与驻留GPU内存的关系。所有加速比均相对于Direct。

处理时间。重复命中可以迅速摊销注册成本，但在两次命中之间，这些资源仍会持续占用内存。在内存压力下，并发请求之间的驱逐和争用可能降低吞吐量或增加尾延迟。因此，仅凭热路径延迟不足以选择 Hybrid。选择取决于资源保持驻留期间，后续命中能够节省多少处理时间。

3.3 用于调度实验的代理适配器验证

为了在调度实验中研究更多适配器和秩下的这些权衡，我们使用与第 3.1 节中训练的语义适配器具有相同张量结构、但 LoRA 权重不同的代理适配器。此前关于多 LoRA 服务的研究也利用张量形状 [4] 构建性能工作负载。为验证这种替代方法，我们使用完全相同的输入和运行时设置，对一个合格的秩16适配器及其代理进行比较。对于 Adapter 和 Hybrid，两种适配器之间的平均吞吐量和平均 P95 SCT 的相对差异均低于 1.3%。从 Adapter 切换到 Hybrid 后，使用语义适配器时吞吐量提升 8.72%，使用代理时提升 10.26%；P95 SCT 分别降低 8.03% 和 9.32%。两组权重带来了相近的执行性能，以及从 Hybrid 中获得的相近收益。这种一致性支持我们在后续调度实验中使用代理适配器研究 RePrism 的调度行为。

4 联合请求调度与表示选择

4.1 RePrism架构

RePrism 在共享 GPU 内存预算下联合调度请求并选择其表示。图5展示了它的两个组件：联合调度器 和资源管理器。我们假设，当会话的第一个请求到达时，调度器可以访问可见工作流状态和请求属性。调度器首先识别依赖关系已满足的请求1。随后，它将每个请求与满足行为要求的表示配对，形成用于联合评分的候选项2，并根据它们对会话进度的贡献及其表示成本3对其进行优先级排序，

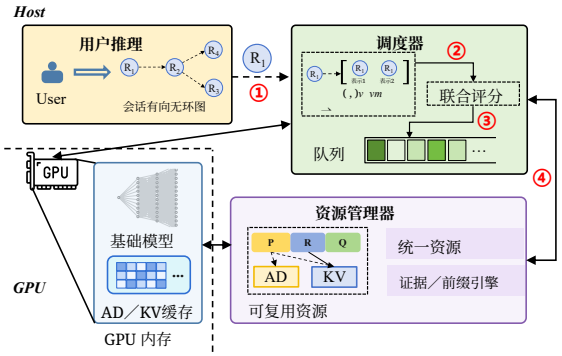

图5：RePrism架构。调度器联合选择请求及其表示，资源管理器协调缓存准入和驱逐。

资源管理器报告哪些资源已驻留，并估算 GPU 内存需求。一旦选定一个批次准入，它便协调资源加载和必要的驱逐4。本文使用单个 GPU，但 RePrism 可以扩展为通过跨 GPU 的张量并行或流水线并行来服务更大的模型。

4.2 将就绪请求与可用表示配对

传统的多LoRA调度器 [15, 37, 38] 将单个请求 v 作为调度单位，通常不跟踪工作流结构。在智能体工作流中，向调度器公开的请求节点只有在其所需的前置节点已完成且满足其分支条件时才能运行 [29,56,57]。因此，提交的请求只有在满足这些条件时，才会在调度时间t 进入就绪集合 $\mathcal{R}_t$。对于每个就绪请求 v，调度器会检查$\mathcal{M}(v) \subseteq$ {中的哪些表示（Direct、键值、适配器、Hybrid}）可用。只有当固定提示词能够由通过验证的适配器表示时，适配器和Hybrid才可用。令$E_t(v、m)$表示两个指标的乘积，其中一个指标表示请求依赖关系已满足，另一个表示表示可用性。在线调度的候选项集合为：

$$\mathcal{C}_t = \{(v,m) \mid v \in \mathcal{R}_t,\ m \in \mathcal{M}(v),\ E_t(v,m) = 1\}.$$

RePrism 将表示可用性与缓存驻留状态分开检查。例如，即使 Hybrid 的残余KV缓存不存在，只要能够使用相应的适配器构建，它仍然具备可用性。然而，即使残余上下文相同，模型也无法直接复用使用不同适配器构建的残余KV缓存。因此，离线资格判定决定了请求可以使用哪些表示；缓存驻留状态决定了使用这些表示的成本，而联合评分则决定了调度器选择哪些候选项执行。

4.3 联合评分与批次选择

RePrism 调度器会对执行候选项进行排序，其设计借鉴了多LoRA调度器 [15, 37, 38, 48]和面向智能体工作流的服务系统 [31, 32, 46]。分别对执行请求节点 v 的价值及其表示的价值进行评分，可能会得到与联合评分截然不同的执行顺序。例如，位于更长剩余路径上的节点可能需要额外时间来为表示 m构建或加载资源，而对会话进度贡献较小的节点则可以立即复用一个驻留缓存条目。在决定如何表示每个可复用上下文之前就固定请求顺序的调度器，几乎没有机会在不同请求之间权衡这两种影响。

图 6 展示了 RePrism 如何比采用 LRU驱逐的先来先服务调度更早完成一个简单的工作流。请求 R_1, R_2, 和 R_3 的时间线省略了 Q 的预填充。联合评分将 R_3 排在 R_2 之前，因为先运行 R_2 会驱逐 GPU 内存中已有的资源1。RePrism 利用可见工作流选择表示，使 R_3 能够复用 R_1 使用的适配器和键值缓存2。由于 R_2 很快就会运行，RePrism 优先保留它所需的驻留资源，从而避免之后重新加载3。本节将解释 RePrism 如何实现 1 和 2。第 4.4 节介绍 3。

联合评分。 对于每个候选项（v，m） $\in \mathcal{C}_t$，RePrism 分别对节点执行价值$V\text{node}_t$（s,v）和表示价值 $V\text{repr}_t$（v,m）进行归一化，然后将二者相乘。我们将归一化范围的下界设为 0.5，以防止任一项的值较小时压倒另一项：

$$S_t(s,v,m) = N_{[0.5,1]}(V_t^{\text{node}}(s,v)) \cdot N_{[0.5,1]}(V_t^{\text{repr}}(v,m)).$$

节点优先级结合了推进会话的收益与其等待调度的时间：

$$V_t^{\text{node}}(s,v) = \alpha_p P_t(s,v) + \alpha_w W_t(s),$$

其中，P_t(s，v)表示推进当前节点v 对帮助会话s 更快完成的价值。RePrism 使用剩余令牌数估算会话s所需的完成时间。该估计值与节点v的路径关键性共同决定P_t(s，v)。路径关键性是指节点对解锁下游分支、解除分叉阻塞或直接完成会话所作的贡献。W_t(s)表示等待紧迫度，并随着会话s等待调度的总时间增加而提高。W_t(s)不包括候选节点的执行时间以及等待并行分支所花费的时间。这样可以防止某个会话仅仅因为执行时间更长，就不断获得更高的优先级。

为了比较以不同表示执行请求的收益，RePrism 从即时执行收益、未来复用收益以及资源的机会成本这几个方面来表示每种表示的价值

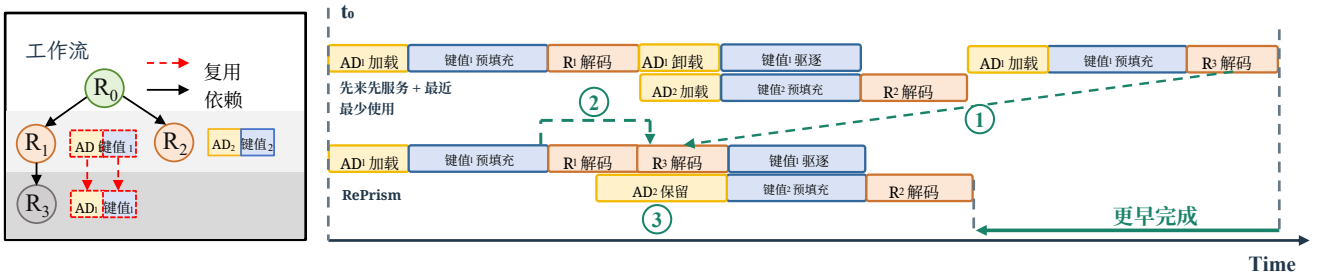

图 6：示意性执行 左侧工作流中先来先服务配合 LRU驱逐以及 RePrism 的时间线。调度 R_3 在 R_2 之前，R_3 复用该 来自适配器和 R_1 的键值缓存。

它使用：

$$V_t^{\text{repr}}(v,m) = \beta_v V_t^{\text{now}}(v,m) + \beta_g G_t^{\text{reuse}}(v,m) - \beta_o O_t^{\text{resource}}(v,m).$$

其中，$V\text{now}_t(v,m)$ 表示使用 m 所带来的即时时间节省，$G_t^{\text{reuse}}(v,m)$ 表示未来复用所带来的预期节省，而 $O_t^{\text{resource}}(v,m)$ 表示因驱逐而导致的预期重建成本变化。三个项均相对于请求 v 在 Direct 下的预计剩余执行时间来表示。我们将所有评分权重均设为 1。补充材料的 A.1 和 A.2 节给出了完整的评分规则。

预测成本与收益。RePrism 根据离线测量构建预测模型，并在运行时使用每个请求的参数应用该模型。它会比较在表示 m 和 Direct 下的预计执行时间，同时考虑为 m加载或构建资源所需的额外时间。为了预测复用，RePrism 会遍历可见工作流，以识别之后可能复用 m的请求。它根据复用距离和分支概率对每个请求的贡献进行加权，使可能很快发生的复用获得更高权重。RePrism 还会估计候选项将驱逐哪些驻留缓存资源，以及重新构建这些资源所需的额外时间，从而避免选择需要高成本驱逐的方案。

调度过程。算法 1 概述了给定可见工作流状态 G_t、运行中的批次B_t和 GPU内存状态 D_t时的调度过程。

调度器首先过滤掉超出 GPU 内存预算或使用不符合条件的表示的候选项。然后，它根据联合评分对剩余候选项进行排序；由于某些评分项取决于运行时条件，调度器会在每轮开始时重新计算联合评分。例如，$V\text{now}_t(v,m)$ 取决于m 所需的缓存是否驻留。为限制等待时间，等待超过 60 秒的请求优先于其他按评分排序的请求，但仍受依赖和资源约束的限制。在一轮调度过程中，排序保持不变。当调度器按照排序将某个请求的第一个符合条件的候选项加入临时批次后，便会跳过该请求的所有其余候选项。

每次调度器将一个候选项加入暂定批次时，都会更新剩余的 GPU 内存预算

算法 1： 联合请求调度与表示选择

输入： G_t、B_t、D_t
输出： Π_t

更新状态(G_t、D_t)
$\mathcal{R}_t \leftarrow$就绪请求(G_t)
$\mathcal{C}_t \leftarrow \{(v,m) \mid v \in \mathcal{R}_t,\ m \in \mathcal{M}(v),\ E_t(v,m) = 1\}$
$\mathcal{C}_t \leftarrow$ 筛选可行($\mathcal{C}_t$,B_t,D_t)
对每个$c \in \mathcal{C}_t$ **执行**
    得分(c) $\leftarrow$ 联合得分(c)
$\mathcal{C}_t \leftarrow$对候选项排序($\mathcal{C}_t$)
$\Pi_t \leftarrow$空计划()
对于每个 $c \in \mathcal{C}_t$ **执行**
    如果 请求(c) $\in \Pi_t$ **则继续**
    好的 $\leftarrow$项目准入(c Π_t,B_t,D_t)
    如果 好的 **则** 添加(Π_t,c)
    如果 批次已满(Π_t)**则中断**
如果 $\Pi_t \neq \emptyset$**则**
    好，$V \leftarrow$ 提交准入(Π_t，B_t，D_t) // 算法 2
    如果不好则返回 $\emptyset$
    Π_t.受害者 $\leftarrow V$
否则如果 IdleSafePoint()**则**
    Π_t.预取 $\leftarrow$ 选择预取(G_t，D_t)
返回 Π_t

。批次组装完成后，调度器会调用 CommitAdmission()，以决定是否准入该批次，以及在必要时驱逐哪些资源。在批次形成过程中，一个分数较高、但过长而无法装入剩余内存的请求，可能会阻塞排在其后的较短请求，导致 GPU 内存闲置。为避免这种情况，调度器允许后续请求越过那些分数较高但无法装入的候选项。

4.4 跨表示的 GPU 内存管理

为了支持算法 1 中联合评分生成的批次计划，资源管理器会在不同表示之间分配 GPU 内存，并处理资源准入和驱逐。

统一资源核算。 不同的表示需要占用不同的 GPU 内存，而联合评分会权衡复用带来的收益与内存成本。

为支持复用，资源管理器会在 GPU 内存中维护一个大小可变的区域。此外，它还会为动态键值缓存预留空间，该缓存会随着合并批次的运行而增长。管理器利用合并批次中所有请求的输入长度和预期输出长度，估算预填充和解码期间需要动态键值存储的令牌峰值数量T^{peak}。缓存资源仅计入缓存区域；多个请求共享的条目只计一次。每次合并批次都必须满足：

$$M_{\text{keep}} + M_{\text{new}} + b_{\text{KV}} T_{\text{peak}} \le B_{\text{runtime}}.$$

其中，M^{keep} 和M_{new} 分别是运行中批次和拟议批次所需的缓存字节数。
b_{KV} 是每个令牌的动态键值缓存字节数，B_{runtime}是运行时 GPU 内存预算。资源管理器将剩余空间分配给缓存区域，以保留资源供后续请求复用。补充材料的 A.3 节介绍了 RePrism 如何计算动态键值需求峰值。

批次资源准入。RePrism 在提交批次之前会检查各个候选项。在检查候选项期间，调度器拟定的批次仍处于暂定状态。算法 1 中的 ProjectAdmit() 会检查新批次能否在不驱逐任何驻留缓存资源的情况下并入，同时不超出 GPU 内存预算。算法 2 中的 CommitAdmission() 会作出最终决定。如果批次的内存需求超出预算，或者即使驱逐所有符合条件的资源，剩余空间仍不足以容纳该批次，则拒绝该批次。否则，它会选出足够的资源进行驱逐，并一并提交资源预留和驱逐操作。

算法 2： 批次资源准入与驱逐

输入： Π_t，B_t，D_t
输出： (成功，受害者)
$\mathcal{U} \leftarrow$唯一资源(Π_t)
$T_{\text{peak}} \leftarrow$ 峰值投影(B_t，Π_t)
如果 硬性不可行($\mathcal{U}$，T^{peak}，D_t) **则**
    返回 false ,$\emptyset$
$b \leftarrow$需要字节数($\mathcal{U}$、T^{peak}、D_t)
$V \leftarrow \emptyset$
如果 $b > 0$ **则**
    $H \leftarrow$ 受保护(B_t、Π_t、D_t)
    $V \leftarrow$选择驱逐对象(可驱逐(D_t) $\setminus H$、b)
    如果释放(V) $< b$ **则**
        返回false ,$\emptyset$
带驱逐的原子预留(Π_t，$\mathcal{U}$，T^{peak}，V，D_t)
返回true ,V

缓存驱逐。仅当直接将新的候选项加入运行中的批次会超出 GPU 内存预算时，管理器才会驱逐驻留缓存。随后，它按照分数顺序选择要驱逐的资源。管理器将运行中的请求持有、正在加载或当前候选项需要的资源标记为受保护。如果资源符合其中任一情况，管理器就会将其标记为受保护。算法 2 中的 SelectVictims() 会排除受保护的资源，并选择当前轮次要驱逐的资源。工作流中的驱逐不仅需要跟踪近期访问。成本感知策略必须同时考虑缓存资源不均衡的受欢迎程度，以及恢复这些资源的成本 [31]。RePrism 根据资源的每字节保留价值对其进行驱逐排序，D_i，其定义如下：

$$D_i = \frac{C_i}{B_i} \cdot (1 + r_i) \cdot (1 + \frac{\min(4, h_i)}{4})$$

其中C_i 表示保留资源i所节省的重建时间，该时间通过性能分析测得；B_i 表示其在 GPU 内存中占用的字节数。二者的比值构成了基于测量成本得出的基础优先级。
RePrism 会将此优先级乘以两个因子，这两个因子分别基于可见工作流和近期重建历史。第一个因子使用r_i，其取值范围为 $[0, 1]$，用于根据算法 1 按联合评分对候选项排序的顺序，估算未来对资源i 的需求。第二个因子使用h_i，表示资源i 在被驱逐后 10 秒内重建的次数。RePrism 会将每条重建记录保留 60 秒，以减少缓存抖动。

管理器会根据缓存资源每字节 GPU 内存的估计保留价值对其排序，并优先驱逐价值较低的资源。仅遵循这一顺序可能会释放远超所需的内存。在管理器选出足够多的资源后，它会按保留价值从高到低重新检查这些资源。只要驱逐其余资源仍能释放足够的内存，它就会将某个资源从驱逐集合中移除。管理器会将被驱逐的适配器卸载到 CPU 内存，并在下次使用时从那里重新加载；而被驱逐的 KV 缓存则会被丢弃，并在下次使用时重建。在我们的测试条件下，重建 KV 缓存所需的时间比将其从 CPU 传输到 GPU 内存更短。当没有批次运行且等待队列为空时，RePrism 也可以预取未来请求可能使用的表示。但在我们的实验中，并未触发这一路径。

5 评估

5.1 实验设置

平台。 我们使用一块配备 24 GB 显存的 NVIDIA RTX 3090 GPU、一颗配置为 52 个逻辑核心的 Intel 8171M CPU，以及 128 GB 的主机内存。我们在 S-LoRA [37] 上实现 RePrism，并在主要实验中使用 Llama-2-7B-Chat 作为基础模型。

资源预算。 对于所有系统，我们将整个 GPU 的峰值内存预算设为 22000 MiB，并允许 3% 的运行时波动。为考虑不同框架在静态 GPU 内存使用方面的差异，我们将 Helium 的动态运行时预算设为 14000 个键值 token 等价量，将其他系统的预算设为 16000

表1：我们评估中的服务机制。DAG访问表示调度器可以使用工作流依赖关系。

系统	提示词表示	DAG访问	服务机制
S-LoRA	Direct，适配器	No	批处理请求跨适配器
Chameleon	Direct，适配器	No	缓存适配器并会考虑适配器大小的调度
SGLang	Direct、键值	No	复用键值状态以用于共享前缀
Helium	Direct、键值	Yes	使用工作流来规划键值复用
RePrism	Direct、键值、适配器，Hybrid	Yes	联合选择请求和表示

。在 Llama- 2 上，每个令牌需要 0.5 MiB。这些预算涵盖可复用状态和动态键值缓存，而 GPU 内存总使用量还包括模型权重和工作区所占用的内存。我们在每组实验中保持模型配置不变。第 5.5.1 节给出了跨模型实验所采用的配置。

基线系统。 我们将 RePrism 与 S-LoRA、Chameleon [15],SGLang [58], 和 Helium [46]进行比较。表 1 列出了各系统可用的表示和工作流信息。S-LoRA 和 Chameleon 可以使用匹配的适配器，但二者都无法利用工作流信息。Chameleon 还可以在闲置的 GPU 内存中缓存适配器。SGLang 可以保留先前计算出的键值状态以供复用，但不使用工作流信息。Helium 则利用工作流信息优化键值重用。所有系统处理相同的工作流，区别仅在于可用的表示和调度逻辑。第 5.4 节进一步在相同的执行后端和资源管理策略下比较调度策略。

工作负载。 要全面评估 RePrism 的联合调度，需要一个能够捕捉工作流依赖关系、请求间前缀共享，以及提示词与适配器之间映射关系的数据集。公开的生产跟踪记录通常只报告请求到达时间以及输入和输出令牌数。例如，Preble [42] 指出，Azure 跟踪记录 [34] 未提供提示词文本，因此无法重建哪些请求共享前缀。

我们设计了具有动态到达的工作负载，以评估端到端性能；同时设计了受控工作负载，以衡量各个组件带来的收益。第 5.3 至 5.5 节中的受控工作负载使用图 7 所示的 CI 修复工作流。第 5.2 节中的动态工作负载混合了 CI 修复、事件响应和多文档分析会话。后两种工作流模板分别包含八个和九个请求节点。所有系统对给定实验使用相同的

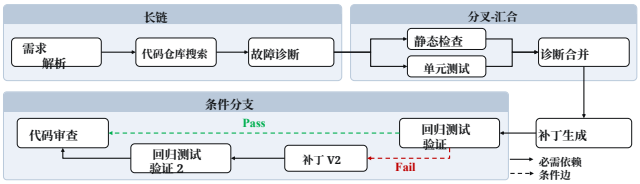

图 7：编码代理的 CI 修复工作流。静态检查和单元测试在合并其结果之前并行运行。失败的回归检查会触发第二次补丁和验证步骤。

工作负载配置。客户端设置每个会话的根请求到达时间，并在依赖关系满足后释放后继请求。客户端最多同时处理 128 个请求，并在本地将额外的就绪请求加入队列。

在动态工作负载中，会话根据泊松过程 [15, 60] 到达，到达率范围为 0.7 至 1.9 会话/秒。客户端提交新会话持续 200 秒，随后停止提交会话，并让系统继续运行，直到所有已提交的请求均完成。提示词P 范围为 409 至 1949 个令牌，长度集中在四个区间内，分别对应第 3.1 节中与四种适配器秩配对的提示长度。在每个秩内，我们根据指数为 1.1 [4] 的 Zipf 分布为请求分配适配器。残余上下文R 范围为 149 至 207 个令牌。查询长度Q 和预期输出长度 O 因任务模板而异，大多数值介于 10 至 100 个令牌之间。

受控实验包括资源压力测试和策略比较。第 5.3 节中的资源压力测试会一次性释放所有会话，以产生高并发。对于第 5.4 至 5.5 节中的策略比较，客户端对所有策略使用相同的会话到达序列。会话间到达时间遵循形状参数为 0.5 的伽马分布，目标到达率为 0.2 会话/秒。提交所有会话后，每次运行会持续进行，直到这些会话完成。每个受控工作负载包含 96 个会话和 1,056 个请求。默认情况下，P、R、Q和O 的长度分别为 1152、256、256 和 128 个令牌，并且该工作负载提供一个包含 32 个秩为 16 的适配器的池，用于替代P。

指标。 我们报告请求吞吐量、会话完成时间（SCT）和首个令牌时间（TTFT）。吞吐量是已完成请求数除以从客户端开始发送请求到服务器完成所有请求处理的总时间。SCT 是从会话计划到达到目标节点完成处理之间的时间。对于受控工作负载，我们主要报告平均 SCT，以衡量总体完成效率。对于动态工作负载，我们主要报告 P95 SCT，以衡量会话尾延迟。所有图表均报告五次运行的平均结果。TTFT 是从客户端提交请求到收到包含令牌的首个响应之间的时间。

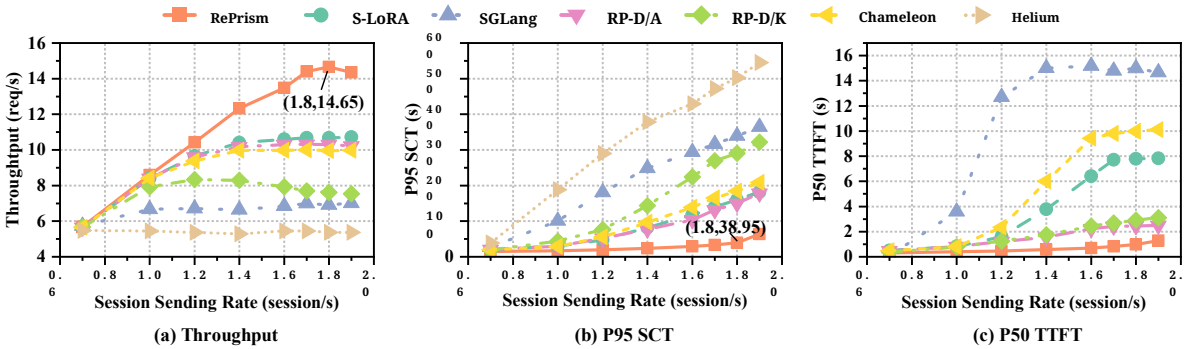

图8：随着会话到达率增加，端到端性能的变化。各面板分别显示：（a）请求吞吐量，（b）P95 SCT，以及（c）P50 TTFT。Helium 不提供逐请求 TTFT。

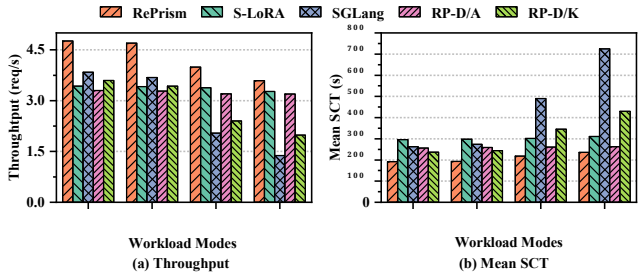

图9：在高并发下，复用距离和不同的（P、R）配对数量对（a）请求吞吐量和（b）平均 SCT 的影响。N 统计不同的配对数量。Short 和 Long 表示会话根请求序列中的短复用距离和长复用距离。

5.2 跨会话到达率的性能

图8比较了 RePrism 与基线系统在动态工作负载下的端到端性能。由于 Helium 不会为每个请求的首个令牌提供时间戳，因此我们省略了 Helium 的 P50 TTFT。RP-D/A 和 RP-D/K 保留 RePrism 的评分机制，但分别将可用表示限制为 Direct/Adapter 和 Direct/KV。在可用表示保持一致的情况下，可以隔离执行顺序和状态保留方面相对于对应基线系统的差异。

随着会话到达率的增加，SGLang 和 He- lium 比其他系统更早接近容量上限。它们的吞吐量增长更慢，而 P95 SCT 则急剧上升。S-LoRA 和 Chameleon 能够维持更高的到达率，但其吞吐量最终会在约 10 到 11 请求/秒处趋于平稳，而尾延迟仍会持续上升。RePrism 在高负载下能够维持约 14 请求/秒。在 1.8 会话/秒时，其吞吐量达到 S-LoRA（次优系统）的 1.37× 倍，并将 P95 SCT 降低了 4.10 倍。复用选项的范围以及调度和缓存管理策略共同促成了 RePrism 的优势。在高工作负载下，RP-D/A 和 RP-D/K 的 P95 SCT 也分别低于 S-LoRA 和 SGLang。这些比较表明，即使可用的表示受到限制，这些策略仍能带来一定收益。这一优势在 P50 TTFT 方面更加明显。在 1.7 会话/秒时，与 S-LoRA 和 SGLang 相比，RePrism 将 P50 TTFT 分别降低了 9.40 倍和 18.01 倍。与 S-LoRA 相比，RP-D/A 也将 P50 TTFT 降低了 3.2 倍。

RePrism 的主要优势体现在中高负载下的吞吐量和 SCT。在更高负载下，即使表示选择受限，RP-D/A 和 RP-D/K 的 P95 SCT 也低于各自的基线系统 S-LoRA 和 SGLang，P50 TTFT 更是显著低于它们。

5.3 高并发下复用模式的影响

我们使用受控工作负载评估高并发下的系统性能。图9比较了四种场景，这些场景在不同的可复用 P、R 配对数量及其会话复用距离方面有所不同。我们用 N 表示不同配对的数量，并将其设为 6 或 32。复用距离是指同一配对连续两次使用之间间隔的会话数。在 Short 中，同一配对会出现在连续会话中，或仅相隔少数几个会话。在 Long 中，会话会轮流遍历全部N 个配对。这些设置仅适用于会话根请求序列。在执行过程中，调度器决定请求的运行顺序，因此也决定实际的复用模式。

如图9所示，在四种场景中，RePrism 的吞吐量均高于 S-LoRA 和 SGLang，平均 SCT 均低于二者。在$N = 6$时，两个指标相对于两种基线系统的加速比约为 1.24× 至 1.54×。在$N = 32$时，SGLang 和 RP-D/K 面临更大的资源压力，并且在复用距离更长时性能进一步下降。即便如此，RP-D/K 仍将平均 SCT 降低至 SGLang 的 1/1.69。RePrism 和 RP-D/A 相对于 S-LoRA 的收益在$N = 32$时有所收窄。

与基线系统相比，RePrism 在高并发工作负载上的优势不如在动态工作负载上明显。这一差异与工作负载中使用的请求长度配置有关。一般而言，每个请求的可复用比例越大，RePrism 就越有机会

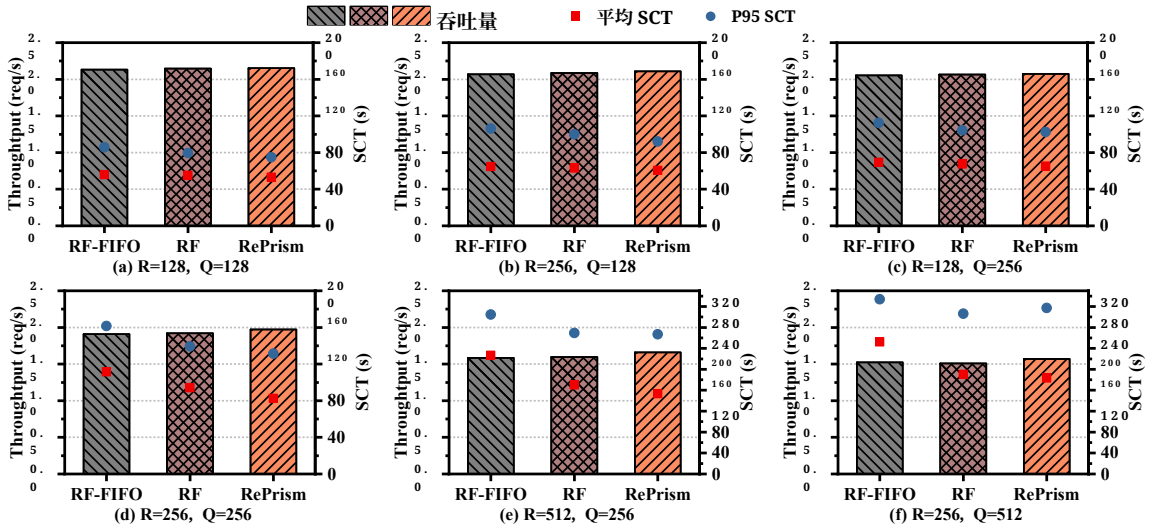

图 10：不同剩余上下文长度 R 和查询长度 Q下的联合调度与请求优先调度比较。以令牌计的（R,Q）对分别为：(a) (128, 128)、(b) (256, 128)、(c) (128, 256)、(d) (256, 256)、(e) (512,256) 和 (f) (256,512)。柱状图显示左轴上的吞吐量。方块和圆点分别显示右轴上的平均 SCT 和 P95 SCT。

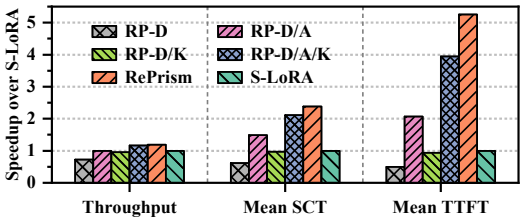

图 11：可用提示表示对吞吐量、平均 SCT 和平均 TTFT 的影响。所有数值均为相对于 S-LoRA 的加速比。数值大于 1 表示更高的吞吐量或更低的延迟。

通过调度提升性能。

5.4 消融实验与策略比较

5.4.1 联合评分的效果

为评估联合评分的必要性，我们将 RePrism 与两种策略进行比较：先对请求排序，再对每个请求的符合条件的表示进行评分。这两种策略的区别在于请求的排序方式。RF 使用与 RePrism 相同的请求评分，而 RF-FIFO 则按请求到达的顺序处理请求。两者对表示的排序方式都与 RePrism 相同。工作负载包含 32 个不同提示P，分两个连续阶段运行。每个阶段中，一半提示词为 Hot，会重复出现；另一半为 Cold，各出现一次。第二阶段中，两组提示词互换角色。我们将 Adapter/Hybrid 资格覆盖率q 设为 50%。一半提示词符合 Adapter 和 Hybrid 的条件，其余提示词只能使用 Direct 和键值。

图 10 展示了三种策略在不同R 和Q长度下的吞吐量和 SCT。在测试的长度范围内，RePrism 相较于 RF-FIFO 和 RF 的吞吐量提升最多为 5%。随着R 和Q 变长，平均 SCT 和 P95 SCT 的差异变得更加明显。例如，在图 10(d) 中，当$R = Q = 256$ 令牌时，RePrism 的平均 SCT 相较于 RF-FIFO 和 RF 分别降低至原来的 1/1.35 和 1/1.14。相应的 P95 SCT 分别降低至原来的 1/1.23 和 1/1.06。

当R 和Q长度适中时，按评分对请求排序相比 FIFO 能带来显著提升，而 RePrism 的联合评分还能进一步扩大这一提升。当R 和Q 较短时，调度带来的收益有限；而R 和Q 较长时，硬件限制会导致请求队列变长。因此，在后续实验中，我们使用$R = Q = 256$个令牌。

5.4.2 可用的提示表示

为评估提示表示的选择对 RePrism 性能的影响，我们构建了四种基线系统，它们各自采用不同的符合条件的表示集合。这些基线系统 RP-D、RP-D/A、RP-D/K 和 RP-D/A/K，采用与 RePrism 相同的评分和资源管理策略。

如图 11 所示，RP-D 对每个请求都使用 Direct，表现最差，吞吐量比 S-LoRA 低 27%。添加 Adapter（RP-D/A）或键值（RP-D/K）都能使吞吐量接近 S-LoRA。随着可用表示增多，RP-D/A/K 和 RePrism 的吞吐量均超过 S-LoRA，且平均 SCT 降低了两倍以上。与 S-LoRA 相比，RePrism 还将平均 TTFT 降低了 5.26 倍。这些结果表明，支持多种表示能够提升性能。具体而言，避免对可复用的提示内容重复进行预填充，可大幅降低 SCT 和 TTFT。

表2：两种复用条件下，采用自适应和固定提示词表示时的平均 SCT（秒）。

复用	S-D	S-K	S-A	S-H	RePrism
Cold	321.38	241.52	**54.96**	60.10	57.84
Hot	330.66	184.02	54.95	**44.17**	45.80

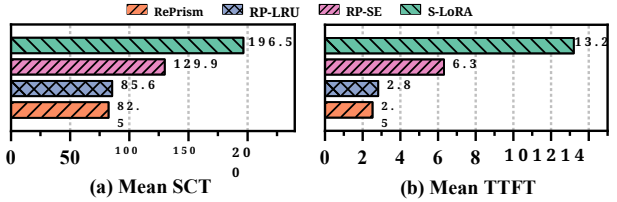

图12：缓存资源驱逐策略对（a）平均 SCT 和（b）平均 TTFT 的影响。

5.4.3 自适应与固定表示选择

本节将 RePrism 在调度时选择表示的策略，与四种预先固定表示的策略进行比较。S-D、S-A、S-K 和 S-H 分别始终使用 Direct、Adapter、键值和 Hybrid，其他方面则遵循 RePrism 的调度策略。为突出这一选择的影响，我们构造了两种工作负载 Cold 和 Hot，其中每个提示词 P 都可以用适配器表示。Cold 没有残差键值复用的机会，而 Hot 会重用 32 个不同的残差键值缓存。

表 2 显示，在 Cold 和 Hot 之间，S-A 与 S-H 的相对性能发生了反转，而 S-D 和 S-K 在两种情况下的表现都明显更差。在 Cold 中，S-A 的平均 SCT 低于 S-H（54.96 s 对 60.10 s）。在 Hot 中，S-H 的平均 SCT 低于 S-A（44.17 s 对 54.95 s）。因此，最佳表示取决于复用条件，单一的固定选择无法同时适应两种情况。RePrism 在运行时调整其选择，在 Cold 中实现了 57.84 s 的平均 SCT，在 Hot 中实现了 45.80 s。与针对每种工作负载预先选定的最佳静态策略相比，这些数值分别仅高出 5.2% 和 3.7%。相对于同一基线，始终使用适配器会使 Hot 中的平均 SCT 最高增加 24.4%，而始终使用 Hybrid 会使 Cold 中的平均 SCT 最高增加 9.3%。

5.4.4 缓存驱逐策略

我们使用两个变体 RP-LRU 和 RP-SE，评估驱逐策略如何影响 RePrism 的性能。两者与 RePrism 的唯一区别在于如何驱逐缓存资源。

图12比较了 RePrism、RP-LRU、RP-SE 和 S-LoRA 的平均 SCT 与平均 TTFT。RP-SE 会在缓存资源不再使用后将其丢弃。其平均 TTFT 比 S-LoRA 低 2.10 倍，但仍远高于 RP-LRU 或 RePrism。当必须进行驱逐时，RP-LRU 会优先驱逐最近最少使用的缓存资源。因此，在 GPU 内存压力较低的情况下，它可以保留

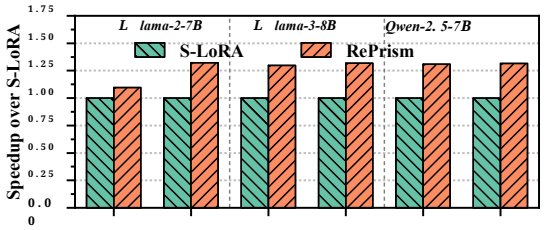

图13：在 N32-Long 工作负载上，RePrism 相对于 S-LoRA 的吞吐量和平均 SCT 加速比。每次比较均使用与 1×基线相同的基础模型上的 S-LoRA。

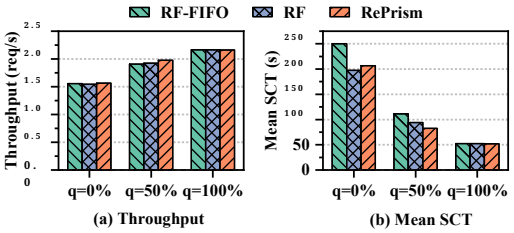

图14：适配器资格覆盖率对 (a) 请求吞吐量和 (b) 平均 SCT 的影响。覆盖率 q 是符合适配器和 Hybrid 条件的不同提示所占的比例。

后续请求所需的资源。与 S-LoRA 相比，RP-LRU 将平均 TTFT 降低了 4.68 倍。RePrism 利用工作流信息对待驱逐的缓存资源进行排序，因此驱逐顺序并不只由最近使用情况决定。与 RP-LRU 相比，这使平均 TTFT 进一步降低了 1.13 倍。

5.5 敏感性分析

5.5.1 基础模型的选择

为评估 RePrism 相较于 S-LoRA 的优势如何随基础模型而变化，我们除 Llama-2-7B-Chat [45] 外，还测试了 Llama-3-8B-Instruct [11]和 Qwen- 2.5-7B-Instruct [50]。由于不同模型的静态 GPU 内存使用量各异，我们将动态内存预算校准至相同的峰值 GPU 内存使用量上限。最终，Llama-2、Llama-3 和 Qwen-2.5 的预算分别为 8000 MiB、5875 MiB 和 5687.5 MiB。对于每个模型，RePrism 和 S-LoRA 使用相同的N32-Long工作负载和资源预算。

图13比较了吞吐量和平均 SCT，并将每个模型上的 S-LoRA 归一化为 1.0。在 Llama-2、Llama-3 和 Qwen-2.5 上，RePrism 的吞吐量分别达到 S-LoRA 的 1.10×、1.30× 和 1.31× ，并将三种模型上的平均 SCT 均降低了 1.32 倍。Llama-3 和 Qwen-2.5 上更大的吞吐量提升部分源于模型架构差异。与 Llama-2 相比，这些模型为调度器提供了更多可复用资源的容量，使 RePrism 能够比 S-LoRA 缓存更多此类资源。在这三种模型上，RePrism 的性能均优于 S-LoRA，表明其收益并不局限于单一模型架构。

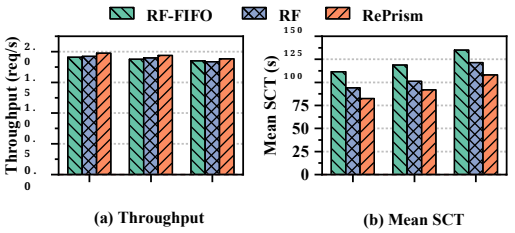

图15：工作流可见性对（a）请求吞吐量和（b）平均 SCT 的影响。完整模式会公开整个已提交的工作流，下一跳模式会公开就绪节点及其直接后继节点，而前沿模式仅公开就绪节点。

5.5.2 适配器资格覆盖率

为评估 RePrism 的性能对符合适配器内部化条件的不同提示词所占比例q 的敏感程度，本节将q 设置为 0%、50% 和 100%。

如图14所示，将 q 从 0% 提高到 100% 后，RePrism 的吞吐量从 1.56 请求/秒提升至 2.16 请求/秒，平均 SCT 则从 206.85 秒降至 51.87 秒。当 q = 50% 时，联合调度相较于请求优先调度的收益最大：RePrism 的吞吐量比 RF 高 2.7%，平均 SCT 比 RF 低 12.3%。当 q = 100% 时，RePrism 相较于 RF 的平均 SCT 降幅最多为 1%。当 q = 0% 时，其平均 SCT 略高于 RF。因此，在该工作负载下，更广泛的资格范围能够提升绝对性能；而当符合资格与不符合资格的请求并存时，联合请求与表示选择则能带来更大的相对收益。

5.5.3 工作流可见性

为评估 RePrism 的性能对调度器可见的工作流节点的依赖程度，本节测试了三个可见性级别。完整 会公开客户端已提交的每个会话的完整工作流。下一跳 会公开当前就绪节点及其直接后继节点。前沿 仅公开当前就绪节点。这三种配置都使用完整工作流来强制执行依赖关系并释放请求以供执行。变化的只有调度器可获取的信息。

如图15所示，将可见性从完整级别降低到前沿级别，会使 RePrism 的吞吐量从 1.97 请求/秒降至 1.89 请求/秒，并使其平均 SCT 从 82.48 秒增至 108.09 秒。在每种可见性级别下，RePrism 的表现都优于 RF 和 RF-FIFO。与 RF 相比，其吞吐量高出 2.1% 至 2.7%，平均 SCT 低 9.1 % 至 12.3%。因此，工作流可见性降低会损害 RePrism 的性能，但它仍优于这两种请求优先策略。

6 相关工作

智能体工作流服务。现有系统 [9,26,29,36,44]利用调用间的依赖关系来优化智能体工作流执行。KVFlow [31] 利用其智能体步骤图中的依赖关系来预测即将发生的调用，并指导前缀KV状态的驱逐和预取。Helium [46] 将工作流建模为查询计划，以优化调用间缓存数据和计算的复用。SAGA [12] 研究工作流级调度。Continuum [24] 使用生存时间策略控制工具调用期间的键值保留，ScaleSim [32] 则使用调用距离管理智能体状态。RePrism 进一步考虑每个请求符合条件的表示，并利用工作流信息联合选择要执行的就绪请求，以及每个请求所采用的表示。

提示内化。 以往研究 [5,7,40,41] 探索如何将提示词或上下文编码到模型参数中，使其在推理时无需显式提供。FIP [7]对固定输入进行参数化，而上下文蒸馏 [41] 和 GenPI [40] 探索如何内化由提示词诱导的行为。生成式适配器 [5],、Doc-to-LoRA [3], 和 Text-to-LoRA [2] 根据上下文或任务描述生成轻量级适配器。提示压缩方法 [17,18,30,33]通过缩短输入或压缩其表示来减少提示词开销。RePrism 将先前构建且通过行为资格认证的适配器视为候选表示，并研究这些适配器与显式输入及 KV 表示共存时的在线表示选择和资源管理。

适配器和 KV 服务。 多 LoRA 服务系统 [4, 15,25,37,48,61] 通过异构批处理、专用算子以及适配器加载和缓存来提升执行效率。ELORA [39] 利用 LoRA 使用与 KV 使用之间的依赖关系，在统一缓存池中管理这两类状态。关于 KV 服务的研究 [6,10,19,21,28,42,53,55,58]主要关注 GPU 内存分配、前缀复用以及跨内存层级的缓存。vLLM [21] 使用 PagedAttention 管理 KV 缓存，而 SGLang [58] 使用基数注意力在请求之间复用前缀。近期研究 [16, 22, 27, 47]还探索了适配器或微调模型之间的 KV 共享。RePrism 通过考虑表示选择如何改变每个请求的执行成本及其所需的共享状态，协调请求调度和 GPU 内存分配。

7 结论

我们提出了 RePrism，这是一种能够为智能体工作流联合调度请求并选择其表示的系统。RePrism 利用可见工作流信息，对候选请求—表示对进行排序以执行。其资源管理器负责协调可复用资源的准入与驱逐。在 1.8 个会话/秒的动态工作负载下，RePrism 实现了 S-LoRA 的 1.37× 倍请求吞吐量，并将 P95 SCT 降低了 4.10 倍。在受控工作负载下，相较于 S-LoRA，它将平均 TTFT 降低了 5.26 倍。联合评分相比于先按执行评分对请求进行排序、再选择其表示，可使请求吞吐量提高 2.7%，并将平均 SCT 降低 12.3%。

参考文献

[1] Amey Agrawal、Nitin Kedia、Ashish Panwar、Jayashree Mohan、Nipun Kwatra、Bhargav S. Gulavani、Alexey Tumanov 和 Ramachandran Ramjee。《使用 Sarathi-Serve 驯服大语言模型推理中的吞吐量—延迟权衡》。载于 第 *18* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第 117–134 页，2024年。https://www.usenix.org/conference/osdi24/presentation/agrawal。

[2] Rujikorn Charakorn、Edoardo Cetin、Yujin Tang 和 Robert Tjarko Lange。Text-to-LoRA：即时变换器适配。在《机器学习国际会议论文集》中，第 267 卷，第 7485–7514 页，2025年。https://proceedings.mlr.press/v267/charakorn25a.html。

[3] Rujikorn Charakorn、Edoardo Cetin、Shinnosuke Ue- saka 和 Robert T. Lange。《文档到 LoRA：学习即时内化上下文》。*arXiv* 预印本 *arXiv:2602.15902*，2026年。https://doi.org/10.48550/arX iv.2602.15902。[4] Lequn Chen、Zihao Ye、Yongji Wu、Danyang Zhuo、Luis Ceze 和 Arvind Krishnamurthy。《Punica：多租户 LoRA 服务》。载于机器学习与系统论文集，第 6 卷，第 1–13 页，2024年。https://proceedings.mlsys.org/paper_files/paper/2024/hash/054de805fcceb78a201f5e9d53c85908-Abstract-Conference.html。[5] T. Chen、H. Fang、P. Xia、X. Liu、B. Van Durme、L. Zettlemoyer、J. Gao 和 H. Cheng。《生成式适配器：通过单次前向传播在参数中融入语言模型的上下文》。载于第十三届国际学习表征会议（*ICLR 2025*），2025年。https://openreview.net/forum?id=bc3sUsS6ck。[6] Yihua Cheng、Yuhan Liu、Jiayi Yao、Yuwei An、Xiaokun Chen、Shaoting Feng、Yuyang Huang、Samuel Shen、Kuntai Du 和 Junchen Jiang。《LMCache：面向企业级大语言模型推理的高效键值缓存层》。*arXiv* 预印本 *arXiv:2510.09665*，2025年。https://doi.org/10.48550/arXiv.2510.09665。[7] Eunbi Choi、Yongrae Jo、Joel Jang、Joonwon Jang 和 Minjoon Seo。《用于高效提示的固定输入参数化》。载于计算语言学协会论文集：*ACL 2023*，第 8428–8441 页，2023年。https://doi.org/10.18653/v1/2023.findings-acl.533。[8] Xingjian Diao、Wenbo Li、Yashas Malur Saidutta、Avinash Amballa、Lazar Valkov 和 Srinivas Chappidi。《文档到原子：学习编译和组合记忆原子》。*arXiv* 预印本 *arXiv:2606.12400*，2026年。

[9] I. Gim、Z. Ma、S. S. Lee 和 L. Zhong。《Pie：面向新兴大语言模型应用的可编程服务系统》。载于 *ACM SIGOPS* 操作系统原理第 *31* 届研讨会论文集（*SOSP '25*），第 415–430 页，2025年。https://doi.org/10.1145/3731569。3764814。

[10] In Gim、Guojun Chen、Seung-Seob Lee、Nikhil Sarda、Anurag Khandelwal 和 Lin Zhong。提示缓存：低延迟推理的模块化注意力复用。在《机器学习与系统论文集》中，第 6 卷，第 325–338 页，2024年。https://proceedings.mlsys.org/paper_files/paper/2024/hash/a66caa1703fe34705a4368c3014c1966-Abstract-Conference.html。

[11] Aaron Grattafiori、Abhimanyu Dubey、Abhinav Jauhri、Abhinav Pandey、Abhishek Kadian、Ahmad Al-Dahle、Aiesha Letman、Akhil Mathur、Alan Schelten 和 Alex Vaughan。Llama 3 模型群。*arXiv* 预印本 *arXiv:2407.21783*，2024年。[12] Dongxin Guo、Jikun Wu 和 Siu-Ming Yiu。SAGA：面向 GPU 集群上人工智能智能体推理的工作流原子调度。载于第 *35* 届高性能并行与分布式计算国际研讨会论文集（*HPDC '26*），2026年。https://doi.org/10.1145/3806645.3807598。[13] Sirui Hong、Mingchen Zhuge、Jonathan Chen、Xiawu Zheng、Yuheng Cheng、Jinlin Wang、Ceyao Zhang、Zili Wang、Steven Ka Shing Yau、Zijuan Lin、Liyang Zhou、Chenyu Ran、Lingfeng Xiao、Chenglin Wu 和 Jürgen Schmidhuber。MetaGPT：面向多智能体协作框架的元编程。载于第十二届国际学习表征会议（*ICLR 2024*），2024年。https://openreview.net/forum?id=VtmBAGCN7o。[14] Edward J Hu、Yelong Shen、Phillip Wallis、Zeyuan Allen- Zhu、Yuanzhi Li、Shean Wang、Lu Wang 和 Weizhu Chen。LoRA：大语言模型的低秩适配。*arXiv* 预印本 *arXiv:2106.09685*，2021年。[15] Nikoleta Iliakopoulou、Jovan Stojkovic、Chloe Alverti、Tong Xu、Hubertus Franke 和 Josep Torrellas。Chameleon：面向多适配器大语言模型推理环境的自适应缓存与调度。载于第 *58* 届 *IEEE/ACM* 微体系结构国际研讨会论文集（*MICRO '25*），第 217–231 页，2025年。https://doi.org/10.1145/3725843.3756083。[16] Hyesung Jeon、Hyeongju Ha 和 Jae-Joon Kim。LRA-gent：面向多 LoRA 大语言模型智能体的高效 KV 缓存共享。载于第 *43* 届国际机器学习会议论文集，第 306 卷，2026年。https://arxiv.org/abs/2602.01053。

[17] Huiqiang Jiang、Qianhui Wu、Chin-Yew Lin、Yuqing Yang 和 Lili Qiu。LLMLingua：通过压缩提示词加速大语言模型推理。载于*2023* 年自然语言处理中的经验方法会议论文集，第 13358–13376 页，2023年。https://doi.org/10.18653/v1/2023.emnlp -main.825。

[18] Huiqiang Jiang、Qianhui Wu、Xufang Luo、Dong- sheng Li、Chin-Yew Lin、Yuqing Yang 和 Lili Qiu。LongLLMLingua：通过提示词压缩加速并增强长上下文场景中的大语言模型。在第 *62* 届计算语言学协会年会论文集，第 *1* 卷：长篇论文，第 1658–1677 页，2024年。https://doi.org/10.186 53/v1/2024.acl-long.91。

[19] Jordan Juravsky、Bradley Brown、Ryan Ehrlich、Daniel Y. Fu、Christopher Ré 和 Azalia Mirhoseini。Hydragen：利用共享前缀实现高吞吐量大语言模型推理。*arXiv* 预印本 *arXiv:2402.05099*，2024年。https://doi.org/ 10.48550/arXiv.2402.05099。

[20] Omar Khattab、Arnav Singhvi、Paridhi Maheshwari、Zhiyuan Zhang、Keshav Santhanam、Sri Vardhamanan、Saiful Haq、Ashutosh Sharma、Thomas T. Joshi、Hanna Moazam、Heather Miller、Matei Zaharia 和 Christo- pher Potts。DSPy：将声明式语言模型调用编译为先进流水线。载于 第十二届国际学习表征会议（*ICLR 2024*），2024年。https://openreview.net/for um?id=sY5N0zY5Od。

[21] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Sheng、Lianmin Zheng、Cody Hao Yu、Joseph E. Gonzalez、Hao Zhang 和 Ion Stoica。利用 PagedAttention 实现大语言模型服务的高效内存管理。载于 第二十九届操作系统原理研讨会论文集（*SOSP '23*），第 611–626 页，2023年。https://doi.org/10.1145/3600006.3613165。

[22] Allison Li、Kristjan Greenewald、Thomas Parnell 和 Navid Azizan。借助激活式 LoRA 通过跨模型 KV 缓存复用实现高效的多适配器大语言模型服务。*arXiv* 预印本 *arXiv:2512.17910*，2025年。https://doi.org/ 10.48550/arXiv.2512.17910。

[23] Guohao Li、Hasan Hammoud、Hani Itani、Dmitrii Khizbullin 和 Bernard Ghanem。CAMEL：用于探索大语言模型社会“心智”的交流式智能体。载于神经信息处理系统进展第 *36* 卷（*NeurIPS 2023*），2023年。https://doi.org/10.52202/075280-2264。

[24] Hanchen Li、Runyuan He、Qiuyang Mang、Qizheng Zhang、Huanzhi Mao、Xiaokun Chen、Hangrui Zhou、Alvin Cheung、Joseph Gonzalez 和 Ion Stoica。Continuum：利用 KV 缓存生存时间实现高效且稳健的多轮大语言模型代理调度。*arXiv* 预印本 *arXiv:2511.02230*，2025年。https://doi.org/10.4 8550/arXiv.2511.02230。

[25] Suyi Li、Hanfeng Lu、Tianyuan Wu、Minchen Yu、Qizhen Weng、Xusheng Chen、Yizhou Shan、Binhang Yuan 和 Wei Wang。Toppings：面向大语言模型推理的 CPU 辅助、秩感知适配器服务。载于 *2025*年 *USENIX* 年度技术会议（*USENIX ATC 25*），第 613–629 页，2025年。https://www.usenix.org/con ference/atc25/presentation/li-suyi-toppings。

[26] Chaofan Lin、Zhenhua Han、Chengruidong Zhang、Yuqing Yang、Fan Yang、Chen Chen 和 Lili Qiu。Par-rot：利用语义变量高效服务基于大语言模型的应用。在 第*18*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第929–945 页，2024年。https://www.usenix.org/confere nce/osdi24/presentation/lin-chaofan。

[27] Yuhan Liu、Yuyang Huang、Jiayi Yao、Shaoting Feng、Zhuohan Gu、Kuntai Du、Hanchen Li、Yihua Cheng、Junchen Jiang、Shan Lu、Madan Musuvathi 和 Esha Choukse。DroidSpeak：跨微调模型变体共享 KV 缓存。在 第*23*届 *USENIX* 网络系统设计与实现研讨会（*NSDI 26*），第319–338页，2026年。https://www.usenix.org/con ference/nsdi26/presentation/liu-yuhan。

[28] Yuhan Liu、Hanchen Li、Yihua Cheng、Shihan Ray、Yuyang Huang、Qizheng Zhang、Kuntai Du、Jiayi Yao、Shan Lu、Ganesh Ananthanarayanan、Michael Maire、Henry Hoffmann、Ari Holtzman 和 Junchen Jiang。《CacheGen：用于快速大语言模型服务的 KV 缓存压缩与流式传输》。载于*ACM SIGCOMM 2024*会议论文集，第 38–56 页，2024年。https://doi.org/10.1145/3651890.3672274。

[29] Michael Luo、Xiaoxiang Shi、Colin Cai、Tianjun Zhang、Justin Wong、Yichuan Wang、Chi Wang、Yanping Huang、Zhifeng Chen、Joseph E. Gonzalez 和 Ion Stoica。《Agentix：面向通用程序的大语言模型智能体高效服务引擎》。载于第*23*届 *USENIX* 网络系统设计与实现研讨会（*NSDI 26*），第 2443–2459 页，2026年。https://www.usenix.org/c onference/nsdi26/presentation/luo。

[30] Jesse Mu、Xiang Lisa Li 和 Noah D. Goodman。《使用 GistTokens 学习压缩提示词》。载于 神经信息处理系统进展，第 36 卷，第 19327–19352 页，2023年。https://proceedings.neurips. cc/paper_ files/paper/2023/hash/3d77c6dcc7f143aa215 4e7f4d5e22d68-Abstract-Conference.html。

[31] Zaifeng Pan、Ajjkumar Patel、Yipeng Shen、Zhengding Hu、Yue Guan、Wan-Lu Li、Lianhui Qin、Yida Wang、

以及 Yufei Ding。KVFlow：用于加速基于大语言模型的多智能体工作流的高效前缀缓存。载于神经信息处理系统进展，第 38 卷，第 139912–139931 页，2025年。https://openre view.net/forum?id=5Iw1nDtYmT。

[32] Zaifeng Pan、Yipeng Shen、Zhengding Hu、Zhuang Wang、Aninda Manocha、Zheng Wang、Zhongkai Yu、Yue Guan 和 Yufei Ding。ScaleSim：使用基于调用距离的内存管理为大规模多智能体仿真提供服务。*arXiv* 预印本 *arXiv:2601.21473*，2026年。https://doi.org/10.48550/arX iv.2601.21473。

[33] Zhuoshi Pan、Qianhui Wu、Huiqiang Jiang、Menglin Xia、Xufang Luo、Jue Zhang、Qingwei Lin、Victor Rühle、Yuqing Yang、Chin-Yew Lin、H. Vicky Zhao、Lili Qiu 和 Dongmei Zhang。LLMLingua-2：面向高效且忠实的任务无关提示词压缩的数据蒸馏。载于 计算语言学协会论文集：*ACL 2024* 年研究成果，第 963–981 页，2024年。https://doi.org/10.18653/v1/2024.findings-acl.57。

[34] Pratyush Patel、Esha Choukse、Chaojie Zhang、Aashaka Shah、Íñigo Goiri、Saeed Maleki 和 Ricardo Bianchini。Splitwise：使用阶段拆分实现高效的生成式大语言模型推理。载于 *2024*年第*51*届*ACM/IEEE*计算机体系结构国际年会（*ISCA*），第118–132页，2024年。

[35] Abhinav Rastogi、Xiaoxue Zang、Srinivas Sunkara、Raghav Gupta 和 Pranav Khaitan。迈向可扩展的多领域对话代理：模式引导的对话数据集。载于 *AAAI*人工智能会议论文集，第34卷，第8689–8696页，2020年。

[36]K. Santhanam、D. Raghavan、M. S. Rahman、T. Venkatesh、N. Kunjal、P. Thaker、P. Levis 和 M. Zaharia。ALTO：面向复合人工智能系统的高效网络编排器。载于 第*4*届机器学习与系统研讨会（*EuroMLSys '24*）论文集，第117–125页，2024年。https://doi.org/10.1145/3642970.3655844。

[37] Ying Sheng、Shiyi Cao、Dacheng Li、Coleman Hooper、Nicholas Lee、Shuo Yang、Chia-Hsiang Chou、Banghua Zhu、Lianmin Zheng、Kurt Keutzer、Joseph E. Gonzalez 和 Ion Stoica。《SLoRA：可扩展地服务数千个 LoRA适配器》。载于机器学习与系统论文集，第 6 卷，第 296–311 页，2024年。https: // proceedings.mlsys.org/paper_files/paper/2024/hash/906419cd502575b617cc489a1a696a67-Abstract-Conference.html。

[38] Ying Sheng、Shiyi Cao、Dacheng Li、Banghua Zhu、Zhuohan Li、Danyang Zhuo、Joseph E Gonzalez 和 Ion Stoica。《大语言模型服务中的公平性》。载于第*18*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第 965–988 页，2024年。

[39] Jiuchen Shi、Hang Zhang、Yixiao Wang、Quan Chen、Yizhou Shan、Kaihua Fu、Wei Wang 和 Minyi Guo。《ELORA：面向多 LoRA 大语言模型服务的高效 LoRA 与键值缓存管理》。载于*2026*年 *IEEE* 高性能计算机体系结构国际研讨会（*HPCA*），第 1–14 页，2026年。https://doi.org/ 10.1109/HPCA68181.2026.11408492。

[40] H. Shin、L. Ji、Y. Gong、S. Kim、E. Choi 和 M. Seo。生成式提示内化。载于计算语言学协会美洲分会 *2025* 年会议论文集：人类语言技术，第 *1* 卷：长篇论文，第 7338–7363 页，2025年。https://doi.org/10.186 53/v1/2025.naacl-long.376。

[41] Charlie Snell、Dan Klein 和 Rui Zhong。通过蒸馏上下文进行学习。*arXiv* 预印本 *arXiv:2209.15189*，2022年。https://doi.org/10.48550/arXiv.2209.15189。

[42] Vikranth Srivatsa、Zijian He、Reyna Abhyankar、Dong- ming Li 和 Yiying Zhang。Preble：面向大语言模型服务的高效分布式提示调度。载于第十三届国际学习表征会议（*ICLR 2025*），2025年。https://openreview.net/forum?id=meKEKDhdnx。

[43] Biao Sun、Ziming Huang、Hanyu Zhao、Wencong Xiao、Xinyi Zhang、Yong Li 和 Wei Lin。Llumnix：大语言模型服务的动态调度。载于第*18*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第 173–191 页，2023年。https: // www.usenix.org/conference/osdi24/presentation/sun-biao。

[44] X. Tan、Y. Jiang、Y. Yang 和 H. Xu。借助 Ayo 实现基于大语言模型的应用端到端优化。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*2*卷（*ASPLOS '25*），第 1302–1316 页，2025年。https://doi.org/10.1145/ 3676641.3716278。

[45] Hugo Touvron、Louis Martin、Kevin Stone、Peter Albert、Amjad Almahairi、Yasmine Babaei、Nikolay Bashlykov、Soumya Batra、Prajjwal Bhargava 和 Shruti Bhosale。Llama 2：开放基础模型与微调聊天模型。*arXiv* 预印本 *arXiv:2307.09288*，2023年。

[46] Noppanat Wadlom、Junyi Shen 和 Yao Lu。《智能体工作流的高效大语言模型服务：数据系统视角》。*ACM* 数据管理会议录，4(3):1–29，2026年。https://doi.org/10.1145/3802 046。

[47] Shao Wang、Rui Ren 和 Lin Gui。《ForkKV：通过写时复制的解耦键值缓存扩展多 LoRA 智能体服务》。*arXiv* 预印本 *arXiv:2604.06370*，2026年。https://doi.org/10.48550/arXiv.2604.06370。

[48] Bingyang Wu、Ruidong Zhu、Zili Zhang、Peng Sun、Xu- anzhe Liu 和 Xin Jin。dLoRA：为 LoRA 大语言模型服务动态编排请求与适配器。载于第 *18* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第 911–927 页，2024年。https://www.usenix.org/conference/osdi24/presentation/wu-bingyang。

[49] Qingyun Wu、Gagan Bansal、Jieyu Zhang、Yiran Wu、Beibin Li、Erkang Zhu、Li Jiang、Xiaoyun Zhang、Shaokun Zhang、Jiale Liu、Ahmed A wadallah、Ryen W. White、Doug Burger 和 Chi Wang。《AutoGen：通过多智能体对话赋能新一代大语言模型应用》。载于语言建模会议（*COLM 2024*），2024年。https://openreview.net/forum?id=BAakY1 hNKS.[50] An Yang、Baosong Yang、Beichen Zhang、Binyuan Hui、Bo Zheng、Bowen Yu、Chengyuan Li、Dayiheng Liu、Fei Huang、Haoran Wei、Huan Lin、Jian Yang、Jian-hong Tu、Jianwei Zhang、Jianxin Yang、Jiaxi Yang、Jin- gren Zhou、Junyang Lin、Kai Dang、Keming Lu、Ke- qin Bao、Kexin Yang、Le Yu、Mei Li、Mingfeng Xue、Pei Zhang、Qin Zhu、Rui Men、Runji Lin、Tianhao Li、Tingyu Xia、Xingzhang Ren、Xuancheng Ren、Yang Fan、Yang Su、Yichang Zhang、Yu Wan、Yuqiong Liu、Zeyu Cui、Zhenru Zhang 和 Zihan Qiu。《Qwen2.5 技术报告》。*CoRR*，abs/2412.15115，2024年。https: //doi.org/10.48550/ARXIV.2412.15115.[51] John Yang、Carlos E. Jimenez、Alexander Wettig、Kil- ian Lieret、Shunyu Yao、Karthik Narasimhan 和 Ofir Press。《SWE-agent：智能体—计算机接口实现自动化软件工程》。载于神经信息处理系统进展第 *37* 卷（*NeurIPS 2024*），2024年。https://doi.org/10.52202/079017-1601.[52] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Karthik R. Narasimhan 和 Yuan Cao。《ReAct：在语言模型中协同实现推理与行动》。载于国际学习表征会议（*ICLR 2023*），2023年。https://openreview.net/forum ?id=WE vluYUL-X._[53] Lei Ye、Ze Tao、Yong Huang 和 Yang Li。《ChunkAttention：借助前缀感知 KV 缓存和两阶段分区实现高效自注意力》。载于计算语言学协会第 *62* 届年会论文集，第 *1* 卷：长篇论文，第 11608–11620 页，2024年。https://doi.org/10.18653/v1/2024.acl-long. 623。

[54] Gyeong-In Yu、Joo Seong Jeong、Geon-Woo Kim、Soo-jeong Kim 和 Byung-Gon Chun。《Orca：面向基于 Transformer 的生成模型的分布式服务系统》。载于第 *16* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 22*），第 521–538 页，2022年。https://www.usenix.org/conference/osdi22/presentation/yu。

[55] Lingfan Yu、Jinkun Lin 和 Jinyang Li。使用 Pensieve 实现有状态大语言模型服务。载于第二十届欧洲计算机系统会议论文集（*EuroSys '25*），第 144–158 页，2025年。https: //doi.org/10.1145/3689031.3696086。

[56] Yichao Yuan、Ankita Nayak、Souvik Kundu 和 Nishil Talati。智能体人工智能工作负载特征。*arXiv* 预印本 *arXiv:2605.26297*，2026 年。

[57] Wei Zhang、Zhiyu Wu、Yi Mu、Rui Ning、Banruo Liu、Nikhil Sarda、Myungjin Lee 和 Fan Lai。JITServe：基于不精确请求信息、面向服务级目标的 LLM 服务。发表于第*23*届 *USENIX* 网络系统设计与实现研讨会（*NSDI 26*），第 825–848 页，2026年。[58] Lianmin Zheng、Liangsheng Yin、Zhiqiang Xie、Chuyue Sun、Jeff Huang、Cody Hao Yu、Shiyi Cao、Christos Kozyrakis、Ion Stoica、Joseph E. Gonzalez、Clark Barrett 和 Ying Sheng。SGLang：高效执行结构化语言模型程序。发表于神经信息处理系统进展，第 37 卷，第 62557–62583 页，2024年。https://proceedings.neurip s.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html。[59] Ziyang Zheng、Zeju Li、Xiangyu Wen、Jianyuan Zhong、Junhua Huang、Lei Chen、Mingxuan Yuan 和 Qiang Xu。上下文蒸馏作为潜在记忆管理。*arXiv* 预印本 *arXiv:2605.28889*，2026年。[60] Yinmin Zhong、Shengyu Liu、Junda Chen、Jianbo Hu、Yibo Zhu、Xuanzhe Liu、Xin Jin 和 Hao Zhang。Dist-Serve：通过分离预填充和解码，实现面向有效吞吐量优化的大语言模型服务。发表于第*18*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第 193–210 页，2024年。https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin。[61] Zhe Zhou、Xuechao Wei、Jiejing Zhang 和 Guangyu Sun。PetS：参数高效的 Transformer 服务统一框架。发表于*2022*年 *USENIX* 年度技术会议（*USENIX ATC 22*），第 489–504 页，2022年。https://www.usenix.org/conference/atc22/presentation/zhou-zhe。

补充材料

本补充材料进一步介绍论文中所用方法的细节。A节介绍第 4 节使用的评分和资源管理规则，并报告批次选择的 CPU 开销。B节进一步阐述第 3 节中的提示内化流程。

A 评分与资源管理

A.1 会话进度值与归一化

第 4.3 节根据两个项计算节点执行价值：会话进度值 $P_t(s,v)$ 和等待项 $W_t(s)$，后者反映会话 s 等待调度所花费的时间。$P_t(s,v)$会提高即将完成的会话的优先级；在一个会话内，则会提高剩余路径更长的节点的优先级。

令 L_v 为可见工作流中从节点 v 到任一终端节点的最长估计时间，其中包括 v 本身以及该路径上所有后续节点所需的时间。令 L_s 为会话 s 的可见工作流中最长的剩余路径时长。为计算 L_v，我们首先估算每个节点的剩余执行时间。对于尚未运行的节点，我们采用其在 Direct 下的估计执行时间。对于正在运行的节点，则根据其剩余输出预算的比例缩减该估计值。对于尚未到达服务器的节点，还会加上已知的工具等待时间。

然后，我们从终端节点开始反向计算。终端节点的路径时长等于其自身估计的剩余时间。对于其他每个节点，我们将其自身估计的剩余时间加上其直接后继节点中最大的路径时长。所有节点中的最大值给出 L_s。我们将会话进度值定义为

$$P_t(s,v) = \frac{H}{H + L_s} \cdot \frac{L_v}{L_s} \tag{A1}$$

我们将 H 设为 60 秒，以确定时间尺度。随着 L_s 减小，第一个比值增大，从而为剩余路径较短的会话赋予更高的进度值。在同一会话内，第二个比值 L_v/L_s 为剩余路径较长的节点赋予更高的进度值。

我们引入等待项 $W_t(s)$，以防止会话等待过长。仅当会话至少有一个就绪请求且没有正在运行的请求时，等待时间 w_s 才会累积。我们计算 $W_t(s) = \min\left(1, w_s/H\right)$，因此执行请求所花费的时间不会增加该项。然后，我们分别对节点执行价值和表示价值进行归一化为 $[0.5, 1]$，并将二者相乘。这样可以防止某个值因其原始分数异常大或异常小而主导秩。

A.2 表示评分的成本估计

第 4.3 节将表示价值分解为即时执行收益 $V\mathrm{now}_t(v,m)$、未来复用收益 $G_t^{\mathrm{reuse}}(v,m)$ 以及资源机会成本 $O_t^{\mathrm{resource}}(v,m)$。为计算 $V\mathrm{now}_t(v,m)$，我们首先估计同一请求分别在 Direct 和表示 m 下的剩余执行时间，分别记为 T_D 和 T_m。二者之差即为使用 m 所节省的执行时间。令 ΔT_{load} 表示准备 m 所需的额外时间。即时执行收益为

$$V_t^{\mathrm{now}}(v,m) = \frac{T_D - T_m - \Delta T_{\mathrm{load}}}{T_D} \tag{A2}$$

T_D 和 T_m 都包括预填充和解码。为计算 ΔT_{load}，RePrism 会比较添加候选项前后的资源准备计划，并且只计算为 m 加载或构建资源所需的额外时间。这些时间估计值来自离线分析。当长度与某个已测量的配置匹配时，RePrism 可以直接使用相应的测量值。否则，它会根据实际输入长度、输出预算和适配器秩估计时间。

未来复用收益 $G_t^{\mathrm{reuse}}(v,m)$ 体现了后续请求复用当前表示所带来的价值。调度器仅考虑工作流中可见的待处理请求。为避免同一收益被重复计算，它会排除当前请求以及本轮中已选入批次的请求。令 ΔT_u 表示后续请求 u 通过复用预计可节省的执行时间。未来复用收益为

$$G_t^{\mathrm{reuse}}(v,m) = \frac{1}{T_D} \sum_u \frac{p_u \Delta T_u}{1 + d_u / T_0} \tag{A3}$$

这里，p_u 是请求 u 将执行的估计概率。对于已提交的请求，RePrism 将 p_u 设为 1；对于其余请求，则使用已知的分支概率。项 d_u 表示 u 的执行依赖项得到满足前的估计时间。我们将 T_0 设为 1 秒，以确定用于折扣计算的时间尺度。因此，执行可能性更高且依赖关系更快得到满足的请求，对未来复用收益的贡献更大。当同一后续请求能够从多个共享资源中受益时，RePrism 会将节省的总时间分摊到这些资源上。

添加候选项也可能给 GPU 内存带来压力，并驱逐驻留缓存资源。资源机会成本 $O_t^{\mathrm{resource}}(v,m)$ 衡量恢复这些资源所需的额外成本。RePrism 会估计添加候选项前后哪些资源将被驱逐，并比较重建这些资源的成本。它将差值除以 T_D，并将结果裁剪至 $[-1, 1]$，以获得用于

评分的成本项。重建成本估计同样来自离线分析。

A.3 批处理与动态 KV 使用峰值

第 4.4 节检查候选批次与运行中的批次合并时所需的 GPU 内存。该需求取决于预填充的执行方式，以及动态键值使用量在解码过程中的变化。每生成一个令牌，键值使用量就会增加；而较短的请求完成后会释放其内存。因此，仅凭当前使用量无法反映批次峰值，并且各个请求可能在不同时间达到峰值。我们按照批次中请求的剩余输出预算降序排列请求。批次在解码期间的峰值键值使用量为

$$T_{\text{decode}} = \max_{k} \left(\sum_{i=1}^{k} u_i + k\, o_k \right) \tag{A4}$$

其中，u_i 表示请求 i当前持有的动态令牌数量，o_i 表示其剩余输出预算。我们对从 1 到请求数量的每个 k 计算该表达式，并取最大值作为 T_{decode}。RePrism 使用公式 (A4)估算解码期间的动态键值需求，并在检查 GPU 内存容量时，同时考虑预填充期间的内存使用量。

A.4 调度开销

我们首先填充就绪队列，然后在 CPU 上对 RePrism 的批次选择例程进行一次计时。每个请求都属于一个独立会话，该会话包含一个节点，并且可以使用全部四种表示。图 S1 展示了就绪请求数为 1 至 128（即候选项数为 4 至 512）时的延迟结果。

中位延迟从 4 个候选项时的 1.01 ms 上升至 128 个时的 17.19 ms，以及 512 个时的 62.45 ms。在 512 个候选项时，P95 和 P99 分别达到 64.21 ms 和 65.64 ms。

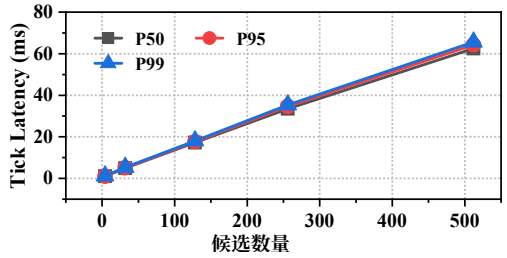

图 S1：每个调度轮次（时钟周期）的 CPU 延迟。每个百分位数均基于每种队列大小下的 1000 个时钟周期计算，并取五次运行的平均值。

B 提示内化的训练与行为检查

我们使用 Schema-Guided Dialogue（SGD）数据集中的 Media_1 状态跟踪任务，来说明第 3.1 节所述的适配器训练流程。在此任务中，P 指定状态更新规则，R 提供先前动作和已知信息，而Q 表达当前请求。目标答案取自 SGD 标注。我们在 Llama-2-7B-Chat 上进行训练，并固定基础模型权重。我们仅更新注意力层中的 LoRA 参数，依据模型对目标答案的预测误差进行训练。附录 C 中的训练示例展示了固定提示词和动态输入如何决定目标输出。Direct 同时接收P 和动态输入。加载适配器后，模型仅接收R 和Q。

在此示例中，当前请求要求启用字幕，因此将 SUBTITLES 设置为 True。意图仍为 PlayMovie，因此标题沿用 R 中的内容，而导演和类型字段设置为 NONE，因为它们不适用于此操作。因此，目标输出同时反映了新请求和相关的先前状态。训练集和开发集分别包含 48 个和 12 个示例，另有一个包含 24 个示例的独立集合用于行为资格认证。在评估期间，我们在每一步选择概率最高的令牌，并生成最多 64 个新令牌，以检查适配器是否遵循这些留出示例上的规则。我们首先检查五个字段是否全部存在，然后将预测的意图和所需的字段值与目标进行比较。行为得分是通过这些检查的示例所占的比例。字段顺序可以不同，只要输出符合预定义的行为规则，即使逐令牌输出与 Direct 不同也没有关系。如果适配器未达到要求的行为得分，且其当前秩低于预设阈值，我们就提高秩，重新训练适配器，并重复这些检查。我们选择能够通过检查的最低秩。

C 媒体_1 StateTrackingExample

固定提示词 P （摘录）	根据 CURRENT_UTTERANCE、 PREVIOUS_INTENT 和 PREVIOUS_SLOTS跟踪 Media_1 状态。仅返回更新后的状态。 使用 CURRENT最具体的提示：字幕 -> PlayMovie；..... 对于相同的意图，携带未被 CURRENT 替换的相关 PREVIOUS字段， 逐字翻译 . SUBTITLES 需要包含字面词“字幕”。 使用/开启/添加/启用/显示/保持开启/需要/想要表示 True；..... PlayMovie 迫使 DIRECTED_BY=NONE和 GENRE=NONE。 输出恰好五行非空的 FIELD=value ， 每行各出现一次。顺序不限。
动态输入 $R + Q$	CURRENT_UTTERANCE=Actually, please play with subtitles. PREVIOUS_INTENT=PlayMovie PREVIOUS_SLOTS=title=Body Double
预期输出	INTENT=PlayMovie DIRECTED_BY=NONE GENRE=NONE SUBTITLES=True TITLE=Body Double