arXiv:2512.15834v1 [cs.PL] 17 Dec 2025

# 通过推测性工具调用优化智能体语言模型推理

DanielNichols PrajwalSinghania劳伦斯利弗莫尔国家实验室 马里兰大学CharlesJekel Abhinav Bhatele劳伦斯利弗莫尔国家实验室 马里兰大学Harshitha Menon劳伦斯利弗莫尔国家实验室

## 摘要

语言模型（语言模型）正日益依赖外部工具。基于语言模型的智能体框架经常借助此类工具与环境交互，以搜索文件、运行代码、调用 API 等。此外，现代的推理型语言模型会使用网络搜索和 Python 代码执行等工具来增强其推理能力。工具虽然大幅提升了语言模型的能力，但也会在推理过程中造成性能瓶颈。本文提出了新颖的系统优化方法，通过推测工具调用，并强制序列常驻于推理引擎中，尽可能减少开销，从而解决此类性能瓶颈。在托管语言模型智能体的推理服务时，我们的优化可将吞吐量提升数百每秒词元数。我们对算法进行了理论分析，以深入了解哪些推测配置能够带来最佳性能。此外，我们建议推出新的“工具缓存”API 端点，使语言模型提供商能够轻松采用这些优化。

## 1 引言

工具与函数调用让语言模型（LM）能够与外部环境交互并收集更多上下文，从而胜任对话之外的任务 [12–14, 20]。尤其是，基于语言模型的智能体工具和框架往往完全依赖外部工具，因为它们的设计目的就是与环境交互，以解决问题或完成任务。软件工程智能体（SWE agents）等常见智能体 [9, 19] 需要与源代码文件和命令行交互以执行操作。尽管访问外部工具能带来丰富得多的能力，并使语言模型能够解决长周期的现实世界任务，但这也给传统推理流水线带来了若干性能瓶颈。模型不再进行单次连续生成，而是在生成与工具调用之间交替，且通常同时处理多个会话，因等待工具执行完毕而产生停顿。

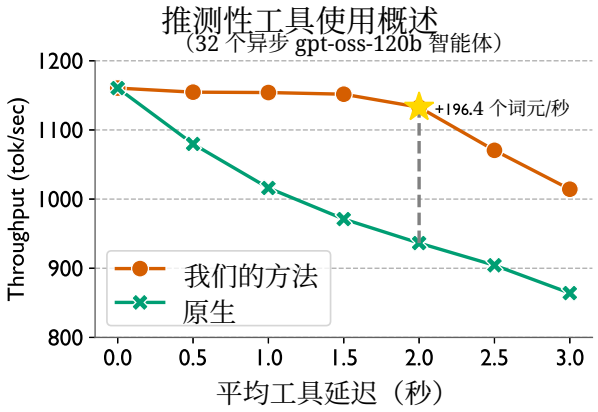



图1：当32个 gpt-oss-120b 智能体使用该方法进行推理时，我们的方法可使 vLLM 服务器的性能提升高达每秒196个词元。

随着工具中心型智能体在代码副驾驶、个人助理和自主工作流中日益普及，使用这些智能体的开销越来越多地取决于等待工具所花费的时间。每次工具调用都会迫使生成暂停并将控制权交还给用户，由用户运行工具并发送回其输出。这种中断驱动过程带来了严格的顺序依赖，单个工具调用的延迟会不断累积，显著延长总生成时间。此外，被驱逐序列和工具输出必须重新调度回推理引擎，从而产生额外开销。即使采用前缀缓存等优化 [5, 7, 18, 22],，移除序列、处理工具输出以及重新调度序列以继续生成，仍会产生显著开销。在多租户环境中，这些开销还会进一步加剧：大量智能体争夺资源，且许多并发提示会驱逐缓存中已有的条目，使调度和前缀缓存优化更难实施，进而影响

KV 缓存。随着智能体语言模型日益普及，并需要实现更高的经济可行性和效率，消除这种顺序依赖并减少工具密集型工作负载中的开销至关重要。

从系统角度看，打破工具调用引入的严格顺序依赖并降低驱逐开销并非易事。工具延迟可能相差数个数量级，且取决于外部服务、文件系统输入/输出或用户定义代码，因此难以预测或限定。对于长时间运行的工具，端到端延迟不可避免地由其主导；而对于短时工具，驱逐和重新进入引擎所带来的开销，可能超过任何潜在的延迟隐藏策略。此外，将工具执行与模型推进解耦会带来正确性挑战，若天真地提前或乱序运行工具，可能会浪费计算，产生未使用的结果，或得到与模型最终决策不一致的输出。最后，推测解码等现有推理优化 [3, 8] 以及优化的 KV 缓存管理 [7] 完全在推理引擎内部实现，并将工具视为不透明的中断，因此客户端创新难以影响引擎性能。

我们通过引入推测性工具执行来应对智能体工具调用中的这些挑战，并提出两种变体。首先，我们介绍一种客户端策略，无需修改推理引擎即可打破工具执行的顺序性，因此能够立即兼容现有推理服务。其次，我们引入一种引擎级机制，使推理系统能够直接接收工具输出，并消除与驱逐相关的开销。这些策略使我们能够在生成过程中隐藏部分工具执行时间，适用于无状态且执行成本低廉的工具。此外，我们为这两种方法提供了分析性能模型，并讨论工具执行时长、预测准确率和解码延迟对性能提升的影响。我们在一系列工作负载上评估了所提出的方法，结果表明吞吐量最高可提升数百个每秒词元数。图1 展示了与采用工具调用智能体的标准推理相比，我们的方法所带来的收益。

在这项工作中，我们做出了以下重要贡献：

- 提出了两种新颖的方法，通过推测工具调用来优化智能体的推理：推测执行任务，并让提示常驻于引擎中。
- 对我们的方法的局限性进行了理论分析，并指出哪些配置能够取得最佳结果。
- 我们详细研究了两种方法在使用工具调用的智能体工作流中的表现。

## 2 背景

本节介绍工具调用和推测解码的背景。

### 2.1 工具调用

大多数现代语言模型都使用工具调用（或函数调用）作为与外部系统（命令行、API 等）交互的接口。在典型设置中，用户会提供（1）描述任务的自然语言提示词，以及（2）一份可用工具列表及其结构化参数模式。模型收到这些输入后，会以自然语言或结构化的工具调用作出响应，指定要调用的工具及其参数。客户端随后执行该工具，并将工具输出连同原始对话一起发送回模型。在语言模型智能体中，这一过程通常会重复进行，直到模型获得足够的上下文以生成最终输出，或完成任务为止。

从推理引擎的角度来看，每次工具调用都会将生成过程拆分为多个较短的请求，并由外部执行工具将这些请求分隔开。对于任何请求，引擎都会先在预填充 阶段处理提示中的所有词元，以生成第一个输出词元，然后在解码阶段通过自回归方式生成后续词元。生成工具调用后，序列的后续解码会暂停，并在客户端执行工具期间将其从活动批次中移除。工具结果就绪后，客户端会提交一个新请求，其中包含原始提示、中间文本和工具输出，迫使引擎重新预填充整个历史记录，之后才能解码下一个工具调用或最终输出。对于涉及许多工具和较长多步计划的智能体工作负载，这会导致生成过程支离破碎，并因反复使用不断变长的提示重启生成而产生开销。在较长的智能体执行过程中，预填充较长的提示以及等待工具响应，很容易成为严重的瓶颈。

目前有两种常见的优化方法可以降低这些开销：

**并行工具调用。** 模型不再每轮只发出一次工具调用，而是可以在一次响应中返回多个相互独立的工具调用 [4, 15]。这样，客户端就能并发执行工具，然后在一次响应中将结果返回给模型。这减少了模型与工具之间的往返次数，但在推理过程中仍会出现间隙，因为引擎需要将控制权交给客户端。此外，有时工具必须按顺序调用，因为它们之间存在依赖关系，或者模型需要先获取某个工具的输出，才能决定是否调用下一个工具。模型也需要经过训练，才能支持并行工具调用。

**前缀缓存。** 许多推理运行时会为最近使用过的提示 [7, 22] 维护前缀缓存，以保存键值状态。如果用户提交

另一条与其不久前提交的请求具有相同历史记录的请求，前缀缓存可在快速进行多轮 API 使用时大幅降低预填充开销。不过，在多租户负载下，它无法消除从批次中移除序列、重新调度这些序列或管理缓存淘汰所带来的开销；如果需要采用隐私保护算法，这些操作的开销实际上可能很高 [17, 18]。

## 2.2 推测解码

推测解码是一种推理优化方法，它利用一个更小、更快的草稿语言模型，为一个更大、更慢的语言模型预先预测词元，从而加速语言模型生成。给定相同的提示，草稿模型会在目标模型之前提出多个推测词元。随后，目标模型会并行验证这些候选词元：只要草稿模型提出的词元符合目标模型的分布，就会予以接受；一旦出现不匹配，算法便会在首次拒绝的位置回退，改为从目标模型采样。Leviathan 等人最早提出了这一方法 [8]，自首次引入以来，它又经过多种改进和调整 [6]。

这种方法之所以有效，有两个原因：草稿模型往往能够准确预测一些较容易的文本，而目标模型则可以通过一次前向传播验证多个词元。后者意味着推理不会变慢：无论如何，目标模型都会执行一次前向传播来采样下一个令牌；如果它验证了推测模型提出的词元，我们就能免费获得多个词元。唯一额外的成本是运行推测模型所需的计算量。

本研究借鉴了推测解码背后的思想，但也有一些细微差异。通常，推测解码中的草稿模型只比目标模型领先 3–10 个词元。这对于推测生成效果很好；但在推测式工具调用中，如果我们在草稿模型生成工具调用时就启动工具，那么工具调用与生成词元的重叠时间就只有几个词元所需的时间。这样最多只能节省几毫秒，而且节省的时间很大程度上会被更长的工具调用耗时和 API 开销所抵消。相反，对于工具，最好尽早进行推测；对于无状态廉价工具，则应尽可能多地进行推测。我们的方法正是围绕这一洞见，以及它与推测解码之间的关键差异而设计的。

# 3 推测式工具调用算法

本研究旨在通过推测工具调用，并在实际需要之前执行这些调用，来优化基于语言模型的智能体中的工具调用性能。我们提出了两种推测式工具调用方法：严格限定在客户端侧的推测方法，可降低工具调用延迟；以及一种推理引擎侧推测算法，可同时缩短解码和预填充时间。

在传统工具调用中，用户首先向 API 端点推理服务器提交提示。服务器开始处理提示，并在遇到停止令牌时将生成的令牌返回给用户。这可能是序列结束，也可能是工具调用令牌。用户收到服务器的响应后，会检查生成停止的原因（例如工具调用、序列结束、超时、最大词元数、API 额度）；如果是工具调用，则使用模型生成的工具参数在本地运行该工具。工具执行完毕后，用户会将输出传回 API，语言模型便可利用工具输出继续生成。该过程详见算法1。

**算法 1**基线工具调用（客户端，无推测性调用）

**要求：** 提示 $p$，聊天历史 $H$，工具$\mathcal{T}$，主语言模型 $\mathcal{M}$
**确保：** 最终模型输出$y$

$r \leftarrow$ 调用 API($\mathcal{M}, p, H, \mathcal{T}$)
**如果** 检查工具调用($r$)**则**
  $\tau \leftarrow$ 提取工具调用($r$)
  $v \leftarrow$ RunTool($\tau$)
  $H' \leftarrow H \cup \{(\tau, v)\}$ ▷ 将工具结果追加到历史记录
  $y \leftarrow$ 调用 API($\mathcal{M}, p, H', \mathcal{T}$) ▷ 返回最终答案
**否则**
  $y \leftarrow$ 渲染($r$) ▷ 不进行工具调用；使用第一遍结果
**结束条件判断**
**返回**$y$

## 3.1 客户端推测性工具调用

我们通过客户端的推测工具调用扩展传统工具调用方法，如算法 2 所述。在此方法中，我们异步地将提示发送至两个 API 端点：主模型 $\mathcal{M}$和推测模型 $\mathcal{S}$。推测模型规模小得多，应能以远低于主模型的延迟返回预估的工具调用。当$\mathcal{S}$返回工具调用时，我们会异步启动该工具并开始执行。我们会将其结果的未来对象存储在工具缓存中。当$\mathcal{M}$最终返回生成的令牌后，我们会检查是否需要工具调用。如果需要，并且缓存中有该工具调用对应的未来对象，我们便等待该未来对象，而不是重新启动工具。图2 概述了我们的方法。我们还可以通过启动多个推测模型实例（在算法 λ 2中表示）进一步提高推测程度。

需要注意的是，我们只能对无状态、低成本的工具进行推测。如果工具或函数需要或会修改状态，那么除非有撤销或回滚机制，否则就无法对其进行推测。这是一种有趣的潜在优化方式，但超出了本文的讨论范围。此外，如果工具调用的金钱成本或执行时间较高，那么若提前启动多次调用却最终发现结果不正确，就并不可取。幸运的是，许多常见的智能体工具，例如网页搜索和文件访问，既成本低廉又无状态，因此有充分的推测空间。

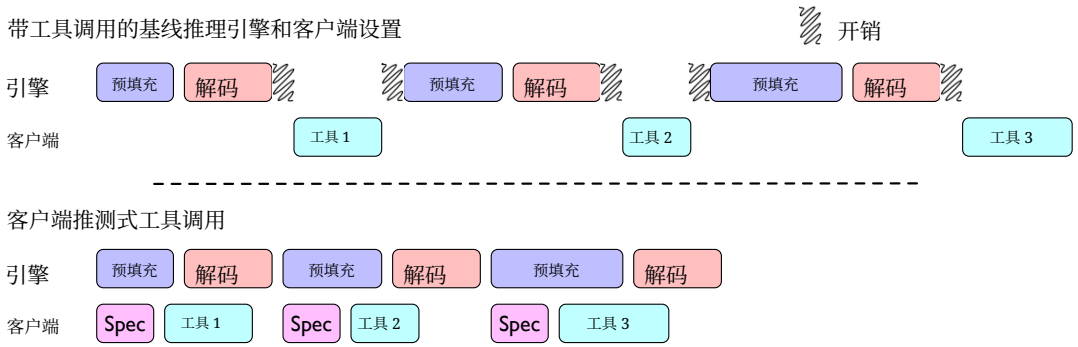



图 2：客户端推测式工具调用方法概览。我们对工具调用进行推测，并使其执行与生成过程重叠。

**算法 2** 客户端推测式工具调用

**要求：** 提示 $p$、聊天历史 $H$、工具 $\mathcal{T}$、采样因子 $\lambda$，主大语言模型 $\mathcal{M}$、推测语言模型 $\mathcal{S}$

**确保：** 最终模型输出 $y$

清 $\leftarrow$ 空缓存映射() ▷缓存：工具调用 $\mapsto$ 未来对象
**生成** 异步 $r_m \leftarrow$ 步调用 API($\mathcal{M}, p, H, \mathcal{T}$) ▷启动主生成
**对于** $i \leftarrow 1$ 到 $\lambda$ **执行** ▷推测样本并行执行
  **生成**：$s_i \leftarrow$ 调用 API($\mathcal{S}, p, H, \mathcal{T}$)
  **如果** 检查工具调用($s_i$)**则**
    $\tau \leftarrow$ 提取工具调用($s_i$)
    **如果** $\tau \notin C$ **则**
      客[$\tau$] $\leftarrow$ 户端工具调用($\tau$) ▷开始正在执行工具；返回一个未来对象
    **结束 if**
  **结束 if**
**结束 for**
$r_m \leftarrow$**等待**$h_m$ ▷等待主生成
**若** 检查工具调用($r_m$)**则**
  $\tau_m \leftarrow$提取工具调用($r_m$)
  **if** $\tau_m \in C$ **则**
    $v \leftarrow$ **等待**$C[\tau_m]$ ▷如有则复用
  **else**
    $v \leftarrow$ 运行工具($\tau_m$) ▷立即执行工具
  **结束条件判断**
  历史$' \leftarrow$ 记录 $\cup \{(\tau_m, v)\}$ ▷将工具结果追加到历史记录
  $y \leftarrow$ 调用 API($\mathcal{M}, p, H', \mathcal{T}$) ▷使用结果调用 $\mathcal{M}$
**否则** ▷不进行工具调用
  $y \leftarrow$ 渲染($r_m$)
**结束 if**
**返回** $y$

### 3.1.1 客户端推测分析

我们考察一种特定情况：用户反复调用 API，且每次都会从 API 收到一个工具调用。这与大多数人工智能代理的使用模式十分相似，因此是一个值得考虑的合理情形。处理<style id='3">N 个用户请求所需的时间，在标准情形（算法1）和推测情形（算法2）下分别如下所示。

avg. $\mathcal{M}$ generation time　　avg. tool call time

$$T_{\text{standard}} = N(G + T)$$

avg. $\mathcal{S}$ generation time

$$T_{\text{spec}} = \alpha N \max\{G, g + T\} + (1-\alpha)N(G + T) \quad (1)$$

$\mathcal{S}$ acceptance rate

$T_{\text{spec}}$ 左侧项表示推测模型预测正确的情况，因此我们只需等待 $g + T$，让工具完成调用。这里取最大值是必要的，因为我们仍然需要等待 $\mathcal{M}$ 完成，以验证右侧的工具调用。因此，如果 $g + T < G$，那么我们仍需等待 $G$。假设 $T$、$g$、$G$ 和 $N$ 均为正的非零值，且 $0 \le \alpha \le 1$，我们可以进一步分析推测算法的加速效果。

$$S_{\text{spec}} = \frac{T_{\text{standard}}}{T_{\text{spec}}} > 1$$
$$= \frac{G+T}{\alpha \max\{G, g+T\} + (1-\alpha)(G+T)} > 1 \quad (2)$$

如果 $g + T \ge G$，那么

$$S_{\text{spec}} = \frac{G+T}{\alpha(g-G) + G + T} > 1$$
$$\alpha(g-G) + G + T < G + T$$
$$\alpha(g-G) < 0$$
$$\Longrightarrow \alpha > 0 \text{ 和 } g < G \quad (3)$$

我们可以立即看出，$_{\text{spec}} > 1$ 当 $SS$ 的接受率大于零，且 $\mathcal{S}$ 比 $\mathcal{M}$ 更快时。情况是

$g + T < G$ 被忽略，因为当 $\alpha > 0$ 时（即推测和工具调用完全被主模型的生成时间掩盖，因此任何正确的推测都应带来加速），这显然会带来大于 1 的加速。根据公式（3），要在客户端通过推测式工具调用实现加速，我们需要找到一个推测模型$\mathcal{S}$，它比主模型更快$\mathcal{M}$，且工具推测准确率大于 $0$。

公式（2）还揭示了一个关键见解：$S_{spec} < 2$，也就是说，这种方法所能实现的最大加速为 2。此外，当 $g \to 0$ 时，加速趋近于 2。图3展示了 $S_{spec}$ 在 $\alpha$、$g/G$ 和 $T$ 取不同值时的分布。我们可以看到上述趋势，即当 $g \to 0^+$ 时 $S_{spec} \to 2$，而 $S_{spec} < 2$；但同时也可以看到，只有当 $T$ 接近 $G$ 时，加速才会显著。当 $T \to 0$ 时，总耗时主要由 $G$ 决定；当 $T \to \infty$ 时，则主要由工具调用决定。因此，当 $T \approx G$ 时，在更多 $\alpha$ 和 $g/G$ 的取值下，我们都能看到显著收益。下文的引理1也证明了上述见解。

**引理1** (加速上界)。设 $G$、$g$、$T > 0$，$g < G$，且 $\alpha \in [0,1]$。定义

$$S_{\text{spec}}(\alpha) = \frac{G+T}{\alpha \max\{G, g+T\} + (1-\alpha)(G+T)}.$$

于是：

1. $S_{spec}(\alpha)$ 关于 $\alpha$ 严格递增，因此对于所有 $\alpha > 0$，$S_{spec}(\alpha) > 1$。
2. $\alpha \in [0,1]$ 上的最大值在 $\alpha = 1$ 处取得，且等于

$$S_{\max} = \frac{G+T}{\max\{G, g+T\}}.$$

3. 此外，

$$S_{\max} \le \frac{2(G+T)}{G+g+T} = 2 - \frac{2g}{G+g+T} < 2.$$

特别地，只有在 $g \to 0^+$ 的极限情况下才能达到上确界 2。

证明。令 $M := \max\{G, g+T\}$，并记

$$S_{\text{spec}}(\alpha) = \frac{G+T}{\alpha M + (1-\alpha)(G+T)} = \frac{G+T}{(G+T) + \alpha(M-(G+T))}.$$

设 $\Delta := M - (G+T)$。由于 $M \le G+T$，因此有 $\Delta \le 0$；这是严格不等式，因为 $G, g, T > 0$。

**(1)** 由于$M \le G+T$以及$G, g, T$ $\alpha > 0$，$S_{spec}(\alpha)$ 的分母随 $\alpha$ 增大而严格递减。因此，我们可以得出结论：$S_{spec}(\alpha)$ 随 $\alpha$ 严格递增。此外，由于 $S_{spec}(0) = 1$，可知对于所有$\alpha > 0$，$S_{spec}(\alpha) > 1$。

**(2)** 单调性意味着 $[0,1]$ 上的最大值在$\alpha = 1$处取得，因此

$$S_{\max} = S_{\text{spec}}(1) = \frac{G+T}{M} = \frac{G+T}{\max\{G, g+T\}}.$$

**(3)** 对于任意 $a, b > 0$，$\max\{a,b\} \ge \frac{a+b}{2}$。将其应用于$a = G$和 $b = g+T$，

$$\begin{aligned} S_{\max} &= \frac{G+T}{\max\{G, g+T\}} \\ &\le \frac{G+T}{\frac{G+(g+T)}{2}} = \frac{2(G+T)}{G+g+T} = 2 - \frac{2g}{G+g+T} \\ &< 2, \end{aligned}$$

其中，严格不等式利用了 $g > 0$。这证明了所述界限及其严格性。 □

> **观察 1：客户端推测限制**
>
> 客户端推测式工具调用的加速幅度受限于 $2\times$：我们只能隐藏两个主要阶段（生成或工具执行）中的一个。对于良好的推测模型（快速且准确），当工具延迟 $T$ 约等于主模型生成时间$G$时，收益最大。

## 3.2 引擎侧推测式工具调用

接下来，我们介绍如何将推测式工具调用方法从客户端移至引擎。首先，我们在算法 3 中介绍带工具调用的非推测式推理引擎的工作方式。引擎会不断将输入队列中的新请求加入活动批次。对于批次中的每个序列，引擎首先对提示进行编码以生成首个令牌（预填充），然后生成后续令牌（解码）。生成停止令牌（序列结束或工具调用令牌）后，引擎便会停止生成。如果停止令牌是工具调用令牌，引擎会将工具调用返回给客户端，并将该序列从活动批次中移除。除非启用了前缀缓存，否则其 KV 缓存会被丢弃。

引擎端的推测式方法通过推理引擎维护的工具缓存来实现。与前一种方法类似，客户端会启动多个推测模型实例，并异步启动工具。不过，它不会存储未来对象，而是等待每次工具执行完成，然后将结果提交给引擎，并以规范化键（工具名称 + 规范化参数）和请求 ID 作为索引。在引擎端，我们基于工具缓存引入了两项优化。当检测到某个序列中出现工具起始令牌（表示模型即将生成工具调用的首个令牌）时，引擎会使用请求 ID 和工具名称进行缓存查找。如果找到匹配项，就会将最近一次匹配的工具调用的参数令牌作为草稿注入

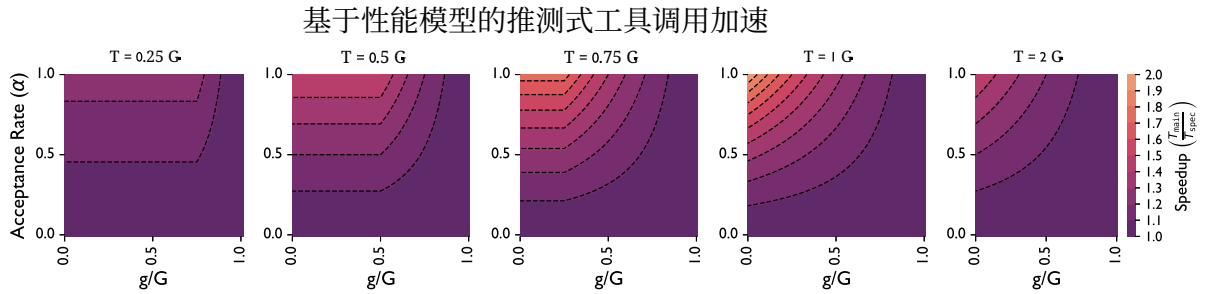



图3：客户端算法在不同接受率、生成时间和工具调用时间下的推测式工具调用加速分布。 α 是推测模型$\mathcal{S}$的接受率，$g/G$ 是推测模型与主模型的生成时间之比，而$T$ 是工具调用时间。$T \approx G$、 $\alpha > 0.5$和$g/G < 0.5$ 的取值会带来更高的加速效果，但最终加速比受限于 2×。

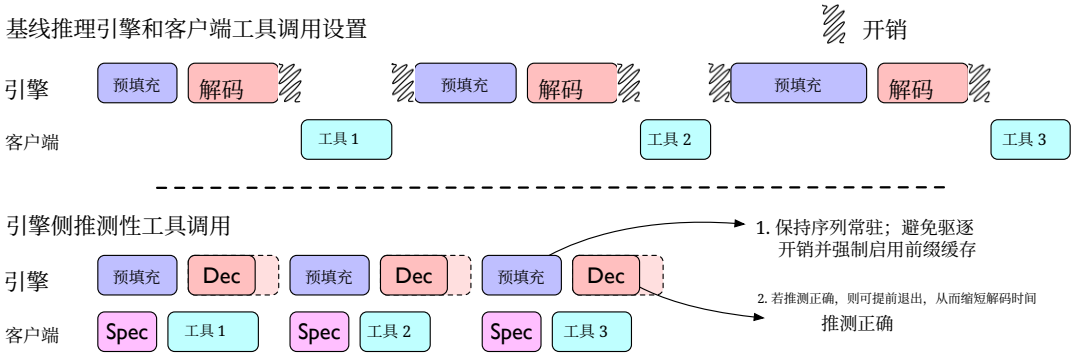
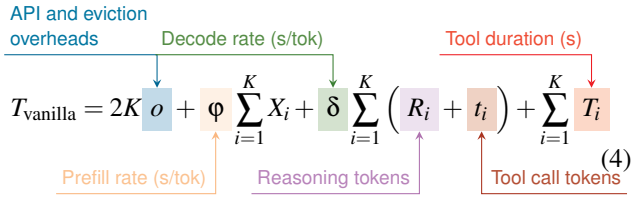



图4：我们提出的面向智能体的引擎侧推测式工具调用算法概览。首先，我们的方法会推测工具调用，并使其执行与生成过程重叠。其次，如果目标模型完成解码前，推测的工具调用已就绪，那么我们就能通过一次前向传播验证该工具调用，并提前退出解码。最后，通过将推测的工具输出发送至服务器，我们可以避免预填充时间持续增长，并消除因将提示从批次中移除而产生的开销。

词元，并通过标准的推测采样进行验证。这样既能确保正确性，也有望避免生成工具调用参数（提前退出解码）。检测到 toolend 标记（表示完整工具调用已生成的令牌）时，引擎会查找完整键。若缓存命中，就会追加工具结果并更新 KV 缓存，使解码能够继续且无需驱逐。否则，引擎会退回基线行为，并将工具调用发送给客户端。图 4 概述了这种方法。该方法下客户端和引擎的行为分别见算法 4和5。

### 3.2.1 引擎侧推测分析

在传统的工具调用场景中，用户接收工具参数、运行工具，然后再次调用 API。从推理引擎的角度来看，我们可以按照公式（4）对 $K$ 个连续智能体轮次的运行时进行建模。这涵盖了每轮的预填充阶段和解码阶段，其中 $X_i$ 个词元会被预填充，$R_i$ 个推理令牌会被解码，并且 $t_i$ 个工具调用令牌会被解码。然后调用工具，耗时 $T_i$ 秒；之后，工具调用及其输出（$t_i + to,\ i$ 个词元）会被发回 API。最后，整个历史记录和新的工具调用会再次进行预填充，因此我们将它们纳入$X_i$递归项中。在这些阶段之间还存在额外开销$o$，即通过 HTTP 将提示传递给 API 并将其载入运行中的批次所需的开销。




$$T_{\text{vanilla}} = 2K\,o + \varphi \sum_{i=1}^{K} X_i + \delta \sum_{i=1}^{K} \left( R_i + t_i \right) + \sum_{i=1}^{K} T_i \tag{4}$$

使得

$$X_{i+1} = X_i + t_i + t_{o,i}, \quad X_1 = \text{initial prompt tokens}$$

在此，我们假设：(1) 与大多数商业 API 通常的做法一样，先前轮次的推理令牌不会被纳入未来轮次的预填充中

**算法 3** 基线推理引擎：批量解码与工具调用时先驱逐后补充

**要求：** 输入队列 Q、模型 $\mathcal{M}$、批大小 $K$

B ← ∅ ▷ 序列的活动批次
**当** SERVICERUNNING()**时**
  **填充** B，直到从 Q 中取出$K$ 个就绪请求
  **对于所有** $s \in$B，**执行**
    **如果** $s$.状态 =新建，**则**
      $t \leftarrow$ 预填充($\mathcal{M}$,$s$) ▷首个令牌
      $s$.状态 ←解码
    **否则如果** $s$.状态 =解码 **则**
      $t \leftarrow$ 解码步骤($\mathcal{M}$,$s$) ▷获取下一个令牌
    **结束如果**
    **如果** IsStop($t$)**则**
      发送到客户端($s$,最终)
      EVICT(B,$s$) ▷ KV 缓存可能已被丢弃
    **结束 if**
  **结束 for**
  **将**客户端发来的任何新请求/后续请求（例如，工具结果）加入 Q
**结束 while 循环**

；(2) 每轮工具调用期间不会同时产生其他输出。前一项假设通常适用于各种 API 模型，而后一项则取决于智能体框架的实现方式（直接使用工具，还是像 SWE-Agent 那样采用思考-行动模式）。尽管如此，我们的模型具有普遍适用性，因为你可以将这些额外的解码词元纳入$t_i$ 项中。

请注意，在前缀缓存的情况下，我们可以像公式（5）所示表示时间。在这里，我们无需对已填充到 KV 缓存中的词元重复进行预填充。

$$T_{\text{cached}} = 2Ko + \varphi\left(X_1 + \sum_{i=1}^{K}(t_i + t_{o,i})\right) + \delta\sum_{i=1}^{K}(R_i + t_i) + \sum_{i=1}^{K} T_i \tag{5}$$

接下来，我们考虑三项主要优化：（1）在执行工具之前对工具调用进行推测（$O1$）；（2）在解码期间通过一次前向传播验证缓存的工具调用（$O2$）；（3）对于输出已可用的工具调用，避免驱逐缓存并重复预填充（$O3$）。第一项优化（$O1$）最多可为我们节省 $\sum_{i=1}^{K} T_i$ 时间，因为它有可能掩盖工具调用时间。$O2$ 优化可能节省 $\delta\sum_{i=1}^{K}(t_i - 1)$ 时间，因为我们将生成工具调用所需的解码次数从$t_i$ 次减少到验证工具调用所需的 1 次。最后，$O3$ 优化在理想情况下可节省 $2(K-1)o$ 开销，并有效地促使前缀缓存减少预填充词元的数量。基于这些优化，我们可以用推测准确率 $\alpha$ 表示预期耗时，如方程（6）所示。

**算法 4** 具有推测式工具执行和缓存提交的客户端 必需：提示 p、聊天历史 H、工具 T、主语言模型 API 端点 E、推测式大语言模型 S、样本 λ确保：最终输出 y 1: (h,rid) ← 调用 API(E, p, H, T ) ▷启动主请求；获取请求 ID 2: 对 i ← 1 到 λ 执行 ▷ 启动推测样本 3: 派生 s $_i$ ← 调用语言模型(S, p, H, T ) 4: 如果检查工具调用(s$_i$) 则 5: τ ← 提取工具调用(s$_i$) 6: f ←异步启动工具(τ) 7: 派生 ONREADY( f ): v ←等待 f k ← 规范键(τ) ▷ 规范化：工具名称+规范化参数 SUBMITTOOLCACHE(E, rid,k,v) 8: 结束如果 9: 结束循环 10: 11: r ←等待 h ▷ 接收结果（可能是最终结果，也可能是工具调用） 12: 13: 如果检查工具调用(r) 则 ▷ 缓存未命中路径：引擎未在其缓存中找到值 14: τ⋆ ←提取工具调用(r) 15: v⋆ ←运行工具(τ⋆) 16: H′ ← H ∪{(τ⋆,v⋆)}17: r ← 调用 API(E, p,H′, T ) ▷继续正常循环 18: 当检查工具调用(r) 时循环 19: τ ← 提取工具调用(r) 20: v ← 运行工具(τ); H′ ← H′ ∪ {(τ, v)}21: r ← 调用 API(E, p,H′, T ) 22: 结束循环 23: 结束如果 24: y ← 渲染(r) 25: 返回 y

$$T^{*}_{\text{spec}} = (1-\alpha)2Ko + \varphi\left(X_1 + \sum_{i=1}^{K}(t_i + t_{o,i})\right) + \\ + \delta\left(\alpha K + \sum_{i=1}^{K} R_i + (1-\alpha)\sum_{i=1}^{K} t_i\right) + (1-\alpha)\sum_{i=1}^{K} T_i \tag{6}$$

我们可以立即看出，这等价于 $T_{\text{cached}}$ for$\alpha = 0$。当 $\alpha = 1$ 我们节省了 $2Ko + \delta(\sum_{i=1}^{K} t_i - K) + \sum_{i=1}^{K} T_i$时间，相较于前缀缓存推理，因为我们省去了 $2Ko$的开销、工具延迟，以及工具调用所需的额外解码。工具调用时长$T_i$ 小于 $\delta R_i$ 时，加速效果最佳，这样工具输出就能用于推测（$O2$）。如第 3.1.1 节所述，单轮的最佳加速效果出现在 $T_i \approx \delta R_i$ 时，因为这样我们可以屏蔽整个工具调用，并节省最多的解码步骤。然——

**算法5** 带工具缓存的推理引擎

**要求：** 输入队列 Q、模型 $\mathcal{M}$、批大小 $K$、工具缓存 C，以（请求 ID、规范键）和（请求 ID、工具名称）为键

```
B ← ∅
当 SERVICERUNNING()时
    ASYNCUPDATE(C)                                  ▷来自 SUBMITTOOLCACHE
    填充 B 从 Q 到 K
    对所有 s ∈B 执行
        如果 s.状态 =新建 则
            t ← 预填充(M,s); s.状态 ← 解码
        否则如果 s.STATUS =D解码 则
            t ← 解码步骤(M,s)
        结束如果
        如果 是否为工具调用起始(t)则
            η ← 提取工具名称(s,t)
            如果 C.HAS(rid(s) η) 则
                c ← C.获取工具调用(rid(s) η)
                t ← 推测采样(M,s,c)                  ▷ 验证工具调用令牌并更新键值缓存
            结束条件判断
        否则，如果 检查工具调用是否结束(t)则
            τ ← 提取工具调用(s,t)
            k ← C匿名KEY(τ)
            如果 C.HAS(rid(s),k) 则                  ▷缓存命中
                v ← C.获取工具结果(rid(s),k)
                t ← 预填充工具结果(M,s τ,v)          ▷ 将工具结果令牌追加到键值缓存
            结束 if
        结束 if

        如果 ISSTOP(t)那么                           ▷缓存未命中/其他停止令牌
            EMITTOCLIENT(s，最终)
            从批次中移除(B,s)
        结束 if
    结束循环
    摄取 新的客户端请求到 Q
结束 while
```

而，任何满足 $T_i < \delta R_i$ 的值都应能节省运行时开销。这是一个理想的发现，因为智能体中常用的许多最先进推理模型，其推理轨迹通常持续数秒到数分钟，这意味着我们可以在这段时间内推测并执行许多常用工具。

# 4 实验设置

定义了这两种算法后，我们现在详细介绍它们的实现方式，以及为测试其有效性而开展的实验。

## 4.1 算法实现

客户端算法基于 Ope- nAI API 实现 [2]，并使用其异步客户端。每当向主模型发送聊天补全或响应时，我们也会异步向推测式 API 端点发送聊天补全/响应。如果使用多个推测样本，则会发送多个异步请求。推测模型的 API 调用返回后，我们会异步启动推测出的工具，并将其未来对象放入工具缓存中，缓存以函数名和参数为键。此实现是一个自定义 Python 接口，使用了 Python 内置的 asyncio 库。主模型返回后，如果它调用了工具，我们就会检查工具缓存。如果缓存命中，我们便等待未来对象，并将其结果用作工具输出。否则，我们会像往常一样调用该工具。

对于引擎端算法，我们修改了一个流行的 vLLM 框架定制分支 [7]，以实现我们的算法。工具缓存端点采用标准 HTTP POST API 端点实现（参见后文的 API 规格）。与客户端设置类似，一个或多个推测模型端点与主模型异步并行运行（*O1*）。推测工具调用的结果会被发送到工具缓存端点。基于 vLLM 的推测解码基础设施，我们实现了一个工具提议器，负责检测生成的令牌流中的工具调用边界。检测到工具调用开始时，提议器会起草与缓存中名称相同的工具最新条目相匹配的工具调用令牌。随后使用标准推测采样 [8],验证这些草拟令牌，以确保正确性（*O2*）。检测到工具调用结束时，提议器会使用工具名称及其参数查询缓存。若缓存命中，则起草相应的工具输出令牌；在这种情况下，所有草拟令牌都可以直接接受，因为名称和参数均完全匹配（*O3*）。如果查询结果为缓存未命中，则停止生成，并由引擎将工具调用返回给客户端，以便正常执行。为使引擎端算法适用于多轮场景，我们确保启用流式传输，这样客户端便能知道何时开始推测新的工具调用。

我们使用的自定义 API 端点详述如下。现有商业 API 端点提供商只需采用这一简单接口，即可向用户提供这项优化。提供商有动力使用缓存工具输出 API，因为它可以缩短推理时间并提高吞吐量。终端用户则可以受益于更低的成本和更快的响应速度。不过，成本降低并非必然会自动实现；许多提供商已经会对命中前缀缓存的提示提供折扣，而我们提出的优化也能很好地融入现有定价模式。

**API：POST /缓存工具输出/{response_id}**

**描述。** 缓存推理引擎可利用的工具输出，以便在工具调用后让序列继续驻留。每个条目以名称和（可选的）参数为键，并与模型在此响应中发出的未来工具调用进行匹配。

**请求正文。** 包含以下字段的 JSON 数组对象：

| 字段 | 描述 |
|---|---|
| name | （字符串，必填）所使用的唯一工具名称作为缓存键的一部分。 |
| 参数 | （对象，可选）规范化的工具参数包含在缓存键中。名称相同且参数等效的调用共享输出。 |
| 输出 | （任意类型，必填）要被序列化工具输出缓存并在模型生成匹配的工具调用时重用。 |
| keep_alive | （数字，可选）可选的生存时间用于缓存条目（例如，以秒为单位）。 |

**响应。**

200 OK { "cached": <int> //接受的条目数量 }

## 4.2 测试环境

我们在一台计算节点上运行实现和实验，该节点配备四块 80GB NVIDIA A100 GPU，由一颗 AMD EPYC 7763 CPU 控制；每个 CPU 有 64 个物理核心，主频为 2.45 GHz。每对 GPU 通过第三代 NVLink 连接，每个方向的链路带宽为 25 GB/s；每块 GPU 则通过 PCIe 4.0 连接到 CPU。我们使用基于 SUSE Enterprise Linux Server 15 SP5 的一个变体，并运行 Python 3.12 和 CUDA 12.9。在实验中，我们运行两台 vLLM 推理服务器：一台用于主模型，另一台用于推测模型。主模型 $\mathcal{M}$ 使用一块 A100 提供服务，而推测服务器则在其余三块 A100 上托管推测模型 $\mathcal{S}$ 的三个数据并行实例。

## 4.3 实验

为了评估我们的推理优化，我们使用 BFCL 项目中的提示和工具数据集 [11]：BFCL 是一个用于评估语言模型工具调用能力的基准。BFCL 定义了大量涵盖各种任务的工具。作为一个排行榜，BFCL 展示了不同语言模型正确调用工具的表现对比。这为选择小型且准确的推测模型提供了良好起点。

根据 BFCL 排行榜，我们选定 xLAM 模型 [21] 作为推测模型的理想候选者。它们是经过微调的 Llama 1B、3B 和 8B 模型，使用工具数据集进行微调，以预测工具调用。在测试过程中，我们发现 8B 模型能够正确预测 BFCL 中约 80% 的工具调用，这与其论文中呈现的结果一致。由于这些模型规模较小且能够准确预测工具，我们将其用作推测式工具调用模型。

BFCL 定义了输入提示与每个提示可调用的工具集合，但未提供具体的工具实现（仅提供规范）。我们的目标是研究智能体使用工具时在推理和系统层面的行为。因此，我们将工具视为具有指定接口和延迟的黑盒。对于每个 BFCL 工具，我们使用语言模型生成的响应和人工编写的响应，预先计算一个具有代表性的输出，并将这些输出存储在内存缓存中。在实验期间，每当模型调用工具时，系统都会返回缓存的工具输出，并通过手动配置的工具延迟来控制其运行时间。

关键的是，我们的算法和测量只取决于何时调用工具以及运行需要多长时间，而不取决于工具输出本身的内容。以这种方式预先计算输出，既保留了模型的计算行为，又使我们能够系统地控制工具延迟。作为一致性检查，我们手动检查了实验中的模型生成结果，发现它们合理且符合基础模型的典型输出。我们还使用简单的手动实现工具进行了一项小规模实验，实际运行这些工具后发现，模型行为在定性上没有变化，性能结果在定量上也没有变化，这支持了我们的工具设置适用于系统级性能评估这一结论。

为了将 BFCL 转化为智能体工作负载，我们将每组提示和工具集视为异步智能体的一项独立任务。我们运行 $M$ 个并发智能体，每个智能体配有 $M \in \{1, 8, 32\}$。每个智能体都会采样一个提示及其关联的工具。它会持续循环：按需调用工具并生成文本，直到任务完成并给出最终响应。随后，它会接收另一组提示和工具，并不断重复这一过程。每项实验的运行时间足够长，确保每个智能体都完成 32 项此类任务；我们记录整个运行期间的性能指标，详见第 5 节。实验中，我们使用 gpt-oss-120b [10]模型作为主要智能体，因为它是最先进的推理模型，可能会在智能体中得到广泛应用。

我们系统地遍历主要实验超参数，以研究它们的影响。对于工具行为，我们从两种正态随机分布中抽取工具延迟样本：均值为$\in \{0, 0.1, \ldots, 0.5\}$的短延迟 分布，以及均值为$\in \{0, 0.5, \ldots, 3\}$的长延迟 分布。对于推测，我们改变每轮的推测样本数（1、3、5、7和 9），以及推测模型（xLAM 1B、3B 和 8B）。对于这一搜索空间中的每个点，我们在三种推理配置下运行工作负载：

标准推理（不进行推测）、客户端推测和引擎端推测。每项实验均运行五次，以计入变化和系统噪声。此外，我们还使用客户端算法以及商业模型 GPT-5 和 GPT-5-nano 运行部分实验，以研究这种方法对商业模型的适用效果。

## 5 评估指标

设置好工具调用代理实验后，我们需要衡量推测带来的收益，以便比较不同方法。墙钟时间等指标较难使用，因为某些智能体可能会针对提示生成大量响应文本，而另一些智能体给出的响应则很短。即使提示相同，采样输出的差异也可能导致这种情况。例如，设智能体 A 和 B 都调用工具、接收工具输出，然后生成响应。智能体 A 正确推测了工具，而智能体 B 没有。如果智能体 A 针对工具输出生成了 2000 个词元的响应，而智能体 B 生成了 50 个词元的响应，那么尽管智能体 A 从推测中获得了更大的收益，其墙钟时间仍会更长。因此，我们使用吞吐量和节省时间百分比 作为比较指标。

**吞吐量** 的计算方式是每秒生成的词元数。我们针对每个智能体计算该指标，并报告平均吞吐量。

**节省时间** 是相对于非推测基线而言的时间缩短百分比。对于相同的提示，我们分别记录智能体在启用和未启用推测时从开始到结束所用的时间。设$T_{\text{base}}$ 表示基线智能体（不使用推测）所用的时间，$T_{\text{spec}}$ 表示采用我们的推测算法时相同智能体配置所用的时间。我们定义

$$\text{Time Saved} = 100 \times \frac{T_{\text{base}} - T_{\text{spec}}}{T_{\text{base}}} \%.$$

我们按每个智能体计算该指标，并报告所有异步代理的平均值。由于两次运行使用相同的提示、工具和工具输出，因此该指标反映了推测所带来的推理时间缩短。

除了吞吐量和节省时间外，我们还分析客户端算法的成本。由于该算法完全可以通过 API 实现，因此也可用于商业模型。我们根据 OpenAI API 返回的令牌使用数据，以及 OpenAI 公布的每令牌价格来计算**成本**（成本依据 2025年 11月 的定价）。报告的成本是相较于仅使用主模型，进行推测所产生的额外成本。由于智能体的数量可能有所不同

无论按轮次还是词元计算，我们均报告每 100 个智能体轮次的成本。

## 6 结果

本节展示了我们对所提出的两种算法进行评估后得到的结果。

### 6.1 客户端推测

图 5 展示了客户端算法中不同接受率下各种推测模型的吞吐量，这些结果基于部分 *BFCL* 提示。我们在更容易预测的一组工具上展示这些结果，以便观察数值 $\alpha > 0.8$，从而验证公式（2）。在后续实验中，我们观察到 $\alpha \approx 0.8$ 对于 xLAM-2-8B。图中展示的是平均工具延迟为 1.5 秒时的运行结果，但其他工具延迟下的趋势也类似。点状标注表示推测模型每轮推测的工具调用数量（$\lambda$ 见算法 2）（例如 1×， 3×， .....）。

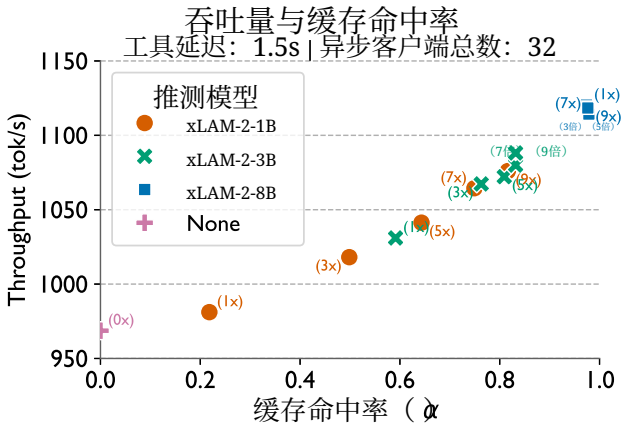



图 5：不同推测模型及其各自工具缓存命中率对应的吞吐量提升。标注的数值表示每次生成的推测次数。随着工具预测率提高，吞吐量也明显提升。

我们发现，性能模型所揭示的趋势依然成立（见第 3.1.1节）；随着推测率提高，推理吞吐量也随之提升。8B 模型的工具预测准确率远高于较小的模型，不过，我们也发现，增加样本数可以提高较小模型的准确率。xLAM-2-1B 的命中率提高了近 60%，使用 9× 个样本时，相比于使用 1× 个样本时更高。这种灵活性很重要，因为受内存限制，我们可能希望运行较小的推测模型；使用较小模型并增加样本数，可以获得接近较大推测模型的性能。

图 6 展示了使用客户端推测算法运行各个智能体时平均节省的时间百分比。即使对于耗时较短的工具（约 0.5 秒），我们仍能观察到显著的提升：通过对工具调用进行推测，总耗时最多可节省 6%。随着工具平均耗时接近 gpt-oss-120b 的平均生成时间，效果也会提升；当工具平均耗时约为 2.0–2.5 秒时，收益最大，节省时间最多可达 21%。随着工具耗时占运行时的比例不断增加，节省的时间百分比会逐渐

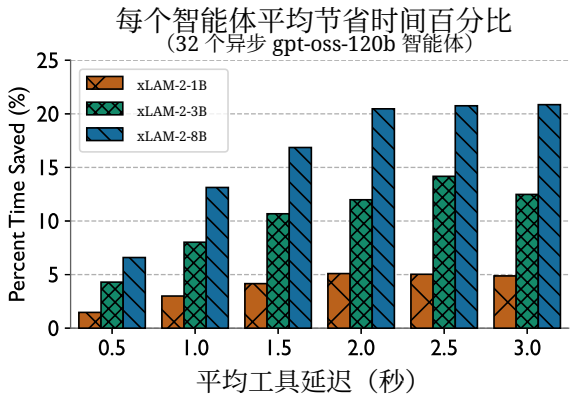



图 6：使用不同推测模型并采用平均工具延迟时，各次运行平均节省的时间百分比。此处展示每次生成进行一次 1×推测的结果。当平均工具延迟为 0.5 至 3 秒时，客户端推测可使每个智能体的端到端时间节省 6% 至 21%。

下降，而对于耗时较长的工具，推测只能带来少量节省。这一现象进一步印证了第 3.1.1 节中的性能模型：当工具调用耗时接近主模型的生成时间时，加速效果最高。

**观察2：客户端推测的收益**

无需对推理引擎或 API 做任何额外改动，智能体框架通过提前推测执行工具，可节省 6–21% 的时间。节省幅度取决于模型生成时间与平均工具调用时长的比值。

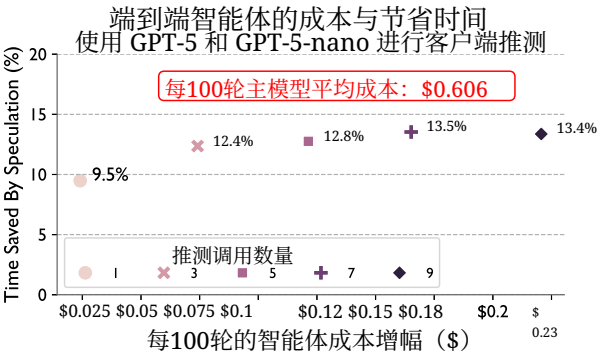



图7：与 GPT-5 智能体一同运行客户端推测时，额外增加的成本（使用 GPT-5-nano）。图中显示 100 个智能体轮次的成本。只需增加 4% 的价格，智能体就能节省近 10% 的时间。

我们还可以在只能访问 API 的情况下，为商业模型运行客户端推测算法。图 7展示了使用推测所节省的时间百分比，以及进行推测所增加的成本。结果针对单个智能体“轮次”（即一次 API 调用后接一次工具调用）给出。我们发现，进行一次推测即可节省近 10% 的时间，而只需额外进行一次 API 调用，其成本比主 API 调用低一个数量级。进行九次推测会略微增加节省的时间，但成本约为主生成成本的 25%–30%。

## 6.2 引擎端推测

图 8 展示了引擎端算法节省的时间百分比。我们还展示了平均工具延迟为 $\{0.1, 0.2, 0.3, 0.4, 0.5\}$ 时的耗时，以突出引擎端推测最能发挥作用的场景。vLLM 的推测解码基础设施是我们实现方案的基础，目前仍处于实验性开发阶段 [1]，而且我们发现，当批大小大于 1 时，其规格解码实现的开销很高。因此，我们仅展示一个异步智能体时的引擎端结果。请注意，只有一个异步智能体时，由于请求调度器和批次较小，vLLM 的生成延迟会低得多。我们的观察与其他研究的发现一致 [16]。由于可用于与工具调用重叠的生成时间更少，因此潜在收益也更低；与图6相比，节省的时间百分比也更低。

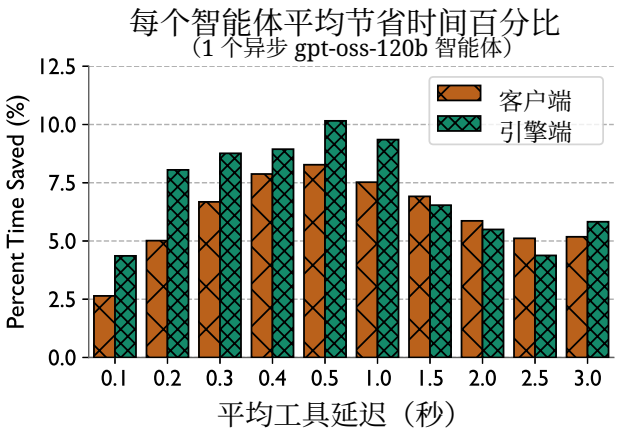



图 8：不同平均工具延迟下，客户端与引擎端算法的平均节省时间百分比对比。结果基于一个异步客户端，并使用 xLAM-2-8B 进行推测。当工具在主模型完成推理之前执行完毕时，相比客户端方法，我们可以多节省 2–3% 的时间。

**观察 3：引擎端推测的收益**

从客户端的角度来看，当工具输出预先可用时，将推测生成的工具输出 POST 到推理服务器，以使序列驻留在服务器上，还可额外节省 2–3% 的时间。当工具延迟低于主模型$\mathcal{M}$的推理时间时，收益最大。这一条件非常理想，因为许多常见的智能体工具耗时都在 0–1 秒范围内（例如 ls、网络搜索、读取文件等）。

与客户端算法相比，引擎端算法的时间节省增加了 2–3%。这是在客户端推测已经带来较高节省的基础上取得的。工具调用耗时在 0 到 1 秒之间时，效果最佳，因为这些调用会在推理阶段结束前完成，并将结果发送到推理服务器。耗时更长的工具调用相比客户端推测没有任何收益，因为在解码准备就绪前，其结果不会发送到推理服务器。

## 7 相关工作

已有多项研究探讨了如何利用推测加速语言模型推理。Leviathan 等人 [8] 提出了推测解码，通过较小的语言模型为较大的语言模型预测词元。虽然这与我们的方法密切相关，但传统的推测解码并不适用于工具调用推测执行。正如我们的分析所示，工具推测执行带来的主要加速来自于工具执行与生成的重叠。在传统的推测解码中，草稿模型通常只比主模型领先 10 个词元，这意味着我们最多只能让工具调用与 10 个解码步骤重叠。在工具调用场景中，越早、越多次地推测执行工具越好。其他研究 [3]还探讨了生成推测的其他变体，例如采样中的推测，但它们也存在与推测解码相同的缺点：在生成过程中，需要尽早推测工具调用。

其他提升智能体工具使用性能的方法，尝试提高小型模型预测应使用何种工具的准确率。通过使用较小的语言模型来执行行动，我们可以减少推理时间。Schick 等人 [14] 和 Patil 等人 [11] 通过在大规模工具调用交互数据集上进行微调来实现这一目标。SWE-Agent 等工作 [19] 通过为智能体提供最佳粒度 的工具，实现了更好、更快的智能体生成。通过为其提供正确的工具，这些方法减少了智能体需要进行的工具调用次数，并最终缩短了智能体执行任务所需的总体时间。

## 8 结论

在本文中，我们展示了工具的推测执行是减少使用工具的智能体推理开销的有效方法。我们介绍了两种互补技术：一种是客户端推测式工具调用算法，可与未经修改的黑盒 API 配合使用；另一种是引擎侧算法，通过验证工具调用并直接从工具缓存中摄取工具输出，使提示常驻于推理服务器中。借助解析模型，我们给出了工具调用能够发挥最大作用的条件；我们强调了工具延迟、推测模型准确率和解码成本在决定实践中可实现的加速幅度方面所发挥的作用。我们的实验表明，这些方法使每个智能体轮次的时间节省达到两位数百分比，并将吞吐量提高每秒数百个词元。推测式工具有望快速加速智能体在工具密集型任务中的使用；然而，要高效地正确实现推测式工具算法，需要谨慎处理；本工作提供了构建有效推测式工具调用智能体所需的洞见和基础构件。

## 致谢

本工作由劳伦斯利弗莫尔国家实验室代表美国能源部（DOE）在合同 DE-AC52-07NA27344（LLNL-CONF-2014336-DRAFT）项下完成。本材料基于美国能源部科学办公室先进科学计算研究计划通过征集公告 DE-FOA-0003264《推进人工智能用于科学研究》所支持的工作。本研究使用了国家能源研究科学计算中心（NERSC）的资源。NERSC 是美国能源部科学办公室用户设施，依据合同编号 DE-AC02-05CH11231 运营，并使用 NERSC 授予的 ALCC-ERCAP0034775 额度。

## 参考资料

[1] 推测解码 - vLLM。https://docs.vllm.ai/en/v0.12.0/features/spec_decode/ 访问日期：2025年12月11日。[2] openai/openai-python，2025年12月。原始日期：2020-10-25T23:23:54Z。[3] Charlie Chen、Sebastian Borgeaud、Geoffrey Irving、Jean-Baptiste Lespiau、Laurent Sifre 和 John Jumper。通过推测采样加速大语言模型解码。 *arXiv* 预印本 *arXiv:2302.01318*，2023年。[4] Mingyang Chen、Haoze Sun、Tianpeng Li、Fan Yang、Hao Liang、Keer Lu、Bin Cui、Wentao Zhang、Zenan

Zhou 和 Weipeng Chen。通过组合式指令微调促进大语言模型的多轮函数调用。*arXiv* 预印本 *arXiv:2410.12952*，2024年。

[5] Bin Gao、Zhuomin He、Puru Sharma、Qingxuan Kang、Djordje Jevdjic、Junbo Deng、Xingkun Yang、Zhou Yu 和 Pengfei Zuo。{面向多轮对话的} 高性价比{大语言模型服务：CachedAttention}。载于*2024*年 *USENIX* 年度技术会议（*USENIX ATC 24*），第 111–126 页，2024年。

[6] Yunhai Hu、Zining Liu、Zhenyuan Dong、Tianfan Peng、Bradley McDanel 和 Sai Qian Zhang。《推测解码及其他：技术深度综述》，2025年10月。arXiv:2502.19732 [计算机科学]。

[7] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Sheng、Lianmin Zheng、Cody Hao Yu、Joseph E. Gonzalez、Hao Zhang 和 Ion Stoica。采用分页注意力的大语言模型服务高效内存管理。载于*ACM SIGOPS* 第 *29* 届操作系统原理研讨会论文集，2023年。

[8] Yaniv Leviathan、Matan Kalman 和 Yossi Matias。通过推测解码实现 Transformer 的快速推理。载于第*40*届国际机器学习会议论文集，第202卷，*ICML'23*，第19274–19286页，美国夏威夷檀香山，2023年7月。JMLR.org。

[9] Junwei Liu、Kaixin Wang、Yixuan Chen、Xin Peng、Zhenpeng Chen、Lingming Zhang 和 Yiling Lou。基于大语言模型的智能体在软件工程中的应用：综述。*arXiv* 预印本 *arXiv:2409.02977*，2024年。

[10] OpenAI、Sandhini Agarwal、Lama Ahmad、Jason Ai、Sam Altman、Andy Applebaum、Edwin Arbus、Rahul K. Arora、Yu Bai、Bowen Baker、Haiming Bao、Boaz Barak、Ally Bennett、Tyler Bertao、Nivedita Brett、Eugene Brevdo、Greg Brockman、Sebastien Bubeck、Che Chang、Kai Chen、Mark Chen、Enoch Cheung、Aidan Clark、Dan Cook、Marat Dukhan、Casey Dvorak、Kevin Fives、Vlad Fomenko、Timur Garipov、Kristian Georgiev、Mia Glaese、Tarun Gogineni、Adam Goucher、Lukas Gross、Katia Gil Guzman、John Hallman、Jackie Hehir、Jo- hannes Heidecke、Alec Helyar、Haitang Hu、Romain Huet、Jacob Huh、Saachi Jain、Zach Johnson、Chris Koch、Irina Kofman、Dominik Kundel、Jason Kwon、Volodymyr Kyrylov、Elaine Ya Le、Guillaume Leclerc、James Park Lennon、Scott Lessans、Mario Lezcano-Casado、Yuanzhi Li、Zhuohan Li、Ji Lin、Jordan Liss、Lily、Liu、Jiancheng Liu、Kevin Lu、Chris Lu、Zo- ran Martinovic、Lindsay McCallum、Josh McGrath、Scott McKinney、Aidan McLaughlin、Song Mei、Steve Mostovoy、Tong Mu、Gideon Myles、Alexander Neitz、Alex Nichol、Jakub Pachocki、Alex Paino、Dana Palmie、Ashley Pantuliano、Giambattista Parascandolo、Jong-soo Park、Leher Pathak、Carolina Paz、Ludovic Peran、Dmitry Pimenov、Michelle Pokrass、Elizabeth Proehl、Huida Qiu、Gaby Raila、Filippo Raso、Hongyu Ren、Kimmy Richardson、David Robinson、Bob Rotsted、Hadi Salman、Suvansh Sanjeev、Max Schwarzer、D. Sculley、Harshit Sikchi、Kendal Simon、Karan Singhal、Yang Song、Dane Stuckey、Zhiqing Sun、Philippe Tillet、Sam Toizer、Foivos Tsimpourlas、Nikhil Vyas、Eric Wallace、Xin Wang、Miles Wang、Olivia Watkins、Kevin Weil、Amy Wendling、Kevin Whinnery、Cedric Whitney、Han- nah Wong、Lin Yang、Yu Yang、Michihiro Yasunaga、Kristen Ying、Wojciech Zaremba、Wenting Zhan、Cyril Zhang、Brian Zhang、Eddie Zhang和Shengjia Zhao。gpt-oss-120b 和 gpt-oss-20b 模型卡片，2025年8月。arXiv:2508.10925 [计算机科学]。

[11] Shishir G. Patil、Tianjun Zhang、Xin Wang 和 Joseph E. Gonzalez。Gorilla：与海量API连接的大语言模型，2023年5月。arXiv:2305.15334 [计算机科学].[12] Yujia Qin、Shengding Hu、Yankai Lin、Weize Chen、Ning Ding、Ganqu Cui、Zheni Zeng、Xuanhe Zhou、Yufei Huang、Chaojun Xiao、Chi Han、Yi Ren Fung、Yusheng Su、Huadong Wang、Cheng Qian、Runchu Tian、Kunlun Zhu、Shihao Liang、Xingyu Shen、Bokai Xu、Zhen Zhang、Yining Ye、Bowen Li、Ziwei Tang、Jing Yi、Yuzhang Zhu、Zhenning Dai、Lan Yan、Xin Cong、Yaxi Lu、Weilin Zhao、Yuxiang Huang、Junxi Yan、Xu Han、Xian Sun、Dahai Li、Jason Phang、Cheng Yang、Tong-shuang Wu、Heng Ji、Guoliang Li、Zhiyuan Liu 和 Maosong Sun。基础模型的工具学习。*ACM* 计算综述，57(4)，2024年12月。[13] Changle Qu、Sunhao Dai、Xiaochi Wei、Hengyi Cai、Shuaiqiang Wang、Dawei Yin、Jun Xu 和 Ji-Rong Wen。大语言模型的工具学习：综述。《计算机科学前沿》，19(8):198343，2025年。[14] Timo Schick、Jane Dwivedi-Yu、Roberto Dessì、Roberta Raileanu、Maria Lomeli、Eric Hambro、Luke Zettle-moyer、Nicola Cancedda 和 Thomas Scialom。Toolformer：语言模型可以自我学习使用工具。神经信息处理系统进展，36:68539–68551，2023年。[15] Simranjit Singh、Andreas Karatzas、Michael Fore、Iraklis Anagnostopoulos 和 Dimitrios Stamoulis。用于融合并行函数调用的语言模型工具编译器。*arXiv* 预印本 *arXiv:2405.17438*，2024年。

[16]Prajwal Singhania、Siddharth Singh、Lannie Dalton Hough、Akarsh Srivastava、Harshitha Menon、Charles Fredrick Jekel 和 Abhinav Bhatele。《单节点之外的大语言模型推理：通过快速全归约通信解决瓶颈并采取缓解措施》，2025年11月。arXiv:2511.09557 [cs]。

[17] Linke Song、Zixuan Pang、Wenhao Wang、Zihao Wang、XiaoFeng Wang、Hongbo Chen、Wei Song、Yier Jin、Dan Meng 和 Rui Hou。《早起的鸟儿抓住泄漏：揭示大语言模型服务系统中的时间侧信道》。*IEEE Transactions on Information Forensics and Security*，2025年。

[18] vLLM。自动前缀缓存。https://docs.vllm.ai/en/stable/design/prefix_缓存/。

[19] John Yang、Carlos E Jimenez、Alexander Wettig、Kilian Lieret、Shunyu Yao、Karthik Narasimhan 和 Ofir Press。SWE-Agent：智能体—计算机接口助力自动化软件工程。神经信息处理系统进展，37:50528–50652，2024年。

[20] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Karthik R Narasimhan 和 Yuan Cao。ReAct：在语言模型中协同推理与行动。发表于第十一届国际学习表征会议，2022年。

[21] Jianguo Zhang、Tian Lan、Ming Zhu、Zuxin Liu、Thai Hoang、Shirley Kokane、Weiran Yao、Juntao Tan、Ak- shara Prabhakar、Haolin Chen、Zhiwei Liu、Yihao Feng、Tulika Awalgaonkar、Rithesh Murthy、Eric Hu、Zeyuan Chen、Ran Xu、Juan Carlos Niebles、Shelby Heinecke、Huan Wang、Silvio Savarese 和 Caiming Xiong。xLAM：赋能人工智能智能体系统的大型行动模型系列，2024年9月。arXiv:2409.03215[计算机科学]。

[22]Lianmin Zheng、Liangsheng Yin、Zhiqiang Xie、Chuyue Livia Sun、Jeff Huang、Cody Hao Yu、Shiyi Cao、Christos Kozyrakis、Ion Stoica、Joseph E Gonzalez 等人。Sglang：结构化语言模型程序的高效执行。神经信息处理系统进展，37:62557–62583，2024年。