arXiv:2601.00397v1 [cs.DC] 1 Jan 2026

REVATI：透明的无 GPU 时间扭曲仿真技术，适用于大语言模型服务 ng

Amey Agrawal*、Mayank Yadav*、Sukrit Kumar†、Anirudha Agrawal†、Garv Ghai†、Souradeep Bera、Elton Pinto、Sirish Gambhira、Mohammad Adain、Kasra Sohrab、Chus Antonanzas 和 Alexey Tumanov

佐治亚理工学院

摘要

高效部署大语言模型需要测试数百种服务配置，但在 GPU 集群上逐一评估每种配置需要数小时，花费数千美元。离散事件模拟器速度更快、成本更低，但需要重新实现服务系统的控制逻辑——随着服务框架不断演进，这一负担也会越来越重。

我们提出了 REVATI，这是一种时间扭曲仿真器，通过直接执行真实的服务系统代码，以近似仿真的速度实现性能建模。该系统拦截 CUDA API 调用以虚拟化设备管理，使服务框架无需物理 GPU 即可运行。它不执行 GPU 内核，而是执行时间跳跃——根据预测的内核执行时长快进虚拟时间。我们提出了一种协调协议，可在分布式进程间同步这些跳跃，同时保持因果关系。在 vLLM 和 SGLang 上，REVATI 在多种模型和并行配置下的预测误差为 <5%，运行速度比真实 GPU 执行快 5-17× 倍。

1 引言

大语言模型推理日益成为人工智能成本的主要来源。为了管理这些成本，运营方需要调整大量配置参数，包括各种并行策略、缓存策略和调度算法 [4, 31]。随着模型、工作负载和硬件平台不断演进，这些参数都必须针对每种情况进行调整。通过反复试错和端到端评估来寻找最优部署配置，既耗时又昂贵，难以承受。即使是在 GPU 集群上评估单个配置，也可能需要数小时，并耗费数千美元的计算资源 [4]。这既制约了生产部署，也阻碍了系统研究。

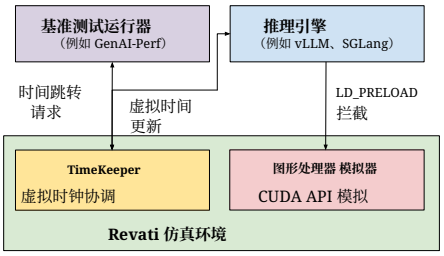


图 1：离散事件模拟器必须重新实现服务系统的整个控制流和调度逻辑，因此，大语言模型推理的快速发展使这些模拟器始终落后于最新进展。REVATI彻底解决了这一问题：它在仿真环境中直接运行服务系统。

当前方法的局限。为应对这一挑战，社区开发了离散事件模拟器 [4, 12, 20]，能够以较低成本模拟服务性能。模拟器具有显著优势：其运行速度比实际执行快几个数量级，而且便于修改，可用于原型开发新策略和优化方案。然而，模拟器需要重新实现所有核心系统组件，包括调度器、内存分配器和请求分发器等。当系统较为简单时，这种方法行之有效：在 Vidur 中，实现 vLLM 最初的连续批处理调度器只需不到 150 行代码 [4]。然而，如今的大语言模型引擎包含预填充与解码分离 [10, 23–25, 32] 、分布式前缀缓存 [10, 23] 等复杂功能，其实现代码多达数万行。此外，推理服务系统每年有数百位贡献者提交数千次代码变更，发展迅速，导致模拟器始终落后，维护负担也不断加重。

关键洞见。 服务系统的大部分执行时间都在等待 GPU 计算，而不是 进行控制决策。CPU 控制平面（调度、

*† 表示贡献相同。Revati 是一位神话人物，出现在最早的时间膨胀虚构故事之一中。

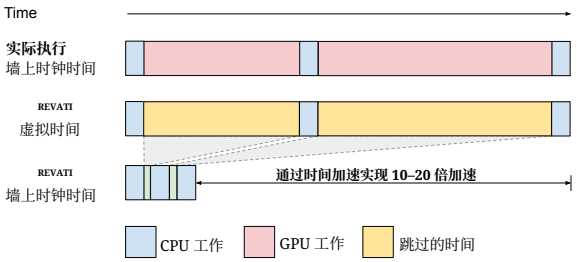


图 2：**时间加速仿真**。 大语言模型推理引擎力求最大限度地提高图形处理器计算量（红色），同时尽可能减少中央处理器开销（蓝色）。REVATI 利用这一特性，在挂钟时间中跳过图形处理器操作，同时通过虚拟时间语义保留应用程序的因果语义。这样，我们便能在不牺牲保真度的情况下，实现数量级的加速。

批次处理、内存管理）在微秒级完成，而 GPU 操作需要数十毫秒，占执行时间的 90% 以上。此外，控制流逻辑在很大程度上与 GPU 计算的输出相互解耦。这种分离带来了一种彻底不同的方案：通过虚拟时间推进，跳过 GPU 等待，同时运行真实的服务系统。我们将其称为时间扭曲仿真。

我们的方法。 REVATI 运行 vLLM 或 SGLang 等实际服务框架，并以虚拟时间跳跃替代图形处理器操作。当工作进程准备执行一个批次时，会询问：“这在 H100 上需要多长时间？”根据预测的时长，它会向协调所有进程虚拟时间的中央*TimeKeeper* 请求一次时间跳转 。*TimeKeeper* 确保所有进程同步推进，同时保留因果关系：如果工作进程 A 需要 15 毫秒，而工作进程 B 需要 25 毫秒，系统会先推进 15 毫秒。CUDA 仿真层会透明地拦截图形处理器调用，向服务框架呈现虚拟设备，同时绕过实际执行。这样，我们就能在加速的虚拟时间中直接运行服务系统的代码，而无需实际图形处理器。

通过运行未经修改的控制平面，REVATI 无需手动建模即可自动捕获复杂的系统行为。自适应批处理、多级前缀缓存管理、PD 解耦等复杂功能均可开箱即用。REVATI 具有极低的维护开销，将服务系统接入所需的修改少于 25 行。这种速度快、成本效益高且易于采用的组合，有助于快速探索最优部署配置，并加速最先进的大语言模型推理技术的开发与采用。

	REVATI	离散事件模拟器					
		VD	LS1	AP	FT	TS	LS2
系统属性							
直接系统仿真	✓	×	×	×	×	×	×
与服务系统无关	✓	×	×	×	×	×	×
极低的维护开销	✓	×	×	×	×	×	×
建模领域							
连续批处理	✓	✓	✓	✓	✓	✓	✓
分块预填充	✓	✓	✓	✓	✓	✓	✓
前缀缓存	✓	×	×	×	×	✓	✓
分层缓存	✓	×	×	×	×	×	✓
PD 解耦	✓	×	×	×	✓	✓	✓
数据并行注意力	✓	×	×	×	✓	×	✓
混合专家 / 专家并行	✓	×	×	✓	✓	×	✓

表 1：大语言模型推理建模方法比较。离散事件模拟器需要手动重新实现各种功能，从而带来维护负担。较早的模拟器，如 Vidur（VD） [4], LLMServingSim（LS1） [12],和 APEX（AP） [20] 缺少关键的现代功能。较新的模拟器，如 Frontier（FT） [13],TokenSim（TS） [29], 和 LLMServingSim2.0（LS2） [11]弥补了这些不足，但仍面临与其前代产品类似的维护挑战。

本文做出了以下贡献：

- 我们提出了 REVATI，这是一种时间扭曲仿真器，无需重新实现系统逻辑，即可在类似仿真的速度下对服务系统性能进行建模。
- 我们开发了虚拟时间协议，在保持因果关系的同时，协调分布式进程之间的时间跳跃。
- 我们在 vLLM 和 SGLang 上评估了 REVATI，实现了 $<5\%$ 的预测误差，同时运行速度比真实图形处理器执行快 5–17$\times$ 倍。

2 背景与动机

性能建模对于高效部署大语言模型服务系统至关重要，因为在 GPU 集群上进行真实环境实验的成本高得令人望而却步。然而，由于功能对等方面的滞后，现有基于模拟器的方法难以在服务系统不断演进时保持保真度。本节阐述配置优化所面临的挑战，并说明离散事件仿真为何无法满足现代服务基础设施的需求。

2.1 配置优化问题

现代大语言模型服务系统拥有极其庞大的配置空间。以部署 Qwen3-235B-A22B [30] 为例：运维人员必须选择张量并行（TP）度数（1-8）、流水线并行（PP）阶段数

（1-8）、专家并行（EP）度数（1-8）、最大批量大小（1-256）、分块预填充大小（256-8192 个词元）、KV缓存驱逐策略（最近最少使用、最不经常使用、成本感知）、路由策略（轮询、粘性、缓存感知）以及解耦策略（共置或分开进行预填充/解码）[6, 25, 32]——由此形成庞大的配置空间。既有研究表明，根据模型和工作负载特征调整配置选项，可将吞吐量提升 3-5× [4, 32]。例如，Mitra 等人 [22] 指出，对于类似 RAG、预填充密集型工作负载（16K 输入词元、2K 输出词元），采用 PD 解耦服务 Llama-70B [24, 32] 可带来 1.8×的吞吐量提升，相较于共置方案 [5] 。然而，对于解码密集型工作负载（2K 输入词元、8K 输出词元），他们发现解耦带来的收益微乎其微（<10 % 的吞吐量提升），甚至可能因 KV 缓存传输开销以及流量模式的动态变化而降低性能。

为了有效部署这些系统，运营人员会在具有代表性的工作负载上评估这些配置，并选择满足延迟要求且能够提供最大吞吐量的配置。评估每种配置大约需要 2–4 小时的性能分析，以收集不同请求负载水平下具有统计显著性的尾延迟特征 [4]。按照当前云定价（H100 每个 GPU 小时 2.50–7 美元），对于一个包含 64 个 GPU 的 GPU 集群，仅评估 100 种配置就需要 12,800–25,600 个 GPU 小时，成本为 32,000–179,200 美元。这些因素使得更广泛地探索设计空间在经济上和实践上都不可行，迫使从业者只能依赖经验法则启发式方法。

2.2 离散事件模拟器

为了降低评估配置的成本，研究人员开发了离散事件模拟器（DES） [4, 11–13, 20, 29]。这些模拟器无需昂贵的硬件，即可提供一种低成本且快速的方式来建立服务系统的模型。这些工具通过在简化的模拟框架中手动重新实现服务系统的控制逻辑——包括调度器、内存分配器和请求路由器——来运行。性能模型会预测该批次的 GPU 执行时间。模拟器将其时钟推进这一预测时长，然后处理下一个事件。由于消除了实际的 GPU 执行，模拟器无需硬件即可实现最高达实时运行十倍的加速比。

2.3 模拟器的局限性

语义鸿沟。 这一实现工作不可避免地会在模拟器与实际系统行为之间引入细微差异。早期系统尚且易于建模。例如，Vidur 使用 ∼150 行代码对原始 vLLM 调度器进行了建模。如今，生产控制平面的代码量已超过数万行[26, 28] ，并包含用于实现分层前缀缓存和解耦执行等功能的复杂逻辑。要捕捉这些功能细微的性能特征，需要投入数量庞大且难以驾驭的工程工作。模拟器逻辑与真实情况的任何偏差，都会导致性能预测出现严重不准确。

框架特异性。 模拟器在通用性方面也面临困难，因为服务框架往往会针对同一功能做出根本不同的实现选择。例如，虽然 vLLM 和 SGLang 都在较高层面实现了 Sarathi-Serve[5] 中描述的批处理策略，但两者的具体实现却存在显著差异。vLLM 采用分块预填充和混合批处理（在同一批次中合并预填充请求与解码请求），而 SGLang 默认不执行混合批处理，而是采用预填充优先策略。这种差异会导致延迟特征各不相同，通用实现无法捕捉到这些差异。我们还在其他功能中观察到类似差异，例如前缀缓存、解耦执行等。例如，vLLM 中的分层缓存（使用 LMCache [3]）采用直写策略，即每个写入图形处理器的缓存元素也会立即写入较低层级（中央处理器、非易失性内存 express 等）。另一方面，SGLang 采用选择性直写策略，仅在首次缓存命中时异步复制数据。

为了准确模拟这一生态，研究人员不得不为每个主要框架分别构建并维护专用模拟器：Vidur [4] 以 Sarathi-Serve [5], 和 LLMServingSim2.0 [11]为模型，vLLM 则以此为模型。这种方法不仅限制了灵活性，还增加了构建这些模拟器所需的工程工作量。

维护负担。 构建离散事件模拟器所需的工程开销，因大语言模型服务器的演进速度极快而进一步加重。仅 vLLM 在 2025 [28] 上半年就收到了数百位贡献者提交的 2,800 多次代码更改。每次提交都可能使模拟器的建模假设失效。因此，Vidur 和 LLMServingSim 等模拟器往往在发布后不久就落后于实际发展。为弥合这一差距，人们也做出了一些尝试——Frontier [13]和 LLMServingSim2.0 [11] 在上一代模拟器的基础上扩展了前缀缓存和 PD 解耦等现代功能。然而，随着领域不断发展，即便是这些努力也注定要面临无法避免的可维护性挑战。

要点： 现有的离散事件仿真方法面临持续不断的维护负担。我们需要一种全新的运行时建模方法，不再需要手动重新实现服务引擎逻辑。

3 时间加速仿真

为了解决离散事件仿真的局限性，我们提出了一种新的建模范式：时间加速仿真。该方法不再重新实现服务系统的逻辑，而是允许我们直接执行未经修改的大语言模型引擎，在保留仿真速度和成本效益的同时，实现高保真度的性能建模。本节将介绍实现这一方法的关键构建模块。

3.1 透明设备虚拟化

Maya 在近期一项关于深度学习训练性能建模的工作中， [31],展示了透明设备仿真的可行性。通过拦截 CUDA API 调用LD_PRELOAD，可以创建“虚拟设备”，在不需要物理 GPU 的情况下对工作负载的运行时间进行建模。Maya 通过构建两步流水线实现这一点：首先，使用虚拟设备执行工作负载。该运行过程会生成工作负载发出的所有 CUDA API 调用的跟踪记录。随后，模拟器重放该跟踪记录，预测每项操作的运行时间，最终输出端到端运行时间估计值。

将此技术应用于推理服务，有望直接执行未经修改的服务系统代码，从根本上消除离散事件仿真的主要局限，为现代大语言模型服务系统实现稳健且易于维护的运行时建模开辟道路。

3.2 在线服务中的时间语义

遗憾的是，与通常离线运行的训练工作负载不同，推理是一个在线过程，请求到达时间会从根本上改变控制流。忽略图形处理器执行时间会破坏这些时间语义，导致批处理决策错误。

例如，假设有两个请求 A 和 B，它们都需要 500 毫秒进行预填充。请求 A 在$t = 0$时到达，请求 B 在$t = 200$ 毫秒时到达。在真实系统中，调度器会开始处理 A；200 毫秒后 B 到达时，图形处理器仍在处理 A。B 会被加入队列，调度器最终会在请求 A 的预填充完成后调度 B——也许会将 B 的预填充与 A 随后的解码步骤批处理。不过，如果我们只是跟踪

未考虑图形处理器实时执行情况的图形处理器

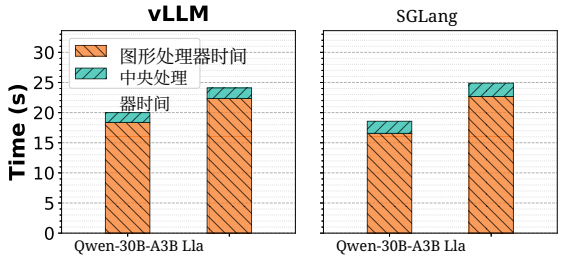


图3：**中央处理器与图形处理器执行时间**。图形处理器执行时间主导服务延迟。较低的中央处理器开销（5-10%）创造了通过时间加速实现数量级加速比的机会。

（如 Maya 的离线仿真方法所示），请求A在$t \approx 0$时立即完成。当请求 B 在$t = 200$ 毫秒到达时，系统会认为图形处理器处于空闲状态且队列为空。时间上的重叠消失了，批处理模式也完全改变了，模型结果与实际情况不符。

一种基线方法。 为了保留时间语义，仿真器必须让控制平面感受到时间的流逝。一种简单但正确的方法是通过休眠 预测的内核运行时长，来模拟图形处理器延迟。如果预测一个批次需要 500 毫秒，仿真器就会阻塞中央处理器线程 500 毫秒。这样便恢复了保真度：请求 A 休眠的这 500 毫秒里，挂钟时间持续前进。请求 B 按计划在$t = 200$毫秒时到达，但此时发现工作进程正处于“忙碌”（休眠）状态。调度器会正确地将 B 加入队列，从而精准保留系统的状态动态和资源竞争情况，使其与真实硬件上的表现完全一致。

尽管基于休眠的仿真能够保证正确性，但其速度慢得难以接受。评估一条持续 2 小时的跟踪记录需要耗费 2 小时的实际时间。对于需要进行数千次评估的配置搜索，这种实时约束使该方法不切实际。我们将这种基线方法称为基于休眠的仿真。

3.3 关键洞察：时间加速

这种简单的仿真方法所耗费的墙上时钟时间与完整的图形处理器执行相同。然而，仔细观察推理引擎的设计便会发现一个机会：这些系统经过精心设计，旨在最大限度地提高图形处理器利用率，同时尽量减少中央处理器开销。图3量化了这一观察结果。我们在离线模式（所有请求在开始时均已就绪）下处理100个请求，并测量两种条件下的执行时间：正常的图形处理器执行，以及通过仿真绕过所有图形处理器执行的跳过图形处理器模式。中央处理器时间代表关键路径控制平面开销，而图形处理器时间则反映实际的内核执行情况。在 vLLM 和 SGLang 中，图形处理器计算均占总执行时间的90%至95%，而中央处理器开销

始终很低，不受模型规模或服务框架的影响。因此，系统绝大部分墙上时钟时间都只是用于等待图形处理器计算完成。

更重要的是，控制平面的逻辑并不依赖由图形处理器计算出的数值。调度器处理的是逻辑抽象（请求、批次、内存块），而图形处理器工作线程执行物理计算，不负责做出控制流决策[1]。这些特性自然引出了一项关键优化：与其在系统空闲的 85%–95% 时间里休眠，我们或许可以让虚拟时间向前跳跃，从而加速仿真。

应对因果关系。尽管从原理上讲，时间跳跃的概念很简单，但要在分布式服务系统中实现它，会带来一个关于因果关系的根本挑战。在服务系统中，多个相互独立的进程（批次调度器、工作进程和请求负载生成器）并发运行。每个进程都有自己对下一个事件的判断。例如，工作进程可能准备向前跳跃 50 毫秒，以模拟批次执行，而请求负载生成器则计划在 10 毫秒后注入一个新请求。

如果工作进程单方面将其本地时钟提前 50 毫秒，就会实际上“跨过”新请求的到达时刻。在虚拟时间线上，请求在批次完成之后才到达，而在现实中，它本应在执行期间到达，从而可能改变调度器对后续批次的决策。这种对因果关系的违-反会导致错误的排队行为和无效的延迟测量。

文献中的时间加速。这是一个经典问题，在分布式事件仿真器领域已得到广泛研究。乐观方法 [17, 18] 允许不同进程独立推进其时钟。当检测到因果关系违-反时，进程状态会回滚到上一个一致的检查点。然而，这种方法不适用于仿真，因为服务系统无法回滚。相比之下，保守方法 [7–9]更适合用于仿真。这些方法要求系统中的每个进程都同意一个共同的前瞻时间，在该时间范围内可以安全地推进执行，而不会违反因果关系。

尽管文献中讨论过几种保守时间加速的实现，离散事件仿真 [7–9], 这些方法都不能直接应用于仿真。在事件驱动模拟器中，事件循环可以直接控制事件的因果关系。相比之下，在仿真系统中，所有分布式组件都以实时方式运行，因此我们需要在不引入任何额外开销的情况下透明地加速时间。例如，Chandy-Misra-Bryant [7, 8] 算法要求每个仿真工作进程向每个对等节点发送一条空消息，并附上下一次前瞻时间。模拟器需要能够控制事件调度，才能实现这种方法。本文提出了一种方法，可在不违反仿真系统中因果关系的前提下实现时间加速。

[1]注：在实际部署中，请求的终止由 EOS 词元的生成决定。然而，出于性能建模的目的，通常会通过设置特定的采样参数，让每个请求运行预先确定数量的词元。

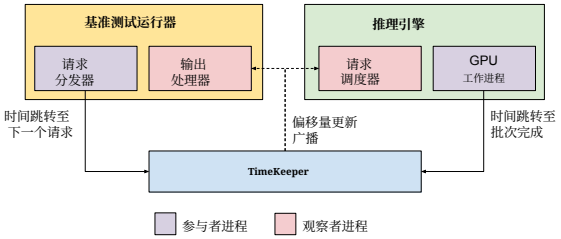


图 4：**跨进程协调虚拟时间**。基准测试运行器和推理引擎作为独立进程运行，每个进程都包含请求时间跳转的参与者（蓝色）以及接收时间更新的观察者（粉色）。*Timekeeper* 负责协调跳转请求，并广播一致的偏移量。

4 REVATI：系统设计

REVATI 通过时间扭曲仿真弥合了并行分布式事件驱动仿真与仿真方法之间的差距。要实现这一范式，必须解决分布式服务系统的异步特性与准确建模所需的严格因果性之间的矛盾。本节其余部分将详细介绍 REVATI 的设计，以及它如何应对与时间虚拟化相关的挑战。

4.1 概述

进程模型。对大语言模型服务系统进行性能评估涉及两个主要部分：提交请求的基准测试运行器，以及处理这些请求的大语言模型推理引擎（图 4）。这两个组件对应 REVATI 执行模型中的两种时间角色。*参与者* 具有确定性的计划，并且知道其未来操作何时完成。例如，基准测试运行器的请求分发器知道何时分发每个新请求（例如，在 t=100ms、t=250ms），而推理引擎的图形处理器工作线程知道批次执行何时结束。这些进程会主动推动虚拟时间前进。*观察者* 则完全是被动响应式的，会在事件发生时作出响应。基准测试运行器的输出处理器和推理引擎的请求调度器就是两个这样的例子。

系统架构。REVATI 由两个主要组件构成：**Timekeeper**，负责在分布式组件之间同步虚拟时间，并管理时间跳跃以加速执行；以及**设备仿真层**，负责透明地拦截并模拟图形处理器操作，无需物理硬件。

将这些部分与 REVATI 集成，需要修补系统中的每个参与者，使其在可能时进行时间跳跃。基准测试运行器必须经过修改，以便在请求分发之间进行时间跳跃。同样，推理引擎的图形处理器工作进程也需要跳过图形处理器批次执行。这些修改加起来，针对 vLLM 和 SGLang 大约需要 50 行代码。设备仿真层通过 LD_PRELOAD 加载，以拦截与图形处理器内核分发分开的任何设备管理调用。执行开始时，参与者会向*TimeKeeper*提交并发时间跳跃请求——基准测试运行器请求跳转到下一个请求到达的时间点，图形处理器工作进程则请求跳过其批次执行所需的时长。TimeKeeper 会计算所有参与者的最小目标时间，将全局虚拟时钟推进至该时刻，并广播更新。目标时间已到的参与者会立即返回。目标时间更晚的参与者仍保持阻塞，并在后续屏障轮次中重新提交请求。观察者则异步查询虚拟时间，为事件添加时间戳，而不参与此协调过程。

4.2 *TimeKeeper*

TimeKeeper 是一项用于管理已连接客户端之间虚拟时间的服务。它提供 TIMEJUMP(Δt) API，允许客户端请求时间跳跃并接收时钟更新。我们将 *TimeKeeper* 的客户端分为参与者或观察者。参与者是仿真的主动驱动组件，它们会处理持续时间可预测的操作，并且必须主动请求推进虚拟时间。观察者是响应式组件，它们使用虚拟时间为事件添加时间戳，但不会阻碍时间推进。这种区分使 REVATI 能够最大限度地减少协调开销：只要求参与者进行同步，同时只需少量代码更改即可集成服务框架。

为确保可扩展性，该架构将请求路径 与更新路径分离。参与者 通过可靠的通道（扇入），分别向 *Timekeeper* 提交时间跳转请求。*Timekeeper* 通过广播通道同时向所有客户端分发时钟更新，
扇出）。这种不对称性使 *Timekeeper* 能够以每轮恒定的序列化成本更新数百个分布式工作节点，从而避免高频屏障解析期间出现瓶颈。

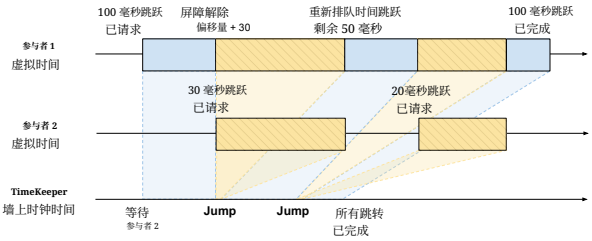


图 5：**基于屏障的时间同步**。 当参与者请求不同的跳跃时长时，*Timekeeper* 会前进到最小安全偏移量。此处，参与者 1 在$t = 0$请求跳转 100 毫秒。执行跳转之前，*Timekeeper* 会等待所有参与者发布各自的安全偏移量。当参与者 2 在$t = 20$请求跳转 30 毫秒时，屏障解除，*Timekeeper* 将时间推进 30 毫秒。参与者 1 还剩余 50 毫秒，因此会重新排队，提交剩余时长的跳转请求。这样，参与者 1 原本的 100 毫秒跳转就分两段完成，而墙上时钟时间总共仅耗时 50 毫秒。

4.2.1 虚拟时间协议

要协调分布式进程之间的虚拟时间，需要一种既能保留因果关系、又不控制进程执行的协议。本节介绍 REVATI 的基于屏障的协议，并分析其正确性属性。

设计约束。 将虚拟时间适配到实时仿真中，会引入传统离散事件模拟器中不存在的三项约束：

•不可回滚。 Time Warp 等乐观方法 [17, 18] 会试探性地推进时间，并在发生因果关系违规时回滚。由于我们执行的是真实系统代码，且会产生不可逆的副作用——例如发送网络消息、修改数据结构——这种方法不可行。

• 无法控制事件调度。Chandy-Misra-Bryant 等保守协议 [8]假定模拟器能够控制事件处理顺序。我们的进程按照墙上时钟时间异步运行；协议必须协调时间推进，而不能控制执行。

• 优雅降级。当协调失败时（例如出现迟到进程或网络延迟），系统必须保持正确，即使可能牺牲速度，也绝不能产生无效结果。

虚拟时间表示。REVATI 将全局虚拟时钟维护为相对于墙上时钟时间的偏移量：

$$t_{virtual} = t_{wall} + offset \tag{1}$$

初始时偏移量 $= 0$，因此虚拟时间等于墙上时间。随着参与者请求时间跳转，*TimeKeeper* 会增加偏移量，使虚拟时间比墙上时间前进得更快

算法 1客户端：时间跳跃(Δt)

要求： $\Delta t > 0$：要推进到的虚拟时间（毫秒）

1: $t_{target} \leftarrow$ 获取虚拟时间() $+ \Delta t$
▷计算一次绝对目标值
2: **当** 获取虚拟时间() $< t_{target}$ **时**
3: 发送时间跳跃请求(t_{target})
▷请求前进至目标
4: 等待确认()$_{t_{target}}$ −GETVIRTUALTIME()
5: $t_{remaining}$
6: **如果** $t_{remaining} > 0$ **则**
7: 等待时钟更新($t_{remaining}$)
▷阻塞，直到收到广播或发生超时
8: **结束 if**
9: **结束 while**

。这种表示方式让观察者无需协调即可查询虚拟时间——只需读取当前偏移量并加上墙上时间。

基于屏障的协调。 当 参与者 调用 TIMEJUMP(Δt) 时，它会计算目标虚拟时间，并向*TimeKeeper*发送请求。*TimeKeeper* 收集所有参与者的请求，然后将虚拟时间推进到所请求目标中的最小值。这种推进最小值的规则可确保没有参与者 超前于其预定时间，从而保持因果关系。例如，如果一个工作进程需要 50 毫秒处理其批次，而另一个需要 10 毫秒，若各自独立推进时间，就会破坏因果关系：虚拟时间为参与者 在 $t = 1000$ 时，可能会观察到仍处于参与者 的事件，其时间为 $t = 500$。算法 2 展示了服务器端协议。

一次 TIMEJUMP 调用可能跨越多个屏障轮次。考虑两个工作进程：W_A 调用 TIMEJUMP(50)，W_B 调用 TIMEJUMP(10)。Timekeeper 会根据W$_{min}$的目标计算出t_B，并将时间推进 10 毫秒。 W_B的调用返回。W_A 仍处于循环中——虚拟时间虽然已经推进，但尚未达到W_A的目标——因此会在下一轮屏障轮次中重新请求。算法 1 展示了客户端协议。客户端只计算一次目标时间（第 1 行），然后循环，直到虚拟时间达到该目标。每次迭代都会将目标发送给*TimeKeeper*，接收确认，并等待时钟更新广播；等待超时时长等于所需的剩余虚拟时间。

优雅降级。 算法1 中的超时机制确保了故障情况下的正确性。如果广播始终未到达——由于网络问题或停滞的参与者——等待会在$t_{remaining}$ 墙上时钟毫秒数后超时。此时，墙上时间已前进$t_{remaining}$，并且由于$t_{virtual} = t_{wall}$ +虚拟偏移量，虚拟时间也随之前进。循环条件得到满足，TIMEJUMP 返回。

操作能够正确完成，但速度仅与实际时间一致。最坏情况下，REVATI 会退化为基于休眠的仿真——速度较慢，但绝不会出错。这一特性保证 REVATI 不会产生无效结果，也不会

算法 2 服务器：ProcessTimeJumpRequests()

要求： *numActors*：已注册参与者的数量

1: 待处理 $\leftarrow \emptyset$， 虚拟偏移量 $\leftarrow 0$
▷ 初始化待处理请求和虚拟偏移量
2: **当 true 时执行**
3: $(c,\ t_{target}) \leftarrow$ R接收R请求()
4: 待处理$[c] \leftarrow t_{target}$
▷存储客户端 c 的请求
5: **如果** |待处理| =参与者数量 **则**
▷ 所有参与者都到达屏障——推进虚拟时间
6: $t_{min} \leftarrow$ 最小$\{t :\ t \in$ 待处理$\}$
▷最小目标保持因果关系
7: **如果** 获取墙上时钟时间() $< t_{min}$ **那么**
8: $t_{wall} \leftarrow$GETWALLTIME()
9: 偏移量 $\leftarrow$ 最大值(偏移量,$t_{min} - t_{wall}$)
10: 广播时钟更新(偏移量)
▷ 通知所有客户端
11: **结束 if**
12: 待处理 $\leftarrow \emptyset$
▷ 为下一轮重置
13: **结束 if**
14: **结束 while**

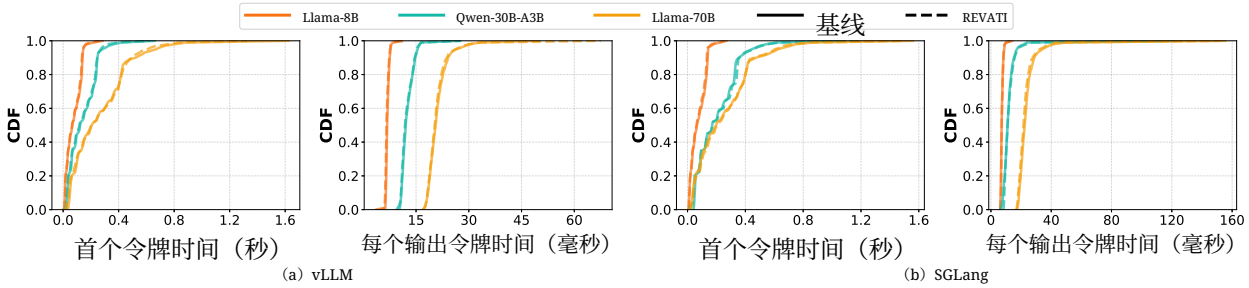
发生死锁；它只会失去加速效果。

处理消息抖动。实时仿真必须考虑网络延迟和 CPU 调度抖动。考虑以下场景：推理服务器在墙上时间 t_w=100毫秒 生成一个词元，此时虚拟时间为t_v=3400毫秒。基准测试运行器及时接收该词元，并记录t_v=3400毫秒。随后，服务器处理下一个批次（预计耗时 20 毫秒），将虚拟时间更新为t_v=3420毫秒，对应墙上时间 t_w=101毫秒。由于网络延迟，运行器在t_w=102.5毫秒读取到下一个词元。与此同时，服务器完成另一个批次，并在t_w=102毫秒将虚拟时间更新为t_v=3450毫秒 。当运行器在t_w=102.5毫秒查询虚拟时间时，它观察到的是 3450 毫秒而非 3420 毫秒，导致延迟测量不准确。

在消息生成后立即为其附加虚拟时间戳，可以消除这种歧义。尽管这种方法很准确，但它也要求对服务引擎进行重大修改（数百行代码）。为了便于集成，REVATI 还提供了有界抖动模型：假设消息在传输时间的J 范围内接收，则在连续的时间跳跃之间插入持续时间为J的冷却时间，可以防止观察者 读取过时的虚拟时间。我们发现，在大多数实际场景中，$J \approx 500\mu s$ 就足够了。

4.3 设备仿真层

服务框架与 CUDA 和 NCCL 深度集成。修改所有调用点需要对每个框架进行侵入性改动，这会损害 REVATI 的



（a）vLLM （b）SGLang

图7：**端到端准确性**。 对于三种模型配置，REVATI（虚线）在首令牌时间和每输出令牌时间分布上都与实际执行（实线）高度吻合。即使在尾部，预测误差也低于 5%。

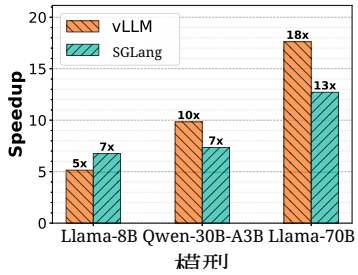


图7：**端到端加速比**。 与图形处理器上的实际执行相比，REVATI 在 vLLM 上实现了 5-17× 倍加速，在 SGLang 上实现了 6-12× 倍加速。

透明性目标。设备仿真层利用 Maya 的 [31]透明设备虚拟化技术，拦截并仿真推理工作负载中的 CUDA API 调用。未经修改的框架代码可以正常执行，同时认为自己能够访问目标硬件。除了保持透明性之外，这种方法还支持在超出可用资源的规模下进行评估。例如，研究人员若想在由 128 个 H200 图形处理器组成的集群上进行评估，但无法访问这类硬件，只需配置 REVATI 来仿真所需硬件即可。

然而，实时进行仿真（与 Maya 的离线方法相对）带来了一些独特的挑战，我们将在下文讨论。

保留分布式依赖关系。 在分布式服务（流水线并行、张量并行）中，NCCL 集合通信会强制建立因果依赖关系——阶段$i{+}1$ 必须等到阶段 i 完成 ncclSend 后才能继续。由于没有实际的图形处理器执行，REVATI 必须阻塞仿真的阶段$i{+}1$，直到仿真的阶段 i 到达相应的发送操作。我们将 NCCL 集合通信转换为参与工作进程之间的屏障同步点，在不传输数据的情况下保留时间顺序。

分离状态内存模型。与训练工作负载不同，服务系统通常使用 GPU 通道进行控制平面通信。例如，SGLang 会通过 NCCL 广播批次组成。纯黑盒仿真会因此失效，因为中央处理器需要这些缓冲区中包含有效数据。为解决这一问题，REVATI 将分配划分为两类。元数据缓冲区——可能用于控制决策的小型分配（默认小于 4MB）——由真实的主机内存支持，其操作会如实执行。计算缓冲区——用于 KV缓存和权重的大型分配——则获得没有物理后备存储的虚拟指针，其操作会变为空操作，持续时间由运行时预测器估算。该模型依赖于中央处理器永远不会读取计算缓冲区。为确保这一不变量，若应用程序试图读取虚拟计算缓冲区，REVATI 会抛出致命异常，而不是返回垃圾值。因此，仿真成功运行即可保证控制平面从未基于虚假数据运行。

4.4 运行时预测

REVATI 提供了可插拔接口，同时支持分析型和基于性能分析的运行时预测器。默认情况下，我们扩展 Vidur 的 [4] 算子级模型，使其支持混合专家路由、融合注意力变体以及经过优化的全归约集合通信。预测器接收批次组成（预填充数量、解码数量、序列长度）和目标硬件规格，并返回估计时长。

5 实现

Timekeeper 服务器和客户端库共包含 ~800 行 C++ 代码，并提供用于框架集成的 Python 绑定。设备模拟器增加了 ~6000 行 C++ 代码。针对 vLLM 和 SGLang 的大语言模型服务器补丁分别少于 50 行。消息层使用 ZeroMQ 实现低延迟通信。

Timekeeper 服务器采用多线程架构：专用的 I/O线程负责序列化和套接字操作，后台线程负责管理屏障状态和时间跳跃逻辑。这种分离确保高频请求不会阻塞屏障处理。

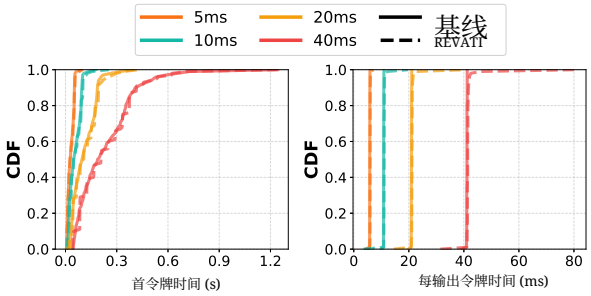


图 8：**不同批次持续时间下的准确性**。 RE- VATI（虚线）在所有批次持续时间下都与基于休眠的仿真（实线）相匹配。

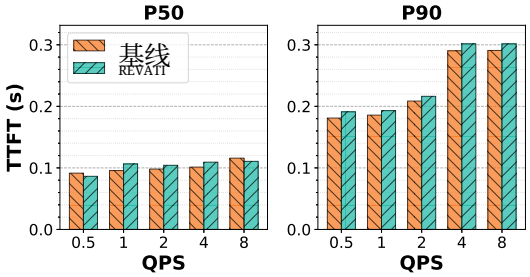


图 9：**不同请求速率下的准确性**。在不同请求到达率下，REVATI 能准确地模拟中位数和尾延迟趋势。

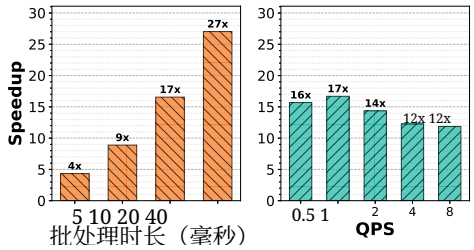


图10：**不同参数下的加速比**。在不同批次持续时间和请求速率下，REVATI 始终保持较高的加速比。

6 评估

本节将全面评估 REVATI，以展示其在仿真大语言模型推理性能方面的保真度。

- **准确性**：REVATI 对不同模型规模和部署配置下端到端推理延迟的预测有多准确?
- **效率**：与基线方法相比，REVATI 能否有效加速评估?

6.1 实验设置

模型与推理系统。 我们针对一系列模型规模开展性能分析和仿真，以确保结果具有普适性。具体而言，我们评估了系统在 2 个稠密模型（Llama-3.1-8B 和 Llama-3.1-70B）以及 1 个稀疏模型（ Qwen3-30B-A3B）上的性能。对于 Llama-3.1-8B 和 Llama-3.1-70B，我们分别将张量并行度设为 1 和 4。对于 Qwen3-30B-A3B，我们采用专家并行度为 2 的配置。为评估系统的普适性，我们在两种不同的服务引擎——vLLM [28] 和 SGLang [26]上进行了实验。为确保比较一致，我们在所有设置中均启用分块预填充和混合批处理，块大小为 512。

硬件。我们在一台配备 4 个 H200 图形处理器的机器上进行评估，该机器采用全互连 NVLink 网络，并搭载一颗 128 核 AMD 9334 EPYC CPU 和 756GB 内存。

6.2 端到端评估

我们使用 ShareGPT 数据集和泊松到达，在两个生产级服务引擎（vLLM 和 SGLang）以及三种模型配置（ Llama-3.1-8B、Llama-3.1-70B 和 Qwen3-30B-A3B）上，通过端到端推理基准测试评估 REVATI 的准确性和加速比。

准确性。 图 6 比较了 REVATI 预测的延迟分布（虚线）与实际执行结果（实线）。在所有模型规模和两种服务引擎中，REVATI 对基线首令牌时间（TTFT）和每输出令牌时间（TPOT）分布的预测都十分准确。预测延迟与累积分布函数（CDF）几乎完全吻合，中位预测误差低于 5%。通过直接执行服务引擎，REVATI 能够准确模拟这些系统调度策略中的细微差异。我们发现，尽管两个系统的 TTFT 性能相似，但在解码 TPOT 方面，SGLang 的长尾表现明显更差——这是因为 SGLang 默认不执行混合批处理，尽管它会进行分块预填充。

加速比。如图7 所示，得益于其时间加速协议，REVATI 在 vLLM 和 SGLang 上的实际执行速度均实现了十倍以上的加速。值得注意的是，随着模型规模增大、图形处理器执行时间增加，我们获得了更高的加速比，其中 Llama-3.1-70B 的性能最高。

6.3 消融实验

为了解工作负载特征如何影响 REVATI 的性能，我们开展了多项消融实验。

不同的批次持续时间。首先，我们评估当模型规模和硬件发生变化时，REVATI 能否准确且高效地运行。我们通过使用静态

批次时间预测来模拟这种情况，预测时长在 5 到 40 毫秒之间。我们将性能与基于朴素睡眠的方法进行比较；在这种方法中，GPU 工作线程只是休眠相应的批次时长，而不进行时间跳转。

如图 8所示，REVATI 生成的首令牌时间和每输出令牌时间分布具有很高的准确性，误差低于 5 %。随着我们增加批处理时长，加速比也随之提高，在批次运行时间为 40ms 时最高可达 27× 。这直接得益于 REVATI 能够跳过 GPU 计算等待时间。

变化的到达率。工作负载的另一个可能直接影响虚拟时间推进协议准确性和有效性的因素是请求到达率。图 9 展示了请求到达率在每秒 0.5 至 8 个查询（每秒查询数）之间变化时 REVATI 的性能。为排除运行时预测器引入的任何误差，我们在固定批次时间为 20 ms 的条件下进行此实验，并与基于休眠的基线进行比较。在所有情况下，系统对首令牌时间的预测误差均保持低于 5%。

随着到达率提高，我们观察到加速比略有下降。这是因为在较高的到达率下，系统以更大的批次运行，从而导致批次形成和记账所需的 CPU 工作增加。然而，由于与批次执行时间相比，这一开销相对较小（通常为< 200μs），即使在高请求负载下，REVATI 仍能保持超过一个数量级的加速比。

7 相关工作

分布式仿真中的虚拟时间。 Jefferson 的 Time Warp [17,18] 开创了乐观虚拟时间的先河：进程以推测方式推进，并在发生因果关系违规时回滚。Chandy-Misra [7,8] 提出了保守协议，进程通过交换前瞻信息来安全推进。这两种方法都假设模拟器能够控制事件调度。REVATI 面临着不同的约束：进程在墙上时钟时间下异步运行，并产生无法回滚的实际副作用。我们的基于屏障的协议将保守原则应用于这一场景，利用最小目标推进来保持因果关系，同时通过超时确保平稳退化为实时执行。

网络仿真的时间膨胀。 时间膨胀 [14,15,27] 会减慢虚拟时间，使物理硬件看起来更快——通过让应用以 10× 更慢的虚拟时间运行，1 Gbps 网卡便可仿真 10 Gbps 的网络。DieCast [14], 、ModelNet [27],和 Mininet [16] 等工具会采用这一技术进行大规模网络测试。REVATI 则反转了这一关系：我们不是通过减慢时间来放大硬件能力，而是加速时间，以跳过图形处理器计算。两者都基于这样一种认识：应用可以在与墙上时钟时间脱钩的虚拟时间中运行，但具体机制截然不同。网络仿真器会拦截系统调用并按比例调整延迟；REVATI 则通过显式的屏障同步来协调分布式进程，以在时间跳跃期间维持因果关系。

8 讨论

迈向完全透明的仿真。 REVATI 要求针对每个框架进行最小的（~50 行）补丁，以调用时间跳转。不过，这引出了一个自然的问题：我们能否像 Maya 在训练中那样，通过内核级拦截彻底消除这一要求？然而，这种方法面临两个挑战。首先，现代推理系统对某些复杂操作（如 Flash Attention、混合专家路由）采用了特定于框架的实现。仅凭内核 API 签名来预测这些操作的运行时间，需要对框架内部进行复杂的逆向工程，而这会带来 REVATI 力求避免的维护负担。最近也有研究尝试直接根据图形处理器内核的源代码和输入参数来估算其运行时间 [19, 21] ——这一方向有望在未来实现完全透明的仿真。

光速执行。系统社区一直在努力降低服务系统中的 CPU 开销，以最大限度地提高图形处理器利用率。随着图形处理器在一代代产品中变得越来越快、功能越来越强，这一点也愈发重要。近期，人们付出了多项努力，尝试使用 C++ 和 Rust [1, 2] 实现高效执行，而非采用传统的 Python 实现。SGLang 等现有框架最近也开始将代码库中的关键部分（如前缀缓存树）迁移到 C++，以实现这一目标 [26]。有趣的是，这引出了一个问题：借助 REVATI，我们最多能实现多大程度的加速？随着仿真速度提升，我们预计会遇到更多有趣的挑战和系统瓶颈，例如，词元化/去词元化步骤可能会带来不可忽视的开销等。我们希望在未来的工作中探索这些挑战。

9 结论

大语言模型服务系统的迅速发展，使传统的离散事件仿真难以为继。模拟器中的模型与前沿级服务系统之间的差距，靠人工弥合的成本实在太高。REVATI 通过直接基于原有服务框架进行低成本、高精度的运行时建模，彻底消除了这一差距。我们展示了，透明的设备仿真层与基于屏障的

虚拟时间协议相结合，能够实现高精度运行时建模（误差为 <5%），同时比物理执行快数个数量级。

参考文献

[1] ggml-org/llama.cpp：用 C/C 实现的大语言模型推理++— github.com。https://github。com/ggml-org/llama。cpp。

[2]Huggingface 文本生成推理：github。https://github。com/huggingface/text-generation-inference。

[3] LM Cache：用最快的键值缓存层为您的大语言模型加速。https://github。com/LMCache/LMCache。

[4] Amey Agrawal、Nitin Kedia、Jayashree Mohan、Ashish Panwar、Nipun Kwatra、Bhargav S Gulavani、Ramachandran Ramjee 和 Alexey Tumanov。Vidur：大语言模型推理的大规模仿真框架。*MLSys*，2024。[5] Amey Agrawal、Nitin Kedia、Ashish Panwar、Jayashree Mohan、Nipun Kwatra、Bhargav S Gulavani、Alexey Tumanov 和 Ramachandran Ramjee。驾驭大语言模型推理中的吞吐量与延迟权衡：Sarathi-Serve。*OSDI*，2024。[6] Amey Agrawal、Ashish Panwar、Jayashree Mohan、Nipun Kwatra、Bhargav S. Gulavani 和 Ramachandran Ramjee。SARATHI：通过分块预填充搭载解码，实现高效的大语言模型推理，2023。[7] Randal R Bryant。计算机系统中分组通信架构的仿真。1977年。[8] K. Mani Chandy 和 Jayadev Misra。通过一系列并行计算实现异步分布式仿真。《*ACM*通讯》，24(4)：198–206，1981。[9] K. Mani Chandy 和 Jayadev Misra。分布式仿真：分布式程序设计与验证案例研究。《*IEEE*软件工程汇刊》，(5)：440–452，2006。

[10] Yihua Cheng、Yuhan Liu、Jiayi Yao、Yuwei An、Xiaokun Chen、Shaoting Feng、Yuyang Huang、Samuel Shen、Kuntai Du 和 Junchen Ji ang。Lmcache：用于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。

[11] Jaehong Cho、Hyunmin Choi 和 Jongse Park。LLMServingSim2.0：面向大语言模型基础设施中异构硬件和服务技术的统一模拟器。*IEEE*计算机架构快报，24(2):361–364，2025年7月。[12] Jaehong Cho、Minsu Kim、Hyunmin Choi、Guseul Heo 和 Jongse Park。LLMServingSim：用于大规模大语言模型推理服务的软硬件协同仿真基础设施。载于*2024*年*IEEE*国际工作负载特征分析研讨会（*IISWC*），第15–29页。IEEE，2024。[13] Yicheng Feng、Xin Tan、Kin Hang Sew、Yimin Jiang、Yibo Zhu 和 Hong Xu。Frontier：模拟下一代大语言模型推理系统，2025年。[14] Diwaker Gupta、Kashi Venkatesh Vishwanath、Marvin McNett、Amin Vahdat、Ken Yocum、Alex Snoeren 和 Geoffrey M Voelker。DieCast：使用精准比例模型测试分布式系统。《*ACM*计算机系统汇刊》（*TOCS*），29(2):1–48，2011年。[15] Diwaker Gupta、Kenneth Yocum、Marvin McNett、Alex C. Snoeren、Amin Vahdat 和 Geoffrey M. Voelker。迈向无限：时间扭曲网络仿真。载于第二十届*ACM*操作系统原理研讨会论文集，SOSP ’05，第1–2页，美国纽约州纽约市，2005年。计算机协会。[16] Nikhil Handigol、Brandon Heller、Vimalkumar Jeyakumar、Bob Lantz 和 Nick McKeown。使用基于容器的仿真进行可复现的网络实验。载于第八届新兴网络实验与技术国际会议论文集，CoNEXT ’12，第253–264页，美国纽约州纽约市，2012年。计算机协会。[17] David Jefferson、Brian Beckman、Frederick Wieland、Leo Blume 和 Mike DiLoreto。时间扭曲操作系统。载于第十一届*ACM*操作系统原理研讨会，第77–93页，1987年。[18] David R. Jefferson。虚拟时间。*ACM*编程语言与系统汇刊，7(3):404–425，1985年7月。[19] Seonho Lee、Amar Phanishayee 和 Divya Mahajan。预测深度学习训练和推理的图形处理器性能。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，第一卷，ASPLOS ’25，第493–508页。ACM，2025年3月。

[20] Yi-Chien Lin、Woosuk Kwon、Ronald Pineda 和 Fanny Nina Paravecino。APEX：一种用于大语言模型服务中自动并行执行的可扩展且感知动态性的模拟器。*arXiv* 预印本 *arXiv:2411.17651*，2024年。

[21] Charith Mendis、Alex Renda、Saman Amarasinghe 和 Michael Carbin。《Ithemal：使用深度神经网络进行准确、可移植且快速的基本块吞吐量估算》，2019年。

[22] Tiyasa Mitra、Ritika Borkar、Nidhi Bhatia、Ramon Matas、Shivam Raj、Dheevatsa Mudigere、Ritchie Zhao、Maximilian Golub、Arpan Dutta、Sailaja Madduri 等人。《超越炒作：对推理解耦的务实看法》。*arXiv*预印本 *arXiv:2506.05508*，2025年。

[23] 英伟达公司。Dynamo：面向数据中心规模的分布式推理服务框架。https://github。com/ai-dynamo/dynamo，2025年。访问日期：2025年10月29日。

[24] Pratyush Patel、Esha Choukse、Chaojie Zhang、Aashaka Shah、Íñigo Goiri、Saeed Maleki 和 Ri- cardo Bianchini。《Splitwise：利用阶段拆分实现高效的生成式大语言模型推理》。载于*2024*年*ACM/IEEE*第*51*届年度国际计算机体系结构研讨会（*ISCA*），第118–132页。IEEE，2024年。

[25] Ruoyu Qin、Zheming Li、Weiran He、Mingxing Zhang、Yongwei Wu、Xinran Xu 和 Weimin Zheng。《Mooncake：面向大语言模型服务的 KVCache 中心化解耦架构》，2024。

[26] SGL项目。SGLang：面向大语言模型和视觉语言模型的高速服务框架。
https://github.com/sgl-project/sglang，2025年。访问日期：2025-10-29。

[27] Amin Vahdat、Ken Yocum、Kevin Walsh、Priya Mahadevan、Dejan Kosti´c、Jeff Chase 和 David Becker。《大规模网络模拟器的可扩展性与准确性》。*SIGOPS* 操作系统评论，36(SI):271–284，2003年12月。

[28] vLLM项目。vLLM：为所有人提供简单、快速、低成本的大语言模型服务。
https://github.com/vllm-project/vllm。

[29] Feiyang Wu、Zhuohang Bian、Guoyang Duan、Tianle Xu、Junchi Wu、Teng Ma、Yongqiang Yao、Ruihao Gong 和 Youwei Zhuo。TokenSim：为大语言模型推理系统实现软硬件探索，2025年。

[30] An Yang、Anfeng Li、Baosong Yang、Beichen Zhang、Binyuan Hui、Bo Zheng、Bowen Yu、Chang Gao、Chengen Huang、Chenxu Lv 等人。Qwen3 技术报告。*arXiv* 预印本 *arXiv:2505.09388*，2025年。

[31] Srihas Yarlagadda、Amey Agrawal、Elton Pinto、Hakesh Darapaneni、Mitali Meratwal、Shivam Mittal、Pranavi Bajjuri、Srinivas Sridharan 和 Alexey Tumanov。Maya：使用仿真虚拟加速器优化深度学习训练工作负载。*arXiv* 预印本 *arXiv:2503.20191*，2025年。

[32] Yinmin Zhong、Shengyu Liu、Junda Chen、Jianbo Hu、Yibo Zhu、Xuanzhe Liu、Xin Jin 和 Hao Zhang。{DistServe}：拆分预填充与解码，实现吞吐效能优化的大语言模型服务。载于第 *18* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第 193–210 页，2024。