# xLLM Technical Report


## Abstract

我们推出了 xLLM，一个智能高效的大型语言模型 (LLM) 推理框架，专为高性能、大规模企业级服务而设计，并针对多种 AI 加速器进行了深度优化。当前主流的推理框架面临着实际挑战。一方面，企业级服务难以应对混合和动态工作负载、对服务高可用性的严格要求以及分布式存储管理。另一方面，由于新的硬件、模型架构和推理算法范式，AI 加速器利用率不足，导致推理执行面临瓶颈。

为了应对这些挑战，xLLM 构建了一个新颖的服务-引擎解耦架构。在服务层，xLLM-Service 具有智能调度模块，可以高效处理多模态请求，并通过统一的弹性调度将在线和离线任务共置，从而最大限度地提高集群利用率。该模块还依赖于工作负载自适应的动态预填充-解码 (PD) 分离策略来实现动态实例调度，以及专为多模态输入设计的新型编码-预填充-解码 (EPD) 分离策略。此外，它还采用分布式架构，提供全局键值缓存管理，以实现高效的 AI 加速器内存处理，并提供强大的容错能力，从而实现高可用性。在引擎层，xLLM-Engine 协同优化系统和算法设计，以充分利用计算资源。这是通过全面的多层执行流水线优化实现的，包括将 CPU 调度与 AI 加速器操作重叠以最大限度地减少计算泡沫，采用双流并行将计算与通信重叠，以及细粒度地重叠各种计算单元以最大限度地提高硬件利用率。此外，自适应图模式可大幅降低内核启动开销，创新的“逻辑连续、物理离散” xTensor 内存管理可解决内存分配冲突。 xLLM-Engine 还进一步集成了算法增强功能，例如优化的推测解码和动态专家并行负载均衡 (EPLB)，从而大幅提升了吞吐量和推理效率。

大量评估表明，xLLM 的性能和资源效率显著提升。在相同的 TPOT 约束条件下，xLLM 在执行 Qwen 系列模型时，吞吐量比 MindIE 高出 1.7 倍，比 vLLM-Ascend 高出 2.2 倍；同时，在执行 Deepseek 系列模型时，xLLM 的平均吞吐量比 MindIE 提升 1.7 倍。这些结果最终验证了 xLLM 相对于其他先进推理系统的性能优势。我们已将 xLLM 部署到生产环境中，以支持京东的一系列核心业务场景，涵盖 LLM、多模态大型语言模型 (MLLM) 和生成式推荐等领域。这些应用包括精研 AI 聊天机器人、营销推荐、产品理解、客服助手等等。 xLLM框架已在 https://github.com/jd-opensource/xllm 和 https://github.com/jd-opensource/xllm-service 上公开发布。


# 1 Introduction

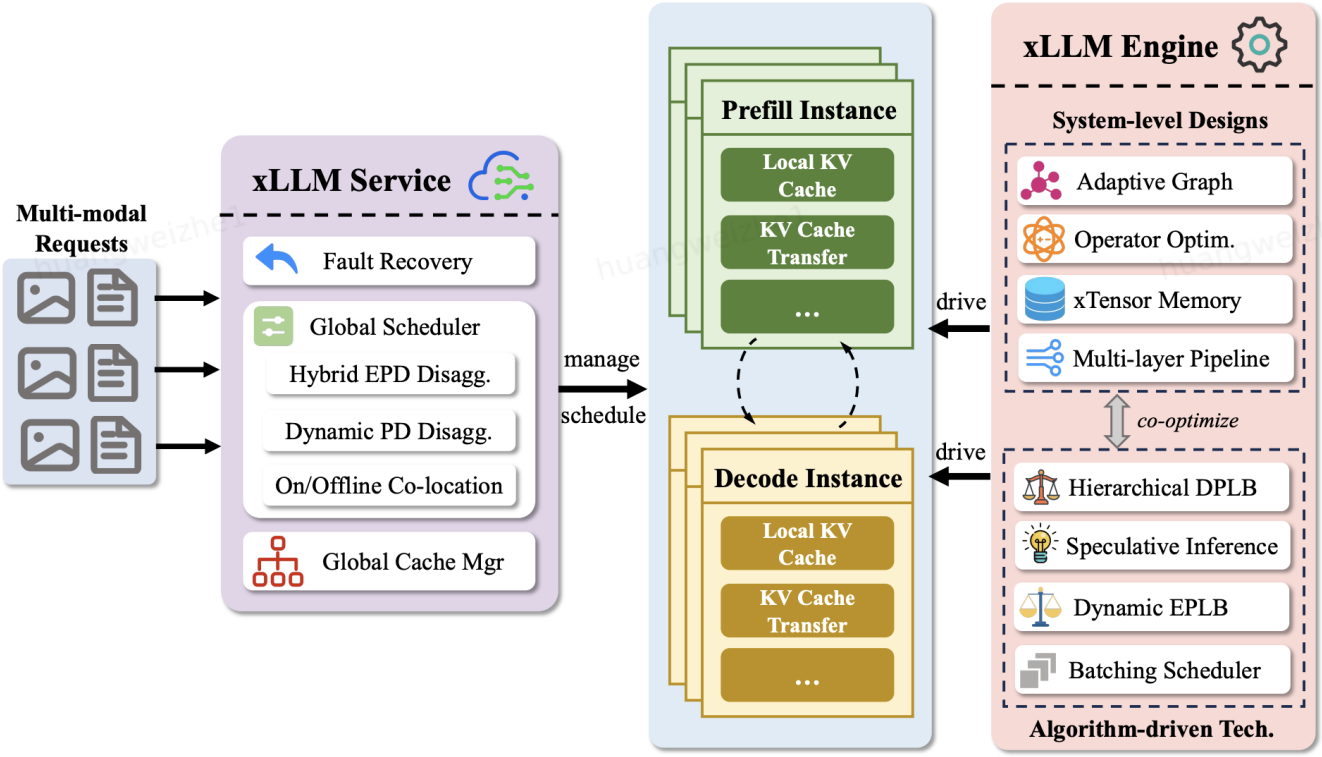


图 1：xLLM 功能概述，具有服务-引擎解耦架构。

近年来，参数规模从数十亿到数万亿的大型语言模型（LLM）（GPT、Claude、DeepSeek、LLaMA 等）在自然语言处理和多模态交互领域取得了突破性进展，推动了业界对高效推理引擎和服务系统的迫切需求。这些模型正在快速部署到智能客服、实时推荐、内容生成 等核心业务场景中。然而，如何降低模型推理成本并提升计算效率仍然是大规模商业服务的关键挑战。

当前主流的LLM推理框架（vLLM 、SGLang 、TensorRT-LLM 等）在企业级服务场景中面临四个关键挑战：(i) 首先，在混合部署的推理集群中，在线推理请求呈现出明显的潮汐特性。当前的调度系统无法满足在线服务的服务级别目标（SLO），同时又无法充分利用在线服务的空闲时间提高离线任务的吞吐量。(ii)其次，现有的预填充-解码（PD）分离架构假设两个阶段的资源分配为静态，无法适应实际应用中动态变化的请求负载（即输入/输出长度波动），导致AI加速器利用率低，并增加了违反SLO的风险。(iii)第三，缺乏有效服务多模态请求（例如图像、语音和文本输入）的策略，包括编码阶段的并行处理以及相应的细粒度资源分配等。(iiii)第四，随着推理集群规模的扩大，确保节点或实例的快速故障检测和服务恢复对于维护推理服务的稳定性至关重要。

不断发展的计算范式也给现有的LLM推理引擎带来了巨大的性能挑战：(i)首先，它们难以充分利用异构AI加速器的计算单元[18, 19]。(ii) 其次，混合专家（MoE）架构[21, 22]中的All-to-All通信开销[20]和专家负载不平衡限制了系统的可扩展性。(iii) 第三，随着模型上下文窗口的不断扩大，高效的kv cahce显存管理对推理性能至关重要[23]。(iiii)第四，由于推理请求的不可预测性，传统的静态调度和运算符策略在数据并行 (DP) 中难以有效地平衡计算单元之间的工作负载。

为了应对上述挑战，我们提出了**xLLM**，一种采用服务-引擎分离设计的高效且智能的LLM推理架构。 xLLM 通过以下创新实现对企业级推理的高效支持：xLLM 通过以下创新实现对企业级推理的高效支持：在*服务层*，xLLM 在 1）在线/离线请求的统一弹性调度、2）工作负载自适应的动态 PD 分离架构、3）新颖的多模态请求的编码-预填充-解码 (EPD) 分离、以及 4）全局 KV

Cache 管理和容错框架方面取得了突破性成果；在*引擎层*，xLLM 提升了“通信-计算-存储”全栈的资源效率，包括 1）多层流水线执行机制、2）高效计算和内存优化、3）智能算法设计。

具体来说，**xLLM-Service** 采用抢占式执行的方式调度在线请求，并以尽力而为的方式调度离线请求，从而在严格保证在线服务 SLO 的同时，最大限度地提高资源利用率。为了克服静态 PD 配置的固有局限性，xLLM 引入了一个自适应调度器，可以动态调整每个请求的 PD 实例比例，并通过监控关键指标（例如首次令牌时间 (TTFT) 和每输出令牌时间 (TPOT) [24]）来支持实例的快速角色转换。对于多模态视觉请求，xLLM 会根据预分析自动选择最佳的 EPD 阶段分离策略，以在吞吐量和延迟之间实现最佳性能平衡。为了处理硬件问题、网络故障或软件错误 [25] 等故障，xLLM 会收集节点错误信息，评估中断请求的键值重新计算或迁移成本，并做出最佳的全局重新调度决策。在多个实例中，xLLM 支持混合存储架构中的键值卸载和路由迁移，从而提升键值存储容量和缓存命中率。

**xLLM-Engine** 采用多层执行流水线，并结合硬件优化：(i) 在框架调度层，实现异步 CPU 加速器调度，以最大限度地减少计算空闲时间；(ii) 在模型图层，利用双流微批并行，将计算与通信重叠；(iii) 在算子层，实现内核计算与内存访问的重叠。在计算效率方面，xLLM-Engine 自动将小内核融合成统一的计算图，并通过一次操作将它们调度到加速器，从而显著降低内核启动开销。在内存管理效率方面，xLLM-Engine 引入了 xTensor 内存管理方案，该方案采用“逻辑连续、物理离散”的键值缓存存储结构，解决了内存连续性要求与动态分配需求之间的冲突。为了进一步提升性能，xLLM-Engine 引入了自适应推测解码机制、基于冗余专家的EP负载均衡算法以及分层DP负载均衡算法。此外，xLLM-Engine 还提供针对特定场景的优化，例如针对京东核心业务之一的生成式推荐，通过host-device操作重叠实现了 23% 的性能提升。

综上所述，在 xLLM 的服务-引擎分离式架构设计中，我们的贡献主要包括：

**xLLM 智能服务能力**

- xLLM设计了统一的在线/离线工作负载调度算法 (§3.1)。
- xLLM实现了工作负载自适应的 PD 分离策略，以应对流量负载和请求输入/输出长度快速变化的场景 (§3.2)。
- xLLM提出了一种针对多模态请求的混合 EPD 分离策略，实现了跨不同阶段的智能资源分配 (§3.3)。
- xLLM利用多级 KV Cache 管理和全局 KV Cache 路由策略来扩展 KV Cache 容量并提高缓存命中率 (§3.4)。
- xLLM设计了多节点容错架构，以确保服务的高可用性 (§3.5)。

**xLLM 智能引擎能力**

- xLLM通过软硬件协同设计，实现智能高效的推理，从而提升硬件计算效率，包括流水线执行 (§4.1)、图优化 (§4.2) 和内存优化 (§4.3)。
- xLLM通过算法优化（§4.4）来增强推理性能，包括优化的推测解码（§4.4.1）、动态EPLB（§4.4.2）和分层DPLB（§4.4.3）。
- xLLM针对生成式推荐场景优化了在线推理（§4.5）。

# 2 System Overview

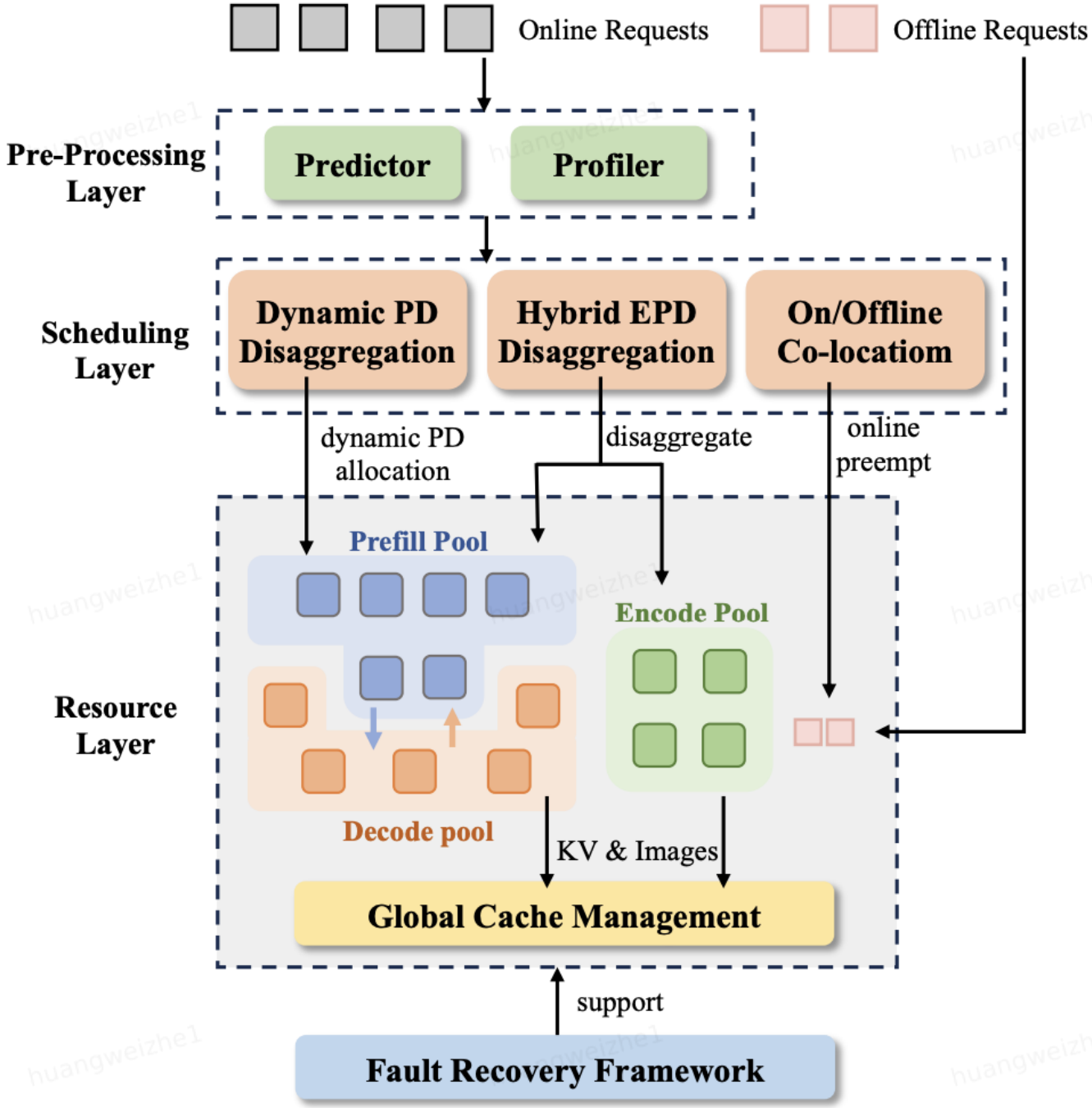


图2：xLLM-Service工作流架构

xLLM 的整体架构如图 1 所示。请求到达后，xLLM-Service 会执行智能调度，将每个请求分配到三个弹性实例池中的一个，并在运行时管理跨这些实例池的实例迁移。xLLM-Engine 随后通过协调系统级和算法级优化来驱动高效的请求推理。

## 2.1 xLLM-Service

图 2 中描述了 xLLM-Service 的工作流程。该系统包含三个主要层：1) 由预测器和分析器组成的预处理层；2) 集成三种策略（动态 PD 分离策略、混合 EPD 分离策略和在线-离线共置策略）的调度层；以及 3) 由三个异构实例池组成的资源层。具体而言，动态 PD 分离策略利用来自预测器的信息，根据两种实例类型的工作负载状态动态地转换 PD 实例。混合 EPD 分离策略利用分析器确定针对多模态请求的最佳 EPD 分离策略。同时，在线-离线共置策略根据请求的在线或离线属性进行调度。键值和图像缓存在分布式实例之间进行卸载和路由，而故障恢复框架则确保

整个服务的高可用性。xLLM系统的服务-引擎分离式框架与流程如图2所示，请求提交后，xLLM服务层负责了请求部署前的预处理以及智能调度，最终被部署至弹性资源池中，底层的容错架构保证了实例的故障发现与快速恢复。

**弹性实例池。** 实例资源被划分为了三个弹性池：

- Prefill实例池：用于处理文本请求的Prefill阶段
- Decode实例池：用于处理文本请求的decode阶段
- Encode实例池：用于处理多模态请求的encode阶段

我们将PD实例池中的实例设计为无状态实例，i.e.，实例无需在资源池间做物理迁移，而是根据实际处理的请求类型实现在PD角色间的灵活切换。

**请求预处理。** xLLM-Service 为在线和离线请求实现了统一的资源管理。在线请求以抢占式和截止期限优先的方式提交，并与尽力而为的离线任务在共享资源池中共同部署。在离线请求的运行时，在线-离线同置策略会根据在线请求的流量特征动态地扩容/缩容离线请求。在线请求提交后，首先会经过预处理层，该层由两个模块组成：

- TTFT Predictor：一个为文本请求构建的 TTFT 预测模型。它通过分析每个预填充实例队列的排队延迟和请求输入长度来评估 SLO 的履行情况，从而指导调度层中动态 PD 分离策略的实例分配。
- EPD Profiler：EPD 分析器：一个用于多模式请求的分析器，使用二分查找来确定最佳部署配置：(1) EPD 分离策略，可从三种方法中选择：EP-D（即编码和预填充阶段聚合执行，解码阶段单独执行）、ED-P 或 E-P-D；(2) 编码阶段的最大批处理大小；(3) 预填充/解码输入的最大令牌数。调度层中的混合 EPD 分离策略将使用为阶段分离和任务调度确定的最佳配置。

**智能调度。** 智能调度层会根据请求的整个生命周期调整资源分配。它包含三种主要的调度策略，适用于各种场景：

- 在线-离线共置策略：此策略实现了一个抢占式调度器，用于管理在线和离线请求。当在线请求的负载达到峰值时，它们会抢占 PD 实例上的部分离线请求。为了将资源移交给离线请求，当 P 实例上的负载下降时（通常比 D 实例更早），P 实例会继续处理离线预填充请求，并将解码离线请求从 D 实例迁移到 P 实例。
- 动态 PD 分离策略：此自适应调度策略负责动态管理 P 和 D 实例的分配。它使用由 TTFT 预测器引导的启发式算法，将请求智能地分配给合适的实例。此外，它通过持续收集计算实例的性能数据来实现反馈机制，从而实现运行时监控和分配决策调整，以保持系统效率。
- 混合 EPD 分离策略：此多模态策略根据 EPD Profiler 搜索到的策略执行三阶段分离。对于 EP-D 分离，融合的 EP 阶段在 P 实例池中执行；对于 ED-P 分离，融合的 ED 阶段在D 实例池中执行；对于 E-P-D 分离，三个阶段分别在三个实例池中执行。此部署还使多模态请求能够受益于动态 PD 分离策略的调整。

**以KV为中心的存储架构。**实例存储采用混合架构（HBM-DRAM-SSD）来缓存键值和镜像令牌。在全局层面，xLLM 借鉴了Mooncake [23] 的理念，并针对性地将其扩展到了国内加速器。跨实例的缓存路由和重用由智能路由策略决定。

**高效容错。**xLLM-Service 的故障恢复框架支持故障检测，并支持从三个弹性池（E、P、D）快速恢复实例。对于故障实例上的请求，该架构管理实例之间image cache的迁移，并自动确定最佳的键值重新计算或迁移策略来处理受影响的KV cache。

## 2.2 xLLM-Engine

在请求执行过程中，xLLM引擎层提供智能计算能力，我们通过以下方式实现了多种计算系统层和算法驱动层的联合推理加速：

**计算系统层**

- 多层流水线执行引擎：在框架层，CPU 任务异步调度，构建具有推理计算能力的流水线，减少计算气泡；在模型层，单个批次被拆分成两个微批次之间的流水线，有效地重叠计算和通信；在算子内核层，操作跨不同的计算单元进行流水线化，从而实现计算和内存访问的重叠。
- 动态输入的图优化：解码阶段的小内核被融合成一个计算图。为了处理可变的序列长度和批次大小，我们对输入维度进行了参数化，并采用多图缓存方案来降低编译开销。
- xTensor 内存管理：它采用“逻辑连续、物理离散”的键值对存储结构。在每次请求生成 token 时按需分配物理内存空间，同时异步预测并智能映射下一个 token 所需的物理页面。当一个请求完成后，现有的物理内存将被重新使用来执行下一个请求。

**算法驱动层**

- 推测解码：xLLM-Engine 集成了优化的推测推理算法，该算法可一次性生成多个 token，从而提升吞吐量 [26]。它通过异步 CPU 处理和减少数据传输等方式进一步优化计算架构。
- EP负载均衡：对于 MoE 模型，xLLM 基于历史专家负载统计数据实现专家权重更新，从而在推理过程中实现有效的动态负载均衡。
- DP负载均衡：对于数据并行部署，xLLM 通过kvcache 感知的实例分配、数据处理单元间请求迁移以及数据处理单元内计算单元分配实现细粒度的负载均衡。

我们将在§3 中详细阐述 xLLM-Service 的详细设计，并在§4 中概述 xLLM-Engine 中的优化。§5 中将对 xLLM 进行全面评估。

# 3 xLLM-Service Designs

## 3.1 在线-离线共置调度策略

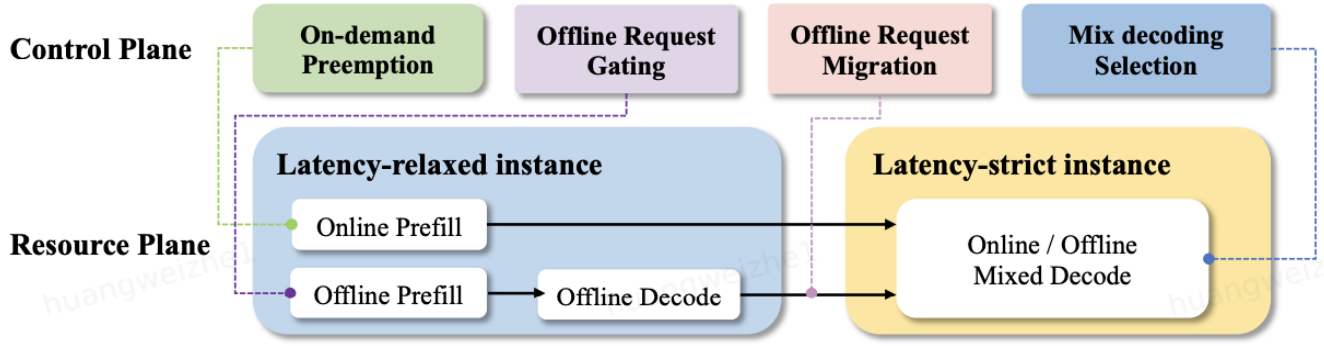


图3：在线-离线共置调度策略

**在线/离线请求特性。**LLM 服务根据其服务模式可分为两类：在线请求和离线请求。在线请求，包括来自聊天机器人 [1, 2, 27]、代码补全 [28–30] 和推荐系统 [31, 32] 的请求，构成了对延迟敏感的工作负载。这些服务必须在请求到达后立即响应，通常通过流输出实时返回每个生成的令牌。因此，它们对 TTFT 或 TPOT 施加了严格的 SLO 要求，以确保令人满意的用户体验。相比之下，离线服务（例如文档分析 [33] 和智能数据注释 [34]）是非实时工作负载，具有最小的延迟限制，因此没有严格的 SLO 要求。

此外，我们观察到在线服务的请求流量通常表现出显著的波动性，包括每小时或每天尺度的潮汐变化以及分钟级间隔的突发 [11, 12]。虽然集群自动扩展 [35] 理论上可以缓解潮汐模式造成的资源利用不足问题，但实例的冷启动延迟（包括模型加载和复杂的初始化）使其无法有效应对快速的流量高峰 [36]，从而增加了违反SLO 的风险。

**在线/离线请求共置部署。** 为了应对这些实际挑战，我们采用了一种混合部署策略，将在线和离线请求共置。在这样的系统中，离线请求可以在在线流量的非高峰时段利用空闲资源。相反，在在线流量高峰期间，离线任务可以被抢占，因为它们不受严格的 SLO 约束。这种方法显著提高了总体资源利用率，并减少了流量低谷期间的空闲情况。尽管最近的一些研究 [37–39] 也尝试共置离线请求，但它们尚未探索多实例场景，尤其是在PD 分离的框架下。

事实上，PD 分离架构已展现出卓越的延迟性能，并日益成为业界主流的设计范式。然而，将在线-离线共置直接应用于 PD 分离系统会引入一个关键的 PD 负载不平衡问题：(1) 此类系统要求预填充阶段和解码阶段之间的负载比与其各自的资源分配比保持一致；否则，其中一个阶段可能成为瓶颈，导致另一个阶段阻塞或资源利用不足。(2) 此外，由于这些 PD 负载变化表现出与在线流量模式类似的高波动性和突发性，现有的 PD 分离技术难以有效应对。

**延迟受限的解耦架构。** 我们重新思考了在线-离线混合部署下的PD分离架构的设计。PD分离架构的延迟优势本质上源于延迟约束的分离：解码阶段对每步延迟高度敏感，并且不会被长时间操作阻塞，因此需要与预填充阶段解耦。受此启发，我们提出了一种延迟受限的解耦架构，如图3所示。该设计将集群资源视为两个池：一个延迟宽松的池（对应于原始预填充实例）和一个延迟严格池（对应于原始解码实例）。然后，所有任务将根据其固有的延迟特性和需求重新分配到两个资源池之一。在该架构下，离线请求的解码阶段可以在任一资源池中执行。这种灵活性使我们能够动态调整两个池之间的负载比，从而最大限度地提高集群的整体资源利用率。

然而，它也带来了额外两个挑战：（1）复杂的调度空间：由于离线请求的解码可以在任何类型的实例上执行，因此如何利用这种灵活性来设计新的智能调度仍然未知。（2）严格的SLO保证：离线请求的执行会消耗资源，其预填充阶段可能会阻塞新到达的在线请求，而其在延迟严格的节点上的解码阶段可能会降低整体响应速度。这两种情况都会影响在线请求的SLO满足度。

**解决方案 1 - 性能瓶颈分析。** 我们基于 Roofline 模型 [40] 和在线因子学习构建了一个 LLM 推理性能模型。该模型旨在预测预填充和解码阶段的延迟、计算利用率和内存利用率。由于在延迟严格型实例上执行的解码操作通常占工作负载的很大比例，并且对性能高度敏感，因此我们将平衡计算和内存资源作为优化目标。通过该模型分析性能瓶颈，我们可以选择更合适的离线请求合并到解码批次中，从而提高资源利用效率。

**解决方案 2 - 高效的抢占机制。** 为了严格确保在线请求的 SLO 不受影响，我们引入了一种抢占机制，允许在线请求抢占离线请求。对于在延迟宽松型节点上运行的离线预填充任务，我们提出了一种模型执行中断技术，该技术可以在可接受的延迟范围内实现抢占，而不会产生额外的模型维护开销。对于在延迟严格节点上运行的解码任务，

我们利用性能模型动态选择解码批处理的请求，确保解码延迟始终满足 SLO 约束。

## 3.2 动态PD分离调度策略

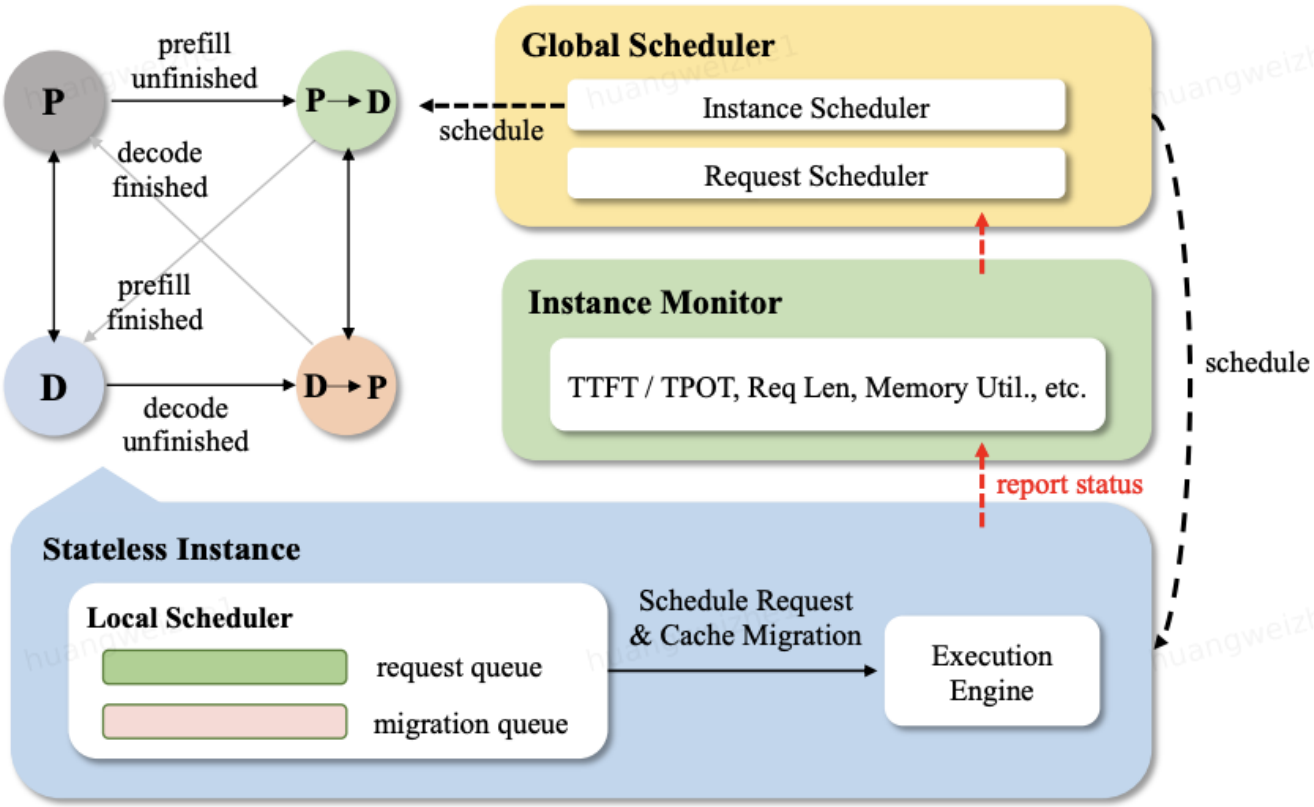


图4：动态PD调度策略

**现有PD分离策略在工作负载波动下的低效性。** 预填充-解码 (PD) 分离推理架构 [13, 24, 23, 41] 将计算实例划分为专用的预填充和解码实例，每个实例负责各自的处理阶段。这种设计减轻了预填充和解码请求之间的干扰，与同地 PD 架构相比，实现了卓越的性能。

然而，我们观察到，大多数采用静态实例划分方案的现有 PD 分离系统存在硬件利用率低和对流量突发的响应能力不足的问题。一方面，现有的基于分析的方法 [13, 24, 23, 41] 通常使用分析或模拟器数据来确定 PD 比率。这些方法仅在请求到达模式和长度分布保持相对稳定时才有效。在工作负载大幅波动的情况下，预先收集的分析数据往往无法准确捕捉实时请求特征，导致预设的负载均衡比例与实际负载需求不匹配。另一方面，当生产工作负载的输入/输出长度发生较大变化时，现有的许多PD 分离架构 [13, 42] 通常采用动态实例类型调整策略 [43]。然而，预填充实例和解码实例之间的在线切换通常涉及多个步骤，包括监控、等待翻转条件以及实例重启，这会带来显著的延迟开销 [15]。为了解决这些限制，我们提出了一种动态 PD 分离策略，该策略可以根据实时工作负载特征自适应地调整资源分配。

**运行时实例监控器。** TTFT 和 TPOT 等性能指标直接反映了实例在预填充和解码阶段处理请求的能力。它们是评估推理系统是否满足 SLO 要求的核心依据。理想情况下，可以采用基于预测的方法来动态评估这些指标。然而，尽管 TTFT 表现出相对可预测的特性（因为其计算时间与输入序列长度的平方成正比，但由于输出令牌长度和传输开销的不确定性，TPOT 难以使用传统的输入/输出特征和集群负载指标准确预测。为了解决这个问题，我们部署了额外的实例监控器来收集每个计算实例的实时性能数据。这些数据包括预填充和解码请求的数量和长度、内存使用情况、

TTFT、TPOT 以及令牌生成间隔等指标。系统可以进一步利用这些实时性能指标来动态评估实例负载并调整调度策略。

**无状态实例和弹性池。**如图 3 所示，动态 PD 分离策略采用无状态实例和弹性实例池的设计，以实现实例角色的快速动态切换。首先，它将预填充或解码阶段视为请求属性，而不是实例属性。实例被设计为无状态，允许每个实例同时处理预填充和解码请求。此外，为了方便管理多个实例，我们进一步将 PD 池扩展为四个弹性实例池（即 P、D、P→D、D→P），如 §2.1 所述。当翻转实例（切换其角色）时，我们只需将其从原始池中移除并移动到新池中即可。这实现了零等待时间的实例调度，避免了传统系统中实例重启或模型重新加载的开销。

**SLO 感知的实例角色切换。**实例调度策略严格基于 SLO 目标进行动态调整：在预填充阶段，如果预测现有实例无法满足 TTFT 要求，则触发解码实例的转换；而在解码阶段，当出现资源短缺、平均令牌生成间隔超过 TPOT 阈值或预填充实例空闲时，将启动预填充实例向解码实例的转换，以应对突发流量激增。具体而言，当解码实例重新分配给预填充实例时，调度器优先从 P→D 池中选择负载最轻（即正在处理的令牌最少）的实例进行角色转换，并始终确保至少有两个解码实例可用；相反，当预填充实例重新分配给解码实例时，调度器优先从 D→P 池进行调度，以避免局部过载并最大化资源利用率。

**SLO 感知的请求调度。**请求调度方案采用两级架构：

- 全局请求调度器：调度器采用贪婪策略，优先处理负载最轻的请求，同时严格遵守 SLO 约束。对于预填充请求，调度器首先评估预填充池中每个实例的预计排队延迟，选择延迟最小的候选实例，然后调用 TTFT 预测模型进行验证：如果将请求分配给该实例后，预计的 TTFT 仍然能够满足 SLO 要求，则立即分配请求；否则，继续在 D→P 池中查找合适的实例。如果没有实例能够满足 TTFT 要求，则触发实例调度机制，从解码端分配资源。对于解码请求，调度器优先让原预填充实例继续处理（以避免 KV Cache 传输开销）；其次，选择解码池中运行令牌最少的实例，并检查其当前批次的令牌总数是否低于预先分析确定的内存容量上限和计算吞吐量上限，以确保新请求不会导致TPOT超标。
- 本地请求调度器：每个实例内部采用精细化的队列管理策略。KV Cache传输事件被放置在独立的迁移队列中，并按照FCFS原则顺序处理；对于前向请求，采用创新的Chunked Prefill[45]和Continuous Batching[8]相结合的方案：在确保解码请求优先进入运行批次的前提下，利用剩余的计算资源并行处理Chunked Prefill请求[46]。

## 3.3 混合EPD分离调度策略

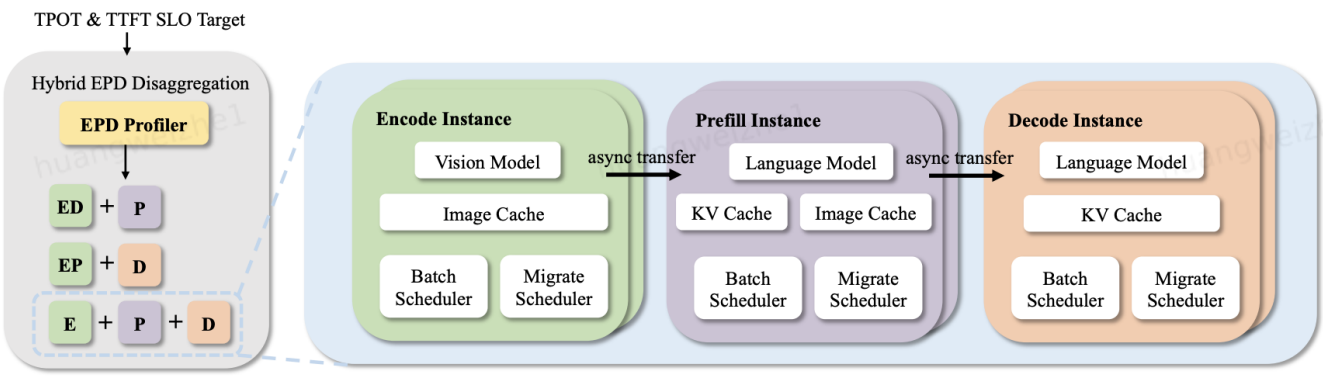

图5：混合EPD分离调度框架图

**多模态推理的挑战。** 多模态大型语言模型 (MLLM) [47–52] 的推理过程通常包含三个阶段：图像编码阶段（用于提取图像特征）、预填充阶段（用于对图像和文本提示进行编码，将其输入语言模型生成第一个输出标记，并缓存中间状态）以及解码阶段（用于根据缓存数据迭代生成后续标记）。现有的主流推理引擎，例如 vLLM [8]、Text-Generation-Inference [53]、SGLang [9] 和 DistServe [13] 都是为 LLM 量身定制的，因此在处理 MLLM 的推理任务时面临以下挑战：

- 并行性不足：例如，视觉模型和语言模型的推理可以并行执行，从而提高计算资源的利用率。然而，大多数现有的推理引擎 [54] 采用串行策略，未能有效利用请求间的并行性。
- 粗粒度调度：解码阶段是内存密集型任务，适合采用批处理来提高吞吐量 [55]；预填充阶段是计算密集型任务，因此，为了平衡延迟和吞吐量 [56]，最好将分块预填充 [45] 与解码阶段结合使用。编码阶段的计算和内存访问开销介于两者之间，也可以从独立的调度和批处理中获益。然而，现有引擎以混合方式处理编码和预填充，无法执行特定阶段的批处理，并且缺乏对分块预填充的支持。粗粒度调度使得执行时间的精细控制变得困难。
- 解耦策略选择：现有架构（例如 DistServe [13]）通过解耦预填充和解码阶段来减少资源干扰。然而，在多模态推理中，编码和预填充共同影响时间延迟 (TTFT)，而预填充和解码共同决定时间延迟 (TPOT)。在不同负载下，如何选择最优解耦策略仍然是一个挑战[57]。例如，E+P+D、EP+D、ED+P等策略在不同任务下的性能尚未得到评估，需要深入分析来指导系统设计。

**双流并行。** 如图 5 所示，我们采用双流并行模式，其中视觉模型和语言模型被分配到不同的执行流：视觉流

专用于执行计算密集型的图像编码任务，而语言流则处理语言生成的预填充和解码操作。通过将工作负载隔离到不同的流中，我们的系统实现了来自不同请求的异构阶段的并发执行。

**三阶段分离。**为了解决多模态推理各阶段任务在计算和内存访问特性上的差异，我们提出了一种阶段感知的调度策略。在一个实例中，请求被细分为三个阶段：编码、预填充和解码。每个阶段分别执行批处理和调度优化：

- 优化批处理：图像编码的最大批大小和语言模型的令牌预算根据用户的服务水平目标 (SLO) 设置。具体来说，在系统启动期间，我们使用二分搜索法来分析最大编码批次大小和模型的令牌预算，以确保每次迭代中后续批次处理任务的执行时间小于TPOT SLO。在每次迭代中，我们 (i) 首先将所有正在运行的解码请求添加到当前批次；(ii)然后检查是否存在已部分计算的分块预填充任务。如果存在，则将其添加到批次中；(iii) 如果没有分块预填充任务，则检查是否存在待处理的编码任务，并添加它们（如果存在）。此方法旨在尽快完成预填充阶段的请求，从而减少其 TTFT。仅当预填充阶段没有请求时，才会处理新请求的编码阶段。
- 阶段感知调度：为了解决多模态推理任务解耦策略选择难题，xLLM-Service 提出了一种创新的多实例协同推理框架——混合编码-预填充-解码解聚架构。在该架构中，每个实例仅执行三个阶段中的部分子任务，而其余阶段则通过将请求迁移到其他实例来处理，从而避免资源浪费和干扰。在运行时，我们根据历史请求分析（通过EPD Profiler），分析自动选择最优解耦策略，实现吞吐量和延迟之间的动态权衡。与传统的将实例绑定到单阶段或全阶段任务的方法相比，该架构在满足 SLO 要求的同时，显著提升了系统整体处理能力和资源利用效率。

## 3.4 全局KV Cache管理

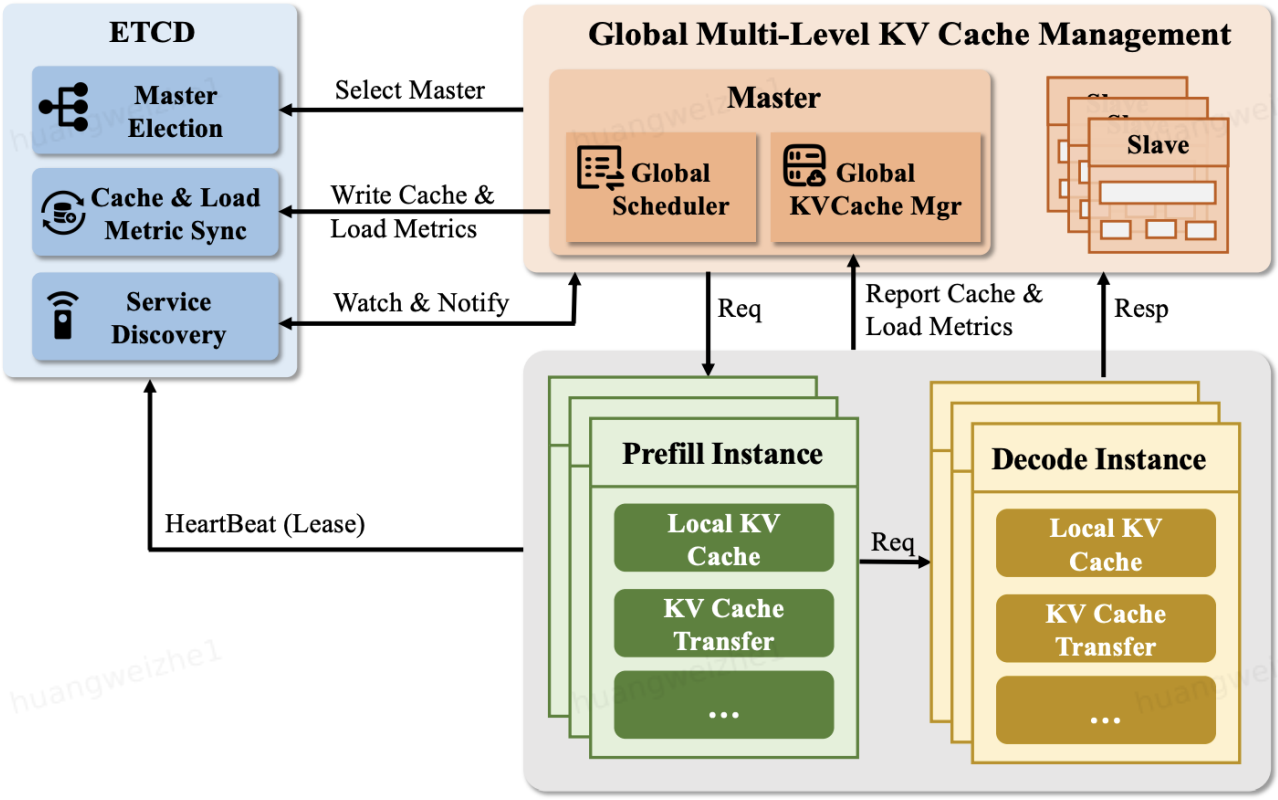


图6：全局多级KV Cache管理

在LLM的解码阶段，后续的token会逐个自回归地生成。虽然每步计算成本相对较低，但需要频繁访问历史KV缓存，这使得内存带宽成为主要瓶颈。随着模型规模的扩大和上下文窗口的增大，KV缓存的内存消耗呈指数级增长。例如，一个包含128K个token的上下文可能消耗超过40GB的内存，这严重影响了单GPU设备的内存资源[3]。尽管目前主流的优化方案，例如vLLM[8]和FasterTransformer[58]，在单实例环境下取得了显著进展，但仍有许多关键问题尚未解决。在长上下文场景下，预填充阶段所需的时间急剧增加，而解码阶段则面临着激烈的内存带宽竞争。为了满足严格的 SLO 要求（例如，TTFT < 2 秒，TBT < 100 毫秒），一个实例通常需要预留过多的资源，而不是利用其他实例的资源。这严重限制了集群的整体资源利用率。为了应对这些挑战，我们提出了一个全局多层的 KV 缓存管理系统，该系统具有内存计算集成架构，如图 6 所示。

**分布式存储和 KV 传输。**我们采用 Mooncake Store [23]（一个以 KV 缓存为中心的存储引擎）作为 xLLM KV 缓存的底层存储基础架构，并使用 Mooncake Transfer Engine 作为传输组件。Mooncake Store 利用条带化和并行 I/O 技术来充分利用多个网卡的聚合带宽。它提供三种持久化策略——Eager、Lazy 和 None，以满足各种场景下的数据持久性需求。此外，Mooncake Store 提供灵活的对象存储能力，支持多副本和最终一致性，有效缓解热点访问压力。作为系统的核心传输引擎，Mooncake Transfer Engine [23] 根据数据位置自动选择最优传输路径，并通过统一的 Segment 和 BatchTransfer 接口抽象底层复杂性。

**多级 KV 缓存管理。**在全局层面，系统采用 ETCD [59] 作为元数据服务中间件，实现集群服务注册、负载信息同步和全局缓存状态管理。每个计算实例维护一个本地多级 KV 缓存池（HBM > DRAM > SSD），并遵循严格的一致性规则：“如果数据驻留在 HBM 中，则它也必须存在于 DRAM 中”。具体而言，当本地发生涉及 KV 缓存（包括前缀缓存）加载和卸载的操作时，这些

操作事件会定期聚合，并通过 ETCD 心跳机制传输到 xLLM-Service，从而实现统一的全局监控和管理。

**KV 缓存感知调度。**在调度策略方面，系统实现了基于 KV 缓存利用率的决策机制，该机制包含三个关键步骤：(1) 前缀匹配检测：通过前缀匹配分析计算每个候选节点的 KV 缓存重用率；（2）性能预估：根据当前负载情况和缓存命中率，预估不同节点的预期延迟；（3）最优节点选择：确定请求处理整体性能最优的节点，从而实现键值缓存的动态卸载和迁移。

## 3.5 快速错误恢复架构

当前的故障处理机制主要用于模型训练[60–62]，由于大型模型推理对延迟的要求较高，因此无法直接应用于此类场景。具体而言，现有方法主要采用“先检查点，后恢复”的方法[63–65]进行故障处理，该方法定期将模型数据作为检查点存储在分布式存储中，并在发生故障后重新加载最新的检查点[66]。随着模型参数的增加，存储和加载的开销会逐渐增加。由于训练阶段对延迟没有严格的要求[67]，这在可接受的范围内；但在推理阶段，这可能会导致故障实例上所有高优先级请求超时，从而造成严重损失。

为了解决这个问题，我们提出了一种专门针对大型模型推理的高效故障转移架构，并在快速请求迁移和快速实例恢复两个方面进行了有针对性的优化。快速请求迁移主要通过高效的KV缓存快速恢复策略来保证高优先级任务的SLO，而快速实例恢复则通过高效的计算和通信屏蔽来实现低开销的恢复。通过以上优化方法，我们实现了快速故障恢复，大大降低了故障带来的性能和经济损失。

# 4 xLLM-Engine

## 4.1 **多层流水线执行引擎**

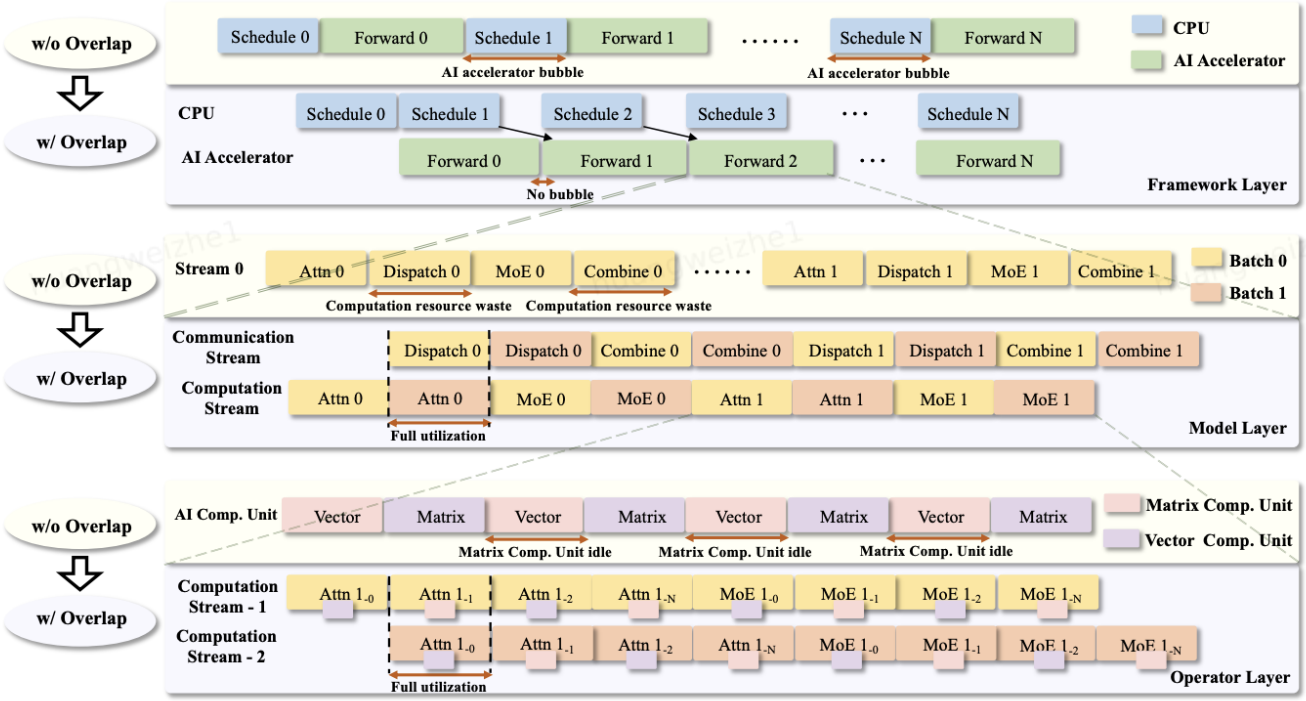


图7：多层流水线执行引擎

传统的自回归LLM推理依赖于单流顺序执行。这种传统方法未能充分利用现代硬件的并行能力，在异构计算环境中常常导致计算资源利用不足、通信停滞和级联阻塞延迟[68, 69]。如图7所示，系统性能受到三个主要低效因素的阻碍：(1) CPU-加速器依赖性：在框架层面，CPU和加速器之间的严格依赖关系迫使加速器在任务调度和数据处理阶段保持空闲状态，从而产生严重的“计算泡沫”[68, 3]。(2) 通信延迟：在分布式工作负载和复杂模型层中，通信延迟都会中断连续计算，从而阻碍可用硬件资源的充分利用[69, 70]。 (3) 架构瓶颈：在某些 AI 加速器上，专注于计算的计算单元（张量核心）和通用计算单元（矢量核心）缺乏共享的高级缓存（例如 L1 或 SRAM）。这种架构上的分离导致它们之间需要额外的数据传输，从而增加了延迟并导致专用计算单元的利用率不足。

为了应对这些挑战，我们提出了一种三层异步流水线并行设计，涵盖框架调度层、模型图层和算子层，如图 7 所示，旨在最大限度地提高硬件效率。

**框架层调度-执行重叠。**在典型的顺序执行过程中，加速器处于空闲状态，而 CPU 则执行调度，为下一个计算周期准备输入数据批次。这种串行依赖性会导致显著的延迟，并导致加速器利用率不足。

为了消除这一瓶颈，我们引入了一种异步流水线调度机制，将 CPU 调度与加速器执行解耦。其核心思想是将这两个阶段重叠：加速器执行当前批次的前向传播时，CPU 同时组装该批次并为下一个批次准备元数据。这种并行工作流有效地隐藏了 CPU 端调度和数据准备的延迟。该过程如下：(1) 加速器计算：加速器忙于执行当前迭代的前向传播，最终将产生输出。(2) CPU 调度：CPU 无需等待，立即开始为下一个迭代调度批次。为此，它使用一组占位符标记来代替尚未计算的输出。这使得所有 CPU 端的批处理准备工作能够与加速器的计算同时进行。(3) 无缝过渡：一旦加速器完成前向传递并生成实际输出，快速替换操作就会将占位符标记与实际生成的标记进行交换。由于所有调度都已完成，加速器可以以最小的延迟开始下一个前向传递。

通过将串行的“准备-计算”序列转换为并行流水线，我们的方法有效地将调度延迟隐藏在加速器的计算时间之后。这种方法消除了加速器执行时间线中的空闲“气泡”，从而最大限度地提高了硬件利用率，并提高了整体吞吐量。

**模型层计算-通信重叠。**最大化计算和通信之间的重叠是近期 LLM 框架 [3, 71] 的一项关键设计原则。这一策略对于隐藏数据传输的显著延迟并实现高硬件利用率至关重要。然而，架构限制可能会使这一目标变得具有挑战性。例如，在典型的 GPU 上，流多处理器 (SM) 中的计算单元（Tensor Core）与通用单元（CUDA Core）紧密耦合。因此，当 SM 被分配给与通信相关的任务时，其强大的 Tensor Core 通常会处于空闲状态，从而浪费计算潜力。相比之下，我们的目标加速器架构允许更灵活、独立地分配其立方体（矩阵）和向量（通用）单元。我们利用这种灵活性来应对上述挑战，引入了一种双流流水线架构，该架构使用微批次拆分来有效地重叠计算和通信。

具体来说，我们的架构由一个用于计算密集型任务（注意力机制、ExpertForward 等）的计算流和一个用于数据分发和收集任务（移动端调度和合并）的通信流组成。为了实现流水线化并最大化整体流水线吞吐量，我们将宏批次 划分为 个微批次[68] { 1，2,⋯， }。这两个流异步执行不同微批次的任务。一方面，我们的调度策略动态地确定微批次的最佳执行顺序。另一方面，我们的资源调度策略自适应地为通信流和计算流分别分配适当数量的AI核心（矢量核心和立方体核心）。图 7 展示了 = 2 的情况，其中通信流对微批次 执行调度操作，而计算流对前一个微批次 -1 执行 ExpertForward 传递。

**算子级矩阵-向量单元重叠。**在异构AI加速器上，矩阵和向量计算单元的串行调度通常会导致严重的资源利用率不足，因为其中一类单元处于空闲状态，而另一类单元处于活动状态。这种低效率促使我们提出一种算子级矩阵-向量重叠策略，以充分利用硬件的并行能力。然而，在没有系统资源协调机制的情况下，通过粗粒度并行调度进行简单的实现也存在问题。这种方法经常导致对有限计算单元的无序争用，造成资源碎片化和访问冲突，最终降低整体性能。

为了应对这些挑战，我们提出了一种基于实时负载监控的计算单元（立方体单元和向量单元）动态资源分配机制。该机制通过动态自适应地为每个并发算子分配所需的精确资源类型和数量，实现了跨异构计算单元的深度流水线。通过这种方式，它可以缓解资源争用，并确保并行算子在高度重叠的时间窗口内执行，从而实现精确的计算重叠并最大化硬件利用率。

我们将动态资源分配表述为一个优化问题。设$\mathcal{C}$和$\mathcal{V}$分别表示矩阵算子和向量算子的集合。设 为分配给i_{th}矩阵算子（ ∈ $\mathcal{C}$）的矩阵单元（Cube）的数量，设 为分配给j_{th}向量算子（ ∈ $\mathcal{C}$）的向量单元（Vector）的数量。为简单起见，我们假设所有算子同时开始执行，不考虑数据依赖性或通信延迟。基于每个算子的已知计算负载，我们的机制寻求一种最优的资源分配方式，使任意两个算子之间的执行时间差异最小化。这个目标，我们称之为对齐损失 ($\mathcal{L}_{align}$)，它有效地同步了所有并行内核的完成时间。优化问题定义为：

$$\text{argmin}_{x_i,y_j} L_{\text{align}} = \max_{i\in\mathcal{C},\, j\in\mathcal{V}} |T_i - T_j|$$

$$T_i = \frac{W_i}{\gamma_{\text{Cube}} \cdot x_i}, \quad T_j = \frac{W_j}{\gamma_{\text{Vector}} \cdot y_j}$$

$$\sum_{i\in\mathcal{C}} x_i \le N_{\text{Cube}}, \quad \sum_{j\in\mathcal{V}} y_j \le N_{\text{Vector}}$$

其中 为算子执行时间， 为计算工作量， 和 为单位峰值性能， 和 是可用的矩阵和向量单元总数。此优化目标的含义是最小化任意一对并发矩阵和向量运算符之间的最大执行时间差异。

## 4.2 自适应图模式

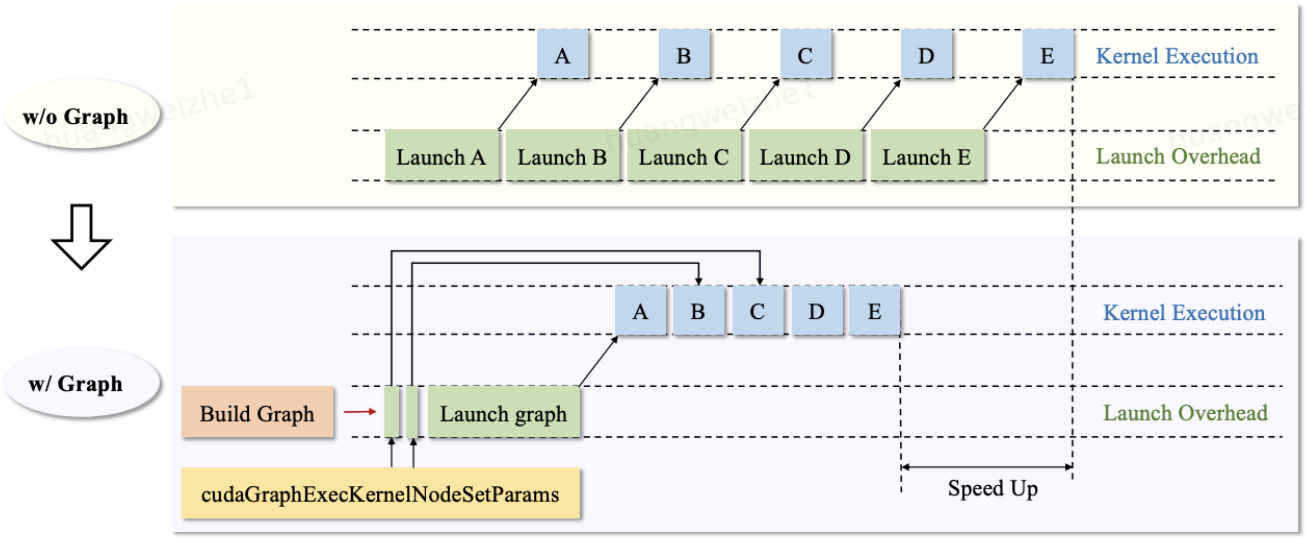


图8：eager模式和graph模式比较

| Method | Eager Mode | Full Graph Mode | Partial Graph Mode |
|---|---|---|---|
| Compilation Times | 0 | 1 | *M* |
| Low Launch Overhead | ✗ | ✓ | ✓ |
| Low Memory Usage | ✓ | ✗ | ✓/✗ |
| High Flexibility | ✓ | ✗ | ✓ |

表 1：不同形状处理方案的比较。我们的自适应图形模式会根据动态形状动态选择最合适的模式。

LLM 推理部署的性能通常会受到主机端 CPU 开销的阻碍，尤其是在计算图由许多细粒度算子组成时 [72]。这种瓶颈表现为显著的 CPU-加速器同步延迟（由于频繁启动内核，每次调用的延迟为 5 ~ 50 ），以及由于这些间歇性算子执行之间的空闲加速器周期导致硬件利用率不理想。主流 AI 加速器，例如 NVIDIA GPU（带有 CUDAGraph）[73] 和 Ascend NPU（带有 ACLGraph）[74]，都采用计算图来尝试解决上述问题并增强主机端调度性能。如图 8 所示，传统的 Eager Mode 依赖于 CPU提交大量小型密集型任务，导致在加速器上频繁启动小型内核。

相比之下，图形模式允许 CPU 提交一个大型任务，之后加速器以精简的方式在内部执行小内核。这种方法显著减少了启动开销和“加速器泡沫”（空闲周期）。具体来说，图形模式（例如 ACLGraph [74]）包含两个不同的阶段：图形捕获和图形执行。在图形捕获阶段，记录整个计算流程以捕获内核调用序列及其依赖关系，包括内核启动参数。然后，将记录的工作流程预编译为可重放的有向无环图 (DAG) 对象。需要注意的是，在此阶段，任务仅在模型的运行时实例中暂存，并未实际执行。在随后的图形执行阶段，整个图形通过单个 CPU 调用启动。然后，加速器将自主执行预定义的流程，无需 CPU 的实时干预。这有效地将多个单独的内核启动合并为一个单一的图启动。

在此基础上，我们进一步推进了 ACL-Graph [74] 的实际实现，并提出了自适应图模式，以解决包括动态形状自适应、跨多个图的内存重用冲突以及与自定义运算符的兼容性等关键挑战。这使得内核序列、内存复制操作和同步点能够预编译成单个计算图，然后在一次执行中将其调度到 AI 加速器。因此，我们的方法显著降低了内核启动开销，并最大限度地减少了加速器空闲时间，从而最大限度地提高了整体硬件效率。

**通过部分图和维度参数化实现动态shape自适应。**ACL-Graph [74] 在图捕获阶段固定了内核参数和内存大小。然而，在实际场景中，LLM 通常处理具有可变序列长度和批量大小的输入，这使得这种静态捕获特性难以适应动态输入需求。为了解决这个问题，可以采用维度参数化，将关键的动态维度（例如批量大小和序列长度）作为整个图的输入参数，从而增强灵活性。在内存分配和内核配置期间，这些动态参数用于计算实际所需的值；例如，可以使用以下公式计算内存大小： _ = h_ × _ × h _ 。在图启动阶段，实际的批次大小和序列长度将作为参数传递，以确保图能够适应不同的输入维度。

虽然参数化可以处理动态维度的变化，但不同硬件上的某些运算符实现可能不支持动态维度参数化。为了克服这个问题，我们提出了部分图模式：将模型划分为具有简单动态形状的模块（例如 FFN 和 LayerNorm，它们仅将 _ 维度作为动态维度）和具有复杂动态形状的模块（例如多头注意力，它涉及多个动态维度，如 h_ 、 _ _ 和 _ _ ）。具有简单动态形状的模块使用部分图模式提取并编译执行，而具有复杂动态形状的模块则以 Eager 模式执行。我们的自适应图模式会根据当前输入形状动态选择最合适的模式。这种智能选择可确保在不同的工作负载条件下实现最佳性能。

表 1 比较了三种动态形状处理模式的主要特性。传统的 Eager 模式不需要预编译，但由于需要启动 N 个内核（其中 为算子数量），运行时会产生较高的调度开销。全图模式通过固定形状，允许单次编译和非常低的启动开销，但其缺乏动态自适应性，限制了其适用性。相比之下，我们提出的参数化和多缓存部分图模式则达到了最佳平衡。它实现了与完整图相当的执行性能，同时保持了高度的灵活性，通过以可控的预编译次数（ ≪ ，其中 是缓存图的数量， 是实际请求的数量）换取单次图启动的高效率。

**高效且可重用的 HBM 内存池。**在计算图捕获过程中，输入、输出和中间张量的具体虚拟地址会被记录下来。为了防止在实际推理请求期间输入张量地址发生变化时可能发生的非法内存访问，我们开发了一个共享的 HBM 内存池。首先，在图初始化期间，预先分配一个足够大且连续的 HBM 内存块作为计算图的内存池。随后，为了进行内部地址管理，计算图会拦截所有张量内存操作，并使用相对于池基地址的偏移量（而非绝对虚拟地址）重新描述这些操作。

在计算图启动之前，用户提供的输入张量会被复制到 HBM 内存池中，并位于为计算图输入预定义的偏移量处。计算图执行完成后，输出张量的数据会被复制到用户指定的输出缓冲区地址。此外，计算图内部的张量也使用池中的固定偏移量进行管理，从而确保安全的重用和高效的内存管理。此托管内存池保证计算图内部使用的所有地址都是已知且安全的，从而能够适应外部地址的变化，并且仅在启动前后进行内存复制的开销。

**集成自定义算子。**将性能关键型自定义算子（例如PageAttention 和 AllReduce）集成到自适应图模式中是一项独特的挑战。这些算子的内部实现通常依赖于 CPU 根据运行时输入执行即时形状计算和内核配置。这种动态行为与计算图的预编译特性相冲突。

为了解决这个问题，我们修改了此类算子的实现。第一步是识别计算图内部具有动态形状依赖性的自定义算子。然后，我们重构这些算子，使其能够直接接受与形状相关的参数（例如，维度大小或循环次数，这些参数只能在运行时确定）作为内核参数。这种方法避免了在图捕获阶段在主机端对这些值进行硬编码。经过这种修改，这些自定义运算符可以被我们的计算图成功捕获。通过利用图的参数化机制，这些内核所需的动态参数可以从图的主要输入参数中派生出来，并在执行过程中传递。该方法实现了必要的自定义运算符在自适应图模式框架内的无缝集成。

## 4.3 高效显存管理

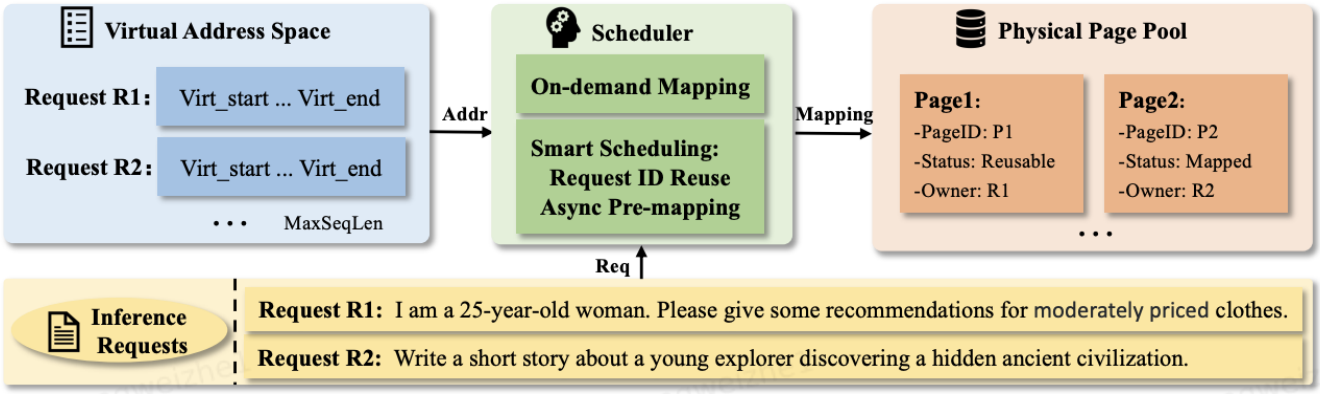


图9：xTensor显存管理框架图

| Features | Contiguous Allocation | Paged Attention | xTensor (Ours) |
|---|---|---|---|
| Efficient Memory Usage | ✗ | ✓ | ✓ |
| Efficient Computation | ✓ | ✗ | ✓ |
| Large Batch Support | ✗ | ✓ | ✓ |
| Operator Development Complexity | ✓ | ✗ | ✓ |

表2：显存管理策略比较

LLM 推理部署面临着内存利用率和计算效率之间的关键平衡，尤其是在自回归生成任务中，高效的 KV Cache 管理已成为关键挑战。传统解决方案分为两类：第一类是连续内存分配方法，该方法在推理前根据最大序列长度静态地预分配内存空间，以确保 KV Cache 的物理连续性。虽然这提高了计算效率，但会导致内存利用率低。第二类是 PagedAttention [75] 方法，该方法通过分页机制支持更大的批处理大小，但频繁访问块表会牺牲计算效率，并且增加的参数会使算子

开发和调试变得复杂。为了应对这一挑战，受操作系统领域虚拟内存管理的启发[76, 77]，我们提出了 xTensor 内存管理方案，该方案采用“逻辑连续、物理离散”的 KV Cache 存储结构，解决了内存连续性和动态分配之间的矛盾，从而兼顾了高计算效率和高内存利用率（如表 2 所示）。

**物理内存页面管理和虚拟内存张量管理。**图9给出了xTensor显存管理框架图。在服务初始化阶段，根据服务可用于KVCache的内存大小，预先分配大量固定大小的离散物理内存页。每个物理页维护三元组状态$\langle \text{PageID}, \text{Status}, \text{OwnerSession} \rangle$，其中PageID表示页面标识符，OwnerSession表示页面所属服务，$\text{Status} \in \{\text{Free}, \text{Allocated}, \text{Mapped}, \text{Reusable}\}$表示空闲、分配、映射、可复用等状态以动态跟踪物理页使用情况。然后为每个推理请求预分配逻辑上连续、地址范围等于MaxSeqLen的虚拟地址空间，该虚拟地址空间在分配时并未实际关联物理页。这种解耦设计能够为算子提供KVCache在连续内存上存储的虚拟视图，从而屏蔽底层物理页的离散性。

**按需映射内存。** 在实际分配过程中，调度器根据实际请求的序列长度动态按需映射物理内存页。当序列生成新token并且KV Cache需要扩展时，调度器从从预分配的物理页池中取出一个或多个空闲页，并将这些物理页映射到请求虚拟地址空间中下一个可用的连续地址位置。由于随着序列增长逐步映射物理内存，对于短序列仅需少量物理内存，从而避免了连续分配策略对短序列仍然需要按最大序列长度预留空间的内存浪费。

当完成内存映射后，内核访问虚拟地址可以自动定位到对应的物理页$\text{phypage}_{idx}$和偏移量$offset$：

$$\text{phypage}_{idx} = \left\lfloor \frac{\text{virt\_addr} - \text{virt\_start}}{\text{page\_size}} \right\rfloor$$

$$\text{offset} = (\text{virt\_addr} - \text{virt\_start}) \mod \text{page\_size}$$

其中$\text{virt_addr}$和$\text{virt_start}$分别表示当前虚拟地址和起始虚拟地址，$\text{page_size}$表示物理页大小。

**低开销分配。**如果在请求结束时立即解除虚拟地址到物理页的映射，新请求到来时需要重新映射物理页，但在NPU上单次Unmap操作开销很大。因此，在高负载情况下，频繁的Map/Unmap操作会成为性能瓶颈，尤其对于大量短期并发请求。我们采用请求ID复用和异步预映射设计，减小Map/Unmap操作带来的延迟。

- **物理页复用。**在请求结束时，不立即释放物理页，将使用的物理页标记为Reusable。当新的请求进入时，如果其所需的KV Cache大小与某Reusable物理页集相匹配，则将该物理页集重新映射到新请求的虚拟地址空间。这样通过快速的虚拟地址重映射，节省了昂贵的Map和Unmap操作，尤为适合请求长度分布集中的场景。
- **异步预映射。**在执行当前token解码步骤时，异步预测并映射下一token所需物理页。由于新token写入KV Cache发生在当前token解码计算之后，因此该设计能够一定程度上隐藏物理页映射操作，从而显著减少页面映射带来的延迟。

**算子适配。**xTensor的物理内存页和虚拟连续内存的解耦设计需要现有算子的适配。一方面，为了适应KV Cache在虚拟地址空间的连续性，算子不再需要block_table参数传入。无论对于注意力算子，还是在Prefill和Decode阶段负责写入KV Cache的write_kvcache算子，仅需将起始虚拟地址及相关偏移量传入，在操作虚拟地址时系统自动关联到对应的物理页上。另一方面，NPU缺乏原生连续KV的FlashMLA算子，只支持针对分页优化的PagedFlashMLA算子。因此通过重构NPU算子库的PagedMLAttention算子，去除Block Table相关的块表查询、跨页边界判断等逻

辑，增加根据传入虚拟地址直接计算的功能，从而实现NPU上支持连续虚拟地址输入的FlashMLA算子。

## 4.4 算法优化

### 4.4.1 优化的推测解码

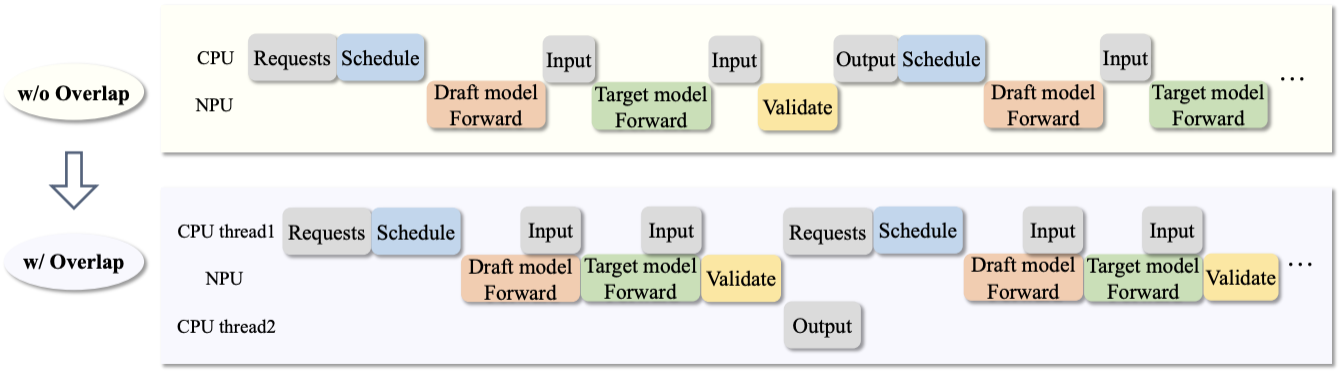


图10：原始和我们优化的推测解码比较

传统的自回归推理需要LLM按顺序逐个生成token，导致计算延迟高且吞吐量受限。推测推理技术通过并行预测候选token序列并快速验证的范式，理论上突破了这一性能瓶颈[82, 83, 26]。然而，在分布式部署环境中，该技术面临一些核心挑战：首先，传统多步推理框架中CPU和加速器之间的同步调度导致计算资源利用率不足，加速器经常处于空闲状态，等待数据准备和处理。其次，现有的注意力机制并未针对推测推理的特点进行优化，导致大量冗余数据移动。为了应对这些挑战，xLLM基于现有框架提出了系统性的优化方案。

**异步解码。**受[83]的启发，xLLM实现了一个轻量级的异步框架。为了优化时序重叠，当加速器执行当前批次的主模型推理时，CPU并行处理上一个批次的输出解码和下一个批次的输入准备，从而尽可能减少CPU和加速器的空闲时间。

**MLA优化。**xLLM专注于改进MLA[84, 3]的计算流程。基于推测推理需要在单个序列上同时处理多个标记的特性，我们对MLA中的自注意力计算进行了深入优化。具体而言，当推测推理m个标记时，MLA的自注意力计算涉及m+1个Q矩阵与同一个K矩阵之间的乘积运算。通过重构计算流程并优化平铺策略，我们有效地降低了Q/K矩阵的数据移动开销。主要优化措施包括：

- 优化 K 矩阵加载时间：通过调整 L1 缓存分配方案，实现多个 Q 矩阵的并行加载，并采用基于滑动窗口的 K 矩阵加载方法，允许 K 矩阵的连续行与多个 Q 矩阵相乘，显著减少了 K 矩阵加载运算次数。
- Q 矩阵缓存驻留机制：由于投机推理场景中有 m+1 个 Q 矩阵，因此将 Q 矩阵移动到 L1 的时间在矩阵乘法过程中占比较大。xLLM 重新设计了计算方案，避免 softmax-V 乘积运算覆盖 L1 缓存中的 Q 矩阵，使 Q 矩阵能够保留在缓存中，显著减少了重复数据移动的开销，从而有效提高了张量核算术逻辑单元的利用率。

### 4.4.2 动态EP负载均衡

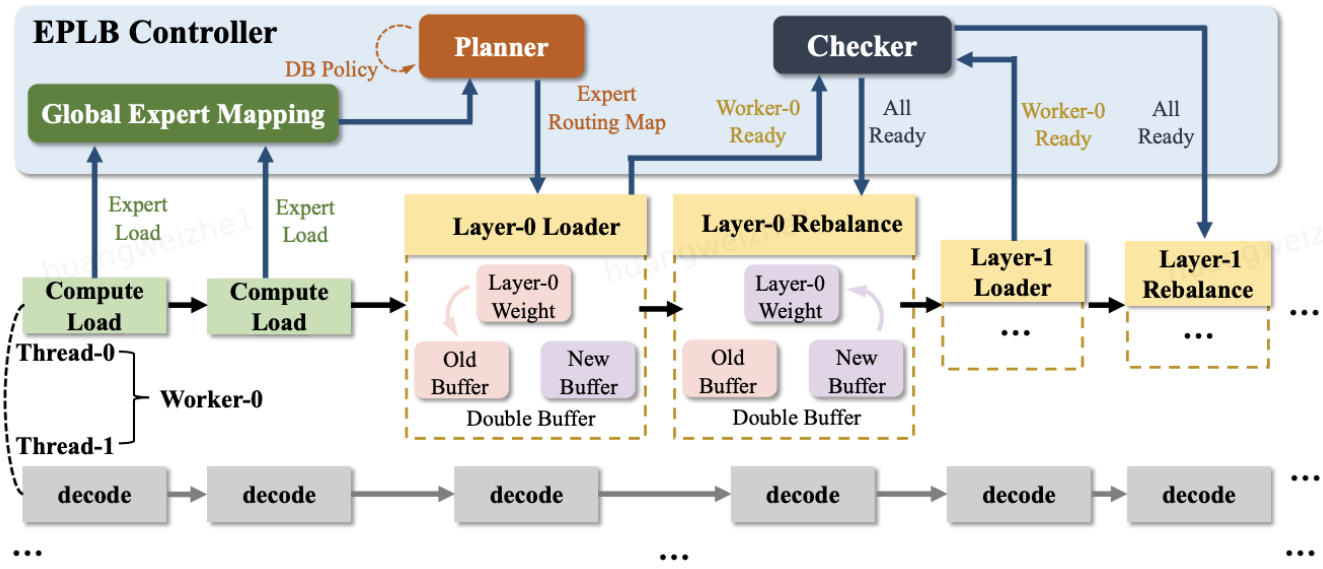


图11：EPLB设计架构

随着混合专家（Mixture of Experts, MoE）模型在生产环境中的大规模应用，其高效的推理能力依赖于专家路由机制，即将输入的tokens分配给不同的专家进行计算。然而，在实际部署中，由于输入数据的分布特性以及模型自身结构的影响，不同专家接收到的tokens数量可能存在显著差异。这种不均衡会导致计算资源利用率低下，部分设备因处理过多tokens而过载，而其他设备则处于空闲状态，影响整体推理效率。

为了优化资源利用，业界提出了专家冗余（Expert Redundancy）策略，即对热点专家（Hot Experts）进行复制，并在多个设备上部署副本，从而分散计算负载。DeepSeek目前采用两种负载均衡策略——分层负载均衡（Group-Limited Load Balancing）和全局负载均衡（Global Load Balancing），分别针对Prefill（预填充）和Decode（解码）阶段的特性进行优化。专家冗余调整（如新增/删除副本）需要消耗额外显存，并可能因权重迁移影响推理延迟，如何高效、平滑地完成是一大挑战，为此我们在Deepseek的基础上做了一些适配性改进，实现了动态的EP负载均衡。

**专家负载统计。**路由器 (Router) 在分发令牌时，会实时记录每个专家的负载状态，并通过模型输出返回统计结果。每个 Worker 会异步启动一个线程，定期汇总专家负载并将其报告给 Master 的 Controller。Controller 根据负载数据计算新的路由表，然后将更新后的路由表分发给每个 Worker。

**权重更新的双缓冲机制。**通常，每个 Worker 都会在每次解码过程中启动一个异步线程来更新单层专家的权重。具体来说，在 Device 完成新权重的准备后，它会主动向 Controller 发送就绪通知。Controller 会在广播更新指令之前验证所有 Worker 节点的权重就绪状态，以确保全局一致性。收到更新指令后，每个 Worker 立即执行权重更新。这里我们采用双缓冲机制，即新旧专家权重分别存储在两个独立的缓冲区中。新的专家权重预加载到空闲的内存空间中，预加载完成后进行地址切换，实现专家的无感知更新。

### 4.4.3 分层DP负载均衡

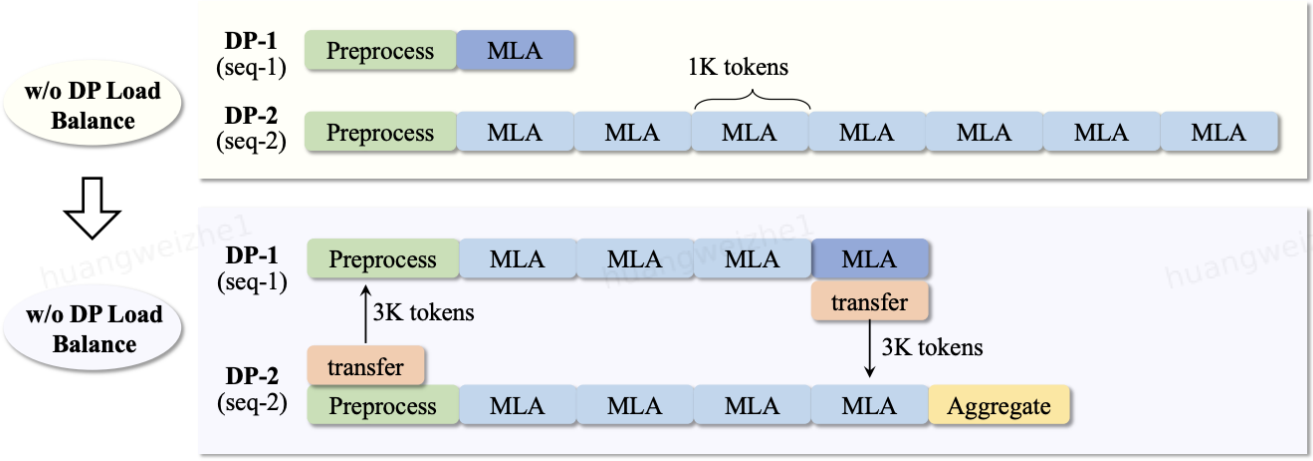


图12：MLA块粒度的DP组间负载迁移

在典型的 MoE 模型架构（例如 DeepSeek）中，MoE 层采用专家并行 (EP)，而注意力层采用数据并行 (DP)。该模型要求所有 DP 组在启动 MoE 的全对全调度操作之前，在单个推理步骤中同步完成注意力计算。这种同步障碍意味着注意力阶段的总时间由最慢的 DP 组（“落后者”）决定。较快的 DP 组在完成任务后无法立即进入下一阶段；相反，它们被迫进入空闲状态，等待落后者完成。这种等待直接导致计算资源的浪费和整体推理延迟的增加。

在实际场景中，由于推理请求的动态性和不可预测性，不同 DP 组内的负载通常难以平衡。特别是在大规模DP（例如80个组）的解码阶段，DP组之间的负载差异可能达到数万个token，导致快速DP组等待慢速DP组，浪费大约5毫秒的延迟。实验数据表明，尽管DP组之间处理的请求数量差异可能只有4个，但解码阶段对应的KV Cache token数量差异可能高达2万个。实际计算负载与系统需要处理的token总数直接相关，因为这决定了KV Cache的内存占用以及注意力机制中矩阵运算的规模。这进一步表明DP负载均衡是决定系统整体效率的关键因素。

目前主流的框架，如vLLM和SGLang，都采用简单的静态循环调度策略。一旦请求被分配，其后续计算就被固定到特定的 DP 组，无法适应动态负载变化。此外，在硬件层面，某些加速器上类似 MLA 的算子的实现采用了“每个张量计算核心一个请求”的分区策略。这可能导致同一 DP 组内的计算核心因请求长度差异而出现空闲。

为了应对这些挑战，xLLM 提出了一种分层的“纵深防御”策略，以解决不同时间尺度和粒度上的 DP 负载不平衡问题。第一层是预防性的、基于键值缓存 (KV) 的请求调度；第二层通过 DP 组间负载均衡进行宏观修正；第三层通过 DP 组内负载均衡进行微观修正。这种分层设计使系统能够灵活地解决各种性能下降的根本原因，从而构建健壮高效的推理服务。

**第一层：KV 缓存感知调度。**xLLM 的第一层策略通过 KV 缓存感知调度实现请求负载分配。该机制超越了简单的轮询调度方法。当新请求到达时，调度器会全面检查所有 DP 组的状态，并特别关注每个 DP 组中剩余的 KV 缓存容量。然后，它会优先将新请求分配给可用空间最大的组。通过长期在系统层面平衡令牌总负载及其内存占用，该策略可以防止严重的负载不平衡，从而实现智能资源分配。

**第二层：DP 组间工作负载的响应式迁移。**xLLM 的第二层策略通过 DP 组之间的工作负载迁移实现响应式平衡。在解码过程中，所选请求的提示符和 KV 缓存长度不同，会导致 DP 组之间的计算延迟不同。为了解决这个问题，xLLM 的中央调度器会在每一轮推理中评估每个 DP 组的当前计算负载。如果检测到严重的不平衡，它会启动工作负载迁移过程，将任务从过载的组迁移到负

载不足的组。调度器还会确定迁移的粒度——是移动整个批次、单个序列，还是序列的部分 MLA 块。

图 12 以 MLA 块粒度展示了此过程。首先，xLLM 将迁移所需的输入令牌分派到源 DP 组和目标 DP 组。然后，当所有组执行 MLA 预处理操作时，请求的 KV 缓存会异步传输，使通信与计算重叠。接下来，所有组开始注意力计算。负载不足的组优先执行迁移后的注意力任务，使其能够在完成后立即将生成的令牌发送回源组，同时并发处理其自身的原生注意力操作，从而进一步重叠开销。最后，所有 DP 组使用聚合运算符汇聚外部计算结果。

**第三层：细粒度内核级优化。**xLLM 的第三层策略通过 DP 组内的内核级迁移实现细粒度优化。此优化在单个 DP 组的矩阵计算内核内部执行。一方面，在调度过程中，它根据请求的负载重新排序，取代了原有计算核心的循环分配策略。这旨在使分配给每个矩阵计算单元的计算令牌总数尽可能保持一致。另一方面，对于序列极长的请求，

它显式地拆分其计算序列，将长序列请求的部分计算迁移到其他短序列计算单元进行计算。通过这种细粒度的请求拆分，它直接解决了 DP 组内负载不平衡导致的计算核心空闲问题。

## 4.5 生成式推荐

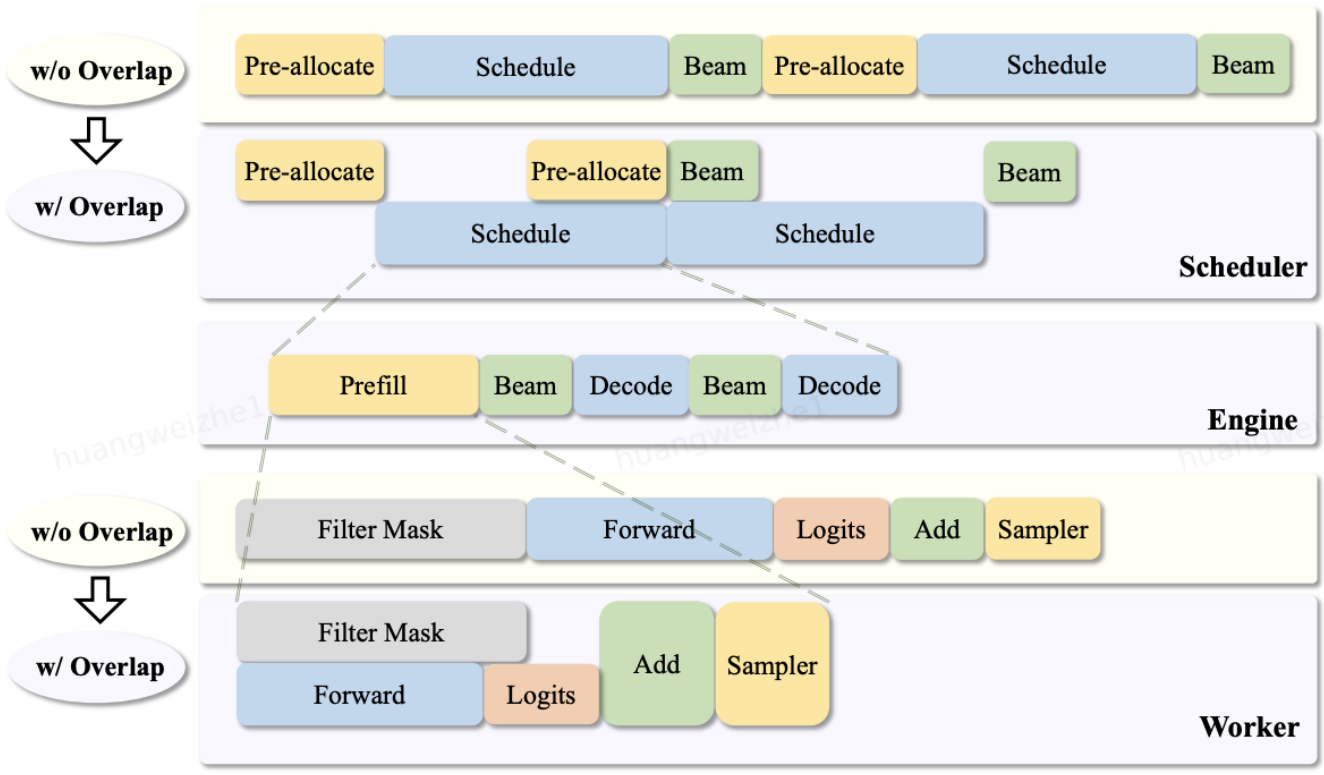


图13：生成式推荐设计

生成模型的最新进展激发了人们对生成推荐的浓厚兴趣 [31, 95–99]。除了增强传统多阶段流程的检索和排序阶段 [99, 95] 之外，单阶段生成推荐框架还可以通过自回归的方式直接生成候选项目列表 [31]。这些框架通常利用集束搜索解码来直接生成多样化的推荐结果，而 xLLM 则针对这些单阶段框架进行了广泛的优化。如图 13 所示，在生成推荐场景中，xLLM 致力于尽可能地重叠主机和设备操作，以提高整体执行效率。当请求到达时，xLLM 会在 CPU 端预先准备前向传递过程中所需的信息。准备完成后，相关任务将交由引擎处理。引擎一次性执行三次前向传递。在此

过程中，中间的Beam搜索是主机操作，其本质决定了它不能与设备操作重叠。为了优化处理流水线，xLLM内部采用了多微批次、多流的方法。在 Worker 执行过程中，生成过滤掩码的 CPU 操作与计算 logits 的设备操作重叠。通过加法运算，在采样器之前生成带掩码的 logits，从而实现数据过滤，提高推荐结果的准确性和有效性。

### 4.5.1 主机侧优化

在 _ h和 _ 参数较大的场景下，候选序列数量的显著增加会导致主机端进行大量的排序和过滤操作。这会导致集束搜索算法的计算瓶颈从AI加速器转移到CPU，最终导致严重的CPU密集型问题。以下概述了xLLM在优化生成式推荐场景中的Beam搜索过程方面所做的努力。

**Beam搜索优化。**在自然语言处理等许多领域，集束搜索优化是一种广泛应用于序列生成任务的高效算法[99]。在处理大量候选序列时，集束搜索的核心操作之一是从众多候选序列中选择 _ h序列。该过程本质上是一个部分排序操作，其独特之处在于它只需要识别前 _ h个元素，而无需对所有候选序列进行完全排序，从而显著降低了计算复杂度。在集束搜索的另一个关键步骤中，从现有的 _ h序列生成 _ h× _ 候选序列，存在一个重要性质：每个序列的 _ 按降序排列。利用此性质，可以采用优化的提前终止机制，当 _ 不满足特定条件时，计算会提前停止，从而进一步提高计算效率。具体操作流程如下：

首先，按顺序访问现有的 _ h序列。对于每个访问的序列，我们创建一个大小为 _ h的小型最小堆，用于动态维护当前选定的局部最优元素集合。接下来，遍历后续序列。在此过程中，如果序列的后续 _ 小于最小堆的顶部元素，则意味着根据当前的过滤条件，该序列的后续元素不能进入当前的最优集合。此时，当前序列的过滤操作可以直接终止，并处理下一个现有序列。相反，如果后续的 _ 大于最小堆的顶元素，则将该元素插入到最小堆中，并根据最小堆的性质调整堆结构以保持其顺序。所有相关序列遍历完成后，依次从最小堆中提取顶元素。这些元素按提取顺序构成按降序排列的 _ h 候选序列。通过这一过程，该方法确保了选择高质量的候选序列，同时显著减少了不必要的计算开销，从而提高了Beam搜索算法在实际应用中的运行效率。

**资源重用和预分配。**在Beam搜索过程中，_ h 是预先确定的，并且在模型的每次前向传播中都是固定的，这意味着每次前向传播所需的计算资源保持相对恒定。然而，过大的 _ h 会导致 CPU 计算资源和内存资源的严重浪费。为了有效地缓解这个问题，xLLM采用了精心设计的资源重用策略。具体来说，在生成新的候选序列时，系统会重用先前由旧序列占用的资源，而无需为每个候选序列分配新的空间。这种方法避免了频繁创建新数据结构所带来的开销。基于最终的搜索结果，系统会在集束搜索算法完成后，用新生成的序列的内容更新旧序列的存储区域。通过这种方式，xLLM 不仅减少了内存占用，还最大限度地减少了 CPU 在资源管理和分配方面的额外开销，从而显著提高了计算资源的使用效率。

### 4.5.2 设备侧优化

**有效商品过滤。**在典型的单阶段生成式推荐框架中，有效商品过滤机制至关重要。例如，OneRec [31] 框架使用三个 token ID 的有序组合来唯一地表示一个有效的商品 ID。然而，由于 token ID 组合的数量巨大，并非所有组合都对应实际有效的商品 [100, 101]。为了确保模型生成的结果都是有效的商品，系统会在模型的前向传播过程中异步生成一个有效掩码。该掩码基于预先构建的有效商品词汇表，然后逐个元素地添加到模型输出的 logits 中。通过这种巧妙的设计，

可以调整与无效 token ID 对应的 logits，从而确保这些无效 token ID 在后续计算中几乎不会被选中。这有效地过滤掉了无效的 token ID，从而保证了最终推荐结果的有效性和准确性。

# 5 Evaluation

在§5.1中，我们针对几个基线推理系统（特别是vLLM-Ascend1(v0.10.rc1)和MindIE2(v2.1.rc1)）在Ascend 910B/910C [18]实例上在一系列场景中评估了xLLM框架的完整实现。此外，我们将xLLM、MindIE和vLLM-Ascend在Ascend 910C上的部署分别表示为xLLM‡、MindIE‡和vLLM-Ascend‡。测试场景分为多个类别，反映了不同的在线服务应用，例如精研AI聊天机器人、客服助理、商家助理、产品理解和营销推荐系统。此外，在§5.2中，我们进行了一项详细的消融研究，以评估每个优化模块的单独贡献。

## 5.1 主实验结果

本节将 xLLM 与主流推理框架进行基准测试，以证明其在各种模型（包括 Qwen2/3 系列 [102, 87] 和 Deepseek [3] 模型）和数据集中的卓越推理效率。§5.1.1 详细介绍了使用 ShareGPT3 数据集进行的公平比较。为了确保在所有 LLM 推理框架之间进行公平比较，**我们实验设置的关键特征是输入和输出序列长度固定，而请求速率动态调整以匹配每个框架的目标 SLO（例如，TPOT）阈值。**我们还在不同的节点配置下评估了此设置，包括单节点和具有 PD 分离的多节点设置。在 §5.1.2 中，我们将在京东（目前已部署 xLLM）的各种实际业务场景中对其进行评估，展示其在实际部署条件下的性能。

### 5.1.1 Benchmarking Performace

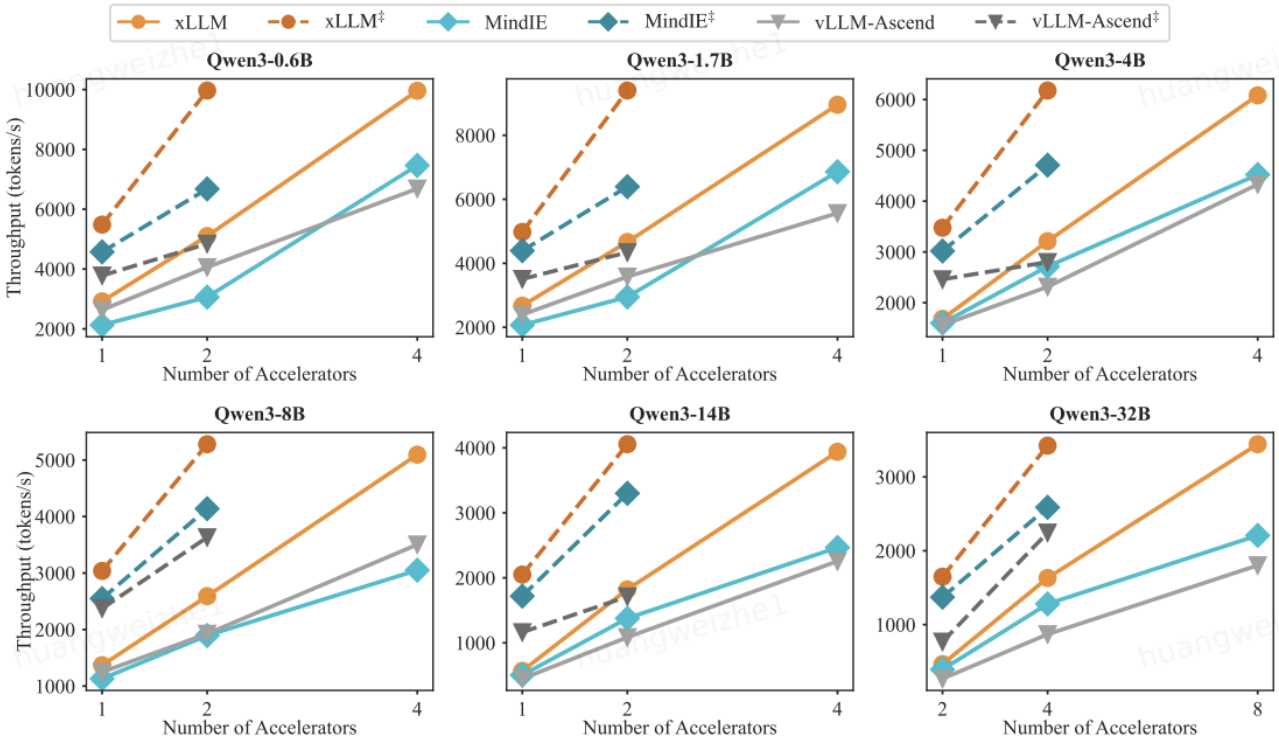


图 14：使用 ShareGPT 数据集在 Qwen3 系列模型上评估的 LLM 推理框架的吞吐量比较，在 TPOT=50ms 和输入/输出长度=2048 的约束下。

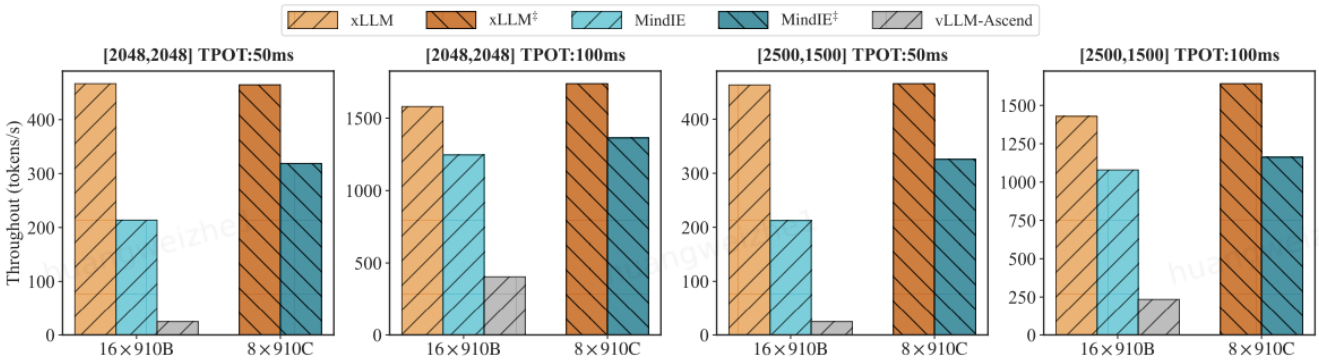


图 15：在给定 TPOT 和输入/输出长度的约束下，使用 DeepSeek-R1 的 LLM 推理框架在 ShareGPT 数据集上的吞吐量比较，其中 [2500, 1500] 表示输入长度为 2500，输出长度为 1500。910C 上的 vLLM-Ascend 被排除在比较之外，因为其在 Deepseek-R1 上的性能未能达到所需的 TPOT 阈值。

| Method | Prompt Length | Output Length | Throughput (tokens/s) | Request Rate (req/s) |
|---|---|---|---|---|
| MindIE | 2048 | 2048 | 8476.44 | 4.14 |
| xLLM | 2048 | 2048 | **11351.58** | **5.54** |

表 3：在 TPOT=100ms 约束下，DeepSeek-R1 与 PD 分离在 ShareGPT 数据集上的比较。

**Qwen3 系列。**如图 14 所示，我们在基准测试中全面比较了 Qwen3 系列（参数数量从 6 亿到 32 亿）中四个推理框架的吞吐量性能。所有测试均在统一配置下进行，输入/输出长度设置为 2048 个 token，TOPT 约束为 50 毫秒。实验结果表明，虽然所有评估框架都通过添加加速器实现了吞吐量提升，但 xLLM 及其 Ascend 910C 实现（xLLM‡）始终保持卓越的性能，证明了其在各种模型规模下都具有近乎线性的强大可扩展性。具体而言，与 vLLM-Ascend 和 MindIE 相比，xLLM 分别实现了高达 1.9 倍和 1.7 倍的吞吐量提升。同样，xLLM‡ 的性能分别比 vLLM-Ascend‡ 和 MindIE‡ 分别高出 2.2 倍和 1.5 倍。此外，xLLM‡在大多数场景下都比基于 Ascend 910B 的 xLLM 表现出稳定的性能提升，验证了软件堆栈对新硬件的有效利用。

**DeepSeek-R1。**图 15 展示了各种推理框架在基准测试中对 DeepSeek-R1 模型的吞吐量性能，其中 Ascend 910B 使用 16 个加速器，Ascend 910C 使用 8 个加速器。结果表明，所提出的 xLLM 框架在 Ascend 910 系列上实现了卓越的吞吐量。定量分析显示，在 Ascend 910B 上运行的 xLLM 的平均吞吐量比 MindIE 提高了约 1.7 倍，比 vLLM-Ascend 提高了 12 倍。此外，xLLM‡ 的平均吞吐量比 MindIE‡ 提高了约 1.4 倍。

**PD分离设置。**表 3 使用 PD 分离架构对 MindIE 和 xLLM 框架在 DeepSeek-R1 模型上的推理性能进行了基准测试。在相同条件下，对于 2048 长度的输入/输出，TPOT 控制在 100ms，xLLM 实现了大约 34% 的吞吐量（11,351.58 vs. 8,476.44 个令牌/秒）和请求率（5.54 vs. 4.14 个请求/秒），标志着效率的显著提升。

### 5.1.2 Business Serving Scenarios

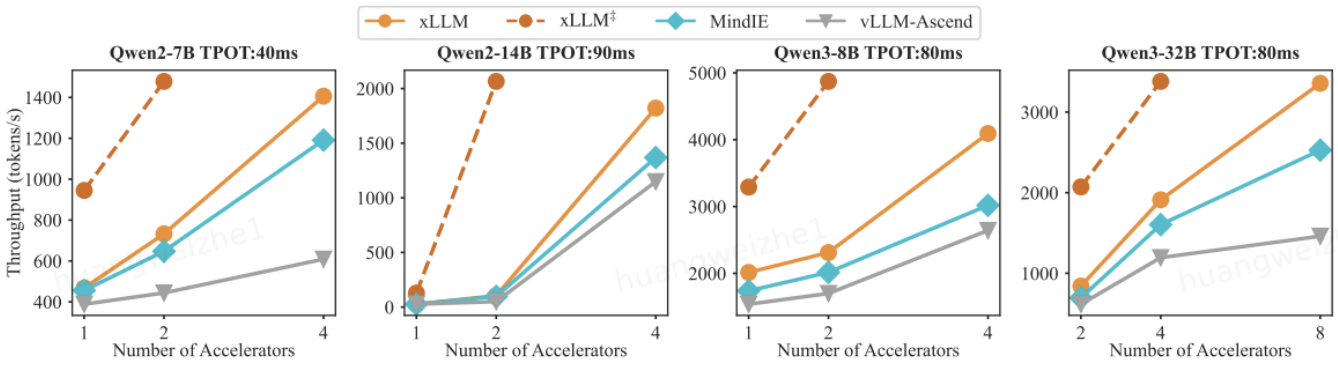


图 16：Qwen2 系列和 Qwen3 系列模型在 JingYan 场景下与各种推理框架的吞吐量对比。

| Method | Prompt Length | Output Length | Throughput (tokens/s) | Request Rate (req/s) |
|---|---|---|---|---|
| vLLM-Ascend | 6800 | 400 | 21.17 | 0.11 |
| MindIE | 6800 | 400 | 144.40 | 0.67 |
| xLLM | 6800 | 400 | **196.45** | **0.89** |

表4：在TPOT=80ms约束的JingYan场景下，Deepseek-V3模型与各框架的对比。

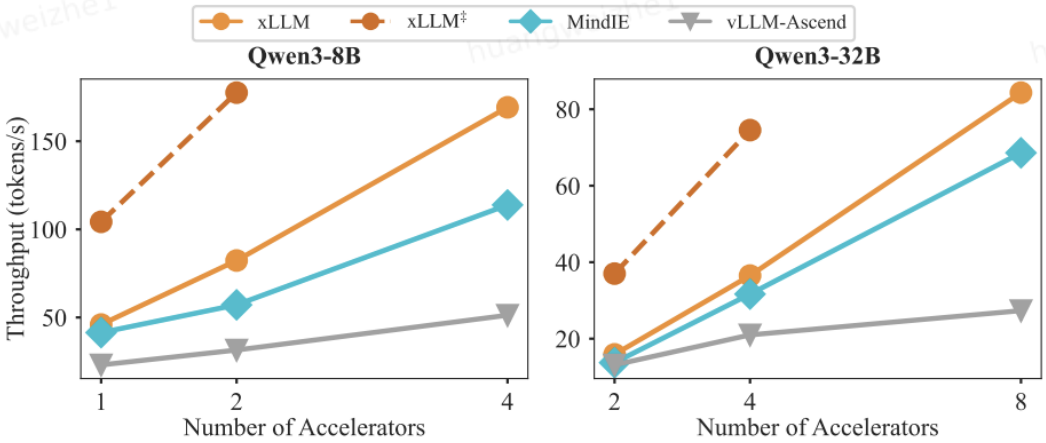


图 17：Qwen3 系列模型与各种推理框架在端到端延迟 (E2E) = 10 秒的客户服务场景下的吞吐量比较。

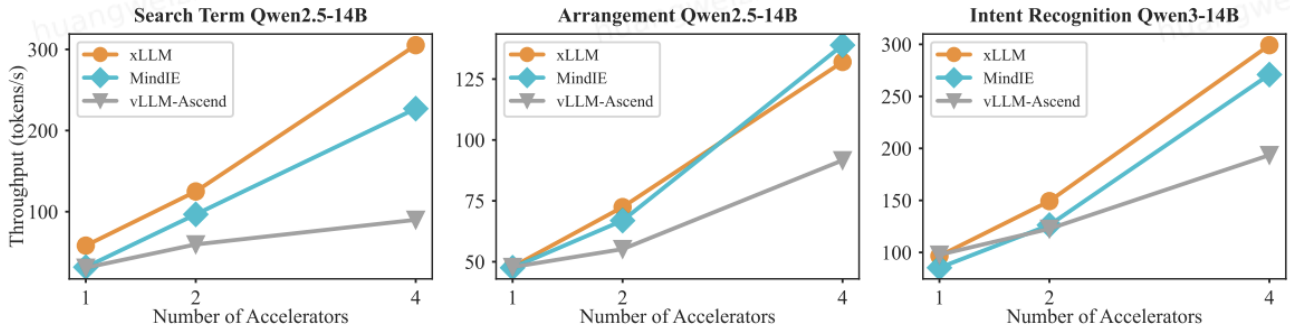


图 18：Qwen 系列模型与各种推理框架在商家助手场景下的吞吐量比较，端到端延迟 (E2E) = 1s。

| Method | Prompt Length | Output Length | Throughput (tokens/s) | | |
|---|---|---|---|---|---|
| | | | #device=1 | #device=2 | #device=4 |
| vLLM-Ascend | 1200 | 40 | 795.77 | 874.97 | 1272.52 |
| MindIE | 1200 | 40 | 944.81 | 1051.44 | 1693.45 |
| xLLM | 1200 | 40 | **1001.91** | **1323.90** | **2425.13** |

表 5：Qwen2-7B 模型与各推理框架在商品理解场景下的吞吐量比较

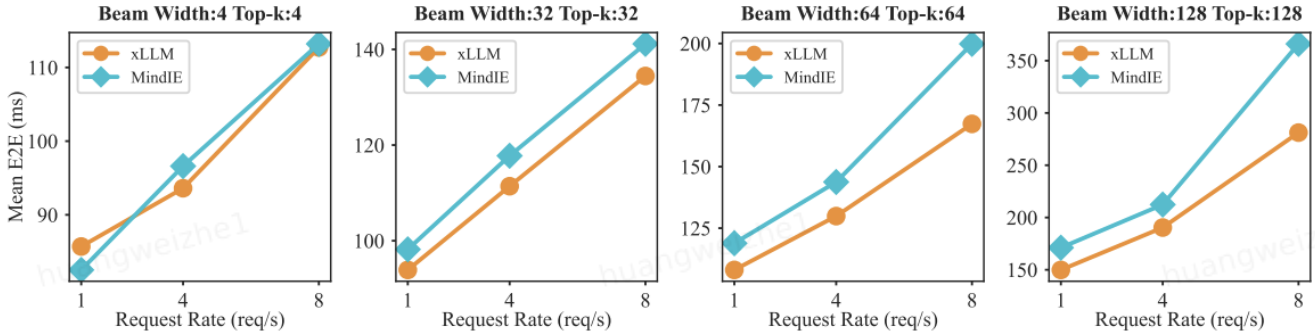


图 19：生成式推荐场景下各推理框架的平均端到端延迟 (E2E) 比较。由于 vLLM-Ascend 不支持beam宽度大于 10，因此图中未绘制相应的结果。即使beam宽度为 4，请求速率为 1，其平均端到端延迟也远远超过了我们的框架。

**JingYan。**JingYan 是一款人工智能购物助手，旨在帮助用户发现新产品、寻找灵感并获得问题的答案。因此，JingYan 的数据集包含模型与用户之间的对话日志，捕捉这些丰富的交互。如图 16 所示，我们系统地评估了四种框架在 JingYan 场景下服务于 Qwen2 系列和 Qwen3 系列模型的推理吞吐量。结果表明，与 MindIE 和 vLLM-Ascend 相比，xLLM 和 xLLM‡ 在所有模型规模下都保持了卓越的吞吐量，并展现出更好的扩展效率。例如，在使用 4 个加速器服务于 Qwen3-8B 模型时，xLLM 的吞吐量约为 vLLM-Ascend 的 1.6 倍，并且显著超越了 MindIE。xLLM 在 910B 上的强劲性能，以及在 910C 硬件上增强的结果，凸显了其对后续硬件版本的有效适配。如表 4 所示，DeepSeek-V3 模型也呈现出类似的趋势，xLLM 的吞吐量是 vLLM-Ascend 的 9 倍以上，比 MindIE 高出 36%。

**客户服务。**客户服务数据集包含客户与客服人员之间的交互式对话。图 17 详细展示了 Qwen-8B 和 Qwen-32B 模型在客户服务场景下不同推理框架的性能差异。结果表明，我们的 xLLM，尤其是在 Ascend 910C 上运行的 xLLM‡，在所有测试配置中都提供了显著更高的吞吐量。例如，在 8 个加速器上使用 Qwen3-32B 时，xLLM 的吞吐量分别是 vLLM-Ascend 和 MindIE 的 3.1 倍和 1.2 倍。值得注意的是，随着加速器数量的增加，vLLM-Ascend 框架出现了明显的扩展瓶颈，而 xLLM 则保持了近乎线性的效率扩展。这验证了 xLLM 框架在管理大规模模型分布式推理方面的高效性和优越性。

**商家助手。**表 18 列出了商家助手场景中三个任务（即搜索词、排列组合和意图识别）对各种推理框架的吞吐量进行的基准测试。提出的 xLLM 框架的性能优于或与 MindIE 相当，并显著领先于 vLLM-Ascend。具体来说，对于使用四张加速卡的搜索词任务，xLLM 的吞吐量比 MindIE 高 34%，大约是 vLLM-Ascend 的 3.4 倍。

**商品理解。**对于商品理解场景，表 5 显示了 Qwen2-7B 模型与多种框架的吞吐量对比。实验结果表明，在不同加速卡数量下，xLLM 的性能分别比 MindIE 和 vLLM-Ascend 平均高出 25% 和 56%。此外，xLLM 的优势随着加速卡数量的增加而增强，这表明它能够有效利用大规模并行计算资源，从而为高性能 LLM 推理提供强大的解决方案。

**生成式推荐。**如图 19 所示，在 Qwen-8B 模型上的评估结果表明，xLLM 在各种请求速率和波束宽度下始终比 MindIE 实现更低的平均端到端延迟，除非是在极低负载条件下。值得注意的是，随着波束宽度（从 4 到 128）和请求速率的增加，xLLM 的性能优势愈发明显。例如，在最

具挑战性的场景下，波束宽度为 128，请求速率为 8，xLLM 相对于 MindIE 降低了约 23% 的延迟。这一显著的提升验证了我们 xLLM 在主机端和设备端的优化有效地缓解了生成推荐任务中的计算瓶颈，显著提升了高负载下的推理效率和可扩展性。

## 5.2 消融实验

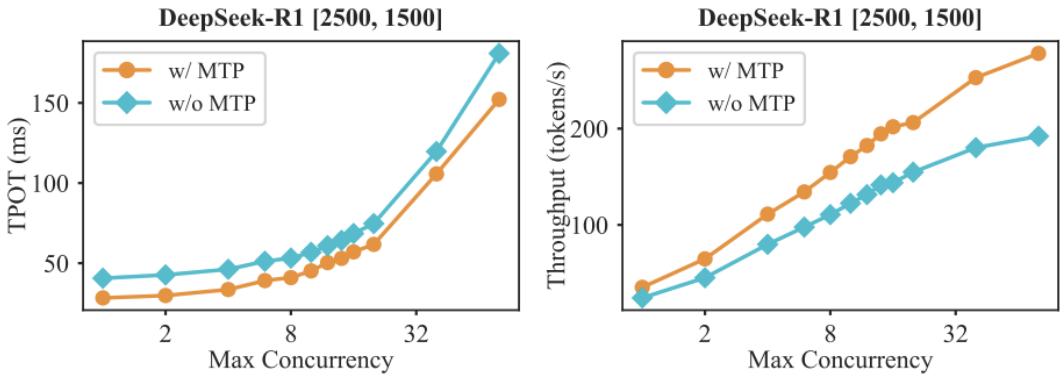


图 20：MTP 对 DeepSeek-R1 模型并发性能的影响。

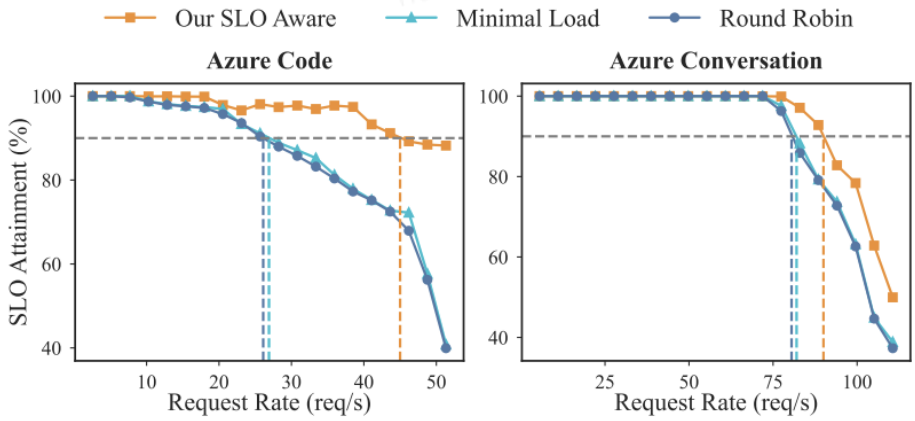


图 21：不同调度策略下的动态 PD 分离策略性能

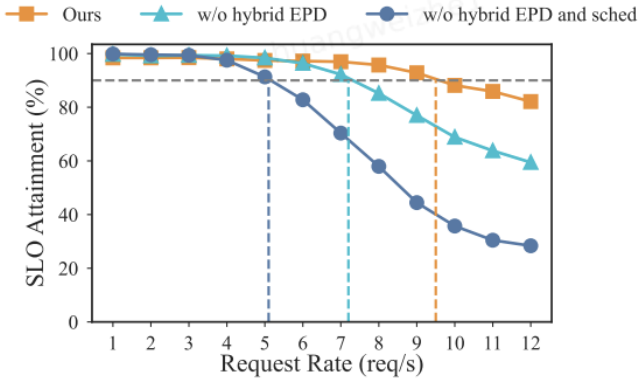


图 22：混合 EPD 分离调度策略的影响

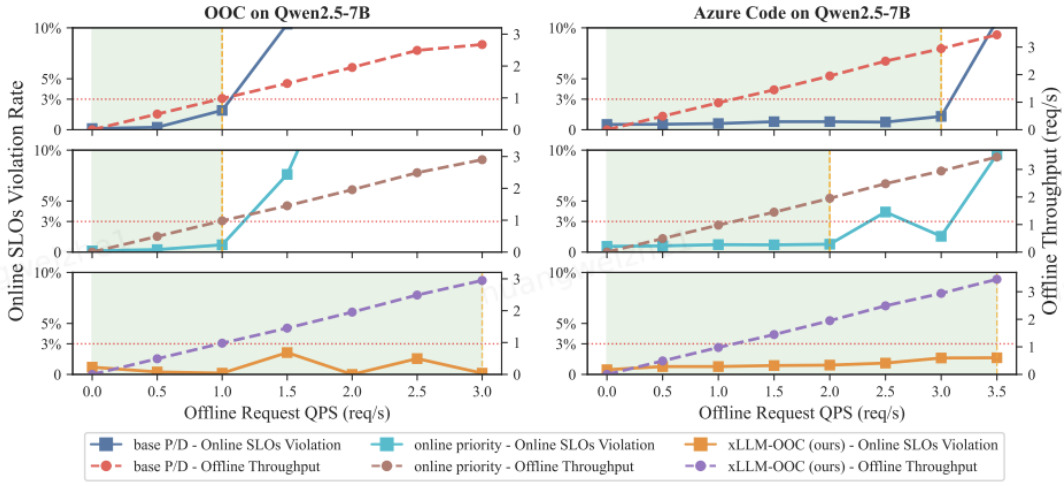


图 23：在线-离线共置调度调度策略的影响

| Model | Prompt Length | Output Length | Async Scheduling | Throughput (tokens/s) |
|---|---|---|---|---|
| DS-Distill-Qwen-1.5B | 1000<br>1000 | 1000<br>1000 | ✗<br>✓ | 8709.90<br>**10223.15** |
| DS-Distill-Qwen-7B | 1000<br>1000 | 1000<br>1000 | ✗<br>✓ | 3183.68<br>**3201.83** |
| DS-Distill-Qwen-14B | 1000<br>1000 | 1000<br>1000 | ✗<br>✓ | 1472.22<br>**1527.23** |
| DS-Distill-Qwen-32B | 1000<br>1000 | 1000<br>1000 | ✗<br>✓ | 1415.20<br>**1508.76** |

表6：多层流水线执行中异步调度机制的影响。

| Metric | Single-Stream (ms) | Dual-Stream (ms) |
|---|---|---|
| Total Communication Time | 9.3 | 12.4 |
| Overlapped Communication Ratio | - | 80% |
| Exposed Communication Time | 9.3 | 2.5 |
| Total Computation Time | 13.0 | 17.0 |
| Reduced Time per Layer | - | 2.8 |
| Total Reduced Time (61 layers) | - | 172.0 |

表 7：使用多层流水线执行的 DeepSeek-R1 单个解码器层的通信和计算开销。

| Model | Prompt Length | Output Length | Adaptive Graph Mode | Throughput (tokens/s) | Mean TPOT (ms) |
|---|---|---|---|---|---|
| Qwen3-1.7B | 2048<br>2048 | 2048<br>2048 | ✗<br>✓ | 2385.58<br>**3038.88** | 39.27<br>**30.63** |
| Qwen3-4B | 2048<br>2048 | 2048<br>2048 | ✗<br>✓ | 1539.93<br>**1671.39** | 55.44<br>**50.58** |

表 8：自适应图模式的影响

**MTP 的影响。**如图 20 所示，在 DeepSeek-R1 模型的输入长度为 1500、输出长度为 2500 的配置下，启用多令牌预测 (MTP) 技术显著优化了推理性能。随着最大并发数的增加，启用 MTP 的版本与基线相比始终表现出更低的 TPOT，表明生成延迟降低。同时，其吞吐量显著高于非 MTP 版本，尤其在超过 32 个并发请求时优势更为显著。这表明 MTP 在高并发条件下有效提升了计算效率和系统吞吐量。

**动态 PD 分离调度策略的影响。**如图 21 所示，我们将我们提出的 SLO 感知型动态 PD 分离策略与最小负载策略和循环调度策略进行了比较。在具有显著突发流量的 Azure Code 数据集上，我

们的 SLO 感知型策略的请求服务率是最小负载策略的 1.67 倍。同时，与循环调度策略相比，最小负载策略可将 SLO 性能提升高达4.3%。对于输入/输出长度变化稳定的 Azure Conversation 数据集，我们的 SLO 感知型策略的请求服务率是最小负载策略的 1.1 倍。虽然最小负载策略的性能与循环调度相似，但它仍可将 SLO 性能提升高达2.4%。这些结果表明，最小负载调度比循环调度方法更接近最优调度策略，而我们的自适应实例调度可提供最佳的整体性能。

**混合 EPD 分离调度策略的影响。**图 22 展示了所提出的混合 EPD 分离策略在 TextCaps 数据集上的有效性，特别是在控制 TPOT 和提高 SLO 达成率方面。在 8 个通用推理实例的配置下，移除混合 EPD 分离方法会导致实际吞吐量从 9.5 个请求/秒下降到 7.2 个请求/秒，而随后移除阶段级调度策略则进一步将实际吞吐量降低到 5.1 个请求/秒。这表明，精心设计的分离策略可以有效地减少阶段间干扰，而我们的阶段级调度策略则有助于更精细地控制每个批次的执行时间。这两个组件的集成显著提升了系统的整体性能和稳定性，尤其是在高并发场景下。

**在线-离线共置调度策略的影响。**如图 23 所示，我们评估了三种调度策略，即基准 P/D、在线优先级以及我们提出的在线-离线共置调度策略（xLLM-OOC），以评估在不违反在线请求的 SLO 的情况下，离线请求可实现的最大吞吐量。图中绿色阴影区域表示离线请求吞吐量的可持续范围，该范围保持在可接受的在线 SLO 违规阈值之内。当离线每秒查询数 (QPS) 超过一定水平时，基准 P/D 和在线优先级策略都会导致在线 SLO 违规率急剧上升，这表明离线工作负载造成了干扰。相比之下，即使离线 QPS 持续上升，xLLM-OOC 也能保持稳定的 SLO 合规性。在我们专有的数据集上，xLLM-OOC 实现了比其他两种方法高三倍的吞吐量。此外，在 Azure Code 数据集上，它分别比在线优先级和基准 P/D 提高了 75% 和 17%。

**多层流水线执行的影响。**表 6 的结果证明，我们提出的异步调度机制在所有评估的模型规模上均实现了一致的吞吐量提升。在 1.5B 参数模型中，相对增益最显著，为 17.4%，这突显了该方法对于调度开销在总计算时间中占比较大的小型架构尤为有效。虽然对于较大的模型，相对增益有所缓和（7B 模型达到 0.6%，14B 模型达到 3.7%，32B 模型达到 6.6%），但所有配置都表现出了统计上显著的绝对增益。这些结果有力地验证了我们的方法能够通过利用占位符标记将 CPU 调度与 NPU 执行解耦和重叠，成功地掩盖调度延迟并消除计算bubble。我们在表 7 中进一步评估了所提出的双流架构对 DeepSeek-R1 模型的有效性。单个解码器层的实验结果表明，双流模式下的总通信时间从单流模式下的 9.3 毫秒增加到 12.4 毫秒。然而，计算-通信重叠机制成功隐藏了 80% 的通信时间，

这使得暴露的通信时间降至仅 2.5 毫秒，每层节省 6.8 毫秒。尽管每层引入了 4 毫秒的计算开销，双流策略在整个 61 层模型中实现了172.0 毫秒的净性能提升，这清楚地证明了我们的调度策略在实际工作负载中的能力。

**自适应图模式的影响。**为了验证自适应图模式优化技术的有效性，我们在 Qwen3-1.7B 和 Qwen3-4B 模型上进行了对照实验。如表 8 所示，当提示和输出长度均设置为 2048 个 token 时，启用自适应图模式后，两个模型的性能均显著提升。Qwen3-1.7B 的吞吐量从 2385.58 提升至 3038.88 个 token/s（提升 27.4%），而其平均 TPOT 降低了22.0%。相应地，Qwen3-4B 的吞吐量提升了 8.5%，延迟降低了 8.8%。这些结果有力地证明了自适应图模式作为一种通用推理加速技术的有效性，并且在参数规模较小的模型上观察到更显著的优化效果。

**分层 DP 负载均衡的影响。**我们提出的分层DP负载均衡方案可将总吞吐量提高5%。内核级优化可显著降低延迟；

例如，对于一个超长的32k令牌请求，重新排序和拆分可以将单个核心的计算负载从32k令牌减少到1.3k令牌，节省约800微秒。相比之下，DP组间迁移带来的延迟节省则较为有限。即使在平衡

了20k令牌的差异后，61个Transformer层的总时间节省也仅为约600微秒。这表明最大的性能瓶颈和优化潜力在于计算单元本身。

# 6 未来工作

尽管本文提出的 xLLM 大模型推理引擎已在提升大模型推理效率，降低推理成本方面展现出潜力，但若要构建真正高效、普惠的智算基础设施，仍需在硬件、模型与应用三个层面进行更深层次的协同创新。未来的研究工作将致力于推动 xLLM 从一个高效的推理引擎，演进为一个能够支撑下一代智能应用的 AI 操作系统。具体而言，我们将聚焦于以下三个方向。

## 6.1 引导开放与多元的硬件生态

当前人工智能基础设施普遍面临与特定硬件架构深度绑定的问题，这不仅导致了算力稳定性的风险，也限制了算力在异构环境（如边缘计算）中的高效部署。为突破这一瓶颈，未来工作的核心之一是**引导并培育一个开放、多元的硬件生态系统**。

**构建统一的硬件抽象层**。我们将构建一个高性能、硬件无关的运行时抽象层。该层通过对底层各类计算（从云端国产芯片到端侧加速器）的指令集与内存架构进行封装，向上提供统一的算子接口与内存管理API。此举旨在实现无需修改代码即可在异构硬件间无缝迁移与高效执行，从根本上降低新型硬件的接入门槛。

**推动软硬件协同设计与生态闭环**。我们将积极与硬件伙伴开展协同设计，共同定义更高效、开放的软硬件接口标准。通过这种方式，不仅能从软件层面反向激励硬件设计的创新与优化，还能逐步构建起一个自主可控、良性竞争的算力供给体系，最终为用户在成本、性能与安全性上提供多元化的最优选择。

## 6.2 培育繁荣与敏捷的模型生态

系统的核心价值在于其所能支撑的模型广度与集成速度。未来的工作将围绕**构建一个既包容多样又响应迅速的模型生态系统**展开。

**多元化的场景支持能力**。我们将突破当前以大语言模型为主的支持范围，将平台的优化与部署能力系统性地扩展至**生成式推荐、文生图、文生视频**等更广泛的生成式AI场景。我们需要为不同场景深度优化执行引擎、请求调度、显存/存储优化等，以确保多样化工作负载均能获得最优性能。

**模型的“天级零”集成能力**。为应对模型架构的快速迭代，我们计划构建基于**模型图化框架+统一算子库+编译优化**的架构，以实现对新发布模型的“天级零”自动集成。该能力将通过实现模型图表达、建设可扩展的统一算子库以及实现自动化模型切分与编译策略来达成，从而将前沿模型的部署周期从数周缩短至数小时，提升系统的技术竞争力。

## 6.3 演进为AI原生的应用框架

为实现人工智能技术的普惠化并加速其创造业务价值，本系统需从模型推理引擎，演进为一个**AI原生的全栈应用框架**。

**提供框架原生的AI中间件能力**。我们计划设计一系列高层次的、框架原生的API与抽象，将复杂的分布式推理、多模型编排、有状态会话管理等能力封装为开箱即用的中间件服务。这使得应用

开发者能够轻松组合出复杂的AI应用（如多模态AI智能体），而无需深入理解底层基础设施的复杂性。

**实现应用的极速集成与上线。**在此AI原生框架之上，我们将打造一套完整的工具链（包括高度封装的SDK、应用模板和集成化的CI/CD流水线），以极致简化从开发到部署的全流程。我们的目标是使应用团队能够在数小时而非数周内，完成AI服务的集成与上线，从而显著提升业务创新的敏捷度，彻底打通从模型研发到价值创造的“最后一公里”。

# 7 总结

我们推出了 **xLLM**，一个智能高效的 LLM 推理框架，其特点是服务与引擎解耦的架构。该框架由两个主要组件组成：(1) **xLLM-Service**，一个多功能服务层，旨在实现高效的实例和集群管理以及请求调度。它融合了统一的弹性调度机制，用于共置在线和离线请求，从而最大限度地提高集群利用率；一个用于 SLO 感知动态实例调度的工作负载自适应 PD 分离架构；一种用于多模态请求的新型编码-预填充-解码 (EPD) 分离机制；一个用于高效内存管理（包括 KV 缓存上传和卸载）的全局 KV 缓存管理器；以及一个分布式容错设计，以确保高可用性。(2) **xLLM-Engine**，一个高性能推理引擎，经过优化，可在各种 AI 加速器上加速 LLM 推理。 xLLM-Engine 采用全栈多层执行流水线优化，通过异步 CPU 端调度和 AI 加速器端模型转发来最大限度地减少计算泡沫，利用微批次的双流并行通过全对全通信实现计算重叠，并进一步在操作符级别重叠各种 AI 计算单元。xLLM-Engine 还实现了一种自适应图模式，将内核序列、内存操作和同步预编译成单个计算图，从而大幅减少启动开销和加速器空闲时间。此外，它通过提出的 xTensor 内存管理采用“逻辑连续、物理离散”的 KV 缓存存储策略，解决了内存连续性和动态分配之间的矛盾。为了进一步提升硬件利用率，xLLM-Engine 集成了多项算法增强功能，例如异步流水线自适应推测解码、键值缓存感知调度、用于动态负载均衡的响应式数据处理单元 (DP) 组间工作负载迁移，以及基于实时专家工作负载统计和透明权重更新的动态 MoE 负载均衡。我们进一步将 xLLM 扩展到新兴的生成式推荐场景，提升了推荐的准确性和效率。

大量实验表明，xLLM 与 MindIE 和 vLLM-Ascend 等领先的推理系统相比，在主流 Qwen 系列和 Deepseek 系列模型以及公开和实际工业数据集上均取得了持续的提升。尤其值得一提的是，在吞吐量方面，xLLM 的性能比 MindIE 和 vLLM-Ascend 分别高出 1.7 倍和 2.2 倍。全面的消融研究进一步验证了关键组件的有效性，包括所提出的调度模块、多层执行流水线、优化的模型张量并行性、自适应图模式等。

通过将 xLLM 框架作为开源项目发布，我们旨在激发进一步的创新，以开发强大的企业级推理解决方案，优化各种 AI 加速器的性能，并为下一代 AI 应用创建紧密集成的服务和引擎架构。