ModelLess：面向共享 GPU 的多层级大语言模型推理无服务器系统

Ding Shi Wei Wu Jiayuan Chen Chenghu Yang 蚂蚁集团

arXiv:2611.32634v1 [cs.DC] 24 Nov 2026

摘要

大语言模型推理平台在按产品配置的 GPU 上提供在线、异步、批处理和 rollout 产品，因此服务的 GPU 卡在突发请求之间处于空闲状态，而其他工作却在等待。我们提出 ModelLess，一种用于大语言模型推理的无服务器系统。用户声明模型服务，平台负责管理实例。空闲服务可缩容至零：其实例连同权重一并休眠在主机内存中（对于 TP8 的 GLM-5.2，释放约需 1 秒，唤醒约需 4 秒，而冷启动则超过 4 分钟）。计费依据服务持有 GPU 的时间。休眠实例释放的 GPU 会被借给低层级使用。集群管理器会抢占低优先级任务，并将驱逐视为全有或全无；但大语言模型实例的驱逐成本取决于其权重、执行状态、键值缓存和进行中的请求中哪些内容得以保留。因此，ModelLess 为每个层级（在线、作业、弹性）定义驱逐语义，说明其被抢占时可能丢失的内容，并运行一套抢占协议：在释放内存前关闭准入，在中止请求前先将其持久化地重新加入队列，并在出现故障时默认拒绝。四个组件（入口组件、调度器、节点代理、SGLang 插件）通过休眠/唤醒、对弹性层级进行中的请求的暂停/恢复，以及用于强化学习 rollout 的权重复制路径（跟随/封存）来执行该协议。

重放来自为长尾在线模型提供服务的 1,100 张 H20 卡的一周跟踪数据表明，在采用粗粒度延迟目标和一个固定空闲阈值的情况下，约 38% 的 GPU 时间（417 个等效 GPU）可以被借出。这是一个理想化的上限，且只有在恢复耗时数秒时才能达到：当冷启动需要 240 秒时，即使采用按服务阈值，可借出的比例也会低于 8%。在单个节点上，针对每种策略各运行一次四小时实验，结果表明，在在线延迟中位数相同的情况下，与中止并重新排队相比，暂停每个保持的 GPU 样本可产出多 22% 的借用方令牌。作业层和跟随/封存尚未进行端到端评估。

1 引言

大语言模型推理平台会同时提供多种产品：具有延迟目标的在线端点、异步请求、批处理作业和强化学习 rollout。这些产品允许请求等待的时间相差几个数量级，但 GPU 却是按产品分别配置的。每个在线服务都会为峰值需求保持 GPU 卡，而 rollout 和批处理工作负载则使用独立容量。空闲在线服务保持的 GPU 卡无法为等待中的批处理作业提供服务，而批处理资源池必须按自身峰值进行规模配置。

ModelLess 采用无服务器视角。用户声明的是模型服务，而非实例；实例、部署位置和扩缩容则由平台负责。没有流量的服务会缩容至零：其实例进入休眠状态，不占用 GPU。计费依据服务持有 GPU 的时间，因此休眠实例不会产生 GPU 费用。对于大语言模型而言，缩容至零存在一个问题：冷启动需要数分钟。ModelLess 将权重保留在主机内存中，使实例能够在数秒内唤醒，并将休眠实例释放的 GPU 提供给低层级。

集群管理器会抢占低优先级工作以释放容量，Borg 就会这样处理其非生产任务 [64], , 但它们将任务视为不透明的整体：任务要么正在运行，要么被驱逐。模型实例并非如此。其昂贵状态（权重和编译后的执行状态）可以停放在主机内存中，并在数秒内恢复；其廉价状态（键值缓存和进行中的请求）则可以保存检查点。因此，驱逐成本是一种设计选择。集群管理器会终止任务。引擎级工作（睡眠模式、InferCept、ConServe）决定单个引擎保留哪些内容；这些方案都不会把释放出的 GPU 交给另一个模型。ModelLess 将保留策略变为集群级的属性，并针对每个层级分别管理，同时将 GPU 连同权重一起交给另一个模型，而不丢失正在处理的请求。

我们首先利用一周的跟踪数据，考察空闲容量是否真实存在。这些数据来自为长尾在线模型提供服务的 1,100 张 H20 卡（第2节）。服务大部分时间处于空闲状态，这一点也得到了其他关于长尾服务和通用服务集群的研究印证 [16, 72, 81],；大语言模型请求流也呈现突发性 [68, 73]。大多数空闲时间窗口都很短，但大部分空闲 GPU 时间集中在一小时或更长的时间窗口内。在较宽松的延迟目标下，采用单一固定空闲阈值可借出 38% 的 GPU 时间，采用按服务阈值则可借出 46%（第7.1节）。这一比例取决于恢复延迟。冷启动需要几十秒到几分钟 [14, 24, 82]，此时可回收的 GPU 时间所占比例很小；而恢复只需几秒时，大部分 GPU 时间都能保留下来。延迟承诺越严格，恢复速度就越重要。

要将这一容量转化为供较低层级使用的工作，需要满足四点。第一，交接必须在数秒内完成，但重建一个模型实例需要数分钟（§2.5）。第二，借用方

经常会被抢占，而且其时间窗口比作业更短：一个批处理请求可能运行数分钟，而所有者会在一两分钟内返回。通过中止并重新排队进行抢占会丢弃解码进度，并重新运行请求；通过重计算续接进行抢占则会保留已生成的词元，但需要再次填充整个上下文。第三，交接必须保留权重和执行状态，因为重建它们需要数分钟；而键值缓存和进行中的请求只需在某个层级需要时保留。第四，借用方的权重会在任务运行期间发生变化，因为推演层提供的是正在训练的策略[13, 56, 67]。

我们在 ModelLess 中解决了这些问题。ModelLess 是一个无服务器系统，其中模型实例是调度和抢占的单位。它建立在四项设计原则之上。

（1）调度实例，而不是容器。实例承载执行状态、权重、键值缓存和请求状态，每次交接都要决定保留其中哪些内容（§4）。对象链将声明内容与运行内容分离，使一个基础运行时能够承载不同形态的工作负载而无需重启，并且层级不与引擎耦合（§3.1）。（2）为每个层级采用各自的驱逐语义，并与其客户端能够容忍的情况相匹配。执行中的在线请求无法容忍停顿，因此较低层级永远不会抢占它。空闲 θ 秒的在线服务可能会被停放，其第一个请求随后需要承担恢复延迟。作业会分配给等待该作业的训练器，因此训练器在最低层级之上的引擎上发出的请求会被迁移并以较低速度运行，但不会被暂停。弹性客户端向队列提交请求并轮询，因此弹性请求可能被隔离并重新入队，或者在引擎支持的情况下被暂停（§4.2）。（3）通过独占式移交保护所有者。同一时间，一个卡组上只有一个引擎处于活跃状态，交接会将整个卡组交还（§6.2）。GPU 无法像 CPU 那样进行细粒度分区，而大型模型会占满一个卡组。（4）通过排序并在故障时安全关闭来确保安全性。被抢占引擎的准入门控会在其内存被释放之前关闭。尝试会在被中止之前重新入队。暂停失败会使引擎处于需要恢复状态，并保持准入关闭（§5.2、§6.2）。

四个组件实现了这些原则（第3.3节）。*mod-elentry*维持客户端连接，并作为持久队列的前端。*modeldispatcher* 负责做出每一个放置决策。*modellet* 管理节点上的卡和引擎生命周期。SGLang 内部的 *engineplugin* 负责执行休眠、唤醒和暂停。

这些组件通过三种操作和两条请求级抢占路径作用于实例：重新入队和重计算续接。休眠/唤醒会释放 GPU 内存，同时将权重保留在主机内存中 [62, 65], ，因此恢复只需几秒。暂停/恢复会将进行中的请求及其 KV页和采样计数器检查点保存到主机内存，并在下一次唤醒后恢复它们，使被抢占的借用方能够从停止处继续。对于弹性层，这是另一种选择，而不是重新入队。作业层使用重计算续接，即重新提交提示以及目前为止生成的令牌。对于第四项要求，该设计提供了*Follow* 和*Seal*。新引擎会从使用相同权重版本的运行中引擎复制其权重。只有在使用这些权重成功完成一次生成后，引擎才会成为这样的源（§5.3）。

平台的商业产品对应三个层级：在线产品属于在线层，展开属于作业层，异步和批处理产品属于弹性层（第4.2节）。塑造这一设计的两种工作负载是长尾在线服务（我们测量其空闲容量）以及强化学习 rollout（其引擎服务于每个训练步骤都会变化的权重）。

在一个八 GPU 节点上，通过四小时比较两种仅在一个开关上有所不同的抢占策略，测量暂停相较于中止并重新排队的价值（第7.3节）。使用相同的二进制文件、在线轨迹和借用方工作负载时，暂停使借用方每个保持占用的 GPU 样本获得的令牌增加了 22%（若排除重新排队运行中由引擎故障造成的空闲时间；不排除时则增加 25%）。在线延迟中位数没有变化（176 毫秒对 170 毫秒），两次运行的尾延迟都由交接决定（第7.3节、第8节）。

贡献。1. 一种无服务器资源模型：用户声明模型服务，平台负责调度实例，每个层级则说明实例被抢占时可以丢弃四层实例状态中的哪些层（第3节、第4节）。2. 在 SGLang 上设计并实现 ModelLess：包含四个组件（modelentry、modeldis-patcher、modellet 和一个引擎插件）、能够保留开销较大的状态层的交接操作（休眠/唤醒、失败时关闭的暂停/恢复，以及跟随/封存），以及具有安全不变量和失败时关闭的引擎状态机的抢占协议（第3节至第6节）。3. 以一秒为分辨率测量生产推理集群中的空闲容量，并根据恢复延迟和 SLO 严格度界定可借出的容量，从而明确设计要求（第2节、第2.5节、第7.1节）。4. 测得交接成本，并在单个节点上对暂停与中止后重新入队进行受控的四小时比较，逐秒说明池时间的去向，同时阐述证据的局限性（第7节、第8节）。

§2介绍各个层级、空闲容量以及它们带来的要求。§3介绍资源模型和架构，§4介绍实例状态和层级语义，§5介绍交接操作，§6介绍控制平面及其安全不变量。§7是评估，§8讨论局限性和未来工作。§9将 ModelLess 与大语言模型服务、无服务器保活和强化学习 rollout 方面的先前工作联系起来。

2 动机：层级与空闲容量

2.1 分级产品与静态分区

生产推理平台提供不止一种服务。客户可以在延迟目标约束下调用在线端点、提交异步请求，或将批处理作业加入队列。这些产品的服务级别目标（SLO）在时间尺度和可容忍的中断程度上各不相同。我们依据两种在线 SLO 进行衡量，以展示其差异：更严格的 SLO 要求输入少于 4K 个令牌时，首令牌时间（TTFT）中位数低于 2 秒，可用性达到 99.5%；标准 SLO 则要求 TTFT 中位数低于 4 秒，可用性约为 95%。异步和批处理产品则允许等待数分钟至数小时。

然而，资源是按产品配置的。每项在线服务都会为应对峰值而保留自己的 GPU，而异步和批处理产品则运行在独立的容量上。结果是，这样的资源池既难以确定合适的规模，也难以重新平衡，生产环境中的模型服务集群也报告了这一问题 [23, 72]。空闲在线服务所保持的 GPU，无法为在其他地方等待的批处理作业提供服务。本节衡量有多少容量处于空闲状态；§2.5 说明交接必须满足哪些条件。

2.2 GPU 集群与观测

我们研究 H20 GPU 上在线服务的长尾。如果某项服务在某个参考日的请求速率不超过 0.05 每秒查询数、实例数和卡数均不超过 10、已运行超过一天，且运行在 H20 级别的卡上，就将其纳入总体。随后，我们持续追踪这些入选服务。部分服务在追踪期间会扩容；在下文采用的常规周中，约半数服务的平均请求速率超过 0.05 每秒查询数，而最繁忙的服务约达到 100 每秒查询数。我们仍将它们纳入总体，因为我们关心的是被追踪的 GPU 集群会发生什么，而不是一个冻结样本的情况。

总体包含约 1,000 个服务，使用约 4,000 张卡（图 1）。所有推理流量都通过单个网关进入，因此我们只能观测到接收网关流量的服务的到达事件。总体分为两组。有流量的服务：243 个服务（1,182 张卡）在两个观测周中的至少一个周内具有有效的推理流量。无流量的服务：其余服务（约 3,000 张卡）没有有效的推理流量；其请求速率记录为零，其 GPU 利用率来自健康检查，而非推理。我们在全文中分别处理这两组，因为它们引发的问题不同。

对于有流量的服务，我们必须考虑能否借出空闲时间而不违反服务等级目标。对于没有流量的服务，无需考虑服务等级目标，但如果健康检查会唤醒引擎，仍会妨碍回收。

所有重放结果均基于两周内有流量的服务。常规周涵盖其中 223 项服务，使用 1,100 张卡；这是论文中所有 GPU 集群级数据的分析总体

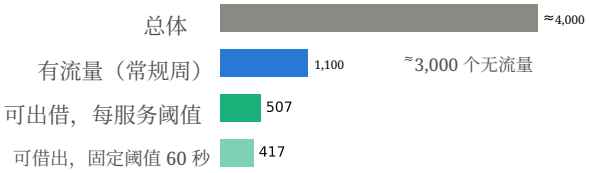

图 1. 总体约有 4,000 张卡，其中可通过休眠与唤醒机制借出的卡：常规周内有流量的卡为 1,100 张，任意时刻平均有 507 张（按服务阈值）或 417 张（固定阈值）可借出。约有 3,000 张卡没有有效的推理流量，另有少数卡仅在节假日周有流量。

（包括摘要中的 1,100 张卡），除非图表标注为国庆节假期周。国庆节假期周涵盖全部 243 项服务，使用 1,182 张卡。计数有所不同，是因为服务会陆续上线和下线，而且有些服务并非每周都能检索到；每项统计均使用对应周的分母。没有流量的服务是最典型的缩容至零场景。平台会保留这些服务的部署，以便回滚和后续测试；它们的健康检查会唤醒引擎，因此约 3,000 张卡不计入这两个分母（§2.4）。请求到达的时间分辨率为秒级。

2.3 空闲结构

空闲现象很常见，但并不平稳，这与其他突发性大语言模型工作负载一样 [68, 73]。在请求之间，大多数长尾服务会让 GPU 长时间闲置，期间穿插着流量突发。我们的重放（第 7.1节）会在 θ 秒内没有收到请求时停放服务；这是一个保持活动式空闲阈值 [54], ，并会释放 GPU，直到收到下一个请求。若设置为 $\theta = 60$ s，恢复时间为 5 s，且不受服务等级目标约束，则常规周内有 47.8% 的 GPU 时间处于停放状态；若设置为 $\theta = 30$ s，该比例为 52.6%。

表 1 和图 2 表明，停驻时段按数量计算较短，但按权重计算较长。在 $\theta = 60$ 秒时，时段中位数仅持续 62 秒，但停放GPU秒数中有 59% 落在持续时间超过一小时的时段中，而停驻卡时段中位数所处的时段长度为 2.1 小时。这种结构的代价是抖动：在 $\theta = 60$ 秒时，一张卡每天约被其所有者唤醒 99 次。因此，借用方获得的大部分 GPU 时间来自较长的时间窗口，但其大多数时间窗口都很短（中位数为 62 秒）；时长不足十分钟的时间窗口承载了 21% 的 GPU 时间。

2.4 可回收容量

并非所有停放时间都可以借出。停放的服务在下一个请求到来时，仍必须满足自身的服务等级目标，而借用方则需要时间加载模型，并需要时间让出。因此，我们将可回收容量 定义为以下 GPU 时间：

表 1。 按空闲阈值划分的停驻时段 θ（常规周，5 秒恢复；毛值，不设 SLO 门控）。“所有者唤醒次数”统计结束停驻时段的到达事件。

θ（秒）	停放份额 ↑	所有者唤醒次数 / 卡片 / 天	时段中位数（按计数）	时段中位数（按卡秒）	时段中的 GPU 秒数 < 10 分钟	GPU 秒数（在时段 ≥ 1 小时内）
30	52.6%	194	29 s	1.4 小时	27%	54%
60	47.8%	99	62 s	2.1 小时	21%	59%
300	37.1%	16	287 秒	4.6 小时	6%	74%
1800	26.7%	2.8	1,739 秒	10 h	1%	90%

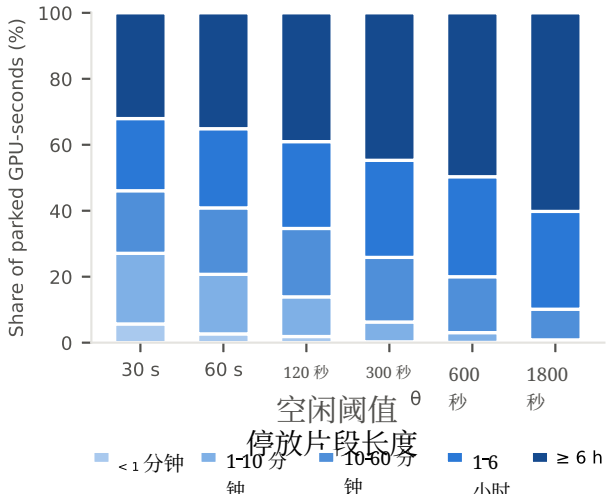

图 2。 每个空闲阈值下按片段长度划分的停放GPU秒数 θ（常规周，C = 5 秒）。尽管大多数片段都很短，但大部分 GPU 时间都集中在超过一小时的片段中。

在每项服务的服务等级目标门控机制均有效的情况下可停放的时间；我们将*净可回收容量* 定义为扣除借用方的加载时间和让出时间，并剔除过短而无法利用的时间段后剩余的容量。

第7.1节定义了门控机制。采用中位数与超时门控（TTFT 中位数以及不超过 0.5% 的 90 秒超时率）、5 秒恢复时间和每个服务的最佳阈值时，常规周的毛可回收份额为 47.7%。扣除 5 秒的借用方负载、3 秒的产出时间和 10 秒的最小有效片段后，剩余 46.1%，相当于 1,100 张卡中的 507 个 GPU 的等效稳定容量。单一固定的 $\theta = 60$ 秒方案不依赖逐服务调优，可提供 417 个 GPU（38%）。在节假日周，采用相同流程可得到 54.5%，即 644 个 GPU。开销影响很小：在产出时间固定为 3 秒的情况下，将借用方负载从 2 秒提高到 30 秒，仅会使该数值从 512 个 GPU 降至 474 个 GPU，因为开销移除的片段都很短。真正会降低容量的是服务等级目标门控机制和恢复延迟，第7.1节对此进行了量化。

这些数据是上界。它们将一项服务视为一个可出借单元，并假设其 Pod 同时处于空闲状态；忽略节点、张量并行和碎片化约束；假设借用方有无限需求；并在事后选择 θ。若将健康检查与引擎隔离，卡数最多还能增加约 3,000 张；我们没有对此进行测量。

2.5 交接要求

一次交接涉及一个所有者（持有一组卡的在线服务），以及一个借用方（所有者空闲时使用这组卡的低层级工作负载）。出借会在所有者连续 θ 秒未收到请求后开始，这段时间称为空闲阈值：所有者的实例会被停放（进入休眠），借用方的实例则会唤醒。当所有者收到下一个请求时，便开始抢占借用方：借用方必须让出卡，所有者的实例则必须恢复。三个时长描述了这一过程。借用方的让出时间是腾出卡所需的时间。所有者的唤醒时间是恢复其实例所需的时间。借用方的加载时间是其在一个时间窗口内开始工作的准备时间。我们用 C 表示恢复延迟，即触发抢占的所有者请求额外等待的时间。C 是所有者实例的唤醒时间，借用方的让出时间会在此基础上增加。长度为 L 的时间窗口内，借用方最多只能获得 L 减去其加载时间和让出时间后的有效工作时间。C =5 s 表示权重位于主机内存时的休眠/唤醒；C = 240 s 表示需要从磁盘读取权重的恢复过程，与冷启动相当。轨迹研究只将唤醒时间计入 C，并将让出时间计入借用方，因此其所有者侧门控不包含让出时间。C = 5 s 低估了尾延迟：第7.3节的端到端运行测得，第99百分位在线首令牌时间为 10–11 s（第7.3.2节）。根据上述测量结果，可以得出四项要求。

恢复必须在数秒内完成，具体需要多少秒取决于服务等级目标。 从磁盘重启（C = 240s）会破坏大多数有流量服务的服务等级目标。其他服务也报告称，冷启动需要几十秒到数分钟 [14, 24, 82]。在θ 从 30 s 到 300 s 的任一时刻，这些服务中有 69–85% 超过 0.5% 的超时阈值。在中位数与超时门控下，此后只有 8% 的 GPU 时间可重新申领（表 8，第7.1节）。假定从预加载于主机内存的权重重启（C = 60 s），在中位数与超时门控下可保留 42.5%，而 C = 5 s 为 47.7%。在平均 TTFT 门限下，前者可保留 18.1%，后者为 46.5%；在尾延迟门槛下，前者约损失四分之一。对于承诺尾延迟的服务，恢复预算为数秒；对于只有中位数和超时要求的服务，恢复预算最多约为一分钟。借用方的让出时间会增加所有者的等待时间，因此也必须很短；在第7.3节的四小时比较中，该时间为数秒。

借用方总体上获得的时间窗口较长，但经常被抢占。在 $\theta = 60$ 这些时间窗口中，59%的停放GPU秒数位于

持续时间超过一小时的时段中。然而，时段中位数的持续时间约为一分钟，所有者每天唤醒一张卡约99次。借用方的大部分GPU时间来自持续数小时的时间窗口，因此数秒的模型加载时间可以忽略不计：当加载时间从2秒增加到30秒时，净可回收容量下降不到10%（§2.4）。每次抢占给借用方带来的成本，决定了借用时间中有多少能够转化为工作。若请求从头重新运行，可能无法在一个时间窗口内完成；重新计算的任务所需时间与其上下文成正比；而保留的任务每次抢占只需付出几秒的代价。具体选择取决于借用方的客户端能够容忍什么，因此不同层级采用不同的驱逐语义（§4.2）。

保留重建成本高昂的部分。 权重和执行环境需要数分钟才能重建，因此交接会保留它们：休眠会将权重保留在主机内存中，并且每个运行时在提供服务前只需准备一次。GPU 内存释放时，KV 缓存和进行中的请求虽然保留成本低，但会丢失。交接仅在工作负载需要的地方保留它们，而 §4 将这种需求转化为分层驱逐语义。

权重会在任务期间发生变化。 强化学习 rollout 并不使用固定权重。每个训练步骤都会产生一个新的权重版本，所有 rollout 引擎都必须在生成之前提供该版本，而且引擎会在一个步骤期间被添加和移除。每次都从存储中加载各个版本，会使大型模型的数十 GB 数据进入每个训练步骤的关键路径，这是强化学习系统中的已知成本 [13, 56, 67]。训练器会将权重推送到各个引擎，因此副本只存在于引擎中。于是，新引擎会跟随一个已经持有该步骤权重的引擎，并从其位于主机内存中的只读快照复制权重（§5.3）。

表 2 总结了各项要求及其对应的解决方案。

2.6 现有机制

每种现有机制都会丢失实例状态的四个层级中的至少一个（第4.1节）。集群管理器优先级和抢占通过设置优雅终止的宽限期来驱逐 Pod。进程会终止，因此执行层和权重层都会丢失，重启需要数分钟；对于我们最大的模型，重启约需 250 秒（第7.2节）。宽限期只能排空请求。只要运行中的 Pod 进程仍在运行，就不能将其 GPU 分配给另一个 Pod；而从外部控制休眠的控制器，也无法获得第6.2节排序规则所需的每个请求的身份信息。普通的缩容至零（Knative、KEDA）会删除 Pod，因此下一个请求必须经历完整的冷启动；表 8 显示，这超出了大多数服务的服务等级目标。ModelLess 保留缩容至零机制，并通过休眠与唤醒降低其成本。更快的冷启动（ServerlessLLM、BlitzScale）缩小了差距：例如，ServerlessLLM 会从 DRAM 或 SSD 层级加载权重。无论重启需要多久，执行状态、键值缓存和请求层都会丢失。表 8 显示

在每种门控机制下，重启约一分钟能保留哪些内容。仅引擎休眠 [62, 65] 会保留权重，但会丢弃前缀缓存，也没有说明如何处理飞行中的请求、层级，或交接必须遵循的顺序。

共享或打包 GPU 的机制回答的是另一个问题。协同服务（ROSE、Prism）会在相同的 GPU 上保留两个模型。它会占用在线引擎的 KV 缓存设备内存：一个 373 GB 的借用方会在八张 141 GB 的卡上各占用约 47 GB，即占内存的三分之一。GPU 目前也无法足够细粒度地分区以隔离二者，因此我们会移交整组卡。整合 长尾服务到更少的卡上（Aegaeon、Prism）会改变某项服务所需的卡数，但其本身并不能为低层级提供服务；剩余卡在突发流量之间的时间窗口，才是借用机制所利用的部分。我们的 GPU 集群显示了仍有多少资源可用。在约 4,000 张卡中，约 3,000 张属于完全没有流量的服务；整合可以处理这些卡，因此我们将其排除在外。在常规周内有流量的服务所占用的 1,100 张卡中，平均而言任意时刻约有 507 张可借用，并且在 95% 的分钟内至少有 342 张可借用（第7.1节）。最后，请求级 *KV* 策略 （例如 InferCept 的策略）决定引擎如何处理其保留的请求；它们会保留 GPU，而借用机制则将 GPU 交给另一个模型。

3 资源模型与架构

3.1 对象与身份

ModelLess 将六种对象串联起来。环境是基础运行时环境和节点信息。目录 条目包含一个基础模型的启动参数和基础权重。基础运行时 （或一组基础运行时）是部署在硬件上的目录条目，而运行时 （或一组运行时）则是能够提供服务的基础运行时。模型服务 是对外提供服务的模型及其容量，而请求 是对该模型的一次调用。这条链实现了不同生命周期之间的解耦，也是核心设计选择之一。基础运行时——包括引擎进程、编译后的内核、CUDA 图和基础模型布局——可持续运行数小时或数天，而强化学习权重修订的生命周期只有一个训练步骤（数分钟），请求的生命周期则从数秒到数分钟不等。传统服务副本将这些生命周期绑定在一起，每当权重或服务形态发生变化，都必须重启整个引擎。ModelLess 将声明的内容（目录条目和模型服务）与实际运行的内容（基础运行时和运行时）分离。基础运行时保存目录条目的执行状态和基础权重，不带优先级或工作负载形态；运行时则将基础运行时与模型服务和权重版本绑定。不同形态和并发度的工作负载，包括在线、作业和弹性服务，位于这条链之上，其下的任何基础运行时都可以承载这些工作负载。因此，在服务之间迁移卡组，或让强化学习 rollout 随连续的权重修订推进，只需在同一基础运行时上重新绑定，而无需重启进程；调度器可以重建运行时，而无需

表 2。 需求及其对应的解答。

需求	证据	解答（章节）
以秒计的恢复时间，以及较短的让出时间	表 8：冷启动不计入服务等级目标；C = 5s 在严格门槛下，相较于 60 秒更为重要	权重存储在主机内存中的休眠/唤醒（§5.1）；通过暂停或排空实现产出（§5.2）
借用方经常被抢占，时间窗口在总体上较长	图 2；表 1	分层驱逐语义（§4.2、§5.4）
保留重建或重新计算成本高昂的内容	权重重新加载需要几分钟；重新计算长上下文需要几十秒（表 11）	休眠/唤醒会保留权重（§5.1）；键值缓存和请求按层级保留：弹性场景下暂停（§5.2），否则重新入队或进行重计算续接（§5.4）
任务执行期间权重会发生变化	强化学习 rollout 的设计要求	直接推送、Seal、Follow（§5.3）

改变客户端所见的内容，而且请求所属的层级永远不会固化在引擎进程中。

身份与位置相互独立。一个运行时由一个引擎进程提供服务，我们也称之为实例，并通过稳定标识符进行标识；其端点仅表示当前可访问的位置，重建后可能发生变化，因此没有任何组件会将端点用作身份。目录条目说明哪些引擎可以承载该条目，它通过标签和版本构成的能力选择器来描述，而每个 modellet 都会报告这些标签和版本；这类似于节点向集群调度器公布其内核版本：只有当报告信息的节点满足选择器时，调度器才会选择相应的运行时。逻辑权重标识符用于命名权重版本，空标识符表示基础模型，另一个独立字段则给出其物理来源。

3.2 API 定义

用户负责声明和调用，而不是放置引擎。平台管理员创建目录条目，用户则创建一个模型服务，指定一个目录条目，可选地指定权重来源、默认层级、其请求可使用的层级集合，以及副本范围（最小值和最大值）。min 表示保持就绪的运行时数量；低于该数量时，服务即未满足要求，并且该值可以为 0。容量配额与物理 GPU 驻留相解耦：就绪的运行时可以处于休眠状态且不占用 GPU 内存，同时仍计入 min（并且不产生 GPU 费用），因此预留容量在逻辑上得到保证，仅在需要时才按需进行物理实例化。max 是按需扩展的上限，并不承诺服务创建时就达到该数量。权重来源可以是 local（路径）、Hugging Face（代码仓库）或 NCCL；最后一种仅允许用于服务，不能用于目录条目，因为目录条目必须能够冷启动其基础模型。对于 NCCL 服务，ModelLess 使用基础权重启动运行时，并返回它们的端点、秩偏移量和世界大小；训练器据此组建自己的通信组并推送权重（§3.4）。更改非空权重标识符背后的物理来源需要创建新服务，从而确保运行中的逻辑权重不会被静默替换。

请求是普通的 OpenAI、SGLang 或 vLLM 风格调用。其 model 字段用于指定服务，也可以附带权重标识符，该标识符必须与服务的绑定相匹配，

表 3. ModelLess 组件。

组件	位置	Role
modelentry	CPU	适配协议并保留客户端连接；转发、排队或由调度器决定是否拒绝；负责面向客户端的请求身份；位于弹性层级。
modeldispatcher	集群	掌握全局视图；做出每个部署位置决策（转发、排队或拒绝请求），并驱动引擎横向扩展和缩容。
modellet	node	承载引擎进程并管理本地 GPU 分配；确保同一时间最多只有一个引擎处于卡组活跃状态；修复引擎。
引擎插件	引擎	公布其支持的操作引擎支持；执行休眠、在引擎内部唤醒和暂停进程 .

还可以指定一个层级，该层级必须属于服务允许的层级之一，否则默认为服务的层级。请求从不携带目录、运行时或端点标识符；部署位置由调度器决定。运行时和运行时集合完全由 ModelLess 创建并确定规模：运行时集合的副本数量只能由服务控制器设置，就像 Kubernetes ReplicaSet 一样；操作员可以删除集合，但只要其服务仍在运行，ModelLess 就会重建该集合。兼容性要求严格：如果没有任何上报节点满足目录条目的选择器，服务将保持不完整或被拒绝，绝不会降级为不兼容的引擎。

3.3 组件

ModelLess 位于推理引擎与业务网关之间，由四个组件分担工作（表 3、图 3）。它不是推理引擎，也不是网关（不执行身份验证、配额或计费），更不是 Kubernetes 控制器；Kubernetes 仅负责最初启动这些组件。计费由 ModelLess 之外的系统负责，并根据服务占用 GPU 的时间收费。休眠实例不占用任何 GPU，也不会产生计费，即使它仍计入最小数量。

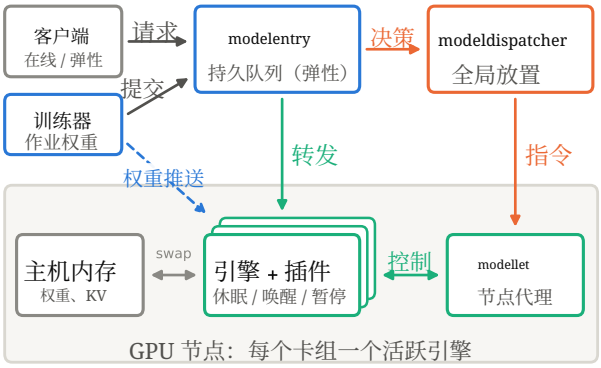

图 3. ModelLess 架构。每个节点由一个 modellet 管理多个引擎；虚线箭头表示训练器向作业引擎推送权重。

3.4 请求

请求以三种形式之一到达 ModelLess，每种形式分别对应 §4.2 中的一个层级，并展示了端到端的控制路径。

在线。 请求到达 modelentry 后，modelentry 会向调度器注册该请求，以请求一个服务引擎。如果某个活动引擎的并发度低于其并发阈值（§6.1），调度器将返回转发决策和端点，请求以毫秒级成本完成转发。如果所有活动引擎都已达到其目标并发度，调度器会首先尝试扩展：如果某个休眠引擎可以在数秒内唤醒，调度器将返回排队决策，通知 modellet 唤醒该引擎，而 modelentry 会保持连接，直到引擎报告健康状态；如果扩展需要冷启动或无法继续，调度器会先将请求分发到现有活动引擎，直至达到各自的硬并发上限，然后再进行排队或拒绝（§6.1）。如果等待时间超过其上限，或无法获得容量，调度器将拒绝该请求，客户端可以重试。调度器将快速的逐请求决策与较慢的引擎扩展分离为两个相互独立的循环，并通过任务队列连接二者，从而避免任何一方阻塞另一方。

作业。 训练框架提交一项任务，并指定权重版本、模型以及最小/最大引擎数。调度器选择卡组并启动最初的最小数量引擎，将其标记为需要封存，并加载基础权重，而非该步骤的权重，因为 ModelLess 负责进程、内存和调度，而训练端负责推送每个步骤的权重。由于基础运行时会在各训练步骤之间持续存在，推进到新的权重版本时，会在相同的基础运行时上重新绑定运行时，而不是重启引擎进程。调度器在引擎启动时返回其端点，训练框架则在这些引擎上建立自己的通信组并推送权重。当最小数量的引擎都收到权重后，提交即告完成；ModelLess 通过训练器向每个引擎发出的首个生成请求来推断这一点。生成请求会经过 modelentry，由其转发；返回的端点仅供训练器推送权重使用，因此无需额外的协议消息。使用这些权重成功完成首次生成后，ModelLess 会封存该引擎的权重。并发度上升时，调度器会沿用已封存的引擎，将引擎数量扩展至最大值，而新引擎可能会被更高层级抢占。释放请求会使引擎进入休眠，并在其引用计数归零且经过宽限期后，将该权重版本交由垃圾回收。每个模型保留的权重版本数量有上限，以保护主机内存。

弹性。 客户端将请求提交到持久队列，并轮询其结果。调度器会将请求交给完全空闲的卡上的引擎，因为弹性只借用整个空闲引擎，而 modelentry 会在队列中记录每次尝试。如果更高层级需要这些卡，该尝试会被隔离，请求会重新入队；如果引擎支持暂停，则请求会在暂停状态下等待（§5.4），客户端会继续轮询同一个请求。

4 实例状态与层级语义

大语言模型实例具有四层状态（§4.1），而驱逐成本取决于其中哪些状态得以保留。每个层级规定其客户端能够容忍哪种损失（§4.2）。这决定了平台在发生抢占时保留什么、采用哪种操作（§5），以及控制平面采取行动的顺序（§6）。层级限定了损失范围，而操作则决定该范围内的成本。表 4 按列展示每个层级，内容从其能够容忍的损失到平台采取的措施。作业引擎在弹性之后、且仅在超过最小值时被抢占。

4.1 实例状态

一个推理实例包含四类状态，而每种交接机制本质上都是在决定保留其中哪些状态。执行层是进程及其编译产物（CUDA 图、内核、预热）。权重层是模型参数，实例提供服务时存储在 GPU 内存中。键值缓存层是正在生成请求以及最近完成请求的注意力缓存。请求层是进行中的请求本身：它们的令牌前缀、采样状态，以及等待这些请求的客户端连接。

冷启动会重建全部四类状态。只有保留重建成本高昂的层级，交接才会便宜。权重和执行状态需要数分钟才能重新加载并完成预热（GLM-5.2 模型需要 244–277 秒）。键值缓存和请求状态占用的保存成本很低，但一旦释放 GPU 内存，它们就会立即丢失。第 §5 节中的操作在保留哪些层级方面各不相同。

4.2 请求层级

层级不只是优先级区间：这三种工作负载也处于不同的延迟–吞吐量工作点。由于大语言模型解码延迟会随批大小增加而恶化，而吞吐量则会随之提升，在线实例必须保持较低的

表格 4. 层级及其驱逐语义。

	在线	job	flex
交付	同步，连接保持	任务提交	异步，持久队列（提交/轮询）
容忍	执行中无任何内容：从不暂停，被迁移或被较低层级抢占	减速和返工（迁移），而非无限期停滞	停滞与延迟，而非丢失
抢占期间保留的状态	执行中：全部（从未被抢占）；停放于 θ 空闲：权重位于主机内存中，而非键值缓存	权重以及已休眠请求的 KV 缓存引擎；进行中的请求继续从其生成的令牌中转移到其他位置	权重和前缀缓存（重新入队），或也包括进行中的请求（暂停）
操作	没有进行中的请求；在.....时休眠/唤醒	在引擎上进行重计算续接	隔离重排队，或在引擎支持的情况下暂停
容量声明	在 θ 空闲后停放 服务副本范围；高值为预留	仅高于最低值 最小值（预留）和最大值（扩展上限）	服务副本范围；使用剩余的部分
并发目标	较低的建议并发数/最大并发数，保障延迟（§6.1）	较高的建议并发数/最大并发数，保障吞吐量	较高的建议并发数/最大并发数，保护吞吐量
当容量被抢占时	永远不会被作业或弹性抢占；高置于默认之前，并且一个高需求的默认引擎会被排空	最小引擎数量以上的引擎上的请求迁移（重新计算续接）；最小引擎永远不会被抢占	请求被隔离并重新入队；已暂停而是在引擎支持时
典型用途	面向用户的服务	强化学习 rollout，数据生成	离线批量推理
第7节中的评估	轨迹回放（第7.1节） + 1-节点端到端（第7.3节）	仅限分析（表12）；跟随/封存未进行端到端评估	单节点端到端（暂停与重新入队，第7.3节） + 解析（表12）

表 5. 抢占时保留的状态层，按层级划分。

层	在线、执行中飞行	在线、停放于 θ 空闲之后	job	flex
执行（进程、编译产物）	kept	保留（休眠）	保留（休眠）	保留（休眠）
权重	保留在 GPU 上	保留在主机内存中	保留在主机内存中	保存在主机内存中
KV 缓存	kept	丢失；下一个请求开始 cold	保存在主机缓存中 休眠的引擎；迁移的请求在目标上重新计算它	由.....保留的前缀缓存 直写（重新入队）；所有键值缓存保留（暂停）
进行中的请求	从不被抢占	没有执行中的任务	从其所在位置在其他地方继续 生成的令牌、键值缓存 重新计算的任务	进度丢失后重新入队，或暂停后恢复

并发度，以保障首令牌时间和令牌间延迟；而作业和弹性实例则倾向于较高的并发度，以充分利用 GPU。ModelLess 并未为每种产品分别运行一个调度器，而是通过各层级建议的并发阈值和最大并发阈值（表格 4）来设定这一工作点，并在所有层级中应用统一的调度器策略（§6.1）。每个层级还规定了工作如何交付、申报何种容量，以及其卡被抢占时会损失什么。该实现包含三个层级，优先级依次为在线 > 作业> 弹性，其理念类似于集群管理器中的优先级类别 [64]。

各层级的差异在于其客户端能够容忍哪种损失。抢占可能使请求损失其工作（请求被中止）、进度（需要重新计算）、速度（继续执行，但速度变慢），或时间（在容量恢复前停顿）。在线 执行中的流量无法容忍上述任何一种损失：用户正在等待每一个词元。因此，作业和弹性永远不会暂停、迁移或抢占在线请求。已空闲 θ秒的在线服务可能会被停放，而它在此之后收到的第一个请求需要承担恢复停顿。作业 流量是代表训练器发出的请求批次，训练器会等待其完成，例如强化学习 rollout 或数据生成。作业不能暂停，因为另一端的训练器会被阻塞。被阻塞的训练器会使其自身的 GPU 空闲，而这些 GPU 的数量通常是轨迹生成模型规模的许多倍，因此一次停顿会浪费相当于该作业 GPU 时间数倍的时间。作业可以容忍减速。其在高于最小配置的引擎上的请求可以迁移。它们会在其他位置继续执行，经历一个有界停顿，即重新预填充所需的时间，而不是等待原来的卡恢复，同时运行速度也会变慢。作业更看重高并发度，而非令牌间延迟。弹性 流量没有持久连接。用户向消息队列提交请求，并轮询进度，因此停顿只会损失时间，不会造成其他损失。只有请求丢失是不可接受的。只有弹性客户端能够等待停顿中的请求，因此暂停属于弹性。

在线流量具有最高优先级，其驱逐语义分为两部分。只要在线实例有流量，较低层级就永远不会抢占它。当它在 θ 秒内没有收到请求时，可以将其停放：其权重保留在主机内存中，而其 KV 缓存会被丢弃；之后到达的第一个请求需要等待第2.5节中的恢复延迟 C。第7.1节的轨迹研究衡量了这一操作的成本。作业和弹性流量永远不会抢占在线引擎或其进行中的请求，较低层级也不会与在线流量共享引擎。

弹性只借用完全空闲的引擎；在线流量到达时，弹性会归还整个卡组。引擎正在唤醒期间到达的请求会在队列中等待。

该层级包含两个层级。*Online-high* 是预留层级：其准入检查保证逻辑容量配额（集群只准入能够同时处于活跃状态的高优先级运行时数量），但不要求这些运行时在空闲时始终驻留于 GPU 内存中。两个争用同一卡的高优先级服务无法同时保证被激活。高优先级容量仅服务于在线流量：作业引擎永远不会被放置在该层级上，而借用该层级的弹性占用者会在高优先级引擎必须激活时立即暂停或重新入队。*Online-default* 是超额承诺层级，叠加在高优先级引擎之后可能需要使用的卡上。当找不到可用容量时，向默认服务发出的请求可能会被拒绝；而只要其创建时执行的准入检查仍然成立，高优先级服务就会被唤醒。高优先级引擎不会抢占默认引擎，而是将其排空。默认引擎不再接收新请求，完成其进行中的请求，然后交出所占用的卡，因此高优先级引擎会在最长默认请求完成并经过唤醒时间 C 后激活。由于较长的默认生成过程会延迟激活，将默认服务叠加在高优先级卡上，意味着默认请求时长必须有界，或采用保守的超额分配；若要强制更早地产出，则需要能够容忍迁移的客户端（§8）。进行中的默认请求永远不会被迁移。

作业 用于任务型吞吐工作负载。提交时需声明最小值和最大值。最小引擎数属于硬预留，也是准入阈值：至少有最小数量的引擎收到作业的权重后，提交才算完成。权重流从推送到首次生成的过程见 §3.4，封存规则见 §5.3。超过最小数量的引擎用于扩容；当更高层级申请其卡时，这些引擎会被抢占。最小数量的引擎绝不会被抢占，在就绪之前也绝不会进入休眠。作业引擎休眠后会在 §5.4 所述的主机缓存层中保留其 KV 缓存。智能体工作负载需要这一点：其请求在多个轮次间共享长前缀；如果没有缓存，该引擎后续每一轮都必须重新计算整个上下文。迁移到其他引擎的请求无法从中受益。由于所有活跃作业的最小值都会降低在线突发流量可用的容量，因此最小值应设为任务所需的数量，而不是能获得的最大数量。

弹性 适用于异步工作，其保障是：已接受的请求不会丢失，并会在容量可用时完成，但不保证带宽或延迟。请求存放在持久队列中，因此被抢占的请求不会丢失（§5.4）。弹性是最低层级：可能被在线流量和作业流量抢占，且不会抢占其他流量。

表 5 用 §4.1 的术语重述了这些语义。它展示了每个层级强制平台保留的层，以及保留与否取决于成本选择的层，例如暂停时的请求层，或直写时的 KV 缓存。

平台的商业产品映射到各个层级：在线产品属于在线层，异步和批处理产品属于弹性层，而展开属于作业层。具有不同延迟目标的在线产品，其差异在于服务的配置方式，例如允许空闲多久后进入休眠，而不在于层级。ModelLess 不会编码产品名称。

除一种情况外，层级与权重来源彼此独立。从本地路径或模型中心加载的基础权重和派生权重，可在任意层级提供服务。训练器通过集体通信组实时推送的权重，仅允许用于作业层，因为它们依赖训练端会话（组创建、推送、验证），而其生命周期只有作业协议能够提供。作业仍可使用基础权重：作业层采用任务协议，并具有预留和可扩展的容量。

5 交接操作

5.1 休眠与唤醒

休眠 会释放实例的 GPU 内存，同时保留其执行层以及存放在主机内存中的一份权重副本，方式类似于引擎休眠模式和张量释放库[62, 65]。实例仍保留在路由表中，因此休眠期间到达的请求会触发唤醒，而不会失败。唤醒会将权重从主机内存恢复到 GPU 内存；预热和图捕获已在实例首次准备时完成，因此唤醒时只需复制。准备步骤（预热）会让每个已声明的实例完整经历一次冷启动，然后使其休眠，从而为每个实例在主机内存中留下一份权重副本；主机内存必须据此配置。在我们的 2 TB 节点上，一名 GLM-5.2 借用方的权重备份（373 GB）、可配置的主机 KV 池以及在线引擎的备份共占用约 1.5 TB（§7.3.1）；权重备份的占用随驻留模型数量扩展，而主机 KV 池则按部署配置。只有当请求所需的模型从未经过准备，或所有已准备的实例都正忙时，才会回退到冷启动。

此处定义的休眠适用于空闲实例：modellet 等待进行中的请求完成后再释放。为此调用引擎自身的生成暂停会冻结其调度器，导致排空永远无法完成，因此普通路径不会使用该功能。测得的成本见第7.2节。排空需要等待耗时最长的进行中请求，而改为中止则会丢失工作，因此不能接受工作丢失的层级无法采用这种方式；下一个操作正是为它们提供的。

5.2 暂停与恢复

暂停是指实例在仍有请求执行中时进入休眠。它不会排空或丢弃这些请求，而是将其保留。model- 随后关闭该实例的准入门控，并按固定顺序执行检查点操作。首先，每个执行中的请求完成当前步骤，并将已提交前缀截取至

最后一个完整页面边界。其次，将每个请求的精确词元前缀，以及其已提交且按页面对齐的键值缓存部分（分页 KV [28]），写入主机内存中的分层缓存 [76, 85]。接着，释放这些请求在设备端的键值缓存。最后，将三个 GPU 内存分区（权重、KV 缓存、计算图）换出。恢复按相反顺序执行：恢复各分区，从主机内存恢复每个请求的前缀，重建最后一个不完整页面，重建解码批次，并恢复采样状态，例如记录已见词元以便应用重复惩罚。每个请求都会从暂停处继续，无需重新预填充其已提交前缀。只需重建最后一个完整页面边界之后的词元，因此在常见情况下，每个请求的重建量不到一页（64 个词元）；在 GLM-5.2 上，中位数为 33 个词元，最大值为 62（第7.2节）。引擎的重新计算词元计数器统计主机备份未覆盖的已提交词元，因此不包括这部分重建；在每个微基准测试单元中，该计数均为零（第7.2节）。

检查点采用故障关闭策略。引擎依次经过活跃、检查点保存、休眠和恢复中状态；任何步骤失败都会将其置于需要恢复状态，在实例恢复或重建之前，准入始终保持关闭。实例绝不会在检查点仅写入一半时提供服务。暂停实例的卡会为其恢复而保留。调度器不会将这些卡分配给其他低层级任务，只有发起抢占的层级可以使用它们。等待恢复时间超过回退期限（默认十五分钟）的检查点，会在被丢弃前通过隔离重排队（§5.4）退出。

只有声明支持暂停功能的引擎才能使用暂停（引擎插件会通过版本和配置协商该功能）。不支持该功能的引擎在排空后会退回到普通休眠。这里有两个语义限制。第一，恢复会还原完全相同的前缀和采样计数器，因此在确定性推理下，贪心解码请求会逐词元继续执行（已在 Qwen3-30B-A3B 上验证，并在 GLM-5.2 上针对单个请求验证，§7.2）；对于批次相关内核，它会从相同状态继续，但可能存在数值非确定性。使用温度采样时，恢复后的前缀和权重在位级别完全一致，重建的部分页面只需进行一次小规模预填充，因此每个词元都从相同的条件分布中采样，但可能存在数值非确定性；不过，采样路径会不同于未中断的路径。§7.2 报告了我们的验证结果。第二，处于预填充阶段的请求不会在预填充步骤中途保存检查点：暂停会等待正在运行的预填充步骤完成，因此新增的产出等待时间最多为一个预填充步骤；启用分块预填充时，该步骤就是一个分块。

5.3 跟随与封存

强化学习 rollout 引擎所服务的权重在存储中无处安放：训练器每一步都会生成一个新版本，并通过自己的通信组将其推送到各个引擎[13, 56, 67], 因此，唯一的副本保存在已经接收到这些权重的引擎中。*Follow* 使新引擎能够从这样的引擎启动，而无需读取存储：它会从同一模型、同一权重版本的健康实例复制权重，该实例已接收到当前步骤的权重。在同一节点上，源实例会将权重写入只读共享内存快照，目标实例再对其进行映射；跨节点时，复制通过远程直接内存访问完成。*Seal* 是一项规则，用于确保权重为空或尚未推送时的安全性：只有快照与 GPU 内存中的权重相匹配的权重版本才能作为源，而且只有实例使用这些权重成功生成内容后，才会创建快照。未封存的实例绝不会作为源，因为从中复制可能会带来空权重或过期权重。控制接口是异步的，并针对每个请求标识符实现幂等性；它会拒绝过期的实例身份，并在传输期间对两个端点实施栅栏。

5.4 抢占容量

当较高层级需要较低层级持有的卡时，较低层级进行中的请求将如何处理取决于其层级，具体总结见表 6。在线流量不会被较低层级抢占，这与生产集群中的优先级方案一致[64]：较低层级不会抢占在线引擎或其请求，并且较低层级只会借用完全空闲的实例。

对于作业 层级（rollout 类工作负载），超过声明的最小数量的实例会通过重新计算续接被抢占。重新提交提示以及截至目前生成的令牌，与 RLBoost[71] 的词元级响应迁移相近。SpotServe 的有状态恢复也会从已提交的令牌恢复中断的请求，但它会在实例之间迁移其键值缓存 [40], ；Llumnix 对运行中的请求也是如此 [60], ，而 Mooncake 则会在预填充池与解码池之间移动键值缓存 [49]；我们的重新计算续接不会移动键值缓存。我们出于两个原因采用重新计算：迁移键值缓存会增加工程复杂性，而重新预填充对于作业的工作负载而言足够快。同样的限制也适用于暂停：其检查点位于某个节点的主机内存中，并在同一卡组上恢复，因为在其他位置恢复将意味着迁移键值缓存。对于重新计算续接，模型入口组件会累积流式请求已经生成的部分输出（提示词 token 标识符、生成的令牌标识符、文本），然后取消该请求。随后，它会将提示以及已生成的令牌重新提交到使用相同模型和权重版本的另一个实例，并将生成预算减去已经生成的部分。客户端会看到完整的输出以及短暂的停顿。目标实例上仍然存在的前缀缓存 [85]可以降低成本，并且该循环有次数限制（默认尝试两次），以防止迁移风暴。

对于弹性层级（排队的异步工作），被抢占的请求会被隔离并重新入队：modelentry 首先通过比较并设置操作，在持久队列中标记当前尝试并入队一次新的尝试，随后才中止引擎上的请求。该顺序确保请求不会

表 6。 按其保留的状态列出的操作和抢占路径。“成本”指第7节中的测量结果。 2.

机制	适用于	保留	丢弃	请求 执行中	未能	Cost	评估
休眠/唤醒	空闲实例	执行、权重 （主机副本）	GPU 内存；键值缓存 除非另有说明 传输到主机 缓存（借用方 引擎，§5.4）	none （先排空 ）	重建或冷 启动	释放 ~1s，唤醒 2.9–4.4 秒 （GLM-5.2，TP8）	第7.2节，第 7.3节
暂停 / 恢复	繁忙实例 （能力 必需）	执行， 权重、KV 前缀、请求， 采样计数器	GPU 内存，最后 部分页面	保留， 已恢复 完全	恢复- 所需， 准入已关闭	每次约 5–9 秒 暂停–恢复 包括键值缓存 从主机重新加载 (~3 μ秒/词元)	第7.2节，第 7.3节
重新入队 （弹性）	异步层级	权重；前缀 缓存（如果保留）	解码进度 （如果提示词 KV 缓存未保留）	已中止， 重新运 行	持久队列	丢失的解码，或完整的 提示词重新计算	第7.2节，第 7.3节
重新计算 续接 （作 业）	高于最小值 实例	权重	迁移的请求的键值缓存 请求（一个休眠的 引擎保留其 主机键值缓存）	re- 预填充 其他位置	有限次数重试	重新预填充 提示加上 生成的令牌	实现； 仅限分析 （第 7.2节）
Follow	横向扩展	—	—	—	如果来源不明则拒绝 未封存或过时	封存权重的副本 来自某个 peer	实现；不 端到端评估

既未丢失（它已重新入队），也不会被交付两次（旧尝试已被标记，因此其结果不会被接受）。如果引擎支持暂停，弹性请求也可以保存在检查点中并恢复，从而避免重复和重新计算；如果不支持，则应采用重新入队的方式。

重新入队和重计算续接具有暂停所没有的条件。重新入队仅在尚未交付任何输出之前有效，因此适用于异步工作；已重启的流式请求会重复其已经发送的输出。其成本还取决于借用方的前缀缓存是否在释放后保留：缓存保留时，重新入队的提示几乎完全命中缓存，只有解码进度会丢失；没有缓存时，则需要重新计算整个提示（第7.2节）。只有弹性客户端在请求完成前看不到任何输出，因此重新入队仅限于弹性客户端。

保留的缓存位于引擎的主机内存中的分层缓存内，已提交前缀会在释放前直写至该缓存 [76], 。社区引擎休眠时会丢弃前缀缓存，因此唤醒后的引擎需要对每个提示进行完整的重新预填充（第7.2节）；平台因此会将已提交前缀直写至主机层。采用直写后，重新入队的提示会命中主机层，只有解码进度会丢失；这使重新入队成为比普通休眠更强的基线，同时暂停所依赖的也是这种直写。引擎一旦宕机，主机层也会随之丢失，后续的重放会对其提示进行完整的重新计算。

只有在释放内存之前关闭准入门控，抢占才是安全的。从决定抢占到释放完成这段时间内，引擎不得接受新的生成过程。若有请求在此期间获准进入，就会要么固定住释放操作无法处理的状态，要么被路由到内存已被释放的引擎。暂停路径会在第一步关闭门控；重新入队路径也会显式执行相同操作。这是协议的要求。

5.5 总结

表 6 总结了各项操作和抢占路径，包括每项操作所保留和丢弃的状态、如何处理执行中的请求、各自如何失败，以及所产生的成本。

6 控制平面与安全性

6.1 放置与牺牲者顺序

ModelLess 中的部署位置由需求驱动：调度器不依赖流量预测在流量突发前预热容量，因为以秒为尺度的唤醒机制可以让实际请求到达直接使休眠容量恢复可用。当请求需要一个引擎时，调度器使用每个层级的两个并发度阈值进行路由——一个软*建议并发度*，用于标记实例的延迟–吞吐量工作点（在线场景较低，作业和弹性场景较高，见表 4），以及一个硬最大并发数上限——按以下有序路径执行：

1. **低于建议并发度进行装载。** 如果一个或多个活动引擎低于其建议并发度，调度器会将请求装载到一个活动引擎上（优先选择已经承载请求的引擎），从而让备用引擎保持空闲、进入休眠，并可借给低层级。

2. **达到建议并发度时扩展。**当每个活动引擎都达到建议并发度时，

调度器会在现有引擎过载前尝试增加容量：首先唤醒一个模型和权重版本相同的休眠引擎（抢占持有其卡的任何借用方），然后从一个已封存的活动对等引擎复制权重（Follow，§5.3）；若仍不可行，则抢占较低层级的占用者（§5.4），并唤醒或冷启动一个新引擎。快速唤醒期间，到达的请求会等待正在唤醒的引擎，而不会让活动引擎的并发度超过建议值。

3. **冷启动或达到上限时的溢出轮询**。 如果扩容需要数分钟才能完成冷启动，或者由于服务已达到最大副本数或无法回收容量而无法继续，调度器不会让请求阻塞在冷启动上；相反，它会按轮询顺序将请求分配给现有的活动引擎，直至各引擎达到硬性最大并发度。只有当所有活动引擎都达到最大并发度时，后续请求才会排队或被拒绝。

较低层级永远不会抢占在线流量。受影响的对象按层级顺序选取：先选弹性占用者，再选超过最小引擎数的作业引擎；在在线引擎中，为高优先级引擎排空默认引擎；作业的最小引擎永远不会被抢占。

6.2 安全不变量

部署位置决定哪些对象获得这些卡。控制平面保留九项不变量，确保在并发变更下，每次所有权变更都是安全的。每项不变量都通过步骤顺序或身份检查来强制执行，并且各自防止一种特定的竞态。我们通过检查步骤顺序和身份检查来验证这些不变量，而非注入故障。

1. **准入在内存释放前关闭**。 正在被抢占的引擎，其门控机制会在释放任何内存之前关闭（§5.4）。释放操作首先执行与休眠相同的空闲检查。某个请求的客户端可能已经收到答案，但引擎仍在解码该请求的最后几个词元；如果释放跳过检查，就会中止该请求。唤醒时，门控机制会保持关闭，直到引擎完全恢复，因此重新入队的请求浪潮无法到达仍处于恢复中的引擎。缓存直写后，释放操作还会检查是否仍有设备内存残留。

2. **中止前设置栅栏**。 弹性请求会通过比较并交换在持久队列中标记，并在其于引擎上的尝试被中止之前重新入队，因此既不会丢失，也不会被交付两次（§5.4）。排空或抢占会在其源端执行相同的比较并交换栅栏，因此决策之后发生的状态变化会中止该操作。

3. **故障关闭**。 暂停步骤故障会使引擎处于需要恢复状态，并关闭准入；引擎绝不会从写入一半的检查点提供服务（§5.2）。空闲检查故障时也会故障关闭。暂停期间进行缓存刷新会使检查点失效，因此

暂停与任何清除缓存的操作都会彼此排序，检查点则是一个事务。4. **每个卡组仅有一个活动引擎**。 modellet 会在每个节点上强制执行这一规则。5. **拒绝过时指令**。 每条指令都会标明确切的运行时身份和实例代次；由于引擎此后可能已经重建，modellet 会拒绝任何实例代次已过时的指令。6. **唤醒时绝不超额分配**。 在唤醒引擎之前，modellet 会在本地检查唤醒是否适配节点，否则将拒绝该操作（§6.4）。7. **作为源之前先 Seal**。 只有在引擎使用相应权重成功完成一次生成后，才能复制这些权重（§5.3）。8. **传输期间进行安静探测**。 在硬件级传输期间，两个端点都会被隔离，健康探测会降级为进程存活性检查。读取或写入正在移动的内存的存活性探针可能会导致引擎崩溃。9. **在线状态绝不会被较低层级抢占，并且作业的**最少 **引擎绝不会被抢占，在准备就绪之前也绝不会进入休眠** （§4.2）。

6.3 引擎状态与故障

ModelLess 外部可见的状态有三种：活跃、休眠和恢复中。ModelLess 还保留创建中、故障 和销毁中这几种状态，各有其原因。之所以设置创建中状态，是因为冷启动需要数分钟，在此期间，引擎既不能被路由到，也不能再次启动。设置故障状态，是为了区分两种情况：一种是需要重建的故障引擎，另一种是只需唤醒的休眠引擎。设置销毁中状态，是因为被终止的进程需要时间才能退出；如果进程已经退出而引擎仍处于该状态，它便可在无需重启 modellet 的情况下恢复。暂停步骤失败会使引擎进入 §5.2 所述的需要恢复状态，而 modellet 会在无需操作员干预的情况下重启不健康引擎。

6.4 控制平面故障

两项规则可防止控制平面的故障演变为数据平面的故障。首先，在唤醒引擎之前，modelentry 会在本地检查唤醒操作是否会导致节点超额分配；如果会，则拒绝执行该指令，从而即使决策有误或已过时，也不会超额分配卡。其次，每个 modelentry 都会保留最近可用副本的列表，并以约 30 秒的间隔从集群元数据刷新该列表。调度器是单个实例；当其发生故障时，每个 modelentry 都会进入故障转移模式，将请求转发给其内存列表中的副本，同时禁用所有唤醒和休眠操作，因为已没有调度器负责协调这些操作，而未经协调的变更会相互冲突。活跃引擎上的服务仍会继续，调度器恢复后，容量变更也会恢复。因此，单个调度器是容量变更的故障点：在服务中断期间，一个已扩展至零容量的

服务无法唤醒，因此其请求会失败；而持有已停放所有者的卡的借用方无法被抢占，因此该所有者会一直处于停放状态，直到调度器恢复。

7 评估

我们评估在线层和弹性层，并以解析方式为作业层定价（表12）。第7.1节测量借出资源对在线层造成的成本：作为恢复延迟 C 的函数，可以让多少容量进入休眠，以及之后的第一个请求需要付出什么代价。第7.2节测量保留或重建实例状态的每一层所需的成本。第7.3节在一个节点上运行弹性层。它首先检查没有弹性请求丢失且每个请求都能完成，然后比较暂停和重新入队这两种实现相同层级语义的方式，以了解保留请求层的价值。

轨迹研究。 第7.1节以一秒为分辨率，重放第2.2节所述服务在两个为期一周的时间段内由网关记录的请求到达序列，并使用离散事件模型模拟资源停放。这是由生产环境到达流量驱动的模拟，而非部署测量。**机制测量**（第7.2节）在 8 × H20（141 GB）节点上进行。大型模型的结果使用 GLM-5.2（W4AFP8，373 GB）并采用张量并行度 8；较小的模型使用一到四张卡。暂停实验使用包含引擎休眠扩展的冻结版 SGLang 构建。

7.1 轨迹回放

方法。 对于每项服务和每个空闲阈值 $\theta \in \{30, 60, 120, 300, 600, 1800\}$ 秒，我们连续回放一周。服务在 θ 秒内没有收到请求后便会停放；下一次到达事件会触发恢复，这需要 C 秒，而恢复期间到达的每个请求也都必须等待。我们将触发恢复的请求及这些等待中的到达事件称为延迟请求。我们对延迟请求评估每个 SLO 门控，因为只评估触发请求会在 C 较大时低估影响。我们评估三种恢复延迟。C =5 秒表示休眠/唤醒，权重保留在主机内存中。C = 60秒表示假定重启，权重已预加载到内存中；我们尚未测量这种情况。C = 240 秒表示重启并从磁盘读取权重；这一时间低于我们测得的 244–277 秒冷启动时间，因此对冷启动而言是保守估计。表 7 定义了各项门控机制。服务在某个阈值下通过门控机制时，即为可回收；其可回收份额是通过门控机制的最佳阈值下的停放份额，而 GPU 集群的数据则为按卡数加权的平均值。事后为每项服务选择最佳阈值会显得过于乐观；因此，我们同时报告固定-θ 数据，并通过对服务进行自助抽样来计算 95% 置信区间。

冷启动在服务等级目标下几乎无法提供可回收容量。 当 C = 240 秒时，69–85% 的服务超过 0.5% 的超时阈值（范围从 θ 300 秒到

表 7。 重放中使用的 SLO 门限。门控机制按服务进行评估，针对该周内的延迟请求。P50、P90 和 P99 门限界定了等待恢复的请求所占比例；它们并非绝对的首令牌时间百分位数，图 4 按服务展示了后者。

Gate	条件（按服务）
avg	平均 TTFT 增幅 $\leq$ 2 秒
P50	延迟请求占请求的 $<$ 50%
P90	延迟请求 $\leq$ 10%
P99	延迟请求 $\leq$ 1%
超时	90 秒超时率 $\leq$ 0.5%（更严格的 SLO）或 $\leq$ 5%（标准 SLO）
中位数和超时	P50 ∧超时（更严格的 SLO）；P50 ∧ 可用性 $\geq$ 95%（标准 SLO） （可用性取 90 秒超时率的补集，因此超时率为 $\leq$ 5%）

30 秒）。在中位数与超时门控下，只有 7.8% [5.2、10.9] 的 GPU 时间可回收（表 8 ）。当 C = 240 秒时，在恢复开始后的前 150 秒内到达的请求需要等待 90 秒以上并最终超时，因此 0.5% 条款限制了这段时间内到达的请求所占比例。在标准服务等级目标（5% 超时、可用性 $\geq$ 95%）下，可回收容量为 16.4%。对于大多数有流量的服务，传统重启不符合服务等级目标；而符合目标的服务可回收的 GPU 时间也很少。

快速恢复会改变边界，而改变幅度取决于服务等级目标的严格程度。 表 8 给出了各门控机制下可重新获取的份额。在中位数与超时门控下，C = 5 s 和 C = 60 s 较为接近：分别为 47.7% 和 42.5%，配对差异为 5.2 个百分点 [3.8、6.8]。在面向体验的门控机制下，差距会扩大。平均 TTFT 门限下分别为 46.5% 和 18.1%（配对差异为 28.4 个百分点 [22.7、33.5]）。P90 下分别为 34.0% 和 24.8%（9.2 [5.3、13.9]），P99 下分别为 14.9% 和 11.2%（3.6 [2.0、5.6]）。粗粒度中位数与超时门限能够容忍一分钟的恢复时间，而平均 TTFT 门限需要数秒，尾延迟门槛则会损失约四分之一的容量。最严格的尾延迟门槛在任何 C 下都几乎没有可重新获取的空间：在 P99 下，即使 C = 5 s，也只有约 15 % 的 GPU 时间可以重新获取。

用户可见的成本集中在少数请求上。在 θ = 120 s 和 C = 5 s 时，0.10% 的请求会发生延迟，平均首令牌时间增加 4 ms，因此集群级成本可以忽略不计。成本分布并不均匀：对于 10–20% 的服务，大多数请求都会发生延迟（从 θ 120 s 降至 30 s）。图 4 按服务展示了这一情况，即缩容至零所付出的代价如何随唤醒时间变化。我们使用引擎报告的首令牌时间作为基线，因为 99.8% 的网关请求都是非流式请求，而且 2 s 的界限适用于输入低于 4K 令牌的情况。

采用休眠/唤醒（C = 5 s，θ = 60s）时，78% 的服务将中位首令牌时间控制在 2 s 以内；始终运行时这一比例为 90%。87% 的服务将其控制在 4 s 以内，而始终运行时这一比例为 95%。在 P90 情况下，54% 的服务将其控制在 2 s 以内。冷启动（C = 240 s）时，56% 的服务在中位数情况下将其控制在 2 s 以内，另有三分之一超过 90 s 的超时限值。

从总量到净值。 表格 9 展示了从总停放份额到净可回收容量的漏斗过程。门控机制带来的成本远高于开销。应用中位数与超时

表 8。 常规周内每服务最优 θ 的 GPU 时间可回收份额（%），以及跨服务计算的 95% 自助法区间。C = 60 s 为假定值，并非测量值。正文中的配对差异按服务计算，因此未必等于四舍五入后的份额之差。

门控机制 ↑	C = 5 秒	C = 60 秒	C = 240 秒
平均 ≤ 2 秒	46.5 [41.5，51.4]	18.1 [14.5，22.1]	8.9 [6.2，12.2]
P50	47.7 [42.6，52.7]	42.5 [38.1, 46.9]	34.4 [30.0, 39.0]
P90	34.0 [29.9，38.3]	24.8 [20.3, 29.8]	19.4 [15.4、23.8]
P99	14.9 [11.3、19.2]	11.2 [8.4、14.6]	8.3 [5.5, 11.3]
超时（90 秒）	52.6 [47.6, 57.6]	49.9 [45.2，55.2]	7.8 [5.1，10.9]
中位数和超时，更严格的 SLO	47.7 [43.1，52.8]	42.5 [37.5，47.0]	7.8 [5.2，10.9]
中位数和超时，标准服务等级目标	47.7 [42.8，52.3]	42.5 [37.7，47.6]	16.4 [13.0，20.4]

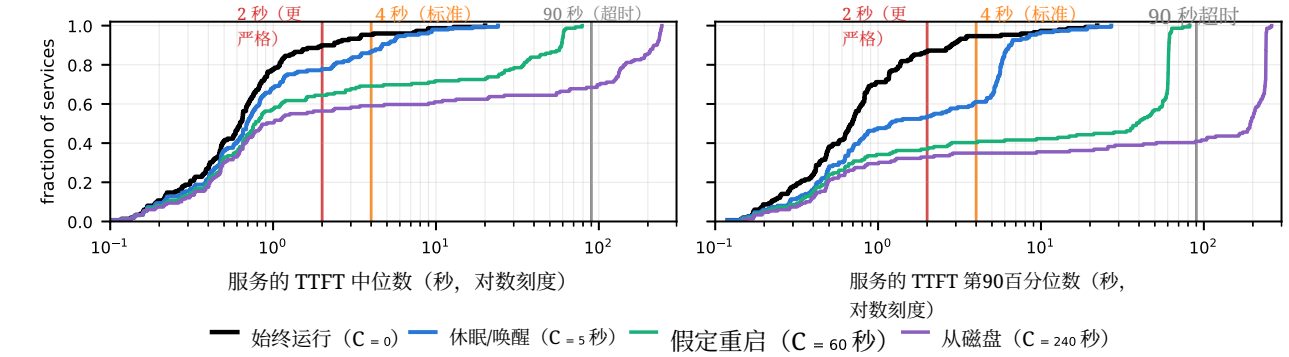

图 4. 四种恢复延迟下各服务的中位数和 TTFT 第90百分位数，以及与服务等级目标线的对比（常规周，θ = 60 秒）。该图涵盖了 212 项服务中有 TTFT 数据且始终在线中位数至少为 100 毫秒的 149 项服务。

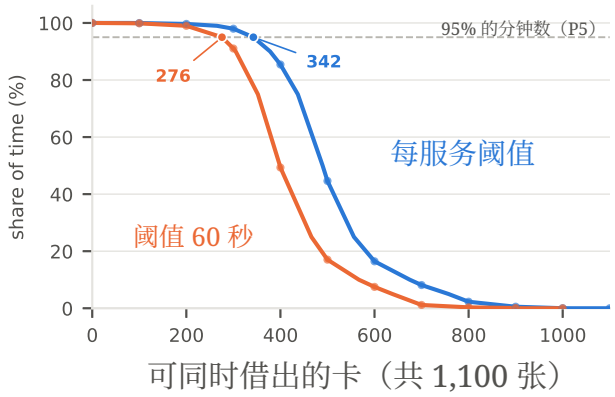

图5. 常规周中，每分钟至少有 X 张（共 1,100 张）卡可供出借的时间占比。

门控后，按服务设置阈值时可保留未门控容量的 91%，使用 θ fixedat60s 时则可保留 81%。扣除借用方负载和让出时间后，剩余容量仅减少 0.6–3.4%。

容量也保持稳定。图5统计同一分钟内可供出借的卡数。每服务最优阈值下，平均有 1,100 张卡中的 507 张可供出借，第5百分位数为 342 张（31%）。θ = 60 s 的平均值为 417 张，第5百分位数为 276 张。几乎所有时间都有约 300 张卡可供出借。

本实验的局限。阈值是在样本内选择的。重放会在 θ 秒没有新请求到达后暂停服务，而系统会等待进行中的请求完成，因此，对于生成过程较长的服务，重放会高估空闲时间。借用单位是整个服务，因此结果忽略了节点、拓扑和碎片化限制，也无法判断某项服务的 Pod 是否会同时空闲。借用方的让出时间只计入借用方一侧，我们没有用 C 加上让出时间重新运行所有者的门控机制。第7.3节测得的释放步骤中位数，在暂停运行中为 2.52 秒，在重新入队运行中为 1.56 秒，与此处假设的 3 秒一致。C = 60秒是一个假设。我们只观察网关可见流量。

7.2 机制成本

休眠与唤醒。表10报告了休眠（释放）和唤醒延迟。对于在 H20 上以 8 路张量并行运行的 GLM-5.2，稳态释放需要 0.9–1.1 秒，唤醒需要 2.9–4.4 秒，而冷启动需要 244–277 秒。320亿参数模型的唤醒需要 0.4–0.7 秒。引擎启动后的首次释放需要 80–108 秒，因为权重的主机内存备份只需构建一次；我们单独报告该数据，并将其排除在稳态数据之外。

在对 GLM-5.2（张量并行度为 8）进行的一项检查中，使用 100 道 GSM8K 题目（半数采用贪心解码，半数以温度 0.6 进行采样），每道题都在完全唤醒后运行，随后进入休眠，共进行 100 个周期。准确率为 95%，而不进行休眠的两次运行中准确率分别为 97% 和

表格 9. 从毛可回收容量到净可回收容量（常规周，1,100 张卡，C = 5s，借用方负载 5 s，产出 3 s，最小有效片段 10 s）。

阈值规则	未门控毛值	中位数与超时门控	Net	净 GPU（当量）
每服务最优 θ	52.6%	47.7%	46.1%	507
固定 θ = 60秒	47.8%	38.8%	37.9%	417
固定 θ = 120秒	42.9%	36.3%	35.9%	395
固定 θ = 300秒	37.1%	33.6%	33.4%	368
每服务最优 θ，C = 60 s	—	42.5%	41.4%	455

表10. H20 上的切换延迟（稳态，秒）。

模型（并行度）	释放 ↓	唤醒 ↓	冷启动
GLM-5.2（TP8）	0.9–1.1	2.9–4.4	244–277
32B级（TP1–4）	—	0.4–0.7	—

98%；此外，贪心解码部分在每次运行中均为 49/50，因此差异处于运行间噪声范围内。批次组成和缓存状态会改变数值结果，因此在该引擎上，贪心解码的令牌路径在不同运行之间并非位级可复现，我们改为比较准确率。恢复后的令牌一致性将在下文进行测试。解码吞吐量分别为 71.6 和 71.7 令牌/秒。休眠确实会丢弃前缀缓存，因此每道题在唤醒后都需要完整地重新预填充（对于 800 令牌的提示，耗时约 0.25 秒）。对于长提示词而言，这一成本不容忽视，而暂停能够保留缓存，从而避免该成本。

暂停和恢复会保留进行中的请求。休眠必须先排空繁忙引擎，而像重新入队那样在未排空的情况下释放引擎，会中止其中正在运行的请求。暂停则会保留每个请求截至最后一个完整页面边界的已提交前缀，将该键值缓存写入主机内存，并释放 GPU 上的权重和键值缓存。恢复时会还原这些内容，重建最后一个不完整页面，并恢复采样状态。我们在 GLM-5.2（W4AFP8、TP8、8 × 个 H20-141 GB）上进行了测量，使用了长度为 4K–128K 个词元的提示，以及 1–8 个并发请求。恢复在每个单元格中都重新计算了零个词元（重新计算的词元数量为 = 0）。在此微基准测试中，对于单个请求，采用贪心解码时，恢复后的输出与不中断运行的输出完全相同。

我们在两个模型上检查了暂停的正确性。在 GLM-5.2（TP8，8 × H20）上，对 100 道 GSM8K 题目（50 道贪心解码，50 道以温度 0.6 采样）分别在生成第 30、60 或 100 个词元时暂停，并在完成一次完整的暂停与恢复循环后继续运行。在 100 个答案中，有 99 个足够长，可以执行暂停。准确率为 98%，而相同题目在不暂停时为 97%；其中贪心解码部分的得分为 50/50，而不暂停时为 49/50，处于运行间噪声范围内。解码吞吐量为每秒 71.6 个词元，而休眠检查的基线为每秒 71.7 个词元。整个前缀缓存均得以保留：恢复后缓存的提示词元中位数约为 779 个（总数约 800 个），而普通的休眠与唤醒会丢失这些缓存。恢复的请求只需重建最后部分页面，中位数为 33 个词元，最多为 62 个，少于一个 64 词元页面（99 次恢复共计 3,407 个词元）。流在暂停与恢复传输期间冻结（中位数 6.9 秒，最多 9.0 秒），

表 11。暂停成本及其避免的成本（GLM-5.2、W4AFP8、TP8）。暂停 + 恢复的实际耗时为暂停调用加恢复调用的实际耗时之和，其中包括推进到页面边界的耗时；4K ×1 和 32K × 8 单元使用重复协议（n = 5），其他单元使用单请求协议（n = 2）。

设置	暂停 +恢复 实际耗时 ↓	KV 重新加载 从主机	重新计算预填充 而不是 ↓
4K × 1	5.05 秒（4.77–5.52）	12 毫秒	0.48 秒
32K × 1	5.7 秒	107 毫秒	6.1 秒
32K × 8	7.25 秒（6.66–7.75）	878 毫秒	平均 29.4 秒（最长可达 50 秒）
128K × 1	6.0 秒	445 毫秒	61.7 秒

随后继续运行，没有丢失或重复任何词元。第二次检查使用了一个更小的模型和不同的配置（Qwen3-30B-A3B、TP= 2、页面大小为 64、确定性推理），在 100 道 GSM8K 题目上进行测试（50 道贪心解码，50 个温度为 0.6、前 k 项为 20 的设定随机种子样本）。全部 100 个恢复序列都与未中断运行完全一致；在恢复点之后，采样对数概率和前 k 项条目也逐位精确一致。无论是否暂停，准确率均为 90%；其中 5 个请求连续恢复了三次，也都完全一致。

表 11 显示，对于一个 4K 请求，暂停加恢复的实际耗时为 5.05 秒（4.77–5.52，n = 5）；对于八个 32K 请求，则为 7.25 秒（6.66–7.75，n = 5）。温度采样的成本相同（5.03 秒和 7.48 秒）。从主机重新加载键值缓存，每个词元约需 3 μ 秒（一个 4K 请求需 12 毫秒，八个 32K 请求需 878 毫秒，一个 128K 请求需 445 毫秒）。相比之下，从头重新计算上下文所需时间随上下文长度增长：4K、32K 和 128K 个词元分别需 0.48 秒、6.1 秒和 61.7 秒。无论上下文长度如何，该引擎冷启动都需要四到五分钟。

表格解读。暂停耗时数秒，主要用于传输权重，休眠与唤醒也需要这部分时间。暂停省去的是上下文相关项，而在 128K 个词元时，这部分耗时超过一分钟。当替代方案是重新计算时，这一点至关重要。

抢占借用方的代价取决于部署的两个属性。所有者返回时，可以通过三种方式处理借用方正在进行的请求。重新入队 会中止请求，并从头重新运行。

重计算续接 会重新提交提示以及迄今生成的词元。暂停则会保留请求。这些方式的成本取决于借用方的前缀缓存是否会在释放后保留，以及请求是否以流式方式输出结果。在我们的部署中，引擎会在释放后保留前缀缓存，因此重新入队的请求可以命中已缓存的提示词前缀（根据引擎日志，提示的 99.9% 已缓存，每次重放约需新增 64 个词元），只会损失解码进度。如果不保留缓存，每次重新入队都要重新计算提示。在一小时的重新入队测试中，106 个中止的请求各自重放了一个约含 65K 个词元的缓存提示。若不保留缓存，我们估计这相当于约 6.9M 个预填充词元，需要 2,000 秒以上的计算时间。此外，只有在尚未交付任何输出时，重新入队才有效。流式请求已经输出了部分结果，重启会导致这些输出重复。暂停同时具备上述两项优势，因为它不依赖缓存，并且会保留请求。

我们使用解析模型量化抢占造成的计算损失。模型抽取 6,000 个真实借用窗口（常规周；所有场景和并发度使用相同的窗口；抽取三次），并运行一个包含 B 个并发请求的借用方实例。预填充按照我们测得的吞吐量串行执行（在 4K、32K 和 128K 下分别为 9.4K、5.6K 和 2.3K 令牌/秒），并根据各策略丢弃的工作量计入相应损失。我们采用第7.3节端到端工作负载中的请求长度（输入约 64K，输出 4K）。表12报告了在我们采用的两个空闲阈值下，可用 GPU 时间的损失占比。保留缓存（与实际部署相同）时，重计算续接和暂停的损失分别为 1–6% 和 1–6%，重新入队的损失为 8–18%。暂停相比重计算续接最多仅有 4 个百分点的优势。不保留缓存时，重新入队和重计算续接的损失为 14–35%，超过暂停损失的十倍。不同抽取结果的平均窗口长度相差约 ±20%，使重新入队的损失浮动几个百分点（表中各场景的标准差为 2–7 个百分点）。相差一两个百分点属于这一噪声范围。该模型是解析模型：请求长度是预先假定的，借用方的解码速率则取自对另一个模型的测量。

7.3 端到端运行

7.3.1 设置

一个节点配备八张 H20 GPU 和 2 TB 主机内存，两个租户共用这八张卡。运行期间，借用方的权重备份和主机 KV 池，以及在线引擎的备份，共占用约 1.5 TB 主机内存。在线租户代表一种生产环境中的搜索代理服务（Qwen3.5-35B-A3B、FP8、张量并行度为 4，使用第 4–7 张卡）；我们通过网关路径重放其包含 336 个请求、时长为 3.94 小时的到达序列。提示内容为合成数据，但复现了跟踪记录中的请求长度和前缀结构。前缀缓存不会被强制设为某个目标命中率，而是在引擎运行期间逐步重建。借用方使用 GLM-5.2（W4AFP8，张量并行度为 8），每当在线引擎处于停放状态时，便占用全部八张卡；其任务来自持久队列，队列中有八个并发请求，每个请求包含 64K 输入令牌和 4K 输出令牌（固定随机种子）。在线请求到达后若 30 秒内没有新的在线请求（θ = 30 s），在线引擎便会停放，其 GPU 卡借给借用方。我们比较在线请求到达时处理借用方进行中请求的两种方式。**重新入队**会中止这些请求，并将其放回队列。这是一种中止并重新排队的基线方案，其思路与 ROSE 类似：ROSE 会中止被服务突发打断的推演请求，并将其重新路由 [16]；但这并非 ROSE 的重新实现。ROSE 会将两个模型共同部署，并把中止的推演任务重新路由到其他 GPU。**暂停**（暂停运行）会将这些请求的键值缓存保存到主机内存，并在中断处恢复运行（§5.2、§5.4）。两次运行使用相同的二进制文件（所有组件均为同一构建版本）、相同的在线跟踪记录、相同的借用方工作负载和相同的随机种子。二者仅在一个调度器开关上有所不同。已部署的重新入队路径会在释放后保留前缀缓存（第 7.2 节）。每种策略均运行一次，持续四小时；先运行重新入队方案，再运行暂停方案，两者启动时引擎均为全新启动且缓存为冷缓存。

7.3.2 结果

暂停能在相同的节点级 GPU 时间内完成更多借用方工作（表 13）。每个保持的样本中，暂停运行产生的输出令牌比重新入队多 25%（总计 1.32×，因为它保持 GPU 的时间也稍长一些）。暂停运行中的采样器运行频率约为 0.83 Hz，重新入队运行中约为 0.81 Hz，因此样本并不代表秒数。按各样本的时间间隔加权后，得到 11,733 秒和 11,460 秒的节点级保持时间，比率为 1.29。其机制正如 §5 所述：重新入队中止了 310 次请求尝试并丢弃其解码进度，而暂停则恢复了相同的请求，只重建其最后部分页面。在线方面，暂停运行中的所有请求均成功，且两次运行的中位延迟相同（176 毫秒和 170 毫秒）。

弹性层的语义保持不变。暂停运行的 375 个借用方请求和重新排队运行的 285 个请求全部完成。运行结束时，没有请求故障，也没有请求仍在执行中。重新排队运行中止的 310 次尝试均重新运行至完成。在这次持续四小时的暂停运行中，没有任何暂停等待恢复超过 224 秒，因此 15 分钟后备机制从未触发。

在线尾延迟是交接的代价。暂停运行中，在线 TTFT 的第 99 百分位数为 10.1 秒，重新排队运行中则为 11.2 秒。在两种运行中，约 12% 的请求首令牌延迟超过 2 秒。卡被借出后到达的请求，需要等待借用方离开、在线引擎唤醒，以及自身预填充完成。对于这条跟踪记录，暂停运行符合第7.1节的中位数与超时门控。重新排队运行中的 HTTP 504 错误（占请求的 1.2%）将无法满足更严格的 SLO 中 0.5% 的超时条款，而

表12。 因抢占而损失的可用 GPU 时间占比（%，三次时间窗口抽样的均值；解析模型；64K 输入/4K 输出请求；越低越好）。

前缀缓存	θ（秒）	并发度 B	重新入队	重计算续接	暂停
保留（已部署）	30	8	12.5	2.5	1.5
保留（已部署）	30	32	18.4	6.0	2.0
保留（已部署）	60	8	8.2	1.5	0.9
保留（已部署）	60	32	12.5	3.7	1.2
未保留	30	8	23.0	19.7	1.5
未保留	30	32	34.6	33.0	2.0
未保留	60	8	16.3	13.8	0.9
未保留	60	32	26.7	25.4	1.2

表 13。 四小时受控比 较（每种策略运行一次；在线租户：336 个请求；借用方：64K 输入/4K 输出请求，8 个并发 t).

	暂停	重新入队	比率
借用方输出令牌↑	1,528,288（375 个请求）	1,159,479（285 个请求）	1.32
借用方保持的节点级 GPU 样本（名义上每秒一个）	9,694	9,193	1.05
每个保持样本的借用方令牌 ↑	157.7	126.1	**1.25**
借用方的抢占次数	41	40	—
八个 GPU 全部空闲的样本↓	0	198	—
已中止的请求尝试（已重新入队）↓	0	310	—
池中 GPU 样本忙碌（nvidia-smi 利用率 >0）↑	81.6%	78.1%	1.04
在线首令牌时间，中位数	176 毫秒	170 毫秒	≈ 1
在线 TTFT，第 99 百分位	10.1 秒	11.2 秒	0.90
在线 TTFT ，最大值	14.1 秒	82.7 秒	—
引擎故障（第7.3节）↓	0	3	—
在线请求故障 ↓	0%	1.2%（HTTP 504）	—

注。 输出令牌指所有请求实际交付的输出；两次运行中的请求均全部完成（重新入队运行中中止的 310 次尝试后来都重新运行至完成）。重新入队运行损失了三个引擎，其中两个是在重新入队特有的负载下发生的，另一个则发生在两种策略都会经过的路径上；其四次超时和 82.7 秒的首令牌时间最大值，均出现在随后发生的重建窗口内。

两种运行中的 P90 和 P99 门控均无法通过。

重新入队运行期间损失了三个引擎：一波重新入队的请求到达仍在恢复中的引擎时，发生了两次唤醒侧崩溃；此外，HiCache 降级导致设备 KV 未能在两种策略共用的路径上释放，从而发生了一次释放侧故障（每次暂停也会将缓存写入主机层，§5.2）。重新入队运行中八个 GPU 全部空闲的 198 个样本（表 13）、四次 HTTP 504 超时，以及 82.7 秒的在线首令牌时间最大值，均发生在这些故障之后的重建窗口内；这直接促使我们设计了 §6.2 中的唤醒侧准入门控和释放侧剩余内存检查。我们不把由此产生的空闲时间计作暂停带来的收益，因此以两种方式报告比率。在两次运行中都排除没有内核运行的 GPU 持有样本（重新入队运行占 2.6%，暂停运行占 0.2%）后，比率为 1.22，而非 1.25；我们将 1.22 视为更保守的估计。

比较的局限性。 每种策略仅运行了一次，因此未测量运行间变异；此前在修复前构建版本上进行的一组四小时运行（因部署缺陷而弃用）以及一组一小时运行也指向相同结论。该工作负载（64K 输入、4K 输出）相对于借用时间窗口而言较长，因此有利于能够保留进度的机制。比较不包括不保留前缀缓存或不进行重计算续接的重新入队，这是一种作业层操作（§5.4）。当借用方的 GPU 内存超过某个阈值时，一个样本即计为保持。

样本名义上每秒采集一次（重新排队运行中为 0.81 Hz，暂停运行中为 0.83 Hz），由于两次运行采用相同的方法，因此每个样本的比率是有效的；上文给出了按时间加权的比率。在线租户是某一服务的单个实例。繁忙 GPU 样本是指 nvidia-smi 报告某个 GPU 具有非零利用率的样本。它们显示的是某个内核是否在该时刻附近运行，而不是 GPU 吞吐量的使用量。两次运行中的样本来自相同的采样过程（约 4 小时、频率为 0.81–0.83 Hz、使用 8 个 GPU），不能与第 7.3.3 节基于日志的份额进行比较。

7.3.3 池时间

我们根据引擎日志，对受控四小时暂停运行（14,184 秒）中的每一秒进行了分类。借用方占用 GPU 共 11,486 秒。其中，7,156 秒记录为计算，3,642 秒是在连续步骤执行期间相邻预填充/解码日志行之间不超过 3 秒的短暂间隔（两者合计占保持时间的 94%）；释放步骤耗时 104 秒，唤醒后的预热耗时 6 秒，等待耗时 578 秒。在线引擎占用 GPU 共 2,484 秒，其中 1,917 秒为等待，296 秒记录为计算，17 秒为衔接间隔，209 秒为健康探测预填充，45 秒为释放步骤。按各引擎使用的卡数加权（借用方使用八张，在线

引擎使用四张），该资源池在运行期间有 51.5% 的时间在进行计算，而仅在线服务时这一比例为 1.0%（计入健康探针后分别为 52.2% 和 1.8%）；这些比例只统计日志记录的计算，不包括衔接间隔。根据日志，运行期间有 214 秒处于交接边界，此时两个引擎都未占用 GPU；而表 13 的内存采样器没有采集到八张 GPU 全部空闲的样本；两者使用的信号不同。参照情形是在线引擎独自在其使用一半资源的节点上运行，这正是我们对所观察到的生产服务建模的方式；它不是静态划分的资源池。这些比例依据日志行和以约 0.83 Hz 采样的 GPU 内存得出，因此分辨率为数秒；第 7.3.2 节中采样器记录的占用时间（11,733 秒）与基于日志的数据相差约 2%。一小时的结构以及与解析模型的比较见附录（第 11 节）。

8 讨论与未来工作

我们的发现。 长尾在线服务约有一半 GPU 时间处于空闲状态，且大多出现在持续一小时或更久的时间窗口中；秒级唤醒时间使这些资源能够出借，同时满足中位数与超时服务等级目标。GLM-5.2 在 TP8 下约 1 秒进入休眠，约 4 秒即可唤醒。对于包含 64K 令牌请求的借用方，暂停后每个保持占用的 GPU 样本所产出的令牌比重新入队多 22%。我们评估了在线层和弹性层；作业层采用解析方式定价（表12），且第7.3节未测试重计算续接。

暂停何时发挥作用。 当被抢占的请求已有大量解码进度可能丢失时，暂停更有优势：长上下文请求会使重新计算代价高昂，而借用方若在较短时间窗口内频繁被抢占，就会把时间耗费在重新构建进度上。该机制仅适用于弹性层，因为其客户端可以等待停顿结束（§4.2）。重计算续接是作业层的操作，遵循 RLBoost 的做法：通过重新提交提示以及迄今生成的令牌，在令牌级迁移部分响应 [71]。ModelLess 仅对高于 min 的引擎应用该机制。重新入队、续接和暂停在抢占后保留请求层状态的程度各不相同。保留前缀缓存时，续接能够弥合与暂停之间的大部分差距；解析模型估计，暂停剩余的优势约为四个百分点（表12）。如何选择取决于产品形态：客户端能够容忍什么，以及引擎支持什么。暂停需要精确保留状态，且不依赖缓存。缓存能够保留且请求较短时，重新入队更合适。我们的长工作负载（64K 输入、4K 输出）更适合保留进度，因此测得的 1.22–1.25 比率（保守值为 1.22）就是在该情形下的收益。

启动经验。构建并运行单节点原型暴露了五个反复出现的陷阱，这些陷阱塑造了 §6.2 中的安全不变量。首先，在普通休眠期间调用引擎的生成暂停会冻结其调度器，并导致排空死锁（§5.1）。其次，在释放前将已提交前缀写入主机缓存，会让重新入队变成一种非常……

强得多的基线，而不是会丢弃基数缓存的社区休眠（§5.4）。第三，在释放或传输期间触及 GPU 内存的存活探针会导致引擎崩溃，因此在内存移动期间，探测必须降级为进程存活性检查（§6.2）。第四，重新排队运行中的三次引擎故障表明，必须保持准入关闭，直到唤醒完成；释放还必须验证降级后不再残留设备 KV（§6.2、§7.3.2）。第五，唤醒空闲引擎的健康检查会阻碍约 3,000 张无流量的卡的回收（§2.2），而一个处于唤醒状态的四卡在线服务，在小于 θ 的请求间隔期间仍会阻塞全部八张卡（§7.3.3、§11.1），因此促成了打包方案。

局限性。 每项实验都说明了自身的局限性（§2.4、§7.1、§7.2、§7.3.2）。其中有四点适用于所有实验。所有测量均来自同一加速器代际（H20）；端到端结果则来自单个节点、单条在线轨迹和一种借用方工作负载，且每种策略仅运行一次。容量数据是上界，而所有者侧门控未计入借用方的产出，因此，对请求延迟敏感的门控结果偏乐观。主机内存限制了单个节点可容纳的所有者和借用方数量；在我们的实验中，2 TB 节点约有 1.5 TB 主机内存。我们未进行故障注入、多 Pod 并发或多节点实验，且尚未测量 §6.4 中的控制平面故障切换在调度器长时间中断时的表现。

未来工作。 需要在训练器参与的情况下评估跟随/封存；在集群规模下，它还可以扩展超额分配，因为新引擎无需读取存储即可从正在运行的引擎启动。空闲阈值和卡的放置也应适应到达模式。对于智能体框架，一种能够容忍网络中断的流式弹性模式将扩大可接受停顿的客户端范围。一旦键值缓存可以在引擎之间移动，无论是通过 Mooncake 或 LMCache 等池化层，还是通过点对点传输，被抢占的请求就可以在另一个引擎上恢复，而不是重新计算。将实例打包到卡组中，可以将借用单位从整个服务细化为卡组。强制默认引擎在其请求排空之前让出卡，需要能够容忍迁移的同步客户端。在引擎之间共享卡组受到硬件限制，因为 GPU 目前还无法像 CPU 在 cgroups 下那样进行细粒度分区，因此容量按整个卡组进行规划。暂停也适用于扩散 Transformer 等多步生成模型。在去噪步骤边界，请求必须保留的状态紧凑且定义明确：潜变量、步骤索引、求解器历史和采样器随机状态；在序列并行或图块并行下[31]，每个秩都必须一同检查点保存其分片。扩散服务系统已经在单个引擎内的步骤边界处抢占并重新调度请求 [37, 77], ，而近似缓存则会跨请求复用中间状态 [2]。在步骤边界暂停，然后将整个 GPU（包括权重）释放给另一个模型，可以让扩散请求进入弹性层。复用跨步骤激活值的特征缓存

跨步骤 [33, 38, 39, 84] 将发挥类似于大语言模型前缀缓存的作用：丢弃它们是安全的，但恢复后需要额外执行完整步骤，因此是否保留它们与是否保留键值缓存一样，都是成本权衡。最后，应使用每项服务的时间窗口对解析模型进行校准，以便部署能够在运行前预测是否值得启用暂停。

9 相关工作

9.1 集群管理器与可抢占资源

Borg 在共享机器上运行生产和批处理工作负载，并通过抢占低优先级任务来释放资源 [64], ; Omega 和 Kubernetes 沿用了这一做法，它们均源自 Borg [6, 53], ; 而 Mesos 则在各框架之间共享资源 [20]。Harvest虚拟机根据明确的服务等级目标条款，向机会型工作负载开放未分配的容量，并以驱逐作为代价 [4]; Heracles 通过隔离共享资源，将尽力而为型工作负载与延迟关键型服务共同部署[35]; Resource Central 和 Autopilot 则会预测并调整工作负载规模 [7, 52]。除了 Harvest虚拟机 在缩减和扩容时所做的资源调整外，这些系统都将任务视为一个整体单位，要么运行，要么被驱逐。大语言模型实例则不同：驱逐成本取决于其权重、执行状态、KV 缓存和进行中的请求中哪些得以保留（§4.1）。我们对这一研究方向的贡献，是将这种选择作为分层驱逐语义呈现，而非提出一种新的优先级方案。

9.2 深度学习的GPU调度器

Gandiva、AntMan 和 Singularity 以时间片方式运行，进行弹性扩展，或透明地抢占并迁移训练作业 [57, 74, 75]。Lyra 与我们的理念最为接近：它将空闲的推理集群GPU借给弹性训练作业，并在需要时回收这些 GPU [30]。我们以模型实例为粒度进行借用，支持秒级抢占和按层级划分的请求语义，而我们的借用方包括批处理推理和展开推理。对生产GPU集群的工作负载研究刻画了利用率和排队情况 [70] , 以及大语言模型开发工作负载中的故障与调度问题 [22]。训练作业通常会在迭代边界处创建检查点，并能容忍数分钟的中断；Singularity 的透明性则作用于作业级别。在线推理实例必须在数秒内恢复，并且可能持有无法重新运行的请求。我们的轨迹研究（§7.1）衡量了这一约束如何限制在线服务长尾部分的可回收容量。

在单个 GPU 内，有若干系统允许对延迟敏感型工作负载与尽力而为型工作负载共享设备。REEF、GPreempt 和 XSched 支持在内核或队列粒度上抢占正在运行的 GPU 工作负载，因此依赖于对启动路径或运行时的修改 [12, 19, 55]。Salus、Paella、Orion 和 Tally 共享一个 GPU，通过细粒度切换、内核级调度和干扰控制来共享作业或请求 [41, 59, 80,83]。这些机制会在共享的 GPU 上以内核或命令队列粒度交错执行或抢占工作负载。我们一次将整个卡组交给一个实例，且不修改内核或 GPU 运行时（§5）。

9.3 大语言模型服务系统与控制平面

服务引擎和调度器针对单个部署进行优化：连续批处理、分页注意力、预填充与解码调度和解耦，以及 KV 缓存池化[3, 28, 45, 49, 78, 85, 86]。Llumnix 会在实例之间迁移正在运行的请求及其 KV 缓存，以实现负载均衡 [60]；而我们的请求在抢占后会重新计算上下文，而不是迁移上下文。AlpaServe、MuxServe 和 QLM 通过多路复用模型或管理队列来满足服务等级目标 [11, 32, 46]; DynamoLLM 会调整集群配置以优化能耗，SpotServe 则在可抢占实例上提供服务[40, 58]。Kubernetes 原生控制平面（如 AIBrix 和 llm-d）以及通用排队和自动伸缩组件（Kueue、Volcano、KubeRay、KEDA、Knative），负责管理和扩展单一产品的部署（Kueue 和 Volcano 负责排队，KubeRay、KEDA 和 Knative 负责自动伸缩） [25–27, 34, 50, 61, 66]。ModelLess 增加了这些控制平面无法提供的一个步骤：将副本的 GPU 交给较低层级使用，并在之后取回，同时明确定义这期间如何处理任务。

9.4 多模型池化、冷启动与保持活动

ModelLess 属于无服务器大语言模型系统这一类，这类系统通过在模型间共享 GPU 来缩短冷启动时间。ModelLess 保留缩容至零的能力，并在分层驱逐语义下，将释放出的 GPU 交给较低层级。Prism 通过在多个模型间动态扩展内存，以应对需求变化，从而让许多模型（包括低流量模型）共享 GPU 内存 [81]。Aegaeon 通过令牌级切换，为并发运行的长尾模型池化 GPU [72]。我们改为移交整个卡组，因为大型张量并行模型没有空间容纳第二个常驻模型（§2.6）。PipeSwitch 和 Clockwork 是此前最接近的休眠/唤醒机制：两者都会在主机内存与 GPU 之间迁移模型权重; PipeSwitch 采用流水线式上下文切换，还能让一块 GPU 在训练与推理之间交替使用；Clockwork 则提供可预测的加载和执行时间 [5, 17]。它们面向较小的模型，且都不会携带执行中请求的状态。因此，我们新增的是生产级大语言模型引擎路径、层级语义，以及围绕该路径构建的暂停与抢占顺序。ConServe 在同一引擎中共同服务离线请求和在线流量，按层抢占离线请求，并从增量检查点保存的 KV 缓存中恢复它们 [48]。ServerlessLLM

通过快速检查点格式缩短模型加载时间；BlitzScale 通过在实例间多播权重实现在线扩缩容，λScale 增加了边执行边加载的扩缩容能力，而 WarmServe 会预热实例 [14, 36, 79, 82]。我们的空闲阈值策略是一种保活策略；无服务器函数领域对此已有深入研究，研究表明，固定时间窗口不如自适应时间窗口 [15, 29, 51, 54]。快速启动和快照相关工作缩短了恢复本身所需的时间 [10, 44, 63]；Dirigent 则降低了编排延迟 [9]。我们采用固定阈值：θ 对所有服务统一使用一个阈值，或 θ 针对每项服务采用事后选出的最佳阈值，并将自适应策略留作未来工作。引擎本身及内存节省工具已支持快速释放和恢复引擎内存 [62, 65]。这类工作的区别在于释放出的 GPU 用途：我们研究将其交给较低层级，同时满足所有者侧的服务等级目标，并衡量计入恢复延迟和服务等级目标严格度后剩余的容量。

9.5 共享服务 GPU 与强化学习展开

强化学习后训练框架会跨引擎调度并交叠展开、奖励和训练阶段 [13, 21, 56, 67]。ROSE 将展开放在处理在线流量的 GPU 上，在控制器下共置二者；该控制器负责保护服务级别目标，在流量突发时中止受影响的展开并重新路由，同时报告端到端结果 [16]。RLBoost 则利用可抢占 GPU，包括竞价实例和生产环境中的闲置容量，用于展开，并在发生抢占时迁移部分响应 [71]。我们的设置在共享单元上有所不同：共享单元是卡组上的一个实例，且每次恰好只有一个模型驻留；而 ROSE 会让两个模型同时驻留，并在以服务优先为准入原则的情况下，在空间维度共享内存和计算资源。让第二个模型的权重驻留会占用设备内存，而在线引擎原本可以将这些内存用于 KV 缓存，因此，大模型借用方会大幅缩减在线引擎能够容纳的上下文；交接卡组则会付出数秒级唤醒延迟的代价。我们还将批处理和异步推理视为借用方，而不仅仅是展开。我们不与这些系统比较端到端结果。

9.6 内存管理与检查点/恢复

动态 GPU 内存管理（包括用于服务的 vAttention 和用于训练的 GMLake）通过将虚拟内存与物理内存解耦来减少碎片 [18, 47]。操作系统级 GPU 检查点与恢复（例如 PhoenixOS、CUDA 的检查点工具以及 CRIU 的 GPU 支持）会对整个进程进行快照 [8, 43, 69]。请求级的键值缓存交换和重计算抢占存在于引擎内部 [28]以及协同服务系统中 [48], ；InferCept 会分析如何丢弃、保留或交换被中断请求的键值缓存状态[1]；表12中的成本比较属于同一类型。我们的暂停操作有所不同：它会释放整个 GPU，包括权重，在交接给另一个模型的过程中。它也比进程快照更受限，并且能够感知引擎。它只写入每个请求已提交的、页对齐前缀，并恢复采样状态。除了承担休眠也需要支付的权重交换成本外，其成本还会随着已提交词元的数量增长（每个词元约 3 μ 秒），而不是随着整个进程映像增长（第 7.2 节）。

9.7 工作负载表征

公开和生产环境中的大语言模型工作负载研究描述了突发性、请求长度分布和缓存行为[42, 68, 73]。SageServe 对云服务提供商的生产环境大语言模型工作负载进行表征，以驱动预测感知自动扩缩容[23]。我们提供了一个网关背后在线服务长尾的周级视图，其中到达序列的分辨率为一秒，并以聚合形式呈现（§2）。

10 结论

多层推理平台中的空闲 GPU 容量很大，集中在较长的时间窗口内；只有在交接耗时几秒的情况下，才能在严格的延迟门限下利用这些容量。粗略的中位数和超时门控机制可容忍约一分钟，但若冷启动需要数分钟，可利用的容量就只剩 8%。ModelLess 是针对这一场景的无服务器系统：用户声明模型服务，空闲服务可在数秒内缩容至零，释放出的 GPU 则分配给低层级。它将模型实例作为调度和抢占的单位，并为每个层级提供针对实例状态的驱逐语义。四个组件（modelentry、modeldispatcher、modellet 和引擎插件）通过抢占协议来落实这些语义：在释放内存前关闭准入，并在出现故障时采取关闭策略。系统使用休眠/唤醒、暂停/恢复和跟随/封存等操作。在一个节点上，根据我们的跟踪记录和工作负载，与中止并重新排队相比，暂停被抢占借用方以保留其进度，使每个保持的 GPU 样本所产出的借用方令牌增加了 22%，且没有改变在线服务的中位延迟。这些结果来自同一加速器代际上每种策略的一次运行；容量数据是离线重放得出的上界；作业层和跟随/封存尚未进行端到端评估（§8）。将实例打包到卡组中，并根据到达模式调整空闲阈值，是接下来的工作。

11 附录：OneLendingHour

11.1 时间线

图6展示了暂停策略运行的一小时，按秒呈现，数据来自一次独立试验，并未纳入受控对比；该试验的请求密度高于跟踪记录的平均值。在线引擎与借用方在相同的 GPU 上交替使用，共发生了 23 次交接（借用方借用 12 次、被抢占 11 次）；在这一小时中，GPU 仅有 1.6 秒未被任何一个引擎

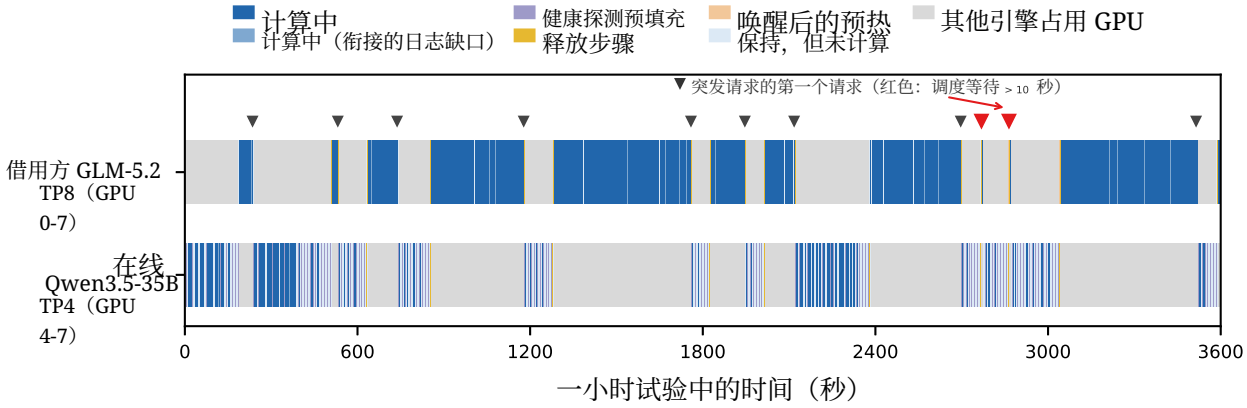

图6。 暂停策略持续一小时（此时的请求密度高于跟踪记录的平均值）：在线引擎（Qwen3.5-35B-A3B，GPU 4–7）和借用方（GLM-5.2，全部八个 GPU）在同一组 GPU 上交替运行。颜色表示引擎日志中的每秒状态：计算中（深蓝色；当日志行之间的间隔不超过 3 秒时，以浅蓝色表示桥接的间隔）、健康探测预填充、释放步骤、唤醒后的预热、保持但未计算，以及在其他引擎占用 GPU 时显示为灰色。三角形标记突发请求的第一个请求，该请求会触发交接；其中两个红色三角形的调度等待时间超过 10 秒（分别为 15.3 秒和 14.8 秒）。

占用。借用方在持有 GPU 的时间里，有 93% 用于计算，其余时间则用于释放步骤、唤醒后的预热和等待。在线引擎持有 GPU 共 1,560 秒，但实际计算时间仅约为其中的四分之一，因为它收到的请求持续时间短，且成簇到达，间隔短于空闲阈值；这就是处于唤醒状态的在线引擎会让卡闲置的原因。按资源池整体计算，所有引擎在这一小时内的计算时间占比为 40.5%，而在线服务单独运行时为 5.1%。

11.2 模型与测量结果对比

第7.2节的解析模型在四小时运行的 θ 并发度（$\theta = 30$秒，$B = 8$）下，预测暂停次数比重新入队多约 1.12×，而实测值为 1.22–1.25×（第7.3.2节）。方向和大致幅度一致。可能有两个解释：该在线服务的突发性高于模型所采用的 GPU 集群时间窗口，因此其抢占会中止更多工作；此外，重新排队运行中的引擎故障破坏了保留的前缀缓存，而模型在不保留缓存时将其代价估计为约 1.28× 。我们未对这两种解释进行测试；对于该跟踪记录和工作负载，实测比率成立。

参考文献

[1] R. Abhyankar、Z. He、V. Srivatsa、H. Zhang 和 Y. Zhang。InferCept：面向增强型大语言模型推理的高效拦截支持。载于第*41*届国际机器学习会议（*ICML*）论文集，2024年。arXiv：2402.01869。[2] S. Agarwal、S. Mitra、S. Chakraborty、S. Karanam、K. Mukherjee 和 S. K. Saini。用于高效服务文本到图像扩散模型的近似缓存。载于 第*21*届 *USENIX* 网络系统设计与实现研讨会（*NSDI 24*），2024年。网址：https: // www.usenix.org/conference/nsdi24/presentation/agarwal-shubham。arXiv：2312.04429。[3] A. Agrawal、N. Kedia、A. Panwar 等。使用 Sarathi-Serve 平衡大语言模型推理中的吞吐量-延迟权衡。载于第*18*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第117–134页，2024年。网址：https://www.usenix.org/conference/osdi24 / presentation/agrawal。[4] P. Ambati、Í. Goiri、F. V. Frujeri 等。为云平台中资源回收型虚拟机提供服务级别目标。载于 第*14*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*），第735–751页，2020年。网址：https://www.usenix.org/system/files/osdi20-paper- ambati.pdf。[5] Z. Bai、Z. Zhang、Y. Zhu 和 X. Jin。PipeSwitch：面向深度学习应用的快速流水线式上下文切换。载于 第*14*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*），第499–514页。USENIX 协会，2020年。网址：https://www.us enix.org/conference/osdi20/presentation/bai。[6] B. Burns、B. Grant、D. Oppenheimer、E. Brewer 和 J. Wilkes。Borg、Omega 与 Kubernetes。*ACM Queue*，14 114 5 1 70 93，2016 10 2898442 2898444 ( )：–。doi：。/。。网址：https://research.google/pubs/borg-omega- and-kubernetes/。[7] E. Cortez、A. Bonde、A. Muzio、M. Russinovich、M. Fontoura 和 R. Bianchini。Resource Central：理解并预测工作负载，以改进大型云平台中的资源管理。载于第*26*届操作系统原理研讨会（*SOSP '17*）论文集，第153–167页，2017年。doi：10.1145/3132747.3132772。[8] CRIU 项目。GPU 检查点 – CRIU。https://criu.org/GPU_检查点，2026年。访问日期：2026-10-28。[9] L. Cvetkovi´c、F. Costa、M. Djokic、M. Friedman 和 A. Klimovic。Dirigent：轻量级无服务器编排。载于*ACM SIGOPS* 第*30*届操作系统原理研讨会论文集（*SOSP '24*），第369–384页，2024年。doi：10.1145/3694715.3695966。[10] D. Du、T. Yu、Y. Xia 等。Catalyzer：借助无初始化启动实现亚毫秒级无服务器计算启动。载于第*25*届国际编程语言与操作系统体系结构支持会议（*ASPLOS '20*）论文集，第467–481页，2020年。doi：10.1145/3373376.3378512。[11] J. Duan、R. Lu、H. Duanmu 等。MuxServe：面向多大语言模型服务的灵活时空复用。载于第*41*届国际机器学习会议（*ICML 2024*）论文集，2024年。网址：https://icml.cc/virtual/2024/poster/34064。[12] R. Fan、T. Ren、M. Xie、S. Gao、J. Shu 和 Y. Lu。GPreempt：通用且高效的 GPU 抢占式调度。载于 *2025 USENIX* 年度技术会议（ *USENIX ATC 25* ），第 263–272 页，2025 年。网址：https://www.usenix.org/conference/atc25/presentation/fan。[13] W. Fu、J. Gao、X. Shen 等。AReaL：面向语言推理的大规模异步强化学习系统。载于神经信息处理系统进展 *38*（*NeurIPS 2025*），2025年。网址：https://neurips.cc/virtual/2025/poster/117538。[14] Y. Fu、L. Xue、Y. Huang 等。ServerlessLLM：面向大语言模型的低延迟无服务器推理。载于第*18*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第135–153页，2024年。网址：https://www.usenix.org/conference/osdi24/prese ntation/fu。[15] A. Fuerst 和 P. Sharma。FaasCache：通过贪心双重缓存维持无服务器计算。载于第*26*届 *ACM* 国际计算机体系结构 *f* 架构会议上的ು ತ್ತಾ ರೆ *pport f* 或程序 *g* 设计 *g* 语言与操作系统（*ASPLOS '21*400, 2021 386 10 1145 3445814 3446757），页码 – 。doi：。/。。[16] W. Gao、Y. Zhao、D. Muhtar 等。ROSE：通过协作式弹性在服务 GPU 上运行 rollout，以支持智能体强化学习，2026 年。URL：https://arxiv.org/abs/2605.06534。[17] A. Gujarati、R. Karimi、S. Alzayat 等。像 Clockwork 一样服务 DNN：自底向上的性能可预测性。载于第 *14* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*），第 443–462 页。USENIX Association，2020 年。[18] C. Guo、R. Zhang、J. Xu 等。GMLake：通过虚拟内存拼接，为大规模深度神经网络训练实现高效且透明的 GPU 内存碎片整理。载于第 *29* 届 *ACM* 国际编程语言与操作系统体系结构支持会议论文集，第 *2* 卷 3620 665*24*2024 10 1145 3640423（*ASPLOS '*），。doi：。/。。[19] M. Han、H. Zhang、R. Chen 和 H. Chen。面向并发 GPU 加速深度神经网络推理的微秒级抢占。载于第 *16* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 22*），第 539–558 页，2022 年。URL：https://www.usenix.org/conference/os di22/presentation/han。[20] B. Hindman、A. Konwinski、M. Zaharia 等。Mesos：数据中心细粒度资源共享平台。载于第 *8* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 11*），第 295–308 页，2011 年。URL：https://www.usenix.org/conference/ns di11/mesos-platform-fine-grained-resource-sharing-data-center。[21] J. Hu、X. Wu、W. Shen 等。OpenRLHF：基于 Ray 的易用、可扩展且高性能的基于人类反馈的强化学习框架。载于*2025* 年自然语言处理经验方法会议论文集：系统演示，第 656 186 53 666 页，2025 10 1 2025 48 2405 11143 – 。doi：。/v / .emnlp-demos. 。URL：https://arxiv.org/abs/。。[22] Q. Hu、Z. Ye、Z. Wang 等。数据中心中的大语言模型开发特征分析。载于第 *21* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 24*），第 709–729 页，2024 年。URL：https://www.usenix.org/system/files/nsdi24-hu.pdf。[23] S. Jaiswal、K. Jain、Y. Simmhan 等。SageServe：利用预测感知自动伸缩优化云数据中心中的大语言模型服务。*ACM* 计算系统测量与分析会议论文集（*POMACS*），9(3)，2025 年。URL：https://arxiv.org/abs/2502.14617。arXiv：2502.14617；v1 标题为“ Serving Models, Fast and Slow”。[24] H. S. Kabakibo、A. Trivedi 和 L. Wang。打破冷启动僵局：分析 vLLM 中的冷启动延迟。载于第 *9* 届机器学习与系统会议（*MLSys 2026*）论文集，2026 年。URL：https://arxiv.org/abs/2606.07362。arXiv：2606.07362。[25] KEDA 作者。KEDA：Kubernetes 事件驱动自动扩缩容。https://keda.sh/，2026 年。CNCF 毕业项目。访问日期：2026-10-28。[26] Knative 作者。配置缩容至零 – Knative。https://knat ive.dev/docs/serving/autoscaling/scale-to-zero/，2026 年。访问日期：2026-10-28。[27] Kubernetes 特别兴趣小组。Kueue：Kubernetes 原生作业排队。https://kueue.sigs.k8s.io/docs/overview/，2026 年。Kubernetes 特别兴趣小组项目。访问日期：2026-10-28。[28] W. Kwon、Z. Li、S. Zhuang 等。利用 PagedAttention 实现大语言模型服务的高效内存管理。载于第 *29* 届操作系统原理研讨会（*SOSP '23*）论文集，第 611–626 页，2023 年。URL：https://arxiv.org/abs/2309.06180。[29] C. Lee、Z. Zhu、T. Yang 等。SPES：面向优化无服务器函数的性能—资源权衡。载于*2024 IEEE* 第 *40* 届国际数据工程会议（*ICDE*），第 165–178 页，2024 年。doi：10.1109/ICDE60146.2024.00020。URL：https://arxiv.org/abs/2403.17574。[30] J. Li、H. Xu、Y. Zhu、Z. Liu、C. Guo 和 C. Wang。Lyra：深度学习集群的弹性调度。载于第十八届欧洲计算机系统会议（*EuroSys '23*）论文集，第 835–850 页，2023 年。doi：10.1145/3552326.3587445。URL：https://henryhxu.git hub.io/share/jiamin-eurosys23.pdf。

[31] M. Li、T. Cai、J. Cao 等。DistriFusion：面向高分辨率扩散模型的分布式并行推理。载于*IEEE/CVF*计算机视觉与模式识别会议（*CVPR*），2024 年。URL：https://arxiv.org/abs/2402.19481。arXiv：2402.19481。[32] Z. Li、L. Zheng、Y. Zhong 等。AlpaServe：利用模型并行实现深度学习服务的统计复用。载于第 *17* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 23*），第 663–679 页，2023 年。URL：https://www.usenix.org/conference/osdi23/presentation/li-zhouhan。[33] F. Liu、S. Zhang、X. Wang 等。时间步嵌入揭示：是时候为视频扩散模型使用缓存了。载于*IEEE/CVF*计算机视觉与模式识别会议（*CVPR*），2025 年。URL：https://arxiv.org/abs/2411.19108。arXiv：2411.19108。[34] llm-d项目。llm-d：架构。https://llm-d.ai/docs/architecture，2026 年。CNCF沙箱项目。访问日期：2026-10-28。[35] D. Lo、L. Cheng、R. Govindaraju、P. Ranganathan 和 C. Kozyrakis。Heracles：大规模提升资源效率。载于第 *42* 届年度国际计算机体系结构研讨会（*ISCA '15*）论文集，第 450–462 页，2015 年。doi：10.1145/2749469.2749475。[36] C. Lou、S. Qi、R. Kang 等。WarmServe：为多大语言模型服务实现一对多 GPU 预热。载于国际机器学习会议（*ICML 2026*），2026 年。URL：https://arxiv.org/abs/ 2512.09472。arXiv：2512.09472。[37] R. Lu、S. He、W. Tan 等。TetriServe：高效服务混合 DiT工作负载，2025 年。URL：https://arxiv.org/abs/2510.01565。[38] Z. Lv、C. Si、J. Song 等。FasterCache：高质量的无训练视频扩散模型加速。载于第十三届国际学习表征会议（*ICLR 2025*），2025 年。URL：https://arxiv.org/abs/2410.19355。arXiv：2410.19355。[39] X. Ma、G. Fang 和 X. Wang。DeepCache：免费加速扩散模型。载于*IEEE/CVF*计算机视觉与模式识别会议（*CVPR*），2024 0 0858 2312 2312 00858 。URL：https://arxiv.org/abs/。。arXiv：。。[40] X. Miao、C. Shi、J. Duan 等。SpotServe：在可抢占实例上服务生成式大语言模型。载于第 *29* 届 *ACM* 国际编程语言与操作系统体系结构支持会议论文集，第 *2* 卷（*ASPLOS '24*），第 1112–1127 页，2024 年。doi：10.1145/3620665.3640411。[41] K. K. W. Ng、H. M. Demoulin 和 V. Liu。Paella：利用软件定义GPU调度实现低延迟模型服务。载于第 *29* 届操作系统原理研讨会（*SOSP '23*）论文集，第 595–610 页，2023 年。doi：10.1145/3600006.3613163。[42] W. Nixon、J. Durbin、F. Standhartinger、H. S. Gunawi 和 J. Yang。大语言模型服务的一年：工作负载演化、缓存与负载均衡，2026 年。URL：https://arxiv.org/abs/2608.13573。[43] NVIDIA。cuda-checkpoint：CUDA检查点与恢复工具。https://github.com/NVIDIA/cuda-checkpoint，2026 年。NVIDIA。访问日期：2026-10-28。[44] E. Oakes、L. Yang、D. Zhou 等。SOCK：利用无服务器优化容器快速配置任务。载于*2018 USENIX* 年度技术会议（*USENIX ATC 18*），第 57–70 页，2018 年。URL：https://www.usenix.org/conference/atc18/presentation/oakes。[45] P. Patel、E. Choukse、C. Zhang 等。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 ACM/IEEE* 第 *51* 届年度国际计算机体系结构研讨会（*ISCA*），第 118–132 页，2024 年。doi：10.1109/ISCA59077.2024.00019。[46] A. Patke、D. Reddy、S. Jha 等。面向 SLO 的大语言模型服务的队列管理。载于*2024 ACM* 云计算研讨会论文集（*SoCC '24*），第 18–35 页，2024 年。doi：10.1145/3698038.3698523。URL：https://arxiv.org/abs/2407.00047。[47] R. Prabhu、A. Nayak、J. Mohan、R. Ramjee 和 A. Panwar。vAttention：无需 PagedAttention 即可服务大语言模型的动态内存管理。载于第 *30* 届 *ACM* 国际编程语言与操作系统体系结构支持会议论文集，第 *1* 卷（*ASPLOS '25*），第 1133–1150 页，2025 年。doi：10.1145/3669940.3707256。[48] Y. Qiao、S. Anzai、S. Yu 等。ConServe：面向大语言模型在线与离线协同服务的细粒度 GPU 回收。载于第 *43* 届国际机器学习会议（*ICML 2026*）论文集，2026 年。网址：https://arxiv.org/abs/2410.01228。arXiv:2410.01228。[49] R. Qin、Z. Li、W. He 等。Mooncake：以更多存储换取更少计算——面向大语言模型聊天机器人的以 KVCache 为中心的架构。载于第 *23* 届 *USENIX* 文件与存储技术会议（*FAST 25*），第 155–170 页，2025 年。网址：https://www.usenix.org/syste m/files/fast25-qin.pdf。[50] Ray Project。Kubernetes 上的 Ray（KubeRay）。https://docs.ray.io/en/lat est/cluster/kubernetes/index.html，2026 年。访问日期：2026-10-28。[51] R. B. Roy、T. Patel 和 D. Tiwari。IceBreaker：利用异构性更好地预热无服务器函数。载于第 *27* 届 *ACM* 国际编程语言与操作系统体系结构支持会议论文集（*ASPLOS '22*767，2022 753 10 1145 3503222 3507750），第 – 页。doi：。/。。[52] K. Rzadca、P. Findeisen、J. Swiderski 等。Autopilot：Google 的工作负载自动扩缩容。载于第十五届欧洲计算机系统会议论文集（*EuroSys '20*），2020 年。doi：10.1145/ 3342195.3387524。[53] M. Schwarzkopf、A. Konwinski、M. Abd-El-Malek 和 J. Wilkes。Omega：面向大型计算集群的灵活、可扩展调度器。载于第八届 *ACM* 欧洲计算机系统会议论文集（*EuroSys '13*），第 351–364 页，2013 年。doi：10.1145/2465351.2465386。[54] M. Shahrad、R. Fonseca、Í. Goiri 等。真实环境中的无服务器：表征并优化大型云服务提供商的无服务器工作负载。载于*2020* 年 *USENIX* 年度技术会议（*USENIX ATC 20*），第 205–218 页，2020 年。网址：https://www.usenix.org/conference/atc20/presentation/shahrad。[55] W. Shen、M. Han、J. Liu、R. Chen 和 H. Chen。XSched：面向多种 XPU 的抢占式调度。载于第 *19* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 25*692，2025 671 25），第 – 页。网址：https://www.usenix.org/conference/osdi / presentation/sh en-weihang。[56] G. Sheng、C. Zhang、Z. Ye 等。HybridFlow：灵活高效的基于人类反馈的强化学习框架。载于第二十届欧洲计算机系统会议论文集（*EuroSys '25*），第 1279–1297 页，2025 年。doi：10.1145/3689031.3696075。[57] D. Shukla、M. Sivathanu、S. Viswanatha 等。Singularity：人工智能工作负载的行星级抢占式弹性调度，2022 年。网址：https://arxiv.org/abs/2202.07848。[58] J. Stojkovic、C. Zhang、Í. Goiri、J. Torrellas 和 E. Choukse。DynamoLLM：设计兼顾性能与能效的大语言模型推理集群。载于*2025* 年 *IEEE* 国际高性能计算机体系结构研讨会（*HPCA*），第 1348–1362 页，2025 年。doi：10.1109/HPCA61900.2025.00102。[59] F. Strati、X. Ma 和 A. Klimovic。Orion：面向机器学习应用的干扰感知细粒度 GPU 共享。载于第十九届欧洲计算机系统会议论文集（*EuroSys '24*），第 1075–1092 页，2024 年。doi：10.1145/3627703.3629578。[60] B. Sun、Z. Huang、H. Zhao 等。Llumnix：面向大语言模型服务的动态调度。载于第 *18* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第 173–191 页，2024 年。网址：https://www.usenix.org/conference/osdi24/presentatio n/sun-biao。[61] AIBrix 团队和 J. Shan。AIBrix：迈向可扩展且高成本效益的大语言模型推理基础设施，2025 年。网址：https: // arxiv.org/abs/2504.03648。[62] torch_memory_saver 贡献者。torch_memory_saver：支持暂时释放张量内存，并在之后恢复。https://github .com/fzyzcjy/torch_memory_saver，2026 年。GitHub 仓库（MIT）。访问日期：2026-10-28。[63] D. Ustiugov、P. Petrov、M. Kogias、E. Bugnion 和 B. Grot。无服务器函数快照的基准测试、分析与优化。载于第 *26* 届 *ACM* 国际编程语言与操作系统体系结构支持会议论文集

（*ASPLOS '21*），第 559–572 页，2021 年。doi：10.1145/3445814.3446714。[64] A. Verma、L. Pedrosa、M. Korupolu、D. Oppenheimer、E. Tune 和 J. Wilkes。利用 Borg 实现 Google 的大规模集群管理。载于第十届欧洲计算机系统会议论文集（*EuroSys '15*）。ACM，2015 年。doi：10.1145/2741948.2741964。网址：https://research.google/pubs/large-scale-cluster-management-at-google-with-borg/。[65] vLLM 项目。睡眠模式——vLLM 文档。https://docs.vllm. ai/en/latest/features/sleep_mode.html，2026 年。访问日期：2026-10-28。[66] Volcano 作者。Volcano：云原生批处理调度系统。https://volcano.sh/en/docs/，2026 年。CNCF 孵化项目。访问日期：2026-10-28。[67] W. Wang、S. Xiong 和 G. Chen。大规模学习的强化学习优化：高效且易用的扩展库，2025 年。网址：https://arxiv.org/abs/2506.06122。[68] Y. Wang、Y. Chen、Z. Li 等。BurstGPT：用于优化大语言模型服务系统的真实世界工作负载数据集。载于第 *31* 届 *ACM* 知识发现与数据挖掘会议论文集 *V.2*3737 413*2*5831 5841，2025 10 1145 3711896（*KDD '*），第 – 页。doi：。/。。[69] X. Wei、Z. Huang、T. Sun 等。PhoenixOS：支持验证式推测的并发操作系统级 GPU 检查点与恢复。载于*ACM SIGOPS* 第 *31* 届操作系统原理研讨会论文集（*SOSP '25*），第 996–1013 页，2025 年。doi：10.1145/3731569.3764813。[70] Q. Weng、W. Xiao、Y. Yu 等。真实环境中的机器学习即服务：大规模异构 GPU 集群中的工作负载分析与调度。载于第 *19* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 22*），第 945–960 页，2022 年。网址：https://www.usenix.org/c onference/nsdi22/presentation/weng。[71] Y. Wu、X. Liu、H. Zheng 等。RLBoost：回收可抢占云资源，以实现大语言模型上高成本效率的强化学习。载于第 *23* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 26*），第 1323–1339 页，2026 年。网址：https://www.usen ix.org/system/files/nsdi26-wu-yongji.pdf。arXiv:2510.19225。[72] Y. Xiang、X. Li、K. Qian 等。Aegaeon：面向市场上并发大语言模型服务的高效 GPU 池化。载于*ACM SIGOPS* 第 *31* 届操作系统原理研讨会论文集（*SOSP '25*），第 1030–1045 页，2025 年。doi：10.1145/3731569.3764815。网址：https://ennanzhai.github.io/pub/sosp25-aegaeon.pdf。[73] Y. Xiang、X. Li、K. Qian、W. Yu、E. Zhai 和 X. Jin。ServeGen：生产环境中大语言模型服务的工作负载表征与生成。载于第 *23* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 26*），2026 年。网址：https: //arxiv.org/abs/2505.09999。arXiv:2505.09999。[74] W. Xiao、R. Bhardwaj、R. Ramjee 等。Gandiva：面向深度学习的自省式集群调度。载于第 *13* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 18*），第 595–610 页，2018 年。网址：https://www.usenix.org/conference/osdi18/prese ntation/xiao。[75] W. Xiao、S. Ren、Y. Li 等。AntMan：深度学习 GPU 集群上的动态扩展。载于第 *14* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*），第 533–548 页，2020 年。网址：https://www.usenix.org/conference/osdi20/presentation/xiao。[76] Z. Xie。SGLang HiCache：借助您钟爱的存储后端实现快速分层 KV 缓存。https://lmsys.org/blog/2025-09-10-sglang- hicache/，2025 年。LMSYS 组织博客，2025-09-10。访问日期：2026-11-05。[77] F. Ye、Z. Li、X. Zhong 等。GenServe：异构扩散模型工作负载的高效协同服务，2026 年。网址：https://arxiv.or g/abs/2604.04335。[78] G.-I. Yu、J. S. Jeong、G.-W. Kim、S. Kim 和 B.-G. Chun。Orca：面向基于 Transformer 的生成模型的分布式服务系统。载于第 *16* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 22*），第 521–538 页，2022 年。网址：https: //www.usenix.org/conference/osdi22/presentation/yu。[79] M. Yu、R. Yang、C. Jia 等。 λScale：实现无服务器大语言模型推理的快速扩展，2025 年。网址：https://arxiv.org/abs/2502.09922.[80] P. Yu 和 M. Chowdhury。Salus：面向深度学习应用的细粒度 GPU 共享原语。载于 机器学习与系统会议论文集（*MLSys 2020*046 10 2020 1902 1902 04610），。网址：https://arxiv.org/abs/。。arXiv：。。[81] S. Yu、Y. Qiao、M. Ma 等。Prism：通过 GPU 内存气球机制实现高成本效益的多大语言模型服务。载于 第 *20* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 26*），第 75–93 页，2026 年。网址：https://www.usenix.org/conference/osdi26/presentation/yu-shan。arXiv：2505.04021。[82] D. Zhang、H. Wang、Y. Liu 等。BlitzScale：借助 O(1) 主机缓存实现快速且在线的大模型自动扩缩容。载于 第 *19* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 25*），第 275–293 页，2025 年。网址：h ttps://www.usenix.org/conference/osdi25/presentation/zhang-dingyan。[83] W. Zhao、A. Jayarajan 和 G. Pekhimenko。Tally：并发深度学习工作负载的非侵入式性能隔离。载于 第 *30* 届 *ACM* 国际编程语言与操作系统体系结构支持会议论文集，第 *1* 卷 3707 282*25*1052 1068，2025 年 10 月 1145 3669940（*ASPLOS '*），第 – 页。doi：。/。。[84] X. Zhao、X. Jin、K. Wang 和 Y. You。借助金字塔注意力广播实现实时视频生成。载于 第十三届国际学习表征会议（*ICLR 2025*），2025 年。网址：https://arxiv.org/abs/2408.12588。arXiv：2408.12588。[85] L. Zheng、L. Yin、Z. Xie 等。SGLang：结构化语言模型程序的高效执行。载于 神经信息处理系统进展第 *37* 卷（*NeurIPS 2024*），2024 年。网址：https://proceedings.neurips.cc/paper _ 2024 724 447216 fil 8 31 1 9 630 15 8 es/paper/ /hash/ be f ba c ac f dec -Abstract-Conference.html。[86] Y. Zhong、S. Liu、J. Chen 等。DistServe：为吞吐效率优化的大语言模型服务解耦预填充与解码。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 年。网址：htt ps://www.usenix.org/conference/osdi24/presentation/zhong-yinmin。