Optimizing Speculative Decoding for Serving Large Language Models Using Goodput

Xiaoxuan Liu[1] Cade Daniel[2] Langxiang Hu[3] Woosuk Kwon[1] Zhuohan Li[1] Xiangxi Mo[1]
Alvin Cheung[1] Zhijie Deng[4] Ion Stoica[1] Hao Zhang[3]
[1]UC Berkeley [2]Anyscale Inc. [3]UCSD [4]SJTU

Abstract

Reducing the inference latency of large language models (LLMs) is crucial, and speculative decoding (SD) stands out as one of the most effective techniques. Rather than letting the LLM generate all tokens directly, speculative decoding employs effective proxies to predict potential outputs, which the LLM then verifies without compromising the generation quality. Yet, deploying SD in real online LLM serving systems (with continuous batching) does not always yield improvement – under higher request rates or low speculation accuracy, it paradoxically *increases* latency. Furthermore, there is no best speculation length work for all workloads under different system loads. Based on the observations, we develop a dynamic framework SmartSpec. SmartSpec dynamically determines the best speculation length for each request (from 0, i.e., no speculation, to many tokens) – hence the associated speculative execution costs – based on a new metric called *goodput*, which characterizes the current observed load of the entire system and the speculation accuracy. We show that SmartSpec consistently reduces average request latency by up to 3.2× compared to non-speculative decoding baselines across different sizes of target models, draft models, request rates, and datasets. Moreover, SmartSpec can be applied to different styles of speculative decoding, including traditional, model-based approaches as well as model-free methods like prompt lookup and tree-style decoding.

1 Introduction

Latency is critical when deploying Large Language Models (LLMs) for online services such as search engines [25, 32], chatbots [30], and virtual assistants [27, 38, 39]. However, LLM generation is inherently slow due to its autoregressive nature, where each generated token depends on all preceding tokens. This sequential data dependency restricts tokens to be generated one by one, resulting in slow generation speeds.

Speculative decoding aims to solve the problem by employing lightweight proxies, such as a small draft model [18, 20, 24, 26, 46] or additional model heads [4, 21, 22, 43], generating multiple tokens which are then verified by the main/target LLM in parallel. Speculative Decoding can reduce the generation latency mainly for two reasons. First, the efficient proxy is much faster to run compared with running a single forward pass on the target model and hence it can generate tokens much quicker. Moreover, the verification of the draft model is done in a single forward pass. Such verification is only *marginally* slower compared to letting LLM generate one new token, but it allows LLM to potentially generate *multiple* new tokens (if the guessed tokens are correct), or at least enable LLM to generate one new token (if all guessed tokens are incorrect).

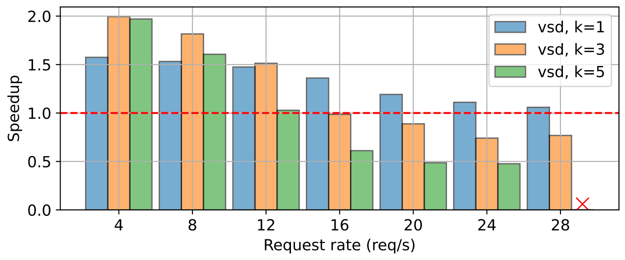


Figure 1. Speedup of vanilla speculative decoding (VSD) with proposed length $k = 1, 3, 5$ on a LLaMA-7B model with fixed input/output length = 128. We also fix the token acceptance rate to 0.7. We use LLaMA-160M as the draft model and conduct experiments on a single A100-80G. The red cross indicates the setting runs out of GPU memory and triggers swapping.

While SD has demonstrated promise in accelerating single-request inference (i.e., batch size = 1), integrating it into online serving systems poses significant challenges. In real-world applications, systems batch multiple tokens to achieve high GPU utilization and SD is less efficient or can even *increases* the query latency as shown in Fig. 1. This increase is primarily due to the additional computational overhead of running the proxy models and verifying tokens that are ultimately not accepted. Whether the extra computational work translates into actual latency reduction depends on two key factors: the speculation accuracy of each request and the current load of the serving system.

First, when the majority of proposed tokens are accepted (i.e., there is a high token acceptance rate), speculative decoding can provide significant speedup, provided that the proxy model operates efficiently. Conversely, if the token acceptance rate is low, speculative decoding can introduce additional overhead rather than accelerating the process. In such cases, the main model ends up regenerating most of the tokens, rendering the proxy model's computations superfluous and potentially slowing down the overall system.

优化大语言模型的投机解码以提高有效吞吐量

刘小轩[1] 凯德·丹尼尔[2] 郎翔·胡[3] 权宇硕[1] 李卓涵[1] 莫向西[1] 张阿尔文[1] 邓志杰[4] 伊昂·斯托伊卡[1] 张浩[3] [1] 加州大学伯克利分校 [2] Anyscale公司 [3] 加州大学圣地亚哥分校 [4] 上海交通大学

摘要

减少大型语言模型（LLMs）推理延迟至关重要，而推测解码（SD）作为最有效的技术之一脱颖而出。推测解码并不是让LLM直接生成所有标记，而是使用有效的代理来预测潜在输出，然后LLM在不影响生成质量的情况下进行验证。然而，在真实的在线LLM服务系统中（具有连续批处理），部署SD并不总是能带来改进——在请求率较高或推测准确性较低的情况下，反而会增加延迟。此外，没有适用于不同系统负载下所有工作负载的最佳推测长度。基于这些观察，我们开发了一个动态框架SmartSpec。SmartSpec动态确定每个请求的最佳推测长度（从0，即无推测，到多个标记）——因此与之相关的推测执行成本——基于一种称为良好吞吐量的新指标，该指标表征了整个系统当前观察到的负载和推测准确性。我们展示了SmartSpec在不同目标模型、草稿模型、请求率和数据集的情况下，平均请求延迟相比于非推测解码基线减少了最多3.2×。此外，SmartSpec可以应用于不同风格的推测解码，包括传统的基于模型的方法以及像提示查找和树形解码这样的无模型方法。

1 引言

延迟在部署大型语言模型（LLMs）用于在线服务时至关重要，例如搜索引擎 [25, 32]、聊天机器人 [30] 和虚拟助手 [27, 38, 39]。然而，由于其自回归特性，LLM 生成本质上是缓慢的，每个生成的标记都依赖于所有前面的标记。这种顺序数据依赖性限制了标记的逐个生成，导致生成速度缓慢。

推测解码旨在通过使用轻量级代理来解决问题，例如小型草稿模型 [18, 20, 24, 26, 46] 或额外的模型头 [4, 21, 22, 43]，生成多个令牌，然后由主/目标 LLM 并行验证。推测解码可以主要出于两个原因减少生成延迟。首先，与在目标模型上进行单次前向传递相比，高效的代理运行速度要快得多，因此它可以更快地生成令牌。此外，草稿的验证

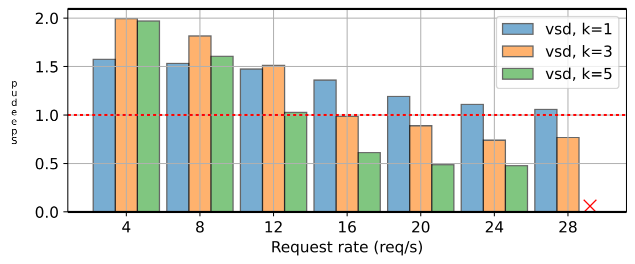


图 1. 在固定输入/输出长度 = 128 的 LLaMA-7B 模型上，使用提议的长度 $k = 1, 3, 5$ 的普通推测解码 (VSD) 的加速效果。我们还将令牌接受率固定为 0.7。我们使用 LLaMA-160M 作为草稿模型，并在单个 A100-80G 上进行实验。红色十字表示设置耗尽了 GPU 内存并触发了交换。

模型在单次前向传递中完成。与让LLM生成一个新标记相比，这种验证仅稍微慢一些，但它允许LLM潜在地生成多个新标记（如果猜测的标记是正确的），或者至少使LLM能够生成一个新标记（如果所有猜测的标记都是错误的）。

虽然SD在加速单请求推理（即批量大小 = 1）方面显示出潜力，但将其集成到在线服务系统中面临重大挑战。在实际应用中，系统会批量处理多个令牌以实现高GPU利用率，而SD的效率较低，甚至可能增加查询延迟，如图1所示。这一增加主要是由于运行代理模型和验证最终未被接受的令牌所带来的额外计算开销。额外的计算工作是否能转化为实际的延迟减少取决于两个关键因素：每个请求的推测准确性和服务系统的当前负载。

首先，当大多数提议的令牌被接受（即，令牌接受率高）时，推测解码可以提供显著的加速，前提是代理模型高效运行。相反，如果令牌接受率低，推测解码可能会引入额外的开销，而不是加速过程。在这种情况下，主模型最终会重新生成大部分令牌，使得代理模型的计算变得多余，并可能减慢整体系统的速度。

Second, due to the inherent imperfection in speculation accuracy, speculative decoding inevitably expends some computational resources on tokens that are not ultimately accepted. Under low system load, this waste of compute is tolerable. However, under high system load conditions—where the system approaches its computational capacity and processes large batch sizes—the availability of surplus compute for speculation is greatly reduced. In such scenarios, it is more effective to allocate the limited computational resources directly to normal token generation rather than continuing to speculate, thereby minimizing the risk and maximizing the use of available compute.

Ideally, speculative decoding should be performed in an adaptive and automatic way, which can be likened to the adaptive streaming techniques used in video delivery systems. In scenarios with few users, the system can afford to engage in "high-resolution" speculative decoding, akin to streaming high-resolution video, where it utilizes abundant computational resources to make more extensive predictions per request. Conversely, as user demand increases and system resources become strained, the system shifts to "low-resolution" speculative decoding. This strategy, much like reducing video resolution during peak traffic, involves making fewer predictions per request to conserve resources while maintaining overall system functionality.

Despite its widespread recognition, speculative decoding has not yet been effectively integrated into production-level serving systems. Most prior research has explored speculative decoding with a batch size of one [4, 20]. Recent studies have extended this investigation to larger batch sizes, but these techniques have been tested primarily on relatively small LLMs [35] or in offline settings [37].

In this work, we integrate speculative decoding into a production-level serving system vLLM [19], marking the first such integration to the best of our knowledge. We also explore the trade-offs between speculation cost and decoding accuracy under varying system loads. A key innovation in our system is the introduction of a metric called "goodput,", defined as the rate of generated tokens per second. Unlike throughput, goodput in the context of speculative decoding measures only those tokens that are both verified and generated by the target model. This reflects a crucial distinction – not all output tokens qualify as generated tokens.

Goodput is an essential metric for determining the extent of speculation. It is derived from two factors: the token acceptance rate and the batch size, with the latter indicating system load. This metric adheres to two core principles. First, it limits speculation under constrained computational resources to maximize system efficiency. For example, under extremely high system loads, goodput would automatically turn off speculation to avoid wasting any compute resources. Second, this metric increases the proposed length for requests that are easy to predict, as indicated by the high token acceptance rate in previous generation steps. By leveraging the predictability of these queries, the system can enhance overall performance.

However, we cannot measure goodput directly because the decision needs to be made before knowing the goodput. We must determine the proposed length and which requests to run (batch size) based on an estimate of goodput, as these two factors influence its value. To estimate goodput, we predict the accepted token length for all requests within a single generation step using the token acceptance rate. A simple linear model is then employed to estimate the batch execution time. By combining the predicted token length and execution time, we can approximate goodput.

Leveraging goodput, we developed the dynamic speculative decoding framework SmartSpec. SmartSpec dynamically modulates each request's speculation length – from no speculation (i.e., zero) to many tokens – based on the estimated *goodput*, adjusting speculation intensities to ensure consistent reduction (instead of increase) of the request latency. SmartSpec also accommodates various speculative decoding methods, including draft model-based approaches and model-free techniques such as prompt lookup [33] and tree-based decoding [4]. Accommodating diverse SD methods is crucial because different techniques are suited to different workloads. For instance, prompt lookup decoding is more beneficial for summarization tasks, while tree-based approaches like Medusa are more useful for online chatting. For all SD methods evaluated across all datasets, SmartSpec guarantees improved performance without any degradation, which is a crucial feature for making SD useful in a production-ready online serving system.

In summary, the paper makes the following contributions:

- We perform the first study on speculative decoding within a real-world, online serving system with continuous batching scheduling (§3).
- We define the goodput for speculative decoding, which takes into account both system throughput and speculation accuracy (§4).
- We design and implement a speculative decoding scheduling framework that utilizes goodput as key metrics to determine the optimal proposal length for different request volumes (§5, §6). Evaluation of SmartSpec on five models across different tasks shows that SmartSpec consistently reduces latency under different system loads, bringing up to 3.2× latency reduction compared with non-speculative decoding baseline (§7).

2 Background

Given a list of tokens $(x_1, \ldots, x_n)$, a large language model (LLM) [1, 3] is trained to predict the conditional probability distribution for the next token: $P(x_{n+1} \mid x_1, \ldots, x_n)$. When deployed as a service [19, 31], the LLM takes in a list of

其次，由于推测准确性固有的不完美，推测解码不可避免地会在最终未被接受的标记上消耗一些计算资源。在低系统负载下，这种计算浪费是可以容忍的。然而，在高系统负载条件下——系统接近其计算能力并处理大批量数据时——用于推测的多余计算资源的可用性大大减少。在这种情况下，直接将有限的计算资源分配给正常的标记生成比继续推测更有效，从而最小化风险并最大化可用计算的使用。

理想情况下，推测解码应以自适应和自动化的方式进行，这可以类比于视频传输系统中使用的自适应流技术。在用户较少的场景中，系统可以进行“高分辨率”的推测解码，类似于流式传输高分辨率视频，在这种情况下，它利用丰富的计算资源为每个请求做出更广泛的预测。相反，随着用户需求的增加和系统资源的紧张，系统转向“低分辨率”的推测解码。这一策略，类似于在高峰流量期间降低视频分辨率，涉及为每个请求做出更少的预测，以节省资源，同时保持整体系统功能。

尽管得到了广泛认可，推测解码尚未有效地融入生产级服务系统。大多数先前的研究探讨了批量大小为一的推测解码 [4, 20]。最近的研究将这一调查扩展到更大的批量大小，但这些技术主要在相对较小的LLM上 [35] 或在离线环境中 [37] 进行了测试。

在这项工作中，我们将推测解码集成到生产级服务系统 vLLM [19] 中，尽我们所知，这是首次进行这样的集成。我们还探讨了在不同系统负载下推测成本与解码准确性之间的权衡。我们系统的一个关键创新是引入了一种称为“有效输出”（goodput）的指标，定义为每秒生成的令牌数量。与吞吐量不同，在推测解码的背景下，有效输出仅衡量那些既经过验证又由目标模型生成的令牌。这反映了一个关键的区别——并非所有输出令牌都符合生成令牌的资格。

良好吞吐量是确定投机程度的重要指标。它由两个因素派生而来：令牌接受率和批量大小，后者表示系统负载。该指标遵循两个核心原则。首先，它在受限的计算资源下限制投机，以最大化系统效率。例如，在极高的系统负载下，良好吞吐量会自动关闭投机，以避免浪费任何计算资源。其次，该指标增加了对易于预测的请求的建议长度，这在先前的生成步骤中通过高令牌接受率得以体现。通过利用 {v*} 这些查询的可预测性使系统能够提高整体性能。

然而，我们无法直接测量有效吞吐量，因为在知道有效吞吐量之前需要做出决策。我们必须根据对有效吞吐量的估计来确定建议的长度和要运行的请求（批量大小），因为这两个因素会影响其值。为了估计有效吞吐量，我们使用令牌接受率预测单个生成步骤中所有请求的接受令牌长度。然后，采用一个简单的线性模型来估计批量执行时间。通过结合预测的令牌长度和执行时间，我们可以近似有效吞吐量。

利用良好吞吐量，我们开发了动态推测解码框架 SmartSpec。SmartSpec 动态调节每个请求的推测长度——从没有推测（即零）到多个标记——基于估计的良好吞吐量，调整推测强度以确保请求延迟的一致减少（而不是增加）。SmartSpec 还兼容各种推测解码方法，包括基于草稿模型的方法和无模型技术，如提示查找 [33] 和基于树的解码 [4]。兼容多样的 SD 方法至关重要，因为不同的技术适用于不同的工作负载。例如，提示查找解码对摘要任务更有利，而像 Medusa 这样的基于树的方法对在线聊天更有用。对于在所有数据集上评估的所有 SD 方法，SmartSpec 保证在没有任何性能下降的情况下提高性能，这是使 SD 在生产就绪的在线服务系统中有用的关键特性。

总之，本文做出了以下贡献：

- 我们在一个具有连续批处理调度的真实在线服务系统中进行首次关于推测解码的研究（§3）。
- 我们定义了用于推测解码的有效吞吐量，它同时考虑了系统吞吐量和推测准确性（§4）。
- 我们设计并实现了一个推测解码调度框架，该框架利用有效吞吐量作为关键指标，以确定不同请求量的最佳提案长度（§5，§6）。对五个模型在不同任务上的SmartSpec评估表明，SmartSpec在不同系统负载下始终减少延迟，与非推测解码基线相比，延迟减少最多可达3.2×（§7）。

2 背景

给定一个令牌列表 $(x_1, \ldots, x_n)$，一个大型语言模型 (LLM) [1, 3] 被训练来预测下一个令牌的条件概率分布：$P(x_{n+1} \mid x_1, \ldots, x_n)$. 当作为服务部署时 [19, 31]，LLM 接收一个令牌列表

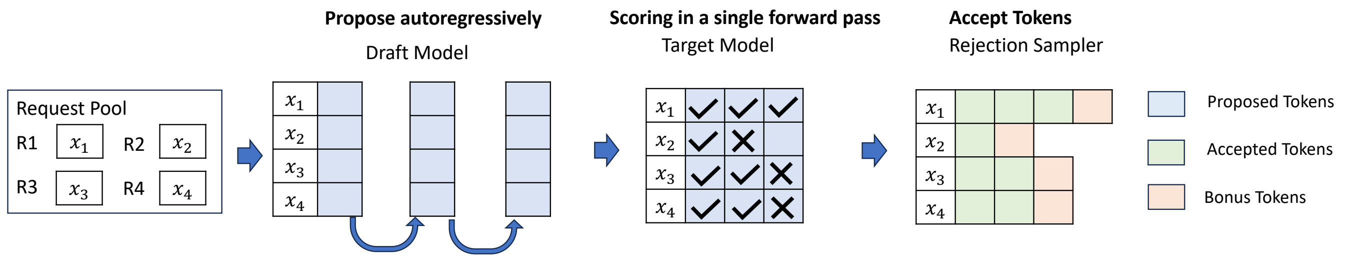


Figure 2. A single generation step when combining continuous batching with speculative decoding. The draft model runs in an autogressive manner. The proposed tokens are sent to the target model for scoring in a single forward pass. A single generation step can generate more than one token for each request.

tokens from the user request and generates an output sequence $(x_{n+1}, \ldots, x_{n+T})$. The generation process requires sequentially evaluating the probability and samples the token at every position for T times.

Due to the sequential data dependency, this computation often suffers from low device utilization when running on GPUs, which leads to high inference latency and low serving throughput [31]. Therefore, many previous works propose different algorithms to decrease the latency or increase the throughput when serving LLMs. In this paper, we focus on two categories of optimization algorithms, *speculative decoding* and *continuous batching*.

2.1 Speculative Decoding

Although LLMs can only generate output tokens sequentially, when facing a list of output tokens $(x_{n+1}, \ldots, x_{n+T})$, LLMs can efficiently evaluate the probabilities for each token $P(x_{n+1} \mid x_1, \ldots, x_n), \ldots, P(x_{n+T} \mid x_1, \ldots, x_{n+T-1})$ in parallel. *Speculative decoding* [5, 20] utilizes this property to reduce the generation latency of LLMs.

Specifically, in speculative decoding, we turn the target LLM into an evaluator. At every step, We use another more efficient draft model to propose a list of candidate tokens $(y_{n+1}, \ldots, y_{n+k})$, where k is the number of proposed candidates. Then, we feed these k tokens to the target LLM to evaluate the probabilities $P(y_{n+1} \mid x_1, \ldots, x_n), \ldots, P(y_{n+k} \mid x_1, \ldots, x_n, y_{n+1}, \ldots, y_{n+k-1})$ in parallel. Based on the probabilities and sampling methods, we will accept a subset of tokens $y_1, \ldots, y_m$, where m is the number of accepted tokens. As an example, for greedy sampling, we check whether each y_{n+i} is the token that maximizes the probability distribution $P(\cdot \mid x_1, \ldots, x_n, y_{n+1}, \ldots, y_{n+i-1})$ and accept the first m tokens $y_1, \ldots, y_m$ that satisfy the condition. Note that for the position $m+1$, we can directly sample y_{m+1} from the distribution $P(\cdot \mid x_1, \ldots, x_n, y_{n+1}, \ldots, y_{n+m-1})$. Finally, we will take $m+1$ tokens $y_1, \ldots, y_{m+1}$ as LLM outputs for this step.

Speculative decoding has two core properties: (1) Speculative decoding does not change the behavior of the LLM sampling process, and thus generates exactly the same output as vanilla decoding algorithms without any accuracy loss. (2) The efficiency and the effective speedup of speculative decoding algorithms depend on two factors: the accuracy of the draft model matching the outputs of the target model and the efficiency of the draft model.

Many previous work focuses on improving the accuracy and efficiency of speculative decoding and can be categorized into two parts: (1) Draft LLM-based speculative decoding, which uses a small LLM as a draft model to propose candidate tokens [6, 24, 26, 40, 46]. (2) Draft-model free speculative decoding, which uses either a branch of the target model or uses other sources (e.g., from a external database) to generate the candidate tokens [4, 10, 21, 22, 33]. In this work, we study the behavior of both types of speculative decoding method.

2.2 Continuous Batching

Due to the sequential dependency, when generating output for a single output, LLMs severely under-utilize the GPUs. To increase the GPU utilization, one can batch multiple requests in one step and process them in parallel. However, batching the requests to an LLM is non-trivial: First, the requests may arrive at different times. A naive batching strategy would either make earlier requests wait for later ones or delay the incoming requests until earlier ones finish, leading to significant queueing delays. Second, the requests may have vastly different input and output lengths. A straightforward batching technique would pad the inputs and outputs of the requests to equalize their lengths, wasting GPU computation and memory.

Continuous batching [11, 41] is proposed to address this problem. Instead of batching at the request level, continuous batching batches at the step level. For each step, completed requests from previous step are removed from the batch, and newly received requests are added. Therefore, a new request can immediately start to be processed after it is received. This leads to a larger batch size at every step, which improves GPU utilization and thus the serving throughput. Moreover, with special GPU kernels, continuous batching can eliminate the need to pad requests of different lengths, which further improves the serving throughput. The technique has been

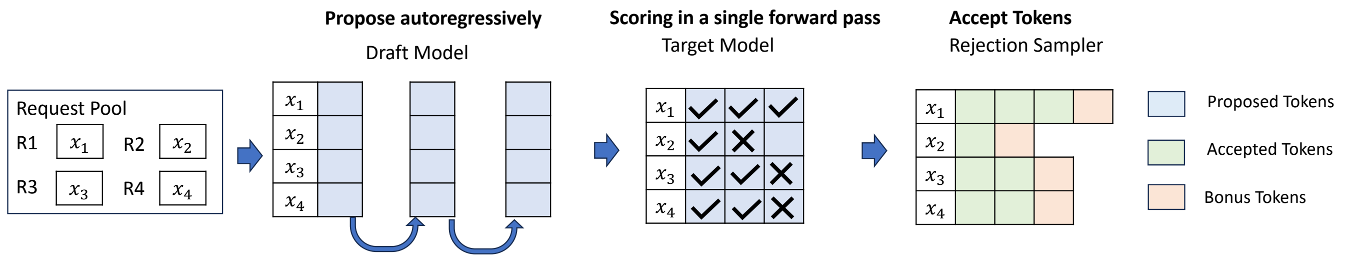


图2. 在将连续批处理与推测解码相结合时的单个生成步骤。草稿模型以自回归方式运行。提议的标记在单次前向传递中被发送到目标模型进行评分。单个生成步骤可以为每个请求生成多个标记。

从用户请求中提取的标记并生成输出序列 $(x_{n+1}, \ldots, x_{n+T})$。生成过程需要依次评估概率，并在每个位置采样标记 T 次。

由于序列数据的依赖性，这种计算在GPU上运行时常常会遭受低设备利用率，从而导致高推理延迟和低服务吞吐量[31]。因此，许多先前的工作提出了不同的算法，以减少延迟或提高服务LLM时的吞吐量。在本文中，我们关注两类优化算法，推测解码和连续批处理。

2.1 投机解码

尽管大型语言模型（LLMs）只能顺序生成输出标记，但在面对输出标记列表 $(x_{n+1}, \ldots, x_{n+T})$ 时，LLMs 可以高效地并行评估每个标记 $P(x_{n+1} \mid x_1, \ldots, x_n), \ldots, P(x_{n+T} \mid x_1, \ldots, x_{n+T-1})$ 的概率。推测解码 [5, 20] 利用这一特性来减少 LLMs 的生成延迟。

具体来说，在推测解码中，我们将目标 LLM 转变为评估器。在每一步，我们使用另一个更高效的草稿模型来提出候选标记列表 $(y_{n+1}, \ldots, y_{n+k})$,，其中 k 是提议的候选数量。然后，我们将这些 k 标记输入目标 LLM，以并行评估概率 $P(y_{n+1} \mid x_1, \ldots, x_n), \ldots, P(y_{n+k} \mid x_1, \ldots, x_n, y_{n+1}, \ldots, y_{n+k-1})$。根据概率和采样方法，我们将接受一部分标记 $y_1, \ldots, y_m$，其中 m 是接受的标记数量。作为一个例子，对于贪婪采样，我们检查每个 y_{n+i} 是否是最大化概率分布 $P(\cdot \mid x_1, \ldots, x_n, y_{n+1}, \ldots, y_{n+i-1})$ 的标记，并接受满足条件的前 m 个标记 $y_1, \ldots, y_m$。请注意，对于位置 $m+1$，我们可以直接从分布 $P(\cdot \mid x_1, \ldots, x_n, y_{n+1}, \ldots, y_{n+m-1})$. 中采样 y_{m+1}。最后，我们将取 $m+1$ 个标记 $y_1, \ldots, y_{m+1}$ 作为此步骤的 LLM 输出。

推测解码有两个核心特性：（1）推测解码不会改变LLM采样过程的行为，因此生成的输出与普通解码算法完全相同，且没有任何准确性损失。(2) 投机解码算法的效率和有效加速取决于两个因素：草稿模型与目标模型输出的匹配准确性以及草稿模型的效率。

许多之前的工作集中在提高推测解码的准确性和效率，并可以分为两部分：（1）基于草稿LLM的推测解码，它使用一个小型LLM作为草稿模型来提出候选标记 [6, 24, 26, 40, 46]。（2）无草稿模型的推测解码，它使用目标模型的一个分支或使用其他来源（例如，来自外部数据库）来生成候选标记 [4, 10, 21, 22, 33]。在这项工作中，我们研究这两种类型的推测解码方法的行为。

2.2 连续批处理

由于顺序依赖性，在为单个输出生成输出时，LLM严重低估了GPU的利用率。为了提高GPU的利用率，可以在一步中批量处理多个请求并并行处理。然而，将请求批量发送到LLM并非易事：首先，请求可能在不同的时间到达。一个简单的批量策略要么让较早的请求等待较晚的请求，要么延迟到达的请求直到较早的请求完成，从而导致显著的排队延迟。其次，请求的输入和输出长度可能差异很大。一种简单的批量技术会对请求的输入和输出进行填充，以使它们的长度相等，这会浪费GPU的计算和内存。

连续批处理 [11, 41] 被提出来解决这个问题。与其在请求级别进行批处理，不如在步骤级别进行连续批处理。对于每个步骤，来自前一步的已完成请求会从批处理中移除，而新接收的请求会被添加。因此，新请求在接收后可以立即开始处理。这导致每个步骤的批量大小更大，从而提高了 GPU 的利用率，进而提高了服务吞吐量。此外，通过特殊的 GPU 内核，连续批处理可以消除对不同长度请求进行填充的需求，这进一步提高了服务吞吐量。该技术已经被

integrated in all popular LLM inference engines, such as vLLM [19] and TensorRT-LLM [29].

3 Speculative Decoding with Continuous Batching

Speculative decoding changes continuous batching by allowing each generation step to produce multiple rather than a single token per request. It utilizes a draft model to suggest a range of possible tokens for a request at each generation step. These proposed tokens for all requests are then collectively processed in a batch by the target model for verification.

Figure 4 illustrates the three phases of speculative decoding: proposal, scoring, and acceptance. In proposal, the draft model examines the request pool and generates tokens in an autoregressive manner. During scoring, all the proposed tokens are evaluated collectively in a single forward pass. After accepting the tokens with rejection sampling, each request can yield multiple tokens in a single pass. The generated tokens comprise those proposed by the draft model and subsequently accepted by the target model, plus a bonus token. This bonus token either corrects a prediction made by the draft model or is generated by the target model when it accepts all proposed tokens.

3.1 Vanilla Speculative Decoding Latency

To understand the performance implication of vanilla speculative decoding in the context of continuous batching, we conduct an analysis shown in Fig. 1, showcasing the speedup achieved under varying request rates. In this analysis, to mitigate confounding variables, we set the token acceptance rate (0.7) and standardize the input and output lengths across all requests (input length=output length=128). The results show that at a low request rate (specifically, a request rate of 4), proposing 3 or 5 tokens results in the most significant speedup. However, as the request rate increases, the advantage of proposing more tokens diminishes rapidly: when the request rate exceeds 12, proposing 5 tokens offers no performance improvements. Likewise, at a request rate greater than 16, proposing 3 tokens yields performance degradation.

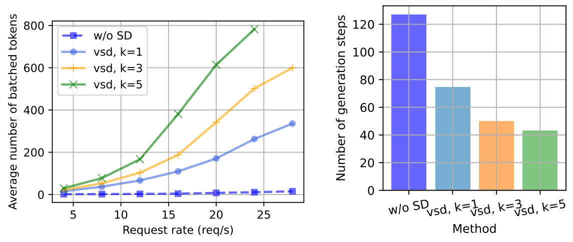
Several insights emerged from this experiment. Firstly, speculative decoding does not invariably lead to improved performance; in fact, it may detrimentally affect performance at higher request rates. Secondly, the optimal length for speculation varies with the request rate. At lower request rates, speculating more is better, but at higher request rates, it may not even make sense to speculate at all.

3.2 Latency Analysis

To understand the phenomenon, we can approximate the request latency as:

$$\textit{request latency} \approx \textit{batch latency} \times \textit{generation steps} \quad (1)$$

where *batch latency* refers to the time required to process a single batch and *generation steps* is the average number



(a) Average batch size. **(b)** Average number of generation steps.

Figure 3. Latency analysis: batch size and generation step.

of iterations needed for the target model to complete a request. For simplicity, we exclude the latency associated with the draft model, assume uniform latency across different generation steps, and focus solely on the *batch latency* per *generation step* incurred by the target model. Fig. 3a illustrates that proposing more tokens at each step leads to a greater accumulation of tokens within the same batch and hence higher *batch latency*. On the other hand, as shown in Fig. 3b, generating more tokens in a single forward pass of the target model reduces the number of *generation steps*. Given the approximation presented in Eq. 1, the feasibility of achieving a speedup is determined by the interplay between these two factors.

3.3 Granularity of Proposed Lengths

Lastly, continuous batching enables flexible scheduling. As shown in Fig. 4, there are three levels of granularity for proposed lengths: (1) Global: This is the simplest way to implement speculative decoding with continuous batching, where the proposed length for all requests across all generation steps is uniform. However, this approach overlooks system behavior; as previously mentioned, speculative decoding can degrade performance. (2) Step Level: Here all requests within the same batch have the same proposed length, although the length can vary between steps. This allows proposed lengths to adapt to different system loads. For instance, when the number of requests is high, the proposed length for a given step can be reduced to conserve computational resources. (3) Request Level: This is the most fine-grain level of scheduling, where each request can have its own proposed length. It allows for the prioritization of 'easier' requests by proposing a higher number of tokens for them, based on the assumption that it is more likely for more tokens to be generated in a single step for these requests.

In this section, we analyze the performance characteristics of naive speculative decoding with continuous batching across various request rates. We explore the reasons for performance degradation and highlight the possibility of implementing flexible scheduling. Determining the optimal proposed length to achieve minimal latency across diverse

集成在所有流行的LLM推理引擎中，例如vLLM [19]和TensorRT-LLM [29]。

3 连续批处理的推测解码

推测解码通过允许每个生成步骤为每个请求生成多个而不是单个标记，从而改变了连续批处理。它利用草稿模型在每个生成步骤中为请求建议一系列可能的标记。然后，这些为所有请求提出的标记被目标模型集体处理，以进行验证。

图4展示了推测解码的三个阶段：提议、评分和接受。在提议阶段，草稿模型检查请求池并以自回归的方式生成标记。在评分阶段，所有提议的标记在一次前向传递中被集体评估。在通过拒绝采样接受标记后，每个请求可以在一次传递中产生多个标记。生成的标记包括草稿模型提议的那些标记以及随后被目标模型接受的标记，外加一个额外的标记。这个额外的标记要么纠正草稿模型所做的预测，要么是在目标模型接受所有提议的标记时生成的。

3.1 香草推测解码延迟

为了理解在连续批处理上下文中香草推测解码的性能影响，我们进行了一项分析，如图1所示，展示了在不同请求速率下实现的加速。在这项分析中，为了减轻混杂变量的影响，我们设置了令牌接受率（0.7），并在所有请求中标准化输入和输出长度（输入长度=输出长度=128）。结果表明，在低请求速率（具体来说，请求速率为4）下，提出3个或5个令牌会带来最显著的加速。然而，随着请求速率的增加，提出更多令牌的优势迅速减小：当请求速率超过12时，提出5个令牌并没有带来性能提升。同样，在请求速率大于16时，提出3个令牌会导致性能下降。

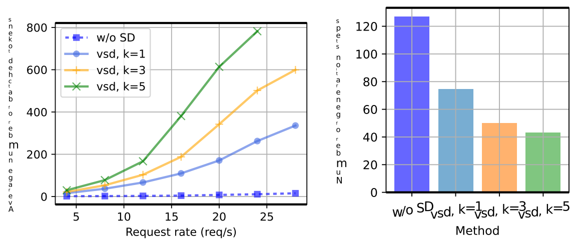
从这个实验中出现了几个见解。首先，推测解码并不总是导致性能提升；事实上，在较高请求率下，它可能会对性能产生不利影响。其次，推测的最佳长度随着请求率的变化而变化。在较低请求率下，更多的推测是更好的，但在较高请求率下，推测可能根本没有意义。

3.2 延迟分析

要理解这一现象，我们可以将请求延迟近似为：

$$request\ latency \approx batch\ latency \times generation\ steps \quad (1)$$

批处理延迟是指处理单个批次所需的时间，而生成步骤是平均数量。



(a) Average batch size.

(b) Average number of generation steps.

图 3. 延迟分析：批量大小和生成步骤。

所需的迭代次数，以便目标模型完成请求。为简化起见，我们排除了与草稿模型相关的延迟，假设不同生成步骤之间的延迟是均匀的，并且仅关注目标模型在每个生成步骤中产生的批量延迟。图3a说明，在每个步骤中提出更多的令牌会导致同一批次内令牌的更大累积，从而导致更高的批量延迟。另一方面，如图3b所示，在目标模型的单次前向传递中生成更多令牌会减少生成步骤的数量。根据公式1中提出的近似，获得加速的可行性取决于这两个因素之间的相互作用。

3.3 提议长度的粒度

最后，连续批处理实现了灵活的调度。如图4所示，提议长度有三个粒度级别：（1）全局：这是实现连续批处理的推测解码的最简单方法，其中所有请求在所有生成步骤中的提议长度是统一的。然而，这种方法忽视了系统行为；如前所述，推测解码可能会降低性能。（2）步骤级别：在这里，同一批次中的所有请求具有相同的提议长度，尽管长度可以在步骤之间变化。这允许提议长度适应不同的系统负载。例如，当请求数量较高时，给定步骤的提议长度可以减少，以节省计算资源。（3）请求级别：这是调度的最细粒度级别，每个请求可以有其自己的提议长度。它允许通过为这些请求提议更多的标记来优先考虑“更简单”的请求，基于这样的假设：在单个步骤中生成更多标记的可能性更高。

在本节中，我们分析了在不同请求速率下，使用连续批处理的简单推测解码的性能特征。我们探讨了性能下降的原因，并强调了实现灵活调度的可能性。确定最佳提议长度以在多样化的环境中实现最小延迟。

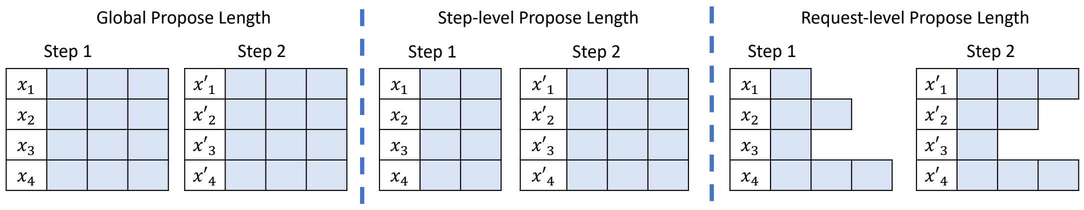


Figure 4. Flexible proposed lengths.

workloads under different request volumes is a significant challenge that we address in the following discussion.

4 The Goodput of Speculative Decoding

We now define the *goodput* of serving LLMs with speculative decoding and elaborate on how it is connected to the overall system efficiency. Then we describe how goodput can be estimated given various prediction methods on the acceptance of speculated tokens.

4.1 Defining Goodput

We define the *per-step* throughput of serving a request using an LLM as:

$$Throughput = \frac{Number\, of\, \textbf{Output}\, Tokens}{Execution\, Time} \quad (2)$$

Throughput refers to the output rate of tokens generated by the model per unit of time. Existing systems such as vLLM [19] and Orca [41] all aim to maximize the throughput, as doing so enhances the overall efficiency of the system.

Goodput in speculative decoding. In speculative decoding, not all tokens output by the target model in the scoring phase (Fig. 4) are guaranteed to pass through the rejection sampling mechanism. Consequently, these tokens might not represent the actual tokens generated in a single step. To address this discrepancy, we define *goodput* as:

$$Goodput = \frac{Number\, of\, \textbf{Generated}\, Tokens}{Execution\, Time} \quad (3)$$

Here, the goodput refers to the rate at which tokens are generated, measured in tokens per second. This includes both the proposed tokens that are subsequently accepted and the bonus tokens that the target model produces during verification.

Parameters of Goodput. While the above definition is general across different speculative decoding algorithms, we focus on three configuration parameters of particular impact in the context of speculative decoding scheduling:

1. Proposed length: the number of tokens proposed by the draft model in each step.
2. Requests to run: which request(s) to run in each step.

4.2 Understanding Goodput

Goodput, essentially defined as the ratio of expected gain to costs, offers valuable insights into how batch size and proposed length should interact to optimize system performance.

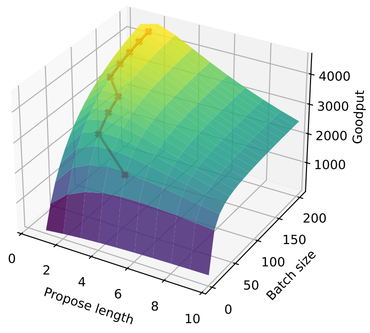


Figure 5. Goodput as a function of proposed length and batch size. We calculated the goodput using the coefficients from the A100-7B model, as detailed in Sec. 4.3. We assumed a uniform token acceptance rate of 0.7 and employed the formula described in Section 4.4 to estimate the accepted length.

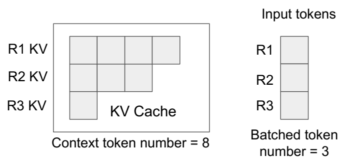


Figure 6. An example of context token number and batched token number used in modeling the forward execution time.

To demonstrate the intuition behind goodput, Fig. 5 shows the goodput values across various batch sizes and proposed lengths, assuming a uniform token acceptance rate.

Propose more for small batches. In Fig. 5, the red line indicates the optimal proposed length for each batch size. Notably, small batch sizes require proposing more than 4 tokens per per request to achieve the maximum goodput. As batch size increases, the optimal proposal length decreases.

Propose less for large batches. For large batch sizes, not speculate altogether can result in higher goodput. This occurs as the cost of unsuccessful speculations increases significantly with larger batch sizes, outpacing the potential gains.

Prefer batching over speculating. Consider a scenario where the acceptance of tokens is independent, with each token having a 0.7 probability of acceptance. In this case, the probability of accepting the first token is 0.7, while the

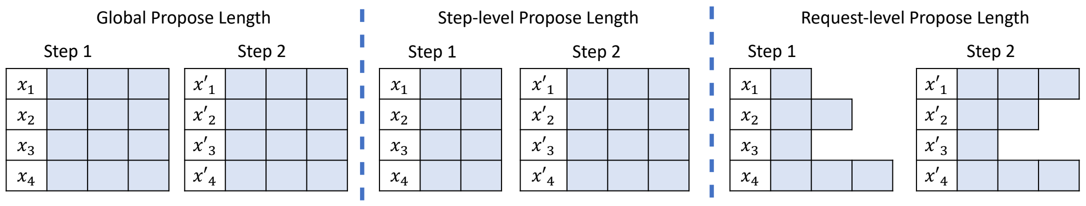


图 4. 灵活的建议长度。

在不同请求量下的工作负载是我们在以下讨论中要解决的一个重大挑战。

4 投机解码的有效吞吐量

我们现在定义使用推测解码服务大型语言模型（LLMs）的有效输出，并详细说明它与整体系统效率的关系。然后，我们描述如何根据对推测令牌接受的各种预测方法来估计有效输出。

4.1 定义有效吞吐量

我们将使用LLM处理请求的每步吞吐量定义为：

$$Throughput = \frac{Number\ of\ \textbf{Output}\ Tokens}{Execution\ Time} \qquad (2)$$

吞吐量是指模型每单位时间生成的令牌输出速率。现有系统如 vLLM [19] 和 Orca [41] 都旨在最大化吞吐量，因为这样可以提高系统的整体效率。

在推测解码中的有效输出。在推测解码中，目标模型在评分阶段（图4）输出的并非所有标记都能通过拒绝采样机制。因此，这些标记可能并不代表在单一步骤中生成的实际标记。为了解决这一差异，我们将有效输出定义为：

$$Goodput = \frac{Number\ of\ \textbf{Generated}\ Tokens}{Execution\ Time} \qquad (3)$$

在这里，良好输出指的是生成令牌的速率，以每秒生成的令牌数来衡量。这包括随后被接受的提议令牌和目标模型在验证过程中产生的奖励令牌。

良好吞吐量的参数。虽然上述定义在不同的推测解码算法中是通用的，但我们关注在推测解码调度的背景下特别影响的三个配置参数：

1. 提议长度：草稿模型在每个步骤中提议的令牌数量。
2. 运行请求：在每个步骤中要运行的请求。

4.2 理解有效吞吐量

良好产出，基本上定义为预期收益与成本的比率，提供了有关批量大小和建议长度如何相互作用以优化系统性能的宝贵见解。

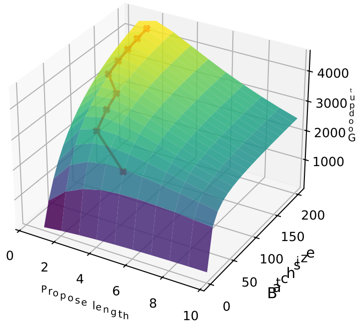


图5. 有效吞吐量与建议长度和批量大小的关系。我们使用A100-7B模型中的系数计算了有效吞吐量，具体细节见第4.3节。我们假设统一的令牌接受率为0.7，并采用第4.4节中描述的公式来估计接受的长度。

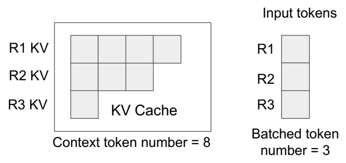


图6. 在建模前向执行时间中使用的上下文令牌数量和批处理令牌数量的示例。

为了展示良效的直觉，图5显示了在假设统一的令牌接受率下，各种批量大小和建议长度下的良效值。

建议小批量时提出更多。在图5中，红线表示每个批量大小的最佳提议长度。值得注意的是，小批量大小需要每个请求提出超过4个令牌才能实现最大吞吐量。随着批量大小的增加，最佳提议长度减少。对于大批量，建议提出更少。对于大批量，完全不进行推测可以导致更高的吞吐量。这是因为随着批量大小的增大，失败推测的成本显著增加，超过了潜在收益。

优先选择批处理而非投机。考虑一个场景，其中令牌的接受是独立的，每个令牌的接受概率为0.7。在这种情况下，接受第一个令牌的概率为0.7，而

probability of accepting both the first and second tokens is $0.7 \times 0.7 = 0.49$. Consequently, increasing the batch size tends to produce more tokens at the same cost. Doubling the batch size results in twice the number of generated tokens, whereas doubling the proposed length does not necessarily yield a proportional increase in output.

Optimizing goodput reduces request latency. Request latency consists of the request's queueing delay and total execution time. When the request rate is low, improving goodput effectively reduces the overall execution time by utilizing speculative decoding with an optimal proposed length. On the other hand, at high request rates, optimizing goodput helps decrease queueing delays by increasing the system's capacity to process requests through large batch sizes and moderate speculative decoding. Overall, strategically adjusting batch sizes and proposed lengths based on goodput enables managing both high and low demand scenarios effectively.

4.3 Modeling Batch Execution Time

The batch execution time is defined as the total time required to complete a speculative decoding step, incorporating both the draft and target model execution times. This can be mathematically represented as:

$$T_{batch} = T_{draft} + T_{target} \tag{4}$$

Modeling T_{draft} and T_{target}. For draft-model-free speculative decoding, we assign a small constant factor $T_{draft} = C$. For draft model-based speculative decoding, the draft model operates in an autoregressive manner and supports continuous batching. Consequently, the total execution time for the draft model is the aggregate of the execution times across all draft steps. Each step involves a single forward pass of the draft model. Given the variability in propose lengths across different requests, the execution time per step may differ. The draft model execution time is a sum of the execution time of each forward pass:

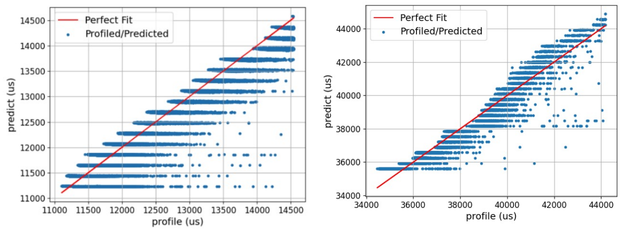
$$T_{draft} = \sum_{i=1}^{s} T_{fwd}(M, N_{context}(s), N_{batched}(s)) \tag{5}$$

Here, s the number of autogressive steps. Concretely, $s = max(s_1, s_2, ...s_n)$, where s_i is the proposed length of request i in the batch. The forward execution time, T_{fwd}, varies across different steps due to different numbers of context tokens ($N_{context}$) and batched tokens ($N_{batched}$) at each step. It also depends on the model M the forward pass is running on. These variations directly influence the duration of the forward execution time, as outlined in **Modeling T_{fwd}.** below.

The target model executes a single forward pass for its operation:

$$T_{target} = T_{fwd}(N_{context}, N_{batched}) \tag{6}$$

Modeling T_{fwd}. We then define T_{fwd}, the time for a forward pass, which is applicable to both the draft and target models:



(a) 7B, TP=1. **(b)** 70B, TP=4.

Figure 7. Predicted versus profiled batch latency. TP: tensor parallel. The x-axis represents the profiled time, while the y-axis shows the predicted execution time using Formula 7. The red line symbolizes perfect prediction, indicating where the predicted time matches the profiled time exactly.

$$T_{fwd}(M, N_{context}, N_{batched}) = \alpha_M \cdot N_{context} + \gamma_M \cdot N_{batched} + \delta_M \tag{7}$$

where $N_{context}$ represents the number of context tokens within the batch and $N_{batched}$ denotes the number of batched tokens, as illustrated in Figure 7. The term $\alpha_M \cdot N_{context}$ reflects the time required to load the key-value cache, scaling linearly with the number of context tokens. The term $\gamma_M \cdot N_{batched}$ accounts for the computation time, while δ_M represents the time to load the model parameters. The coefficients α_M, γ_M, and δ_M are contingent upon the model and hardware configuration, necessitating distinct sets of (α_M, γ_M, δ_M) for various models or hardware environments. In practical terms, we systematically profile the batch execution time for each specific model and hardware combination, and subsequently adjust the values of α_M, γ_M, and δ_M to accurately reflect these profiles.

Modeling $N_{batched}$. If at each position the proposal method suggests only one token (top 1 candidate), the number of tokens sent for verification is simply the sum of the proposed lengths of each request. For top-k tree-style speculative decoding, assuming full tree verification, assume full tree verification, there are H heads and we propose k_i tokens for head i, the number of batched tokens is $\sum_{h=1}^{H} \prod_{i=1}^{h} k_i$ [4]. Figure 8 illustrates an example of the number of batched tokens in Medusa. As shown, one characteristic of full tree speculative decoding is its high cost. In the example, even if a maximum of four tokens can be accepted (three proposed tokens plus one bonus token), a total of 27 tokens are sent for verification. This can be problematic for large batch sizes. Smarter methods to construct the tree, such as those proposed by [6] and SmartSpec, are needed to make topk candidate speculative applicable in a real serving system.

Validating performance model. Figure 7 illustrates the application of our T_{fwd} function, designed to estimate batch

接受第一个和第二个标记的概率是 $0.7 \times 0.7 = 0.49$。因此，增加批量大小往往会以相同的成本产生更多的标记。将批量大小加倍会导致生成的标记数量翻倍，而将提议的长度加倍并不一定会导致输出的成比例增加。

优化有效吞吐量可以减少请求延迟。请求延迟由请求的排队延迟和总执行时间组成。当请求速率较低时，通过使用具有最佳建议长度的推测解码来提高有效吞吐量，可以有效减少整体执行时间。另一方面，在高请求速率下，优化有效吞吐量有助于通过增加系统处理请求的能力（通过大批量和适度的推测解码）来减少排队延迟。总体而言，根据有效吞吐量战略性地调整批量大小和建议长度，可以有效管理高需求和低需求场景。

4.3 批处理执行时间建模

批处理执行时间被定义为完成一个推测解码步骤所需的总时间，包括草稿模型和目标模型的执行时间。这可以用数学表示为：

$$T_{batch} = T_{draft} + T_{target} \tag{4}$$

建模 T_{draft} 和 T_{target}。对于草稿模型无关的推测解码，我们分配一个小的常数因子 $T_{draft} = C$。对于草稿模型相关的推测解码，草稿模型以自回归方式运行，并支持连续批处理。因此，草稿模型的总执行时间是所有草稿步骤的执行时间的总和。每个步骤涉及草稿模型的单次前向传递。考虑到不同请求中提议长度的变化，每个步骤的执行时间可能会有所不同。草稿模型的执行时间是每次前向传递的执行时间之和：

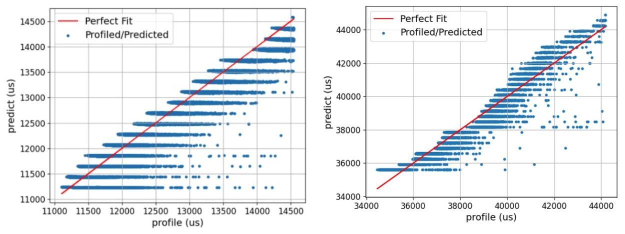
$$T_{draft} = \sum_{i=1}^{s} T_{fwd}(M, N_{context}(s), N_{batched}(s)) \tag{5}$$

在这里，s 是自回归步骤的数量。具体来说，$s = max(s_1, s_2, ...s_n)$，其中 s_i 是批次中请求 i 的建议长度。前向执行时间 T_{fwd} 在不同步骤之间变化，因为每个步骤的上下文令牌数量 ($N_{context}$) 和批处理令牌 ($N_{batched}$) 不同。它还取决于模型 M 前向传递运行的情况。这些变化直接影响前向执行时间的持续时间，如建模 T_{fwd} 中所述。

目标模型执行一次前向传播以进行操作：

$$T_{target} = T_{fwd}(N_{context}, N_{batched}) \tag{6}$$

建模 T_{fwd}。然后我们定义 T_{fwd}，这是前传的时间，适用于草稿和目标。



(a) 7B, TP=1. **(b)** 70B, TP=4.

图7. 预测与剖析的批处理延迟。TP：张量并行。x轴表示剖析时间，而y轴显示使用公式7的预测执行时间。红线象征完美预测，表示预测时间与剖析时间完全匹配的地方。

模型：

$$T_{fwd}(M, N_{context}, N_{batched}) = \alpha_M \cdot N_{context} + \gamma_M \cdot N_{batched} + \delta_M \tag{7}$$

其中 $N_{context}$ 代表批次内的上下文令牌数量，$N_{batched}$ 表示批处理令牌的数量，如图 7 所示。术语 $\alpha_M \cdot N_{context}$ 反映了加载键值缓存所需的时间，随着上下文令牌数量的增加而线性增长。术语 $\gamma_M \cdot N_{batched}$ 考虑了计算时间，而 δ_M 代表加载模型参数的时间。系数 α_M、γ_M 和 δ_M 取决于模型和硬件配置，因此对于不同的模型或硬件环境需要不同的 (α_M、γ_M、δ_M) 集合。在实际操作中，我们系统地分析每个特定模型和硬件组合的批处理执行时间，并随后调整 α_M、γ_M 和 δ_M 的值，以准确反映这些分析结果。

建模 $N_{batched}$。如果在每个位置提议方法仅建议一个标记（前 1 个候选），则发送进行验证的标记数量仅仅是每个请求提议长度的总和。对于 top-k 树状推测解码，假设完全树验证，假设完全树验证，有 H 个头，我们为头 i 提议 k_i 个标记，批处理标记的数量为 $\sum_{h=1}^{H} \prod_{i=1}^{h} k_i$ [4]。图 8 说明了 Medusa 中批处理标记数量的一个示例。如图所示，完全树推测解码的一个特征是其高成本。在这个例子中，即使最多可以接受四个标记（三个提议的标记加一个额外标记），总共发送了 27 个标记进行验证。这对于大批量大小可能是个问题。需要更智能的方法来构建树，例如 [6] 和 SmartSpec 提出的那些方法，以使 top-k 候选推测在实际服务系统中适用。

验证性能模型。图7展示了我们T_{fwd}函数的应用，该函数旨在估计批量。

execution times. This is demonstrated under a uniform workload condition, where each request maintains identical input/output sizes (input length = output length = 128). Furthermore, we adjust the request rates to evaluate the function's performance across a spectrum of batch sizes, with each data point representing an execution step. Our analysis encompasses comparisons between models of 7B and 70B parameters, employing tensor parallelism settings of 1 and 4, respectively. Overall, the results demonstrate that our model accurately captures the trends present in the observed data, effectively adapting to variations in request rates, model sizes, and levels of parallelism.

4.4 Modeling Generated Length

To accurately predict goodput, as defined in Equation 3, our methodology necessitates modeling the number of tokens produced during each generation step. The capability to accurately predict the length of generated content is crucial for minimizing computational waste and enhancing the efficiency of scheduling, as the predictions directly influence the determination of the proposed length for generation and the subsequent scheduling decisions. SmartSpec explores three methods to model the accepted length.

Top1 candidate generated length. SmartSpec employs a moving average method to estimate the token acceptance rate for specified pairs of draft and target on a given dataset. Concretely, SmartSpec records the token acceptance rate from previous generation steps. For predicting the rate in the current step, it calculates the average from these past rates. The moving average method used requires a window size; however, we find that the performance is relatively insensitive to the choice of this window size. This approach presupposes uniform token acceptance behavior across diverse requests. The acceptance length is predicted using the formula introduced in the original speculative decoding paper [20].

$$l(\alpha, k) = \frac{1 - \alpha^{k+1}}{1 - \alpha} \tag{8}$$

In this context, l represents the generated length for each request, inclusive of the bonus token. The variable α denotes the average token acceptance rate observed within the calibrated dataset, while k corresponds to the number of tokens proposed. We can then write out the total number of generated tokens in a single batch:

$$generated\ tokens = \sum_{i \in R} \frac{1 - \alpha_i^{k_i+1}}{1 - \alpha_i} \tag{9}$$

Here, we can have different granularity of token acceptance rate. (1) Global token acceptance rate: it is assumed that each request exhibits identical acceptance behavior. Consequently, different requests within the same batch share the same token acceptance rate and proposed length, $k_1 = k_2 = ... = k_n$, $\alpha_1 = \alpha_2 = ... = \alpha_n$. (2) Request level token acceptance rate: individual requests exhibit distinct acceptance rates (αs) and proposed lengths (ks) due to varying levels of difficulty, necessitating the proposal of a differing number of tokens for each.

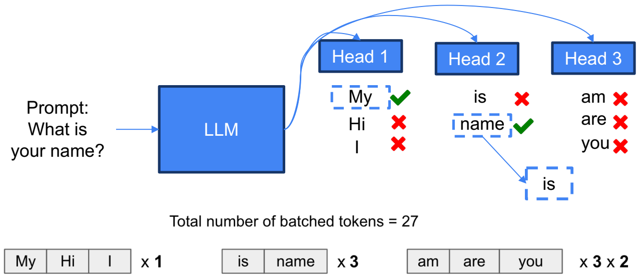


Figure 8. An example Medusa-style speculative decoding utilizing three distinct heads. Each head is tasked with generating proposals for one position. In this scenario, the first head proposes the top three most likely tokens, the second head selects the top two, and the third head also chooses the top three. During this forward run, three tokens ['My', 'name', 'is'] are generated – two from the proposals plus one bonus token. Collectively, this setup represents a total of 18 (3x2x3) possible continuations.

Topk tree-style generated length. In tree-style speculative decoding, we can have multiple candidates for the same position. In Medusa, each head is responsible for proposing for a position. Figure 8 shows an example, where there are three candidates for position 1, two candidates for position 2, and three candidates for position 3.

To estimate the accepted length, we make the following assumptions: 1. The token acceptance rate for each proposed token at head i is denoted as α_i. All tokens within the top-k share the same token acceptance rate. 2. The acceptance behavior is independent across different heads; that is, the acceptance of a token at head i does not influence the acceptance of a token at head $i + 1$. Suppose there are h heads, and we propose $k_1, k_2, \ldots, k_h$ tokens for heads $1, 2, \ldots, h$, respectively. The token acceptance rates for these heads are $\alpha_1, \alpha_2, \ldots, \alpha_h$. For simplicity in the formula below, we define $\alpha_{h+1} = 0$. The expected accepted length given the structure can be formulated as:

$$l(\alpha_1 \ldots \alpha_h, k_1 \ldots k_h) = \sum_{i=1..h} (i+1) \times (1-\alpha_{i+1}) \times \Pi_{j=1 \cdots i} [1-(1-\alpha_j)^{k_j}] \tag{10}$$

Here, $(1 - \alpha_{i+1}) \times \prod_{j=1}^{i} [1 - (1 - \alpha_j)^{k_j}]$ represents the probability of accepting $(i + 1)$ tokens, where the "+1" accounts for the bonus token. To understand this probability, it hinges on two conditions: (1) At least one token is accepted from heads 1 through i. (2) None of the proposed tokens at head $i + 1$ are accepted. Thus, the probability is calculated as the product of the probabilities of conditions (1) and (2).

Since SmartSpec is a versatile speculative decoding framework, users can integrate various estimation methods, such

执行时间。这在均匀工作负载条件下得以证明，其中每个请求保持相同的输入/输出大小（输入长度 = 输出长度 = 128）。此外，我们调整请求速率以评估该函数在一系列批量大小上的性能，每个数据点代表一个执行步骤。我们的分析包括对7B和70B参数模型的比较，分别采用1和4的张量并行设置。总体而言，结果表明我们的模型准确捕捉了观察数据中存在的趋势，有效适应了请求速率、模型大小和并行级别的变化。

4.4 生成长度建模

为了准确预测良好输出，如方程3所定义，我们的方法论需要对每个生成步骤中产生的令牌数量进行建模。准确预测生成内容的长度对于最小化计算浪费和提高调度效率至关重要，因为预测直接影响生成的建议长度的确定以及随后的调度决策。SmartSpec 探索了三种建模接受长度的方法。

Top1候选生成长度。SmartSpec采用移动平均法来估计给定数据集上指定草稿和目标对的令牌接受率。具体而言，SmartSpec记录了之前生成步骤中的令牌接受率。为了预测当前步骤中的接受率，它计算这些过去接受率的平均值。所使用的移动平均法需要一个窗口大小；然而，我们发现性能对这个窗口大小的选择相对不敏感。这种方法假设在不同请求中令牌接受行为是均匀的。接受长度是使用原始推测解码论文中引入的公式{v*}进行预测的。

$$l(\alpha, k) = \frac{1-\alpha^{k+1}}{1-\alpha} \tag{8}$$

在这种情况下，l 代表每个请求生成的长度，包括奖励令牌。变量 α 表示在校准数据集中观察到的平均令牌接受率，而 k 对应于提议的令牌数量。我们可以写出单个批次中生成的令牌总数：

$$generated\ tokens = \sum_{i \in R} \frac{1-\alpha_i^{k_i+1}}{1-\alpha_i} \tag{9}$$

在这里，我们可以有不同粒度的令牌接受率。(1) 全局令牌接受率：假设每个请求表现出相同的接受行为。因此，同一批次中的不同请求共享相同的令牌接受率和提议长度，$k_1 = k_2 = ... = k_n$，$\alpha_1 = \alpha_2 = ... = \alpha_n$。(2) 请求级别令牌接受率：单个请求表现出不同的接受率（αs）

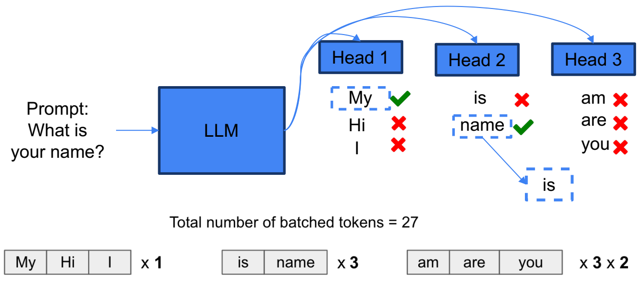


图8. 一个示例的美杜莎风格的推测解码，利用了三个不同的头。每个头负责为一个位置生成提案。在这种情况下，第一个头提出了三个最可能的标记，第二个头选择了两个，第三个头也选择了三个。在这个前向运行中，生成了三个标记 [’My’，’name’，’is’] – 两个来自提案加上一个额外的标记。总体而言，这个设置代表了总共18个 (3x2x3) 可能的延续。

由于难度水平的不同，提出了不同的长度 (ks)，因此需要为每个提议不同数量的标记。

Topk树状生成长度。在树状推测解码中，我们可以为同一位置拥有多个候选项。在美杜莎中，每个头负责为一个位置提出建议。图8展示了一个例子，其中位置1有三个候选项，位置2有两个候选项，位置3有三个候选项。

为了估计接受的长度，我们做出以下假设：1. 每个提议的令牌在头 i 的接受率记作 α_i。所有在前 k 的令牌共享相同的令牌接受率。2. 接受行为在不同的头之间是独立的；也就是说，头 i 中一个令牌的接受不会影响头 $i+$ 中一个令牌的接受。假设有 h 个头，我们分别为头 1, 和 $2, \ldots, h$ 提出 $k_1, k_2, \ldots, k_h$ 个令牌。这些头的令牌接受率为 $\alpha_1, \alpha_2, \ldots, \alpha_h$。为了简化下面的公式，我们定义 α_{h+1} = 为 0。给定结构的预期接受长度可以表示为：

$$l(\alpha_1 \ldots \alpha_h, k_1 \ldots k_h) = \sum_{i=1..h} (i+1) \times (1-\alpha_{i+1}) \times \Pi_{j=1 \cdots i} [1-(1-\alpha_j)^{k_j}] \tag{10}$$

在这里，$(1-\alpha_{i+1}) \times \prod_{j=1}^{i} [1-(1-\alpha_j)^{k_j}]$ 代表接受 $(i+1)$ 代币的概率，其中 "+1" 代表额外代币。要理解这个概率，它依赖于两个条件：(1) 从头部 1 到 i 至少接受一个代币。(2) 在头部 $i+1$ 中没有接受任何提议的代币。因此，概率计算为条件 (1) 和 (2) 概率的乘积。

由于 SmartSpec 是一个多功能的推测解码框架，用户可以集成各种估计方法，例如 {v*}。

as the confidence-based method discussed in the Appendix. Additionally, users may even employ machine learning models for predictions. We leave it as future work to develop an accurate and efficient predictor for accepted length.

5 Serving Requests Using SmartSpec

We now describe the flow of a single decoding step outlined in Algorithm 2. Initially, we enumerate potential requests for a single batch (line 2). SmartSpec uses a first-come, first-served strategy. Assuming n requests in the pending batch, we construct candidate batches by selecting prefixes of increasing length: batches with 1 request, 2 requests, etc., up to n requests. For each potential batch, we use goodput to determine the optimal proposed length (line 4). Additionally, we verify if there is sufficient space in the KV cache (lines 5-7). For Top-1 candidate speculative decoding, SmartSpec will determine the optimal proposed length. For Top-k tree-style speculative decoding, SmartSpec will identify the optimal Top-k value for each proposal head. In both cases, the token acceptance rate is factored into Eqn. 8 to calculate the estimated generation length. SmartSpec then uses this estimated length along with Eqn. 3 and a performance model (elaborated in Sec. 4.3) to estimate goodput. After identifying the optimal proposed lengths or Top-k value, SmartSpec executes these steps sequentially. For draft-model based speculative decoding, the proposal phase operates in an autoregressive manner and incorporates continuous batching (line 12). Then SmartSpec verifies the proposed tokens and records the token acceptance rate of current step (line 13).

To estimate the current token acceptance rate, SmartSpec records the acceptance rate from previous scoring steps (line 13 in Alg. 2) and computes the moving average of these rates (lines 5-8 in Alg. 1). Although moving average is an imperfect estimator for token acceptance rate, goodput remains robust and results in latency reduction to be discussed in Sec. 7.2.1.

Prefill disabling. In draft-model based speculative decoding, the prefill phase of the running draft model can introduce overhead, especially when request rate is high. By default, speculative decoding is disabled during the prefill phase. However, to synchronize the KV cache between the draft and target models, the system still executes the draft model's prefill phase by default, even if no tokens are subsequently proposed by SmartSpec. This can lead to the wasteful consumption of memory and computational resources. To address this issue, we use a feedback-based method that automatically disables the draft model's prefill run. For each generation step, SmartSpec records the proposed length. During each prefill phase, SmartSpec checks the proposed length from previous decoding steps. If the percentage of times no tokens were proposed exceeds a predefined threshold, SmartSpec will disable the draft model's prefill run for the current request and classify the request as non-speculative. Since the draft model's KV cache is not maintained for these requests, speculative decoding is also disabled for all subsequent decoding steps for these requests. The threshold for disabling the prefill run is adjustable, allowing users to tailor the level of conservative speculative decoding to their needs. Empirically, setting this threshold to 0.7 has yielded good performance, balancing resource efficiency with decoding effectiveness.

Algorithm 1 Goodput Estimation for Step-level Proposed Length

```
Require: Proposed length $k$ for all requests in the batch for Top-1 candidate speculative decoding. Number of sampled tokens $k_1 \dots k_h$ for each head for Top-k tree-style speculative decoding. Estimation method $Method$. Token acceptance rate $prev\_alphas$ of previous steps. All requests in the batch $R$.
1:  $n \leftarrow len(R)$
2:  if $Method$ == Top-1 candidate speculative decoding then
3:      $\alpha \leftarrow MovingAvg(prev\_alphas)$
4:      $generated\_len = n \times \frac{1-\alpha^{k+1}}{(1-\alpha)} \times n$
5:      $batch\_execution\_time = T(R, [k])$
6:  else if $Method$ == Top-k tree-style speculative decoding then
7:      $\alpha_1, \alpha_2 \dots \alpha_h \leftarrow MovingAvg(prev\_alphas)$
8:      $generated\_len = n \times \sum_{i=1..h}(i+1) \times (1-\alpha_{i+1}) \times \Pi_{j=1\cdots i}[1-(1-\alpha_j)^{k_j}]$
9:      $batch\_execution\_time = T(R, [k_1, k_2 \dots k_h])$
10: end if
11: return $\frac{generated\_len}{batch\_execution\_time}$
```

Algorithm 2 SmartSpec token acceptance rate based proposal and verification.

```
Require: Pending requests $R$. Max proposed length $V$. Token acceptance rates of previous decoding steps $prev\_alphas$.
1:  $best\_goodput, best\_proposed\_lens \leftarrow -1, []$
2:  $batch\_candidates \leftarrow$ GetBatchCandidates()
3:  for $batch$ in $batch\_candidates$ do
4:      $cur\_goodput, cur\_proposed\_lens \leftarrow Argmax_{k_1,k_2 \dots k_n}$(Goodput($k_1, k_2 \dots k_n, prev\_alphas$))
5:      if not HasSlot($batch$) then
6:          continue.
7:      end if
8:      if $cur\_goodput > best\_goodput$ then
9:          $best\_goodput, best\_proposed\_lens \leftarrow cur\_goodput, cur\_proposed\_lens$
10:     end if
11: end for
12: Propose($R, best\_proposed\_lens$)
13: $\alpha_{cur}$ = Score($R, best\_proposed\_lens$)
14: $prev\_alphas$.append($\alpha_{cur}$)
```

Discussion and Complexity Analysis. For each batch, the overhead of SmartSpec consists of three components: accepted length estimation, batch execution time modeling, and goodput-guided proposal length optimization. Computing accepted length and batch execution time are $O(1)$, as

如附录中讨论的基于信心的方法。此外，用户甚至可以使用机器学习模型进行预测。我们将开发一个准确且高效的接受长度预测器作为未来的工作。

5个使用SmartSpec的服务请求

我们现在描述算法 2 中概述的单个解码步骤的流程。最初，我们列举单个批次的潜在请求（第 2 行）。SmartSpec 使用先到先服务的策略。假设 n 个请求在待处理批次中，我们通过选择逐渐增加长度的前缀来构建候选批次：包含 1 个请求、2 个请求等，直到 n 个请求。对于每个潜在批次，我们使用良好吞吐量来确定最佳提议长度（第 4 行）。此外，我们验证 KV 缓存中是否有足够的空间（第 5-7 行）。对于 Top-1 候选的投机解码，SmartSpec 将确定最佳提议长度。对于 Top-k 树状投机解码，SmartSpec 将为每个提议头识别最佳 Top-k 值。在这两种情况下，令牌接受率被纳入公式 8 以计算估计的生成长度。然后，SmartSpec 使用这个估计长度以及公式 3 和性能模型（在第 4.3 节中详细说明）来估计良好吞吐量。在确定最佳提议长度或 Top-k 值后，SmartSpec 按顺序执行这些步骤。对于基于草稿模型的投机解码，提议阶段以自回归方式运行，并结合连续批处理（第 12 行）。然后，SmartSpec 验证提议的令牌并记录当前步骤的令牌接受率（第 13 行）。

为了估计当前的令牌接受率，SmartSpec 记录了之前评分步骤的接受率（算法 2 中的第 13 行），并计算这些接受率的移动平均值（算法 1 中的第 5-8 行）。尽管移动平均是令牌接受率的不完美估计器，但良好的吞吐量仍然保持稳健，并导致延迟减少，这将在第 7.2.1 节中讨论。预填禁用。在基于草稿模型的推测解码中，运行草稿模型的预填阶段可能会引入开销，特别是在请求速率较高时。默认情况下，推测解码在预填阶段是禁用的。然而，为了在草稿模型和目标模型之间同步 KV 缓存，即使 SmartSpec 随后没有提出任何令牌，系统仍然默认执行草稿模型的预填阶段。这可能导致内存和计算资源的浪费。为了解决这个问题，我们使用了一种基于反馈的方法，自动禁用草稿模型的预填运行。对于每个生成步骤，SmartSpec 记录提出的长度。在每个预填阶段，SmartSpec 检查之前解码步骤中提出的长度。如果没有提出令牌的次数所占的百分比超过预定义的阈值，SmartSpec 将禁用当前请求的草稿模型预填运行，并将该请求分类为非推测性请求。由于草稿模型的 KV 缓存不会为这些请求维护，

Algorithm 1 Goodput Estimation for Step-level Proposed Length

```
Require: Proposed length $k$ for all requests in the batch for Top-1 candidate speculative decoding. Number of sampled tokens $k_1 \dots k_h$ for each head for Top-k tree-style speculative decoding. Estimation method $Method$. Token acceptance rate $prev\_alphas$ of previous steps. All requests in the batch $R$.
 1: $n \leftarrow len(R)$
 2: if $Method$ == Top-1 candidate speculative decoding then
 3:     $\alpha \leftarrow MovingAvg(prev\_alphas)$
 4:     $generated\_len = n \times \frac{1-\alpha^{k+1}}{(1-\alpha)} \times n$
 5:     $batch\_execution\_time = T(R, [k])$
 6: else if $Method$ == Top-k tree-style speculative decoding then
 7:     $\alpha_1, \alpha_2 \dots \alpha_h \leftarrow MovingAvg(prev\_alphas)$
 8:     $generated\_len = n \times \sum_{i=1..h}(i+1) \times (1-\alpha_{i+1}) \times \Pi_{j=1\cdots i}[1-(1-\alpha_j)^{k_j}]$
 9:     $batch\_execution\_time = T(R, [k_1, k_2 \dots k_h])$
10: end if
11: return $\frac{generated\_len}{batch\_execution\_time}$
```

Algorithm 2 SmartSpec token acceptance rate based proposal and verification.

```
Require: Pending requests $R$. Max proposed length $V$. Token acceptance rates of previous decoding steps $prev\_alphas$.
 1: $best\_goodput, best\_proposed\_lens \leftarrow -1, []$
 2: $batch\_candidates \leftarrow$ GetBatchCandidates()
 3: for $batch$ in $batch\_candidates$ do
 4:     $cur\_goodput, cur\_proposed\_lens \leftarrow Argmax_{k_1,k_2 \dots k_n}$(Goodput($k_1, k_2 \dots k_n, prev\_alphas$))
 5:     if not HasSlot($batch$) then
 6:         continue.
 7:     end if
 8:     if $cur\_goodput > best\_goodput$ then
 9:         $best\_goodput, best\_proposed\_lens \leftarrow cur\_goodput, cur\_proposed\_lens$
10:     end if
11: end for
12: Propose($R, best\_proposed\_lens$)
13: $\alpha_{cur}$ = Score($R, best\_proposed\_lens$)
14: $prev\_alphas$.append($\alpha_{cur}$)
```

请求时，所有后续解码步骤的推测解码也被禁用。禁用预填充运行的阈值是可调的，允许用户根据自己的需求调整保守推测解码的水平。根据经验，将此阈值设置为 0.7 可以获得良好的性能，平衡资源效率与解码效果。

讨论与复杂性分析。对于每个批次，SmartSpec 的开销由三个部分组成：接受长度估计、批次执行时间建模和良率引导的提案长度优化。计算接受长度和批次执行时间是 $O(1)$，如

they use moving average based token acceptance rate and offline profiled model coefficients, as detailed in Secs. 4.4 and 4.3.

The complexity of goodput-guided optimization varies depending on whether it utilizes request-level or global token acceptance rates. In this work, we empirically find that both granularities yield similar performance gains. However, given that the request-level token acceptance rate introduces significant engineering complexity, we opt to use the global token acceptance rate to estimate the accepted length. Since the maximum proposed length V is typically less than 10, we can efficiently enumerate all possible proposed lengths to find the one that maximizes goodput. This is a very small overhead in comparison with LLM forward pass: let s be the sequence length, ℓ be number of decoder layers in the model, h be the hidden dimension size, n be the batch size, each forward pass's complexity is at the order of $O(\ell n(sh^2 + s^2h)) \gg O(nV)$.

6 System Design and Architecture

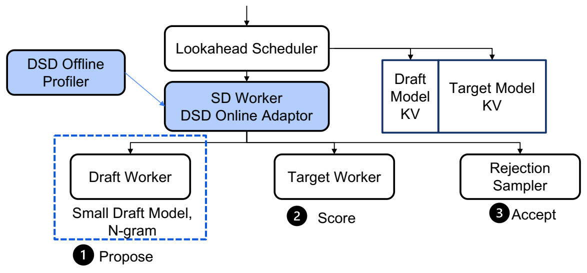


Figure 9. System architecture of SmartSpec in vLLM.

We implement SmartSpec within vllm [19] and illustrate the system architecture in Figure 9. Initially, upon receiving a request, the lookahead scheduler takes charge. This scheduler is tasked with dispatching requests for immediate execution and managing resource allocation within the key-value (KV) cache. It is termed a "lookahead" scheduler because it proactively reserves multiple KV cache spots for tokens that have yet to be generated. Subsequently, the engine forwards these active requests to the speculative decoding worker, which, in turn, activates the draft worker. The draft worker operates the draft model through several steps, each generating a proposed token. It offers a standardized interface that supports various speculative decoding approaches, such as a small draft model or an n-gram model. Subsequently, the target worker utilizes the target model for scoring and verification purposes. Note here both the draft and target models interact with the KV cache during their forward.

For effective memory management within speculative decoding, it is essential to maintain the key-value (KV) cache for both the draft and target workers. In scenarios where draft-model-based speculative decoding is employed, we divide the total memory allocated for the KV cache into two distinct segments: one dedicated to the draft model and the other to the target model. Our observations have consistently shown that the number of required slots for both the draft and target models remains constant both before and after the execution step. Consequently, we allocate the KV cache to provide an equal number of slots for each model, with the scheduler assigning the same number of slots to both models. This approach not only simplifies the system's architecture but also reduces its complexity. On the other hand, when carrying out draft-model free speculative decoding, SmartSpec maintains the KV cache as it without speculative decoding.

To dynamically adjust the proposed length, we employ the online adaptor in conjunction with the offline profiler. The offline profiler is responsible for analyzing both the draft (if any) and target models to derive the performance coefficients critical for the performance model, as detailed in Section 4.3. These coefficients are subsequently utilized by the online adaptor, which aggregates batch information. Based on this data, the online adaptor adjusts the proposal length for the draft worker and the verification length for the target model.

7 Evaluation

Model and server configurations. We test on two popular open source model families: Llama and Mistral. For Llama, we use Vicuna-7B-v1.5, Vicuna-33B-v1.3 [44], and Llama-2-70b-chat-hf [36]. For Mistral, we use Mistral-7B-Instruct-v0.1 [13] and Mixtral-8x7B-Instruct-v0.1 [14]. We use a single A100-80G [28] GPU for the 7B model, 4×A100-80G GPUs for the 33B model, and 8×A100-80G GPUs for the 70B model. For the draft model-based method, the draft model operates with tensor parallelism set to 1, and only the target model is sharded across multiple GPUs. We provide detailed specifications of the setup in Table 1.

Evaluated methods and baselines. We test the efficiency of SmartSpec on both standard speculative decoding, where a draft model is used to make the proposal, and prompt lookup decoding, where the proposal is made by looking up ngrams in the prompt. Both mechanisms are detailed in Sec. 2.1. For standard speculative decoding, we fine-tune Llama-160M on the shareGPT dataset to improve its general proposal capability and use it as the draft model for Vicuna-7B. For Llama2-70B model, we use Vicuna-7B as the draft. For all the settings, the draft model shares the same tensor parallel degree as the target model.

We compared SmartSpec against two baselines: vanilla auto-regressive inference, which does not incorporate speculative decoding, and using speculative decoding with a fixed proposed length of 1, 3, and 5 across all execution steps.

Workloads. In our study, we focus on four types of workloads, online chatting, text-to-SQL, summarization, and question answering given context. For online chatting, we utilize datasets from ShareGPT [2], Chatbot Arena [44]. For text-to-SQL, we use the spider [42] dataset. For summarization and

他们使用基于移动平均的代币接受率和离线配置的模型系数，详细信息见第4.4节和第4.3节。

良好产出引导优化的复杂性取决于它是使用请求级别还是全局令牌接受率。在这项工作中，我们实证发现这两种粒度都能带来类似的性能提升。然而，考虑到请求级别的令牌接受率引入了显著的工程复杂性，我们选择使用全局令牌接受率来估计接受的长度。由于最大提议长度 V 通常小于 10，我们可以高效地枚举所有可能的提议长度，以找到最大化良好产出的长度。这与 LLM 前向传播相比是一个非常小的开销：设 s 为序列长度，ℓ 为模型中的解码器层数，h 为隐藏维度大小，n 为批量大小，每次前向传播的复杂性大约为 $O(\ell n(sh^2+s^2h)) \gg O(nV)$。

6 系统设计与架构

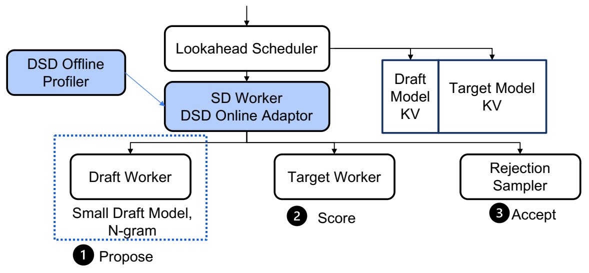


图9. SmartSpec在vLLM中的系统架构。

我们在 vllm [19] 中实现了 SmartSpec，并在图 9 中展示了系统架构。最初，在接收到请求后，预调度器负责处理。这个调度器的任务是调度立即执行的请求并管理键值 (KV) 缓存中的资源分配。它被称为“预调度”调度器，因为它主动为尚未生成的令牌预留多个 KV 缓存位置。随后，引擎将这些活跃请求转发给推测解码工作者，后者又激活草稿工作者。草稿工作者通过几个步骤操作草稿模型，每个步骤生成一个建议的令牌。它提供了一个标准化接口，支持各种推测解码方法，例如小型草稿模型或 n-gram 模型。随后，目标工作者利用目标模型进行评分和验证。在此需要注意的是，草稿模型和目标模型在其前向过程中都与 KV 缓存进行交互。

在推测解码中有效的内存管理，维护草稿和目标工作者的键值（KV）缓存是至关重要的。在采用基于草稿模型的推测解码的情况下，我们将分配给KV缓存的总内存分为两部分。不同的段落：一个专门用于草稿模型，另一个用于目标模型。我们的观察始终表明，草稿模型和目标模型所需的槽位数量在执行步骤之前和之后保持不变。因此，我们分配KV缓存，以为每个模型提供相同数量的槽位，调度程序为两个模型分配相同数量的槽位。这种方法不仅简化了系统的架构，还降低了其复杂性。另一方面，在进行草稿模型的自由推测解码时，SmartSpec保持KV缓存不变，和没有推测解码时一样。

为了动态调整提议的长度，我们结合使用在线适配器和离线分析器。离线分析器负责分析草稿（如果有的话）和目标模型，以得出对性能模型至关重要的性能系数，如第4.3节所述。这些系数随后被在线适配器利用，在线适配器汇总批量信息。基于这些数据，在线适配器调整草稿工作者的提议长度和目标模型的验证长度。

7 评估

模型和服务器配置。我们在两个流行的开源模型系列上进行测试：Llama 和 Mistral。对于 Llama，我们使用 Vicuna-7B-v1.5、Vicuna-33B-v1.3 [44] 和 Llama-2-70b-chat-hf [36]。对于 Mistral，我们使用 Mistral-7B-Instruct-v0.1 [13] 和 Mixtral-8x7B-Instruct-v0.1 [14]。我们为 7B 模型使用一台 A100-80G [28] GPU，为 33B 模型使用 4×A100-80G GPU，为 70B 模型使用 8×A100-80G GPU。对于基于草稿模型的方法，草稿模型的张量并行度设置为 1，只有目标模型在多个 GPU 之间进行分片。我们在表 1 中提供了详细的设置规格。

评估的方法和基准。我们测试了SmartSpec在标准的推测解码和提示查找解码上的效率，其中草稿模型用于提出建议，提示查找解码则是通过查找提示中的n-grams来提出建议。这两种机制在第2.1节中有详细说明。对于标准的推测解码，我们在shareGPT数据集上微调Llama-160M，以提高其一般提案能力，并将其用作Vicuna-7B的草稿模型。对于Llama2-70B模型，我们使用Vicuna-7B作为草稿。在所有设置中，草稿模型与目标模型共享相同的张量并行度。

我们将 SmartSpec 与两个基线进行了比较：普通的自回归推理，它不包含推测解码，以及在所有执行步骤中使用固定提议长度为 1、3 和 5 的推测解码。

工作负载。在我们的研究中，我们关注四种类型的工作负载：在线聊天、文本到SQL、摘要和基于上下文的问题回答。对于在线聊天，我们利用来自ShareGPT [2]和Chatbot Arena [44]的数据集。对于文本到SQL，我们使用spider [42]数据集。对于摘要和

Task	Dataset	SD Method	Draft Model (TP)	Target Model (TP)	Hardware (Total Mem.)
Online Chatting	Arena[44]	VSD[20]	Llama-160M(1)	Vicuna-7B(1)	A100 (80G)
			Llama-160M(1)	Vicuna-33B(4)	4×A100 (320G)
			TinyLlama-1.1B (1)	Llama2-70B (8)	8×A100 (640G)
	ShareGPT[2]		Llama-160M(1)	Vicuna-7B(1)	A100 (80G)
			Llama-160M(1)	Vicuna-33B(4)	4×A100 (320G)
			TinyLlama-1.1B (1)	Llama2-70B (8)	8×A100 (640G)
Tex-to-SQL	Spider[42]		Llama-160M(1)	Vicuna-7B(1)	A100 (80G)
			Llama-160M(1)	Vicuna-33B(4)	4×A100 (320G)
Summarization	CNN/Daily Mail[12, 34]	PTL[33]	None	Mistral-7B (1)	A100 (80G)
				Mixture-8×7B (8)	8×A100 (640G)
Context QA	HAGRID[16]			Mistral-7B (1)	A100 (80G)
				Mixture-8×7B(8)	8×A100 (640G)

Table 1. Dataset, model and server configuration. VSD: Vanilla speculative decoding. PTL: Prompt lookup decoding.

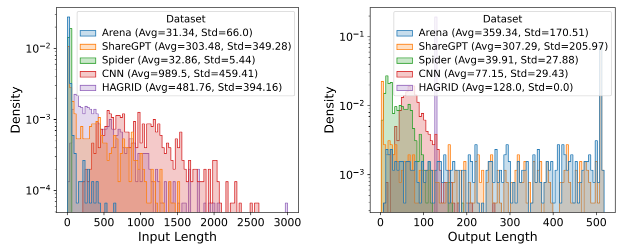


Figure 10. Input/Output length distributions.

question answering, we use the original dataset CNN/Daily Mail [34] and HAGRID [16] from the prompt lookup decoding work. We show the workload characteristics (average input length and average output length) in Fig. 10. For online chatting, the input is of medium length while the output length is longer and show higher variance. For summarization and question answering, the input length is long and the output is short. For the question answering, since we do not have the ground truth in the dataset, we set a fixed output length 128. For all workloads, we generate request arrival times using Poisson distribution with different request rates. We use greedy sampling for all served requests.

7.1 Latency Measurement

We first evaluate the effectiveness of SmartSpec in reducing request latency for both draft model-based and draft model-free speculative decoding.

7.1.1 Draft-model based speculative decoding. We evaluated the average request latency across different datasets, focusing on the performance of SmartSpec in comparison to baseline strategies in Figs. 11 and 12. In environments with a low request rate, SmartSpec demonstrates performance similar to that from greedily predicting a higher number of tokens (e.g., $k = 3$ or 5) to maximize speedup. This similarity suggests that SmartSpec predicts just enough tokens under light loads to effectively reduce request latency through speculative decoding. However, on a few occasions when request rates are low, there are slight performance differences between SmartSpec and standard SD. In those cases, standard SD, which predicts a higher number of tokens, marginally outperforms SmartSpec with a more pronounced speedup. This discrepancy can be attributed to our imperfect acceptance length predictor, resulting in standard SD with a longer proposed length performing better as more tokens could potentially be accepted.

Conversely, in scenarios with a high request rate, the system's performance mirrors situations where no tokens are predicted, effectively indicating that speculative decoding is disabled under conditions of high request volume. It is important to note that in this regime, the latency associated with using standard SD and proposing high numbers of tokens escalates rapidly, leading to significant performance degradation. This is exemplified in Fig. 11 (c) when the request rate is at 32 and (f) when the request rate exceeds 5.

In cases such as Fig. 11 (d) and (e), the relative speedup for these baselines rebounds after initially dropping when the request rate exceeds 2. This rebound occurs because, as the request rate continues to increase, it reaches a point where even baseline decoding without speculation begins to suffer from queuing delays. This phenomenon relatively improves the speedup of standard SD baselines, despite them experiencing higher overall latencies.

In general, speculative decoding is most effective when the system has sufficient computational resources: the larger the model, the smaller the request rate region where we see speedup from speculation. This is expected as speculative decoding requires additional computational power; when the model is large and computational resources do not scale proportionally, the system is likely to be compute-bound. This underscores the importance of SmartSpec. It is crucial for SmartSpec to consistently match or outperform the established baseline across different scenarios. This characteristic is vital for implementing speculative decoding techniques in real production environments, as it ensures that adopting

Task	Dataset	SD Method	Draft Model (TP)	Target Model (TP)	Hardware (Total Mem.)
Online Chatting	Arena[44]	VSD[20]	Llama-160M(1)	Vicuna-7B(1)	A100 (80G)
			Llama-160M(1)	Vicuna-33B(4)	4×A100 (320G)
			TinyLlama-1.1B (1)	Llama2-70B (8)	8×A100 (640G)
	ShareGPT[2]		Llama-160M(1)	Vicuna-7B(1)	A100 (80G)
			Llama-160M(1)	Vicuna-33B(4)	4×A100 (320G)
			TinyLlama-1.1B (1)	Llama2-70B (8)	8×A100 (640G)
Tex-to-SQL	Spider[42]		Llama-160M(1)	Vicuna-7B(1)	A100 (80G)
			Llama-160M(1)	Vicuna-33B(4)	4×A100 (320G)
Summarization	CNN/Daily Mail[12, 34]	PTL[33]	None	Mistral-7B (1)	A100 (80G)
				Mixture-8×7B (8)	8×A100 (640G)
Context QA	HAGRID[16]			Mistral-7B (1)	A100 (80G)
				Mixture-8×7B(8)	8×A100 (640G)

表1. 数据集，模型　　和服务器配置。VSD：香草推测解码。　　　　　　　PTL：提示查找解码。

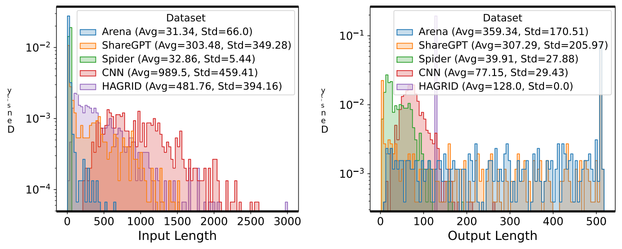


图10. 输入/输出长度分布。

在问答中，我们使用原始数据集CNN/Daily Mail [34]和HAGRID [16]，这些数据集来自提示查找解码工作。我们在图10中展示了工作负载特征（平均输入长度和平均输出长度）。对于在线聊天，输入长度适中，而输出长度较长且方差较高。对于摘要和问答，输入长度较长而输出较短。对于问答，由于我们在数据集中没有真实值，我们设置了固定的输出长度128。对于所有工作负载，我们使用不同请求速率的泊松分布生成请求到达时间。我们对所有服务请求使用贪婪采样。

7.1 延迟测量

我们首先评估 SmartSpec 在减少草稿模型基础和草稿模型无关的推测解码请求延迟方面的有效性。

7.1.1 基于草稿模型的推测解码。我们评估了不同数据集的平均请求延迟，重点比较了 SmartSpec 与基线策略在图 11 和图 12 中的表现。在请求率较低的环境中，SmartSpec 的表现与贪婪地预测更多的标记（例如，$k = 3$ 或 5）以最大化加速的表现相似。这种相似性表明，SmartSpec 在轻负载下预测的标记数量刚好足够，通过推测解码有效减少请求延迟。然而，在少数情况下，当请求费率较低时，SmartSpec和标准SD之间存在轻微的性能差异。在这些情况下，标准SD预测的令牌数量较高，略微优于SmartSpec，且加速效果更为明显。这种差异可以归因于我们不完美的接受长度预测器，导致标准SD在提议的长度较长时表现更好，因为可以潜在地接受更多的令牌。

相反，在请求率高的场景中，系统的性能反映了没有预测到令牌的情况，实际上表明在高请求量的条件下，推测解码被禁用。值得注意的是，在这种情况下，使用标准SD和提出大量令牌所带来的延迟迅速上升，导致显著的性能下降。这在图11 (c)中得到了体现，当请求率为32时，以及在图11 (f)中，当请求率超过5时。

在图11 (d) 和 (e) 的情况下，当请求速率超过2时，这些基线的相对加速在最初下降后反弹。这种反弹发生是因为，随着请求速率的持续增加，它达到一个点，即使是没有推测的基线解码也开始遭受排队延迟。这种现象相对改善了标准SD基线的加速，尽管它们经历了更高的整体延迟。

一般来说，当系统具有足够的计算资源时，推测解码最为有效：模型越大，我们看到的从推测中获得加速的请求速率区域就越小。这是可以预期的，因为推测解码需要额外的计算能力；当模型很大且计算资源不成比例地扩展时，系统很可能会受到计算限制。这突显了SmartSpec的重要性。SmartSpec在不同场景中始终与既定基准相匹配或超越的能力至关重要。这一特性对于在真实生产环境中实施推测解码技术至关重要，因为它确保了采用{v*}。

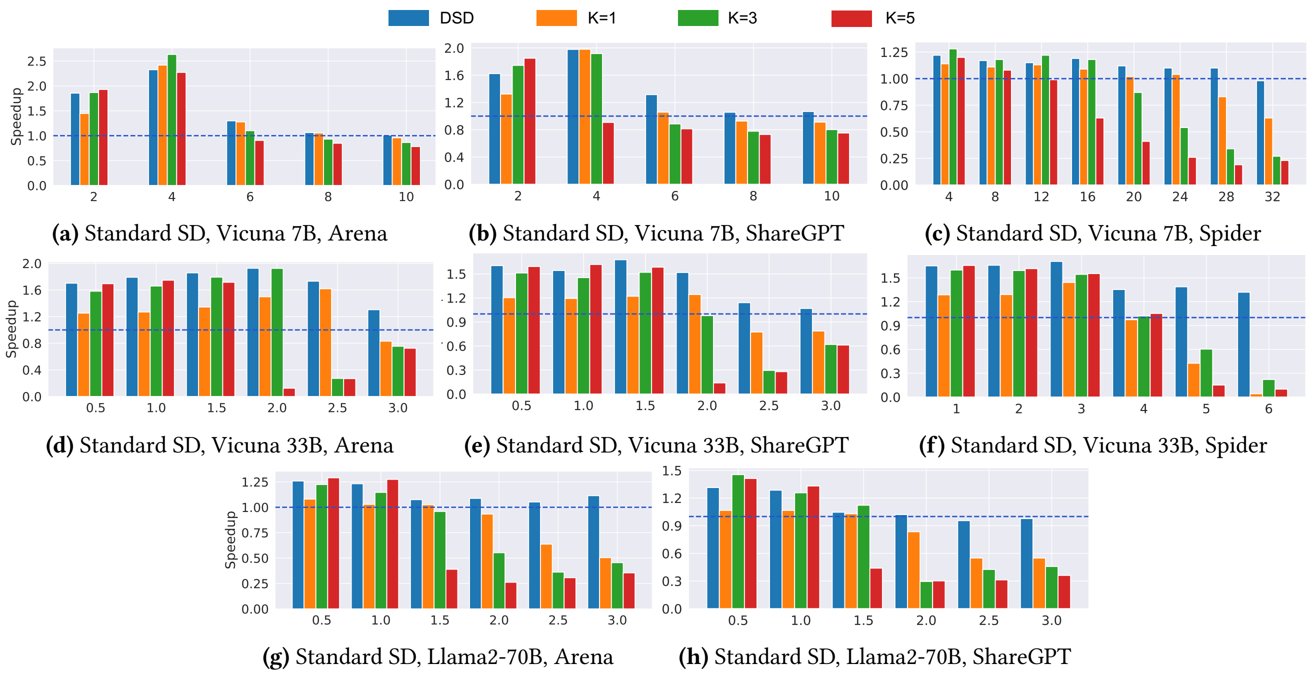


Figure 11. Latency Measurement on standard SD: Speedup across different datasets. X-axis: request rate.

speculative decoding will not compromise system performance.

7.1.2 Draft model free speculative decoding. We next evaluate the efficiency of SmartSpec with prompt lookup decoding. Like our tests with draft-model-based speculative decoding, we compare SmartSpec using fixed proposal lengths of 1, 3, and 5, against the baseline of no speculative decoding. As shown in Fig. 12, SmartSpec consistently achieves the best speedup. Notably, prompt lookup decoding with Mistral-7B (Fig. 12 (a) and (b)) shows substantial speedup even with a relatively low token acceptance rate (the measured token acceptance rate on those two settings is between 0.3 and 0.4). Unlike scenarios involving a draft model, prompt lookup does not incur significant overhead for proposing tokens, leading to notable speed gains even with lower speculative accuracy.

Like draft model-based speculative decoding, SmartSpec does not compromise performance even when the request rate is high. This is because SmartSpec adopts a conservative approach under high request rates, minimizing the wasteful computation of verifying incorrect tokens. This strategy ensures that SmartSpec maintains its efficiency across varying system loads.

7.2 Simulation Experiments

We conduct the following experiments using a simulator for several reasons. First, efficiently integrating speculative decoding into a real-world system poses a substantial engineering challenge. For instance, devising an effective verification kernel for tree-structured speculative decoding proves difficult. The absence of such a kernel negates the advantages of speculative decoding. Additionally, we aim to explore how the system behaves under various models, workloads, and resource configurations. In particular, we are interested in assessing how the accuracy of token acceptance prediction affects overall performance. Moreover, carrying out comprehensive experiments across all possible configurations is both time-consuming and cost-prohibitive. Consequently, we utilize a simulator for all subsequent experiments.

Simulator construction. The design of the simulator is closely aligned with the operational flow of vLLM [19], accurately replicating its model scheduling and control logic. The primary distinction between the simulator and actual hardware lies in its use of an event clock to simulate kernel execution times, allowing it to operate efficiently on CPUs without the need for real GPU hardware. Essentially, the simulator employs an event clock to replace all kernel execution times while preserving all other operational logic. To ensure the event clock advances accurately and reflects realistic execution times, we have profiled GPU execution times across various hardware configurations and model settings utilized in the experiments. This approach allows us to simulate real-world operations with high fidelity.

Simulator fidelity. The data we have collected for each job allows our simulator to accurately model several system effects. This includes the performance impact of various scheduling policies and system overheads such as slow sampling and Python overhead, identified through profiling. However,

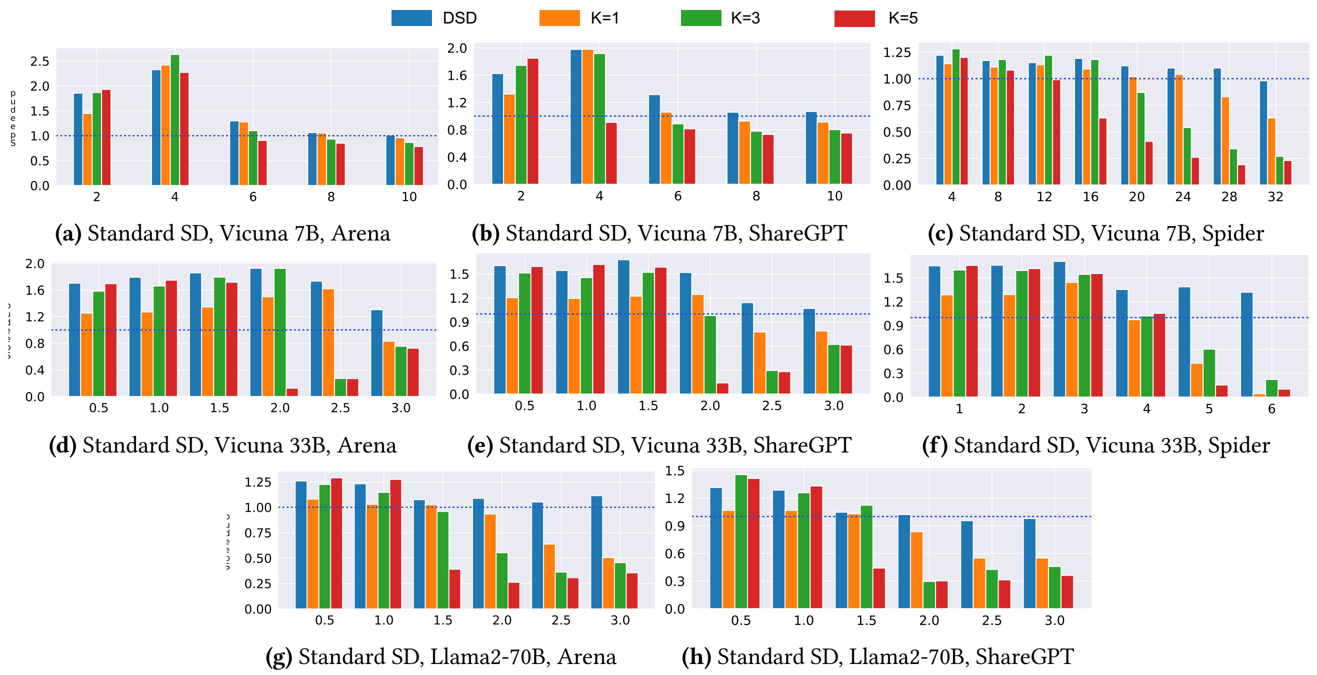


图11. 标准SD上的延迟测量：不同数据集的加速。X轴：请求速率。

推测解码不会影响系统性能。

7.1.2 草稿模型无模型推测解码。接下来，我们评估SmartSpec在提示查找解码中的效率。与我们在草稿模型基础的推测解码测试一样，我们将SmartSpec使用固定提议长度1、3和5与没有推测解码的基线进行比较。如图12所示，SmartSpec始终实现最佳加速。值得注意的是，使用Mistral-7B的提示查找解码（图12（a）和（b））即使在相对较低的令牌接受率下（这两个设置的测量令牌接受率在0.3和0.4之间）也显示出显著的加速。与涉及草稿模型的场景不同，提示查找在提议令牌时不会产生显著的开销，即使在较低的推测准确率下也能带来显著的速度提升。

像基于模型的草稿推测解码一样，SmartSpec 在请求速率高时也不妥协性能。这是因为 SmartSpec 在高请求速率下采取保守的方法，最小化验证错误标记的浪费计算。这一策略确保 SmartSpec 在不同系统负载下保持其效率。

7.2 仿真实验

我们进行以下实验是使用模拟器出于几个原因。首先，将推测性解码有效地集成到现实世界系统中面临着巨大的工程挑战。例如，设计有效的验证树结构推测解码的内核证明是困难的。缺乏这样的内核否定了推测解码的优势。此外，我们旨在探索系统在各种模型、工作负载和资源配置下的表现。特别是，我们对评估令牌接受预测的准确性如何影响整体性能感兴趣。此外，在所有可能的配置中进行全面实验既耗时又成本高昂。因此，我们为所有后续实验使用模拟器。

模拟器构建。模拟器的设计与 vLLM [19] 的操作流程紧密对齐，准确复制其模型调度和控制逻辑。模拟器与实际硬件之间的主要区别在于它使用事件时钟来模拟内核执行时间，使其能够在 CPU 上高效运行，而无需真实的 GPU 硬件。基本上，模拟器使用事件时钟来替代所有内核执行时间，同时保留所有其他操作逻辑。为了确保事件时钟准确推进并反映现实的执行时间，我们对在实验中使用的各种硬件配置和模型设置的 GPU 执行时间进行了分析。这种方法使我们能够以高保真度模拟现实世界的操作。

模拟器的保真度。我们为每个作业收集的数据使我们的模拟器能够准确地模拟多个系统效应。这包括各种调度策略的性能影响和系统开销，例如通过分析识别的慢采样和Python开销。然而，

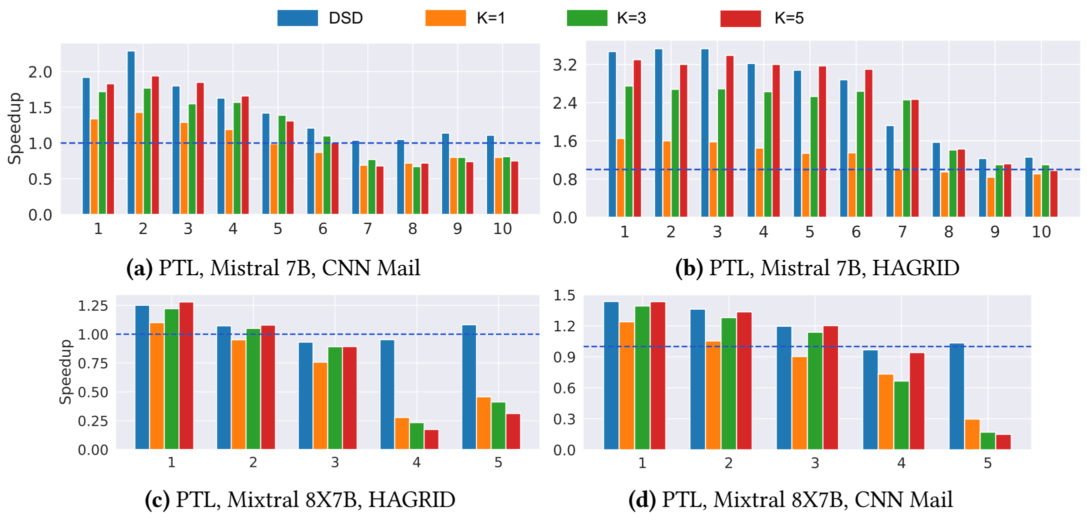
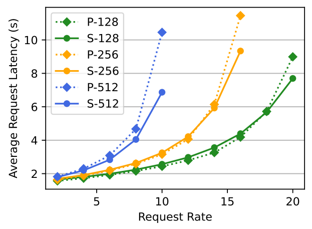


Figure 12. Latency Measurement on PTL: Speedup across different datasets. X-axis: request rate.



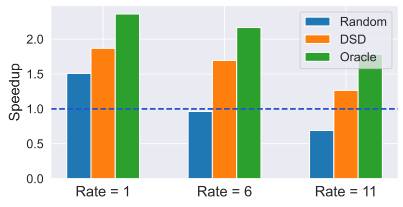


Figure 13. Simulator Fidelity. This experiment was conducted on a single Azure NC24ads machine equipped with an A100-80G GPU. The labels 'P' and 'S' indicate profiled and simulated data, respectively. The figure demonstrates that the simulated average request latency closely mirrors the measured latency across various input/output lengths and request rates.

Figure 14. Optimal speedup vs SmartSpec speedup. SmartSpec means we use moving average to predict the goodput and use goodput to guide the decision. Random all means we randomly predict the accepted length without using goodput.

our simulator does not account for network latency. After calibration, as shown in Fig. 13, the simulator demonstrates an error rate below 10% when the request rate is lower than the service rate. This accuracy is consistent across various input/output lengths and request rates. It is important to note that the simulator tends to under-predict latency when the request rate exceeds the service rate due to its limited ability to simulate queuing delays. For the subsequent experiments presented in this paper, we will focus exclusively on scenarios where the request rate is less than the service rate.

Using the simulator, we initially identify the discrepancy between the current speedup and the optimal speedup, where "optimal" implies foreknowledge of the accepted length. Subsequently, we implement tree-style speculative decoding Medusa with continuous batching to test SmartSpec's generality.

7.2.1 Accepted Length Prediction and Speedup In this section, we explore how the accuracy of acceptance modeling impacts computational speedup. We keep the request rate constant to ensure a controlled comparison of different acceptance prediction techniques. We assess the effectiveness of SmartSpec's accepted length prediction, which employs a moving average based on historical token acceptance rates, and compare it to an oracle. This oracle assumes access to a perfect predictor and uses the actual accepted lengths to calculate goodput and determine the optimal proposed length. As shown in Figure 14, there is a noticeable performance gap between SmartSpec's speedup and that of the oracle. Developing a more efficient accepted length predictor remains an area for future research. However, it is important to note that, even with the moving average approach, the speedup achieved by SmartSpec is substantial and represents a significant improvement over strategies that rely on random proposals.

7.2.2 Tree-style Speculative Decoding In this section, we evaluate the applicability of SmartSpec to Medusa [4], a tree-style speculative decoding method. Prior to integrating SmartSpec, Medusa could only be implemented with a

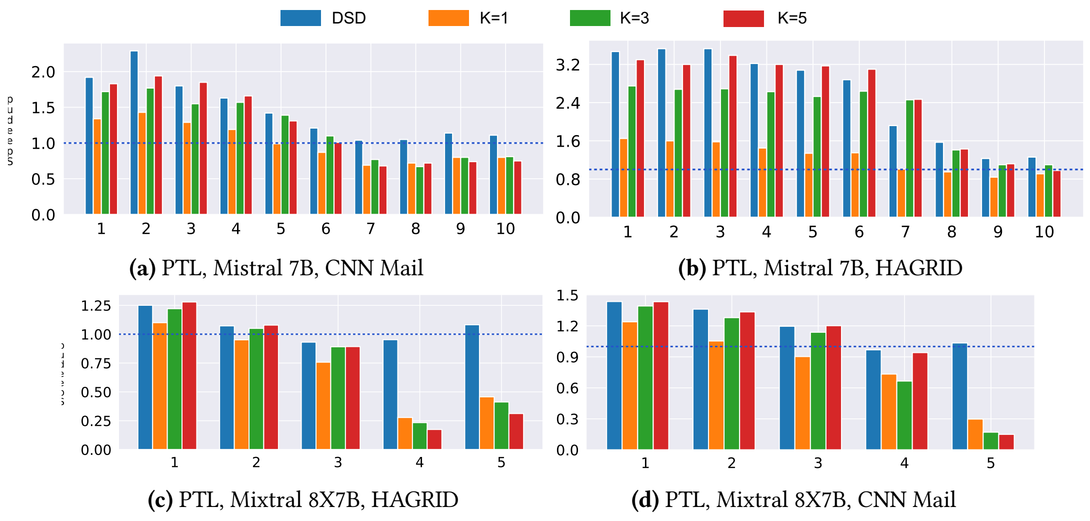


图12. PTL上的延迟测量：不同数据集的加速。X轴：请求速率。

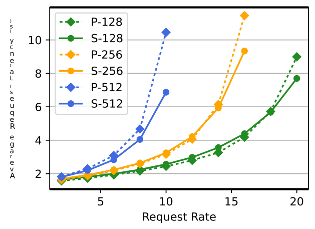


图13. 模拟器保真度。该实验在一台配备A100-80G GPU的单个Azure NC24ads机器上进行。标签“P”和“S”分别表示分析数据和模拟数据。该图表明，模拟的平均请求延迟与在各种输入/输出长度和请求速率下测量的延迟密切相符。

我们的模拟器不考虑网络延迟。经过校准，如图13所示，当请求速率低于服务速率时，模拟器的错误率低于10%。这种准确性在各种输入/输出长度和请求速率下是一致的。需要注意的是，当请求速率超过服务速率时，由于其有限的排队延迟模拟能力，模拟器往往会低估延迟。在本文后续的实验中，我们将专注于请求速率低于服务速率的场景。

使用模拟器，我们最初识别当前加速与最佳加速之间的差异，其中“最佳”意味着对接受长度的预先了解。随后，我们实现了树状的推测解码Medusa，并采用连续批处理来测试SmartSpec的通用性。

7.2.1 接受长度预测和加速 在本节中，我们探讨接受建模的准确性如何

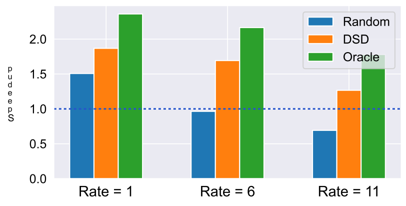
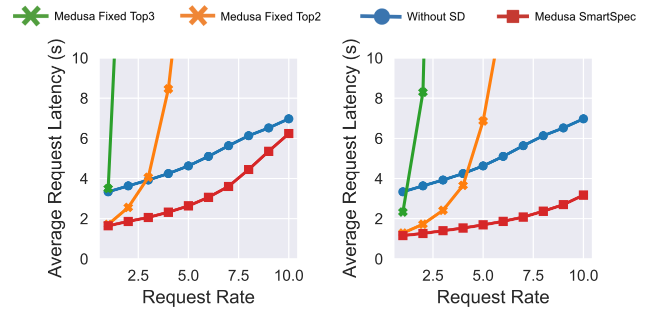


图14. 最优加速与SmartSpec加速。SmartSpec意味着我们使用移动平均来预测有效吞吐量，并使用有效吞吐量来指导决策。随机全部意味着我们随机预测接受的长度，而不使用有效吞吐量。

影响计算速度提升。我们保持请求速率不变，以确保对不同接受预测技术的控制比较。我们评估SmartSpec的接受长度预测的有效性，该预测采用基于历史令牌接受率的移动平均，并将其与一个oracle进行比较。这个oracle假设可以访问一个完美的预测器，并使用实际接受的长度来计算良好输出并确定最佳提议长度。如图14所示，SmartSpec的加速与oracle的加速之间存在明显的性能差距。开发更高效的接受长度预测器仍然是未来研究的一个领域。然而，重要的是要注意，即使采用移动平均方法，SmartSpec所实现的加速也是相当可观的，并且相较于依赖随机提议的策略，代表了显著的改进。

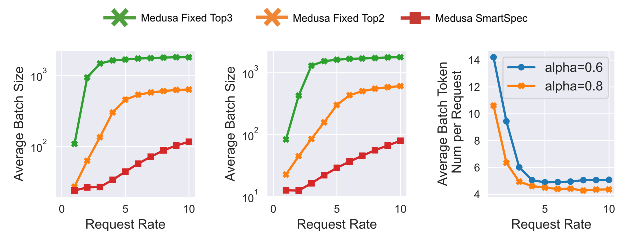
7.2.2 树状风格的推测解码 在本节中，我们评估了 Smart Spec 在 Medusa [4] 中的适用性，Medusa 是一种树状风格的推测解码方法。在集成 SmartSpec 之前，Medusa 只能以一种方式实现。



(a) $\alpha = 0.6$

(b) $\alpha = 0.8$

Figure 15. Average request latency of Medusa with fixed top k values and SmartSpec: Simulating the performance of Llama-7B with three fixed heads across 500 randomly sampled requests of varying input/output Lengths under different request rates. α: token acceptance rate of candidate tokens across all heads.



(a) $\alpha = 0.8$, average batch token number. **(b)** $\alpha = 0.6$, average batch token number. **(c)** Average token number per request.

Figure 16. (a), (b) show the average batch token number across batches with vanilla Medusa and SmartSpec Medusa. (c) shows the average token number across requests with SmartSpec.

batch size of 1. To test the enhanced capabilities, we simulate Medusa with continuous batching and assess its performance both with and without SmartSpec integration. For our experiments, we maintain a consistent token acceptance rate across all Medusa heads and for all tokens within the top-k selection. Additionally, we model Medusa with dense connectivity, ensuring that each of the top-k nodes is linked to the corresponding top-k tokens at the subsequent position.

We illustrate the average request latency across various k-values under differing token acceptance rates in Fig. 16. As demonstrated in the figure, the tree-style speculative decoding method substantially increases request latency at high request rates. This outcome aligns with expectations outlined in [4], which describes how dense connections among different heads greatly increase the the number of batched tokens. As shown in Figs. 16a and 16b, fixed top2/top3 Medusa quickly explode the batch size. Specifically, the number of batched tokens per request is represented by $\sum_{h=1}^{H} \Pi_{i=1}^{h} s_i$, where s_i denotes the number of tokens sampled by head i. For example, selecting 3, 2, and 2 tokens for heads 1, 2, and 3, respectively, results in the addition of 21 tokens in the next batch for verification (calculated as $3 + 3 \times 2 + 3 \times 2 \times 2$). Conversely, with only three heads corresponding to three positions, for each request, a maximum of 4 tokens (3 plus 1 bonus token) can be processed in a single forward pass. This inefficiency underscores the need for future advancements such as those proposed by sequoia [6], which aims to develop a more structured verification tree to effectively prune less likely candidates under high request volumes.

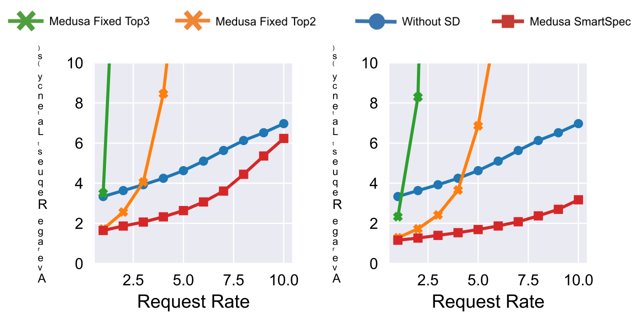
Finally, we show the performance of SmartSpec when integrated with Medusa. We model the accepted token length as outlined in Section 4.4 and evaluate performance using goodput to decide the number of sampled tokens per head. As illustrated in Figure 16, SmartSpec effectively maintains manageable request latencies even under high request rates. Additionally, we depict the average number of tokens per request in Figure 16c. In both scenarios (α is 0.6 or 0.8), SmartSpec quickly reverts to top-1 sampling. It is noteworthy that the average number of batched tokens approximates to four; this consists of one input token plus one sampled token per head. Given that there are three Medusa heads, the total number of batched tokens per request remains four when employing top-1 sampling for each head.

8 Related Work

Aside from speculative decoding (Sec 2.1) and continuous batching (Sec 2.2), the system community has proposed many orthogonal methods to improve LLM inference performance.

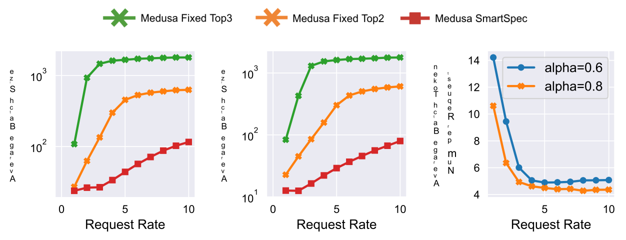
Quantization Methods. Works like (LLM.int8() [7], GPTQ [9], Marlin [8], AWQ [23], SqueezeLLM [17]) bring down the latency of the LLM inference by using lower precision data types such as 2/3/4/6/8 bit integers. GPUs For example, a single A100 GPU can support 624 Teraops of `INT8` computation through its tensor core, while only able to support 312 TFLOPS for `FLOAT16`. However, this class of method trade off accuracy for performance and commonly requires a calibration step. In the context of SmartSpec, quantization optimizations can be applied to both draft and target models to further improve our performance.

Prefix Caching techniques save compute of commonly repeated prefixes across requests. Systems like SGLang [45], Cascade Inference [40], and Hydragen [15] proposes efficient GPU kernels to compute and cache the computation for shared prefixes across request and deliver lower inference latency. In the context of SmartSpec, prefix caching can be applied to both draft and target workers.



(a) α = 0.6　　**(b)** α = 0.8

图15. Medusa在固定的top k值和SmartSpec下的平均请求延迟：模拟Llama-7B在不同请求速率下对500个随机采样的输入/输出长度变化的请求的性能，使用三个固定的头。α: 所有头部候选令牌的接受率。



(a) α = 0.8，平均批次令牌数量。(b) α = 0.6，平均批次令牌数量。(c) 每个请求的平均令牌数量。

图16. (a), (b) 显示了使用普通Medusa和SmartSpec Medusa的批次中平均批次令牌数量。(c) 显示了使用SmartSpec的请求中平均令牌数量。

批量大小为1。为了测试增强的能力，我们通过连续批处理模拟Medusa，并评估其在有无SmartSpec集成的情况下的性能。在我们的实验中，我们在所有Medusa头部和所有位于top-k选择中的令牌之间保持一致的令牌接受率。此外，我们使用密集连接建模Medusa，确保每个top-k节点都与后续位置的相应top-k令牌相连。

我们在图16中展示了在不同的令牌接受率下，各种k值的平均请求延迟。如图所示，树状的推测解码方法在高请求率下显著增加了请求延迟。这个结果与[4]中概述的预期一致，该文描述了不同头部之间的密集连接如何大大增加批处理令牌的数量。如图16a和16b所示，固定的top2/top3 Medusa迅速扩大了批处理大小。具体而言，每个请求的批处理令牌数量用$\sum_{h=1}^{H}\Pi_{i=1}^{h}s_i$表示，其中$s_i$表示由头$i$采样的令牌数量。例如，为头1、2和3分别选择3、2和2个令牌，结果是在下一个批次中增加21个令牌以供验证（计算为$3+3\times2+3\times2\times2$）。相反，只有三个头对应于三个位置，对于每个请求，最多可以在一次前向传递中处理4个令牌（3个加1个额外令牌）。这种低效性突显了未来进步的必要性，例如sequoia [6]所提出的，旨在开发一个更结构化的验证树，以有效地在高请求量下修剪不太可能的候选者。

最后，我们展示了SmartSpec与Medusa集成时的性能。我们按照第4.4节中概述的内容对接受的令牌长度进行建模，并使用有效吞吐量评估性能，以决定每个头部的采样令牌数量。

如图16所示，SmartSpec在高请求速率下有效地保持可管理的请求延迟。此外，我们在图16c中描绘了每个请求的平均令牌数。在这两种情况下（α为0.6或0.8），SmartSpec迅速恢复到top-1采样。值得注意的是，批处理令牌的平均数量接近四；这包括一个输入令牌加上每个头部一个采样令牌。考虑到有三个美杜莎头部，每个请求的批处理令牌总数在对每个头部采用top-1采样时仍然保持为四。

8 相关工作

除了推测解码（第2.1节）和连续批处理（第2.2节）之外，系统社区提出了许多正交方法来提高LLM推理性能。

量化方法。像 (LLM.int8() [7]、GPTQ [9]、Marlin [8]、AWQ [23]、SqueezeLLM [17]) 这样的工作通过使用较低精度的数据类型，如 2/3/4/6/8 位整数，降低了 LLM 推理的延迟。例如，单个 A100 GPU 可以通过其张量核心支持 624 Teraops 的 INT8 计算，而仅能支持 312 TFLOPS 的 FLOAT16。然而，这类方法在性能与准确性之间进行权衡，并通常需要一个校准步骤。在 SmartSpec 的背景下，量化优化可以应用于草稿模型和目标模型，以进一步提高我们的性能。

前缀缓存技术可以节省在请求中常见重复前缀的计算。像 SGLang [45]、Cascade Inference [40] 和 Hydragen [15] 这样的系统提出了高效的 GPU 内核，以计算和缓存请求中共享前缀的计算，从而降低推理延迟。在 SmartSpec 的上下文中，前缀缓存可以应用于草稿和目标工作者。

9 Conclusion

Speculative decoding has recently emerged as a means to reduce inference latency at the cost of increased computational overhead. To harness the benefits of speculative decoding without compromising efficiency, we introduce an adaptive decision-making framework SmartSpec guided by the concept of goodput. Our evaluation across three distinct datasets show that SmartSpec can reduce latency by a factor of 1.2× to 3.2× when request rates are low, while sustaining performance levels even under high request rates.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] anon8231489123. 2024. *ShareGPT dataset.* https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered

[3] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A neural probabilistic language model. *Advances in neural information processing systems* 13 (2000).

[4] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774* (2024).

[5] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318* (2023).

[6] Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. 2024. Sequoia: Scalable, Robust, and Hardware-aware Speculative Decoding. *arXiv preprint arXiv:2402.12374* (2024).

[7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. arXiv:2208.07339 [cs.LG]

[8] Elias Frantar and Dan Alistarh. 2024. Marlin: a fast 4-bit inference kernel for medium batchsizes. https://github.com/IST-DASLab/marlin.

[9] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 [cs.LG]

[10] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of llm inference using lookahead decoding. *arXiv preprint arXiv:2402.02057* (2024).

[11] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[12] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching Machines to Read and Comprehend. In *NIPS*. 1693–1701. http://papers.nips.cc/paper/5945-teaching-machines-to-read-and-comprehend

[13] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[14] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[15] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. 2024. Hydragen: High-Throughput LLM Inference with Shared Prefixes. arXiv:2402.05099 [cs.LG]

[16] Ehsan Kamalloo, Aref Jafari, Xinyu Zhang, Nandan Thakur, and Jimmy Lin. 2023. HAGRID: A Human-LLM Collaborative Dataset for Generative Information-Seeking with Attribution. *arXiv:2307.16883* (2023).

[17] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. 2024. SqueezeLLM: Dense-and-Sparse Quantization. arXiv:2306.07629 [cs.CL]

[18] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W Mahoney, Amir Gholami, and Kurt Keutzer. 2024. Speculative decoding with big little decoder. *Advances in Neural Information Processing Systems* 36 (2024).

[19] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica.

9 结论

投机解码最近作为一种减少推理延迟的方法出现，但代价是增加计算开销。为了在不影响效率的情况下利用投机解码的好处，我们引入了一个自适应决策框架SmartSpec，该框架以良好吞吐量的概念为指导。我们在三个不同数据集上的评估显示，当请求速率较低时，SmartSpec可以将延迟减少1.2×到3.2×倍，同时在高请求速率下仍能维持性能水平。

参考文献

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat 等. 2023. Gpt-4 技术报告. arXiv 预印本 arXiv:2303.08774 (2023). [2] anon8231489123. 2024. ShareGPT 数据集. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered [3] Yoshua Bengio, Réjean Ducharme 和 Pascal Vincent. 2000. 一种神经概率语言模型. 神经信息处理系统进展 13 (2000). [4] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen 和 Tri Dao. 2024. Medusa: 简单的 LLM 推理加速框架，具有多个解码头. arXiv 预印本 arXiv:2401.10774 (2024). [5] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre 和 John Jumper. 2023. 通过投机采样加速大型语言模型解码. arXiv 预印本 arXiv:2302.01318 (2023). [6] Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia 和 Beidi Chen. 2024. Sequoia: 可扩展、稳健且硬件感知的投机解码. arXiv 预印本 arXiv:2402.12374 (2024). [7] Tim Dettmers, Mike Lewis, Younes Belkada 和 Luke Zettlemoyer. 2022. LLM.int8(): 大规模变换器的 8 位矩阵乘法. arXiv:2208.07339 [cs.LG] [8] Elias Frantar 和 Dan Alistarh. 2024. Marlin: 一种快速的 4 位推理内核，适用于中等批量大小. https://github.com/IST-DASLab/marlin. [9] Elias Frantar, Saleh Ashkboos, Torsten Hoefler 和 Dan Alistarh. 2023. GPTQ: 生成预训练变换器的准确后训练量化. arXiv:2210.17323 [cs.LG] [10] Yichao Fu, Peter Bailis, Ion Stoica 和 Hao Zhang. 2024. 通过前瞻解码打破 LLM 推理的顺序依赖. arXiv 预印本 arXiv:2402.02057 (2024). [11] Pin Gao, Lingfan Yu, Yongwei Wu 和 Jinyang Li. 2018. 低延迟 RNN 推理与单元批处理. 在第十三届 EuroSys 会议论文集中. 1–15. [12] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman 和 Phil Blunsom. 2015. 教机器阅读和理解. 在 NIPS. 1693–1701. http://papers.nips.cc/paper/5945-teaching-machines-to-read-and-comprehend [13] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier 等. 2023. Mistral 7B. arXiv 预印本 arXiv:2310.06825 (2023). [14] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand 等. 2024. 专家混合模型. arXiv 预印本 arXiv:2401.04088 (2024). [15] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré 和 Azalia Mirhoseini. 2024. Hydragen: 具有共享前缀的高吞吐量 LLM 推理. arXiv:2402.05099 [cs.LG] [16] Ehsan Kamalloo, Aref Jafari, Xinyu Zhang, Nandan Thakur 和 Jimmy Lin. 2023. HAGRID: 一种用于生成信息检索的人工-LLM 协作数据集，具有归属. arXiv:2307.16883 (2023). [17] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney 和 Kurt Keutzer. 2024. SqueezeLLM: 稠密与稀疏量化. arXiv:2306.07629 [cs.CL] [18] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W Mahoney, Amir Gholami 和 Kurt Keutzer. 2024. 使用大小解码器的投机解码. 神经信息处理系统进展 36 (2024). [19] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang 和 Ion Stoica.

2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[20] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[21] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077* (2024).

[22] Feng Lin, Hanling Yi, Hongbin Li, Yifan Yang, Xiaotian Yu, Guangming Lu, and Rong Xiao. 2024. BiTA: Bi-Directional Tuning for Lossless Acceleration in Large Language Models. *arXiv preprint arXiv:2401.12522* (2024).

[23] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, Chuang Gan, and Song Han. 2023. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. arXiv:2306.00978 [cs.CL]

[24] Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Ion Stoica, Zhijie Deng, Alvin Cheung, and Hao Zhang. 2023. Online speculative decoding. *arXiv preprint arXiv:2310.07177* (2023).

[25] Yusuf Mehdi. 2023. *Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web.* https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web/ Accessed: 2024-02-21.

[26] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2023. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781* (2023).

[27] Microsoft. 2023. *Copilot.* https://copilot.microsoft.com/ Accessed: 2024-02-21.

[28] Nvidia. 2024. A100 GPU Spec. https://www.nvidia.com/en-us/data-center/a100/ Accessed: 2024-03-10.

[29] NVIDIA. 2024. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[30] OpenAI. 2022. *ChatGPT.* https://chat.openai.com/ Accessed: 2024-02-21.

[31] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems* 5 (2023).

[32] Elizabeth Reid. 2023. *Supercharging Search with generative AI.* https://blog.google/products/search/generative-ai-search/ Accessed: 2024-02-21.

[33] Apoorv Saxena. 2023. Prompt Lookup Decoding. https://github.com/apoorvumang/prompt-lookup-decoding/

[34] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1073–1083. https://doi.org/10.18653/v1/P17-1099

[35] Qidong Su, Christina Giannoula, and Gennady Pekhimenko. 2023. The synergy of speculative decoding and batching in serving large language models. *arXiv preprint arXiv:2310.18813* (2023).

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[37] Siqi Wang, Hailong Yang, Xuezhu Wang, Tongxuan Liu, Pengbo Wang, Xuning Liang, Kejie Ma, Tianyu Feng, Xin You, Yongjun Bao, et al. 2024. Minions: Accelerating Large Language Model Inference with Adaptive and Collective Speculative Decoding. *arXiv preprint arXiv:2402.15678* (2024).

[38] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs.AI]

[39] Yiran Wu, Feiran Jia, Shaokun Zhang, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, Qingyun Wu, and Chi Wang. 2023. An Empirical Study on Challenging Math Problem Solving with GPT-4. In *ArXiv preprint arXiv:2306.01337*.

[40] Zihao Ye, Ruihang Lai, Bo-Ru Lu, Chien-Yu Lin, Size Zheng, Lequn Chen, Tianqi Chen, and Luis Ceze. 2024. Cascade Inference: Memory Bandwidth Efficient Shared Prefix Batch Decoding. https://flashinfer.ai/2024/02/02/cascade-inference.html

[41] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[42] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887* (2018).

[43] Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2023. Draft & verify: Lossless large language model acceleration via self-speculative decoding. *arXiv preprint arXiv:2309.08168* (2023).

[44] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2024. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems* 36 (2024).

[45] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2023. Efficiently Programming Large Language Models using SGLang. arXiv:2312.07104 [cs.AI]

[46] Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. 2023. Distillspec: Improving speculative decoding via knowledge distillation. *arXiv preprint arXiv:2310.08461* (2023).

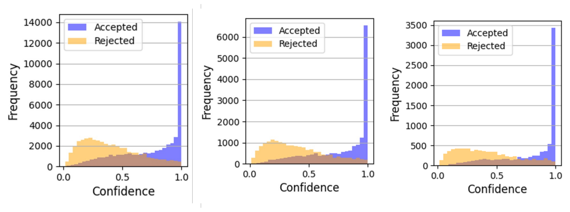
2023年。针对大型语言模型服务的高效内存管理与分页注意力。在第29届操作系统原理研讨会论文集中。611–626。[20] Yaniv Leviathan, Matan Kalman 和 Yossi Matias。2023年。通过推测解码实现快速推理。国际机器学习会议。PMLR, 19274–19286。[21] Yuhui Li, Fangyun Wei, Chao Zhang 和 Hongyang Zhang。2024年。Eagle：推测采样需要重新思考特征不确定性。arXiv 预印本 arXiv:2401.15077 (2024)。[22] Feng Lin, Hanling Yi, Hongbin Li, Yifan Yang, Xiaotian Yu, Guangming Lu 和 Rong Xiao。2024年。BiTA：用于大型语言模型无损加速的双向调优。arXiv 预印本 arXiv:2401.12522 (2024)。[23] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, Chuang Gan 和 Song Han。2023年。AWQ：用于LLM压缩和加速的激活感知权重量化。arXiv:2306.00978 [cs.CL] [24] Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Ion Stoica, Zhijie Deng, Alvin Cheung 和 Hao Zhang。2023年。在线推测解码。arXiv 预印本 arXiv:2310.07177 (2023)。[25] Yusuf Mehdi。2023年。通过新的AI驱动的Microsoft Bing和Edge重新定义搜索，成为您的网络副驾驶。https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web/ 访问时间：2024-02-21。[26] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar 和 Zhihao Jia。2023年。Specinfer：通过推测推理和令牌树验证加速生成LLM服务。arXiv 预印本 arXiv:2305.09781 (2023)。[27] Microsoft。2023年。Copilot。https://copilot.microsoft.com/ 访问时间：2024-02-21。[28] Nvidia。2024年。A100 GPU规格。https://www.nvidia.com/en-us/data-center/a100/ 访问时间：2024-03-10。[29] NVIDIA。2024年。TensorRT-LLM。https://github.com/NVIDIA/TensorRT-LLM。[30] OpenAI。2022年。ChatGPT。https://chat.openai.com/ 访问时间：2024-02-21。[31] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal 和 Jeff Dean。2023年。高效扩展变压器推理。机器学习与系统会议论文集 5 (2023)。[32] Elizabeth Reid。2023年。通过生成AI超级充电搜索。https://blog.google/products/search/generative-ai-search/ 访问时间：2024-02-21。[33] Apoorv Saxena。2023年。提示查找解码。https://github.com/apoorvumang/prompt-lookup-decoding/[34] Abigail See, Peter J. Liu 和 Christopher D. Manning。2017年。直达要点：使用指针生成网络进行摘要。在第55届计算语言学协会年会上（第1卷：长篇论文）论文集中。计算语言学协会，加拿大温哥华，1073–1083。https://doi.org/10.18653/v1/P17-1099[35] Qidong Su, Christina Giannoula 和 Gennady Pekhimenko。2023年。推测解码与批处理在大型语言模型服务中的协同作用。arXiv 预印本 arXiv:2310.18813 (2023)。[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale 等。2023年。Llama 2：开放基础和微调聊天模型。arXiv 预印本 arXiv:2307.09288 (2023)。[37] Siqi Wang, Hailong Yang, Xuezhu Wang, Tongxuan Liu, Pengbo Wang, Xuning Liang, Kejie Ma, Tianyu Feng, Xin You, Yongjun Bao 等。2024年。Minions：通过自适应和集体推测解码加速大型语言模型推理。arXiv 预印本 arXiv:2402.15678 (2024)。

[38] 吴青云, 班萨尔, 张杰宇, 吴怡然, 李北滨, 朱尔康, 姜丽, 张晓云, 张少坤, 刘佳乐, 阿赫迈德·哈桑·阿瓦达拉, 瑞恩·W怀特, 道格·伯杰, 和王驰. 2023. AutoGen: 通过多智能体对话启用下一代 LLM 应用. arXiv:2308.08155 [cs.AI] [39] 吴怡然, 贾飞然, 张少坤, 李航宇, 朱尔康, 王悦, 李尹达, 彭理查德, 吴青云, 和王驰. 2023. 关于使用 GPT-4 解决具有挑战性的数学问题的实证研究. 在 ArXiv 预印本 arXiv:2306.01337. [40] 叶子豪, 来瑞航, 陆博如, 林建宇, 郑思泽, 陈乐群, 陈天奇, 和路易斯·塞泽. 2024. 级联推理: 内存带宽高效的共享前缀批解码. https://flashinfer.ai/2024/02/02/cascade-inference.html [41] 俞庆仁, 郑周成, 金建宇, 金秀晶, 和全炳根. 2022. Orca: 一种用于 {Transformer-Based} 生成模型的分布式服务系统. 在第 16 届 USENIX 操作系统设计与实现研讨会 (OSDI 22). 521–538. [42] 余涛, 张锐, 杨凯, 安智宏, 王东旭, 李子凡, 马詹姆斯, 李艾琳, 姚青宁, 罗曼·香奈儿, 等. 2018. Spider: 一个用于复杂和跨领域语义解析及文本到 SQL 任务的大规模人工标注数据集. arXiv 预印本 arXiv:1809.08887 (2018). [43] 张俊, 王珏, 李欢, 守丽丹, 陈克, 陈刚, 和沙拉德·梅赫罗特拉. 2023. 草拟与验证: 通过自我推测解码实现无损大语言模型加速. arXiv 预印本 arXiv:2309.08168 (2023). [44] 郑连敏, 蔡伟林, 盛颖, 庄思源, 吴张浩, 庄永浩, 林子, 李卓涵, 李大成, 邢艾瑞克, 等. 2024. 通过 mt-bench 和聊天机器人竞技场评判 llm-as-a-judge. 神经信息处理系统进展 36 (2024). [45] 郑连敏, 尹良生, 谢志强, 黄杰夫, 孙楚悦, 余浩宇, 曹诗怡, 克里斯托斯·科齐拉基斯, 斯托伊卡, 约瑟夫·E·冈萨雷斯, 克拉克·巴雷特, 和盛颖. 2023. 使用 SGLang 高效编程大语言模型. arXiv:2312.07104 [cs.AI] [46] 周永超, 吕凯峰, 拉瓦特·安基特·辛格, 梅农·阿迪提亚·克里希纳, 罗斯塔米扎德·阿夫辛, 库马尔·桑吉夫, 卡吉·让-弗朗索瓦, 和阿加尔瓦尔·瑞沙布. 2023. Distillspec: 通过知识蒸馏改进推测解码. arXiv 预印本 arXiv:2310.08461 (2023).

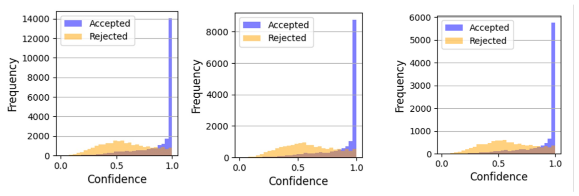
Algorithm 3 SmartSpec confidence based propose and verify.

```
Require: Pending requests $R$. Confidence threshold $T$. Max propose length $P$.
1: $best\_goodput, best\_verify\_lens \leftarrow -1, 0$
2: for each $r$ in $R$ do
3:     // Initialize the confidence before proposing the first token for each request
4:     $r.conf = 1$
5: end for
6: $propose\_steps \leftarrow 0$
7: while $len(R) > 0$ and $propose\_stpes <= P$ do
8:     $R' = \{r$ for $r$ in $R$ if $r.conf > T\}$
9:     // Propose will update the $conf$ attribute for each request $r$ in $R'$
10:    $R = \text{Propose}(R')$
11:    $propose\_steps++$
12: end while
13: ArgmaxGoodput($R$)
14: Score($R$)
```

10 Appendix



(a) 7B, TP=1.



(b) 70B, TP=4.

Figure 17. Confidence distribution.

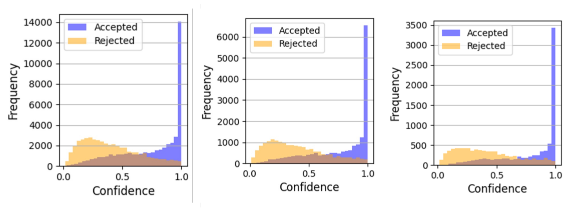
In this section, we explore the utilization of confidence as a criterion for token acceptance. Confidence is defined as the output probability of the proposed token generated by the draft model As depicted in Figure 17, a discernible distinction exists between the confidence distributions of accepted and rejected tokens. This distinction intuitively suggests that tokens proposed by the draft model, when accompanied by high confidence levels, are likely to be accurate. Conversely, proposals made with low confidence are less likely to be accepted by the target model. In practice, we establish a threshold T. Tokens with a confidence level exceeding T are predicted to be accepted, while those below this threshold are anticipated to be rejected.

Initially, SmartSpec sets the confidence level of each request to 1, adhering to the definition in Section 4.4 where confidence is treated as a probability and thus cannot exceed 1. This ensures that the draft model will propose at least one token, activating the procedure described in lines 7-12 at least once, provided that $P > 0$. During each proposal step (lines 7-12), SmartSpec selectively processes only those requests, denoted as R', whose confidence levels surpass the specified threshold T from the preceding proposal step. Subsequently, SmartSpec batches these R' requests for a forward pass execution. Following the strategy outlined above, SmartSpec also explores all potential lengths for verification and opts for the length that maximizes goodput.

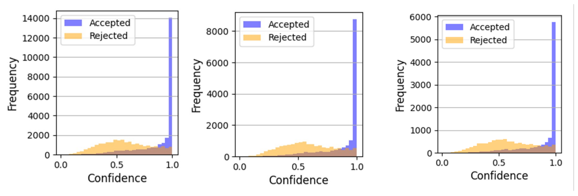
Algorithm 3 SmartSpec confidence based propose and verify.

```
Require: Pending requests $R$. Confidence threshold $T$. Max propose length $P$.
1:  $best\_goodput, best\_verify\_lens \leftarrow -1, 0$
2:  for each $r$ in $R$ do
3:      // Initialize the confidence before proposing the first token for each request
4:      $r.conf = 1$
5:  end for
6:  $propose\_steps \leftarrow 0$
7:  while $len(R) > 0$ and $propose\_stpes <= P$ do
8:      $R' = \{r$ for $r$ in $R$ if $r.conf > T\}$
9:      // Propose will update the $conf$ attribute for each request $r$ in $R'$
10:     $R = \text{Propose}(R')$
11:     $propose\_steps++$
12: end while
13: ArgmaxGoodput($R$)
14: Score($R$)
```

10 附录



(a) 7B, TP=1.



(b) 70B, TP=4.

图 17. 置信度分布。

在本节中，我们探讨将置信度作为令牌接受标准的利用。置信度被定义为草稿模型生成的提议令牌的输出概率。如图17所示，接受和拒绝的令牌的置信度分布之间存在明显的区别。这种区别直观地表明，当草稿模型提出的令牌伴随高置信度水平时，可能是准确的。相反，低置信度的提议不太可能被目标模型接受。在实践中，我们建立了一个阈值 T。置信度水平超过 T的令牌被预测为会被接受，而低于该阈值的令牌则预计会被拒绝。

最初，SmartSpec将每个请求的置信度水平设置为1，遵循第4.4节中的定义，其中置信度被视为概率，因此不能超过1。这确保了草稿模型至少会提出一个标记，从而在$P > 0$的前提下至少激活一次第7-12行中描述的程序。在每个提议步骤（第7-12行）中，SmartSpec仅选择处理那些置信度水平超过前一个提议步骤中指定阈值T的请求，这些请求被标记为R'。随后，SmartSpec将这些R'请求批量处理，以便进行前向传递执行。按照上述策略，SmartSpec还探索所有潜在的验证长度，并选择最大化良好吞吐量的长度。