# REST: Retrieval-Based Speculative Decoding

**Anonymous ACL submission**

## Abstract

We introduce Retrieval-Based Speculative Decoding (REST), a novel algorithm designed to speed up language model generation. The key insight driving the development of REST is the observation that the process of text generation often includes certain common phases and patterns. Unlike previous methods that rely on a draft language model for speculative decoding, REST harnesses the power of retrieval to generate draft tokens. This method draws from the reservoir of existing knowledge, retrieving and employing relevant tokens based on the current context. Its plug-and-play nature allows for seamless integration and acceleration of any language model, all without necessitating additional training. When benchmarked on 7B and 13B language models in a single-batch setting, REST achieves a significant speedup of $1.62\times$ to $2.36\times$ on code or text generation. We will release our source code.

## 1 Introduction

Transformer-based Large Language Models (LLMs) have emerged as a foundation model in natural language processing (Vaswani et al., 2017; Devlin et al., 2019; Brown et al., 2020; Zhang et al., 2022; Scao et al., 2022; Chowdhery et al., 2022; Zeng et al., 2022; Touvron et al., 2023). While they achieve impressive performance across various tasks, the inference cost is huge in practical scenarios. During inference, the model autoregressively uses the preceding context to generate the next token. Each iteration requires reloading the billion-parameter LLM from the High-Bandwidth Memory (HBM) to the on-chip cache of modern accelerators like GPUs, making the whole generation inefficient and time-consuming.

A recent direction in accelerating the LLM generation is to reduce the number of forward processes with LLMs while guaranteeing the quality of the output sequence simultaneously. Speculative decoding (Leviathan et al., 2023; Chen et al., 2023; Miao et al., 2023; Spector and Re, 2023) is one of the typical approaches in this direction. Intuitively, speculative decoding methods leverage a small LM to generate tokens with less computational cost. During inference, the method first uses the small LM to create a *draft token sequence* and then uses the LLM for verification. If the predictions from both models are consistent, we can accept the draft and return it to the user. Here, the actual token generation is carried out using the small LM, and the large LM is only used to validate the draft, which can be performed *in parallel* and requires reloading the memory only once. Consequently, the entire framework of speculative decoding reduces the overall inference cost.

However, obtaining a high-quality draft model remains challenging: It must balance small size and strong predictive power while matching the vocabulary of the base model; also, it should integrate well into a distributed system for serving. Therefore, people often need to train a draft model specifically for their model and use cases (Chen et al., 2023; Miao et al., 2023; Cai et al., 2023). In this study, rather than relying on an additional small LM, we investigate using a data corpus directly to construct the draft token sequence in speculative decoding. We develop a retrieve-based approach, called Retrieval-Based Speculative Decoding (REST) (Figure 1). Compared to previous approaches, our retrieval-based system replaces the parametric draft model with a non-parametric retrieval datastore, which can easily port to any LLM and accelerate its inference.

To use REST, the first step is constructing the datastore. In this paper, we leverage either the pretraining data corpus or the instruction-tuning data corpus to build our datastore, which serves as the source for the draft token sequence. During each inference step, we first use previous tokens

(pre-generated tokens or prompt tokens) as queries to identify exact matches in the datastore. The subsequent tokens from these exact matches are considered generation candidates. A Trie is constructed using these candidates. The nodes with the highest frequencies are selected as the draft tokens. This sequence then undergoes verification by the LLM through a single forward pass, aided by a meticulously designed attention mask known as tree attention (Cai et al., 2023; Miao et al., 2023; Spector and Re, 2023). As many subsequences during generation likely appear in the datastore, REST can frequently generate multiple correct tokens per step.

We conduct extensive experiments to test the efficiency and effectiveness of REST in different scenarios. For the code domain, we use a portion of Python pretraining code (2.7M samples) from The Stack (Kocetkov et al., 2022) as the datastore and accelerate CodeLlama (Rozière et al., 2023) 7B and 13B respectively. The results show on HumanEval (Chen et al., 2021) REST achieves $2.12\times$ to $2.36\times$ speedup. For the general domain, we construct a datastore using UltraChat (Ding et al., 2023), containing around 774K conversations. The results show on MT-Bench (Zheng et al., 2023) REST accelerates 7B and 13B Vicuna (Chiang et al., 2023) by $1.62\times$ to $1.77\times$ respectively.

## 2 Related Work

Improving the efficiency of LLM inference has been an emergent research direction in recent years. Broadly, previous attempts can be divided into two categories: lossless acceleration and lossy acceleration. Lossy acceleration approaches aim to learn efficient models that can execute faster and act similarly to a target LLM. These methods include pruning (Wang et al., 2021; Hubara et al., 2021; Ma et al., 2023; Frantar and Alistarh, 2023), quantization (Yao et al., 2022; Park et al., 2022; Dettmers et al., 2022; Frantar et al., 2022; Xiao et al., 2023; Liu et al., 2023) and knowledge distillation (Sanh et al., 2019). Lossless acceleration strategies focus on directly accelerating the target LLM from different perspectives, such as memory and IO optimization (Dao et al., 2022; Dao, 2023; Kwon et al., 2023; Sheng et al., 2023), and ways to reduce the function calls of LLM during decoding, e.g., speculative decoding (Stern et al., 2018; Leviathan et al., 2023; Chen et al., 2023; Miao et al., 2023; Spector and Re, 2023; Cai et al., 2023). This work falls within the second branch. Speculative decoding (Leviathan et al., 2023; Chen et al., 2023; Miao et al., 2023; Spector and Re, 2023) leverages a smaller model to generate a draft and use LLM to verify the draft tokens with a single forward pass. In this framework, blockwise parallel decoding (Stern et al., 2018) and Medusa (Cai et al., 2023) train multiple heads based on the LLM for draft token generation.

Our method diverges from these approaches by retrieving draft tokens from a datastore, presenting a novel avenue for efficiency improvement in large language model generation. While there is a similar study, LLMA (Yang et al., 2023), that employs retrieval to accelerate generation, our work distinguishes itself in two primary ways: (1) The LLMA approach is tailored towards scenarios where referred contexts (as in Retrieval-Augmented Generation and Cache-Assisted Generation) are provided during generation, and it only retrieves from these referred contexts. In contrast, our method retrieves draft tokens from a comprehensive datastore, thereby not being confined to a small context. (2) In the LLMA framework, the retrieved instance is typically limited to one or a handful. Our method, however, is designed to handle a much larger number of retrieved instances. This difference in approach allows us to leverage a wider information base during the generation process.

## 3 Retrieval-Based Speculative Decoding

In this section, we first provide notations and a background overview of speculative decoding and then introduce our proposed REST framework.

### 3.1 Background: Speculative Decoding

We use $x \in \mathcal{V}$ to denote a token where $\mathcal{V}$ is the vocabulary. At each time step $t$, given the preceding context $s = (x_1, ..., x_{t-1}, x_t)$, the autoregressive decoding method generates the token at position $t+1$ according to:

$$x_{t+1} \sim p(x|x_1, \ldots, x_t; \theta_{large}),$$

where $p(\cdot)$ is the conditional probability distribution calculated by the LLM with parameter $\theta_{large}$. In this process, a forward run of the LLM is required at each step of generation. This is significantly time-consuming due to the memory bandwidth and cannot fully exploit the computational power of modern GPU hardware (Shazeer, 2019).

Speculative decoding aims to reduce the computational cost during inference by reducing the count

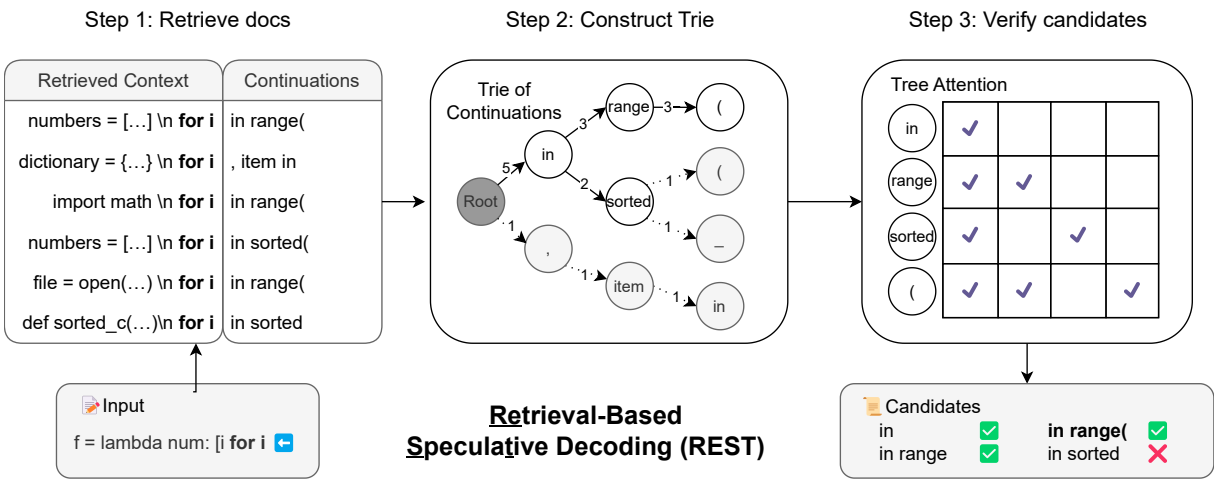


Figure 1: Overview of REST. During inference, the input context is utilized as the query to retrieve docs from the datastore that match the longest suffix of the input. A Trie is constructed using the continuations from the retrieved docs. We prune the low-frequency (weight) branches and the remaining subtree is further used as candidates. Candidates will be fed into the LLM with a tree attention mask for verification. All correct tokens from the start will be accepted, and the draft tokens after the first mistake will be rejected.

of executions with $\theta_{large}$. In addition to the LLM $\theta_{large}$, speculative decoding leverages another language model of a much smaller size with parameter $\theta_{small}$. At step $t$, the method operates by iteratively executing the following steps.

**Draft construction** Given $s = (x_1, \ldots, x_t)$, the small LM $\theta_{small}$ is used to generate the next $m$ tokens $\tilde{x}_{t+1}, \ldots, \tilde{x}_{t+m}$ in an autoregressive way:

$$\tilde{x}_{t+i} \sim p(x|s, \tilde{x}_{t+1}, \ldots, \tilde{x}_{t+i-1}; \theta_{small}), \quad \text{where } i = 1, \ldots, m.$$

Although the tokens are still generated one by one, the computational cost of this process is reduced as it uses $\theta_{small}$ instead of $\theta_{large}$.

**Draft verification** After the draft tokens $\tilde{x}_{t+1}, \ldots, \tilde{x}_{t+m}$ are generated, they are fed into the LLM together with the context $s$. The LLM $\theta_{large}$ then calculates the conditional probabilities with a single forward pass:

$$p(x|s; \theta_{large}),$$
$$p(x|s, \tilde{x}_{t+1}; \theta_{large}),$$
$$\ldots$$
$$p(x|s, \tilde{x}_{t+1}, \ldots, \tilde{x}_{t+m-1}; \theta_{large}).$$

**Draft acceptance** Starting from the first generated tokens in the draft, the conditional probability derived from $\theta_{small}$ is compared with that of $\theta_{large}$. We can use modified rejection sampling to match the generated distribution with the LLM (Leviathan et al., 2023; Chen et al., 2023). For position $t + i$, we first sample a value $r$ from a uniform distribution $U(0, 1)$. If $r < \min\left(1, \frac{p(x|s,\tilde{x}_{t+1},\ldots,\tilde{x}_{t+i-1};\theta_{large})}{p(x|s,\tilde{x}_{t+1},\ldots,\tilde{x}_{t+i-1};\theta_{small})}\right)$, we accept the draft token $\tilde{x}_{t+i}$ and continue to validate the next token $\tilde{x}_{t+i+1}$. Otherwise, we stop the acceptance process, resample $x_{t+i}$, reject all the draft tokens after $x_{t+i}$, and move to the new speculative decoding process at the position $t + i + 1$.

### 3.2 Our Approach: REST

While in the classic speculative decoding, a smaller LM is used as the draft model, finding a high-quality draft model is usually challenging for several reasons: (1) For efficiency, the draft model needs to be lightweight enough to not introduce much overhead. (2) For quality, it needs to predict the LLM output accurately. (3) For system integration, it needs the same vocabulary set as the LLM, and an architecture that distributes easily with a similar configuration to the LLM (Chen et al., 2023). These challenges require carefully selecting or even training custom draft models for each new LLM.

In this paper, we solve the challenges differently. We develop a training-free approach to speculative decoding that can easily integrate with any new model to accelerate inference. Instead of relying on a parametric draft model, our method Retrieval-Based Speculative Decoding (REST) proposes using retrieval for draft construction. An

overview of REST is shown in Figure 1. In the following, we first describe constructing a datastore and operations on it, then demonstrate using it for draft construction and verification. Together, REST provides an efficient, high-quality, and easy-to-integrate solution for accelerating the inference of LLMs.

**Datastore construction** REST operates based on a pre-built datastore $D = \{(c_i, t_i)\}$, where $c_i$ represents a context and $t_i$ represents the corresponding continuation of the context $c_i$. Given a text/code corpus, we construct the datastore $D$ using the prefix context and the corresponding continuation at each position.

**Retrieving from the datastore** At inference, given a context $s = (x_1, \ldots, x_t)$, our objective is to construct the draft tokens which are likely the continuations of $s$. Different from vanilla speculative decoding that uses a small LM to construct the draft, we leverage the built datastore $D$ and directly retrieve draft tokens from the datastore. We first use the context $s$ to retrieve context-continuation pairs from the datastore $D$ and construct a set of continuation candidates $S$:

$$S = \{t_i \mid (c_i, t_i) \in \texttt{Retrieve}(D, s)\},$$

where $\texttt{Retrieve}(D, s)$ implements a retrieval process in the datastore $D$ that returns a set of context-continuation pairs $\{(c_i, t_i)\}$ by using $s$ as the query. It is straightforward to use recent dense retrieval models (Khandelwal et al., 2020; Karpukhin et al., 2020) to find contexts $c_i$ that are similar to $s$. However, using dense retrievers adds additional overhead during inference. We instead use a fast exact-match method to retrieve continuation candidates.

Our retrieval process is shown in Algorithm 1. We aim to find contexts in $D$ that match the longest suffix of $s$. We employ a greedy strategy and start from a pre-defined match length upper limit $n_{max}$[1]. For each suffix length $n$, we obtain the context $s$'s suffix with $n$ tokens $q$ (line 5), and obtain all the contexts $c_i$ that match $q$ as a suffix (line 6). If at least one context in $D$ matches the current $q$ (i.e., $S \neq \emptyset$), we return the corresponding context-continuation pairs as the retrieval result; otherwise we decrease the matching length $n$ by one and try to match a shorter suffix (line 7). We use a suffix array (Manber and Myers, 1993) to implement efficient exact match in datastore $D$ for a given $q$. The retrieval process leads to negligible overhead ($< 6\%$) in our experiments (see details in Section 5).

[1]We set $n_{max}$ as 16 in our experiments, as only few cases lead to a maximum match with more than 16 tokens.

**Algorithm 1** Exact-match based retrieval algorithm $\texttt{Retrieve}(D, s)$. We return context-continuation pairs in $D$ that match the longest suffix of $s$.

1: **Input:** Context $s$, datastore $D$, maximum suffix length $n_{max}$
2: Initialize $n \leftarrow n_{max}$
3: Initialize $S \leftarrow \emptyset$
4: **while** $S = \emptyset$ **do**
5: $\quad q \leftarrow \texttt{suffix}(s, n)$
6: $\quad S \leftarrow \{(c_i, t_i) \mid q = \texttt{suffix}(c_i, n)\} \subseteq D$
7: $\quad n \leftarrow n - 1$
8: **end while**
9: **return** $S$

**Draft construction from retrieved results** The retrieved result $S$ includes possible continuations of the context $s$. For each $t_i \in S$, any prefix of $t_i$ can serve as draft tokens of $s$ in the speculative decoding and be further verified by the LLM. Note that the retrieved set of continuation candidates $S$ can be large. It is not feasible to use all candidates as draft tokens and feed them into the LLM for verification. Here we present how we select high-quality draft tokens from the retrieved set $S$. A naive strategy is to sample a subset of sequences in $S$ as the draft tokens. However, this is suboptimal as the sampling contains randomness when $S$ is large.

We select draft tokens from the retrieved result $S$ using a Trie. In the Trie, the unique path from a node to the root node corresponds to a prefix of $t_i \in S$. For each node, we assign a weight reflecting the number (frequency) of the corresponding prefix that appears in the retrieved candidates. As shown in Algorithm 2, we first construct a Trie using all sequences in $S$, and the node weight is updated when a candidate $t_i$ is inserted into the Trie (lines 2-7). The Trie data structure allows us to prioritize tokens using the weights and select high-frequency prefixes (lines 8-15). In the practical implementation, we choose a subtree that contains the top $c$ nodes with the highest weights, which equals to selecting the top $c$ high-frequency prefixes as the draft sequences.

**Draft verification of REST** In REST, multiple draft sequences may be retrieved from the datastore.

**Algorithm 2** Draft sequences selection using Trie.

**Input:** Continuation Candidates $S$, hyperparameter $c$
Initialize Trie $T$
**for** each $t_i \in S$ **do**
  **for** each $prefix$ of $t_i$ **do**
    Insert $prefix$ into $T$ and update node weights
  **end for**
**end for**
Initialize empty priority queue $Q$ (Max Heap based on node weights)
**for** each $node$ in $T$ **do**
  Add $(node.prefix, node.weight)$ to $Q$
**end for**
**while** $Q.size > c$ **do**
  Pop the $prefix$ with the smallest weight from $Q$
**end while**
**return** $Q$

While one might initially approach the drafts independently and feed them into the LLM as distinct sequences in a batch, practical observations reveal that many drafts share common prefixes. This leads to redundant computation of Transformer layers on these shared prefixes across different sequences, resulting in a waste of computational power. To optimize the efficiency, we construct a pseudo sequence from the subtree using breadth-first search. By definition, it can be immediately obtained that each draft constitutes a sub-sequence of this pseudo sequence, and any shared prefix appears only once. To correctly execute LLM on this pseudo sequence, we implement a carefully designed attention mask in each attention layer, ensuring that the computation of each token precisely reflects its dependencies in the original draft sequence. This attention strategy is also known as tree attention (Cai et al., 2023; Miao et al., 2023; Spector and Re, 2023).

**Draft acceptance of REST** We adopt a similar acceptance strategy compared to the original speculative decoding. By feeding the drafts into LLM, we obtain the conditional distribution at each position given by $\theta_{large}$ and then check the correctness of the draft token. All correct tokens from the start will be accepted, and the draft tokens after the first mistake will be rejected.

**Comparison with existing approaches** Although REST follows a schema similar to that of speculative decoding, it offers significant advantages over existing approaches. Current speculative decoding methods rely on a high-quality small model to generate draft tokens (Leviathan et al., 2023; Chen et al., 2023). Such methods must strike a balance between a small size and strong predictive power, while also matching the vocabulary of the base model. Moreover, they require additional GPU memory and introduce complexity during inference. In contrast, REST directly retrieves draft tokens from a datastore, which can be easily integrated with language models of any size, vocabulary, or architecture. Different from Stern et al. (2018) and Cai et al. (2023) which train specialized modules to create a draft model, REST eliminates the need for any additional training steps and can serve as a plug-and-play solution of efficient decoding across different models. Furthermore, the effectiveness of REST is affected by the quality of retrieval results. This opens up the opportunities to further enhance REST by using a better/larger datastore or an advanced retrieval model. We also note that in addition to using REST directly, it is possible to combine REST with the vanilla speculative decoding. This combination can enhance the generation speed of the small LM. We leave this for future work.

## 4 Experiments

### 4.1 Experimental Setup

**Sampling strategies** We implement two sampling mechanisms: greedy sampling and nucleus sampling (Holtzman et al., 2019) for the LLM. Greedy sampling selects the token with the highest probability at each step. Nucleus sampling, also known as top-$p$ sampling, generates tokens by sampling from the most probable tokens in the model's predicted distribution until their cumulative probability reaches the threshold $p$. It is worth noting that under our approach, we only accept draft tokens if they match the tokens sampled from the LLM. As a result, the sequences produced using REST are identical to those generated by standard autoregressive generation.

**Datasets and models** We conduct experiments on two datasets: HumanEval (Chen et al., 2021) and MT-Bench (Zheng et al., 2023). HumanEval is a dataset that includes 164 human-written Python programming problems. The goal for the models is to generate code solutions using provided

| Benchmark | Model | Method | *Mean Token Time*(↓) | Speedup(↑) |
|---|---|---|---|---|
| HumanEval (1 shot) | CodeLlama 7B | Autoregressive (Greedy) | 27.89 ms/token | 1× |
| | CodeLlama 7B | Speculative (Greedy) | 15.90 ms/token | 1.75× |
| | CodeLlama 7B | REST (Greedy) | 11.82 ms/token | 2.36× |
| | CodeLlama 13B | Autoregressive (Greedy) | 44.32 ms/token | 1× |
| | CodeLlama 13B | Speculative (Greedy) | 19.39 ms/token | 2.29× |
| | CodeLlama 13B | REST (Greedy) | 19.53 ms/token | 2.27× |
| HumanEval (10 shot) | CodeLlama 7B | Autoregressive (Nucleus) | 27.99 ms/token | 1× |
| | CodeLlama 7B | Speculative (Nucleus) | 18.83 ms/token | 1.49× |
| | CodeLlama 7B | REST (Nucleus) | 13.18 ms/token | 2.12× |
| | CodeLlama 13B | Autoregressive (Nucleus) | 44.46 ms/token | 1× |
| | CodeLlama 13B | Speculative (Nucleus) | 22.68 ms/token | 1.96× |
| | CodeLlama 13B | REST (Nucleus) | 20.47 ms/token | 2.17× |
| MT-Bench | Vicuna 7B | Autoregressive (Greedy) | 25.48 ms/token | 1× |
| | Vicuna 7B | Speculative (Greedy) | 19.44 ms/token | 1.31× |
| | Vicuna 7B | REST (Greedy) | 15.12 ms/token | 1.69× |
| | Vicuna 13B | Autoregressive (Greedy) | 44.30 ms/token | 1× |
| | Vicuna 13B | Speculative (Greedy) | 29.80 ms/token | 1.49× |
| | Vicuna 13B | REST (Greedy) | 25.08 ms/token | 1.77× |
| MT-Bench | Vicuna 7B | Autoregressive (Nucleus) | 25.93 ms/token | 1× |
| | Vicuna 7B | Speculative(Nucleus) | 20.65 ms/token | 1.26× |
| | Vicuna 7B | REST(Nucleus) | 16.02 ms/token | 1.62× |
| | Vicuna 13B | Autoregressive (Nucleus) | 44.32 ms/token | 1× |
| | Vicuna 13B | Speculative (Nucleus) | 31.78 ms/token | 1.39× |
| | Vicuna 13B | REST (Nucleus) | 25.92 ms/token | 1.71× |

Table 1: Speed on HumanEval and MT-Bench with standard autoregressive decoding, speculative decoding and REST. The temperature is set to 0.8 and the top-$p$ to 0.95 for nucleus sampling in HumanEval. For MT-Bench, the settings are 0.7 for temperature and 0.8 for top-$p$. For speculative decoding, we conduct experiments using different numbers of draft tokens and different small LMs and record the best results (detailed results can be found in Appendix A). All the experiments are conducted on a single NVIDIA A6000 GPU and 96 CPU cores with a batch size of 1.

docstrings as prompts. On the other hand, MT-Bench contains 80 multi-turn questions designed to emulate real-world multi-turn dialogues. We compare the generation speed of standard autoregressive generation with REST, focusing on both the HumanEval and MT-Bench datasets. For HumanEval, we perform 1-shot evaluation for greedy sampling and 10-shot evaluation for nucleus sampling and employ the CodeLlama (Rozière et al., 2023). While for MT-Bench, we perform 1-shot evaluation for both greedy sampling and nucleus sampling and utilize Vicuna (Chiang et al., 2023). We test both the 7B and 13B configurations of CodeLlama and Vicuna, with a maximum generation limit of 512 tokens and 1024 tokens, respectively. All experiments are conducted on a single NVIDIA A6000 GPU and 96 CPU cores. All results are averaged across three different runs.

**Hyperparameters** When performing exact match in the datastore, the starting context suffix length, $n_{max}$, is set to 16, and is progressively reduced by one until we find matching contexts in the datastore. The length of each retrieved continuation candidate denoted as $m$, is truncated to 10. Empirical results from Medusa (Cai et al., 2023) suggest 64 draft tokens to be an optimal computation configuration. Hence, we limit the maximum number of selected draft tokens in the constructed Trie to 64, designated as $c$.

**Metrics** The first metric we use is *Mean Token Time*, which is the average generation time of one token for the LLM. Another metric, *Mean Generated Length*, is calculated as the ratio of the length of the generated tokens to the number of forward steps taken by the original LLM. Formally, if $L$ denotes the length of the generated tokens and $F$

represents the number of forward steps, the *Mean Generated Length*, $M$, is given by:

$$M = \frac{L}{F}.$$

Note that the *Mean Generated Length* ($M$) acts as the upper limit of the speedup that REST can achieve, ignoring the overhead for retrieving and constructing draft tokens.

**Datastores** For CodeLlama, we construct a datastore using a portion of the Python pretraining code from The Stack (Kocetkov et al., 2022). This dataset comprises approximately 2.7M Python code samples and results in a datastore with a size of 27GB. On the other hand, for Vicuna, we construct a datastore using data derived from UltraChat (Ding et al., 2023). This dataset consists of around 774K conversations from ChatGPT[2], yielding a datastore with a size of 12GB.

**Baseline** We implement speculative decoding (Leviathan et al., 2023; Chen et al., 2023) as the baseline for comparison. For the small draft LMs, we test a variety of model sizes, including Llama 68M and Llama 160M trained by Miao et al. (2023), TinyLlama 1.1B and TinyLlama-Chat 1.1B trained by Zhang et al. (2023). We also test different numbers of draft tokens ranging from 1 to 15 (performance degrades when larger than 15).

### 4.2 Main Results

Table 1 compares the generation speed of REST with the speed of the standard autoregressive decoding and speculative decoding.

Regarding generation speed, REST demonstrates a significant speed enhancement compared to standard autoregressive decoding and speculative decoding, achieving $2.16\times$ to $2.36\times$ increase for CodeLlama in the HumanEval benchmark. The MT-Bench benchmark also reveals a speedup for Vicuna when using our method, with a factor ranging from $1.62\times$ to $1.77\times$. These empirical results lend weight to the effectiveness of our method for speeding up the generation process of LLMs. Note that the speedup of nucleus sampling is not as good as that of greedy sampling. We speculate that this drop in performance is caused by the randomness introduced by nucleus sampling. Since speculative decoding may achieve better results with a more powerful draft LM that aligns with the LLM, we do not claim that REST can outperform speculative decoding under all circumstances. Yet, REST undoubtedly provides a potent and straightforward approach for faster inference of LLMs.

[2]https://chat.openai.com/

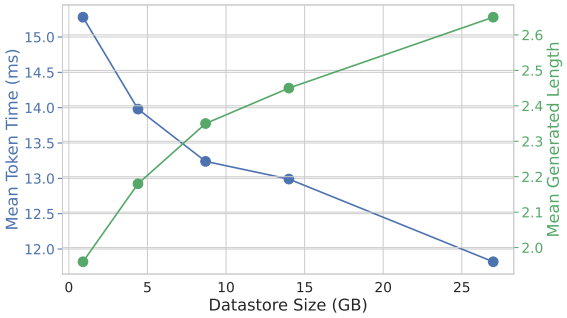


Figure 2: Generation speed of REST with different sizes of the datastore (CodeLlama 7B on HumanEval).

Another intriguing observation that emerges from these results is the domain-dependent nature of the speed improvements. This characteristic has also been noted in other methods like speculative decoding (Chen et al., 2023) and Medusa (Cai et al., 2023). Specifically, the speedup achieved with REST is significantly greater in the HumanEval benchmark than in the MT-Bench benchmark, suggesting that the effectiveness of REST may vary depending on the specific domain.

Additionally, it is important to note that the average time (divided by the total number of tokens) required for retrieval (which includes the time taken to construct the Trie) is less than 1 ms. This time is very small and can, for all practical purposes, be considered negligible. This negligible retrieval time further underscores the efficiency of REST.

## 5 Ablation Study

To gain a deeper understanding of our method, we conduct a series of ablation studies and analyses focused on each individual component. More ablation studies can be found in Appendix B.

**Effect of the datastore size** Increasing the size of the datastore is an effective strategy for enhancing the accuracy of retrieved draft tokens in the Trie, which in turn can significantly boost generation speed. In Table 2, we show that as the datastore size increases, both the *Mean Generated Length* and *Mean Token Time* correspondingly improve. However, it's important to note that the speedup growth is not as pronounced as that of the *Mean Generated Length*. This discrepancy could be at-

| Method | Datastore Size | Retrieval Time | $M$ | *Mean Token Time*(↓) | Speedup(↑) |
|---|---|---|---|---|---|
| Baseline(Greedy) | - | - | 1 | 27.89 ms/token | 1× |
| REST(Greedy) | 0.9 GB | 0.2 ms | 1.96 | 15.28 ms/token | 1.83× |
| REST(Greedy) | 4.4 GB | 0.5 ms | 2.18 | 13.98 ms/token | 1.99× |
| REST(Greedy) | 8.7 GB | 0.6 ms | 2.35 | 13.24 ms/token | 2.11× |
| REST(Greedy) | 14 GB | 0.6 ms | 2.45 | 12.99 ms/token | 2.15× |
| REST(Greedy) | 27 GB | 0.7 ms | 2.65 | 11.82 ms/token | 2.36× |

Table 2: Generation speed with different datastore sizes (CodeLlama 7B with greedy sampling on HumanEval). The datastores are all constructed from the Python pretraining code from the Stack (Kocetkov et al., 2022).

| Selecting Methods | $M$(↑) | *Mean Token Time* (↓) |
|---|---|---|
| Random(Greedy) | 2.51 | 12.80 |
| Trie(Greedy) | **2.65** | **11.82** |
| Random(Nucleus) | 2.44 | 14.19 |
| Trie(Nucleus) | **2.57** | **13.18** |

Table 3: Generation speed with different selecting methods of draft tokens (CodeLlama 7B with greedy sampling on HumanEval).

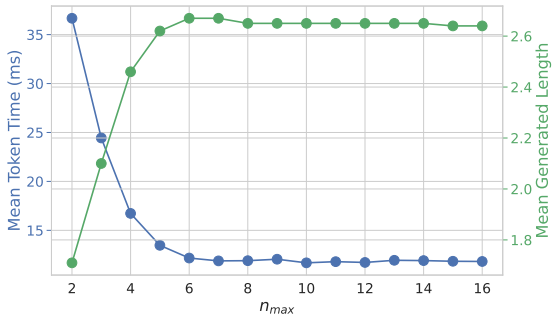


Figure 3: Generation speed of REST with different maximum suffix length $n_{max}$ (CodeLlama 7B with greedy sampling on HumanEval).

tributed to the overhead of getting draft tokens. We assume that in industry applications, there will be ample disk storage to build a large datastore and ample CPU cores for fast retrieval. We also visualize the trend of scaling the retrieval datastore size in Figure 2. From this, we can infer that there is still potential to achieve even faster speeds with a larger datastore.

**Effect of draft token selecting strategies** We compare selecting draft tokens in the Trie with randomly sampling retrieved continuation candidates as draft tokens. For an equitable comparison, we employ a random sampling technique to sample at most eight sequences from all the retrieved candidates. Furthermore, each sequence is truncated to a maximum length of 8. This results in a maximum number of 64 draft tokens, corresponding to the maximum number of selected draft tokens from the Trie. The data presented in Table 3 indicates that selecting draft tokens from the Trie, as opposed to employing a random sampling approach, enhances the performance.

**Effect of the choice of the maximum suffix length** We vary the value of $n_{max}$ to test the generation speed of REST. The outcomes of this study are depicted in Figure 3. An interesting observation is that when the value of $n_{max}$ is set to less than 6, there is a substantial increase in the generation time. Conversely, when $n_{max}$ exceeds 6, the generation speed remains consistently high and appears to be largely unaffected by further changes to the $n_{max}$ value. Hence, in practice, there is no substantial need to expend excessive efforts in selecting the precise optimal value of $n_{max}$.

## 6 Conclusion

In this work, we propose REST: retrieval-based speculative decoding. Instead of requiring a small LM, REST employs a datastore for retrieving and employing draft tokens. We construct a Trie to select the most probable draft tokens. REST is not only straightforward to implement but also easily integrates into the generation processes of any existing language models without necessitating additional training. We would like to explore large-scale retrieval in the next step. For situations where disk storage is limited, we will also explore methods of minimizing the size of the datastore without compromising performance.

## Limitations

The limitations of our work are as follows:

- Despite the plug-and-play nature of REST, it is important to acknowledge that the performance of REST is directly influenced by the accuracy and completeness of the datastore. For improved alignment with the LLM, it might be advantageous to consider constructing datastores from content generated by the LLM itself.

- Lack of in-context abilities. For instance, the challenge of retrieving personalized variable names in code generation—a task that inherently requires understanding context—raises an interesting question: How can we empower retrieval methodologies to effectively deal with such complexities?

## References


Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, and Tri Dao. 2023. Medusa: Simple framework for accelerating llm generation with multiple decoding heads. https://github.com/FasterDecoding/Medusa.

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*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing GPT-4 with 90%* ChatGPT quality.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics (NAACL)*.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. *arXiv preprint arXiv:2305.14233*.

Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. Optq: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. In *International Conference on Learning Representations (ICLR)*.

Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph Naor, and Daniel Soudry. 2021. Accelerated sparse neural training: A provable and efficient method to find n: m transposable masks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations (ICLR)*.

Denis Kocetkov, Raymond Li, LI Jia, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, et al. 2022. The stack: 3 tb of permissively licensed source code. *Transactions on Machine Learning Research*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Symposium on Operating Systems Principles (SOSP)*.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning (ICML)*.

Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. 2023. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Udi Manber and Gene Myers. 1993. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948.

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*.

Gunho Park, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. 2022. nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu.

Benjamin Spector and Chris Re. 2023. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623*.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

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*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning (ICML)*.

Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Inference with reference: Lossless acceleration of large language models. *arXiv preprint arXiv:2304.04487*.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. In *International Conference on Learning Representations (ICLR)*.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2023. Tinyllama.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*.

# A Detailed Results of Speculative Decoding

For the small draft LMs, we test a variety of model sizes, including Llama 68M and Llama 160M trained by Miao et al. (2023), TinyLlama 1.1B[3] and TinyLlama-Chat 1.1B[4] trained by Zhang et al. (2023). We also test different numbers of draft tokens from 1 to 15. Since Leviathan et al. (2023) and Chen et al. (2023) didn't release their code, we use an open-source reproduction code[5] and additionally use `torch.compile` to accelerate the generation speed of the small draft LMs.

## A.1 Results on CodeLlama 7B

**Greedy Sampling** The generation speed of speculative decoding on CodeLlama 7B with greedy sampling is shown in Figure 4. From the figure, we can see that the best setting is to use Llama 68M with 4 draft tokens, resulting in 15.90 ms/token. So, we report 15.90 ms/token in Table 1

**Nucleus Sampling** The generation speed of speculative decoding on CodeLlama 7B with nucleus sampling is shown in Figure 5. From the figure, we can see that the best setting is to use TinyLlama 1.1B with 4 draft tokens, resulting in 18.83 ms/token. So, we report 18.83 ms/token in Table 1.

## A.2 Results on CodeLlama 13B

**Greedy Sampling** The generation speed of speculative decoding on CodeLlama 13B with greedy sampling is shown in Figure 6. From the figure, we can see that the best setting is to use TinyLlama 1.1B with 10 draft tokens, resulting in 19.39 ms/token. So, we report 19.39 ms/token in Table 1

**Nucleus Sampling** The generation speed of speculative decoding on CodeLlama 13B with nucleus sampling is shown in Figure 7. From the figure, we can see that the best setting is to use TinyLlama 1.1B with 6 draft tokens, resulting in 22.68 ms/token. So, we report 22.68 ms/token in Table 1

## A.3 Results on Vicuna 7B

**Greedy Sampling** The generation speed of speculative decoding on Vicuna 7B with greedy sampling is shown in Figure 8. From the figure, we can see that the best setting is to use Llama 68M with

[3]The TinyLlama-1.1B-intermediate-step-955k-2T version.
[4]The TinyLlama-1.1B-Chat-V0.4 version.
[5]https://github.com/feifeibear/LLMSpeculativeSampling

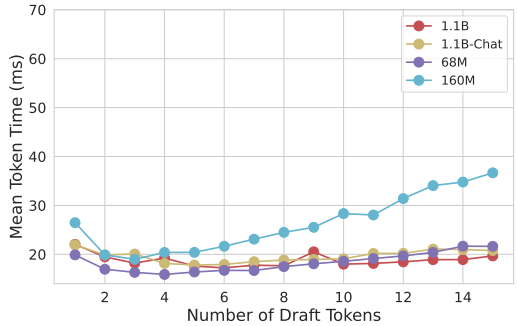


Figure 4: Generation speed of speculative decoding with different draft LMs and numbers of draft tokens (CodeLlama 7B with greedy sampling on HumanEval).

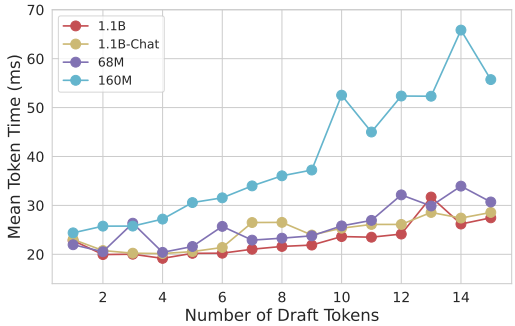


Figure 5: Generation speed of speculative decoding with different draft LMs and numbers of draft tokens (CodeLlama 7B with nucleus sampling on HumanEval).

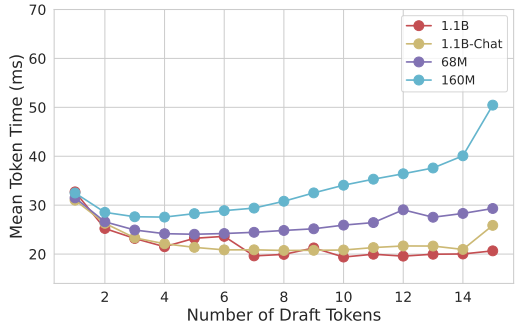


Figure 6: Generation speed of speculative decoding with different draft LMs and numbers of draft tokens (CodeLlama 13B with greedy sampling on HumanEval).

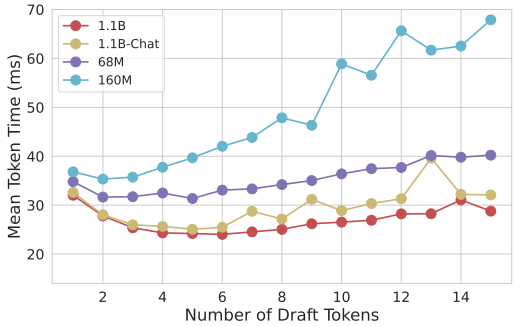


Figure 7: Generation speed of speculative decoding with different draft LMs and numbers of draft tokens (CodeLlama 13B with nucleus sampling on HumanEval).

3 draft tokens, resulting in 19.44 ms/token. So, we report 19.44 ms/token in Table 1

**Nucleus Sampling** The generation speed of speculative decoding on Vicuna 7B with nucleus sampling is shown in Figure 9. From the figure, we can see that the best setting is to use Llama 68M with 3 draft tokens, resulting in 20.65 ms/token. So, we report 20.65 ms/token in Table 1.

### A.4 Results on Vicuna 13B

**Greedy Sampling** The generation speed of speculative decoding on Vicuna 13B with greedy sampling is shown in Figure 10. From the figure, we can see that the best setting is to use Llama 68m with 3 draft tokens, resulting in 29.80 ms/token. So, we report 29.80 ms/token in Table 1

**Nucleus Sampling** The generation speed of speculative decoding on Vicuna 13B with nucleus sampling is shown in Figure 11. From the figure, we can see that the best setting is to use Llama 68M with 3 draft tokens, resulting in 31.78 ms/token. So, we report 31.78 ms/token in Table 1

## B Additional Ablation Studies

**Effect of the maximum number of draft tokens** Increasing the volume of draft tokens can potentially lead to a higher *Mean Generated Length* by the LLM. However, this also escalates the computational burden on GPUs during verification. As shown in Figure 12, an initial speed increase is observed as the maximum number of draft tokens increases. However, beyond the threshold of 48 draft tokens, the speed stabilizes to an average of approximately 11.75 ms per token. When the token count exceeds 200, it leads to a slowdown. Therefore, while it is possible to achieve similar speeds with a large maximum number of draft tokens, it's more efficient to limit the number to a smaller one to avoid unnecessary strain on GPUs.

**Visualization of matched suffix length** The distribution of matched suffix length of the context $s$ is illustrated in Figure 13. From this graphic, it is apparent that almost all cases contain a matched suffix length. Notably, shorter suffix lengths ranging from 2 to 9 comprise the majority of the matched cases,

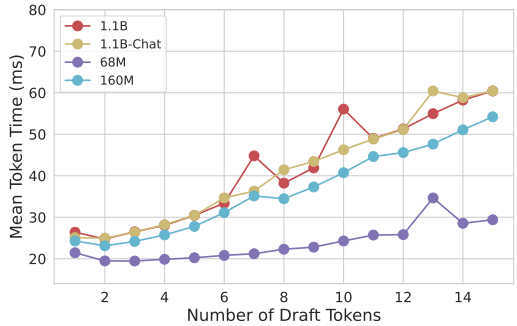


Figure 8: Generation speed of speculative decoding with different draft LMs and numbers of draft tokens (Vicuna 7B with greedy sampling on MT-Bench).

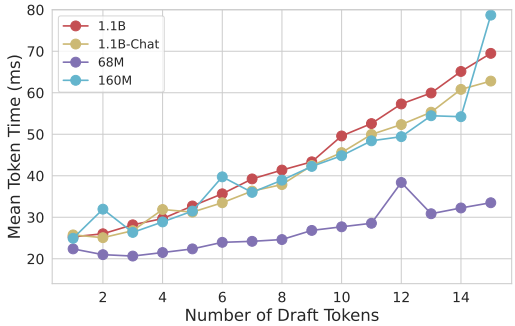


Figure 9: Generation speed of speculative decoding with different draft LMs and numbers of draft tokens (Vicuna 7B with nucleus sampling on MT-Bench).

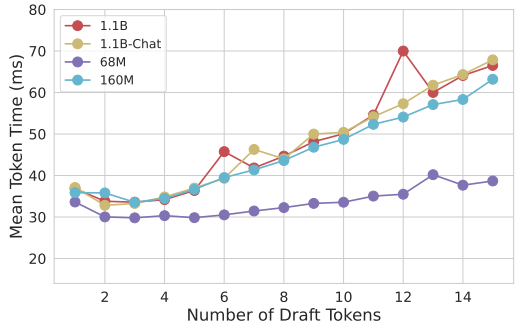


Figure 10: Generation speed of speculative decoding with different draft LMs and numbers of draft tokens (Vicuna 13B with greedy sampling on MT-Bench).

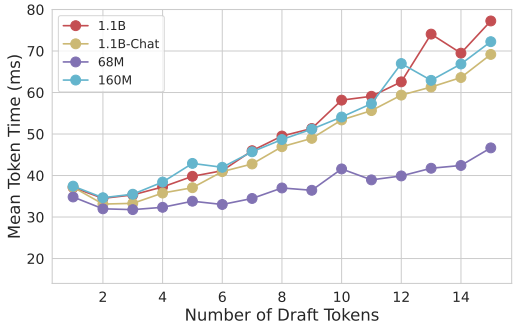


Figure 11: Generation speed of speculative decoding with different draft LMs and numbers of draft tokens (Vicuna 13B with nucleus sampling on MT-Bench).

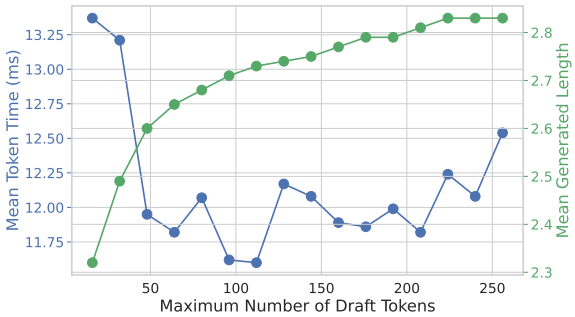


Figure 12: Generation speed of REST with different maximum numbers of selected draft tokens in the Trie (CodeLlama 7B with greedy sampling on the HumanEval).

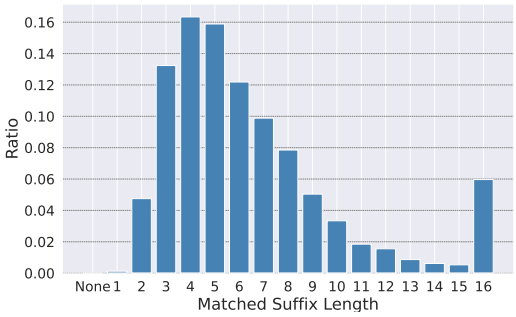


Figure 13: Distribution of matched suffix length (CodeLlama 7B with greedy decoding on HumanEval).

accounting for a substantial 85% of the total. In contrast, longer suffix lengths, which range from 10 to 16, constitute a minority, making up only 15% of the cases.