DReSD: Dense Retrieval for Speculative Decoding

Anonymous ACL submission

Abstract

Speculative decoding (SD) accelerates Large Language Model (LLM) generation by using an efficient draft model to propose the next few tokens, which are verified by the LLM in a single forward call, reducing latency while preserving its outputs. We focus on retrieval-based SD where the draft model retrieves the next tokens from a non-parametric datastore. Sparse retrieval (He et al., 2023, REST), which operates on the surface form of strings, is currently the dominant paradigm due to its simplicity and scalability. However, its effectiveness is limited due to the usage of short contexts and exact string matching. Instead, we introduce **D**ense **Re**trieval for **S**peculative **D**ecoding (DReSD), a novel framework that uses approximate nearest neighbour search with contextualised token embeddings to retrieve the most semantically relevant token sequences for SD. Extensive experiments show that DReSD achieves (on average) 87% higher acceptance rates, 65% longer accepted tokens and 19% faster generation speeds compared to sparse retrieval (REST).

1 Introduction

Generative transformers (Vaswani, 2017) are currently the dominant artificial intelligence paradigm with recent LLMs scaled to tens (or even hundreds) of billions of parameters (Brown et al., 2020; Liu et al., 2024; Dubey et al., 2024). In spite of their strong capabilities, the auto-regressive nature of generation requires a costly forward pass for each new token. Various solutions have been proposed to accelerate LLMs such as Flash Attention (Shah et al., 2024), Mixture of Experts (Fedus et al., 2022; Jacobs et al., 1991), Tensor Parallelism (Shoeybi et al., 2019), Linear Attention (Qin et al., 2024) and others. The focus of our work is Speculative Decoding (Leviathan et al., 2023), which seeks to accelerate generation by using an efficient draft model to propose the next few tokens that are verified in a single forward call of the LLM (Stern et al., 2018), guaranteeing its outputs. While several viable SD paradigms exist (Xia et al., 2024; Zhang et al., 2024a; Ryu and Kim, 2024), this work specifically focuses on retrieval-based SD where a draft model retrieves token sequences from a non-parametric datastore, usually a suffix array/automaton. Sparse retrieval has established itself as the dominant paradigm (He et al., 2023; Yang et al., 2023; Saxena, 2023; Hu et al., 2024), largely due to its simplicity and efficiency. However, we hypothesise that this approach suffers from limitations such as lower precision due to the use of short contexts and reduced recall due to exact string matching. As an alternative, we introduce **D**ense **Re**trieval for **SD** (DReSD) which seeks to overcome these limitations by utilising approximate nearest neighbour search with contextualised token representations. DReSD is a novel plug-and-play SD framework based on semantic similarity that shows significantly improved acceptance rates. Through extensive experimentation, we identify critical factors of dense retrieval and show how an optimal configuration can accelerate generation by up to 4.64x. Code and data available at `url.com`.

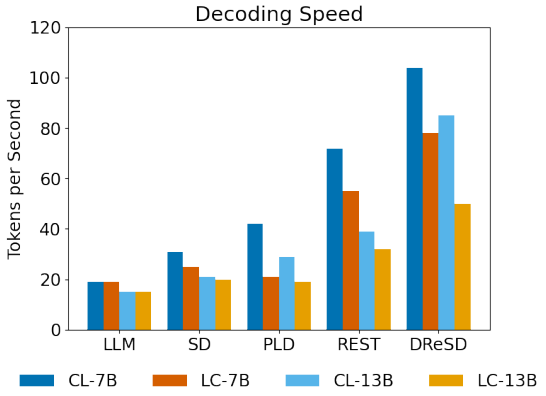

Figure 1: Fastest configurations for selected SD methods (greedy decoding), relative to auto-regressive generation (LLM), CL = CodeLlama, LC = Llama2-Chat.

Summary of Contributions: We conduct a detailed comparative analysis of sparse and dense retrieval for SD in order to identify the critical factors of effective dense retrieval. To address these, we propose a novel SD framework (for the first time, to our best knowledge) for easy LLM integration. Results show that DReSD achieves (on average, across all experiments) 87% higher acceptance rates, 65% longer accepted tokens and 19% faster generation compared to sparse retrieval.

2 Background

2.1 Speculative Decoding

Let x represent the input tokens $(x_1, x_2, ..., x_t)$ such as a prompt and any tokens generated up to time step t. Auto-regressive generation requires a full forward pass through the model $x_{t+1} = \text{LLM}(x)$ to decode every new token x_{t+1}, which is very resource-intensive for large LLMs. Therefore, a smaller draft model $\mathcal{M}_{\text{DRAFT}}$ efficiently proposes k next tokens $(x_{t+1}, x_{t+2}, ..., x_{t+k})$, denoted x_d, which can then be verified with a single forward call $x_v = \text{llm\_verify}(x_d)$. Verification only accepts tokens x_v that would have been generated by the LLM, irrespective of utilising SD. $\mathcal{M}_{\text{DRAFT}}$ can be a small LLM (Miao et al., 2023), a retrieval-based model (He et al., 2023), a subset of LLM's parameters called 'draft heads' (Cai et al., 2024; Li et al., 2024b; Ankner et al., 2024) or no auxiliary draft model at all, called 'self-drafting' (Mamou et al., 2024). Each paradigm has its trade-offs and the landscape is evolving rapidly (Xia et al., 2024; Zhang et al., 2024a; Ryu and Kim, 2024).

2.2 Retrieval-based Speculative Decoding

Since SD operates at the token level, it requires a continuous interaction between $\mathcal{M}_{\text{DRAFT}}$ and the LLM. In retrieval-based SD, $\mathcal{M}_{\text{DRAFT}}$ is represented by a non-parametric, training-free, static or dynamic datastore from which next token sequences are efficiently drafted and finally verified by the LLM. Retrieval-based SD can be broadly divided into sparse and dense retrieval.

2.2.1 Sparse Retrieval for SD

Sparse retrieval employs exact string matching[1] to retrieve k next tokens $(x_{t+1}, x_{t+2}, ..., x_{t+k})$ from the datastore, which contains a large body of pre-tokenized text similar to the target task(s), allowing for appropriate drafting. There are two types of sparse retrieval datastores for SD.

A static datastore keeps its content *unchanged during inference*. The most similar work (and our main baseline) is Retrieval-based Speculative Decoding (He et al., 2023, REST). REST matches the longest possible suffix of the current context x, a sequence of up to c tokens $(x_{t-c}, x_{t-c+1}, ..., x_t)$, to exact token sequences (suffixes) in the datastore to provide k draft candidates $(x_{t+1}, x_{t+2}, ..., x_{t+k})$ for LLM verification. The main limitation of exact string matching is that minor perturbations in x will result in a failure to retrieve useful candidate drafts.

A dynamic datastore keeps updating its content *continuously during inference* (Yang et al., 2023; Luo et al., 2024; Saxena, 2023), which means it benefits from recently generated token sequences that align well with the LLM, particularly for tasks with repetitive texts. Combinations of static and dynamic datastores are also possible (Hu et al., 2024). However, as the focus of our work is a systematic 'apples to apples' comparison of sparse and dense retrieval, these methods are not appropriate for a direct comparison with DReSD (or REST). We aim to explore (for the first time) the comparative efficacy of static datastores for the purpose of SD.

2.3 Dense Retrieval for SD

The key assumption behind DReSD is that semantic similarity of contextualised token embeddings should provide superior retrieval compared to exact string matching. Therefore, $\mathcal{M}_{\text{DRAFT}}$ is represented by a non-parametric datastore that employs *approximate nearest neighbour search* (Shrivastava and Li, 2014; Sun et al., 2023, ANNS) to match the (full) current context x to similar contexts in the datastore in order to draft the next tokens x_d for LLM verification. ANNS is a technique for finding the closest data point(s) for a given query in a possibly high-dimensional vector space (Karpukhin et al., 2020). Nearest Neighbour Speculative Decoding (Li et al., 2024a, NEST) is the only work using dense retrieval, to our best knowledge. However, its primary focus is *retrieval augmented fusion with attribution*, not SD. NEST relies on approximate verification to fuse the LLM and the retrieved knowledge, which means the LLM outputs are not guaranteed. Additionally, while NEST did not consider exact verification in their experiments, we can estimate from their results that minimal speed-ups would be achieved under that setting.

[1] https://en.wikipedia.org/wiki/Suffix_array or https://en.wikipedia.org/wiki/Suffix_automaton.

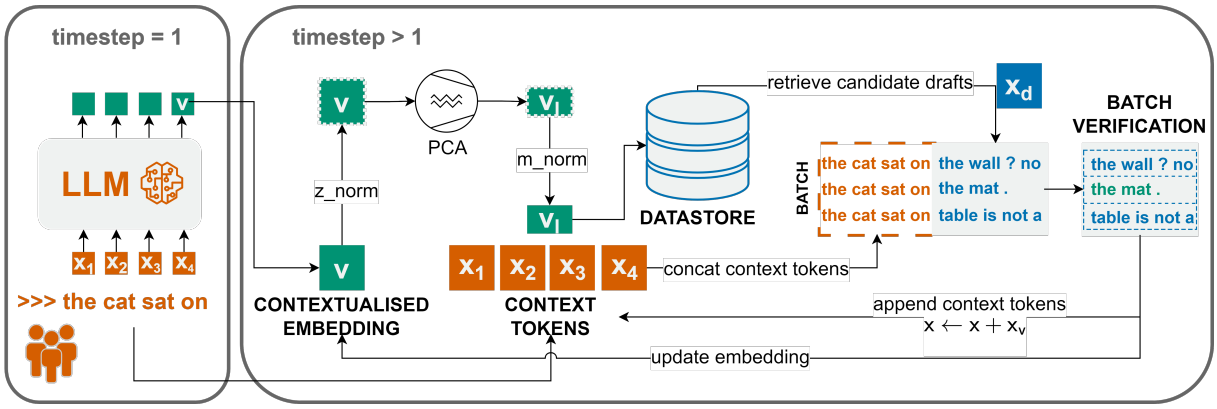

Figure 2: A flowchart of the DReSD framework.

3 DReSD

We now introduce **D**ense **Re**trieval for **S**peculative **D**ecoding, shown in Figure 2 and Algorithm 1. Focusing on the latter, the user prompt is tokenised in step 1 and embedded in step 2. Entering the loop (3), the embedding is normalised (4), then reduced (5) to optimise storage and compute requirements. After a second normalisation step (6), we query the datastore to retrieve the draft next tokens (7). They are verified by the LLM (8), returning the accepted token(s) and the embedding of the last accepted token. We append the accepted token(s) to the current context and begin a new iteration, which ends when we reach max length or the <EOS> token.

Algorithm 1 DReSD: An algorithmic overview.

1: $x \leftarrow \text{tokenizer}(\text{prompt})$
2: $v \leftarrow \text{LLM}(x)$ ▷ Section 3.1.
3: **while** not(EOS $\vee$ MAX_LEN) **do**
4: $v \leftarrow \text{z\_norm}(v)$ ▷ Section 3.2.
5: $v_l \leftarrow \text{PCA}(v)$ ▷ Section 3.3.
6: $v_l \leftarrow \text{m\_norm}(v_l)$ ▷ Section 3.4.
7: $x_d \leftarrow \mathcal{M}_{\text{DRAFT}}(v_l)$ ▷ Section 3.5.
8: $v, x_v \leftarrow \text{batch\_verify}(x_d)$ ▷ Section 3.6.
9: $x \leftarrow x + x_v$ ▷ Append x_v
10: **return** x

3.1 Token Embeddings

The initial step is to generate a contextualised token embedding $v \leftarrow \text{LLM}(x)$ to represent the current state of the LLM that will be used to retrieve candidates for the next tokens. In DReSD, v is the last hidden state before the language modelling head[2]. As in standard SD, LLM(x) will also generate the next token x_{t+1}, which we additionally use to filter retrieved candidate drafts. Even if all draft tokens are rejected, x_{t+1} ensures that each SD iteration produces *at least one valid token*.

[2]Alternative LLM components may be used for the current state representation but this is out of the scope of this work.

3.2 Z-scores Normalisation

Before we perform dimensionality reduction, we centre the empirical mean around 0 with a standard deviation of 1 to reduce the correlation between different embedding dimensions (Ethayarajh, 2019; Reimers and Gurevych, 2019), see Equation 1. We randomly sample ∼1 million (full size) token embeddings V from the datastore to estimate the mean and standard deviation for efficient inference.

$$v = \frac{v - \text{E}[V]}{\sqrt{\text{Var}[V] + \epsilon}} \quad (1)$$

3.3 Dimensionality Reduction

Using the full LLM hidden state v with thousands of dimensions is not scalable. As such, data compression and noise reduction are necessary steps for DReSD to reduce storage requirements and accelerate nearest neighbour search. Principal Component Analysis (Shlens, 2014, PCA) is a highly effective and algorithmically simple solution for this, allowing for efficient inference, too. We use PCA to transform v into a low-dimensional vector v_l that captures the largest variation in the data, using the first l principal components W_l by computing $v_l \leftarrow vW_l$. We fit the PCA model on the same ∼1 million token embeddings V from section 3.2.

3.4 Magnitude Normalisation

We further standardise the embedding v_l by scaling each to have a unit length of 1 using L_p normalisation over the last dimension (columns), see Eq.

2. This is a standard transformation required for effective (dot product) nearest neighbour search.

$$\mathsf{v_l} = \frac{\mathsf{v_l}}{\max(\|\mathsf{v_l}\|_2, \epsilon)} \quad (2)$$

3.5 Datastore

We utilise Scalable Nearest Neighbours[3](Guo et al., 2020) for approximate nearest neighbour search (time complexity $\mathsf{O}_{\log_n}$). The datastore $\mathcal{D}$ is formatted as a **key-value store** $f_{\mathcal{D}} : k \mapsto v$ where k is a **token embedding** $\mathsf{v_l^t}$ at time step t and v is a **sequence of the next** N **tokens** $(\mathsf{x_{t+1}}, ..., \mathsf{x_{t+N}})$, obtained from datasets similar to the target task(s). Cosine similarity is used as a standard distance metric, see Equation 3. The next token $\mathsf{x_{t+1}}$ obtained from step 3.1 is used to filter drafts that do not start with $\mathsf{x_{t+1}}$, further enhancing retrieval accuracy.

$$\mathcal{M}_{\mathsf{DRAFT}}(\mathsf{v_l}) = f_{\mathcal{D}}(\underset{\mathsf{v_l^t} \in \mathcal{D}}{\operatorname{argmax}} \, \mathsf{sim}(\mathsf{v_l}, \mathsf{v_l^t}))$$
$$\mathsf{sim}(\mathsf{v_l}, \mathsf{v_l^t}) = \frac{\mathsf{v_l} \cdot \mathsf{v_l^t}}{\max(\|\mathsf{v_l}\|_2 \cdot \|\mathsf{v_l^t}\|_2, \epsilon)} \quad (3)$$

3.6 Batch Verification

We use batch verification (Yang et al., 2024; Stewart et al., 2024) for all experiments, which generalises standard SD verification to multiple drafts, see Figure 3. Batch verification has shown benefits for SD, particularly at lower batch sizes (Ni et al., 2024; Zhang et al., 2024b). As this requires a forward call to the LLM, we extract the embedding v from the last accepted token of $\mathsf{x_v}$ to efficiently feed into the next iteration (step 8, Algorithm 1). Following our baseline, for nucleus and greedy generation, we first sample tokens conditioned on the draft sequences, then accept the longest sequence that **exactly matches the outputs** of the LLM.

draft1	15	2	66	75	0	0	0	0
draft2	15	2	66	40	31	2	0	0
draft3	15	75	21	12	98	13	8	2
draft4	20	33	75	13	7	1	0	0
draft5	20	33	75	0	0	0	0	0

Figure 3: An illustration of batch verification with 5 drafts (rows) with a length of 8 (columns). The EOS id (0 in this example) is used as padding. The green sequence is accepted, blue sequences are discarded.

[3] https://github.com/google-research/google-research/tree/master/scann

4 Experimental Setup

4.1 Models

We evaluate methods on LLMs from the Llama2 family (Touvron et al., 2023), courtesy of Huggingface transformers (Wolf, 2019). Specifically, we benchmark CodeLlama (7B and 13B), CodeLlama-Instruct (7B) and Llama2-Chat (7B and 13B). The $\mathcal{M}_{\mathsf{DRAFT}}$ for vanilla Speculative Decoding (with a small LLM drafter) features Llama-Chat-68M[4], fine-tuned from Llama-68M (Miao et al., 2023).

4.2 Datasets and Tasks

We test models on 100 randomly selected CodeAlpaca (Chaudhary, 2023) prompts, which include code generation, debugging, explanation and other code tasks. The datastore for this code assistant is built from EvolInstructCode (Luo et al., 2023), comprising ∼78K prompts with responses, truncated to 1,024 max tokens. We also evaluate on 80 MT-Bench[5] (Zheng et al., 2023) prompts (first turn specifically, due to the compute required for the number of experiments). The datastore for this general personal assistant is built from a random subset of 80K ('train-sft') UltraChat-200K[6] (Ding et al., 2023) examples, prompts and responses truncated to 1,024 max tokens, once again, first turn to limit the scope of the long, multi-turn conversations.

Models	EVOL	MRR	U-CHAT	MRR
OOD datastores (Sec. 4.2.1)				
CL-7B	30.9M	93.7	46.3M	97.5
CL-13B	30.9M	92.5	-	-
LC-7B	30.9M	93.6	46.3M	97.9
LC-13B	30.9M	93.7	-	-
CL-I-7B	30.9	93.9	46.3M	97.9
Sec. ID datastores (4.2.1)				
CL-7B	19.3M	87.6	-	-
CL-13B	19.3M	85.5	-	-
LC-7B	19M	90.3	56.8M	75.2
LC-13B	19M	90.4	-	-
CL-I-7B	-	-	57.2M	75.9

Table 1: Datastore sizes in tokens + corresponding MRR. EVOL = EvolInstructCode, U-CHAT = UltraChat.

[4] https://huggingface.co/Felladrin/Llama-68M-Chat-v1

[5] https://huggingface.co/datasets/HuggingFaceH4/mt_bench_prompts

[6] https://huggingface.co/datasets/HuggingFaceH4/ultrachat_200k

4.2.1 In-Distribution Data

The datasets used to populate the datastore are often generated by some version of ChatGPT[7] whose outputs are not necessarily representative of the target LLM (Llama2). That is, the datastore outputs are out-of-distribution (OOD) with respect to the LLM. As this divergence increases, the acceptance rates and decoding speeds are expected to decrease. To investigate the effect of in-distribution (ID) token sequences, we generate responses for each LLM and use those to populate the datastore. We refer to Llama2 responses as the **ID datastore** and ChatGPT responses as the **OOD datastore**.

4.3 Metrics

Hardware Dependent metrics are heavily influenced by the choice, availability and optimisation level of hardware components. Nevertheless, in order to provide indicative walltime improvements, we use **tokens-per-second** (abbreviated to TPS) as the standard metric, reporting the median of three runs. TPS is measured on a single NVIDIA V100 (32GB) GPU with 96 CPUs and 500GB of RAM.

Hardware Independent metrics are more appropriate for algorithmic comparisons that are independent of optimisation tricks and hardware quality. **Mean Acceptance Rate** (**MAR**) is the number of tokens drafted divided by the number of tokens accepted by the LLM. MAR is computed at the prompt level, then averaged over all prompts.

Retrieval Only We also conduct intrinsic evaluation to assess the quality of the nearest neighbour search. We use Mean Reciprocal Rank, shown in Equation 4, where rank_i is the position of the correct item and N is the number of embeddings in the datastore (a score of 1 equates to perfect retrieval[8]).

$$\text{MRR} = \frac{1}{\text{N}} \sum_{\text{i}=1}^{\text{N}} \frac{1}{\text{rank}_\text{i}} \tag{4}$$

5 Results

We provide reference metrics for auto-regressive decoding with the base LLM, vanilla speculative decoding (Leviathan et al., 2023) and Prompt Lookup Decoding (PLD), a dynamic retrieval method that uses the current input tokens for drafting (Saxena, 2023). Our main point of reference is REST (He et al., 2023), which is created from the same data as DReSD. For all experiments, we generate up to 128 new tokens per prompt.

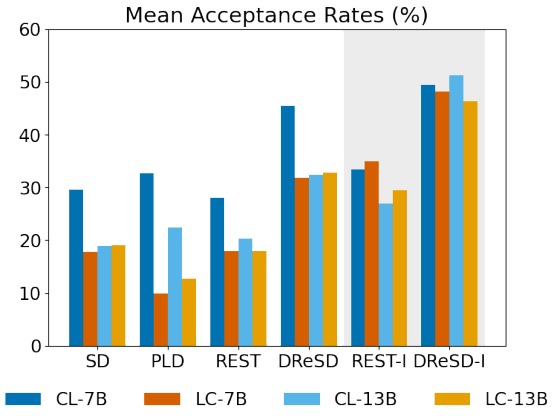

Figure 4: Mean Acceptance Rates (MAR) for the Code Assistant. Suffix "-I" denotes the ID datastore setting.

5.1 Mean Acceptance Rates

We first examine the average acceptance rates in a highly controlled setting where the draft lengths are as identical as possible. This is to test our core assumption that, all things being equal, dense retrieval would more accurately match the current context to useful sequences of next tokens in the datastore, relative to sparse retrieval. The hypothesis is confirmed in Figure 4 as significantly higher MAR for DReSD, on average 87% higher. This translates to fewer verification calls due to longer accepted drafts, on average 65% longer than REST.

5.2 Effective Dense Retrieval

Sparse retrieval libraries had been highly optimised over time[9], therefore, a high-performing datastore is a critical component of DReSD. It is imperative to maximise the algorithmic efficiency of DReSD to amortise the relatively higher cost of "vanilla" dense retrieval. Reducing the large dimensionality[10] of LLM hidden states while preserving the most informative features is a top priority. Figure 5 shows the cumulative explained variance ratio for the first 256 principal components from which we select 64 after a dimensionality ablation. Table 2 shows that retaining more than 64 principal components provides no meaningful improvement in downstream metrics. The pre-PCA

[7] https://platform.openai.com/docs/api-reference

[8] A presence of duplicate embeddings in a large datastore can lead to lower scores, even with near-perfect retrieval.

[9] https://github.com/FasterDecoding/REST

[10] https://pypi.org/project/torch-pca/

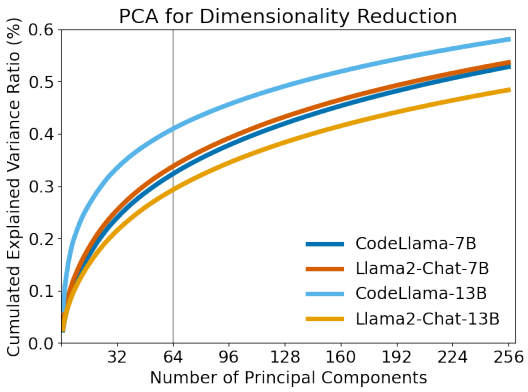

Figure 5: Cumulative Explained Variance Ratio for a 256-dimensional PCA model. We use the first 64 dims.

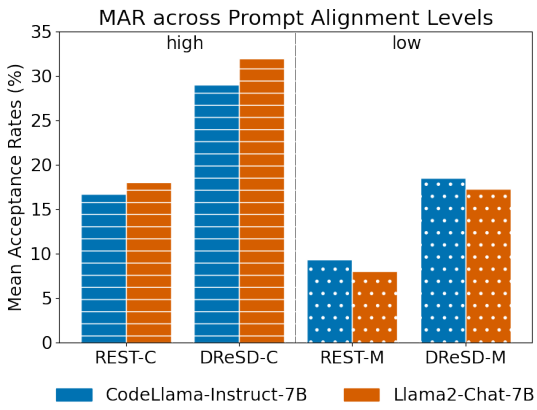

Figure 6: Mean Acceptance Rates with high (CodeAlpaca, "-C") & low (MT-Bench, "-M") prompt alignment.

(Equation 1) and post-PCA (Equation 2) normalisation steps are particularly important since the MRR scores sharply drop without these transformations. The most remarkable observation is that selecting just over 1% of the 4096 to 5120 dimensional LLM hidden state features is enough to capture between 30%-40% of explained variance in PCA, and achieves such strong retrieval performance (see MRR, in Table 1 and 2). There is a surprisingly high degree of redundancy in the LLM hidden state in terms of the minimum features required for effective dense retrieval, an important discovery for the feasibility of DReSD and any future work.

Metrics	32D	64D	96D	128D
MAR	20.1	22.5	22.8	23
Calls	37	34.6	34.7	34.5
TPS	32	35	36	35
MRR	**85.5**	**93.6**	**94.2**	**94.2**

Table 2: An ablation of PCA dimensionality reduction. 'Calls' = the average number of LLM verification calls.

5.3 Importance of Datastore Alignment

Another critical component of retrieval-based SD is *datastore alignment*, which we split into a) **prompt alignment**, b) **response alignment**, and c) **sampling alignment**. These *multiplicatively* influence overall effectiveness, which means that poor alignment in any of them can adversely impact performance. Let us examine why in more detail.

Prompt alignment is typically satisfied by populating the datastore with prompts highly related to the target task(s) such as code, maths or general question answering. After a brief qualitative assessment, this appears to have been reasonably satisfied for the code assistant, however, only to some degree for MT-Bench tasks. Despite strong response alignment (ID datastore) and retrieval (MRR, Table 1), the poor prompt alignment leads to lower acceptance rates (see Figure 6), relative to CodeAlpaca. Since this result reflects prior findings (He et al., 2023), we think that choosing a more prompt-aligned dataset will result in faster decoding. In any case, dense retrieval (DReSD) outperforms sparse retrieval (REST) by ∼90% in Figure 6.

Response alignment is the similarity of outputs between the LLM and the model(s) that generated the datastore, i.e. draft sequences with a low probability under the LLM lead to high rejection rates and slow decoding speeds, regardless of the capabilities of the model(s) that generated the datastore. A qualitative comparison of Llama2 (ID datastore) and ChatGPT (OOD datastore) responses reveals significant differences in writing styles, knowledge depth and response lengths. The effects of these differences can be observed in Figures 1, 4 and 9 by comparing methods with and without the suffix "-I", showing that the ID datastore has a strong positive effect in all cases. For example, MAR increased by ∼70% on average for CodeAlpaca with an ID datastore despite being ∼40% smaller than the OOD datastore, emphasising the importance of response alignment over sheer data quantity.

Sampling alignment refers to the similarity of hyperparameters with which the datastore content was generated and the sampling hyperparameters at inference time. For instance, in Figure 9 (left), the best speed-ups were achieved with greedy sampling

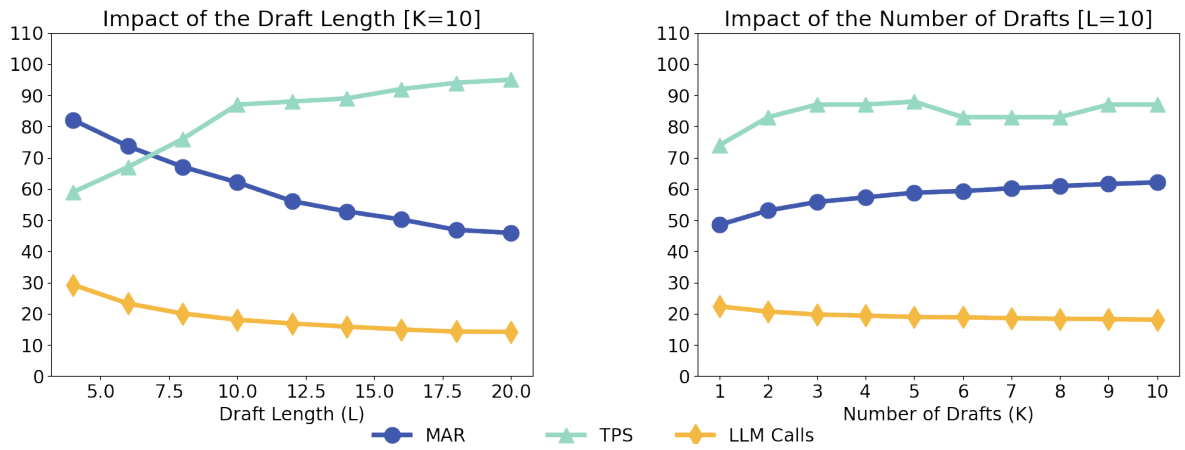

Figure 7: Investigating the impact of increasing draft lengths (left) and the number of drafts (right) on MAR, TPS and LLM verification calls, using a greedy datastore with greedy generation on code assistant tasks (CodeAlpaca).

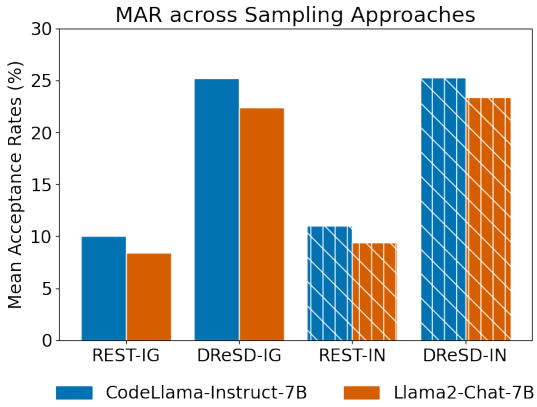

Figure 8: MAR for "-I" = ID datastore (nucleus generation) with "-N" = nucleus and "-G" greedy sampling.

and a 'greedy' datastore. In contrast, nucleus sampling (temperature=0.7, p=0.95[11]) with a 'greedy' datastore resulted in lower speed-ups (Figure 9, right). In another ablation (Figure 8), the ID datastore was generated with nucleus sampling, using 3 responses of up to 128 tokens per prompt (see Table 1 for final datastore sizes). There was no significant difference between MAR scores with greedy or nucleus sampling this time. In summary, the more permissive the sampling parameters are, e.g. a high temperature, the greater the LLM's expressivity that will need to be covered by the datastore. This is particularly the case for open-ended tasks such as creative writing where the number of 'correct' responses is usually much greater than in a coding or maths task, for example. In contrast, low temperature sampling can achieve fast decoding speeds with a much smaller 'greedy' datastore.

[11]Nucleus hyperparameters are the same in all experiments.

5.4 Optimal Draft Shape

The final critical factor for achieving high decoding performance is the shape of the draft because each additional token sent for verification increases the cost of the LLM forward pass (see Figure 3 for an illustration of a 5 x 8 draft). While REST outputs drafts that encode shallow and wide trees (10 x 6, on average, cannot be altered), DReSD can modify this shape to potentially achieve higher speed-ups. For instance, when the draft shape is matched to REST, DReSD outperforms it by 15% to 28% with nucleus sampling and 10% to 29% with greedy decoding (TPS, Figure 9). Switching to the ID datastore (suffix "-I"), the margins are slightly smaller, 2% to 15% for nucleus sampling and 4% to 20% for greedy generation. However, the optimal REST draft (wide and shallow) is not necessarily optimal for DReSD (narrow and deep).

Therefore, we investigate the optimal draft shape using an ablation of **the number and the length of drafts** with a 'greedy' ID datastore and greedy decoding, shown in Figure 7. The number of drafts is fixed to 10 for 'Impact of Draft Length' (left) and the length of each draft is fixed to 10 tokens for 'Impact of the Number of Drafts' (right). Based on the findings of this ablation, the best configurations are DReSD-IBN (10 x 10), yielding between 10% and 23% speed-up relative to REST for nucleus sampling and DReSD-IBG, (3 x 20), yielding 42% to 218% faster speeds for greedy decoding. As a general principle: 1) we should increase the draft length for higher acceptance rates and vice versa, and 2) include fewer drafts for greedy generation but more drafts (shorter) for nucleus sampling.

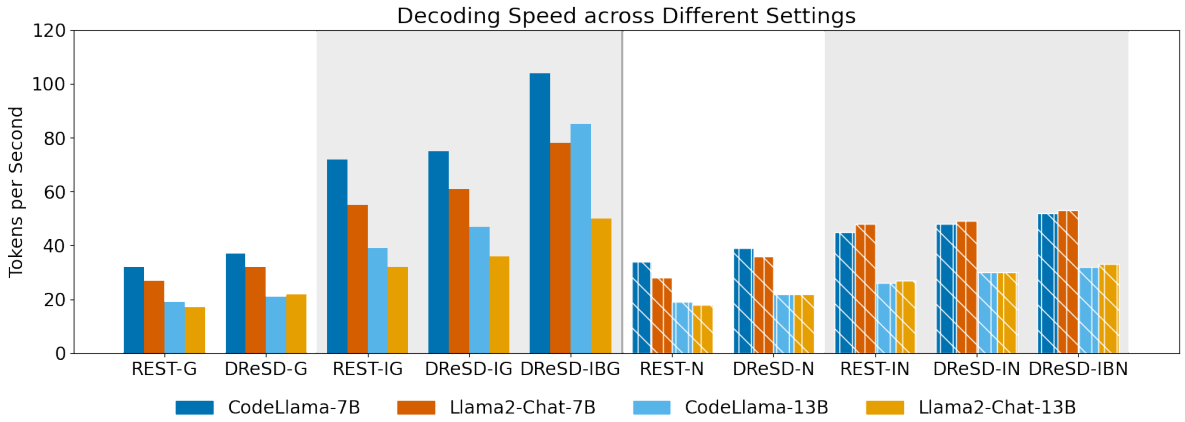

Figure 9: TPS metrics for a selection of LLMs and configs: "-G" = greedy decoding, "-I" = uses ID datastore (greedy generation), "-N" = nucleus sampling, "-B" = our best setup (see section 5.5), LLM = auto-regressive generation.

Models	SD	PLD	REST	DReSD
CL-7B	1.63x	2.21x	3.79x	**5.47x**
LC-7B	1.32x	1.11x	2.89x	**4.11x**
CL-13B	1.40x	1.93x	2.60x	**5.67x**
LC-13B	1.33x	1.27x	2.13x	**3.33x**
Average	1.42x	1.63x	2.85x	**4.64x**

Table 3: Average speed-ups relative to auto-regressive generation on the code assistant tasks (CodeAlpaca).

5.5 Walltime Improvements

Table 3 provides the speed-up ratios for vanilla SD with Llama-Chat-68M, Prompt Lookup Decoding (dynamic sparse retrieval), REST (static sparse retrieval) and DReSD (static dense retrieval). The averages show that every SD method accelerates standard auto-regressive LLM generation by at least 40%, with higher speed-ups observed for the 'smaller' 7B models. PLD is most effective for very repetitive outputs, which can be a property of the task and/or the model. For example, CodeLlama (not instruction-tuned), has a tendency to produce repetitive texts, particularly on code assistant tasks. This is why the effectiveness of PLD drops sharply for instruction-tuned models. Our best configuration for REST, shown in Figure 9 with suffix "-IG", gives an average 2.85x speed-up. DReSD using drafts of shape (3 x 20 tokens) and the ID datastore (suffix "-IBG") averaged a remarkable 4.64x improvement over auto-regressive decoding. Figure 1 provides a visual summary of these configurations in relation to other baseline methods. The speed-ups on MT-Bench are relatively more modest, up to ∼1.52x, due to poor prompt alignment discussed earlier (5.3). Still, DReSD significantly outperformed REST, which achieved only ∼1.15x speed-up with the same dataset(s). This confirms our hypothesis that dense retrieval is the superior search paradigm for speculative decoding.

6 Conclusions

We have presented a comparative analysis of dense and sparse retrieval for speculative decoding in order to identify and overcome the limitations of the dominant (sparse) paradigm. To address these, we have introduced DReSD, **D**ense **Re**trieval for **S**peculative **D**ecoding, a novel framework that retrieves candidate drafts from a non-parametric datastore based on semantic similarity (via approximate nearest neighbour search) instead of exact string matching. DReSD introduces a scalable and effective dense retrieval protocol that can easily integrate into modern LLMs. Exhaustive comparisons using several model and task configurations have demonstrated that DReSD achieves (on average across all settings) 87% higher acceptance rates, 65% longer accepted tokens and 19% faster generation speeds compared to sparse retrieval (REST). This is enabled by three critical factors: a) a fast and accurate dense retrieval via dimensionality reduction and dual normalisation of LLM embeddings, b) a careful datastore alignment (particularly the ID datastore) with high acceptance rates, longer drafts and fewer LLM calls, c) an optimal draft shape explored via careful ablations that enabled up to 4.64x average speed-ups over baseline auto-regressive generation. We hope that our findings will enable new retrieval-based SD methodologies in the future.

7 Limitations

We acknowledge that sparse retrieval methods typically have lower storage and preprocessing requirements, which can make their adoption more feasible for low compute budgets compared to our proposed methodology. Simultaneously, we recognise the lack of software and/or hardware optimisation for DReSD that could fully realise its potential in terms of faster decoding speeds compared to the highly optimised sparse retrieval libraries. Finally, related work has shown that combinations of dynamic and static retrieval may bring complementary strengths to the overall approach, therefore, DReSD could be extended to such hybrid speculative decoding version in future work.

References

Zachary Ankner, Rishab Parthasarathy, Aniruddha Nrusimha, Christopher Rinard, Jonathan Ragan-Kelley, and William Brandon. 2024. Hydra: Sequentially-dependent draft heads for medusa decoding. *arXiv preprint arXiv:2402.05109*.

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. *Advances in neural information processing systems*, 33:1877–1901.

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*.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

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. *Preprint*, arXiv:2305.14233.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Kawin Ethayarajh. 2019. How contextual are contextualized word representations? Comparing the geometry of BERT, ELMo, and GPT-2 embeddings. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 55–65, Hong Kong, China. Association for Computational Linguistics.

William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason D Lee, and Di He. 2023. Rest: Retrieval-based speculative decoding. *arXiv preprint arXiv:2311.08252*.

Yuxuan Hu, Ke Wang, Jing Zhang, Cuiping Li, and Hong Chen. 2024. Sam decoding: Speculative decoding via suffix automaton. *arXiv preprint arXiv:2411.10666*.

Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

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

Minghan Li, Xilun Chen, Ari Holtzman, Beidi Chen, Jimmy Lin, Wen-tau Yih, and Xi Victoria Lin. 2024a. Nearest neighbor speculative decoding for llm generation and attribution. *arXiv preprint arXiv:2405.19325*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024b. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Xianzhen Luo, Yixuan Wang, Qingfu Zhu, Zhiming Zhang, Xuanyu Zhang, Qing Yang, Dongliang Xu, and Wanxiang Che. 2024. Turning trash into treasure: Accelerating inference of large language models with token recycling. *arXiv preprint arXiv:2408.08696*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Jonathan Mamou, Oren Pereg, Daniel Korat, Moshe Berchansky, Nadav Timor, Moshe Wasserblat, and Roy Schwartz. 2024. Accelerating speculative decoding using dynamic speculation length. *arXiv preprint arXiv:2405.04304*.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. 2023. Specinfer: Accelerating generative large language model serving with tree-based speculative inference and verification. *arXiv preprint arXiv:2305.09781*.

Yunsheng Ni, Chuanjian Liu, Yehui Tang, Kai Han, and Yunhe Wang. 2024. Ems-sd: Efficient multi-sample speculative decoding for accelerating large language models. *arXiv preprint arXiv:2405.07542*.

Zhen Qin, Weigao Sun, Dong Li, Xuyang Shen, Weixuan Sun, and Yiran Zhong. 2024. Lightning attention-2: A free lunch for handling unlimited sequence lengths in large language models. *arXiv preprint arXiv:2401.04658*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Hyun Ryu and Eric Kim. 2024. Closer look at efficient inference methods: A survey of speculative decoding. *arXiv preprint arXiv:2411.13157*.

Apoorv Saxena. 2023. Prompt lookup decoding.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*.

Jonathon Shlens. 2014. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

Anshumali Shrivastava and Ping Li. 2014. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). *Advances in neural information processing systems*, 27.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31.

Lawrence Stewart, Matthew Trager, Sujan Kumar Gonugondla, and Stefano Soatto. 2024. The n-grammys: Accelerating autoregressive inference with learning-free batched speculation. *arXiv preprint arXiv:2411.03786*.

Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. Soar: Improved indexing for approximate nearest neighbor search. In *Neural Information Processing Systems*.

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*.

A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*.

T Wolf. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. 2024. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *arXiv preprint arXiv:2401.07851*.

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*.

Sen Yang, Shujian Huang, Xinyu Dai, and Jiajun Chen. 2024. Multi-candidate speculative decoding. *arXiv preprint arXiv:2401.06706*.

Chen Zhang, Zhuorui Liu, and Dawei Song. 2024a. Beyond the speculative game: A survey of speculative execution in large language models. *arXiv preprint arXiv:2404.14897*.

Zhihao Zhang, Alan Zhu, Lijie Yang, Yihua Xu, Lanting Li, Phitchaya Mangpo Phothilimthana, and Zhihao Jia. 2024b. Accelerating retrieval-augmented language model serving with speculation. *arXiv preprint arXiv:2401.14021*.

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. *Advances in Neural Information Processing Systems*, 36:46595–46623.

A Appendix

The following tables contain the results used to generate the bar charts in the paper.

TPS	CodeLlama-7B	Llama2-Chat-7B	CodeLlama-13B	Llama2-Chat-13B
LLM	19	19	15	15
SD	31	25	21	20
PLD	42	21	29	19
REST	72	55	39	32
DReSD	104	78	85	50

Table 4: Fastest configurations for a selection of methods (greedy decoding), relative to auto-regressive generation (LLM), CL = CodeLlama, LC = Llama2-Chat.

MAR	CodeLlama-7B	Llama2-Chat-7B	CodeLlama-13B	Llama2-Chat-13B
SD	29.6	17.8	18.9	19.1
PLD	32.7	10.0	22.4	12.7
REST	28.1	18.0	20.3	17.9
DReSD	45.5	31.9	32.4	32.9
REST-I	33.4	35.0	27.0	29.5
DReSD-I	49.4	48.1	51.3	46.3

Table 5: Mean Acceptance Rates (MAR) for the Code Assistant. Suffix "-I" denotes the ID datastore setting.

TPS	CodeLlama-7B	Llama2-Chat-7B	CodeLlama-13B	Llama2-Chat-13B
REST-G	32	27	19	17
DReSD-G	37	32	21	22
REST-IG	72	55	39	32
DReSD-IG	75	61	47	36
DReSD-IBG	104	78	85	50
REST-N	34	28	19	18
DReSD-N	39	36	22	22
REST-IN	45	48	26	27
DReSD-IN	48	49	30	30
DReSD-IBN	52	53	32	33

Table 6: Tokens-per-second for a selection of LLMs and configurations: "-G" = greedy decoding, "-I" = uses the ID datastore, "-N" = nucleus sampling, "-B" = our best setup (see section 5.5), LLM = auto-regressive generation.

MAR	CodeLlama-Instruct-7B	Llama2-Chat-7B
REST-C	16.7	18.0
REST-M	9.3	8.0
DReSD-C	29.0	31.9
DReSD-M	18.5	17.3

Table 7: Mean Acceptance Rates with high (CodeAlpaca, "-C") & low (MT-Bench, "-M") prompt alignment.

MAR	CodeLlama-Instruct-7B	Llama2-Chat-7B
REST-IG	10.0	8.4
DReSD-IG	25.2	22.4
REST-IN	11.0	9.4
DReSD-IN	25.3	23.4

Table 8: Mean Accepted Rates with an ID datastore "-I", nucleus sampling "-N" and greedy "-G" decoding.