SuffixDecoding: Extreme Speculative Decoding for Emerging AI Applications

Gabriele Oliaro[§,†] **Zhihao Jia**[†] **Daniel Campos**[§] **Aurick Qiao**[§]
[§]Snowflake AI Research [†]Carnegie Mellon University
{goliaro,zhihaoj2}@cs.cmu.edu,
{daniel.campos,aurick.qiao}@snowflake.com

Abstract

Speculative decoding is widely adopted to reduce latency in large language model (LLM) inference by leveraging smaller draft models capable of handling diverse user tasks. However, emerging AI applications, such as LLM-based agents, present unique workload characteristics: instead of diverse independent requests, agentic frameworks typically submit repetitive inference requests, such as multi-agent pipelines performing similar subtasks or self-refinement loops iteratively enhancing outputs. These workloads result in long and highly predictable sequences, which current speculative decoding methods do not effectively exploit. To address this gap, we introduce *SuffixDecoding*, a novel method that utilizes efficient suffix trees to cache long token sequences from prompts and previous outputs. By adaptively speculating more tokens when acceptance likelihood is high and fewer when it is low, SuffixDecoding effectively exploits opportunities for longer speculations while conserving computation when those opportunities are limited. Evaluations on agentic benchmarks, including SWE-Bench and Text-to-SQL, demonstrate that SuffixDecoding achieves speedups of up to 5.3×, outperforming state-of-the-art methods – 2.8× faster than model-based approaches like EAGLE-2/3 and 1.9× faster than model-free approaches such as Token Recycling. SuffixDecoding is open-sourced at `https://github.com/snowflakedb/ArcticInference`.

1 Introduction

Large language models (LLMs) are foundational to a new generation of agentic AI applications, such as automated coding assistants [Wang et al., 2025, Xia et al., 2024a, Yang et al., 2024], multi-agent workflows [Wang et al., 2024a, Chen et al., 2024a, Zhang et al., 2024b], and retrieval-based search systems [Zheng et al., 2025, Wang et al., 2024d, Gao et al., 2024b]. Unlike basic chatbots, these agentic workloads typically issue repetitive and predictable inference requests. For instance, each agent in a multi-agent system repeatedly perform similar inference tasks, and reasoning loops [Wang et al., 2023a, Madaan et al., 2023] regenerate similar token sequences to improve their final outputs. Despite this predictable repetition, existing inference methods often fail to fully exploit recurring patterns, leaving latency as a significant bottleneck in agent-driven applications.

A popular strategy for mitigating inference latency is *speculative decoding* [Leviathan et al., 2023, Chen et al., 2023, Miao et al., 2024, Cai et al., 2024, Lin et al., 2024, Zhang et al., 2024a]. While an LLM can only generate one token per forward pass, it can verify *multiple* tokens. Leveraging this phenomenon, speculative decoding methods use small “draft” models or additional decoding heads to predict multiple candidate tokens, which the LLM then verifies in parallel.

To efficiently handle the long repetitions common in agent-driven applications, speculative decoding methods must satisfy two critical requirements. First, they need to generate draft tokens rapidly and with minimal overhead, enabling maximal exploitation of long speculation lengths. Second, they must do so *adaptively*—only generating more draft tokens when acceptance likelihood is high and fewer tokens when acceptance likelihood is low, to prevent verification from becoming a bottleneck.

However, existing speculative decoding approaches fall short in meeting these dual requirements. Model-based methods can use significant GPU time when speculating long sequences, and can incur memory contention and kernel-level transitions [Chen et al., 2024b, Li et al., 2024] that must be managed carefully. Conversely, existing model-free approaches, such as prompt-lookup decoding (PLD) [Saxena, 2023], achieve low overhead and rapid token generation, but typically lack adaptivity. These methods speculate a fixed number of tokens irrespective of acceptance likelihood, leading to wasted computational resources on verifying long and improbable draft sequences.

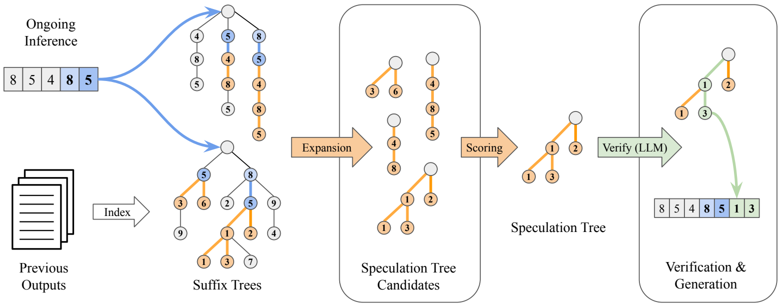


Figure 1: Overview of SuffixDecoding's algorithm. Two suffix trees track ongoing inference (top-left) and previous outputs (bottom-left). SuffixDecoding uses these trees to find matching patterns based on recently generated tokens. It constructs a speculation tree (middle) by selecting the most likely continuations, scoring them based on frequency statistics. Finally, the best candidate is verified by the LLM in a single forward pass (right), with accepted tokens (shown in green) being added to the output and used for the next round of speculation.

To address these limitations, we introduce *SuffixDecoding* (illustrated in Fig 1), a novel model-free speculative decoding method specifically designed for repetitive, agent-driven workloads. SuffixDecoding leverages efficient suffix trees to cache long token sequences from prompts and previous outputs. Each node represents a token, and paths from the root encode previously observed subsequences. This structure enables rapid pattern matching: given recently generated tokens, SuffixDecoding efficiently identifies possible continuations based on prior occurrences, generating draft tokens extremely quickly—on the order of 20 microseconds per token—without incurring any GPU overhead.

At each inference step, SuffixDecoding adaptively limits its number of draft tokens based on the length of the pattern match, and uses frequency-based statistics captured within the suffix trees to score and select the best speculation candidate. Longer pattern matches enable confident speculation of longer token sequences, maximizing its effectiveness on agentic workloads, while shorter pattern matches trigger conservative speculation to avoid computational waste. Moreover, SuffixDecoding can seamlessly integrate with existing model-based speculative decoding methods. This flexibility enables a hybrid approach that leverages suffix-tree-based speculation for repetitive, predictable agentic workloads, while exploiting the strengths of model-based speculation methods for open-ended conversational tasks, thus achieving the best of both worlds.

We evaluate SuffixDecoding on two practical agent-driven workloads: SWE-Bench, an LLM-based software engineering benchmark, and AgenticSQL, a proprietary multi-agent pipeline application for SQL generation. We compare with state-of-the-art model-based and model-free speculative decoding methods using Spec-Bench [Xia et al., 2024b], showing up to 2.8× faster decoding than EAGLE-2/3 [Li et al., 2025], and 1.9× faster decoding than Token Recycling [Luo et al., 2024]. For SWE-Bench, we also measured the comprehensive, end-to-end task completion time—including prompt prefilling, token generation, and execution of external actions—and demonstrate speculative speedups of up to 4.5×. These results highlight that SuffixDecoding substantially reduces latency for real-world agentic applications, addressing a critical bottleneck in practical inference scenarios.

2 Background

LLM Inference. LLM inference involves two stages: given a prompt $x_{\text{prompt}} = (x_1, x_2, \ldots, x_m)$, the LLM first processes the prompt in parallel (prefill), then sequentially generates new tokens (decode), with each token $x_{t>m}$ conditioned on previously generated tokens:

$$x_{t+1} = Sample(x|x_{1,\ldots,t}).$$

For example, in greedy sampling, the highest-probability token from the model's predicted distribution is selected iteratively until a stopping condition, such as reaching an end-of-text token or maximum length. Since each token depends on preceding outputs, token generation is inherently sequential, requiring a separate forward pass per generated token. This sequentiality limits inference throughput and can underutilize parallel hardware accelerators such as GPUs or TPUs.

Speculative Decoding. Speculative decoding [Leviathan et al., 2023] accelerates inference by generating multiple candidate tokens quickly using a lightweight model, which can then be verified in parallel by the primary LLM. The basic method has two core steps:

1. *Speculation*: A smaller "draft" model rapidly produces speculative tokens $x_{\text{spec}} = (x_{t+1}, \ldots, x_{t+n})$ based on the existing token prefix $x_{<t}$.
2. *Verification*: The LLM verifies the draft tokens in parallel, accepting tokens up to the first discrepancy and discarding the rest.

This approach reduces bottlenecks by shifting computation from sequential generation to parallel verification. However, the draft model, despite its smaller size, still requires compute resources and can add orchestration complexity in deployment. This limitation motivates recent *model-free* speculative decoding methods, such as Prompt Lookup Decoding [Saxena, 2023], which sources draft tokens directly from the prompt, and Token Recycling [Luo et al., 2024], which enhances this approach using an adjacency matrix and tree speculation [Miao et al., 2024].

Agentic AI Algorithms. Agentic applications often structure complex tasks as sequences or compositions of LLM calls, issuing multiple inference requests per task. These requests tend to generate long and repetitive token sub-sequences due to this structure. For example:

Self-consistency [Wang et al., 2023a] samples multiple reasoning paths in parallel before selecting a final answer based on consensus. While each path is independently sampled, they all start from the same prompt and often share similar reasoning steps or chain-of-thought sequences.

Self-refinement [Madaan et al., 2023], commonly used in coding agents, improves initial outputs by iteratively identifying and fixing errors. Each iteration typically revises only a small portion of the text—such as a few lines of code—while preserving the majority of the surrounding content.

Multi-agent workflows [Khot et al., 2023] decompose tasks into modular subtasks performed by specialized agents (e.g., retrieval, reasoning, synthesis). Because each agent handles a narrowly scoped function, their outputs can exhibit highly repetitive structures.

These patterns result in a high degree of redundancy across LLM calls, presenting opportunities for speculative decoding strategies that can exploit long repeated token sequences for greater acceleration.

3 SuffixDecoding

The goal of *SuffixDecoding* is to enable fast, adaptive speculative decoding over long sequences, particularly suited for agentic applications where repeated inference calls often contain highly predictable and overlapping token sequences. In such settings, long stretches of output can be accurately predicted from prior and ongoing requests.

To fully exploit these opportunities, SuffixDecoding must address two key challenges. First, it must support *fast generation of speculative sequences*—including long continuations—without relying on draft models or expensive token-by-token prediction. Second, it must be *adaptive* to the current prediction context: aggressively speculating long continuations only when they are likely to be accepted, and speculating shorter sequences when uncertain to avoid wasted verification compute.

To support fast speculation, SuffixDecoding builds a *suffix tree* [Weiner, 1973] over the tokens in the current and prior requests, and uses the suffix tree to generate speculative tokens. The root node of the tree represents the beginning of a suffix of any token sequence stored in the tree, each child of a node represents a specific token which is a possible continuation from its that node, and the path from the root to each node represents a distinct subsequence.

For each request, we consider speculating token sequences from (1) the prompt and output of that request, and (2) the outputs of prior requests. Doing so captures the source of output token repetition from many agentic algorithms, including self-consistency, self-refinement, and multi-agent pipelines.

SuffixDecoding leverages suffix trees to perform fast pattern matching and find possible continuations of token sequences. Suppose the prompt and output tokens of an ongoing inference is $x_{1:t}$. Consider a suffix $x_{t-p+1:t}$ of length p, which we will refer to as the *pattern* sequence. We walk the suffix tree starting from the root node, and at each step taking the child that corresponds to token x_{t-p+i}. If no such child exists, then the pattern is not found and SuffixDecoding reverts to standard non-speculative decoding. Otherwise, after p steps, we arrive at a node whose descending paths are the possible continuations of the pattern sequence.

Although this procedure can quickly find a (potentially large) set of candidate sequences, verifying all of them in speculative decoding may be cost-prohibitive. Instead, SuffixDecoding builds a much smaller and more likely speculation tree through a greedy expansion and scoring procedure, and uses this smaller tree in tree-based speculation. An overall illustration of SuffixDecoding is shown in Fig. 1, which we detail in the rest of this section.

Suffix Tree Construction. Building the suffix tree and updating it as part of an online inference service involves two stages. First, the previous inference outputs can be added to the tree in a single offline processing step (e.g. from historical logs), or online during inference serving after each inference request completes. Second, the current ongoing prompt and out tokens are added online as new requests are received and as each new token is generated.

In reality, we found it convenient to maintain two different suffix trees: a *global* tree for the previously generated outputs, and a separate *per-request* tree for the current ongoing inference request. This circumvents the complexities and overheads due to synchronizing the suffix tree updates from multiple concurrent requests. The global tree can be constructed offline in $O(n)$ time, while the per-request tree can be efficiently constructed and updated online [Ukkonen, 1995].

Although suffix trees are memory-efficient at $O(n)$ space, the global tree can still become large when there are many previous outputs. However, they only require CPU memory, which is typically plentiful and under-utilized in LLM serving scenarios. For example, AWS `p5.48xlarge` are often used for LLM serving and have 2TB of main memory, which is easily enough to support a suffix tree over millions of historical outputs and billions of tokens.

Speculation Tree Expansion. Given a pattern sequence $x_{t-p+1:t}$ of an ongoing inference $x_{1:t}$, SuffixDecoding can quickly find a node N_p in the global or per-request suffix tree whose descending paths are the possible continuations of the pattern sequence. To select a smaller more likely sub-tree that is of a more practical size for speculative verification, we start with the single node N_p and grow a sub-tree greedily by expanding one leaf node at a time.

In particular, we define:

$$C(N) = \frac{\texttt{COUNT}(N)}{\sum_{M \in \texttt{CHILDREN}(\texttt{PARENT}(N))} \texttt{COUNT}(M)}$$

$$D(N) = \begin{cases} D(\texttt{PARENT}(N)) \times C(N), & \text{if } N \neq N_p \\ 1, & \text{otherwise} \end{cases},$$

where $\texttt{COUNT}(N)$ is the number of occurrences of node N in the reference corpus, which can be computed when constructing the suffix tree. Starting with the single node N_p in our speculation sub-tree, we consider all children of all of its leaf nodes, and add the node N with the highest $D(N)$. This process is repeated until the sub-tree reaches a predetermined size limit, `MAX_SPEC`.

Intuitively, $C(N)$ estimates the probability that $\texttt{TOKEN}(N)$ would be the next observed token in a subsequence $\texttt{TOKEN}(N_p), \ldots, \texttt{TOKEN}(\texttt{PARENT}(N))$, and $D(N)$ estimates the probability that $\texttt{TOKEN}(N)$

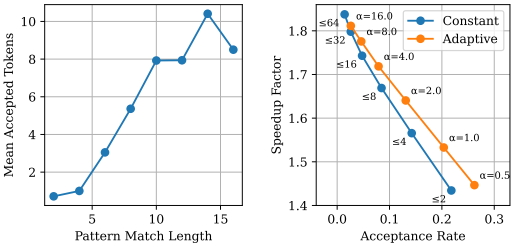
would be ultimately accepted by the speculative tree verification, assuming the output tokens follow historical patterns. Thus, SuffixDecoding builds the speculation tree by greedily adding leaf nodes that it believes to be the most likely to be accepted during verification.

Algorithm 1 Speculation Tree Generation

```
function ExpandSpeculationTree(N_p, MAX_SPEC)
    Input: Suffix tree node $N_p$, MAX_SPEC
    Initialize $T \leftarrow \{N_p\}$
    while $|T| <$ MAX_SPEC do
        $N \leftarrow \arg\max_{N \in \text{CHILDREN(LEAVES}(T))} D(N)$
        $T \leftarrow T \cup \{N\}$
    end while
    return $T$
end function
function MatchPattern(S, x_{1:t}, p)
    Input: Suffix tree $S$, sequence $x_{1:t}$, length $p$
    Initialize $N_p \leftarrow \text{ROOT}(S)$
    for $i = 1$ to $p$ do
        if NO_CHILD$(N_p, x_{t-p+i})$ then
            return $\emptyset$
        end if
        $N_p \leftarrow \text{CHILD}(N_p, x_{t-p+i})$
    end for
    return $N_p$
end function
function GenerateCandidateTree(S_g, S_r, x_{1:t}, $\alpha$, P)
    Input: Global suffix tree $S_g$, prompt suffix tree $S_r$, sequence $x_{1:t}$, max spec factor $\alpha$, max pattern size $P$
    Initialize $T_{\text{best}} \leftarrow \emptyset$, $\text{SCORE}_{\text{best}} \leftarrow 0$
    for $S$ in $\{S_g, S_r\}$ do
        for $p = 1$ to $P$ do
            $N \leftarrow \text{MatchPattern}(S, x_{1:t}, p)$
            $T \leftarrow \text{ExpandSpeculationTree}(N, \alpha p)$
            if $\text{SCORE}(T) > \text{SCORE}_{\text{best}}$ then
                $T_{\text{best}} \leftarrow T$
                $\text{SCORE}_{\text{best}} \leftarrow \text{SCORE}(T)$
            end if
        end for
    end for
    return $T_{\text{best}}$
end function
```

Adaptive Speculation Lengths. While the procedure above allows SuffixDecoding to cache and quickly speculate long token sequences based on empirical probability estimates, it also needs a mechanism for *adaptively* controlling the number of tokens it speculates. SuffixDecoding achieves this by dynamically adjusting `MAX_SPEC`. Low values mean fewer but more likely tokens would be chosen for speculation, while higher values mean more but less likely tokens would be chosen. If too low, then the speedup from speculation can be limited, and if too high, then compute may be wasted on verifying unlikely tokens.

To guide how to adaptively set `MAX_SPEC`, we observed that the number of accepted tokens in practice typically increases with longer pattern sequence lengths p (Fig. 2a). Thus, we define



(a) Avg accepted tokens vs pattern match length.

(b) Using constant vs adaptive `MAX_SPEC`.

Figure 2: (a) the mean number of accepted tokens increases with the length of the pattern match, which motivates `MAX_SPEC` $= \alpha p$. (b) shows that this choice achieves a better trade-off between acceptance rate and speculative speedup.

`MAX_SPEC` adaptively as

$$\texttt{MAX\_SPEC}(p) = \alpha p,$$

where α is a user-defined max speculation factor. Fig. 2b shows that setting `MAX_SPEC` adaptively according to the pattern length results in a better trade-off between acceptance rate and speculative speedup. In practice, we found that $\alpha \in [1, 4]$ works well for agentic applications.

Speculation Tree Scoring. So far, we have discussed how to obtain a speculation tree given a suffix tree and a pattern length p. However, SuffixDecoding maintains two suffix trees, the global suffix tree and the per-request suffix tree, each with many choices for p. To obtain just a single speculation tree, we build speculation trees for both the global suffix tree and the per-request suffix tree, and for a range of values of p. Then, a single speculation tree is selected according to a scoring function:

$$\texttt{SCORE}(T_{spec}) = \sum_{N \in T_{spec}} D(N).$$

Intuitively, if $D(N)$ estimates the probability that node N in a speculation tree T_{spec} would be accepted, then $\texttt{SCORE}(T_{spec})$ estimates the expected number of accepted tokens. SuffixDecoding then selects the T_{spec} with the highest `SCORE` as the final speculation tree to be verified. The end-to-end candidate generation from speculation tree expansion to scoring is described in Alg. 1.

***Hybrid* Suffix Speculative Decoding.** Lastly, we find that $\texttt{SCORE}(T_{spec})$ can be used to dynamically decide between using SuffixDecoding or falling back to a model-based speculation method, which is useful for practical scenarios when the workload can be mixed between agentic and more diverse applications. Specifically, for each decoding iteration, we always speculate using SuffixDecoding first. If $\texttt{SCORE}(T_{spec}) > \tau$, where τ is a configurable threshold, then SuffixDecoding's draft tokens are used. Otherwise, we use a fall-back speculation method, such as EAGLE-3 [Li et al., 2025].

4 Evaluation

4.1 Evaluation Methodology

Baseline Comparisons. We compare with both model-based and model-free speculative decoding methods using Spec-Bench [Xia et al., 2024b]. (1) *EAGLE-2* and *EAGLE-3* [Li et al., 2025], state-of-the-art model-based speculators, (2), *Prompt-Lookup Decoding (PLD)* [Saxena, 2023], a simple model-free speculator based on ngram-matching, and (3) *Token Recycling* [Luo et al., 2024], a more recent model-free speculator that sources token sequences from both the prompt and previous outputs. EAGLE-3 and Token Recycling both leverage tree speculation [Miao et al., 2024].

Datasets and Agentic Applications. We constructed our evaluation datasets by running two *real* agentic applications, tracing the requests that they sent to their LLMs, and replaying their requests in Spec-Bench. First, we ran the OpenHands [Wang et al., 2024c] agent on *SWE-Bench* [Jimenez et al., 2024], a benchmark for resolving real-world GitHub issues. The agent generates multiple solutions, executes code in a secured environment, and iteratively refines its solution based on execution results [Wang et al., 2024b]. Second, we ran *AgenticSQL*, a proprietary multi-agent workflow for SQL code generation, described in Fig. 3. AgenticSQL exhibits both high task diversity (between workflow stages) and specialization (each stage may only perform a very narrow task).

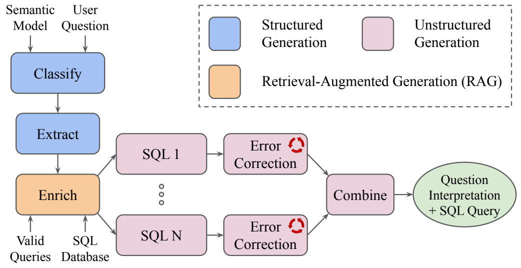


Figure 3: AgenticSQL is a multi-agent workflow consisting of stuctured generation, unstructured generation, and retrieval-augmented generation steps across several different LLMs. Useful features are extracted from the user question (Classify and Extract) and supplemented with retrieved context (Enrich). Several text-to-SQL steps propose solutions to the user question (SQL 1…N) in parallel with feedback from an error corrector. A last Combine step synthesizes the proposed SQL candidates into a final SQL query and text response.

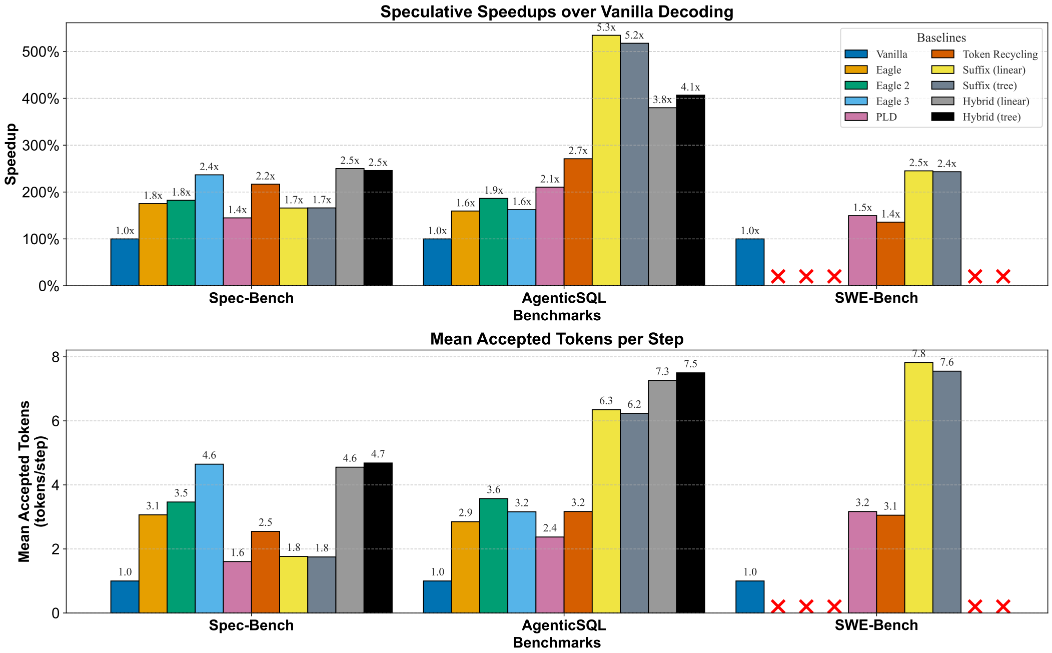


Figure 4: Speculative speedups (top) and mean accepted tokens per step (bottom) compared to vanilla decoding for SuffixDecoding and baseline methods on three benchmarks: Spec-Bench, AgenticSQL, and SWE-Bench. Experiments use Llama-3.1-8B-Instruct on a single H100 GPU with batch size 1. Speedup is measured as the ratio of time-per-output-token relative to vanilla decoding. Suffix (tree) and Hybrid (tree) use SuffixDecoding's tree speculation algorithm, which constructs a speculation tree from the suffix tree for parallel verification. Suffix (linear) and Hybrid (linear) use a simpler linear speculation approach that only allows sequential token chains. The hybrid variants combine SuffixDecoding with EAGLE-3, dynamically selecting between suffix-based and model-based speculation based on pattern match confidence. Note that EAGLE-2/3 and Token Recycling failed to run on several SWE-Bench tasks due to long context lengths (>8192 tokens), indicated by missing bars. Spec-Bench represents a non-agentic workload and is included for comparison. Further sub-task breakdowns, including the raw time-per-output-token and mean acceptance lengths, can be found in Appendix A.1.

End-to-end System Evaluation. We additionally implemented SuffixDecoding in vLLM [Kwon et al., 2023], a popular inference system used for real-world deployments. By running OpenHands live on vLLM, we show that SuffixDecoding accelerates end-to-end task completion times, which includes prefill and code execution time, thereby addressing a crucial bottleneck in agentic applications.

Hardware configuration. We conducted our experiments on a single `p5.48xlarge` AWS instance equipped with 8× NVIDIA H100 80G GPUs and 2TB of main memory.

Simulated Ablations. In addition to our main evaluation using real hardware, we also leverage a simulated verifier for additional experiments in Sec. 4.4 and continued in Appendix A.2. Given a prompt $x_{1:n}$ and example ground-truth response $y_{n+1:t}$, we can accurately simulate speculative verification for greedy sampling by verifying that speculated token $x_{n+i} = y_{n+i}$.

4.2 Baseline Comparisons

We compare SuffixDecoding with EAGLE-2, EAGLE-3, PLD, and Token Recycling on SWE-Bench and AgenticSQL. We also run the Spec-Bench standard dataset, which is a more traditional non-agentic workload. Fig. 4 shows the results. First, on the agentic workloads, SuffixDecoding outperforms all baselines. In AgenticSQL, SuffixDecoding obtains a mean speedup of 5.3× over vanilla decoding, a 2.8× improvement over EAGLE-2/3, and 1.9× higher than Token Recycling. In SWE-Bench, EAGLE-2/3 fail due to their maximum sequence length limitations. SuffixDecoding obtains a mean speedup of $2.5\times$ over vanilla decoding, a $1.7\times$ improvement over PLD, the next best baseline. SuffixDecoding's superior performance in agentic workloads can be attributed to its consistently higher mean accepted tokens per decoding step. In AgenticSQL, SuffixDecoding reaches 6.3 mean accepted tokens per step—substantially higher than EAGLE-3 (3.6 tokens) and Token Recycling (3.2 tokens). In SWE-Bench, SuffixDecoding achieves 7.8 mean accepted tokens per step, while PLD only accepts 3.2 tokens per step on average.

On high-entropy non-agentic workloads such as Spec-Bench, SuffixDecoding is outperformed by EAGLE-2/3 and Token Recycling. In this scenario, we use the hybrid approach of SuffixDecoding + EAGLE-3 to achieve the best of both worlds: we speculate with the faster SuffixDecoding method whenever possible and fall back to EAGLE-3 when the speculation score is too low. The Hybrid approach obtains a mean speedup of $2.5\times$ over vanilla decoding, outperforming the $2.4\times$ speedup from standalone EAGLE-3 and the $2.2\times$ speedup from Token Recycling.

The hybrid approach also performs well in AgenticSQL, achieving a $4.1\times$ speedup in the tree variant, significantly better than the $1.9\times$ speedup from standalone EAGLE-2/3 and the $2.7\times$ speedup from Token Recycling. These speedups are achieved thanks to the hybrid approach's impressive 7.5 mean accepted tokens per step, more than $2\times$ higher than EAGLE-2/3 and Token Recycling. SuffixDecoding has a slightly lower mean acceptance length of 6.3, but its much lower speculation cost and higher acceptance rate make it the winning solution in agentic tasks ($5.3\times$ average speedup compared to the $4.1\times$ speedup of the hybrid approach).

A peek into a speculation tree. To gain some intuition into why SuffixDecoding performs so well for certain tasks, we examine how it builds a speculation tree for the AgenticSQL Extract task. The outputs of the Extract task have many characteristics in common. First, they are all JSON documents following the same format and key names, with keys often appearing in the same order. Second, many of the features are discrete values, and in particular, boolean true/false values. These patterns are recorded in SuffixDecoding's global suffix tree and guide its speculation tree construction.

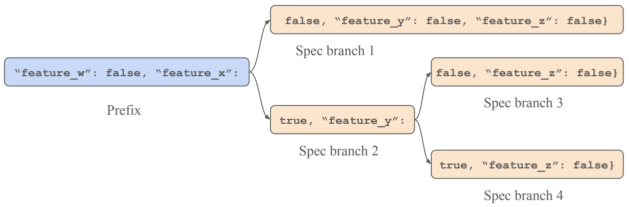

"feature_w": false, "feature_x":
Prefix
false, "feature_y": false, "feature_z": false}
Spec branch 1
true, "feature_y":
Spec branch 2
false, "feature_z": false}
Spec branch 3
true, "feature_z": false}
Spec branch 4

Figure 5: A SuffixDecoding speculation tree containing 66 tokens for the AgenticSQL Extract task.

Fig. 5 shows an example of a speculation tree constructed by SuffixDecoding. We observed many instances of large speculation trees that branch at each boolean true/false value of several consecutive features. These speculation tree always contains a branch with high acceptance, advancing output generation by dozens of tokens or more in one step. Although this is one specific example of a speculation tree, it demonstrates that SuffixDecoding can find complex patterns in previous outputs, particularly for structured generation tasks, that help accelerate output generation.

4.3 End-to-End SWE-Bench on vLLM

In this section, we show that SuffixDecoding can be efficiently integrated into vLLM, a popular inference system used in production deployments, and it can effectively accelerate accelerate end-to-end agentic task completion time. For this experiment, we run OpenHands directly on vLLM with SuffixDecoding, so the agent is solving each benchmark problem live. We also use the specially-trained LLM `all-hands/openhands-lm-32b-v0.1-ep3`, which was fine-tuned for SWE-Bench and achieves 37.2% on SWE-Bench Verified. Since there are no model-based methods with draft models trained for this LLM, we compare with vLLM's native implementation of PLD.

Fig. 6 shows the results. First, we note that decoding time (i.e. output generation) takes a majority of the time across all SWE-Bench tasks, dominating both prefilling and agentic actions (i.e. code execution). In this end-to-end scenario, SuffixDecoding outperforms PLD by 1.3–3×, leading to a 1.8–4.5× speculative speedup over vanilla decoding. Since SuffixDecoding exactly preserves the output distribution of the LLM, it matches the original model's 37.2% score on SWE-Bench Verified.

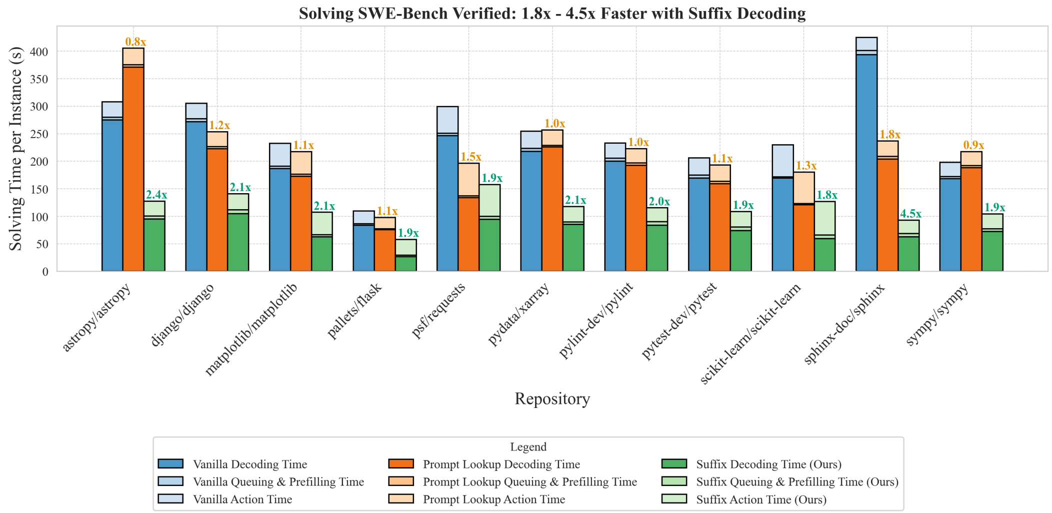


Figure 6: End-to-end task-completion time of the OpenHands agent on SWE-Bench Verified. The benchmarks are run with a concurrency of 8 tasks running simultaneously. vLLM is deployed on 4 H100 GPUs configured with 4-way tensor parallelism and prefix caching enabled. The results are broken down by the different code repositories in SWE-Bench.

4.4 Ablation Experiments

In this section, we present a few ablation studies on SuffixDecoding using a simulated verifier on offline traces. Given a ground-truth prompt-response pair from an LLM, we can verify the draft tokens proposed by SuffixDecoding by comparing with the ground truth responses. Additional ablation studies can be found in Appendix A.2.

Global vs per-request suffix trees. We study the impact of the two suffix trees: the global suffix tree containing previous outputs, and the per-request suffix tree containing the prompt and generation of the current ongoing request. To do so, we ran the tasks in AgenticSQL using SuffixDecoding (1) with the global suffix tree only, (2) with the per-request suffix tree only, and (3) using both trees.

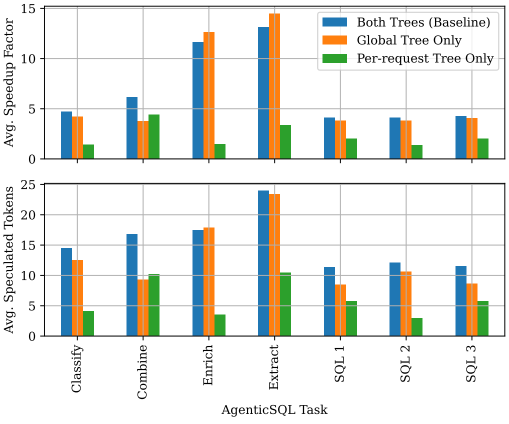


Figure 7: Speedup factor and number of speculated tokens for the tasks in AgenticSQL. SuffixDecoding was run with only the global suffix tree, only the per-request suffix tree, and both (baseline).

Fig. 7 shows the results. First, we note that with the exception of the Content Enrichment (Enrich) and Extract steps, using both suffix trees performs better than using just one. The small degradations on the enrich and extract steps suggest that, when both trees are present, SuffixDecoding may sometimes choose a speculation tree from the per-request suffix tree when the global suffix tree may have been the better choice. Improvements to SuffixDecoding's speculation tree scoring mechanism may help bridge this gap.

Second, the global tree outperforms the per-request tree on all tasks except for Combine. This is because the Combine task heavily re-uses tokens from its context, which are the proposed SQL solutions from the previous steps in the workflow. Although there is a diversity of task characteristics, SuffixDecoding is able to achieve high speedups on all of them by combining both suffix trees.

SuffixDecoding in open-ended scenarios. Although SuffixDecoding is designed for agentic workloads with long repeated token sequences, it is also interesting to evaluate it using more open-ended workloads like WildChat (open-ended chat) [Zhao et al., 2024] and Magicoder (code-oriented chat) [Wei et al., 2023]. Details on these datasets can be found in Appendix A.2.

In Fig. 8, we show the speedup and acceptance rate of SuffixDecoding on WildChat and Magicoder across a range of suffix tree sizes between 256 and 10,000 output examples. First, we note a promising pattern: the speedup consistently improves as the size of the suffix tree grows. This indicates that SuffixDecoding can learn useful patterns even in workloads with lower token repetition, and may be a substitute for model-based methods when a draft model is not available.

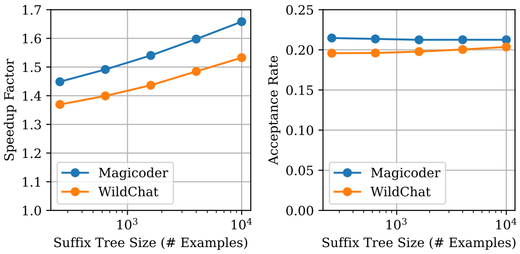


Figure 8: Speedup (left) and acceptance rate (right) vs global suffix tree size for Magicoder and Wildchat ($\alpha = 1$). The speedup from SuffixDecoding continues to increase with more previous output examples, while the acceptance rate holds steady.

Second, perhaps surprisingly, the acceptance rate does not change much even when the suffix tree size varies across almost two orders of magnitude. We believe this is primarily due to the effect of the adaptive speculation length `MAX_SPEC` $= \alpha p$. Although less data may mean less certainty in the speculated tokens, the pattern matches are also shorter, which results in fewer speculated tokens.

5 Related Works

Speculative decoding. Speculative decoding can improve LLM inference latency without compromising the quality of the generated text. Other model-based methods include Medusa Cai et al. [2024] and SpecInfer Miao et al. [2024], which first introduced tree-based speculation. Medusa uses multiple decoding heads for parallel candidate generation, while SpecInfer uses multiple draft models or speculative sampling to generate a tree of candidate tokens. There are also several other model-free speculation methods like LLMA Yang et al. [2023] and ANPD Ou et al. [2024]. These methods rely on small reference texts such as the prompt or retrieved contexts, and lack the adaptive speculation mechanisms of SuffixDecoding. Compared to prior speculative decoding methods, SuffixDecoding is uniquely designed for the emerging class of agentic LLM applications.

Methods for accelerating LLM agents. Recent works also target the problem of latency in agentic applications. ALTO [Santhanam et al., 2024] takes a systems-oriented approach to optimize the latency of multi-agent workflows through more efficient pipelining and scheduling. Dynasor [Fu et al., 2024] monitors the certainty of agentic reasoning algorithms as they run, and early-terminates reasoning paths that are unlikely to improve the final answer. Compared to these systems, SuffixDecoding takes an orthogonal speculative decoding approach, and they can be used in combination.

6 Conclusion

In this paper, we presented SuffixDecoding, a model-free speculative decoding approach designed for emerging agentic applications. Using efficient suffix tree data structures, SuffixDecoding effectively exploits long and repetitive token sequences found in many agentic algorithms, such as self-consistency, self-refinement, and multi-agent pipelines. Using two practical agentic applications, OpenHands and AgenticSQL, we showed that SuffixDecoding significantly accelerates their decoding latency and task-completion times, and is also significantly faster than other model-based and model-free speculative decoding baselines.

References

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024. URL `https://arxiv.`

org/abs/2401.10774.

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

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023. URL https://arxiv.org/abs/2302.01318.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F. Karlsson, Jie Fu, and Yemin Shi. Autoagents: A framework for automatic agent generation, 2024a. URL https://arxiv.org/abs/2309.17288.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. Sequoia: Scalable, robust, and hardware-aware speculative decoding, 2024b. URL https://arxiv.org/abs/2402.12374.

Yichao Fu, Junda Chen, Siqi Zhu, Zheyu Fu, Zhongdongming Dai, Aurick Qiao, and Hao Zhang. Efficiently serving llm reasoning programs with certaindex, 2024. URL https://arxiv.org/abs/2412.20993.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145, May 2024a. ISSN 2150-8097. doi: 10.14778/3641204.3641221. URL https://doi.org/10.14778/3641204.3641221.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024b. URL https://arxiv.org/abs/2312.10997.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks, 2023. URL https://arxiv.org/abs/2210.02406.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-2: Faster inference of language models with dynamic draft trees, 2024. URL https://arxiv.org/abs/2406.16858.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test, 2025. URL https://arxiv.org/abs/2503.01840.

Feng Lin, Hanling Yi, Hongbin Li, Yifan Yang, Xiaotian Yu, Guangming Lu, and Rong Xiao. Bita: Bi-directional tuning for lossless acceleration in large language models, 2024. URL https://arxiv.org/abs/2401.12522.

Xianzhen Luo, Yixuan Wang, Qingfu Zhu, Zhiming Zhang, Xuanyu Zhang, Qing Yang, Dongliang Xu, and Wanxiang Che. Turning trash into treasure: Accelerating inference of large language models with token recycling, 2024. URL https://arxiv.org/abs/2408.08696.

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 932–949, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651335. URL `https://doi.org/10.1145/3620666.3651335`.

Jie Ou, Yueming Chen, and Wenhong Tian. Lossless acceleration of large language model via adaptive n-gram parallel decoding, 2024. URL `https://arxiv.org/abs/2404.08698`.

Keshav Santhanam, Deepti Raghavan, Muhammad Shahir Rahman, Thejas Venkatesh, Neha Kunjal, Pratiksha Thaker, Philip Levis, and Matei Zaharia. Alto: An efficient network orchestrator for compound ai systems. In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 117–125, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705410. doi: 10.1145/3642970.3655844. URL `https://doi.org/10.1145/3642970.3655844`.

Apoorv Saxena. Prompt lookup decoding, November 2023. URL `https://github.com/apoorvumang/prompt-lookup-decoding/`.

E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995. ISSN 0178-4617. doi: 10.1007/BF01206331. URL `https://doi.org/10.1007/BF01206331`.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities, 2024a. URL `https://arxiv.org/abs/2406.04692`.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024b. URL `https://arxiv.org/abs/2402.01030`.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents, 2024c. URL `https://arxiv.org/abs/2407.16741`.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025. URL `https://arxiv.org/abs/2407.16741`.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023a. URL `https://arxiv.org/abs/2203.11171`.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada, July 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.754. URL `https://aclanthology.org/2023.acl-long.754`.

Zilong Wang, Zifeng Wang, Long Le, Huaixiu Steven Zheng, Swaroop Mishra, Vincent Perot, Yuwei Zhang, Anush Mattapalli, Ankur Taly, Jingbo Shang, Chen-Yu Lee, and Tomas Pfister. Speculative rag: Enhancing retrieval augmented generation through drafting, 2024d. URL `https://arxiv.org/abs/2407.08223`.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.

Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT '73, page 1–11, USA, 1973. IEEE Computer Society. doi: 10.1109/SWAT.1973.13. URL `https://doi.org/10.1109/SWAT.1973.13`.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024a. URL `https://arxiv.org/abs/2407.01489`.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding, 2024b. URL `https://arxiv.org/abs/2401.07851`.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. Inference with reference: Lossless acceleration of large language models, 2023. URL https://arxiv.org/abs/2304.04487.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL https://aclanthology.org/D18-1425.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. Draft & verify: Lossless large language model acceleration via self-speculative decoding, 2024a. URL https://arxiv.org/abs/2309.08168.

Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan Ö. Arik. Chain of agents: Large language models collaborating on long-context tasks, 2024b. URL https://arxiv.org/abs/2406.02818.

Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. Wildchat: 1m chatGPT interaction logs in the wild. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bl8u7ZRlbM.

Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. Deepresearcher: Scaling deep research via reinforcement learning in real-world environments, 2025. URL https://arxiv.org/abs/2504.03160.

A Technical Appendices and Supplementary Material

A.1 Details for Main Experiments

A.1.1 Experiment setup details

Setup of Spec-Bench experiments (Sec. 4.2). We conducted our Spec-Bench experiments by running the Spec-Bench codebase from the original repository with the following modifications. First, we updated the code to work with the latest version of the `transformers` library, which is required to run recent open-source LLMs such as `meta-llama/Llama-3.1`. We also added support for arbitrary datasets (such as SWE-Bench and AgenticSQL) and implemented SuffixDecoding within the framework. We ran the experiments on a 8xH100 80GB GPU cluster, with 1TB RAM. We ran each baseline using one GPU, and a batch size of 1, just like in the original SpecBench code.

Setup of vLLM SWE-Bench experiment (Sec. 4.3). We conducted the end-to-end SWE-Bench experiment on a 8xH100 80GB GPU cluster, with 1TB RAM. We served the `all-hands/openhands-lm-32b-v0.1-ep3` model locally using vLLM, with a tensor parallelism degree of 4 and with prefix caching enabled. We used the flashinfer kernels for sampling. We made some minor modifications to vLLM to record the per-request and per-step statistics of interest (time-per-token latency, throughput, acceptance length, acceptance rate). We used the same settings for all baselines. We ran the OpenHands daemon on the same machine, and used the OpenAI API to interact with the vLLM server. We ran OpenHands with the CodeActAgent Wang et al. [2024b] with `ITERATIVE_EVAL_MODE=true`, and a maximum of 100 iterations, as recommended by the OpenHands authors. We used a maximum of 16 concurrent workers to run the SWE-Bench tasks.

A.1.2 Detailed sub-task results

SWE-Bench: Mean accepted tokens (tokens/step)

System	astropy	django	matplotlib	seaborn	flask	requests	xarray	pylint	pytest	scikit-learn	sphinx	sympy	**Overall**
suffix (linear)	6.415	6.546	5.521	7.207	3.772	6.480	7.137	4.635	5.412	5.933	17.140	5.165	7.821
suffix (tree)	6.262	6.221	4.992	7.064	3.708	5.922	6.764	4.452	5.311	5.600	16.876	5.020	7.552
pld	2.831	3.008	2.756	3.080	2.195	2.996	3.223	2.629	2.904	2.669	4.724	2.641	3.168
recycling	3.159	3.058	3.004	3.133	-	-	2.978	3.072	2.992	3.046	2.994	-	3.054
vanilla	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
eagle3	-	-	-	-	-	-	-	-	-	-	-	-	-
eagle2	-	-	-	-	-	-	-	-	-	-	-	-	-
eagle	-	-	-	-	-	-	-	-	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-

SWE-Bench: Mean Acceptance Rate

System	astropy	django	matplotlib	seaborn	flask	requests	xarray	pylint	pytest	scikit-learn	sphinx	sympy	**Overall**
suffix (linear)	0.252	0.255	0.238	0.293	0.167	0.250	0.281	0.204	0.230	0.243	0.554	0.217	0.296
suffix (tree)	0.235	0.230	0.195	0.272	0.153	0.222	0.250	0.183	0.217	0.216	0.539	0.196	0.274
pld	0.191	0.210	0.184	0.215	0.128	0.208	0.231	0.174	0.199	0.175	0.379	0.175	0.225
recycling	0.028	0.027	0.026	0.028	-	-	0.025	0.027	0.025	0.026	0.026	-	0.026
vanilla	-	-	-	-	-	-	-	-	-	-	-	-	-
suffix-1.0-tree	0.385	0.381	0.348	0.420	0.330	0.388	0.409	0.349	0.375	0.375	0.632	0.373	0.421

SWE-Bench: Time per output token (ms)

System	astropy	django	matplotlib	seaborn	flask	requests	xarray	pylint	pytest	scikit-learn	sphinx	sympy	**Overall**
suffix (linear)	30.563	21.881	26.013	37.012	22.238	23.828	23.598	27.739	29.313	31.949	25.613	17.072	27.436
suffix (tree)	30.800	22.238	26.278	37.095	22.728	23.998	23.847	28.388	29.451	32.496	25.853	17.199	27.711
pld	41.675	31.029	34.447	47.588	31.549	31.155	31.330	37.386	38.238	41.531	41.877	25.142	37.758
recycling	41.328	31.847	34.155	49.670	-	-	33.434	34.304	38.438	39.667	78.153	-	43.774
vanilla	50.080	41.102	43.019	57.069	35.040	39.268	41.213	42.107	45.633	46.061	77.815	32.527	50.074
eagle3	-	-	-	-	-	-	-	-	-	-	-	-	-
eagle2	-	-	-	-	-	-	-	-	-	-	-	-	-
eagle	-	-	-	-	-	-	-	-	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-

SWE-Bench: Speculation time per generated token (ms)

System	astropy	django	matplotlib	seaborn	flask	requests	xarray	pylint	pytest	scikit-learn	sphinx	sympy	**Overall**
vanilla	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
suffix (linear)	0.156	0.122	0.175	0.185	0.198	0.182	0.178	0.203	0.217	0.203	0.125	0.167	0.171
suffix (tree)	0.170	0.135	0.172	0.172	0.191	0.165	0.165	0.235	0.235	0.212	0.138	0.150	0.175
pld	0.191	0.197	0.208	0.176	0.266	0.200	0.191	0.222	0.208	0.228	0.126	0.217	0.191
recycling	9.298	6.023	7.982	13.731	-	-	6.607	7.665	8.081	11.748	19.653	-	10.582
eagle3	-	-	-	-	-	-	-	-	-	-	-	-	-
eagle2	-	-	-	-	-	-	-	-	-	-	-	-	-
eagle	-	-	-	-	-	-	-	-	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-

SWE-Bench: Speedup over vanilla decoding

System	astropy	django	matplotlib	seaborn	flask	requests	xarray	pylint	pytest	scikit-learn	sphinx	sympy	**Overall**
suffix (linear)	2.213	2.442	2.039	1.951	1.792	1.962	2.210	1.792	1.888	2.356	4.453	2.292	2.452
suffix (tree)	2.158	2.412	1.991	1.985	1.759	1.972	2.190	1.754	1.912	2.296	4.417	2.280	2.433
pld	1.440	1.516	1.425	1.360	1.242	1.399	1.486	1.270	1.326	1.429	2.006	1.483	1.495
recycling	1.427	1.458	1.483	1.264	-	-	1.360	1.311	1.290	1.399	1.328	-	1.358
vanilla	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
eagle3	-	-	-	-	-	-	-	-	-	-	-	-	-
eagle2	-	-	-	-	-	-	-	-	-	-	-	-	-
eagle	-	-	-	-	-	-	-	-	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-

AgenticSQL: Mean accepted tokens (tokens/step)

System	Classify	Extract	Enrich	Combine	SQL1	SQL2	SQL3	**Overall**
hybrid (tree)	4.180	14.813	15.081	6.448	3.983	3.932	4.006	7.500
hybrid (linear)	4.008	13.473	15.304	6.362	3.876	3.842	3.913	7.262
suffix (linear)	3.577	11.833	12.395	5.924	3.665	3.005	4.005	6.349
suffix (tree)	3.470	11.724	12.137	5.834	3.633	2.914	3.904	6.236
eagle2	3.156	5.173	3.249	3.534	3.066	3.761	3.057	3.572
recycling	2.915	4.125	3.125	3.138	2.951	2.929	2.994	3.169
eagle3	2.529	3.374	4.198	3.179	2.142	4.622	2.056	3.160
eagle	2.305	4.062	2.877	3.127	2.166	3.295	2.109	2.851
pld	1.427	4.134	1.455	3.914	2.074	1.452	2.151	2.373
vanilla	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

AgenticSQL: Mean Acceptance Rate

System	Classify	Extract	Enrich	Combine	SQL1	SQL2	SQL3	**Overall**
vanilla	-	-	-	-	-	-	-	-
suffix (linear)	0.212	0.628	0.740	0.428	0.318	0.245	0.330	0.415
suffix (tree)	0.189	0.605	0.614	0.397	0.294	0.225	0.298	0.375
hybrid (tree)	0.131	0.642	0.683	0.249	0.138	0.143	0.140	0.304
hybrid (linear)	0.124	0.595	0.750	0.242	0.130	0.139	0.133	0.302
pld	0.061	0.365	0.069	0.355	0.137	0.076	0.144	0.173
eagle	0.052	0.122	0.075	0.085	0.047	0.092	0.044	0.074
eagle2	0.036	0.070	0.037	0.042	0.034	0.046	0.034	0.043
eagle3	0.025	0.040	0.053	0.036	0.019	0.060	0.018	0.036
recycling	0.025	0.041	0.028	0.027	0.025	0.024	0.026	0.028

AgenticSQL: Time per output token (ms)

System	Classify	Extract	Enrich	Combine	SQL1	SQL2	SQL3	**Overall**
suffix (linear)	9.552	2.687	3.007	6.023	9.559	10.588	9.767	7.306
suffix (tree)	9.594	2.876	3.188	6.164	10.339	10.935	10.090	7.592
hybrid (tree)	11.975	3.491	3.633	7.883	14.329	11.600	14.399	9.604
recycling	10.316	7.314	9.470	9.724	10.981	9.981	10.702	9.782
hybrid (linear)	12.563	3.681	3.762	8.603	14.827	11.639	21.219	10.874
eagle2	16.814	9.210	15.078	13.877	17.967	12.312	17.531	14.677
pld	20.146	7.735	20.321	8.173	14.605	20.546	14.646	15.169
eagle	21.221	11.308	15.760	15.688	23.353	13.396	24.595	17.887
eagle3	23.334	14.729	12.513	16.863	26.997	10.747	27.728	18.966
vanilla	26.578	25.292	25.032	25.406	27.011	25.851	26.307	25.924

AgenticSQL: Speculation time per generated token (ms)

System	Classify	Extract	Enrich	Combine	SQL1	SQL2	SQL3	**Overall**
vanilla	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
suffix (linear)	0.060	0.015	0.015	0.033	0.059	0.058	0.061	0.043
suffix (tree)	0.064	0.017	0.020	0.035	0.064	0.062	0.064	0.047
pld	0.419	0.124	0.422	0.130	0.281	0.434	0.276	0.298
recycling	0.762	0.303	0.399	0.577	0.928	0.260	0.922	0.592
hybrid (tree)	1.728	0.249	0.376	1.112	2.350	0.927	2.372	1.299
hybrid (linear)	1.776	0.256	0.358	1.177	2.387	0.930	4.389	1.604
eagle	3.189	1.693	2.423	2.372	3.423	2.197	3.616	2.700
eagle2	4.118	2.006	3.292	3.308	4.599	2.597	4.509	3.487
eagle3	6.092	3.526	3.021	4.350	7.156	2.543	7.332	4.854

AgenticSQL: Speedup over vanilla decoding

System	Classify	Extract	Enrich	Combine	SQL1	SQL2	SQL3	**Overall**
suffix (linear)	3.016	9.854	10.406	4.848	3.205	2.839	3.211	5.345
suffix (tree)	2.998	9.545	10.009	4.765	3.008	2.733	3.133	5.175
hybrid (tree)	2.338	7.672	8.191	3.613	2.057	2.496	2.077	4.068
hybrid (linear)	2.243	7.137	7.965	3.327	1.993	2.483	1.405	3.799
recycling	2.588	3.472	2.672	2.640	2.502	2.604	2.492	2.710
pld	1.330	3.695	1.255	3.298	1.936	1.311	1.905	2.105
eagle2	1.591	2.751	1.673	1.855	1.527	2.119	1.527	1.864
eagle3	1.328	1.720	2.025	1.619	1.108	2.496	1.056	1.623
eagle	1.324	2.246	1.598	1.669	1.229	1.955	1.138	1.595
vanilla	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

Spec-Bench: Mean accepted tokens (tokens/step)

System	coding	extraction	humanities	math	math_reasoning	qa	rag	reasoning	roleplay	stem	summarization	translation	writing	**Overall**
hybrid (tree)	6.325	5.418	4.980	5.817	5.080	4.958	4.812	4.610	4.556	5.155	4.550	3.432	5.306	4.684
eagle3	5.975	5.373	5.180	5.745	5.562	4.351	5.009	4.956	4.664	5.299	4.767	2.909	5.093	4.647
hybrid (linear)	5.958	5.249	4.966	5.446	5.121	4.452	4.753	4.607	4.557	5.089	4.520	3.345	5.158	4.553
eagle2	4.766	3.842	3.612	4.242	4.129	3.324	3.630	3.892	3.353	3.736	3.254	2.605	3.383	3.466
eagle	4.149	3.469	3.177	3.724	3.617	2.857	3.239	3.328	2.968	3.311	2.926	2.352	3.082	3.065
recycling	3.044	2.610	2.539	3.128	2.980	2.372	2.352	2.537	2.338	2.697	2.614	2.305	2.417	2.548
suffix (linear)	1.981	1.757	1.454	2.161	1.661	1.878	1.999	1.521	1.252	1.485	1.725	1.705	1.435	1.766
suffix (tree)	1.960	1.754	1.461	2.134	1.638	1.874	1.957	1.504	1.259	1.501	1.703	1.705	1.424	1.750
pld	1.911	1.670	1.387	1.957	1.475	1.461	1.967	1.490	1.206	1.430	1.816	1.362	1.394	1.606
vanilla	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

Spec-Bench: Mean Acceptance Rate

System	coding	extraction	humanities	math	math_reasoning	qa	rag	reasoning	roleplay	stem	summarization	translation	writing	**Overall**
suffix (linear)	0.255	0.216	0.162	0.240	0.163	0.152	0.216	0.171	0.116	0.169	0.199	0.216	0.208	0.190
vanilla	-	-	-	-	-	-	-	-	-	-	-	-	-	-
suffix (tree)	0.244	0.211	0.156	0.232	0.149	0.144	0.203	0.156	0.115	0.165	0.189	0.208	0.194	0.179
pld	0.132	0.097	0.063	0.126	0.071	0.088	0.119	0.076	0.037	0.068	0.103	0.130	0.072	0.099
eagle	0.126	0.099	0.087	0.109	0.105	0.074	0.090	0.093	0.079	0.092	0.077	0.054	0.083	0.083
hybrid (linear)	0.109	0.084	0.072	0.100	0.078	0.074	0.075	0.069	0.062	0.075	0.068	0.044	0.077	0.070
hybrid (tree)	0.107	0.084	0.072	0.101	0.077	0.075	0.074	0.069	0.062	0.076	0.068	0.045	0.077	0.070
eagle3	0.083	0.073	0.070	0.079	0.076	0.056	0.067	0.066	0.061	0.072	0.063	0.032	0.068	0.061
eagle2	0.063	0.047	0.044	0.054	0.052	0.039	0.044	0.048	0.039	0.046	0.038	0.027	0.040	0.041
recycling	0.026	0.020	0.019	0.027	0.025	0.017	0.017	0.020	0.017	0.021	0.020	0.017	0.018	0.020

Spec-Bench: Time per output token (ms)

System	coding	extraction	humanities	math	math_reasoning	qa	rag	reasoning	roleplay	stem	summarization	translation	writing	**Overall**
hybrid (linear)	7.218	9.235	8.970	7.974	8.783	11.534	10.576	9.899	10.195	8.698	9.857	14.858	8.806	10.747
hybrid (tree)	7.251	9.588	9.519	8.203	9.194	11.880	11.098	10.543	10.775	9.133	10.214	15.262	9.141	11.153
recycling	9.475	11.528	11.333	9.224	9.749	13.475	12.957	11.489	12.425	10.691	11.373	13.104	12.019	11.947
eagle2	9.721	13.256	12.822	11.157	11.338	14.903	14.012	12.631	14.229	12.461	14.422	19.116	13.992	14.387
eagle	10.201	13.193	13.570	11.485	11.895	16.219	14.222	13.215	14.639	13.022	14.840	19.425	14.280	14.925
suffix (tree)	13.833	16.284	18.670	13.547	16.304	19.839	15.285	18.382	21.339	18.034	15.825	16.595	19.154	16.875
suffix (linear)	13.595	16.245	18.212	13.292	16.447	19.999	15.378	18.061	21.217	18.041	15.925	16.670	18.925	16.936
pld	14.681	17.524	20.469	14.450	19.179	22.693	16.383	19.173	23.300	19.714	15.949	22.214	20.438	19.190
vanilla	24.721	24.903	24.979	24.466	24.992	25.082	25.698	24.475	24.433	24.871	25.114	24.904	24.463	25.076

Spec-Bench: Speculation time per generated token (ms)

System	coding	extraction	humanities	math	math_reasoning	qa	rag	reasoning	roleplay	stem	summarization	translation	writing	**Overall**
vanilla	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
suffix (linear)	0.071	0.079	0.099	0.067	0.087	0.097	0.089	0.084	0.101	0.094	0.092	0.076	0.084	0.088
suffix (tree)	0.074	0.080	0.104	0.069	0.091	0.099	0.093	0.088	0.103	0.097	0.096	0.079	0.086	0.091
recycling	0.243	0.318	0.291	0.240	0.255	0.391	0.419	0.304	0.320	0.275	0.321	0.365	0.310	0.340
pld	0.291	0.357	0.421	0.277	0.392	0.480	0.326	0.391	0.492	0.407	0.325	0.466	0.428	0.395
hybrid (linear)	1.369	1.838	2.026	1.510	1.854	2.455	2.152	2.070	2.353	1.925	2.118	3.096	1.949	2.259
eagle	1.698	1.977	2.268	1.877	1.892	2.389	2.034	2.060	2.421	2.166	2.434	2.885	2.351	2.289
hybrid (tree)	1.445	1.935	2.152	1.580	1.947	2.553	2.238	2.196	2.506	2.010	2.200	3.141	2.056	2.344
eagle3	1.913	2.172	2.218	2.001	2.030	2.674	2.419	2.334	2.500	2.160	2.402	4.084	2.237	2.634
eagle2	2.055	2.656	2.714	2.338	2.346	2.962	2.807	2.584	2.997	2.628	3.062	3.826	2.946	2.936

Spec-Bench: Speedup over vanilla decoding

System	coding	extraction	humanities	math	math_reasoning	qa	rag	reasoning	roleplay	stem	summarization	translation	writing	**Overall**
hybrid (linear)	3.441	2.795	2.800	3.139	2.866	2.404	2.542	2.496	2.461	2.871	2.573	1.759	2.833	2.500
hybrid (tree)	3.442	2.717	2.644	3.132	2.736	2.611	2.445	2.344	2.330	2.732	2.485	1.713	2.736	2.458
eagle3	3.115	2.634	2.713	2.910	2.888	2.172	2.474	2.446	2.394	2.764	2.520	1.447	2.622	2.367
recycling	2.619	2.234	2.209	2.660	2.577	1.951	2.057	2.150	1.979	2.334	2.218	1.932	2.057	2.169
eagle2	2.548	1.998	1.958	2.215	2.220	1.733	1.877	1.968	1.751	2.007	1.756	1.336	1.791	1.825
eagle	2.427	1.956	1.848	2.142	2.112	1.600	1.841	1.871	1.703	1.922	1.702	1.305	1.753	1.752
suffix (tree)	1.796	1.620	1.350	1.930	1.558	1.785	1.886	1.364	1.149	1.389	1.624	1.627	1.310	1.661
suffix (linear)	1.828	1.630	1.383	1.967	1.544	1.773	1.890	1.389	1.156	1.386	1.618	1.618	1.327	1.659
pld	1.712	1.484	1.231	1.753	1.323	1.326	1.807	1.320	1.055	1.275	1.628	1.219	1.232	1.448
vanilla	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

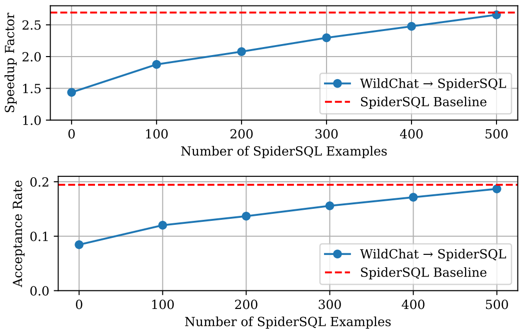


Figure 9: The performance of SuffixDecoding under input distribution shift. SuffixDecoding was trained on outputs from WildChat, while being evaluated on SpiderSQL. X axis: the number of SpiderSQL inputs, which are added to the global suffix tree after they are processed. Red line: the performance of SuffixDecoding if trained on 500 output examples from only SpiderSQL offline.

A.2 Additional Ablation Experiments

In this appendix, we share ablation studies that reveal the impact of several design decisions in SuffixDecoding. The studies are conducted using the simulated verifier described in Sec. 4.1.

A.2.1 Additional Dataset Details

We performed additional ablation experiments, which used additional datasets described below.

1. *WildChat* Zhao et al. [2024]. We use instructions from the WildChat dataset, which consists of real-world interactions between users and the ChatGPT service. WildChat represents the most diverse and open-domain dataset used in our evaluations.

2. *Magicoder* Wei et al. [2023]. Specifically, we use instructions from the Magicoder-Evol-Instruct-110K dataset, which consists of code-related questions and instructions generated via Self-Instruct Chaudhary [2023], Wang et al. [2023b] and further augmented for difficulty and diversity Luo et al. [2023] by GPT-4.

3. *SpiderSQL*. Spider Yu et al. [2018] is a dataset of manually-annotated questions and SQL responses over 200 different databases with multiple tables, covering 138 different domains. We use instructions from DAIL-SQL Gao et al. [2024a], which consists of LLM prompts with instructions to answer questions from Spider using structured SQL code.

A.2.2 Effect of input distribution shift

In real-world LLM serving, the input characteristics of requests may change over time, and may be out-of-distribution from the output examples that SuffixDecoding was trained on. To evaluate this scenario, we run SuffixDecoding trained on WildChat outputs, and begin to send it inputs from SpiderSQL, which represents a very sudden distribution shift.

Fig. 9 shows the results. SuffixDecoding starts from having 4,000 output examples from WildChat, and begins to receive SpiderSQL inference requests. Without any adaptation, SuffixDecoding still achieves $1.5\times$ speedup and 8% acceptance rate, but is far from the $2.6\times$ speedup and 20% acceptance rate it would achieve if it were trained on 500 examples from SpiderSQL instead.

After processing each SpiderSQL inference request, SuffixDecoding can insert its output into its global suffix tree, which means it can adapt in an online fashion to the new input distribution. As Fig. 9 shows, the performance of SuffixDecoding improves with the number of SpiderSQL inference requests processed. Perhaps surprisingly, after observing 500 SpiderSQL and adapting online, SuffixDecoding's performance is almost indistinguishable to its performance if it were trained offline on the 500 SpiderSQL examples alone. This suggests that SuffixDecoding is able to adapt to input distribution shifts quickly and at no loss in performance.

A.2.3 Predicting SuffixDecoding Effectiveness

SuffixDecoding tends to perform better on more structured tasks compared to very open-ended ones (e.g., AgenticSQL vs WildChat). We can measure this "structuredness" using empirical entropy. The steps are as follows: (1) create a suffix tree from example model outputs (100 examples is typically enough), (2) calculate the entropy of each node's output distribution by determining how often each child node is accessed, and (3) compute a weighted average of this entropy across all nodes. A low average entropy indicates that output tokens are more predictable based on their prefixes, which generally suggests that Suffix Decoding will perform better.

Table 1: Measured empirical entropy of our various evaluation datasets.

Dataset	Average Entropy
AgenticSQL (Enrich)	0.171
AgenticSQL (Classify)	0.738
AgenticSQL (Extract)	0.0862
AgenticSQL (SQL1)	1.52
AgenticSQL (SQL2)	1.49
AgenticSQL (SQL3)	1.51
AgenticSQL (Combine)	1.49
Spider	2.50
WildChat	3.43
Magicoder	2.95

Table 1 shows the empirical entropy measured on samples from each of our evaluation datasets. We find that the average entropy is closely related to the intuitive understanding of the "structuredness" of each dataset. Additionally, it correlates well with the performance of SuffixDecoding on those datasets. Therefore, practitioners can calculate this value using a small number of output examples to assess whether SuffixDecoding is appropriate for their specific tasks.