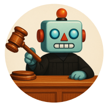


AutoJudge: Judge Decoding Without Manual Annotation

Roman Garipov [*] [†]
HSE University, Yandex

Fedor Velikonivtsev [*]
HSE University, Yandex

Ivan Ermakov
HSE University, Yandex

Ruslan Svirschevski
Yandex

Vage Egiazarian [‡]
IST Austria

Max Ryabinin
Together AI

Abstract

We introduce AutoJudge[1], a method that accelerates large language model (LLM) inference with task-specific lossy speculative decoding. Instead of matching the original model output distribution token-by-token, we identify the generated tokens that affect the downstream quality of the response, relaxing the distribution match guarantee so that the "unimportant" tokens can be generated faster. Our approach relies on a semi-greedy search algorithm to test which of the mismatches between target and draft models should be corrected to preserve quality and which ones may be skipped. We then train a lightweight classifier based on existing LLM embeddings to predict, at inference time, which mismatching tokens can be safely accepted without compromising the final answer quality. We evaluate AutoJudge with multiple draft/target model pairs on mathematical reasoning and programming benchmarks, achieving significant speedups at the cost of a minor accuracy reduction. Notably, on GSM8K with the Llama 3.1 70B target model, our approach achieves up to $\approx 2\times$ speedup *over speculative decoding* at the cost of a $\leq 1\%$ drop in accuracy. When applied to the LiveCodeBench benchmark, AutoJudge automatically detects programming-specific important tokens, accepting ≥ 25 tokens per speculation cycle at a 2% drop in Pass@1. Our approach requires no human annotation and is easy to integrate with modern LLM inference frameworks.

1 Introduction

Recent advances in LLM capabilities, including chain-of-thought reasoning [Wei et al., 2022, Kojima et al., 2022, Suzgun et al., 2022], writing complex software [Rozière et al., 2023, Li et al., 2023, Jiang et al., 2024], or interacting with external tools [Schick et al., 2023, Qin et al., 2023], increasingly rely on inference-time computation [Snell et al., 2024, Beeching et al., 2024]. This progress is further accelerated with the release of reasoning-capable models, both proprietary [OpenAI et al., 2024, Anthropic, 2024, Google DeepMind, 2025] and open-access [DeepSeek-AI et al., 2025, Meta, 2025, Qwen Team, 2025], that were explicitly trained to perform these kinds of inference-time computation. However, as LLMs tackle harder problems, they also tend to generate longer sequences [Muennighoff et al., 2025] with tens of thousands of tokens [Yeo et al., 2025], taking up tens of minutes (and hundreds of dollars in costs) per task [ARC Prize Foundation, 2024].

A popular way to speed up LLM inference is through speculative decoding [Leviathan et al., 2023, Chen et al., 2023], which uses a small "draft" model to propose the likely next tokens, then verifies these tokens with the main model in parallel. This method, along with its successors [Miao et al., 2023,

[1]Our code is available at `github.com/garipovroma/autojudge`.
[*]Equal contribution. [†]Corresponding author: `devilgar@gmail.com`.
[‡]Work done during employment at Yandex.

Cai et al., 2024, Li et al., 2024d], can speed up LLM inference while guaranteeing that the generated outputs match those of the original model (for greedy inference) or follow the same distribution. To achieve this, speculative decoding algorithms check if the draft tokens match the original model predictions. If there is a mismatch, they discard the incorrect token and all subsequent ones.

Speculative decoding can accelerate reasoning and other test-time computations, but it can be overly strict in how it discards tokens [Bachmann et al., 2025, Pan et al., 2025, Tran-Thien, 2024]. Intuitively, if a model generates a reasoning chain, not all mismatching tokens are equally important: errors in derivation should be fixed, while minor word choices should not. Judge Decoding [Bachmann et al., 2025] takes advantage of this by labeling which tokens are important for reasoning (and which are not) and allowing speculative decoding to accept more tokens by skipping the unimportant ones. However, their approach relies on human annotators to determine which tokens are important for reasoning. This complicates adoption and can be prone to human errors, particularly if the task requires expert knowledge (e.g., complex mathematical proofs or software engineering)

In this work, we look for ways to streamline this process. Instead of relying on human annotators, we propose **AutoJudge**, a search-based algorithm that detects which tokens are important for the task at hand based on how they affect the final answer. The algorithm is based on the idea that a token is deemed "important" not by itself, but in combination with other generated tokens. Thus, we propose a procedure that selects a small subset of important mismatching tokens that affect the final answer. Using this procedure, we can automatically mine a dataset to train an important token classifier that can then be used to accelerate speculative decoding.

The proposed search algorithm finds a small set of task-specific contextual "important tokens" — cases where target and draft models disagree on the next token in a way that affects the final response quality. We then train a classifier to detect these important tokens and use it to improve traditional speculative decoding by relaxing its verification procedure.

Our experiments with Llama 3.x models demonstrate that the proposed approach can indeed identify important tokens and save time on speculation, accepting on average over 40 tokens per target model forward pass (approx. $2\times$ that of speculative decoding), at the cost of a $\leq 1\%$ drop in accuracy on GSM8K [Cobbe et al., 2021] and even more with a minor accuracy drawdown. We obtain similar results with the Qwen2.5[Yang et al., 2024] family of models, observing comparable number of accepted tokens and accuracy trade-offs. When applied to programming tasks on LiveCodeBench [Jain et al., 2024], our approach is able to determine different task-specific important tokens, showing similar performance gains. The proposed framework is simple and general, using a classifier only when the original algorithm would reject a token, making it compatible with arbitrary speculative decoding algorithms. The main contributions of our work can be summarized as follows:

- We formulate AutoJudge, an algorithm for detecting which of the tokens generated in speculative decoding affect the downstream accuracy for a given task. Our algorithm requires no human annotation and can be applied to most popular LLM tasks.
- We verify the efficacy of AutoJudge on mathematical reasoning and programming benchmarks for several speculative decoding setups (model pairs). Our evaluations demonstrate favorable tradeoffs between the accuracy and the inference speedup, generating 20–45 tokens per speculative decoding cycle at the cost of a slight accuracy drawdown.
- We integrate AutoJudge with the vLLM framework [Kwon et al., 2023] and report the inference speed on A100 GPUs for both 8B and 70B target models, with up to $2\times$ speedup over speculative decoding at a $\leq 1\%$ quality decrease, and on H100 GPUs with a 405B target model.

2 Background

Speculative Decoding. Our work builds on top of speculative decoding [Stern et al., 2018, Leviathan et al., 2023, Chen et al., 2023], a family of inference algorithms that accelerate token generation by improving hardware utilization. Speculative Decoding uses an auxiliary "draft" model to generate $K>1$ possible future tokens, then runs the main "target" model *in parallel* to verify[*] the generated tokens. The drafted tokens that agree with the target model predictions are accepted by the algorithm. In turn, the first mismatching token and all subsequent ones are rejected. This way, the method

[*]For greedy decoding, it checks that the drafted tokens are the same as the target model's own next token predictions. For sampling, it uses a procedure that matches the sampling probabilities [Leviathan et al., 2023].

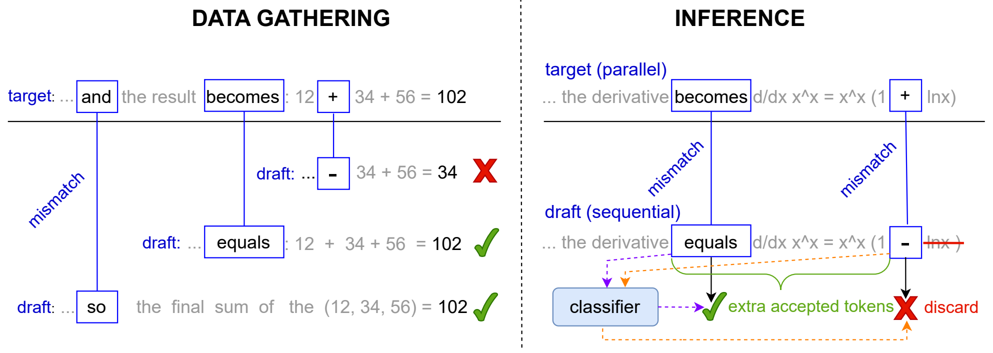

DATA GATHERING
INFERENCE
target: ... and the result becomes : 12 + 34 + 56 = 102
draft: ... - 34 + 56 = 34
draft: ... equals : 12 + 34 + 56 = 102
draft: ... so the final sum of the (12, 34, 56) = 102
mismatch
target (parallel)
... the derivative becomes d/dx x^x = x^x (1 + lnx)
draft (sequential)
... the derivative equals d/dx x^x = x^x (1 - ~~lnx)~~
mismatch
mismatch
classifier
extra accepted tokens
discard

Figure 1: Intuitive scheme of the proposed approach: **(left)** data collection: detecting mismatching tokens that affect final response quality; these tokens are then used to train a classifier **(right)** using the trained classifier to generate more tokens per cycle with speculative decoding.

guarantees that all generated tokens follow the same distribution as sampling from the target model. Subsequent works improve on this idea by generating draft trees instead of single sequences [Miao et al., 2023, Liu et al., 2023, Chen et al., 2024, Svirschevski et al., 2024], training specialized "heads" to draft next tokens based on the model's hidden states [Cai et al., 2024, Ankner et al., 2024, Li et al., 2024d,c], and more [Fu et al., 2023, Spector and Re, 2023, Sun et al., 2023, He et al., 2023].

Lossy Speculative Decoding. The core guarantee of Speculative Decoding is that all generated tokens follow the probability distribution of the original model. However, there are practical scenarios where this guarantee can be sacrificed in favor of faster inference, which is known as lossy speculative decoding algorithms [Tran-Thien, 2024, Narasimhan et al., 2025, Kim et al., 2023]. Our work extends one such method: Judge Decoding [Bachmann et al., 2025]. The core idea of Judge Decoding is that speculative decoding should only reject the mismatching token if accepting it would harm the response quality. For instance, in mathematical reasoning, errors in the equations or logical fallacies are important for the final quality, while minor style changes are not. When writing code, algorithmic errors are important, while minor variable renames can be skipped in favor of faster inference.

The main challenge of Judge Decoding is determining which of the generated tokens can be skipped this way. Bachmann et al. [2025] address this problem by manually labeling a training dataset for the classifier. Judge Decoding requires human annotators to find the "mistake" — the first mismatching token that led the draft model to diverge from the original answer. The resulting dataset of high-quality training examples is then used to train a linear classifier that detects such "mistakes" during inference. Authors demonstrate that the collected dataset can, in principle, be reused across tasks and models. However, using the data from one task for inference on a different task results in substantial performance drawdown. Intuitively, different tasks (such as creative writing, math, or programming) have different criteria for which parts of the generated response matter most. Hence, it is best to train the important token classifier *for the exact task at hand*. However, doing so with Judge Decoding would require relabeling the data by human annotators, which can be costly and time-consuming in specialized domains such as medicine or law. To alleviate this problem, we develop an automated search procedure for determining important tokens without external human (or LLM) annotators.

3 Method Overview

Our approach consists of three important stages. First, we detect which of the mismatching tokens affect the model quality using a semi-greedy search algorithm that we describe in Section 3.1. We then use the gathered data to train a lightweight classifier that can detect important tokens at inference time (Section 3.2). Finally, we use the trained classifier to augment a speculative decoding algorithm as described in Section 3.3, so that it can generate more tokens per speculation-verification cycle.

3.1 Mining Important Tokens

In this section, we describe an algorithm to identify which draft tokens that mismatch with the target ones influence the final output quality. To achieve this, we systematically alter the generation output, swapping between draft and main model tokens and test how this affects the downstream task output, such as the final answer to a math problem or test outputs for a programming task. If replacing a

Algorithm 1 SEARCH FOR IMPORTANT TOKENS

```
1: Input: $x$: prompt, $\theta_{\text{draft}}$: draft model, $\theta_{\text{target}}$: target model
2: Output: a sequence of $\mathcal{M}$ mismatches, labeled as important or unimportant
3: $\mathcal{M} \leftarrow \emptyset$    ▷ A set of tuples (position, target token, draft token, important)
4: $y \leftarrow$ GENERATE$(x, \theta_{\text{target}})$
5: $\alpha \leftarrow$ EXTRACTANSWER$(y)$
6: $\widetilde{y} \leftarrow$ FORWARD$(x \oplus y, \theta_{\text{draft}})$.argmax(-1)[len(x)-1:-1]
7: $\mathcal{I} \leftarrow \{i \mid y_i \neq \widetilde{y}_i\}$    ▷ Indices where draft and target tokens mismatch
8: while $\mathcal{I} \neq \emptyset$ do
9:     $t \leftarrow \min(\mathcal{I})$    ▷ The earliest position where mismatch happened
10:    $\hat{y} = y_{1:t} \oplus \widetilde{y}_t \oplus$ GENERATE$(x \oplus y_{1:t} \oplus \widetilde{y}_t, \theta_{target})$ ▷ Replace $\widetilde{y}_t$ and continue with $\theta_{target}$
11:    $\hat{\alpha} \leftarrow$ EXTRACTANSWER$(\hat{y})$
12:    if $\alpha \equiv \hat{\alpha}$ then
13:        $\mathcal{M} \leftarrow \mathcal{M} \cup \{(t, y_t, \widetilde{y}_t, \texttt{False})\}$    ▷ Equivalent answer, token $y_t$ is not important
14:        $y \leftarrow \hat{y}$    ▷ Continue search from the new response
15:        $\widetilde{y} \leftarrow$ FORWARD$(x \oplus y, \theta_{\text{draft}})$.argmax(-1)[len(x)-1:-1]
16:    else
17:        $\mathcal{M} \leftarrow \mathcal{M} \cup \{(t, y_t, \widetilde{y}_t, \texttt{True})\}$    ▷ Different answer, token $y_t$ is important, keep it
18:    end if
19:    $\mathcal{I} \leftarrow \{i | y_i \neq \widetilde{y}_i \ \cap \ i > t\}$    ▷ Continue with the remaining mismatches after $t$
20: end while
21: return $\mathcal{M}$
```

target model token with its draft version does not change the final answer, we deem this token swap "unimportant" and allow it to be generated with the faster draft model. In turn, if swapping out the token changes the final answer, it is deemed "important" and should be generated by the main model.

In more formal terms, consider the task defined as a prompt x with and two models: the larger θ_{target} and the smaller θ_{draft}. Both models can generate a response $y = (y_1, \ldots, y_T) =$ GENERATE(x, θ_{draft}) with up to $T \leq T_{max}$ total tokens. For simplicity, we first assume that the GENERATE procedure is deterministic (e.g., greedy) and generalize to sampling in Appendix A.

Without the loss of generality, we also assume that there is a problem-specific way to extract the final answer from the model's response, $a =$ EXTRACTANSWER(y). In mathematical reasoning tasks such as GSM8K [Cobbe et al., 2021], the final answer is literally whatever the model puts after `"the final answer (is)"`. In programming tasks, the "answer" would be the output from the testing system given the generated code — either a report about passing and failing tests or a testing error (e.g., an Out Of Memory or Syntax Error). Finally, we say that two answers are equivalent $a_{ref} \equiv a_{alt}$ if they are the same from the downstream task perspective. Note that this does not require them to be exactly equal as strings: in math problems, `1.5`$\equiv$`3/2`, whereas in programming tasks, two programs can be equivalent despite having different variable names. If the task at hand does not have a formalized evaluation procedure, e.g., general conversation agents, we can define EXTRACTANSWER$(y) = y$ and detect if two answers are equivalent using an LLM or human judges.

Following this notation, let $y_{target} =$ GENERATE(x, θ_{target}) be the main model outputs. A token $y_t \in y_{target}$ is **un**important if swapping that token for the draft model's output results in an equivalent answer. Likewise, if replacing y_i (and continuing target generation from there) results in a different answer, then the original token was "important" and the token should be generated with θ_{target}.

Note that even if θ_{draft} is significantly smaller than θ_{target}, most of the individual tokens will match between the two. As such, we are only interested in the mismatches — the cases where draft and target models produce different tokens *given the same prefix*:

$$\mathcal{I}(x) = \{t \in [1, T) : \arg\max_{y_{next}} P(y_{next}|x, y_{1:t}, \theta_{target}) \neq \arg\max_{y_{next}} P(y_{next}|x, y_{1:t}, \theta_{draft})\},$$

where $y_{1:t} = y_1, \ldots, y_{t-1}$ denotes taking a prefix of y up to, but excluding index t. In practice, we can find these tokens quickly by re-encoding the target model response with the draft model: FORWARD$(x \oplus y, \theta_{draft})$`.argmax(dim=-1)[M-1:M+T-1]` , where $x \oplus y$ denotes concatenation, FORWARD$(\cdot, \cdot)$ is a parallel transformer forward pass that outputs next token logits, and the `logits.argmax(dim=-1)[M-1:M+T-1]` takes the most likely next tokens for every position, excluding the prompt and accounting for the shift from next token prediction.

When deciding if a mismatching token is important for the final response, we need to account for the fact that changing one token will most likely lead to changes in subsequent tokens. A naïve way to account for that change is by continuing* the response after replacing one token $\widetilde{y}_t$:

$$\hat{y} = y_{1:t} \oplus \widetilde{y}_t \oplus \textsc{generate}(x \oplus y_{1:t} \oplus \widetilde{y}_t, \theta_{target})$$

However, this approach has a significant downside in that it assumes that all subsequent tokens will be generated by θ_{target}, whereas in reality, some of them may be generated by θ_{draft} following the same algorithm. In preliminary experiments, we found that, with a capable enough θ_{target}, even significant generation errors can be detected and self-corrected (similar to the "Aha moment" from DeepSeek-AI et al. [2025], Muennighoff et al. [2025]). However, if the model makes multiple mistakes, they eventually reach a critical mass, leading to an incorrect answer.

To address this, we reframe our task from detecting individual important tokens to finding combinations of tokens that jointly affect the final answer. This changes our problem to **finding the minimal set of mismatching tokens that need to be generated by θ_{target} while still producing an equivalent answer***. Since replacing a single mismatching token affects all subsequent token choices, the exact solution to this problem requires a tree search over possible token assignments. While this type of tree search is possible, it would take up significant runtime due to the large number of LLM forward passes required to try all mismatch combinations.

To simplify the procedure, we opt instead for a simpler, semi-greedy search that starts from the target model response and iteratively tries to replace mismatching target model tokens with their draft counterparts. If replacing a token affects the final answer, we consider this token important and keep the original (target model) version. If, however, replacing the token results in an equivalent answer, we deem this token unimportant, replace it with the draft model version *and continue the search from the new sequence*, with a different suffix and possibly a different $\mathcal{I}$. That way, we guarantee that the search algorithm is aligned with what happens during inference: the important tokens are generated with the target model and the unimportant ones are kept from the draft model. We summarize the resulting search procedure in Algorithm 1 and discuss some of its implications in Appendix A.

3.2 Classifier Training

Once we gather a dataset of task-specific important tokens with Algorithm 1, we can train a classifier that would detect such tokens for use during inference. This classifier can, in principle, be any type of model, from a simple linear model or decision tree to a fine-tuned transformer layer. However, in our work, we default to training lightweight **linear models with existing LLM hidden states as features**, since those would introduce the least overhead during inference. There are several important design choices that can affect the effectiveness of such a classifier: we address each one separately.

1. Which token representations to use: the hidden states that predicted the mismatched token, or the next hidden states that encode the mismatched token itself? In our experiments, we found that using the latter representations results in substantially greater classifier accuracy (see Appendix B). However, obtaining these representations comes with a caveat.

Normally, when doing speculative decoding, one generates a draft "window" of W tokens with θ_{draft}, then verifies these tokens by processing them (in parallel) with θ_{target}. This automatically computes the necessary hidden representations for all but for the very last token — the next token predicted from the last hidden state in the window, which is not encoded. There are two ways to address this: either encoding the extra token alongside the window, or simply assuming that if the very last token mismatches between θ_{draft} and θ_{target}, it is automatically discarded without the classifier. However, in practice, we found that the overhead from either strategy is negligible and is outweighed by greater classifier accuracy that translates to more accepted tokens.

2. Which token alternative to use? Since the classifier works best with the representations from encoding the mismatching token, it is natural to ask which token should be encoded: the draft token, the mismatching target token, or both? When analyzing this, we found that using both token representations comes with a *slight* increase in classifier accuracy (see Appendix B). However,

*For notation simplicity, we assume that the $\textsc{generate}(\cdot, \cdot)$ function can be called with a prefix of a response. In that case, we assume that the *total* response length (and not just newly generated tokens) does not exceed T_{max}, so that the response cannot grow indefinitely with each subsequent replacement.

*More precisely, find the fastest-to-generate sequence, accounting for the differences in response length.

```
[GSM8K] Arnel had ten boxes of pencils ... how many pencils are in each box?
Arnel kept ten pencils and shared the remaining pencils with his 5 friends.
                       [.] He shared the ... ✓      [equally] with ... ✓
This means that the total number of pencils he shared is 10 * x - 10. ...
                                        [Arnel] ... ✓   [-] x - 10 ... ✗
```

```
[GSM8K] Adlai has 2 dogs and 1 chicken.
How many animal legs are there in all?
To find the total number of animal legs,
we need to calculate the legs [total] of
each animal and then add them up.

- 2 dogs have 4 [2] legs each, so 2
dogs have 2 * [times] 4 = 8 legs.
- 1 chicken has 2 legs.

Now [Adding] , let's add the legs
together [of] , we get 8 (from the dogs)
+ 2 (from the chicken) = 10 legs.

The final answer is 10.
```

~~~
[LCB] Given a string S of lowercase...
If there are adjacent occurren- ces of
a and b in S, print Yes; ...
```python
# -*-[YOUR] coding: utf-8 -*-
def[#] solve[check](s):
  for i in range(len(s) -[)] 1):
    if s[i] == 'a' and s[i+1] == 'b':
      return "Yes"
    if s[i] == 'b[a]' and s[i+1] == 'a':
      return "Yes"
  return "No"

if[#] __name__ == "__main__":...
~~~

Figure 2: Excerpts from GSM8K (top, left) and LiveCodeBench (right) labeled by Algorithm 1. Important mismatching tokens that are in red, unimportant ones are in green. Alternative tokens are shown in [brackets]. Black tokens are where θ_{draft} and θ_{target} gave the same prediction. The top example additionally shows θ_{target} continuations after mismatching tokens (✓ if $\alpha \equiv \hat{\alpha}$, ✗ if not).

obtaining these representations in practice would require running θ_{target} more than once during the verification stage, which would complicate inference and introduce performance overhead. For this reason, we opt to use only the draft token representations for the classifier, since those are already available in regular speculative decoding.

3. Which model provides feature representations? During the verification stage, we have access to both the draft and the target model hidden states: we can use either or both of them as the input. In practice, we found that concatenated draft and target model representations give slightly better results than just those of the target model, and using draft model representations alone is substantially worse. Since both representations are already available during inference, we opt to use each of them.

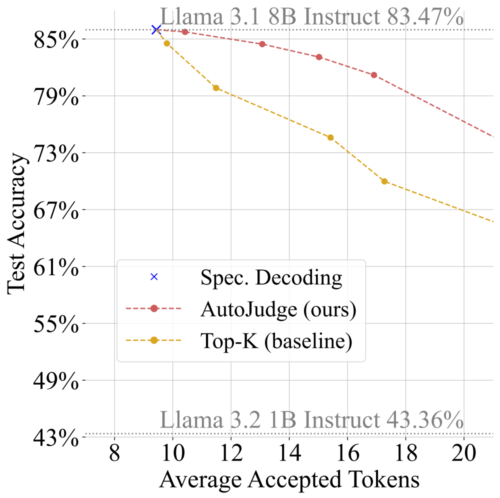
Classifier model & training. In this work, we train a simple logistic regression to detect important tokens. While a more complex model could achieve greater accuracy, logistic regression is significantly easier to deploy, has less runtime & memory overhead and needs less training data. Furthermore, it can be fused with the existing "LM head" layer of the draft and target LLMs, which would make its computation virtually free. To control overfitting, we perform a simple grid search over the L_2 regularization coefficient ("C") with a logarithmic grid. We report additional details in Appendix B.

3.3 Inference

The resulting classifier can be used with an arbitrary speculative decoding algorithm that has a verification stage. During said verification stage, the classifier is called when the original algorithm would reject a token. If the would-be-rejected token is deemed to be unimportant, i.e. not to affect the response quality, then we override the verification procedure and accept the token instead, proceeding to test subsequent tokens (if any) as per the original algorithm.

Generality. In our initial experiments, we focus on traditional speculative decoding [Leviathan et al., 2023, Chen et al., 2023] for simplicity. However, our algorithm is compatible with arbitrary speculative decoding algorithms, including tree-based [Miao et al., 2023, Svirschevski et al., 2024, Chen et al., 2024] and single-model multi-head algorithms [Cai et al., 2024, Li et al., 2024d,c]. This also means that our approach can be integrated into existing inference frameworks such as vLLM [Kwon et al., 2023], TensorRT-LLM [NVIDIA, 2023] or TGI [Hugging Face, 2023].

Thresholds. To balance computational efficiency and downstream performance, we select a decision threshold that achieves a high recall ($\geq$90%) in order to retain quality. Since the classifier is accurate enough, this threshold can also achieve a reasonable rejection rate, i.e., the rate of tokens correctly predicted to be unimportant. This allows us to retain downstream accuracy while skipping a large



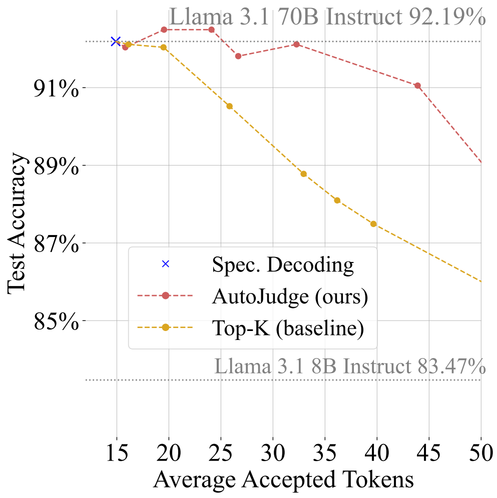


Figure 3: Accuracy and the number of accepted tokens on GSM8K for **(left)** 8-shot Llama-3.2 1B draft / Llama-3.1 8B target and **(right)** 0-shot Llama 3.1 8B draft / Llama 3.1 70B target (all Instruct)

portion of unimportant tokens, thus enabling efficient speculative decoding. In Section 4, we evaluate various threshold values to show their effect on accuracy and acceptance rate.

Comparison with Judge Decoding. As we discussed earlier, our approach can be seen as an extension of Judge Decoding that enables automatic dataset mining. As such, the dataset generation algorithm from Section 3.1 can be used in conjunction with the Judge Decoding training and inference protocol, which appears to be similar to ours up to possible minor details. In Appendix N, we additionally compare against manual human annotation similar to Judge Decoding. Unfortunately, the original source code and data from Judge Decoding are not available, making it difficult to compare directly.

4 Experiments

We organize our evaluations as follows: in Section 4.1 we evaluate AutoJudge on the GSM8K [Cobbe et al., 2021] mathematical reasoning benchmark. Next, in Section 4.2, we evaluate our approach on programming tasks from LiveCodeBench [Jain et al., 2024]. Finally, Section 4.3 contains GPU inference speed benchmarks with our vLLM implementation. We focus on two pairs of Llama 3.x models: 1) Llama-3.2-1B-Instruct draft / Llama-3.1-8B-Instruct target[*] and 2) Llama-3.1-8B-Instruct draft / Llama-3.1-70B-Instruct target. We also report results with Qwen2.5 models on the GSM8K benchmark to showcase the method's transferability across model families. Our main experiments run in `bfloat16` precision (see Appendix C). We run AutoJudge on top of the standard speculative decoding algorithm [Leviathan et al., 2023] in main experiments and explore EAGLE-2 in Appendix J.

4.1 Mathematical Reasoning with GSM8K

Our first set of experiments is based on the GSM8K dataset with grade school mathematical problems. This dataset has a natural split with ≈7.47K training samples and ≈1.32K test samples. Following the standard evaluation procedure, we use the training set to "mine" important tokens with Algorithm 1 and train the classifier, then run inference and evaluate on the test set with the recommended parameters [Gao et al., 2021] for zero-shot and 8-shot evaluation: greedy inference with a prompt that encourages chain-of-thought reasoning. During training, we consider two responses equivalent ($a \equiv \hat{a}$ in Algorithm 1) if the extracted final answers (numbers) are equal. For reference, we provide an example of important token assignments found by our algorithm in Figure 2.

We train a classifier on the last hidden state embeddings from both draft and target models (concatenated) for encoded draft tokens. The training dataset contains ≈130K mismatches, about 20% of which are deemed important. We train a logistic regression with the L_2 regularization coefficient tuned individually for each setup(for instance 10^{-4} for 8B/70B), found by grid search over a logarithmic grid between $10^0, \ldots, 10^{-7}$. During inference, we integrate the trained classifier into the

[*]The reason why the two models have different minor versions (i.e. 3.1 and 3.2) is that the 3.2 version does not have the larger 8B models, and the 3.1 version does not have the smaller 1B models.

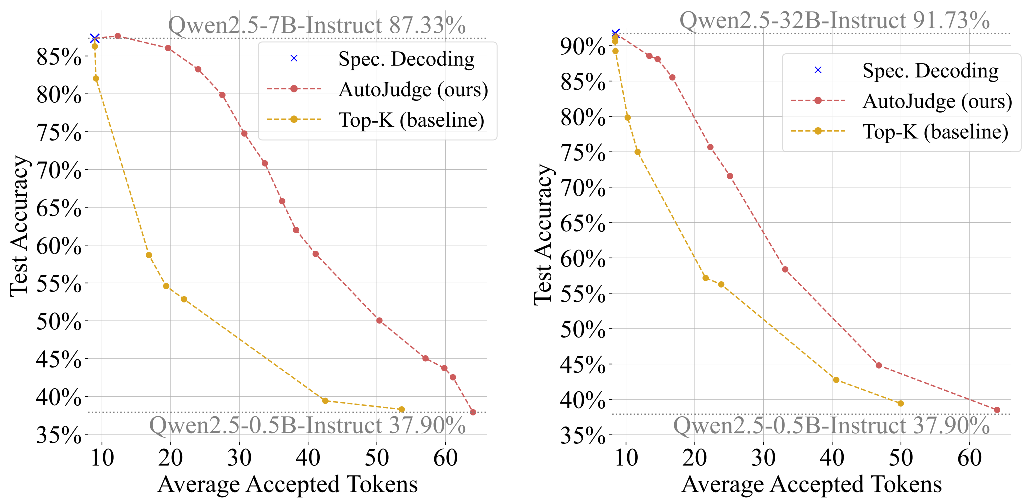


Figure 4: Accuracy and the average number of accepted tokens on GSM8K 8-shot for **(left)** Qwen2.5-0.5B draft / Qwen2.5-7B target and **(right)** Qwen2.5-0.5B draft / Qwen2.5-32B target (all Instruct).

speculative decoding loop from Leviathan et al. [2023] during verification. Whenever the original algorithm would reject a token, we run the classifier to determine if changing that token affects the final response quality, and if not — accept the token and continue verification for subsequent tokens (if any). Since the resulting algorithm can accept additional tokens, we use the increased draft window size of $W{=}64$ tokens for all evaluations.

We report two main metrics: downstream accuracy and the number of accepted tokens per speculative decoding cycle. The accuracy is measured as the exact match rate for the final answer extracted from the response as per standard GSM8K evaluation protocol. In turn, we report the decoding speed in terms of the number of tokens accepted per target model forward pass, aiming to decouple our results from the specific hardware configuration. We evaluate AutoJudge with different classifier thresholds, balancing between accuracy and speed. Our baselines are traditional speculative decoding, decoding with the draft model, and a simpler lossy speculative decoding method. In the latter, we accept a mismatching draft token if it is within top-K most likely tokens of the target model, similarly to how it is defined in Bachmann et al. [2025]. We report $K{=}2, 4, 8, \ldots, |V|$ for different speed-accuracy tradeoffs: increasing K results in more accepted tokens but reduces accuracy.

The results in Figure 3 demonstrate that AutoJudge decoding can achieve substantial speedups. Notably, our algorithm can accept over 40 tokens per target model forward pass in 8-shot evaluations for the 8B draft / 70B target model pair with a $\leq 1\%$ change in accuracy. Varying the classifier threshold allows us to achieve even greater speedups at the cost of several percentage points drop in accuracy. The heuristic-based top-K baseline also achieves some speedups, but at the cost of a significantly higher accuracy drawdown. We report additional setups in Appendix D.1.

4.2 Programming with LiveCodeBench

Next, we test if the AutoJudge search algorithm is able to generalize between domains. For this purpose, we evaluate the same model family on the LiveCodeBench [Jain et al., 2024] programming benchmark. Here, we use the `code_generation_lite`[*] dataset with the version tag `release_v5`. The dataset contains 880 programming tasks; we evaluate on all three subsets: `easy`, `medium` and `hard`. Since LiveCodeBench does not have a dedicated training split, we evaluate using out-of-fold predictions. Namely, we split the dataset randomly into 5 folds. For each fold, we evaluate using the classifier trained on the 4 remaining folds. We use the standard evaluation protocol: extracting the generated code and evaluating it using the benchmark's built-in test suite. We follow the standard evaluation protocol for this benchmark and report Pass@1 in zero-shot setting.

Similarly to Section 4.1, we use the training data to find important tokens — this time, in terms of the resulting program correctness, measured as passing tests. Since the calibration dataset is smaller and further subdivided into folds, we only have ≈27K mismatching tokens to train the classifier (with a

[*] `huggingface.co/datasets/livecodebench/code_generation_lite`

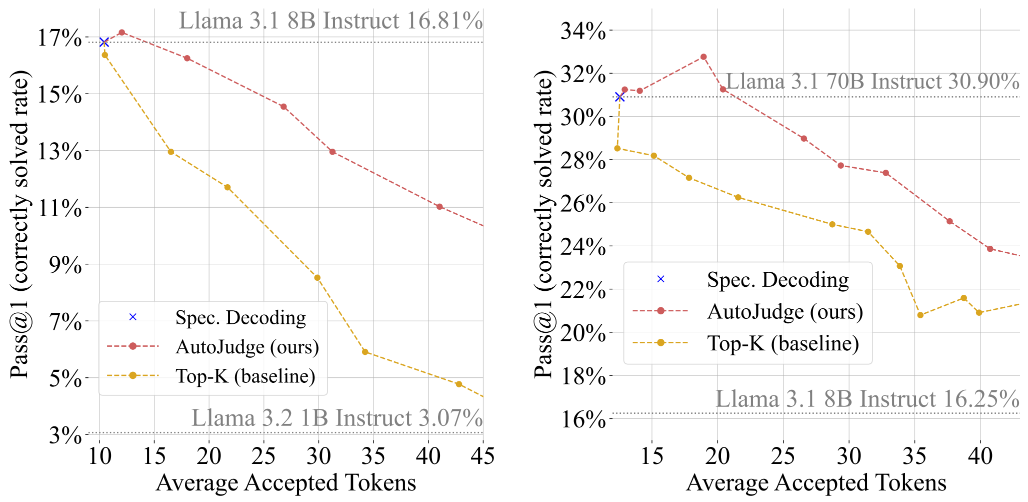


Figure 5: Downstream Pass@1 and the average number of accepted tokens on LiveCodeBench for **(left)** Llama-3.2 1B draft / Llama-3.1 8B target and **(right)** Llama 3.1 8B draft / Llama 3.1 70B target. We use Instruct versions for both model pairs and report additional details in Section 4.2.

slight $\leq 0.5K$ variation depending on the active fold). Furthermore, we found that only $\approx 5\%$ of the mismatching tokens were deemed to affect the output quality. We provide example token assignments in Figure 2 (right): notably, the tokens deemed important in that case would not appear in GSM8K in the same context. We otherwise follow the same training and evaluation protocol as before.

The results in Figure 5 are similar to what we observed in Section 4.1: AutoJudge decoding can accept over 22 tokens per speculative decoding cycle at the cost of a $\approx 2\%$ accuracy decrease. This results in approximately $2\times$ increase over traditional speculative decoding [Leviathan et al., 2023]. The top-K baseline can similarly achieve *some* increase in the number of accepted tokens, but AutoJudge decoding offers significantly better quality-speed tradeoffs across all configurations. We report additional configurations and threshold values in Appendix D.2, including the setup for AutoJudge decoding "out-of-domain", i.e. using the classifier trained on GSM8K for LiveCodeBench evaluation. This out-of-domain configuration results in inferior performance, which aligns with our hypothesis that the important tokens depend on the problem type and evaluation criteria.

4.3 Inference Benchmarks with vLLM

In this section, we report the GPU inference speed of speculative decoding with AutoJudge classifiers. To benchmark real-world inference speed, we integrated AutoJudge into vLLM speculative decoding. We use the same GSM8K evaluation setup as in Section 4.1 and evaluate 3 model pairs from Llama 3.x family: 1B/8B, 8B/70B and 8B/405B draft/target respectively (all Instruct).

Table 1: Inference speed benchmarks on GSM8K 0-shot with vLLM for **(left)** 1B draft / 8B target models with tuned window size (baseline = 8, AutoJudge = 10) and **(right)** for 8B draft / 70B target models (all Instruct) with tuned window size (baseline = 8, AutoJudge = 32).

Llama 3.2 1B draft / 3.1 8B target (0-shot)

Threshold	0.06	**0.12**	0.15	0.16
Accuracy, %	83.1	**80.2**	79.8	77.4
Speed, tokens/s	149.2	**169.2**	171.2	173.9
Speculative Decoding: 147.7 tokens/s				
Speedup(ours)	1.01	**1.14**	1.15	1.17

Llama 3.1 8B draft / 3.1 70B target (0-shot)

Threshold	0.005	0.031	**0.145**	0.230
Accuracy, %	92.0	91.9	**89.9**	88.0
Speed, tokens/s	72.3	80.6	**107.4**	109.6
Speculative Decoding: 72.3 tokens/s				
Speedup(ours)	1.0	1.11	**1.49**	1.52

Table 2: Inference speed with vLLM for **(left)** Llama 3.1 8B draft / 405B target models on GSM8K 0-shot with tuned window size (baseline=14, AutoJudge=20). **(right)** Llama 3.1 8B draft / 70B target (all Instruct) on GSM8K 8-shot with offloading, tuned window size (baseline=10, AutoJudge=48).

Llama 3.1 8B draft / 3.1 405B target (0-shot)				
Threshold	0.01	**0.05**	0.09	0.14
Accuracy, %	96.1	**93.4**	92.5	91.5
Speed, tokens/s	50.6	**58.0**	58.5	60.1
Speculative Decoding: 50.0 tokens/s				
Speedup(ours)	1.01	**1.16**	1.17	1.20

Llama 3.1 8B draft / 3.1 70B target (offload)				
Threshold	0.03	0.05	0.11	**0.28**
Accuracy, %	95.4	94.8	93.4	**90.4**
Speed, tokens/s	1.4	1.6	1.9	**2.4**
Speculative Decoding: 1.19 tokens/s				
Speedup(ours)	1.20	1.31	1.59	**1.98**

We compare AutoJudge decoding against lossless vLLM speculative decoding [Leviathan et al., 2023]. To ensure fair comparison, we tune draft window size for AutoJudge and baseline independently (see Appendix G). In each configuration, we report the absolute speed in tokens per second (batch size 1), as well as the relative **speedup over speculative decoding**. We run 1B/8B model pair on a single A100-SXM4-80GB GPU; 8B/70B on 4 A100-SXM4-80G GPUs in tensor-parallel mode. Finally, the 8B/405B runs on 8 H100-SXM5-80GB GPUs with the 405B model loaded in FP8 precision. Additionally, we consider a setup where 8B/70B model pair runs on a single A100-SXM4-80GB GPU with RAM offloading (see Appendix H). We provide additional configuration details in Appendix E.

Our results in Tables 1 and 2 demonstrate consistent improvements across all model pairs, with particularly high speedups for 8B/70B with and without offloading. This confirms that our earlier results in terms of accepted tokens (Sections 4.1 & 4.2) translate to real-world tokens per second. Namely, if we allow $\approx 2\%$ change in accuracy, our vLLM inference with AutoJudge classifier can achieve 107.4 tokens per second for 8B/70B model pair, which translates into about $1.5\times$ speedup compared to regular speculative decoding. This improvement is even more noticeable for hybrid setup with offloaded target model (in Table 2), where AutoJudge achieves up to $2\times$ speedup.

Additional evaluations. To better explore the wide variety of tasks, methods and hardware setups, we report multiple series of additional experiments in supplementary materials. We evaluate AutoJudge decoding with EAGLE-2 in Appendix J. Next, we generalize to open-ended problems (question answering and creative writing) with LLM-as-a-judge evaluation in Appendix L. We also evaluate how AutoJudge classifiers transfer to adjacent problems in Appendix M. Finally, we benchmark with equal window size in Appendix D and control for vLLM nondeterminism in Appendix I.

Limitations. Our approach assumes that the downstream task has a way to determine whether or not two solutions are equivalent. While this is true for many tasks, we found that AutoJudge sometimes struggles in open-ended tasks with no clear criteria for correct answers. It would be interesting to explore alternative algorithm designs more suitable for open-ended problems. Additionally, running Algorithm 1 consumes compute resources, mostly LLM inference, in the form of local inference or API calls. We discuss this in more detail in Appendices P and Q.

5 Conclusion

In this work, we propose and evaluate a fully automated protocol for task-specific acceleration of speculative decoding. Our experiments show that a simple procedure can automatically determine which of the mismatching tokens in the LLM response affect the downstream quality on a variety of tasks. Our runtime benchmarks demonstrate significant speedups on top of already tuned speculative decoding and EAGLE algorithms with minimal inference code modification. We hope that AutoJudge can facilitate the use of Judge Decoding across different task types, languages and modalities.

One promising direction for future research is to focus on open-ended problems such as instruction following or creative writing. While AutoJudge already demonstrates some speedups on open-ended tasks, it was primarily designed for technical problems such as math and programming. It would be interesting to investigate how it can be extended for longform generations with noisy and informal quality criteria. Another possible direction is to combine AutoJudge with additional speculative decoding algorithms, such as speculative decoding with tree-based drafts [Miao et al., 2023, Chen et al., 2024, Svirschevski et al., 2024] or learned drafting heads [Cai et al., 2024, Li et al., 2025].

Acknowledgements

We would like to express our sincere gratitude to Denis Mazur for his valuable contributions to the implementation of API calls used in Algorithm 1 and for supporting inference with the Llama 405B model. We are also thankful for his positive influence on the overall atmosphere and team morale throughout the course of this project.

References

Zachary Ankner, Rishab Parthasarathy, Aniruddha Nrusimha, Christopher Rinard, Jonathan Ragan-Kelley, and William Brandon. Hydra: Sequentially-dependent draft heads for medusa decoding, 2024. URL `https://arxiv.org/abs/2402.05109`.

Anthropic. Claude 3.7 sonnet and claude code, 2024. URL `https://www.anthropic.com/news/claude-3-7-sonnet`. Accessed: 2025.04.02.

ARC Prize Foundation. Openai's new o3 system scores breakthrough on arc-agi-pub, 2024. URL `https://arcprize.org/blog/oai-o3-pub-breakthrough`. Accessed: 2025.03.28.

Gregor Bachmann, Sotiris Anagnostidis, Albert Pumarola, Markos Georgopoulos, Artsiom Sanakoyeu, Yuming Du, Edgar Schönfeld, Ali Thabet, and Jonas K Kohler. Judge decoding: Faster speculative sampling requires going beyond model alignment. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=mtSSFiqW6y`.

Edward Beeching, Lewis Tunstall, and Sasha Rush. Scaling test-time compute with open models, 2024. URL `https://huggingface.co/spaces/HuggingFaceH4/blogpost-scaling-test-time-compute`.

Tianle Cai, Xinyun Li, Zhiruo Wang, Yuhuai Wang, and Dawn Song. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

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

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

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

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

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of llm inference using lookahead decoding. Accessed: 2023-11-29, 2023.

Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, September 2021. URL https://doi.org/10.5281/zenodo.5371628.

Google DeepMind. Gemini 2.5: Our Newest Gemini Model with Thinking. https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/#gemini-2-5-thinking, 2025. Accessed: 2025-04-07.

E. J. Gumbel. *Statistical Theory of Extreme Values and Some Practical Applications*, volume 33 of *Applied Mathematics Series*. National Bureau of Standards, Washington, D.C., 1954. U.S. Government Printing Office.

Horace He and Thinking Machines Lab. Defeating nondeterminism in llm inference. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20250910. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

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

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *ICLR*. OpenReview.net, 2020. URL http://dblp.uni-trier.de/db/conf/iclr/iclr2020.html#HoltzmanBDFC20.

Hugging Face. Text generation inference, 2023. URL https://github.com/huggingface/text-generation-inference. GitHub repository.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024. URL https://arxiv.org/abs/2406.00515.

Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, Vancouver, Canada, July 2017. Association for

Computational Linguistics. doi: 10.18653/v1/P17-1147. URL `https://aclanthology.org/P17-1147/`.

Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W. Mahoney, Amir Gholami, and Kurt Keutzer. Speculative decoding with big little decoder. In *Neural Information Processing Systems*, 2023. URL `https://api.semanticscholar.org/CorpusID:256868484`.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *ArXiv*, abs/2205.11916, 2022. URL `https://api.semanticscholar.org/CorpusID:249017743`.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Randy Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nourhan Fahmy, Urvashi Bhattacharyya, W. Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jana Ebert, Tri Dao, Mayank Mishra, Alexander Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean M. Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *ArXiv*, abs/2305.06161, 2023. URL `https://api.semanticscholar.org/CorpusID:258588247`.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E Gonzalez, and Ion Stoica. From crowdsourced data to high-quality benchmarks: Arena-hard and benchbuilder pipeline. *arXiv preprint arXiv:2406.11939*, 2024a.

Tianle* Li, Wei-Lin* Chiang, Evan Frick, Lisa Dunlap, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. From live data to high-quality benchmarks: The arena-hard pipeline, April 2024b. URL `https://lmsys.org/blog/2024-04-19-arena-hard/`.

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

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty. In *Proceedings of the 41st International Conference on Machine Learning*, pages 31147–31162. PMLR, 2024d.

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`.

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

Paul Joe Maliakel, Shashikant Ilager, and Ivona Brandic. Investigating energy efficiency and performance trade-offs in llm inference across tasks and dvfs settings, 2025. URL `https://arxiv.org/abs/2501.08219`.

Meta. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. `https://ai.meta.com/blog/llama-4-multimodal-intelligence/`, 2025.

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

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

Harikrishna Narasimhan, Wittawat Jitkrittum, Ankit Singh Rawat, Seungyeon Kim, Neha Gupta, Aditya Krishna Menon, and Sanjiv Kumar. Faster cascades via speculative decoding. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=vo9t20wsmd`.

NVIDIA. Tensorrt-llm, 2023. URL `https://github.com/NVIDIA/TensorRT-LLM`.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. Openai o1 system card, 2024. URL `https://arxiv.org/abs/2412.16720`.

Rui Pan, Yinwei Dai, Zhihao Zhang, Gabriele Oliaro, Zhihao Jia, and Ravi Netravali. Specreason: Fast and accurate inference-time compute via speculative reasoning, 2025. URL `https://arxiv.org/abs/2504.07891`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*. Neural Information Processing Systems Foundation, 2019.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

Yujia Qin, Shi Liang, Yining Ye, Kunlun Zhu, Lan Yan, Ya-Ting Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Marc H. Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. *ArXiv*, abs/2307.16789, 2023. URL `https://api.semanticscholar.org/CorpusID:260334759`.

Qwen Team. Qwq-32b: Embracing the power of reinforcement learning, March 2025. URL `https://qwenlm.github.io/blog/qwq-32b/`.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cristian Cantón Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950, 2023. URL `https://api.semanticscholar.org/CorpusID:261100919`.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *ArXiv*, abs/2302.04761, 2023. URL `https://api.semanticscholar.org/CorpusID:256697342`.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

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

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

Ziteng Sun, Ananda Theertha Suresh, Jae Hun Ro, Ahmad Beirami, Himanshu Jain, and Felix Yu. Spectr: Fast speculative decoding via optimal transport. *arXiv preprint arXiv:2310.15141*, 2023.

Mirac Suzgun, Nathan Scales, Nathanael Scharli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Annual Meeting of the Association for Computational Linguistics*, 2022. URL `https://api.semanticscholar.org/CorpusID:252917648`.

Ruslan Svirschevski, Avner May, Zhuoming Chen, Beidi Chen, Zhihao Jia, and Max Ryabinin. Specexec: Massively parallel speculative decoding for interactive llm inference on consumer devices, 2024. URL `https://arxiv.org/abs/2406.02532`.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vivien Tran-Thien. An optimal lossy variant of speculative decoding, June 2024. URL `https://huggingface.co/blog/vivien/optimal-lossy-variant-of-speculative-decoding`. Hugging Face Blog.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Edward Yeo, Yuxuan Tong, Morry Niu, Graham Neubig, and Xiang Yue. Demystifying long chain-of-thought reasoning in llms, 2025. URL `https://arxiv.org/abs/2502.03373`.

Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. Give me fp32 or give me death? challenges and solutions for reproducible reasoning, 2025. URL `https://arxiv.org/abs/2506.09501`.

A Additional Considerations for Section 3.1

Generalization to sampling. In Section 3.1, we assume that the generation procedure is deterministic, i.e. that the model performs "greedy inference". In practice, however, many applications work better with stochastic sampling [Holtzman et al., 2020]. However, this has an obvious caveat for Algorithm 1: if the text generation process is stochastic, a token can be deemed important based not on its actual impact on the model outputs, but on the randomness of the decoding procedure.

To generalize our approach for stochastic generation, we take advantage of the well-known Gumbel-max trick [Gumbel, 1954]. To recap, if we add independent Gumbel-distributed random variables to each predicted logit and take the index of the maximum, the probability that a certain index will be chosen is equal to the softmax of the original logits.

In case of Algorithm 1, we use the Gumbel-max trick to reparameterize stochastic sampling from the model as deterministic sampling conditioned on a predefined random state $s \leftarrow \textsc{randBits}(N)$. Given a prompt x, a response prefix $y_{1:t}$ and model parameters θ, we generate the next token as follows:

$$y_{next} = \arg\max_i \log P(i|x \oplus y_{1:t}, \theta) + \textsc{GumbelPRNG}(s \oplus x \oplus y_{1:t}),$$

where GUMBELPRNG is a function that samples a pseudo-random variable from standard Gumbel distribution based on an input seed $s \oplus x \oplus y_{1:t}$. To recall, $\oplus$ denotes concatenation. This way, y_{next} is distributed as $P(y_{next}|x \oplus y_{1:t}, \theta)$, but it is deterministic when conditioned on the random state s. Hence, we sample a random state s once at the beginning of Algorithm1, the entire procedure after that will also be conditionally deterministic (given s).

Issues with naïve important token mining. As we described earlier, Algorithm 1 is inherently sequential because it searches not for individual important tokens, but for important token combinations. In principle, it is tempting to consider a simpler algorithm that considers each token replacement in isolation and can run in parallel. However, when considering `[target_model_gen_0, draft_token, target_model_gen_1]` sequences only, a sufficiently strong target model might recover from even a low-quality token and still produce the correct answer. This results in a failure mode where all tokens are individually unimportant, but when all such tokens are *jointly* replaced with their draft versions, the model fails to produce the correct answer. In our preliminary experiments, when using Llama-3.1-70B-Instruct Touvron et al. [2023] as the target model and Llama-3.2-1B-Instruct as the draft model, fewer than 1% of the tokens were labeled as important with this simplified algorithm, whereas our main Algorithm 1 found substantially more. One interesting guarantee of Algorithm 1 over its naïve counterpart is that whenever draft and target models produce different (non-equivalent) answers to a given prompt, our algorithm will find at least one important token, whereas the naïve algorithm may find none.

On starting conditions for the important token search. To recall, mining important tokens can be viewed as a shortest path search algorithm in a tree of possible mismatch choices. When performing this type of search, there are two possible directions that one can search from. In Algorithm 1, we start from the target model outputs and iteratively (greedily) replace the mismatching tokens with their draft versions. However, one could also start from the draft model outputs and iteratively swap in target model outputs until the answer becomes equivalent to that of the target model. If we were to use an exhaustive search algorithm, both approaches would converge to the same important token labeling. However, since we are using a semi-greedy algorithm, it is easier to start with an already correct solution and simplify it, as opposed to starting with a wrong one and attempting to fix it.

B Additional Details on Classifier Training

As we discussed earlier in Section 3.2, there are several important design choices that can affect the performance of an important token classifier in our setting. Here, we report the experiments that led us to use a linear classifier based on draft token embeddings encoded with both θ_{draft} and θ_{main}. To that end, we compare the different classifier variants using the important token embeddings from the GSM8K [Cobbe et al., 2021] training subset (see Section 4.1).

To compare different classifier configurations, we further divide the GSM8K training set into classifier training (90%) and validation (10%) subsets. We perform this division at the sample level, i.e., all labeled tokens from a given GSM8K sample are used either entirely for classifier training, or entirely for validation. We use the same training and validation subsets throughout this section.

For the first set of experiments (Figure 6), we compare regularizer coefficients for Logistic Regression (left). We also report different classifier types: Logistic Regression, a Random Forest with 128 trees and a multi-layer perceptron (MLP) with a single hidden layer consisting of 128 hidden units with ReLU activation. For consistency, we run all models using scikit-learn [Pedregosa et al., 2011] v1.4.2 with all other settings kept to their default values. For LogisticRegression, we set `max_iter` parameter to 500. For MLP, we perform early stopping on yet another 10% subset of the training set with the built-in default MLPClassifier early stopping parameters. For RandomForest, we use `min_samples_leaf` set to 0.001. For this evaluation, all classifiers use draft and target model hidden states (concatenated) encoding the draft token, which is our main setup from Section 3.2.

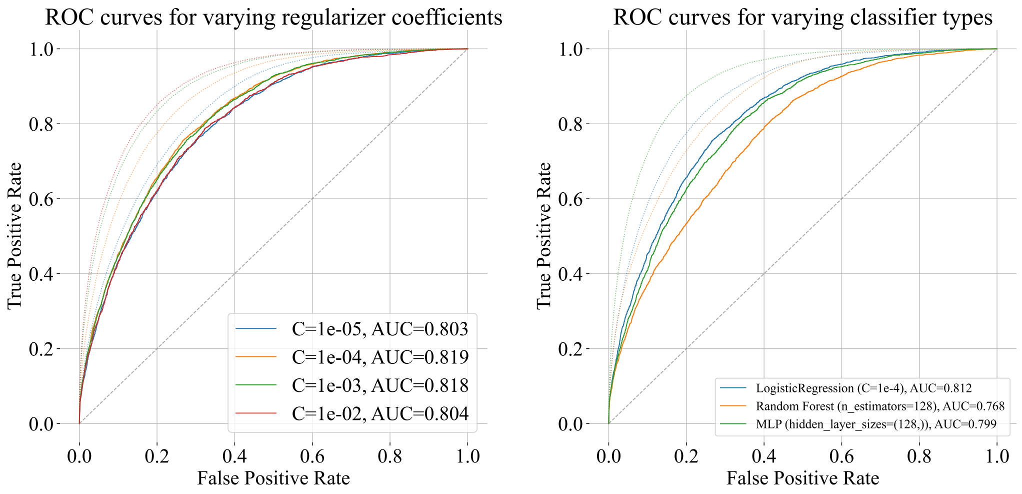


Figure 6: Receiver Operating Characteristics and the corresponding AUC values values for different Logistic Regression regularizers (left) and classifier types (right). Bold lines are validation curves and the dotted lines represent training curves. The AUC is reported in the legend (bottom right).

The results in Figure 6 demonstrate that the classifier quality is fairly robust to the choice of the regularization hyperparameter. It is also fairly robust to the choice of the classifier architecture, barring perhaps the Random Forest classifier, which is overfitting the training data more than other models. Note that this does not necessarily mean that the MLP or tree-based classifiers are generally worse than linear models — only that linear model is enough in our exact setup with a limited training set. We hypothesize that, if allowed to train on much larger dataset, the more complex models will be able to match and possibly outperform logistic regression.

Next, we compare classifier **inputs**. As we describe in Section 3.2, we use existing LLM hidden states from the last layer of θ_{draft} and θ_{target} since they are already computed during speculative decoding. This, however, leaves several possible choices about which hidden states should be used:

- **Previous token embeddings**, last hidden states used to predict the mismatching token;
- **Draft token embeddings** are the next embeddings, obtained by encoding the draft token;
- **Target token embeddings** are the next embeddings, obtained by encoding the target token;
- **Both token embeddings** are concatenations of the draft and target token embeddings;

We compare the four input configurations in Figure 7 (left), using Logistic Regression with $C{=}10^{-4}$ and both draft and target model hidden states (concatenated) for each case. The results suggest that a classifier that uses mismatching token embeddings (for draft *or* target token) is significantly more accurate than using the preceding token embeddings (the ones used to predict the mismatch). In turn,

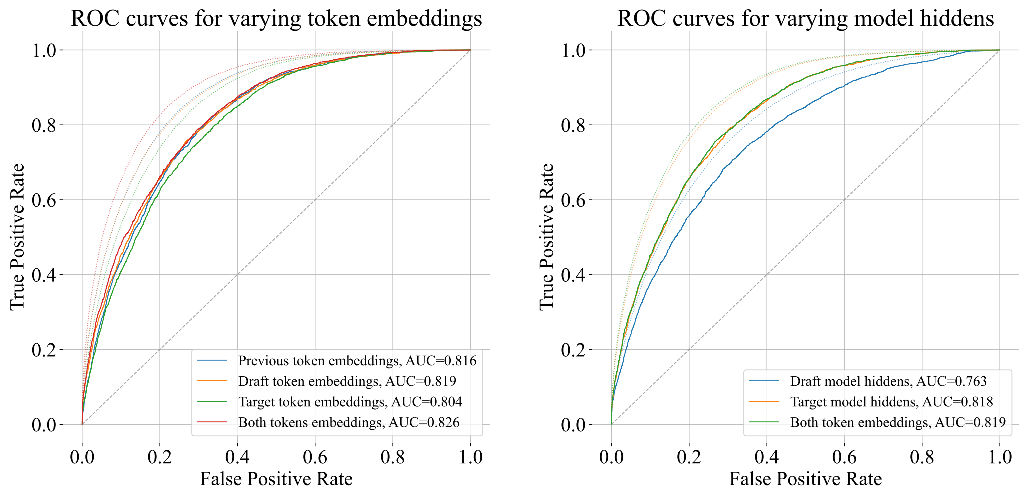


Figure 7: Receiver Operating Characteristics and the corresponding AUC scores (in the plot legend) for different classifier input tokens (left) and models (right). See Appendix B for details.

using both token embeddings results in somewhat better performance than either of them. However, using both token embeddings introduces complications during inference time.

In normal speculative decoding, the algorithm already computes hidden states for draft tokens with both θ_{draft} (during draft generation) and θ_{target} (during verification). However, it does not compute embeddings for target tokens since those tokens are not known before the end of the verification stage — and computing them already requires a forward pass with θ_{target}. As a result, computing target (or both draft & target) *token* embeddings would require two sequential forward passes with θ_{target} — one to determine the target tokens and detect mismatches, and the other to compute embeddings for those mismatching target tokens. In principle, one could devise a more sophisticated algorithm that computes only the θ_{draft} embeddings for mismatching target tokens or guesses the target tokens prior to the verification stage, but doing so would greatly complicate the implementation. Since the increase in the AUC score compared to using just the draft token embeddings is relatively small (Figure 7, on the left), we default to using draft token embeddings.

Additionally, we also test three model hidden states configurations for draft token embeddings: draft model hidden states, target model hidden states, and concatenated hidden states from both models in Figure 7 (right). Here, using the target model hidden states results in superior accuracy to using the draft model. In turn, using both θ_{draft} and θ_{target} produces an additional, if marginal, increase in accuracy. However, since both hidden states are already available during inference, using them both does not pose additional complications. Though, some real world inference systems may make it more convenient to only use θ_{target} for classifier inputs since the AUC difference is within 1%.

C Precision Matters for Speculative Decoding

When validating the AutoJudge algorithm, we found a peculiar implementation detail that can affect the real world performance of speculative decoding. Namely, *when using the LLM in half precision, token embeddings can differ significantly (up to 10%) between parallel and sequential forward passes on the same data.* In other words, if we record model hidden states as it generates a sequence, then encode the same sequence in parallel to recompute said hidden states, the two sets of hidden states will not match exactly. We attribute this to the fact that encoding tokens in parallel has a different summation order to encoding tokens one by one, which introduces small numeric errors. These errors compound over consecutive layers, resulting in larger errors in the final hidden states.

This is important for AutoJudge, since Algorithm 1 runs sequential inference with θ_{target} and parallel inference on θ_{draft}, whereas inference-time speculative decoding does it the other way around: sequential calculations of θ_{draft} during the draft generation phase, then parallel forward pass with θ_{target} during the verification phase. As a result, the classifier is trained on features that can be significantly different from what they would be during inference. In contrast, running in full precision (`float32`) does not have such problems.

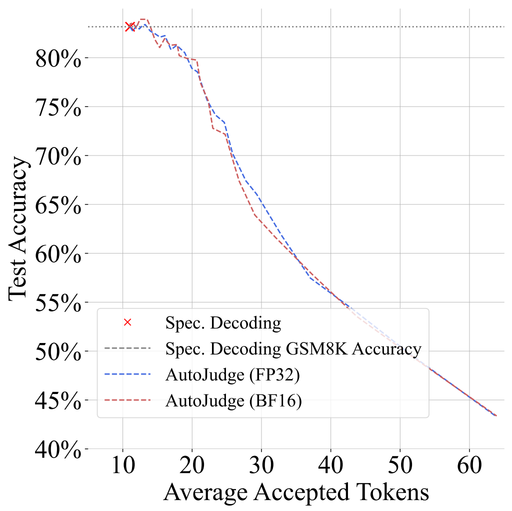


Figure 8: Accuracy on GSM8K and the number of accepted tokens per speculative decoding cycle in `float32` and `bfloat16` precision. The setup is the same as in Section 4.1.

In Figure 8, we compare accuracy and acceptance rate trade-offs for different classifier thresholds in the same setup as in Section 4.1. There are several ways to circumvent this problem. The most practical one would be to recompute target model embeddings for Algorithm 1 in a parallel forward pass and *not* using the draft model embeddings (since adding them has negligible effect on accuracy, see Figure 7, right). As a result, the classifier would use θ_{target} embeddings computed in parallel over draft tokens during both training and inference.

D Additional Evaluations

D.1 Additional Evaluations for Section 4.1

In Figure 9, we report accuracy and the number of accepted tokens for Llama-3.1-8B draft / Llama-3.1-70B target and Llama-3.1-8B draft / Llama-3.1-70B target model pairs in GSM8K 8-shot setup.

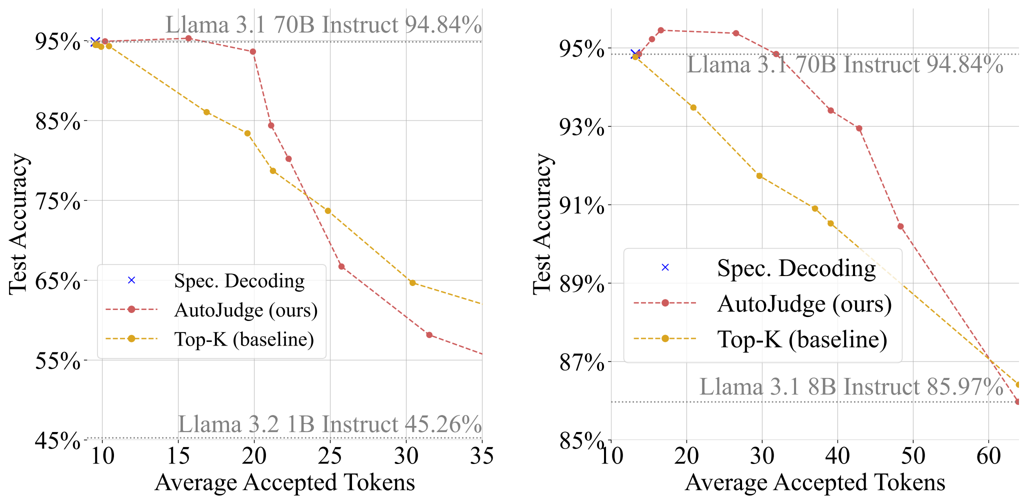
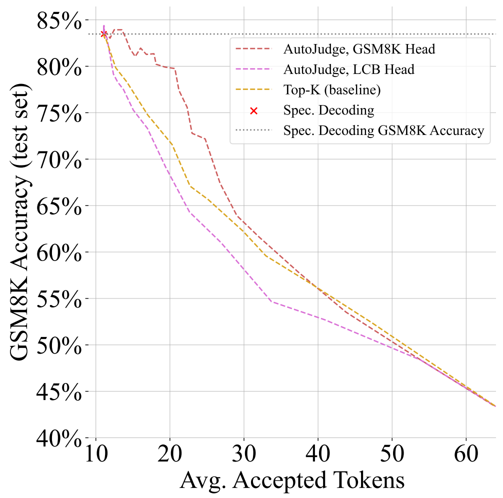


Figure 9: Downstream accuracy and the average number of accepted tokens for GSM8K 8-shot with Llama-3.1-70B-Instruct target and Llama-3.2-1B-Instruct draft models (left) and Llama-3.1-70B-Instruct target and Llama-3.1-8B-Instruct draft models (right)



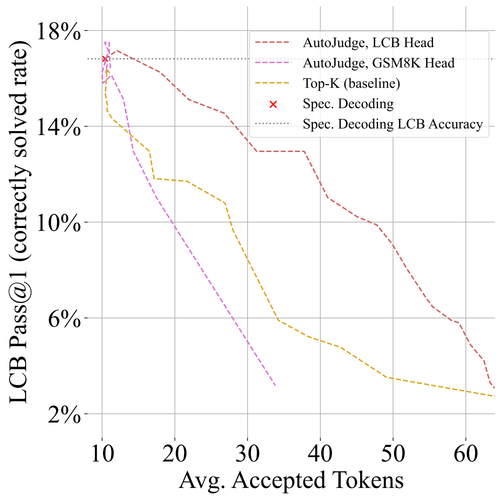
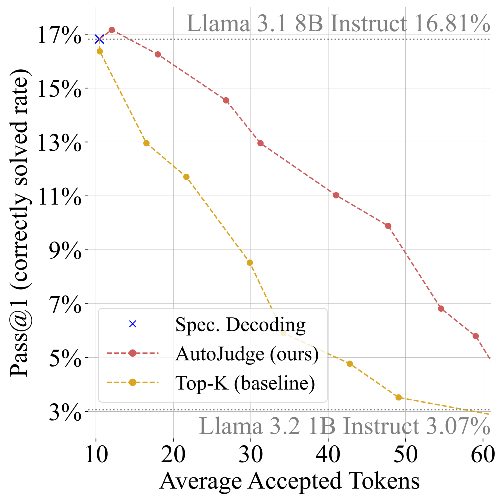


Figure 10: Downstream accuracy and the average number of accepted tokens for GSM8K (left) and LiveCodeBench (right) with Llama-3.1-8B-Instruct target and Llama-3.2-1B-Instruct draft models.



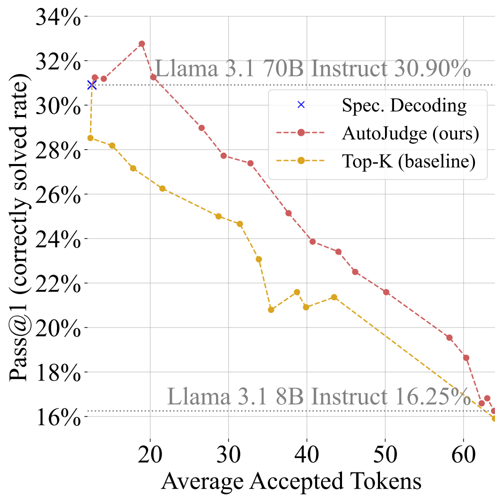


Figure 11: Downstream accuracy and the average number of accepted tokens for LiveCodeBench with Llama-3.1-8B-Instruct target and Llama-3.2-1B-Instruct draft models (left) and Llama-3.1-70B-Instruct target and Llama-3.1-8B-Instruct draft models (right)

D.2 Additional Evaluations for Sections 4.2

In this section, we provide additional classifier threshold evaluations for both model pairs in our LiveCodeBench setup. These results are reported in Figure 11, with 1B draft / 8B target models on the left and 8B draft / 70B target on the right. We use the same setup as in Section 4.2 and the portion of the results at the top are exactly the same values that we reported in Figure 5.

Additionally, we evaluate the AutoJudge classifier trained on LiveCodeBench on GSM8K and vice versa to gauge the effect of task-specific training. We repost results for Llama-3.2-1B draft / Llama-3.1-8B target models pair in Figure 10. Predictably, these out-of-domain classifiers perform significantly worse. We attribute this to the fact that the GSM8K-trained classifier likely did not see any Python source code, whereas the LiveCodeBench classifier did not perform arithmetic operations and did not solve equations that are common in GSM8K.

E vLLM Inference Implementation Details

To measure time efficiency of AutoJudge, we incorporate it into vLLM inference library. The integration is built upon `vllm==0.8.5`, `torch==2.7.0` with CUDA 12.8 and `transformers==4.51.3`.

We use a window size of 32 and a batch size of 1. The implementation allows to perform batched inference without modifications, but we choose to evaluate with batch size 1 for accurate measurements. For efficiency, the current implementation predicts important tokens using only the hidden state of the target model, as we have shown that adding hidden states of the draft model does not substantially increase the accuracy. We run evaluations for the 1B/8B model pair and for the 8B/70B pair 1 and 4 NVIDIA A100-SXM4-80GB GPUs respectively and on 8 NVIDIA H100-SXM5 80GB GPUs for 8B/405B in FP8 precision in the tensor parallel setup.

F Additional Inference Benchmarks: Equal Window Size

In this section, we report additional runtime benchmarks that extend our evaluations from Section 4.3. For these experiments, we use a fixed window size of 32 for both vanilla speculative decoding and AutoJudge (as opposed to individually tuned window size in Section 4.3).

We report our results for GSM8K with 1B/8B model pair in Table 3, 8B/70B in Table 4 and 1B/70B in Table 5. All three model pairs show significant speedups relative to standard speculative decoding. Notably, our 8B/70B setup has up to 2.5x with about 2.5% loss in accuracy.

Table 3: Inference speed benchmarks on GSM8K for 1B draft / 8B target model on: (left) GSM8K 0-shot evaluation and (right) GSM8K 8-shot evaluation.

Llama 3.2 1B draft / 3.1 8B target (0-shot)

Threshold	0.04	0.09	0.12	**0.16**
Accuracy, %	83.9	82.0	80.2	**77.4**
Speed, tokens/s	89.1	106.8	121.0	**146.2**
Speculative Decoding:	84.5 tokens/s			
Speedup(ours)	1.05	1.26	1.43	**1.72**

Llama 3.2 1B draft / 3.1 8B target (8-shot)

Threshold	0.050	0.087	**0.098**	0.133
Accuracy, %	85.7	85.1	**84.5**	83.1
Speed, tokens/s	66.5	74.3	**76.3**	83.3
Speculative Decoding:	62.4 tokens/s			
Speedup(ours)	1.07	1.19	**1.22**	1.33

Table 4: Inference speed benchmarks on GSM8K for 8B draft / 70B target model on: (left) GSM8K 0-shot evaluation and (right) GSM8K 8-shot evaluation.

Llama 3.1 8B draft / 3.1 70B target (0-shot)

Threshold	0.005	0.031	**0.145**	0.230
Accuracy, %	92.0	91.9	**89.9**	88.0
Speed, tokens/s	41.8	80.6	**107.4**	109.5
Speculative Decoding:	40.6 tokens/s			
Speedup(ours)	1.03	1.98	**2.64**	2.70

Llama 3.1 8B draft / 3.1 70B target (8-shot)

Threshold	0.03	0.05	**0.18**	0.28
Accuracy: Accuracy, %	95.4	94.8	**92.9**	90.4
Speed, tokens/s	64.4	71.1	**79.3**	86.2
Speculative Decoding:	40.5 tokens/s			
Speedup(ours)	1.58	1.74	**1.94**	2.11

Table 5: Inference speed benchmarks on GSM8K 8-shot with vLLM implementation for 1B draft / 70B target models.

Llama 3.2 1B draft / 3.1 70B target (8-shot)

Threshold	0.01	0.03	**0.05**	0.11
Accuracy, %	95.1	95.3	**94.6**	92.3
Speed, tokens/s	50.1	65.6	**75.5**	79.9
Speculative Decoding:	45.7 tokens/s			
Speedup(ours)	1.10	1.44	**1.65**	1.75

G Evaluation with Individually Tuned Window Sizes

As we discussed in Section 4.3, we tune the speculation window size individually for AutoJudge and vanilla speculative decoding to provide a more competitive baseline. This is because traditional speculative decoding accepts, on average, less tokens and does not benefit from having a larger draft window size. Thus, the two algorithms often work best with different draft sizes.

In this section, we account for this by evaluating AutoJudge and speculative decoding with optimal window sizes for every model pair. We consider window sizes between 6 and 64 with the following values: $[6, 8, 10, 12, 14, 16, 20, 24, 26, 28, 32, 40, 48, 54, 64]$ and choose the best window size in terms of tokens per second. Note that this evaluation protocol suffers from the natural variance in latency (e.g. due to varying individual clock rates) since we report the highest value measured.

The results for 1B/8B models pair are reported in Table 6. The results for 8B/70B pair are reported in Table 7. Overall, AutoJudge decoding still significantly outperforms standard speculative decoding.

Table 6: **(left)** Inference speed benchmarks on GSM8K 0-shot with vLLM implementation for 1B draft / 8B target models with tuned window size (Spec. Decoding = 8 tokens, AutoJudge = 10 tokens) and **(right)** for 1B draft / 8B target models on GSM8K 8-shot with tuned window size (Spec. Decoding = 6 tokens, Autojudge = 10 tokens).

Llama 3.2 1B draft / 3.1 8B target (0-shot)

Threshold	0.06	**0.12**	0.15	0.16
Accuracy, %	83.1	**80.2**	79.8	77.4
Speed, tokens/s	149.2	**169.2**	171.2	173.9
Speculative Decoding: 147.7 tokens/s				
Speedup(ours)	1.01	**1.14**	1.15	1.17

Llama 3.2 1B draft / 3.1 8B target (8-shot)

Threshold	0.050	**0.098**	0.133	0.164
Accuracy, %	85.7	**84.5**	83.1	81.2
Speed, tokens/s	114.4	**125.0**	132.1	139.3
Speculative Decoding: 116.8 tokens/s				
Speedup(ours)	0.98	**1.07**	1.13	1.19

Table 7: **(left)** Inference speed benchmarks on GSM8K 0-shot with vLLM implementation for 8B draft / 70B target models with tuned window size (Spec. Decoding = 8 tokens, Autojudge = 32 tokens) and **(right)** for 8B draft / 70B target models on GSM8K 8-shot with tuned window size (Spec. Decoding = 8 tokens, Autojudge = 16 tokens).

Llama 3.1 8B draft / 3.1 70B target (0-shot)

Threshold	0.005	0.031	**0.145**	0.230
Accuracy, %	92.0	91.9	**89.9**	88.0
Speed, tokens/s	72.3	80.6	**107.4**	109.6
Speculative Decoding: 72.3 tokens/s				
Speedup(ours)	1.0	1.11	**1.49**	1.52

Llama 3.1 8B draft / 3.1 70B target (8-shot)

Threshold	0.03	0.05	**0.18**	0.28
Accuracy, %	95.4	94.8	**92.9**	90.4
Speed, tokens/s	69.0	73.1	**83.6**	84.5
Speculative Decoding: 57.3 tokens/s				
Speedup(ours)	1.20	1.27	**1.45**	1.47

H Offloading

As we presented in Section 4, AutoJudge can accept up to 40 tokens per verification cycle, which makes it naturally well-suited for scenarios with a large draft window. In offloading setups, the drafting step is significantly cheaper than verification, yet speculative decoding suffers from being able to accept only a few tokens on average. AutoJudge avoids this limitation by accepting substantially more tokens per verification cycle, leading to notable gains in offloading configurations.

As reported in Table 8, the accuracy drop with AutoJudge does not exceed 3% across thresholds, while achieving throughput between 1.4 and 2.4 tokens/s, compared to 1.19 tokens/s for speculative

Table 8: Inference speed benchmarks on GSM8K 8-shot for 8B draft / 70B target models **with offloading** on a single NVIDIA A100-SXM4 GPU with tuned window size (Spec. Decoding = 10 tokens, AutoJudge = 48 tokens).

Llama 3.1 8B draft / 3.1 70B target (8-shot)				
Threshold	0.03	0.05	0.11	**0.28**
Accuracy, %	95.4	94.8	93.4	**90.4**
peed, tokens/s	1.4	1.6	1.9	**2.4**
Speculative Decoding: 1.19 tokens/s				
Speedup(ours)	1.20	1.31	1.59	**1.98**

decoding. This corresponds to relative speedups of 1.2×–1.98×. It turns out that the optimal window size for AutoJudge is 48 tokens, whereas for speculative decoding it is only 8.

I On the Instability of vLLM Inference for Accuracy Benchmarks

In Section 4, we evaluate AutoJudge decoding in terms of the number of accepted tokens per phase and the real-world tokens per second. For convenience, we only use vLLM implementation to evaluate the number of tokens per second and report all other metrics based on transformers / pytorch implementation. This is because, in our preliminary experiments, we observed discrepancies between benchmark accuracy with vLLM inference and PyTorch. Upon further investigation, we found that vLLM inference sometimes produces inconsistent results. Namely, if we run vLLM **greedy** inference in `bfloat16` precision with speculative decoding, changing "technical" hyperparameters such as window size can significantly affect the accuracy. To illustrate this, we measure GSM8K accuracy on 132 random test samples in the same setup as Section 4.3. We use standard vLLM implementation of speculative decoding Leviathan et al. [2023] *without any modifications to the vLLM codebase*.

Table 9: Llama 3.2 1B draft / 3.1 70B target, 10% random test samples from GSM8K. Accuracy is measured based on vLLM generations with varied window size.

Window Size	Accuracy, %
8	93.9
10	93.9
12	93.9
14	93.9
16	95.4
20	95.4
24	94.6
26	94.6

The results in Table 9 illustrate the observed inconsistency: changing window size can affect the accuracy to a significant degree. This coincides with concurrent observations about inconsistent LLM inference from Yuan et al. [2025] and He and Lab [2025]. To make our measurements more consistent, we report all accuracy and accepted token rates using a more stable implementation based on Hugging Face Transformers (included in our repository), only using vLLM to measure the speed in terms of tokens/second.

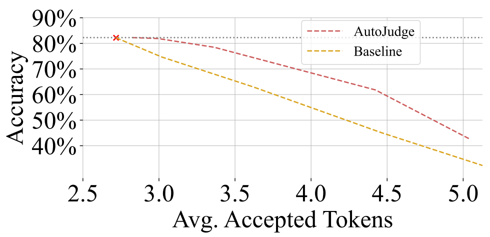
J EAGLE experiments

Following the invention of speculative decoding, several lines of work proposed follow-up algorithms that incorporate trained speculation "heads"[Cai et al., 2024, Li et al., 2024d,c], tree decoding [Miao et al., 2023, Chen et al., 2024, Svirschevski et al., 2024] and many other improvements. In this section, we explore how AutoJudge generalizes to these more advanced decoding algorithms.

To that end, we integrate AutoJudge with the popular EAGLE-2 algorithm [Li et al., 2024c]. Unlike the original speculative decoding, EAGLE-2 does not have a separate draft model, but trains a lightweight "head" to predict future tokens from target model hidden states. This allows EAGLE to draft tokens much faster, albeit less accurately than powerful standalone draft model.

We evaluate Llama 3.1 8B Instruct using the official pre-trained EAGLE heads*. Since there is no separate draft model, the classifier is trained we only use target model hidden states when training. Additionally, since EAGLE draft model was not trained to produce long coherent drafts, we use a shorter draft window size of 8. We integrate AutoJudge with the official EAGLE implementation and set parameters that are compatible with the vLLM EAGLE implementation (`depth=window_size-1`). Aside from that, we use the same evaluation protocol as in Section 4.1 for GSM8K.

In Figure 12, we report GSM8K accuracy and the average number of accepted tokens for AutoJudge with the official PyTorch implementation* of EAGLE 2. We also report real-world inference speed (tokens per second) using the vLLM implementation on a single A100-80GB GPU. The results suggests that integrating AutoJudge with EAGLE can produce additional speedups on top of the highly efficient speculative decoding algorithm.



Llama 3.1 8B Instruct + EAGLE head (0-shot)

Threshold	0.05	0.15	**0.30**	0.40
Accuracy, %	81.4	81.3	**81.0**	78.1
Speed, tokens/s	91.1	96.8	**102.6**	107.5
EAGLE:	89.8 tokens/s			
Speedup(ours)	1.01	1.08	**1.14**	1.20

Figure 12: Evaluating AutoJudge with EAGLE-2 draft head on GSM8K using Llama 3.1 8B Instruct model. (Left) Accuracy to accepted tokens in PyTorch, (right) vLLM inference speed on A100-80GB.

K Rule-Based Approaches Ablation

Table 10: Comparison between AutoJudge and simple rule-based heuristic focused on mathematical tokens, GSM8K 0-shot, Llama 3.2 1B Instruct draft / 3.1 8B Instruct target.

Method	**Math Only**	**Math & Top-1024**	**Math & Top-128**
Heuristic criterion			
Accuracy, %	65.05	72.27	80.36
Accepted Tokens	39.7	22.3	15.5
AutoJudge (nearest threshold)			
Accuracy, %	57.8	75.6	81.0
Accepted Tokens	37.4	22.3	15.3

To explore how AutoJudge decoding compares to rule-based methods for mathematical reasoning, we compare it against two heuristic alternatives. The first heuristic approach is to consider only the mathematical symbol tokens as "important". To that end, we filter the mismatching tokens that contain numbers, operations (e.g., + - * / =, etc.), as well as some common variables. Mismatches in these mathematical tokens are rejected, whereas mismatches in non-mathematical tokens are allowed. As it turns out (Table 10), this algorithm misses important planning and logical steps that do not contain computations explicitly, which results in poor accuracy. To address this, we introduce a second, more complex heuristic approach that combines the mathematical rule with the Top-K baseline we use in Section 4. This works somewhat better, but still does not outperform the learned AutoJudge classifier.

*`https://huggingface.co/yuhuili/EAGLE-LLaMA3.1-Instruct-8B`
*`https://github.com/SafeAILab/EAGLE/`

L Open-Ended Generation with LLM-as-a-Judge

As we discuss earlier in Section 3.1, the choice of important tokens largely depends on what counts as an "equivalent answer quality". For mathematical reasoning, we can check whether the alternative response leads to the same numerical answer (up to notation) or an equivalent formula. For programming, we compare how the two programs behave in testing. However, not all LLM tasks have formal quality criteria. Open-ended tasks like creative writing and question answering have implicit quality criteria that are difficult to evaluate. In this section, we investigate how AutoJudge generalizes to two open-ended problems: generative question answering and creative writing.

Case A: Question Answering. For this task, we evaluate Llama 3.x LLMs generative question answering on questions from the TriviaQA dataset [Joshi et al., 2017]. We use the "closed book" setup, where the LLM receives only the question itself as the prompt, without any additional information (i.e. no search results). Then, we use a more powerful LLM-as-a-judge to compare responses against target model outputs. We mine important tokens on 500 training samples and evaluate on a subset of 100 validation samples. Since this dataset was intended for short answers (typically 1 sentence), we stop generation on `\n` or after generating 120 tokens.

Case B: Creative Writing. We use the "Creative Writing" subset of Arena-Hard-Auto-v2.0 [Li et al., 2024b,a] that contains 250 creative writing tasks sourced from Chatbot Arena. These tasks include requests to write a poem on a certain topic, an imaginary dialogue transcript, or similar, in several languages. These tasks are even harder to judge than question answering, often boiling down to personal preference. We set 100 queries aside for evaluations and use the rest to mine important tokens, using the standard generation prompt from Arena-Hard-Auto-v2.

LLM-as-a-judge. We evaluate generations using a pairwise LLM-as-a-judge protocol inspired by Arena-Hard-Auto-v2.0. Under this protocol, the LLM "judge" does not test model answers against a pre-defined "correct" response, but compares two generations against each other. For our speculative decoding setup, we ask the LLM judge to compare target model generation against the output of speculative decoding with AutoJudge head. When comparing two generations, the LLM judge sees both responses A and B and chooses one of 5 options:

1. Assistant A is significantly better: [[A>>B]]
2. Assistant A is slightly better: [[A>B]]
3. Tie, relatively the same: [[A=B]]
4. Assistant B is slightly better: [[B>A]]
5. Assistant B is significantly better: [[B>>A]

When evaluating on creative writing, we found that the default LLM judge often produces different answers between consecutive runs, both with sampling and greedy decoding, likely due to numerical instability [Yuan et al., 2025]). To make our results more reliable we switch to a stronger `claude-sonnet-4-20250514` judge model and run it 3 times with majority voting:

- If there are more votes [[A>>B]] & [[A>B]] than the opposite, then A is better.
- If there are more votes [[B>>A] & [[B>A]]) than the opposite, then B is better.
- If there are equal number of votes favoring A and B or all votes are [[A=B]], we declare A and B are equivalent. Note that this rule does not differentiate between [[A>B]] and [[A>B]] as we found them to be uninformative.

Mining important tokens. We mine important tokens using Algorithm 1 with one change: in L12 (**if** $a \equiv \hat{a}$ **then** ...), we ask the LLM judge to compare the alternative response $\hat{y}$ (with the draft token) against y. If the new response is rated strictly worse, we label the corresponding draft token as important and roll back to the target token. Otherwise (better or equal), we keep the draft token and label it unimportant, same as in the original algorithm. Note that the LLM judge compares **not** against the original target model response (L4 Alg. 1), but against the current running response y that contains unimportant draft tokens from previous iterations (L14). This change would have no effect for tasks with formal $a \equiv \hat{a}$ criteria from math and programming benchmarks due to transitivity. However, for LLM-as-a-judge, this results in more accurate labels and improves classifier accuracy. We only use this for training, not evaluation.

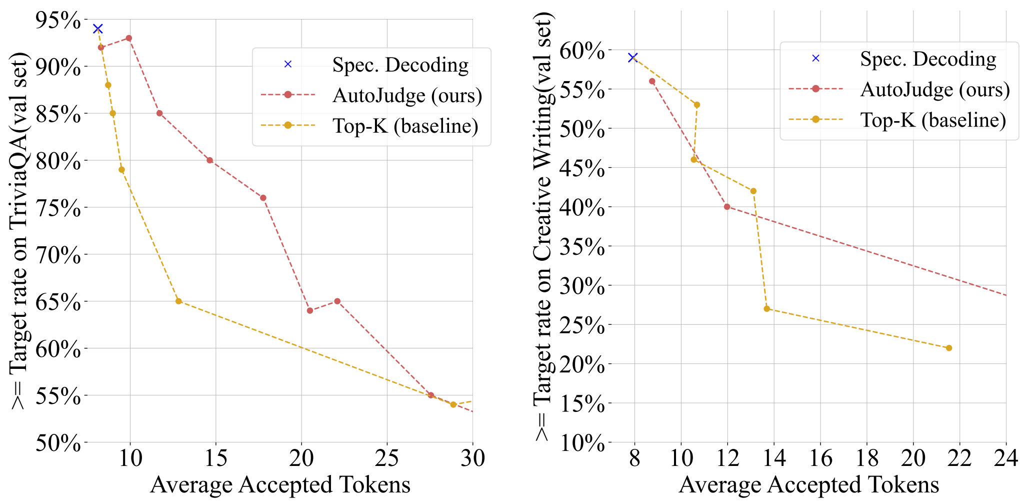


Figure 13: AutoJudge performance on open-ended generation with LLM-as-a-judge evaluation. (Left) closed-book generative question answering on TriviaQA questions (Right) creative writing on Arena-hard-Auto-v2.0 queries. Both use Llama-3.2-1B draft / Llama-3.1-8B target models (Instruct). Creative writing was later found to have noisy labels (>70% mismatches judged non-equivalent).

Results. We report our evaluations on question answering and creative writing in Figure 13 left and right, respectively. Similarly to Sections 4.1 and 4.2, we evaluate AutoJudge against the top-K baseline in terms of i) the average number of accepted tokens for greedy decoding and ii) the rate at which our outputs were equal or better than the target model. We use the same Claude Sonnet 4 judge and best of 3 voting as specified above. Note that the lossless decoding results are not exactly at 0.5 quality because some response pairs were deemed equal, i.e. [[A=B]].

The results suggest that AutoJudge classifier outperforms traditional speculative decoding on question answering, but not on creative writing. Upon closer inspection, we found that the creative writing benchmark has a very high rate of important tokens — over 70% mismatching tokens result in non-equivalent answers, much larger than all other problems. We attribute this anomaly to creative writing being inherently subjective, resulting in noisy LLM judgements. With such high rate of important tokens, there is little room for speed up with AutoJudge. To summarize, we found that AutoJudge can be indeed be used for open-ended problems with the LLM-as-a-judge paradigm, outperforming the baseline in some but not all cases.

M Trained Classifier Transfer Between Tasks

M.1 From GSM8K to MATH-hard Subset

To evaluate the task transfer capability of AutoJudge, we test whether a classifier trained on one mathematical reasoning dataset can generalize to a different, more challenging dataset. For this evaluation, we use the AutoJudge classifier trained on GSM8K 0-shot with the Llama-3.2-1B-Instruct draft / Llama-3.1-8B-Instruct target model pair (from Section 4.1) and apply it to the MATH-hard subset from the LLama Codebook [*] based on the lm-evaluation-harness benchmark [Gao et al., 2021]. The MATH-hard dataset contains significantly more challenging mathematical problems compared to GSM8K, requiring more advanced mathematical reasoning. We follow the standard evaluation protocol from the lm-evaluation-harness framework and report exact match accuracy as the downstream metric.

The results are presented in Figure 14 (left). While AutoJudge can still accept more tokens than standard speculative decoding, the accuracy-speed trade-offs are less favorable than the top-K baseline across most operating points. This suggests that the patterns of important tokens learned from grade-school math problems (GSM8K) might not transfer well to more advanced math reasoning tasks.

[*]We reproduce the setup and get the problems from the official Meta evaluations repository.

M.2 From GSM8K to Long Context

To test AutoJudge on long-context tasks, we construct a synthetic long-context mathematical reasoning benchmark based on GSM8K. For each test problem, we concatenate 250 randomly sampled GSM8K questions (without their solutions) as context, followed by a single target question that the model must solve. The resulting prompts contain approximately 8-10K tokens before applying the chat template, significantly longer than the standard GSM8K setup.

We evaluate the Llama-3.2-1B-Instruct draft / Llama-3.1-8B-Instruct target model pair using the AutoJudge classifier trained on hiddens mined described long context GSM problems. The results are presented in Figure 14 (right). AutoJudge maintains its ability to accept more tokens than standard speculative decoding in this long-context regime, achieving better accuracy while maintaining a similar number of accepted tokens compared to the Top-K baseline.

These results demonstrate that AutoJudge classifiers trained on standard-length examples can generalize to significantly longer contexts, suggesting that the notion of "important tokens" for mathematical reasoning remains consistent regardless of the input length.

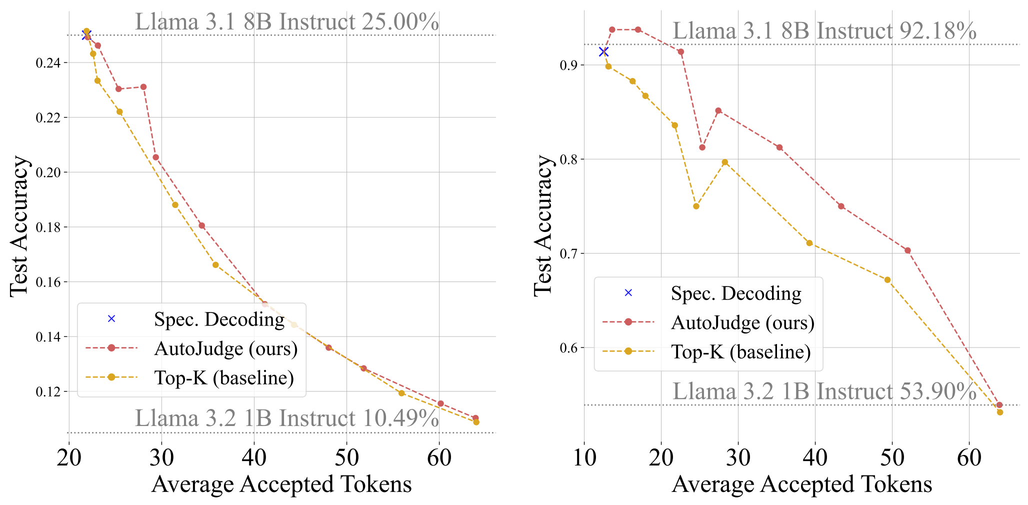


Figure 14: (Left) Task transfer: accuracy vs. accepted tokens on MATH-hard using GSM8K-trained classifier. (Right) Long-context performance: accuracy vs. accepted tokens on GSM8K with 8-10K token prompts. Both use Llama-3.2-1B draft / Llama-3.1-8B target setup (Instruct).

N Manual Annotation

To compare our automated approach with the manual annotation procedure from Judge Decoding (Bachmann et al. [2025]), we conduct an experiment following their methodology as closely as possible. Unfortunately, the original paper does not provide public access to their annotated dataset or source code, which limits direct comparison. Therefore, we recreate their data collection and training procedure based on the methodology described in Section 4.1 of their paper.

Dataset construction. We curate a dataset of 167 high-quality questions from two sources: the `ARC-Challenge` benchmark Clark et al. [2018] (25 questions) and the `Alpaca` dataset Taori et al. [2023] (142 questions, split between 71 mathematical reasoning problems and 71 coding problems). Since the exact question indices used in the original Judge Decoding work are not publicly available, we manually selected challenging and diverse questions from each dataset that represent a range of difficulty levels similar to those described in the original paper.

For each question, we generate responses using three different models from the Llama family: Llama-3.2-1B-Instruct, Llama-3.1-8B-Instruct, and Llama-3.1-70B-Instruct. This produces 501 total generated responses with varying quality levels, providing a diverse set of correct and incorrect continuations for training.

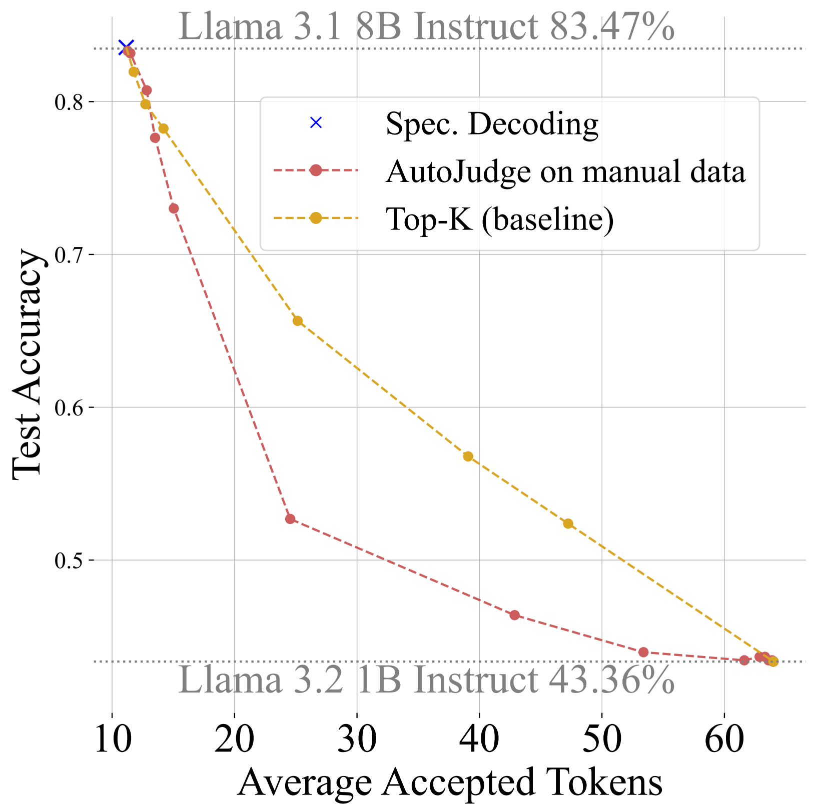


Figure 15: GSM8K 0-shot evaluation of Llama-3.2-1B draft / Llama-3.1-8B target model pair comparing AutoJudge with manual annotation mining (following [Bachmann et al., 2025]) against Top-K baseline.

Manual annotation. Following the procedure described in "Dataset Curation" in Section 4.1 of [Bachmann et al., 2025], we manually annotate each generated response to identify the precise location where errors occur. We encode all responses using the Llama-3.1-8B-Instruct model to extract hidden state representations, then manually label each token as either correct or incorrect based on whether accepting that token would lead to a wrong final answer. This annotation process required approximately 8-10 hours of careful manual review by the authors.

Training and evaluation. Using these manually annotated hidden states, we train AutoJudge classifier with the same architecture and hyperparameters described in Section 4.1 and Appendix B. We then evaluate the resulting "manually-annotated" AutoJudge classifier on GSM8K 0-shot with the Llama-3.2-1B-Instruct draft / Llama-3.1-8B-Instruct target setup, comparing against Top-K baseline. The results are presented in Figure 15. We observe that the manually-annotated classifier performs substantially worse than Top-K baseline and, consecutively, our automatically-mined approach. The manually-annotated classifier achieves lower acceptance rates while exhibiting greater accuracy degradation, suggesting that manual annotation is both less effective and more labor-intensive. We attribute this gap to several factors: (1) human annotators may inconsistently label edge cases or fail to account for how subsequent tokens interact, (2) the manual process is prone to errors over hundreds of annotations, and (3) our automated search procedure (Algorithm 1) systematically tests token combinations in a way that better reflects actual inference conditions. These results validate that our automated mining procedure not only eliminates the substantial human effort required for manual annotation (8-10 hours in our case), but also produces more robust training data that leads to better classifier performance.

O Correlation between Draft Model and AutoJudge Classifier Probabilities

One reasonable heuristic for speculative decoding is to consider a token important if a model is certain in its generation, measured in terms of that token's probability. Here, we check whether the AutoJudge classifier is similar to that heuristic. To that end, we measure the correlation between the Llama 3.2 1B Instruct draft model's probability of the chosen draft token and the corresponding AutoJudge classifier probability on the 1B/8B model pair using the GSM8K dataset (0-shot setting). The resulting Pearson correlation varies within approximately ± 0.3 per generation, and the full-sample covariance is -0.073. This suggests that AutoJudge classifier does not act on that heuristic.

P Hardware Configurations

We run our experiments primarily on A100-SXM4 GPUs with 80GB DRAM on servers with dual Epyc 7742 CPU and 1TiB RAM. For 8B/405B model pair we used 8 NVIDIA H100-SXM5 80GB GPUs.

For model pairs that do not fit on a single GPU, we use distributed inference with naive model parallelism (`device_map="auto"`) when using `transformers` and tensor parallelism for vLLM experiments.

The actual time per experiment varies by the dataset and model pair: 1B draft / 8B target model pair takes, on average, 65.6 seconds to process a GSM8K example, whereas the 8B draft / 70B target takes up an average of 706.4 seconds per sample. On LiveCodeBench, the same 8B draft / 70B target model takes up 449 seconds per sample. Since Algorithm 1 can run independently for each sample, we were able to run our code on low-priority preemptible hardware. However, this also makes it hard to measure the exact amount of computations used in our experiments since some of them were lost due to preemption. For running on a single A100/H100 server, please refer to the time per sample above to estimate the total runtime requirements. Please also note that there are ways to mine important tokens more efficiently using APIs (below).

Q Power consumption

To estimate the energy consumption of AutoJudge, vanilla speculative decoding, and sequential decoding, we run each inference method on a 10% sample of the GSM8K test set using vLLM. We measure real-world power usage on Llama 3.2 1B draft / 3.1 8B target models with a single A100-SXM4-80GB GPU (Watts) as reported by nvidia-smi, and multiply it by the mean inference time. This represents the GPU-reported power consumption, before adjusting for PSU inefficiency (equally for AutoJudge and baselines) and will vary between GPU types. For convenience, we convert all results to kJ, similar to Maliakel et al. [2025].

Table 11: Estimated energy consumption of running inference on 10% of GSM8K test set.

Method	**Autoregressive**	**Speculative Decoding**	**AutoJudge**
Power Usage (kJ)	87	43	37

R Scaling Up Algorithm 1 with API Calls

We would also like to mention that it is possible to scale up the important token discovery in AutoJudge by reframing it in terms of API calls. Note that the search algorithm only ever runs regular generation (greedy or sampling) with the target model and runs parallel forward pass on θ_{draft}.

Hence, we can run Algorithm 1 by replacing `GENERATE(...)` on lines 4 and 10 with a call to an LLM generation API with the specified input. With this reframing, we can mine important tokens by querying LLM API providers even if they cannot inference the large target model locally. This can help in a number of use cases: for instance, when developing a speculative decoding algorithm for use with offloading [Svirschevski et al., 2024, Miao et al., 2023].

Additionally, modern open-source inference libraries for LLMs often expose an an OpenAI-compatible API. This way, one can run the important token mining algorithm efficiently over a pool of LLM inference servers, taking advantage of server-side batching and optimized kernels.

Note, however, that this regime requires tokenizing and detokenizing the target model's messages received from API calls, since most public services operate on non-tokenized text strings. Since there are several ways to spell the same text with a given BPE merge table, this can sometimes lead to unintentional "token healing".

In our repository, we provide a variant of the important token mining algorithm that leverages the Together Inference API to run the target model. This enables training the AutoJudge classifier even on models that cannot be hosted locally. As a demonstration, we conducted an experiment using Llama 3.1-405B-Instruct (accessed via API) as the target model and Llama 3.1-8B-Instruct as the mining model. The experiment required $\approx$ \$800 in API credits. This example illustrates that Algorithm 1 is applicable to models available exclusively through an API. Results obtained for this model pair are presented in Table 2 (left).

S Dataset and Model Licenses

In this section, we summarize our use of licensed models and datasets.

- The GSM8K benchmark [Cobbe et al., 2021] is licensed under the MIT License;
- The LiveCodeBench benchmark [Jain et al., 2024] is licensed under the MIT License;
- Llama 3.1 and 3.2 models Dubey et al. [2024] are under the Llama Community License Agreement.

Our work would also be impossible without open-source software including (but not limited to) PyTorch [Paszke et al., 2019], Hugging Face Transformers [Wolf et al., 2019], vLLM [Kwon et al., 2023] and hundreds of other open-source packages in the Python data science & machine learning ecosystem. Enumerating and acknowledging all these individual packages would take up several pages, but they can be recovered automatically from our repository.