Test-Time Speculation

Avinash Kumar* **Sujay Sanghavi** **Poulami Das**
Department of Electrical and Computer Engineering
The University of Texas at Austin

Abstract

Speculative decoding accelerates LLM inference by using a fast draft model to generate tokens and a more accurate target model to verify them. Its performance depends on the *acceptance length*, or number of draft tokens accepted by the target. Our studies show that the acceptance length of even state-of-the-art speculators, like DFlash, EAGLE-3, and PARD, degrades with generation length, reaching values close to 1 (i.e., no speedup) within a few thousand output tokens, making speculators ineffective for long-response tasks. Acceptance lengths decline because most speculators are trained offline on short sequences, but are forced to match the target on much longer outputs at inference, well beyond their training distribution. To address this issue, we propose *Test-Time Speculation (TTS)*, an online distillation approach that continuously adapts the speculator at test time. TTS leverages the key insight that the token verification step already invokes the target model for each draft token, providing the training signal needed to adapt the draft at no additional cost. Treating the draft as the student and the target as the teacher, TTS adjusts the draft over several speculation rounds, with each update improving the draft's accuracy as generation proceeds. Our results across ten models from the Qwen-3, Qwen-3.5, DeepSeek-R1, GPT-OSS, MiniMax, and Llama3.1 families show that TTS improves acceptance lengths over state-of-the-art speculators by up to 72% and 41% on average, with the benefits scaling with generation lengths.

1 Introduction

Large Language Models (LLMs) can perform various tasks, such as summarization (Stiennon et al. [2020]), code generation (Chen et al. [2021a]), content creation (Brown et al. [2020]), etc., and are becoming increasingly ubiquitous. However, they pose critical performance concerns because tokens can only be generated auto-regressively. Speculative decoding (Leviathan et al. [2023]) is a widely adopted inference technique that breaks this dependency by using a fast but less accurate draft model to quickly generate a sequence of tokens and an accurate but slow target model to verify these tokens in parallel. The draft model optimizes for low latency, whereas the target model *ensures* that the final output tokens *exactly match* those that it would have produced auto-regressively. The performance of speculative decoding depends on the number of draft tokens accepted by the target model per round, also known as the *acceptance length*. In practice, *higher acceptance lengths are desirable*.

State-of-the-art speculators improve acceptance lengths by increasing the accuracy of the draft model. EAGLE-3 (Li et al. [2025]) produces draft tokens using a transformer layer that conditions on multi-level hidden states from the target model. PARD (An et al. [2025]) appends mask-token placeholders to the current sequence and fine-tunes a pretrained auto-regressive draft model to predict all masked positions in parallel. DFlash (Chen et al. [2026b]) uses a block-diffusion drafter and injects fused multi-layer target features into the speculator's KV cache. However, although effective for tasks producing only a small number of output tokens, these methods do not retain their benefits for *long-response tasks* producing thousands of tokens. For example, Figure 1(a) shows the average

*Correspondence: avinkumar@utexas.edu, Project page: https://avinkumarut.github.io/TTS/

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

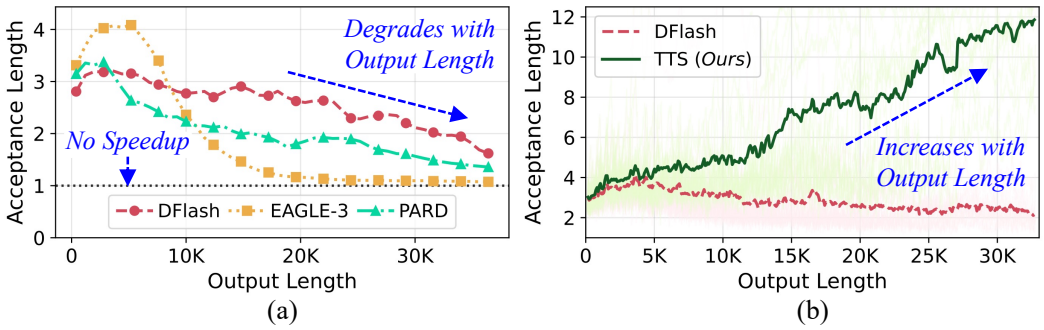


Figure 1: (a) Acceptance length (AL) for the LiveCodeBench dataset on Qwen3-8B with increasing output lengths for three speculators. Beyond a certain number of output tokens, the draft model fails to accurately approximate the target model and the AL becomes vanishingly low (too close to 1), yielding practically no benefits. (b) Our proposal, *Test-Time Speculation (TTS)*, adapts the draft model in real time to address this limitation and consistently improves acceptance lengths.

acceptance length with increasing number of output tokens for LiveCodeBench (Jain et al. [2024]) on Qwen3-8B using the above speculators. The acceptance length decreases with output length. More importantly, after a certain point, speculative decoding yields diminishing returns because acceptance lengths are too low to offer any practical speedup benefits. For instance, the acceptance length reduces to 1.1 beyond $\sim 20K$ tokens for EAGLE-3, a widely used speculator in industrial serving systems (Tang et al. [2025]), defeating the purpose of speculative decoding at that point. As long-response tasks dominate many real-world LLM deployment scenarios (Jain et al. [2024], Rein et al. [2023]), improving acceptance lengths is critical to retain the benefits of speculative decoding.

The degradation in acceptance length stems from a mismatch between the draft's training and inference distributions. State-of-the-art speculators are small networks trained on short sequences (Chen et al. [2026b], Li et al. [2025]), typically less than $2K$ tokens. However, at inference time, the draft is forced to match the target on much longer sequences, causing its predictions to progressively diverge from the target model. Consequently, a larger number of draft tokens become inaccurate and are eventually rejected by the target model, decreasing acceptance lengths. Closing this gap requires adapting the draft model to match the target distribution at inference time.

Our Proposal: We propose *Test-Time Speculation (TTS)*, an online distillation approach that adapts the draft model to match the target's distribution at inference time. TTS exploits the key insight that each token verification step already produces the supervision needed to adapt the draft, as the target model computes its full distribution over all generated draft tokens. TTS leverages this signal by treating the target as the teacher, the draft as the student, and the target's distribution over the draft tokens as the distillation sample on which the draft performs an optimization step. Our evaluations across ten state-of-the-art models show that TTS consistently improves acceptance lengths by up to 72% and 41% on average, with the benefits scaling with generation lengths, as shown in Figure 1(b).

Overall, this paper makes the following contributions:

1. We observe that the efficacy of state-of-the-art speculators, like DFlash, EAGLE-3, and PARD, degrades with generation length, with acceptance lengths decreasing to values too close to 1 within just a few thousand tokens, making them ineffective for real-world long-response tasks.

2. We demonstrate that acceptance length degrades because draft models are trained on short sequences, but are forced to match the target distribution on much longer sequences at inference.

3. We propose *Test-Time Speculation (TTS)*, an online distillation approach that leverages the insight that the token verification step already produces the supervision signal needed to adapt the draft model at inference time. We enable this by using the target as the teacher, the draft as the student, and the target's distribution over the drafted tokens in each round as distillation samples.

4. Our experiments across three state-of-the-art speculators show that TTS improves acceptance length by up to 72% and 41% on average, with benefits scaling with increased generation lengths.

2 Problem: Acceptance Length Decreases With Output Length

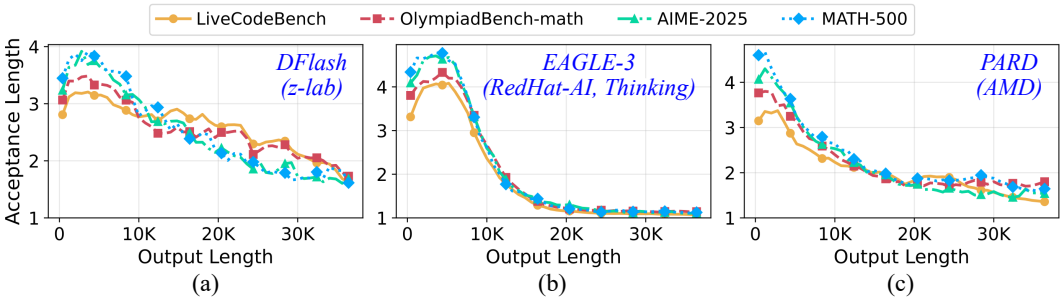


Figure 2: Acceptance length of four tasks using (a) DFlash, (b) EAGLE-3, and (c) PARD speculators on the Qwen3-8B model. Acceptance length consistently decreases with increasing generation length.

Although effective for tasks generating short responses, such as GSM8K (Cobbe et al. [2021]) and HumanEval (Chen et al. [2021b]), our experiments show that state-of-the-art speculators are ineffective for long-response generation tasks producing thousands of output tokens. Long-response tasks, such as content creation (Bai et al. [2024]), long-form reasoning (Guo et al. [2025]), and agentic workflows (Yao et al. [2023]), are critical in modern real-world LLM deployment scenarios and continue to dominate most inference serving systems (Qin et al. [2024], Agrawal et al. [2024]). However, our experiments show that acceptance length decreases with increasing number of output tokens even with state-of-the-art speculators. Figure 2(a-c) shows the mean acceptance length across 64 sequences for four different tasks (LiveCodeBench, OlympiadBench-Math, AIME 2025, MATH-500) on the Qwen3-8B model using three state-of-the-art speculators (DFlash, EAGLE-3 and PARD). We observe that acceptance lengths decrease consistently for all tasks and speculators. For the MATH-500 dataset, the average acceptance length is 3.7 for the first $10K$ output tokens, but sharply decreases to 1.5 for the last $10K$ output tokens in the sequence. In fact, when EAGLE-3 is used as the speculator, acceptance length reduces to 1.1 for all the tasks after generating $\sim 20K$ tokens and therefore, practically yields no speedup.

Moreover, the problem persists even when larger models are used. Figure 3 shows that acceptance length consistently degrades for the same four tasks even when using the most recent DFlash speculator when larger models, such as Qwen3.5-35B, Qwen3.6-35B, and Qwen3.5-122B, are used. The acceptance length decreases from 15 to 1.7 for the AIME 2025 task on the Qwen3.6-35B model within a span of $25K$ tokens, with a sharp decline from 11 to 3.8 going from $20K$ to $28K$ tokens.

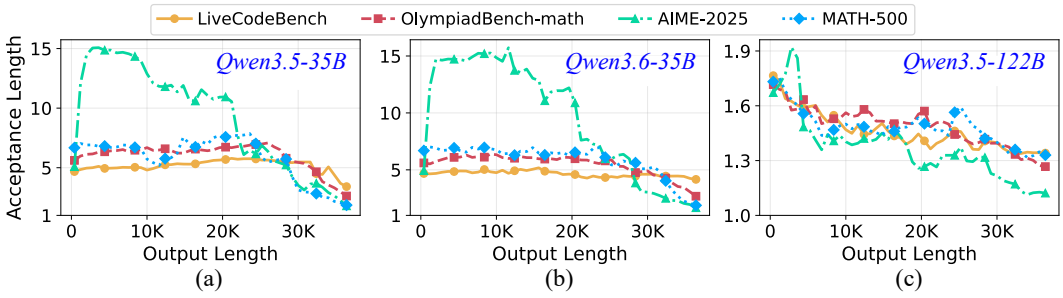


Figure 3: Acceptance length of four tasks using the DFlash speculator on (a) Qwen3.5-35B, (b) Qwen3.6-35B, and (c) Qwen3.5-122B models. Acceptance length consistently reduces with increasing output generation lengths even when larger models are used.

Higher acceptance lengths are necessary to attain meaningful performance benefits with speculative decoding. Acceptance length, in turn, depends on the accuracy of the draft model. We therefore diagnose why the draft model degrades as the generation length increases.

3 Diagnosis: Why The Draft Model Degrades At Higher Output Lengths

The draft is a small model trained offline to match the target model's token distribution. Training is typically performed on short sequences $\leq 2K$ tokens (Chen et al. [2026b], Li et al. [2025]) drawn from instruction-following datasets such as ShareGPT (ShareGPT [2023]) and UltraChat (Ding et al. [2023]). However, at inference time, the draft is forced to match the target on much longer sequences, causing its predictions to progressively diverge from the target and acceptance length to degrade.

Figure 4(a) shows that the distribution entropy $H(p^i_{\text{target}})$ of the target model decreases with output length, indicating that the target model becomes increasingly confident in its predictions. Therefore, with each additional token, the output of the target model becomes more predictable. In contrast, Figure 4(b) shows that the cross-entropy of the draft on the target's $Top{-}1$ token $(-\log p^i_{\text{draft}}(t^*_i))$ increases monotonically from 2.8 to 6.4 nats over the same range. At a generation length of $12K$ tokens, the draft assigns a negligible probability of $1/600$ to the target's $Top{-}1$ token. Thus, the draft's generated tokens *drift* farther from the target with output length, even though the target model itself becomes more decisive. Furthermore, Figure 4(c) shows that cross-entropy increases steeply irrespective of whether the token ID was previously unseen or had already been generated in the sequence. To summarize, the draft loses agreement with the target because its offline training on short sequences leaves it miscalibrated with the target at higher output lengths. *Closing this gap requires adapting the draft model to the target's trajectory at inference time.*

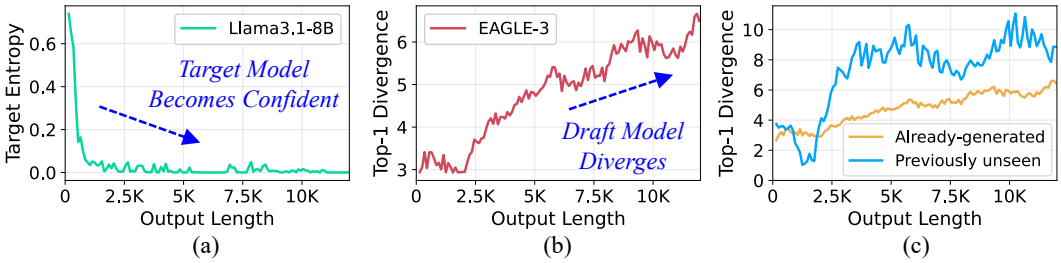


Figure 4: Distribution entropy (in nats) for Llama3.1-8B (target) with EAGLE-3 (draft). (a) Target entropy decreases with output length, indicating that the target becomes increasingly confident in its predictions. (b) Over the same range, the draft's cross-entropy on the target's $Top{-}1$ token increases sharply. (c) This divergence persists regardless of whether the token appeared earlier or was unseen.

4 Our Proposal: Test-Time Speculation (TTS)

The core algorithmic realization of our paper is that the existing speculative decoding workflow already has all the ingredients needed for online on-policy distillation, *if* one were willing to interleave generation with updates to the speculator. We now develop notation to formalize this intuition.

Setup: Let $p(x)$ denote the target model, i.e., the large base model that we want to generate with. Typically, this is a large auto-regressive model, and vanilla text generation from this model involves iteratively sampling from $p(\cdot)$:

$$x_{t+1} \sim p(X_{t+1} \mid x_{1:t}) \quad \text{(Vanilla auto-regressive generation)}$$

Here, we use the standard notation $x_{1:t}$ to denote the sequence $x_1, \ldots, x_t$. Each output token, therefore, requires an entire forward pass through the target model $p(\cdot)$, a cost that grows with model size and becomes prohibitive for state-of-the-art LLMs.

Standard speculative decoding pairs the target with a much smaller, and therefore faster, auxiliary model $q(\cdot)$, called the *draft*. The draft model (or speculator) itself need not be auto-regressive. The most capable modern speculators, including the EAGLE-3 and DFlash drafters considered in this work, also reuse the target model's intermediate hidden states. Token generation proceeds in rounds. In each round, the draft model first generates a canvas of tokens. Next, the target model verifies this entire canvas in parallel and "accepts" a prefix of the draft tokens. The next round begins from the last accepted token, and since the target model has already executed a forward pass over the entire canvas, its hidden states are consistent up to this new starting point.

Standard speculative decoding: Suppose that we are currently at time step t, i.e., tokens $x_{1:t}$ have been generated and verified by the target model, and we want to speculate-verify the next few tokens.

1. First, a canvas of tokens is generated from $q(\cdot)$, and then this canvas is evaluated by the target model. If the canvas length is C,

$$\text{Generate} \quad x_{t+1:t+C} \sim q\left(X_{t+1:t+C} \mid x_{1:t}\right) \quad \text{(Canvas tokens from drafter)}$$

2. Recall that the target model has already processed $x_{1:t}$. It now appends the canvas tokens $x_{t+1}, \ldots, x_{t+C}$ to this, and executes a single forward pass. This yields all the sequentially conditioned likelihoods for canvas tokens:

$$\text{Evaluate} \;\; p(x_{t+k}|x_{1:t+k-1}) \text{ for all tokens } 1 \leq k \leq C \;\; \text{(All in one forward pass)}$$

   Then, canvas tokens are considered from left to right for acceptance, as follows:

   $k \leftarrow 1$
   while $k \leq C$ **do**
   accept x_{t+k} with probability $\min\left\{1, \dfrac{p(x_{t+k}|x_{1:t+k-1})}{q(x_{t+k}|x_{1:t+k-1})}\right\}$
   if x_{t+k} rejected **then break**
   end if
   $k \leftarrow k+1$
   end while

3. Advance t to be the last accepted token, and go back to step 1.

The key metric of interest in speculative decoding is the **acceptance length**, i.e., the average number of tokens accepted in each round. The higher the acceptance length, the greater the speedup, as each round has the same computational cost regardless of how many tokens are accepted per round.

4.1 Test-Time Speculation

In the standard workflow above, each time a draft canvas is produced, we naturally have access to both the speculator's probabilities $q(x_{t+1}, \ldots, x_{t+C}|x_1, \ldots, x_t)$ and the target's probabilities $p(x_{t+1}, \ldots, x_{t+C}|x_1, \ldots, x_t)$. Our **main insight** is to treat the draft model as the student, the target model as the teacher, and the generated-and-evaluated canvas as a distillation-style training sample for the student. After each round of speculation and verification, our idea is to update the draft model via a single gradient step on a distillation-style loss—before the next canvas is generated. In practice, the draft need not be updated after every round; the update can be applied lazily every few rounds and pipelined on a separate CUDA stream. We study the impact of these choices in Section 6.

Formally, suppose as before that generation is at time t. Since the speculator is updated over time as generation progresses, we denote it by $q_t(\cdot)$ instead of just $q(\cdot)$. Initially, $q_t(\cdot)$ is set to the existing draft, i.e., $q_0 = q$. We now explain the update mechanism in detail:

First, we follow the standard workflow and *(a)* use this speculator $q_t(\cdot)$ to generate the canvas $x_{t+1:t+C}$, *(b)* evaluate the target model likelihoods on this canvas, and *(c)* determine the acceptance length τ. We now update $q_t(\cdot)$ to obtain the new speculator $q_{t+\tau}(\cdot)$. This is done via **one optimization step on the standard drafter distillation loss** on the canvas. Specifically, let $x_{\mathcal{C}} = x_{t+1}, \ldots, x_{t+C}$ denote the sequence of canvas tokens; the update then minimizes the following objective:

$$L_t(q) \;=\; \widetilde{\mathrm{KL}}\left(\, p(x_{\mathcal{C}}|x_{1:t}) \,||\, q(x_{\mathcal{C}}|x_{1:t}) \,\right) \;+\; \lambda\, \widetilde{\mathrm{KL}}\left(\, q_t(x_{\mathcal{C}}|x_{1:t}) \,||\, q(x_{\mathcal{C}}|x_{1:t}) \,\right) \tag{1}$$

The first term is an approximate KL divergence (specified below) between the target p and the new q, while the second is a regularization term that prevents q from drifting too far from q_t. Here

$$\widetilde{\mathrm{KL}}\left(\, p(x_{\mathcal{C}}|x_{1:t}) \,||\, q(x_{\mathcal{C}}|x_{1:t}) \,\right) \;=\; \sum_{k=1}^{C} w_k \,\mathrm{KL}(p(X_{t+k}|x_{1:t+k-1})||q(X_{t+k}|x_{1:t+k-1}))$$

and $\mathrm{KL}\left(\, p(X_{t+k}|x_{1:t+k-1}) \,||\, q(X_{t+k}|x_{1:t+k-1}) \,\right)$ is the standard KL divergence between the target and draft distributions at position $t+k$, conditioned on the prefix $x_{1:t+k-1}$. With uniform weights (w_k=1 for all k), this reduces to the standard distillation loss, with $p(\cdot|x_{1:t})$ as the teacher, $q(\cdot|x_{1:t})$ as the student, and $x_{\mathcal{C}}$ as the string on which the loss is calculated.

Finally, the w_k are position-dependent weights that decrease with k, as a rejected token invalidates all succeeding canvas tokens, making early positions more important. We reuse these weights directly from the training codebase of each speculator. Algorithm 1 gives an overview of the TTS algorithm.

Algorithm 1 Test-Time Speculation (TTS)

Input: Target model p, draft model q, learning rate η, regularizer λ, canvas size C
  Input prompt $x_{1:T_0}$ // Prompt of length T_0 tokens
  Max new tokens $L_{\max}$ // Generation length cap
  Weights $\{w_k\}_{k=1}^{C}$ // Per-position weights, matching the drafter's training recipe
Algorithm:
$t \leftarrow T_0,\ q_t \leftarrow q$
while $x_t \neq \text{EOS}$ **and** $t - T_0 < L_{\max}$ **do**
  Draft: sample canvas $x_{t+1:t+C} \sim q_t(\cdot \mid x_{1:t})$
  Verify: compute $p(x_{t+k} \mid x_{1:t+k-1})$ for $k = 1, \ldots, C$ in one forward pass of p
  Accept: run standard speculative rejection sampling; let τ be the number of accepted tokens
  Loss: $L_t(q) \leftarrow \widetilde{\mathrm{KL}}(p \,\|\, q) + \lambda\, \widetilde{\mathrm{KL}}(q_t \,\|\, q)$ // Distillation loss + regularization
  where $\widetilde{\mathrm{KL}}(a\|b) := \sum_{k=1}^{C} w_k\, \mathrm{KL}\big(a(X_{t+k} \mid x_{1:t+k-1}) \,\big\|\, b(X_{t+k} \mid x_{1:t+k-1})\big)$
  Update: $q_{t+\tau} \leftarrow q_t - \eta\, \nabla_q L_t(q)\big|_{q=q_t}$ // One gradient step on L_t
  $t \leftarrow t + \tau$
end while
return $x_{1:t}$

For EAGLE-3, we instead use the *LK loss*, following prior work (Samarin et al. [2026]) and the SpecForge (Li et al. [2026]) training recipe. Concretely, $\widetilde{\mathrm{KL}}$ in (1) is replaced with:

$$\widetilde{\mathrm{LK}}\left(\, p(x_{\mathcal{C}}|x_{1:t}) \,||\, q(x_{\mathcal{C}}|x_{1:t}) \,\right) = -\sum_{k=1}^{C} w_k \log \sum_{x_{t+k}} \min\left\{p(x_{t+k}|x_{1:t+k-1}), q(x_{t+k}|x_{1:t+k-1})\right\}$$

The overall loss retains the same form: a w_k-weighted sum over the C canvas positions.

5 Evaluation Methodology

Benchmark: We consider eight single-turn benchmarks, spanning competition mathematics (AIME 2024/2025, MATH-500, and OlympiadBench-Math), physics (OlympiadBench-Physics), graduate-level science (GPQA $\diamond$), theorem reasoning (TheoremQA), and coding (LiveCodeBench), all of which elicit output lengths ranging up to $32K$ tokens. We also consider multi-turn agentic tasks drawn from OpenThoughts (Team [2026]). We provide more details about these tasks in Appendix B.

Setup: We conduct all our experiments on a *commercial* Amazon EC2 G7e instance with 8×NVIDIA RTX PRO 6000 *Blackwell* GPUs, each with 96GB memory. We implement TTS on top of the public EAGLE-3 (Li et al. [2025]), DFlash (Chen et al. [2026b]), and PARD (An et al. [2025]) codebases.

Models and Baselines: We evaluate on ten state-of-the-art models spanning the Qwen-3 (Yang et al. [2025a]), Qwen-3.5 (Qwen Team [2026]), DeepSeek-R1-Distill-Qwen (Guo et al. [2025]), GPT-OSS (OpenAI [2025]), MiniMax (Chen et al. [2026a]) and Llama3.1 (Grattafiori et al. [2024]) families. We apply TTS to, and compare against, the official EAGLE-3, DFlash, and PARD checkpoints. Please refer to Appendix C for a detailed description of baselines and the exact HuggingFace identifiers.

Metrics: We evaluate TTS on mean acceptance length and *throughput*. Acceptance length measures how well the speculator matches the target distribution, and a higher value directly translates into fewer invocations of the target model. It is *hardware-independent*, decoupling speculator quality from the underlying software stack. Throughput, in contrast, depends on the inference stack and is sensitive to system-level optimizations like fused kernels and quantization. We do not report accuracy metrics as they *remain unchanged* since TTS only adapts the draft model and leaves the target untouched. Hence, the generated outputs remain consistent with the target model (Leviathan et al. [2023]).

Our comprehensive selection of benchmarks, models, and comparison against state-of-the-art speculators enables a rigorous evaluation of TTS across diverse real-world tasks and model sizes.

6 Experimental Results

6.1 TTS Is Generalizable Across Speculators, Models And Tasks

Table 1 shows the performance of TTS against DFlash. We observe that TTS consistently outperforms vanilla DFlash across all benchmarks and models, improving AL by up to 72% and 41% on average. We show that TTS generalizes across both EAGLE-3 and PARD in Appendix A.

Table 1: Acceptance length of TTS and DFlash across models and tasks. $\Delta(\%)$ denotes the relative improvement of TTS over DFlash. TTS consistently achieves higher acceptance length.

Dataset	Qwen/Qwen3-4B			Qwen/Qwen3-8B			Qwen/Qwen3.5-122B		
	DFlash	TTS (*Ours*)	$\Delta(\%)$	DFlash	TTS (*Ours*)	$\Delta(\%)$	DFlash	TTS (*Ours*)	$\Delta(\%)$
AIME 2024	3.3	**4.6**	38.0	3.3	**4.6**	38.5	2.9	**4.8**	62.2
AIME 2025	3.2	**4.7**	47.0	3.1	**4.2**	33.1	2.9	**4.9**	71.8
GPQA ◇	3.6	**5.2**	47.2	3.2	**4.7**	47.3	2.6	**3.5**	33.9
LiveCodeBench	3.0	**4.7**	55.2	2.9	**4.5**	55.2	2.4	**4.0**	63.5
MATH-500	3.7	**4.3**	13.8	3.8	**4.3**	13.3	3.4	**4.5**	32.2
OlyBench-Math	3.0	**4.4**	48.9	2.7	**4.1**	50.3	3.4	**4.7**	40.2
OlyBench-Phy	3.1	**4.5**	48.1	2.8	**4.3**	53.1	3.0	**3.9**	28.4
TheoremQA	3.6	**4.4**	23.4	3.5	**4.3**	22.8	3.3	**4.2**	25.1

6.2 Improvement Scales With Generation Length

TTS accumulates training signal across speculation rounds. Consequently, sequences with longer outputs provide more opportunity to refine the draft model and improve its accuracy further as generation lengths increase. Figure 5 shows that the improvement in acceptance length with TTS increases sharply with generation (or output) length. On AIME 2024, it only improves by 15% for the first $10K$ tokens, but the gap widens to 183% for the tokens in the $20{-}30K$ range. Similarly, on LiveCodeBench, the acceptance length only improves by 17% for the first $10K$ tokens but goes up to 245% for the tokens in the $20{-}30K$ range. With the increasing dominance of tasks requiring LLMs to produce long output responses, TTS provides a critical advantage in those regimes.

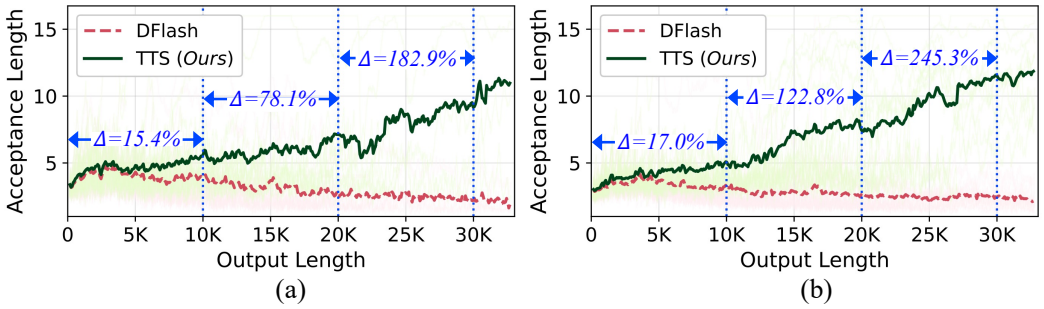


Figure 5: Acceptance length of TTS versus DFlash for (a) AIME 2024 and (b) LiveCodeBench on Qwen3-8B. The improvement $(\%)$ increases sharply with output length as TTS refines the drafter.

6.3 Beyond Single-Turn Reasoning: TTS In Agentic Workloads

Unlike the single-turn workloads considered so far, agentic workloads unfold over many *turns*. In each turn, the model typically emits an *action*, i.e., a call to an external tool, and the environment returns an *observation*, i.e., the output produced by the tool. Both are appended to the context for the following turn, and this repeats until the request is served. Such workloads are a particularly favorable setting for TTS: token generation continues across the entire session, so the speculator keeps accumulating training signal instead of resetting after a single response. Note that only the model's output tokens are speculated and verified, and thus only these are used to adapt the speculator.

Figure 6 shows the improvement in AL over turns on tasks drawn from OpenThoughts-Agent-SFT-100K (Team [2026]), orchestrated by the Terminus-2 agent protocol (Merrill et al. [2026]) with MiniMax-M2.5 (Chen et al. [2026a]) as the target. We observe that the improvement in AL grows monotonically with turns, from 3.5% at turn 2 to 86.0% at turn 15. While the AL of the baseline DFlash falls from 3.1 to 1.8 over the session, TTS sustains an AL between 3.3 and 3.4 throughout.

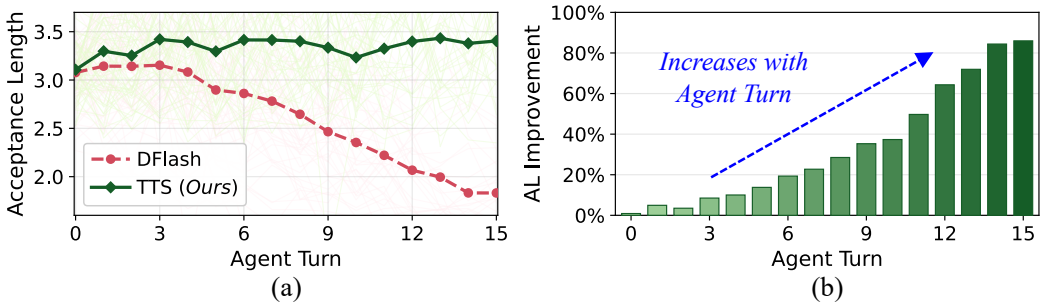


Figure 6: Acceptance length (AL) of TTS versus DFlash on multi-turn agentic tasks from OpenThoughts with MiniMax-M2.5 as the target. (a) The AL of DFlash degrades as the session progresses, whereas TTS sustains it across turns. (b) The improvement (%) of TTS over DFlash grows with turns, reaching 86% at turn 15, as TTS keeps adapting the draft throughout the session.

6.4 Extensive Offline Training Does Not Eliminate The Need For TTS

TTS improves AL even on speculators trained extensively on long-context data. To demonstrate this, we evaluate TTS on the GPT-OSS-120B model using two vendor-released EAGLE-3 checkpoints. Both checkpoints are extensively trained on long-context data, with traces extending well beyond $100K$ output tokens. Table 2 shows that TTS improves AL by 39.0% and 44.3% respectively. Thus, extensive offline training alone does not eliminate the need for online adaptation with TTS.

Table 2: Acceptance length of EAGLE-3 and TTS on GPT-OSS-120B for two vendor-released speculators that are extensively trained on long-context data. $\Delta(\%)$ denotes the relative improvement of TTS over EAGLE-3. TTS improves AL across all benchmarks and both speculators.

Dataset	nvidia/gpt-oss-120b-Eagle3-v3			nvidia/gpt-oss-120b-Eagle3-long-context		
	EAGLE-3	TTS (*Ours*)	$\Delta(\%)$	EAGLE-3	TTS (*Ours*)	$\Delta(\%)$
AIME 2024	4.9	**6.4**	28.7	4.0	**5.9**	47.1
AIME 2025	4.6	**6.2**	34.1	3.9	**6.4**	62.8
GPQA ◇	4.9	**6.7**	37.7	5.0	**6.4**	27.3
LiveCodeBench	4.0	**5.8**	43.0	3.8	**6.0**	57.4
MATH-500	5.2	**6.8**	31.7	4.6	**6.6**	42.3
OlyBench-Math	4.5	**6.4**	43.9	4.1	**5.9**	43.3
OlyBench-Phy	3.8	**5.6**	48.8	4.2	**5.6**	31.6
TheoremQA	4.4	**6.4**	44.0	4.5	**6.4**	42.5

6.5 Do Acceptance Length Improvements Translate To Throughput?

By default, TTS updates the speculator after each speculate-verify round. While this improves acceptance length, it introduces the overhead of performing a backward pass and an optimizer step on the speculator. Thus, there exists a trade-off between acceptance length and overall throughput. We navigate this trade-off through scheduling and system-level optimizations. *First*, instead of performing the gradient update in every round, we run it every $\mathcal{S}$ (update stride) rounds. Each step *only* uses the latest round's training signal, without any accumulation across skipped rounds. *Second*, to ensure this sporadic update does not fall on the critical path, we offload the update to a dedicated CUDA stream that runs in parallel with the next $\mathcal{S}-1$ speculate-verify rounds. A synchronization barrier is inserted at the end of these $\mathcal{S}-1$ rounds to ensure the update is complete before the speculator

generates the next canvas of draft tokens. Together, these optimizations improve the applicability of TTS in real-world deployment settings. Please see Appendix D for more implementation details.

Table 3 shows that although $\mathcal{S}{=}1$ (lowest possible stride) achieves the highest acceptance length since the speculator is updated after each round, $\mathcal{S}{=}5$ maximizes throughput ($1.54\times$ on average and up to $1.71\times$ over DFlash) as it operates at the sweet spot in the acceptance length–throughput trade-off.

Table 3: Acceptance length and throughput of TTS relative to DFlash on Qwen3.5-122B with update stride ($\mathcal{S}$). While $\mathcal{S}{=}1$ has the highest acceptance length, $\mathcal{S}{=}5$ achieves the best throughput.

Benchmark	Acceptance Length / Throughput (Tokens/Sec)					
	DFlash	$\mathcal{S}{=}20$	$\mathcal{S}{=}10$	$\mathcal{S}{=}5$	$\mathcal{S}{=}2$	$\mathcal{S}{=}1$
AIME 2024	2.9 / 28.2	3.3 / 42.6	3.5 / 44.8	3.8 / **46.8**	4.3 / 46.7	**4.8** / 40.5
AIME 2025	2.9 / 27.6	3.7 / 43.2	3.8 / 44.6	4.3 / **45.7**	4.7 / 45.4	**4.9** / 37.5
GPQA ⋄	2.6 / 24.1	3.0 / 30.3	3.1 / **32.5**	3.3 / 32.3	3.5 / 30.3	**3.5** / 24.4
LiveCodeBench	2.4 / 24.9	3.1 / 31.3	3.2 / **31.5**	3.4 / 31.3	3.8 / 30.4	**4.0** / 25.7
MATH-500	3.4 / 29.7	3.8 / 43.7	4.0 / 46.5	4.2 / **49.7**	4.3 / 43.5	**4.5** / 35.9
OlyBench-Math	3.4 / 27.7	3.9 / 39.4	4.0 / 42.0	4.3 / **44.4**	4.6 / 41.4	**4.7** / 33.2
OlyBench-Phy	3.0 / 27.9	3.4 / 34.4	3.6 / **36.5**	3.7 / 36.4	**4.2** / 35.7	3.9 / 27.8
TheoremQA	3.3 / 30.0	3.7 / 41.9	3.7 / 42.5	4.1 / **51.3**	4.1 / 44.5	**4.2** / 30.4
Mean Speedup ($\times$)	1.0 / 1.0	1.16 / 1.39	1.21 / 1.46	1.30 / **1.54**	1.41 / 1.45	**1.43** / 1.15

6.6 Parameter-Efficient TTS For Batched Inference Serving

The experimental results so far adapt the speculator for a single in-flight request, whereas production systems serve many requests concurrently. Batched serving with TTS is challenging for two reasons. *First*, by default, all requests in the batch share a single set of speculator weights. As TTS specializes the speculator to the current prefix per request, an update driven by one request corrupts the draft distribution of every other request in the batch. *Second*, maintaining a per-request private copy of the speculator incurs significant memory overheads. Table 4 shows that a full-weight copy of the Qwen3.5-122B-A10B DFlash speculator along with its optimizer state costs $\sim$2445MB per request on each GPU at a tensor parallelism (TP) of four. At a batch size (BS) of sixty-four, this overhead scales to $\sim$156GB per GPU, which is $2.5\times$ the footprint of the `bf16` target model itself.

Per-request low-rank (LoRA) adapters: We address the above challenges by freezing the speculator's pretrained weights and instead specializing through per-request LoRA adapters (Hu et al. [2021]). In this design, TTS augments each adapted projection in the speculator with a rank-r adapter and *only trains the adapter's parameters*, leaving the speculator's weights *untouched* and therefore *safely shared across the batch*. Table 4 shows that specializing through LoRA adapters incurs $\sim 86\times$ lower memory overhead than adapting the entire speculator. At $r{=}16$, the adaptation state per request is only 28.6MB per GPU at the same tensor parallelism. Consequently, a batch size of sixty-four only requires 1.83GB per GPU, which is $0.03\times$ the target model's footprint.

Table 4: Per-request adaptation state for the Qwen3.5-122B-A10B DFlash speculator at TP=4. LoRA adapters reduce adaptation state by $\sim 86\times$, enabling batched serving with TTS.

Adapted Parameters	Trainable Parameters Per Request	Adaptation State Per GPU (GB)			
		$BS{=}8$	$BS{=}16$	$BS{=}32$	$BS{=}64$
Full speculator weights	174.6M	19.6	39.1	78.2	156.5
LoRA adapters ($r{=}16$)	2.0M	0.23	0.46	0.91	1.83

We also note that state-of-the-art inference engines such as SGLang (Zheng et al. [2024]) and vLLM (Kwon et al. [2023]) already support a similar execution model for serving distinct LoRA adapters concurrently over a shared base model (Chen et al. [2024], Sheng et al. [2024]). Table 5 shows that TTS with per-request LoRA adapters consistently outperforms the baseline DFlash speculator across batch sizes. We report additional batched serving results in Appendix F.

Table 5: Throughput (Tokens/Sec) of TTS relative to DFlash (z-lab/Qwen3.5-122B-A10B-DFlash) on Qwen3.5-122B-A10B with $\mathcal{S}$=10. Values in parentheses denote speedup over DFlash. TTS consistently improves throughput by $1.5-2.2\times$ across batch sizes and benchmarks.

Benchmark	Approach	BS=8	BS=16	BS=32	BS=64
AIME 2024	DFlash	241.2	313.5	423.1	384.9
	TTS (*Ours*)	**428.6** (1.78×)	**551.7** (1.76×)	**930.9** (2.20×)	**585.1** (1.52×)
AIME 2025	DFlash	213.2	292.7	434.8	272.8
	TTS (*Ours*)	**455.5** (2.14×)	**579.7** (1.98×)	**895.2** (2.06×)	**549.8** (2.02×)
GPQA ⋄	DFlash	192.0	257.1	428.2	289.7
	TTS (*Ours*)	**381.8** (1.99×)	**543.9** (2.12×)	**728.7** (1.70×)	**521.6** (1.80×)
LiveCodeBench	DFlash	199.3	264.6	378.3	202.9
	TTS (*Ours*)	**377.0** (1.89×)	**500.4** (1.89×)	**809.2** (2.14×)	**445.9** (2.20×)

6.7 Why TTS Works: A Distribution-Mismatch Perspective

The drafter $q(\cdot)$ is trained to approximate the target $p(\cdot)$, such that $q(\cdot \mid x) \approx p(\cdot \mid x)$ for prefixes x drawn from a fixed training distribution. At inference time, however, the prefix $x_{1:t}$ grows well beyond the training distribution for real-world long-response tasks. Consequently, the approximation $q(\cdot \mid x_{1:t}) \approx p(\cdot \mid x_{1:t})$ degrades, causing a divergence between the draft and the target, as shown in Figure 7. Each update in TTS distills the current $p(\cdot \mid x_{1:t})$ back into the drafter, restoring $q_t(\cdot \mid x_{1:t}) \approx p(\cdot \mid x_{1:t})$ at the current prefix. This progressive restoration explains the compounding improvements with TTS at longer generation lengths.

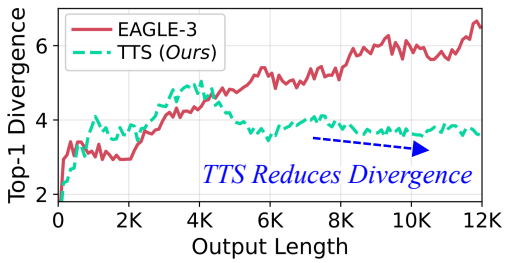


Figure 7: Cross-entropy of EAGLE-3 on Llama3.1-8B target's $Top-1$ token. At longer output lengths, EAGLE-3 diverges from the target; TTS reduces this divergence by adapting the draft in real time.

7 Related Work

Prior online methods, OSD (Liu et al. [2023]) and ATLAS (ang et al. [2025]), fine-tune the speculator using a buffer of sequences accumulated across requests and users, assuming the request stream decomposes into separable domains. In practice, however, the request stream frequently crosses domains (Zheng et al. [2023]), and a speculator tuned to past traffic falls out of alignment at every crossing, dropping acceptance length until it is re-tuned. A separate line of work retrains the speculator offline for long contexts (Yang et al. [2025b], Lee et al. [2025]), but the speculator remains frozen during deployment and matches the target only on average over the training corpus. In contrast, TTS adapts the draft within a single generation, resets it at the request boundary, and makes no assumption about the request stream. We present detailed qualitative and quantitative comparisons in Appendix E.

8 Conclusion

State-of-the-art speculators suffer from acceptance length degradation on real-world long-response tasks, with acceptance lengths decreasing to values too close to one within a few thousand tokens and yielding negligible speedup. This arises from a gap between the speculator's training and inference distributions. We propose *Test-Time Speculation (TTS)*, an online distillation approach that closes this gap by leveraging the token verification step itself as the supervision signal needed to adapt the draft at inference time, with the target model acting as the teacher, the draft as the student, and each verification round as a distillation sample. Our evaluations across ten models and three state-of-the-art speculators show that TTS improves acceptance lengths by up to 72% and 41% on average, with the benefits scaling with generation lengths.

Acknowledgments and Disclosure of Funding

This research was conducted using computing resources on the Vista GPU Cluster through the Center for Generative AI (CGAI) and the Texas Advanced Computing Center (TACC) at the University of Texas at Austin. We thank the generous support from the Cockrell School of Engineering and the Amazon AI PhD Fellowship Program through the Amazon Science Hub at UT Austin, including the AWS cloud credits that enabled the large-scale experiments conducted in this work. Poulami Das acknowledges the generous support through the AMD endowment at UT Austin.

References

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pages 117–134, 2024.

Zihao An, Huajun Bai, Ziqiong Liu, Dong Li, and Emad Barsoum. Pard: Accelerating llm inference with low-cost parallel draft model adaptation. *arXiv preprint arXiv:2504.18583*, 2025.

Junxiong ang, Shirley Wu, Zelei Shao, Vikranth Srivatsa, Jue Wang, Roy Yuan, Qingyang Wu, Alpay Ariyak, Rupert Wu, Wai Tong Chung, Chenfeng Xu, Yonatan Oren, Pragaash Ponnusamy, Yineng Zhang, Avner May, Leon Song, Tri Dao, Percy Liang, Ce Zhang, and Ben Athiwaratkun. Atlas: Adaptive-learning speculator system. Oct 2025. Together AI Blog.

Yushi Bai, Jiajie Zhang, Xin Lv, Linzhi Zheng, Siqi Zhu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longwriter: Unleashing 10,000+ word generation from long context llms. *arXiv preprint arXiv:2408.07055*, 2024.

Gaurang Bharti. Finance-alpaca: An instruction-following dataset for financial question answering. `https://huggingface.co/datasets/gbharti/finance-alpaca`, 2023. HuggingFace Dataset.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Aili Chen, Aonian Li, Baichuan Zhou, Bangwei Gong, Binyang Jiang, Boji Dan, Changhao Zhang, Changqing Yu, Chao Wang, Cheng Ma, et al. The minimax-m2 series: Mini activations unleashing max real-world intelligence. *arXiv preprint arXiv:2605.26494*, 2026a.

Guanzheng Chen, Qilong Feng, Jinjie Ni, Xin Li, and Michael Qizhe Shieh. Rapid: Long-context inference with retrieval-augmented speculative decoding. *arXiv preprint arXiv:2502.20330*, 2025.

Jian Chen, Yesheng Liang, and Zhijian Liu. Dflash: Block diffusion for flash speculative decoding. *arXiv preprint arXiv:2602.06036*, 2026b.

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant LoRA serving. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 1–13, 2024.

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

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

Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. Theoremqa: A theorem-driven question answering dataset. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7889–7901, 2023.

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

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3029–3051, 2023.

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

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3828–3850, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

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. *arXiv preprint arXiv:2403.07974*, 2024.

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.

Jaeseong Lee, Aurick Qiao, Gabriele Oliaro, Ye Wang, Samyam Rajbhandari, et al. Owl: Overcoming window length-dependence in speculative decoding for long-context inputs. *arXiv preprint arXiv:2510.07535*, 2025.

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

Shenggui Li, Chao Wang, Yikai Zhu, Yubo Wang, Fan Yin, Shuai Shi, Yefei Chen, Xiaomin Dong, Qiaoling Chen, Jin Pan, et al. Specforge: A flexible and efficient open-source training framework for speculative decoding. *arXiv preprint arXiv:2603.18567*, 2026.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *arXiv preprint arXiv:2503.01840*, 2025.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The twelfth international conference on learning representations*, 2023.

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

MathArena. Aime 2024 (i), 2024a. https://huggingface.co/datasets/MathArena/aime_2024_I.

MathArena. Aime 2024 (ii), 2024b. https://huggingface.co/datasets/MathArena/aime_2024_II.

MathArena. Aime 2025, 2025. https://huggingface.co/datasets/MathArena/aime_2025.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pages 40903–40986, 2026.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, et al. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage*, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Alexander Samarin, Sergei Krutikov, Anton Shevtsov, Sergei Skvortsov, Filipp Fisin, and Alexander Golubev. Lk losses: Direct acceptance rate optimization for speculative decoding. *arXiv preprint arXiv:2602.23881*, 2026.

ShareGPT. ShareGPT. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered, 2023.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. SLoRA: Scalable serving of thousands of LoRA adapters. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 296–311, 2024.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in neural information processing systems*, 33:3008–3021, 2020.

Bangsheng Tang, Carl Chengyan Fu, Fei Kou, Grigory Sizov, Haoci Zhang, Jason Park, Jiawen Liu, Jie You, Qirui Yang, Sachin Mehta, et al. Efficient speculative decoding for llama at scale: Challenges and solutions. *arXiv preprint arXiv:2508.08192*, 2025.

OpenThoughts-Agent Team. OpenThoughts-Agent: Data Recipes for Agentic Models. https://www.openthoughts.ai/blog/agent, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Penghui Yang, Cunxiao Du, Fengzhuo Zhang, Haonan Wang, Tianyu Pang, Chao Du, and Bo An. Longspec: Long-context lossless speculative decoding with efficient drafting and verification. *arXiv preprint arXiv:2502.17421*, 2025b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pages 3911–3921, 2018.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P Xing, et al. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998*, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Appendix

Table of Contents

A TTS Generalizes Across Speculators 16

B Benchmarks Used For Evaluation 17

C Evaluated Models And Speculators 18

D Implementation Details Of TTS 20

D.1 Navigating The Acceptance Length–Throughput Trade-Off 20

D.2 Limitations And Future Work 21

D.3 System-Level Challenges In SGLang Integration 21

E Related Work 22

E.1 Online Adaptation Methods 22

E.2 Offline Adaptation Methods 23

F Batched TTS With LoRA Adapters: Implementation And Results 25

F.1 Implementation Of Per-Request LoRA Adapters 25

F.2 Batched Serving Results 25

A TTS Generalizes Across Speculators

TTS generalizes across speculators. To demonstrate this, we evaluate the performance of TTS across several models for both EAGLE-3 and PARD. Table 6 shows that TTS consistently outperforms vanilla EAGLE-3 across all benchmarks and models, improving acceptance length by up to 66.8%.

Table 6: Acceptance length of EAGLE-3 (EGL-3) and TTS across models and tasks. $\Delta(\%)$ denotes the relative improvement of TTS over EGL-3. TTS consistently achieves higher acceptance lengths.

Dataset	Llama3.1-8B			Qwen/Qwen3-8B			Qwen/Qwen3-32B		
	EGL-3	TTS(*Ours*)	$\Delta(\%)$	EGL-3	TTS(*Ours*)	$\Delta(\%)$	EGL-3	TTS(*Ours*)	$\Delta(\%)$
AIME 2024	1.2	**2.0**	64.7	1.6	**1.9**	18.7	1.4	**1.9**	26.3
AIME 2025	1.2	**1.9**	63.7	1.6	**1.9**	20.4	1.5	**1.9**	25.5
GPQA ⋄	1.2	**2.0**	60.4	1.5	**1.9**	24.5	1.4	**1.8**	21.5
LiveCodeBench	1.4	**2.1**	47.0	1.7	**1.9**	12.7	1.5	**1.8**	22.1
MATH-500	1.2	**2.0**	61.8	1.6	**1.9**	18.8	1.5	**1.8**	21.5
OlyBench-Math	1.3	**2.1**	66.8	1.6	**1.9**	18.5	1.5	**1.9**	24.5
OlyBench-Phy	1.2	**1.9**	65.4	1.6	**1.9**	17.2	1.5	**1.8**	21.6
TheoremQA	1.2	**1.9**	56.9	1.6	**1.9**	16.8	1.5	**1.8**	21.9

Table 7 shows the performance of TTS against PARD. We observe that TTS consistently outperforms vanilla PARD across all benchmarks and models, improving acceptance length by up to 272.2%.

Table 7: Acceptance length of TTS and PARD across models and tasks. $\Delta(\%)$ denotes the relative improvement of TTS over PARD. TTS consistently achieves higher acceptance lengths.

Dataset	DeepSeek-R1-Qwen-7B			DeepSeek-R1-Qwen-14B			DeepSeek-R1-Qwen-32B		
	PARD	TTS(*Ours*)	$\Delta(\%)$	PARD	TTS(*Ours*)	$\Delta(\%)$	PARD	TTS(*Ours*)	$\Delta(\%)$
AIME 2024	3.0	**6.0**	100.0	3.2	**5.3**	65.6	3.4	**5.5**	61.8
AIME 2025	2.9	**5.9**	103.4	2.7	**4.6**	70.4	2.9	**4.8**	65.5
GPQA ⋄	2.4	**5.6**	133.3	3.2	**5.4**	68.8	2.8	**6.1**	117.9
LiveCodeBench	1.8	**6.7**	272.2	2.0	**6.1**	205.0	1.9	**6.1**	221.1
MATH-500	4.2	**5.7**	35.7	3.9	**5.0**	28.2	4.0	**5.1**	27.5
OlyBench-Math	2.0	**6.0**	200.0	2.6	**5.2**	100.0	2.3	**5.5**	139.1
OlyBench-Phy	2.0	**6.1**	205.0	2.2	**5.2**	136.4	2.3	**5.7**	147.8
TheoremQA	3.8	**5.4**	42.1	3.9	**5.0**	28.2	4.0	**5.1**	27.5

B Benchmarks Used For Evaluation

All benchmarks used to evaluate TTS elicit long outputs, often generating tens of thousands of tokens before reaching the final answer, either within a single response or across a multi-turn session. These benchmarks collectively represent long-generation tasks that dominate real-world workloads. For each benchmark, we randomly select 32 sequences (except AIME, which comprises only 30 sequences) using a fixed seed for deterministic comparison. We consolidate the identifiers for each benchmark in Table 8 and describe them in detail below:

Table 8: HuggingFace identifiers for benchmarks used in our evaluations, grouped by domain.

Benchmark	**HuggingFace Dataset Identifier**
Competition mathematics	
AIME 2024	`MathArena/aime_2024_I, MathArena/aime_2024_II`
AIME 2025	`MathArena/aime_2025`
MATH-500	`HuggingFaceH4/MATH-500 (Level 5)`
OlympiadBench-Math	`Hothan/OlympiadBench` (`OE_TO_maths_en_COMP`)
Physics	
OlympiadBench-Physics	`Hothan/OlympiadBench` (`OE_TO_physics_en_COMP`)
Graduate-level scientific reasoning	
GPQA ⋄	`Idavidrein/gpqa` (`gpqa_diamond`)
University-level theorem application	
TheoremQA	`TIGER-Lab/TheoremQA`
Code generation	
LiveCodeBench v6	`livecodebench/code_generation_lite (Hard)`
Multi-turn agentic tasks	
OpenThoughts-Agent	`open-thoughts/OpenThoughts-Agent-SFT-100K`

AIME: The AIME dataset collects the American Invitational Mathematics Examination problems from each year. Both the AIME 2024 (MathArena [2024a,b]) and the AIME 2025 (MathArena [2025]) comprise 30 sequences with a final integer answer in $[0, 999]$. The AIME dataset is a strong test for olympiad-style reasoning as it elicits long outputs of up to $32K$ tokens.

MATH-500: The MATH-500 dataset is a curated 500-problem subset of the MATH benchmark (Hendrycks et al. [2021]), introduced in OpenAI's PRM800K (Lightman et al. [2023]) for evaluating mathematical-reasoning models. Problems are drawn from several competitions such as AMC, and span seven subject areas with difficulty levels 1 through 5. We use only the Level 5 problems, the most challenging subset.

OlympiadBench-Math: OlympiadBench (He et al. [2024]) is a bilingual olympiad-style benchmark drawn from international and Chinese olympiad competitions. We use English-language problems from this dataset in our evaluations. Furthermore, we restrict evaluation to text-only sequences as not all evaluation models are multi-modal. Each sequence is accompanied by a symbolic or numeric final answer. Similar to AIME and MATH-500, OlympiadBench-Math elicits long generation lengths.

OlympiadBench-Physics: OlympiadBench-Physics (He et al. [2024]) is the physics counterpart of OlympiadBench-Math, drawn from the same olympiad-competition pipeline and similarly restricted to its English-language subset. Each problem requires deriving a closed-form or numeric answer from first principles, often achieved via multi-step modeling and symbolic manipulation.

GPQA ⋄: GPQA ⋄ (Rein et al. [2023]) is the hardest 198-question subset of the GPQA benchmark. It comprises graduate-level multiple-choice questions in biology, chemistry, and physics. PhD-level domain experts answer GPQA ⋄ correctly only $\sim 65\%$ of the time, while non-experts achieve $\leq 35\%$ accuracy even with unrestricted web access, making it a strong test of expert-level scientific reasoning.

TheoremQA: TheoremQA (Chen et al. [2023]) is a STEM benchmark consisting of sequences that require identifying and applying a specific named theorem from mathematics, physics, or engineering to produce a numeric or symbolic final answer. TheoremQA is a strong test of theorem application, and elicits long reasoning traces as the model recalls, applies, and verifies the relevant result.

LiveCodeBench: LiveCodeBench (Jain et al. [2024]) is a competitive-programming benchmark drawn from LeetCode, AtCoder, and CodeForces. Each sequence includes the problem statement, the starter code, and an explicit instruction to reason step by step before emitting a solution.

OpenThoughts-Agent: OpenThoughts-Agent-SFT-100K (Team [2026]) is a collection of $100K$ agentic tasks paired with reference trajectories, sourced from SWE-Smith, StackExchange, and GitHub issue resolution. We use only the task specifications and discard the reference trajectories, since TTS adapts the speculator to the trajectory produced by the target model itself. Each task is executed in a sandboxed shell environment under the Terminus-2 agent protocol (Merrill et al. [2026]), where the model interleaves tool calls with environment observations until the task is complete. Unlike the single-turn benchmarks above, generation here is distributed across an entire session spanning many turns, which collectively elicits tens of thousands of output tokens.

C Evaluated Models And Speculators

We use ten state-of-the-art models across the Qwen-3 (Yang et al. [2025a]), Qwen-3.5 (Qwen Team [2026]), DeepSeek-R1-Distill-Qwen (Guo et al. [2025]), GPT-OSS (OpenAI [2025]), MiniMax (Chen et al. [2026a]), and Llama3.1 (Grattafiori et al. [2024]) families. Our selection of models encompasses parameter sizes from 4 billion to 229 billion. The Qwen3 models (4B, 8B, 32B), Llama3.1-8B-Instruct, and the DeepSeek-R1-Distill-Qwen models (7B, 14B, 32B) are dense transformer models. GPT-OSS-120B, Qwen3.5-122B-A10B, and MiniMax-M2.5 are mixture-of-experts models (Shazeer et al. [2017]). The first two interleave full-attention layers with cheaper alternatives, namely sliding-window attention in GPT-OSS-120B and linear attention in Qwen3.5-122B-A10B, whereas MiniMax-M2.5 uses full attention in every layer. Table 9 summarizes the specifications of each model.

Table 9: Our selection of models spans parameter sizes from 4B to 229B and encompasses dense, MoE, and hybrid MoE architectures across six model families.

HuggingFace Identifier	Architecture	Active / Total	Layers	Context Length
Qwen/Qwen3-4B	Dense	4B / 4B	36	40K
Qwen/Qwen3-8B	Dense	8B / 8B	36	40K
Qwen/Qwen3-32B	Dense	32B / 32B	64	40K
deepseek-ai/DeepSeek-R1-Distill-Qwen-7B	Dense	7B / 7B	28	128K
deepseek-ai/DeepSeek-R1-Distill-Qwen-14B	Dense	14B / 14B	48	128K
deepseek-ai/DeepSeek-R1-Distill-Qwen-32B	Dense	32B / 32B	64	128K
meta-llama/Llama-3.1-8B-Instruct	Dense	8B / 8B	32	128K
openai/gpt-oss-120b	MoE	5.1B / 117B	36	128K
Qwen/Qwen3.5-122B-A10B	Hybrid + MoE	10B / 122B	48	256K
MiniMaxAI/MiniMax-M2.5	MoE	10B / 229B	62	192K

For each model in Table 9, we evaluate TTS on top of the matching publicly released DFlash (Chen et al. [2026b]), EAGLE-3 (Li et al. [2025]), and PARD (An et al. [2025]) checkpoints. We describe each speculator below in detail and summarize their HuggingFace identifiers in Table 10.

DFlash: DFlash is a *block-diffusion* speculator that generates a block of B tokens in a single forward pass. It first extracts the intermediate hidden states of a selected subset of target model layers and then passes them through the speculator's W_k and W_v projection matrices to produce the context Key-Value (KV) cache. Next, this context KV cache is concatenated with the speculator's own KV vectors for the block of tokens being generated. Therefore, the query vectors attend to the target context and the block context in a single forward pass.

EAGLE-3: EAGLE-3 is an *auto-regressive* speculator that generates one token at a time. It first extracts the hidden states from three layers (typically, an early, a middle, and a late layer) of the target model. These hidden states are then concatenated and down-projected through a fully connected layer to produce a k-dimensional context feature, where k represents the hidden state dimension of the target model. This context feature is then concatenated with the embedding vector of the previously generated output token, reduced through a second fully connected layer and fed into the EAGLE-3

draft, a single transformer layer. Its output is then passed to the LM head of the target model to produce the next-token distribution.

PARD: PARD is a *parallel* speculator that generates a block of K tokens in a single forward pass. It appends K mask tokens to the current sequence, one for each draft token to be generated. The speculator then predicts the token at every masked position simultaneously, where each prediction conditions on the committed prefix and the preceding mask tokens, allowing all K tokens to be generated in parallel. Unlike EAGLE-3 and DFlash, the PARD speculator is a *standalone* model that does not reuse the target model's hidden states. It adapts an existing auto-regressive model offline to predict the masked tokens in parallel through fine-tuning.

We use a block size of 16 and 8 for DFlash and PARD, respectively, and a generation depth of 8 for EAGLE-3, across all target models, consistent with the original papers (Chen et al. [2026b], An et al. [2025], Li et al. [2025]). We use the publicly released speculator checkpoints from the original authors, except for EAGLE-3 on the Qwen3 and GPT-OSS target models, for which the authors have not released checkpoints; there, we use the AngelSlim and NVIDIA releases from HuggingFace, respectively. Table 10 summarizes the speculator checkpoints used in our evaluations.

Table 10: HuggingFace identifiers for each speculator and target model pair.

Target Model	Speculator	HuggingFace Identifier
meta-llama/Llama-3.1-8B-Instruct	EAGLE-3	yuhuili/EAGLE3-LLaMA3.1-Instruct-8B
Qwen/Qwen3-8B		AngelSlim/Qwen3-8B_eagle3
Qwen/Qwen3-32B		AngelSlim/Qwen3-32B_eagle3
openai/gpt-oss-120b		nvidia/gpt-oss-120b-Eagle3-v3
openai/gpt-oss-120b		nvidia/gpt-oss-120b-Eagle3-long-context
Qwen/Qwen3-4B	DFlash	z-lab/Qwen3-4B-DFlash-b16
Qwen/Qwen3-8B		z-lab/Qwen3-8B-DFlash-b16
Qwen/Qwen3.5-122B-A10B		z-lab/Qwen3.5-122B-A10B-DFlash
MiniMaxAI/MiniMax-M2.5		z-lab/MiniMax-M2.5-DFlash
DeepSeek-R1-Distill-Qwen-7B*	PARD	amd/PARD-DeepSeek-R1-Distill-Qwen-1.5B
DeepSeek-R1-Distill-Qwen-14B*		
DeepSeek-R1-Distill-Qwen-32B*		

* Hosted under the `deepseek-ai/` organization on HuggingFace.

D Implementation Details Of TTS

In this section, we provide the implementation details of TTS, including scheduling and system-level optimizations introduced in Section 6.5. We then describe the system-level challenges we encountered in integrating TTS into the SGLang inference framework.

D.1 Navigating The Acceptance Length–Throughput Trade-Off

Every gradient update in TTS invokes a backward pass and an optimizer (Adam) step on the speculator. While this improves acceptance length by continuously adapting the speculator to match the target distribution, it introduces additional overhead on the critical path in every speculate-verify round. The latency overhead of a single backward pass and optimizer step is comparable to a few speculation rounds. Thus, applying TTS naively can negate the throughput improvements from higher acceptance length. We navigate this trade-off via scheduling and system-level optimizations as described below:

Strided Updates: By default, TTS performs a gradient update after every speculate-verify round. A natural alternative is to perform the gradient update only once every $\mathcal{S}$ rounds, where $\mathcal{S}$ is the *update stride*. Each update consumes the supervision signal only from the most recent round, without any accumulation across the $\mathcal{S}-1$ skipped rounds. However, fewer updates imply weaker adaptation to the ongoing generation and therefore reduce the improvements in acceptance length. For example, Table 11 shows that the improvements in acceptance length are the highest for $\mathcal{S}=1$ and monotonically decrease as the update stride is increased.

Table 11: Acceptance length of TTS relative to DFlash on Qwen3-8B with the update stride $\mathcal{S}$. The default $\mathcal{S}=1$ achieves the largest improvement across benchmarks. Higher values of $\mathcal{S}$ reduce the improvement in acceptance length as the speculator is adapted less frequently.

Benchmark	DFlash (*Baseline*)	TTS (*Ours*) / $\Delta(\%)$				
		$\mathcal{S}=1$*(default)*	$\mathcal{S}=5$	$\mathcal{S}=15$	$\mathcal{S}=30$	$\mathcal{S}=50$
AIME 2024	3.3	**4.6** / 38.5	4.3 / 30.7	3.9 / 19.0	3.7 / 12.6	3.6 / 10.2
AIME 2025	3.1	**4.1** / 33.1	3.9 / 27.0	3.7 / 18.1	3.5 / 12.1	3.3 / 8.0
GPQA ◇	3.2	**4.7** / 47.3	4.3 / 34.3	4.0 / 24.1	3.8 / 17.8	3.7 / 14.7
LiveCodeBench	2.9	**4.5** / 55.2	4.2 / 44.9	3.9 / 32.9	3.6 / 24.4	3.5 / 19.8
MATH-500	3.8	**4.3** / 13.3	4.1 / 8.3	4.0 / 5.4	3.9 / 3.7	3.9 / 2.3
OlyBench-Math	2.7	**4.1** / 50.3	3.8 / 42.1	3.5 / 28.9	3.2 / 20.0	3.1 / 14.2
OlyBench-Phy	2.8	**4.3** / 53.1	3.9 / 39.2	3.5 / 26.2	3.3 / 18.2	3.2 / 13.3
TheoremQA	3.5	**4.3** / 22.8	4.0 / 15.2	3.8 / 9.0	3.7 / 6.3	3.7 / 4.8

However, increasing the stride $\mathcal{S}$ amortizes the update overhead, since the gradient step is now performed every $\mathcal{S}$ rounds rather than at every round. For example, Table 12 shows that the best wall-clock speedup ($\mathbf{1.83}\times$) is achieved at $\mathcal{S}=10$. Beyond $\mathcal{S}=10$, speedup decreases because the degradation of acceptance length outpaces the savings from amortizing the gradient update.

Table 12: Relative acceptance length (AL) and wall-clock speedup of TTS on LiveCodeBench with Qwen3-8B. Both metrics are reported relative to $\mathcal{S}=1$. The sweet spot is at $\mathcal{S}=10$ ($\mathbf{1.83}\times$ speedup) where the update overhead is amortized while acceptance length is preserved within $0.88\times$ of $\mathcal{S}=1$.

Metric	Update Stride ($\mathcal{S}$)							
	1*(default)*	5	10	15	20	30	40	50
Relative AL	1.00×	0.93×	0.88×	0.85×	0.84×	0.80×	0.79×	0.77×
Relative Speedup	1.00×	1.77×	**1.83**×	1.79×	1.80×	1.65×	1.64×	1.56×

Although strided updates reduce the frequency of the gradient steps, each update still lies on the critical path of the round where it is performed. For small target models, the cost of one gradient update to the speculator is comparable to a few speculation rounds and thus can degrade wall-clock speedup. We further hide this latency through *asynchronous pipelining*, described next.

Asynchronous Pipelining: If the gradient update could be performed in parallel with the subsequent speculation rounds, we could hide its latency (largely) from the critical path. Building on this insight, we design mechanisms in TTS to offload the gradient update to a dedicated CUDA stream (the *side stream*) that executes concurrently with the next $\mathcal{S}-1$ speculation rounds on the main stream. However, asynchronous pipelining does not fully eliminate the latency overhead of the gradient update. This is because the side stream and the main stream contend for the same GPU resources, slightly slowing the speculation rounds that run concurrently with a gradient update. Nonetheless, the additional overhead is much smaller than executing the gradient update on the critical path. A synchronization barrier is inserted after $\mathcal{S}-1$ rounds to ensure that the speculator is adjusted with the prior update before the next canvas of draft tokens is generated.

Figure 8 illustrates the resulting execution timeline of TTS with both these optimizations applied.

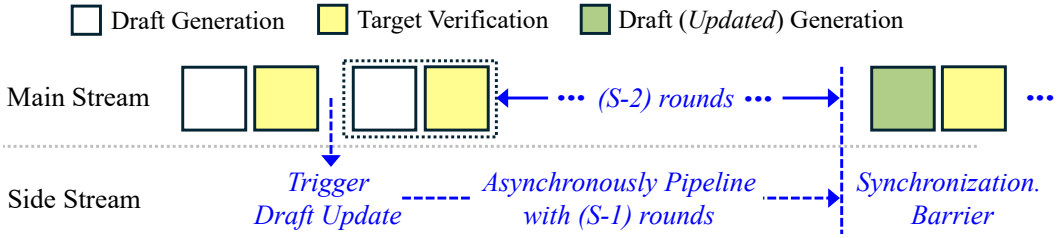


Figure 8: Execution timeline of TTS with strided updates and asynchronous pipelining. Every $\mathcal{S}$ speculation rounds, a gradient update is dispatched to a side CUDA stream and overlaps with the next $\mathcal{S}-1$ rounds on the main stream, keeping the update off the critical path.

D.2 Limitations And Future Work

While TTS delivers substantial speedup, the achievable throughput remains limited by two missing system-level optimizations. *First*, the gradient update to adapt the speculator is currently implemented as a sequence of standard PyTorch kernels. Fusing these into a custom kernel would reduce the latency of each update and improve throughput across all strides. *Second*, the update stride is currently fixed and applied statically across all speculation rounds. We could, however, adapt the stride online based on the observed acceptance length, automatically choosing a lower stride when the acceptance length degrades and a higher stride when it remains stable. Both these optimizations are orthogonal to the main contributions of this paper, and therefore we defer them to future work.

D.3 System-Level Challenges In SGLang Integration

SGLang is a framework designed for fast inference rather than online adaptation. We describe below the two primary challenges of integrating TTS into SGLang, along with their resolutions:

1. **Inference Kernels Do Not Support Gradient Computation:** To maximize throughput, SGLang uses fused CUDA/Triton kernels for attention, normalization, and activation. These kernels are not designed for gradient computation. During the backward pass, `autograd` breaks silently. As a result, the gradient update has no effect on the speculator weights. We resolve this by switching to the differentiable PyTorch implementation for the speculator in the update round, ensuring that the gradients flow correctly through the backward pass. On non-update rounds, we retain the fused kernels to preserve throughput.
2. **CUDA-Graph Replay Does Not Honor Cross-Stream Synchronization:** To improve performance, each draft generation round in SGLang runs inside a captured CUDA graph, a pre-recorded sequence of GPU operations replayed for efficiency. Graph replay runs on its own internal stream and does not honor GPU synchronization barriers. Thus, the update made by TTS can be silently skipped before the next round begins. We resolve this by introducing a CPU-side synchronization barrier that blocks the GPU until the side stream completes the update before the next graph is replayed. This barrier is inserted only at update boundaries, leaving the CUDA graph performance unaffected on non-update rounds.

These optimizations and resolutions enable TTS to run end-to-end on SGLang.

E Related Work

E.1 Online Adaptation Methods

In this section, we compare and contrast TTS with prior online methods that adapt the speculator after deployment. These works adapt the speculator across requests and users, and assume that production traffic decomposes into cleanly separable domains.

Online speculative decoding (OSD) (Liu et al. [2023]) periodically fine-tunes the draft model to match the target distribution on a buffer of accumulated sequences *across users and requests*. The draft is updated when the buffer exceeds a certain size. OSD additionally orchestrates multiple speculators, each specialized to a particular domain, and uses a router to direct each incoming request to the best-matching speculator. Each draft is only fine-tuned on the requests it receives. However, this multi-draft *setup diverges* from how state-of-the-art speculators are typically deployed in practice (Tang et al. [2025]), where a single speculator (such as EAGLE-3 or DFlash) is paired with the target model and serves all incoming queries.

ATLAS (ang et al. [2025]) extends this paradigm to a production serving system. Rather than routing between domain-specialized speculators, ATLAS pairs a heavyweight static speculator (Together AI's proprietary Turbo speculator) with a lightweight adaptive speculator that is updated on production traffic. A confidence-aware controller chooses which to trust at each step. However, the lightweight speculator is still adapted across requests over accumulated sequences. Consequently, its adaptation reflects the aggregate of recent traffic rather than the request currently being served.

This cross-request adaptation, however, relies on the *assumption that production traffic cleanly decomposes into separable domains*. The evaluation of OSD structures the request stream as a concatenation of single-domain benchmarks. Specifically, OSD evaluates on streams of $2K$ sequences each from Spider (SQL) (Yu et al. [2018]), GSM8K (math) (Cobbe et al. [2021]), Code-search-Python (code) (Husain et al. [2019]), and Alpaca-finance (finance) (Bharti [2023]), concatenated sequentially before the stream switches to the next benchmark. Similarly, ATLAS adapts its lightweight speculator on accumulated live traffic, aligning it to the aggregate distribution of recent requests rather than the current request being served. In practice, however, request streams cross domain boundaries frequently (Zheng et al. [2023], ShareGPT [2023]), and a speculator tuned to the aggregate of recent traffic falls out of alignment at every crossing. Furthermore, *neither method demonstrates compatibility* with public state-of-the-art speculators such as EAGLE-3, DFlash, and PARD; OSD relies on an auxiliary draft model, while ATLAS utilizes Together AI's proprietary Turbo speculator.

Fundamentally, prior works adapt the speculator at the granularity of traffic accumulated across many requests, whereas acceptance length is determined by how closely the draft distribution ($q(\cdot \mid x_{1:t})$) matches the target ($p(\cdot \mid x_{1:t})$) for the current prefix ($x_{1:t}$). Adapting to the aggregate of recent traffic improves the speculator only when the current request resembles the past request stream, and cannot specialize to the request being served. In contrast, TTS adapts the speculator within each generation at the granularity of each speculation round (or every $\mathcal{S}$ rounds with strided updates). It distills the target's per-prefix conditional $p(\cdot \mid x_{1:t})$ back into the speculator at every update and resets it between requests, so per-request adaptation is bounded. It also operates directly on top of public state-of-the-art speculators. Moreover, TTS makes no assumption about the structure of the request stream and consistently improves acceptance length across benchmarks. Furthermore, to improve applicability, we implement TTS in the production-grade SGLang inference serving engine (Zheng et al. [2024]). Table 13 summarizes the key differences between TTS and prior work.

Table 13: Comparison of online speculator adaptation methods.

Comparison Property	**OSD**	**ATLAS**	**TTS**
Adapts within a single generation	✗	✗	✓
No assumption on the request stream structure	✗	✗	✓
Prevents cross-request leakage	✗	✗	✓
Compatible with state-of-the-art speculators (EAGLE-3, DFlash)	✗	✗	✓
Single-speculator deployment (no specialized routing)	✗	✓	✓
No proprietary dependencies	✓	✗	✓
Production-grade serving integration	✗	✓	✓

E.1.1 Quantitative Comparison Against Online Adaptation Methods

To demonstrate the efficacy of the per-request adaptation in TTS against the cross-request adaptation used by prior works, we implement and integrate OSD in SGLang and compare its performance against TTS on the DFlash speculator.

Following the official public codebase[2], we perform one gradient update on the speculator at an interval of eight requests (I=8), minimizing the soft cross-entropy between the speculator's and target's next-token distributions at the positions where the target rejected the speculator's proposed tokens. For this comparison, both OSD and TTS train the entire speculator and use an identical learning rate of `1e-4`. Thus, both methods only differ in the granularity and persistence of adaptation. We evaluate both approaches on the Qwen3.5-122B-A10B target and its DFlash speculator (`z-lab/Qwen3.5-122B-A10B-DFlash`), serving an identical request stream of four benchmarks: LiveCodeBench, MATH-500, OlympiadBench-Math (zh), and OlympiadBench-Physics (zh), with requests issued in short bursts of two per domain.

Table 14 shows that TTS achieves higher acceptance length than OSD across all benchmarks. As OSD *persists* the speculator state across requests, the measured acceptance length is strongly influenced by the preceding requests. Furthermore, as adaptation in OSD is *unbounded*, the speculator's weights drift further from their pretrained values with each gradient update. In contrast, TTS adapts within each request and resets the speculator at the request boundary, so its adaptation always reflects the current request and cannot degrade any subsequent one.

Table 14: Mean acceptance length (AL) on an interleaved request stream for the Qwen3.5-122B-A10B model and its DFlash speculator. While OSD adapts across requests and cannot specialize to the request being served, TTS resets at the request boundary and consistently achieves higher AL.

Benchmark	Acceptance Length		
	DFlash	OSD	TTS (*Ours*)
LiveCodeBench	2.06	2.50 (+21.4%)	**3.57** (+73.3%)
MATH-500	4.66	5.38 (+15.5%)	**5.49** (+17.8%)
OlyBench-Math (zh)	3.11	3.84 (+23.5%)	**4.72** (+51.8%)
OlyBench-Physics (zh)	3.30	3.78 (+14.5%)	**4.25** (+28.8%)
Mean	3.28	3.88 (+18.1%)	**4.51** (+37.3%)

E.2 Offline Adaptation Methods

In this section, we compare and contrast TTS with prior offline methods that improve long-context speculation before deployment. These works redesign and retrain the speculator offline on long-context data to reduce the training-inference mismatch described in Section 3.

LongSpec (Yang et al. [2025b]) addresses the training-inference data mismatch by proposing and training a speculator on long-context data. The position encoding for their proposed speculator remains consistent with the target model beyond the training context, allowing it to sustain high acceptance lengths. OWL (Lee et al. [2025]) similarly trains a recurrent speculator for long contexts, which only conditions on the hidden state of the target model's last token, decoupling the achieved acceptance length from the prefix length. While both these methods improve acceptance length, the resulting speculator remains frozen during deployment, and thus matches the target model only on average over its training corpus.

RAPID (Chen et al. [2025]) takes a different approach: instead of training a speculator from scratch, it directly adjusts the target's distribution. RAPID runs the speculator over a retrieved context, instead of the entire input, and shifts the target's distribution toward the speculator's during verification. However, this approach alters the target's output distribution and is therefore *lossy*. This is in contrast to TTS and all methods discussed above, which leave the target's output distribution unchanged, and is thus orthogonal.

[2] `https://github.com/LiuXiaoxuanPKU/OSD`

E.2.1 Quantitative Comparison Against Offline Adaptation Methods

As TTS adapts the speculator online and preserves the target model's distribution, it is orthogonal to the prior offline methods and can in fact be applied on top of them. To showcase this, we apply TTS to the released LongSpec speculators[3] on two long-context target models. The exact configuration and model pairs used for this study are listed in Table 15. As LongSpec already reduces the training-inference mismatches described in Section 3, any further improvement can be attributed to online adaptation alone, regardless of how extensively the speculator is trained offline. Table 16 shows that TTS improves the acceptance length of the LongSpec speculator by up to 13.9% and 10.1% on the two targets, and improves it consistently across all eight benchmarks. This reiterates our earlier finding in Section 6.4 that although extensive offline training reduces the distribution mismatch between the speculator and the target, it does not eliminate the need to specialize the speculator to the request being served.

Table 15: Target/speculator pairs and TTS configuration used for the LongSpec experiments (Table 16). Both speculator checkpoints are officially released by LongSpec and are used as-is, without any retraining or fine-tuning. All hyperparameters are identical across the two configurations.

Property	**longchat-7b-v1.5-32k**	**Llama-3-8B-Instruct-262k**
Target model	lmsys/longchat-7b-v1.5-32k	gradientai/Llama-3-8B-Instruct-262k
Speculator	sail/longspec-longchat-7b-v1.5-32k	sail/longspec-Llama-3-8B-Instruct-262k
Draft tree shape	$[4, 16, 16, 16, 16]$ (5 draft levels + 1 bonus token, AL ≤ 6)	
Sampling	Greedy (temperature 0)	
Max output length	32K tokens	
TTS variant	Entire speculator adapted	
Update stride $\mathcal{S}$	10	

Table 16: Acceptance length of the released LongSpec speculators with and without TTS. Although LongSpec reduces the training-inference data mismatches, TTS recovers further acceptance length across all benchmarks by specializing the speculator to the request being served.

Benchmark	longchat-7b-v1.5-32k			Llama-3-8B-Instruct-262k		
	LongSpec	LongSpec+ TTS	$\Delta(\%)$	LongSpec	LongSpec+ TTS	$\Delta(\%)$
AIME 2024	4.7	**4.9**	+5.6	4.8	**5.0**	+5.6
AIME 2025	4.4	**5.0**	+13.9	4.4	**4.9**	+10.1
MATH-500	4.6	**4.9**	+6.8	4.3	**4.5**	+5.4
OlyBench-Math	4.2	**4.8**	+12.9	4.6	**4.9**	+7.9
OlyBench-Phy	4.4	**4.9**	+12.1	4.6	**4.8**	+3.7
GPQA ⋄	4.2	**4.7**	+12.3	4.5	**4.8**	+6.6
LiveCodeBench	4.4	**4.7**	+6.7	4.4	**4.5**	+4.2
TheoremQA	4.2	**4.5**	+5.5	4.5	**4.7**	+2.9

[3] https://github.com/sail-sg/LongSpec

F Batched TTS With LoRA Adapters: Implementation And Results

In this section, we provide the implementation details of batched TTS with per-request LoRA adapters introduced in Section 6.6, and report batched serving results on more benchmarks and speculators.

F.1 Implementation Of Per-Request LoRA Adapters

Per-Request Adapter Pools: TTS maintains the LoRA adapters for all concurrent requests in pre-allocated pools, with one slot per request indexed by the engine's request identifier. When a request is admitted, its slot is initialized with the adapter's B matrix set to zero. Thus, the adapter begins speculation from the unmodified pretrained speculator checkpoint. Upon request completion, the slot, along with its optimizer state, is released and assigned to the next request.

Serving With Shared Speculator Weights: During speculation, the adapter delta for each adapted projection is computed as two batched matrix multiplications. Tokens of each request are routed only through their own adaptation slot, thereby ensuring that the adaptation state of one request is *invisible* to every other request in the batch. This eliminates cross-request corruption (Section 6.6) by construction. The speculator's pretrained weights are read-only and shared across the batch, mirroring the execution model that production engines already employ for serving distinct LoRA adapters over a shared base model (Chen et al. [2024], Sheng et al. [2024]).

Per-Request Gradient Updates: At each update, TTS adapts every request in the batch independently: each request's distillation loss is computed over its own generated tokens and used as a training signal for only its own LoRA adapters. To avoid launching one backward pass per request, the per-request losses are summed and differentiated in a single batched backward pass. As no loss term touches another request's adapter, this is purely an execution optimization, and each adapter receives exactly the gradient it would have otherwise received if served in isolation. Similarly, each request carries its own optimizer state (Adam moments and step count). The strided updates and asynchronous pipelining of Appendix D carry over unchanged.

Adapted Projections: The set of adapted projections determines both the per-request memory footprint and the computational cost of the gradient update. Therefore, we want to select the smallest set that preserves the improvements in acceptance length. In our ablations, expanding the adapted set beyond this minimal set improves acceptance length only marginally while inflating the adaptation state and slowing every update. For DFlash, we adapt the attention query and output projections along with the MLP projections. All studies use rank $r{=}16$ adapters with a learning rate of `1e-4`.

F.2 Batched Serving Results

We now evaluate TTS under batched serving. Table 17 extends the Qwen3.5-122B-A10B results of Table 5 to the remaining four benchmarks. As the baseline acceptance lengths are lower on this large target ($1.7{-}3.1$), TTS has more headroom and improves throughput by up to $2.4\times$.

Table 17: Throughput (Tokens/Sec) of TTS versus DFlash (z-lab/Qwen3.5-122B-A10B-DFlash) on Qwen3.5-122B with $\mathcal{S}{=}10$ for the remaining benchmarks. TTS improves throughput by up to $2.4\times$.

Benchmark	Approach	$BS{=}8$	$BS{=}16$	$BS{=}32$	$BS{=}64$
MATH-500	DFlash	280.3	319.9	435.8	376.6
	TTS (*Ours*)	**445.2** ($1.59\times$)	**602.3** ($1.88\times$)	**877.5** ($2.01\times$)	**666.9** ($1.77\times$)
TheoremQA	DFlash	233.8	289.1	439.9	313.0
	TTS (*Ours*)	**511.9** ($2.19\times$)	**661.3** ($2.29\times$)	**987.9** ($2.25\times$)	**663.5** ($2.12\times$)
OlyBench-Math	DFlash	193.9	267.7	392.9	196.8
	TTS (*Ours*)	**444.8** ($2.29\times$)	**575.8** ($2.15\times$)	**908.4** ($2.31\times$)	**466.9** ($2.37\times$)
OlyBench-Phy	DFlash	197.6	261.3	387.7	249.4
	TTS (*Ours*)	**401.4** ($2.03\times$)	**503.3** ($1.93\times$)	**779.3** ($2.01\times$)	**467.2** ($1.87\times$)

We also evaluate on the smaller Qwen3.6-27B target. Table 18 shows that TTS improves throughput by $1.16{-}1.73\times$ across benchmarks and batch sizes.

Table 18: Throughput (Tokens/Sec) of TTS relative to DFlash (z-lab/Qwen3.6-27B-DFlash) on Qwen3.6-27B with $\mathcal{S}=24$. TTS consistently improves throughput by $1.16-1.73\times$ across benchmarks.

Benchmark	Approach	$BS=8$	$BS=16$	$BS=32$	$BS=64$
AIME 2024	DFlash	371.7	505.2	671.2	807.8
	TTS (*Ours*)	**480.4** (1.29×)	**676.3** (1.34×)	**879.9** (1.31×)	**1042.8** (1.29×)
AIME 2025	DFlash	366.1	503.3	689.6	827.7
	TTS (*Ours*)	**491.0** (1.34×)	**672.0** (1.34×)	**910.4** (1.32×)	**1066.2** (1.29×)
GPQA ⋄	DFlash	323.2	430.2	613.2	693.4
	TTS (*Ours*)	**428.7** (1.33×)	**607.5** (1.41×)	**790.8** (1.29×)	**844.8** (1.22×)
LiveCodeBench	DFlash	329.9	447.2	622.3	700.3
	TTS (*Ours*)	**450.0** (1.36×)	**607.2** (1.36×)	**800.6** (1.29×)	**881.8** (1.26×)
MATH-500	DFlash	365.8	520.8	650.9	757.2
	TTS (*Ours*)	**487.8** (1.33×)	**605.4** (1.16×)	**853.4** (1.31×)	**916.4** (1.21×)
TheoremQA	DFlash	319.9	427.8	626.9	828.2
	TTS (*Ours*)	**553.3** (1.73×)	**614.8** (1.44×)	**862.2** (1.38×)	**1010.3** (1.22×)
OlyBench-Math	DFlash	358.0	515.7	671.9	717.1
	TTS (*Ours*)	**500.5** (1.40×)	**686.0** (1.33×)	**922.4** (1.37×)	**978.4** (1.36×)
OlyBench-Phy	DFlash	345.3	464.1	643.1	711.8
	TTS (*Ours*)	**481.2** (1.39×)	**651.8** (1.40×)	**866.7** (1.35×)	**939.9** (1.32×)

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and the introduction make three main claims: (i) TTS adapts the speculator at inference time, (ii) It integrates with state-of-the-art speculators such as DFlash and EAGLE-3, (iii) It consistently improves acceptance length across models and families. Each claim is supported by experimental results in Section 6.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We describe current limitations of our work in Appendix D.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not include formal theoretical results. The lossless property of speculative decoding is established by prior work (Leviathan et al. [2023]); TTS preserves this property by leaving the target model untouched.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The paper provides a full description of the algorithm in Section 4. The evaluation setup, models, speculators, and benchmarks are detailed in Section 5 and Appendices C and B respectively.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Our code is released on the project page (`https://avinkumarut.github.io/TTS/`), together with instructions to reproduce the main experimental results. All evaluated models and benchmarks are publicly available and fully documented in Appendix C and B.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The paper provides all experimental setup details in Section 5 and Appendix B, C and D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: All reported performance metrics (acceptance length and throughput) are means over each benchmark. Decoding is deterministic ($T{=}0$), thus re-running the same experiment produces identical results.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: All experiments ran on a single Amazon EC2 G7e node with 8×NVIDIA RTX PRO 6000 Blackwell GPUs (96 GB HBM), as described in section 5. The model configuration and benchmarks used in our evaluations are listed in Appendix C and B respectively.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS code of ethics, and the research presented in this paper conforms to it in every respect.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [N/A].

Justification: TTS preserves the target model's output distribution exactly (Section 4) and introduces no new generative capabilities. Thus, the societal impacts reduce to those of the underlying target models, which are inherited from existing public releases (Appendix C).

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A].

Justification: This paper introduces an inference-time acceleration technique and does not release new pre-trained models, datasets, or any artifacts with a high risk of misuse. All target models and benchmarks used are existing public releases and inherit the safeguards of the original providers.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All existing assets used in this paper including models, benchmarks and the inference framework are properly cited in the main text and the appendix and used in accordance with their license. HuggingFace identifiers are listed in Appendix B and Appendix C.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes].

Justification: We release our code on the project page (`https://avinkumarut.github.io/TTS/`), documented with a README, reproduction instructions, and a license. We do not introduce any new datasets or pre-trained models; all evaluated models, speculators, and benchmarks are existing public assets. Implementation details are also described in Section 4 and Appendix B, C and D.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A].

Justification: This paper does not conduct any crowd-sourcing studies or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.

- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A].

    Justification: Our work does not conduct any research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A].

    Justification: We do not use an LLM to develop our core methodology or research. LLMs appear in this work as a subject of study, rather than a tool for conducting research.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.