ADASPEC: SPECULATIVE DECODING WITH IN-ENGINE CONTINUAL LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Speculative decoding accelerates large language model inference with a small drafter. However, the drafter is frozen after offline training, while it serves narrow and shifting traffic on which its predictions degrade. Recent work trains the drafter online outside the inference engine, which costs extra GPUs and data movement and serves with stale weights. We introduce **AdaSpec**, an inference engine in which the drafter learns continually from the traffic it serves, on the same GPUs and from the same forward passes. AdaSpec makes learning a by-product of serving: it reuses the activations and verification results already on the GPU and overlaps the backward pass with decoding. The drafter learns to maximize the expected number of accepted tokens, and a cost-aware gate trains it only while the gain pays for the cost. On streams that mimic real deployments, from agents to low-resource languages and real users, AdaSpec improves the acceptance length by 10.6% and 14.7% on average at two model scales, and the end-to-end throughput by 4.3% and 4.7%, including all learning costs. These results indicate that an inference server can become faster the longer it serves.

1 INTRODUCTION

Speculative decoding (Leviathan et al., 2023; Chen et al., 2023) is a widely used technique for accelerating the inference of large language models (LLMs). A small drafter proposes several tokens, and the target model verifies them in one forward pass without changing its output (Cai et al., 2024; Li et al., 2025; Chen et al., 2026). The speedup depends on how well the drafter predicts the target on the requests it actually serves. However, drafters are trained once offline and then frozen. In deployment, they serve narrow and shifting traffic, such as the tool calls of one agent, the users of one product, or a low-resource language. On such traffic, their predictions degrade and the speedup shrinks (Yi et al., 2024; Liu et al., 2024b; Kumar et al., 2026). Yet this traffic is also the best training data a drafter could get. It is abundant and specific to the deployment, and it comes already labeled, because every verification step compares the drafter's guesses with the target's distribution. The drafter can therefore learn continually from the traffic it serves.

Motivated by this, recent work has explored training the drafter online on the traffic it serves (Liu et al., 2024b; Wang et al., 2026a; Qian et al., 2026a). These methods learn outside the inference engine. The server streams its traffic to a trainer on separate GPUs, which recomputes the drafter's forward pass and periodically swaps new weights back. This design costs extra GPUs and per-token data movement, and the drafter serves with stale weights. To avoid these costs, we advocate *in-engine continual learning*, where the drafter learns inside the inference engine, on the same GPUs and from the same forward passes that serve the traffic.

In-engine continual learning pays off only if the throughput gained from a better drafter exceeds the throughput spent on learning. This raises two challenges. (1) *Cost*: learning competes with decoding for the GPU, and a naive in-engine learner costs about half of the throughput. (2) *When to learn*: the gain saturates once the drafter has adapted, and returns only when the traffic shifts, so learning all the time wastes throughput.

In this paper, we propose **ADASPEC**, an inference engine in which the drafter learns continually from the traffic it serves. Built on SGLang (Zheng et al., 2024), it serves requests like a standard engine. The key insight is that learning can be a by-product of serving: every step already runs the drafter and the target's verification, so the learner only back-propagates through activations and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

target distributions already on the GPU (Figure 1). The remaining cost lies in per-weight work, such as weight gradients and optimizer steps, which a naive learner repeats after every step although each step trains only a few positions. ADASPEC amortizes this work over many steps and runs the backward pass concurrently with decoding, which cuts the cost of learning from 51% to about 3% of the throughput.

On top of this engine, ADASPEC decides what and when to learn. Under sampled verification, the speedup depends on how much the drafter's distribution overlaps with the target's, not on whether their top tokens agree. The drafter therefore directly maximizes the expected number of tokens that verification accepts. When to learn is a cost–benefit decision: training pays off only if its gain, collected until the traffic changes, exceeds its cost. ADASPEC casts this rule into a gate (Figure 3), which measures the gain by comparing the drafter before and after training on the same requests. The gate pauses once the gain no longer pays, and resumes training when the traffic shifts or a periodic probe finds new gain; all quantities it needs are estimated online.

We evaluate ADASPEC on streams that mimic real deployments, from software-engineering, web and tool-use agents to low-resource languages and the chronological traffic of real chat and coding users, on two target–drafter pairs: Qwen3.8-27B with DFlash2 and Qwen3.5-4B with DFlash. Every stream is served by an always-on server, and all learning is paid for inside the measured wall-clock time. ADASPEC improves the acceptance length on every workload, by *10.6%* and *14.7%* on average for the two pairs, and turns it into end-to-end throughput gains of 4.3% and 4.7%. The gains grow with the mismatch between the drafter and the traffic, reaching *23%* in throughput on low-resource languages, and they grow along each stream as the drafter adapts (Figure 4). Together, these results indicate that an inference server does not have to be static: it can become faster the longer it serves.

In summary, our contributions are threefold: (1) We present **ADASPEC**, to our knowledge the first inference engine that trains the drafter inside the engine, on the same forward pass and the same GPUs as serving. (2) We propose a **cost-aware continual learning** algorithm, which optimizes the expected acceptance length and learns only while learning pays for itself. (3) We evaluate ADASPEC on diverse real-world serving streams, from agents to real users, and show consistent end-to-end speedups at two model scales, with the largest gains where offline drafters fall short.

2 RELATED WORK

Speculative decoding. Speculative decoding accelerates autoregressive generation: a cheap drafter proposes several tokens, and the target model verifies them in parallel without changing its output distribution (Stern et al., 2018; Leviathan et al., 2023; Chen et al., 2023; Xia et al., 2024). Its speedup is governed by how many drafted tokens the target accepts, which has driven increasingly capable drafters, from small language models (Leviathan et al., 2023; Zhou et al., 2024) to heads on the target's hidden states (Cai et al., 2024; Li et al., 2024; 2025; Gloeckle et al., 2024; DeepSeek-AI, 2024) and parallel block drafters (Chen et al., 2026; Zhang et al., 2026; Cheng et al., 2026). Notably, DFlash (Chen et al., 2026) drafts a whole block in one forward pass conditioned on the target's features, so drafting becomes cheap and the speedup rests mostly on acceptance. All of these drafters are trained once, offline, on a fixed corpus of target-generated responses (Li et al., 2025; Chen et al., 2026; Lei et al., 2026). However, a frozen drafter only matches the traffic it was trained on. For example, Yi et al. (2024) have to train a separate drafter for each language to retain the speedup in multilingual settings, and Kumar et al. (2026) find that acceptance decays as outputs grow beyond the training lengths.

Continual and online learning. Continual and online learning keep updating a deployed model on a non-stationary data stream (De Lange et al., 2021; Wang et al., 2024; Shi et al., 2024), and test-time training adapts a model to the test distribution during inference (Sun et al., 2020; Hardt & Sun, 2024). Speculative decoding fits this paradigm naturally, since verification reveals the target distribution at every step and thus labels the drafter's errors for free. Online speculative decoding (Liu et al., 2024b) first exploited this signal by periodically distilling the drafter on observed queries, and OnlineSpec (Qian et al., 2026b) later analyzed the loop with online-learning theory. To handle production traffic, Aurora (Wang et al., 2026a) and AdaFlash (Qian et al., 2026a) decouple training from serving: the inference server streams its traffic to a trainer on separate GPUs, which periodically hot-swaps new weights back. Test-Time Speculation (Kumar et al., 2026) instead updates the drafter

during generation, and resets it after every request. Similar ideas keep drafters aligned with the evolving policy during RL rollouts (Hu et al., 2025; Chen et al., 2025; Wang et al., 2026b). However, these methods mostly pay for extra GPUs and serve stale weights, which incur additional cost. Moreover, they train continuously, although learning competes with serving for the GPU and its gain shrinks once the drafter has adapted. Budgeted and real-time continual learning charge the compute of learning against the stream (Prabhu et al., 2023; Ghunaim et al., 2023), and change-point detection decides when a model should adapt again (Page, 1954; Adams & MacKay, 2007). ADASPEC brings both ideas into serving: a gate learns only while the gain pays for its cost.

LLM inference engines. Inference engines such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) serve many concurrent requests on shared GPUs, with continuous batching, chunked prefill, and optimized attention kernels (Yu et al., 2022; Agrawal et al., 2024; Dao et al., 2022; Ye et al., 2025). They also integrate speculative decoding and tune it to the batch load (Miao et al., 2024; Liu et al., 2024c; Cheng et al., 2026). All of these optimizations treat the model weights as constants during serving. A few systems place training next to inference: FlexLLM (Oliaro et al., 2026) co-serves inference and parameter-efficient finetuning on the same GPUs, and RL frameworks such as HybridFlow (Sheng et al., 2025) alternate rollout generation and policy training on shared devices. In both cases, training is a separate workload that brings its own data and runs its own forward passes. In contrast, ADASPEC makes learning a by-product of serving: the drafter learns from the activations and verification results that the engine has already computed.

3 ADASPEC

We introduce ADASPEC, an inference engine in which the drafter learns continually from the traffic it serves. As shown in Figure 1, ADASPEC consists of two components: (i) *in-engine learning*, which trains the drafter inside the serving engine at a small cost; and (ii) *cost-aware continual learning*, which decides what the drafter learns and when learning pays off. We first formalize speculative decoding and offline drafter training in §3.1, and then describe in-engine learning in §3.2 and cost-aware continual learning in §3.3.

3.1 PRELIMINARIES

Speculative decoding. We formulate speculative decoding as follows. Given a context x, a target model p generates the continuation of x with the help of a drafter q_θ with parameters θ. Each decoding step consists of two stages: (i) *drafting*, where the drafter proposes a *draft* of K tokens $d_1, \ldots, d_K$, with d_j drawn from the drafter's distribution q_j at the j-th drafted position; and (ii) *verification*, where the target computes $p_j = p(\cdot \mid x, d_{<j})$ for all positions in one forward pass and accepts d_j from left to right with probability $\min\big(1, p_j(d_j)/q_j(d_j)\big)$; the first rejected position is resampled from $\mathrm{norm}\big(\max(0, p_j - q_j)\big)$ (Leviathan et al., 2023; Chen et al., 2023). The generated tokens thus follow the target distribution p exactly, and the drafter only affects the decoding speed. The speed is determined by the *acceptance length* τ, i.e., the number of tokens committed per step, whose expectation is

$$\mathbb{E}[\tau] = 1 + \sum_{j=1}^{K} \prod_{i=1}^{j} \alpha_i, \qquad \alpha_j = \sum_{v} \min\big(p_j(v),\, q_j(v)\big), \tag{1}$$

where α_j is the probability that position j is accepted. With T_{draft} and T_{verify} denoting the time of the two stages, the average time per generated token is $(T_{\text{draft}} + T_{\text{verify}})/\mathbb{E}[\tau]$.

Drafter training. Existing methods train the drafter offline and keep it frozen after deployment. Formally, given an offline dataset $\mathcal{D}_{\text{offline}}$ of prompts with responses generated by the target (Chen et al., 2026), the drafter is trained as

$$\theta^* = \arg\min_{\theta}\ \mathbb{E}_{x \sim \mathcal{D}_{\text{offline}}}\Big[\sum_{j=1}^{K} w_j\, \ell\big(p_j,\, q_{\theta,j}\big)\Big], \tag{2}$$

where ℓ is the cross-entropy toward the target's token (Li et al., 2025; Chen et al., 2026) or a divergence toward p_j (Zhou et al., 2024), and w_j are optional position weights. The resulting θ^* then serves all traffic. However, deployed traffic often differs from $\mathcal{D}_{\text{offline}}$, and a fixed θ^* cannot adapt to it.

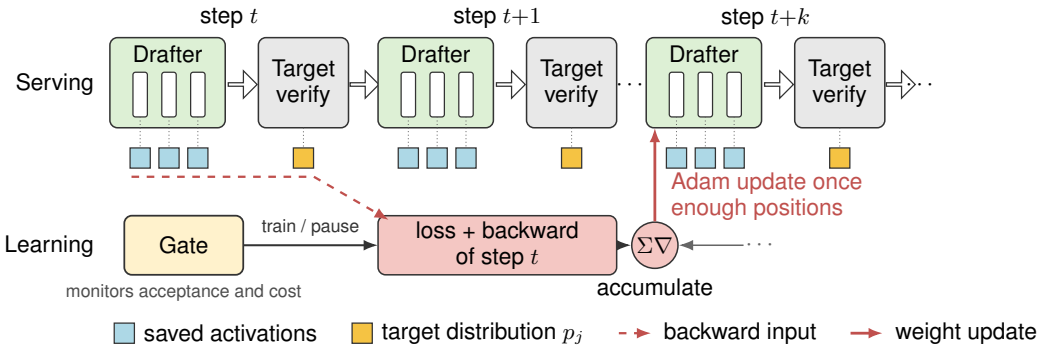

Figure 1: **Overview of AdaSpec.** Serving and learning share the same engine and GPUs. The backward pass of step t reuses the saved activations and the target distribution p_j while step $t{+}1$ decodes, and accumulated gradients update the drafter between decoding steps.

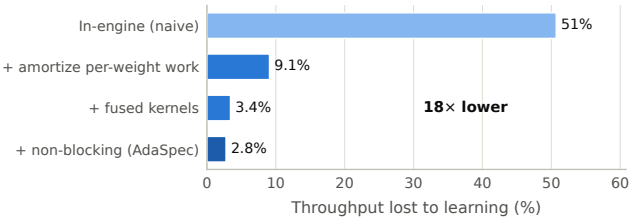

Figure 2: **Cost of in-engine learning.** Throughput lost to learning (Qwen3.8-27B with DFlash2). Each bar adds one optimization; amortizing the per-weight work gives the largest drop.

3.2 In-Engine Learning

A straightforward way to learn from served traffic is to decouple *training* from *inference* (Liu et al., 2024b; Wang et al., 2026a). The serving engine ships the target's features and outputs to a separate trainer, which recomputes the drafter's forward pass, updates the drafter, and periodically swaps the new weights back. This design needs extra GPUs, moves data for every token, repeats computation that serving has already done, and serves with stale weights. AdaSpec instead learns *inside the inference engine*, much like a just-in-time compiler that optimizes a running program in the background.

Learning in the serving loop. As shown in Figure 1, serving and learning share one forward pass. The drafter keeps its activations on the GPU, and verification provides the target distribution p_j at every drafted position. After each verification step t, the learner back-propagates the learning objective $\mathcal{J}_t$ of that step (§3.3) through the drafter, reusing the saved activations, and updates the drafter once N drafted positions have been accumulated:

$$G \leftarrow G + \nabla_\theta \mathcal{J}_t \;\text{ after every step,} \qquad \theta \leftarrow \mathrm{Adam}(\theta, G),\;\; G \leftarrow 0 \;\text{ every } N \text{ drafted positions.} \tag{3}$$

The backward pass of step t runs concurrently with step $t{+}1$, so learning does not block decoding. New weights are installed between decoding steps, so each step drafts and verifies with one version of the drafter, and verification stays lossless.

Cost of learning. Learning shares the GPU with serving. Extending the latency model of §3.1, the average time per generated token becomes

$$\frac{T_{\text{draft}} + T_{\text{verify}} + \Delta T}{\mathbb{E}[\tau]}, \tag{4}$$

where ΔT is the learning time exposed on the decoding path. Learning therefore pays off only if it raises $\mathbb{E}[\tau]$ by more than a factor of $1 + \Delta T/(T_{\text{draft}} + T_{\text{verify}})$.

Reducing the cost. A naive in-engine learner runs a complete training step after every decoding step and costs about half of the throughput (Figure 2). The reason is that a training step does two kinds of work: *(i) Per-position work*, i.e., the objective and the activation gradients, grows with the number m of drafted positions in the step. *(ii) Per-weight work*, i.e., the weight gradients and the optimizer step, grows with the size of the drafter instead, and it dominates because each step drafts only a few positions. With a and b denoting the two costs, the naive learner pays, per decoding step,

$$\Delta T_{\text{naive}} = a\,m + b. \tag{5}$$

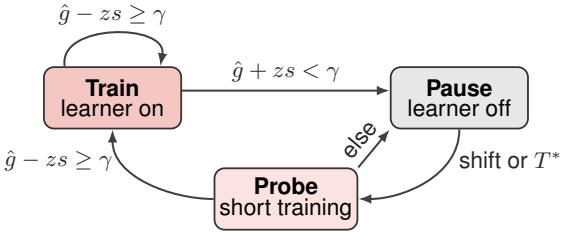

Figure 3: **The gate decides online when the learner trains.** It keeps training while the measured gain $\hat{g}$ (standard error s) confidently exceeds the break-even threshold $\gamma = c\,n/(\kappa H)$, and pauses otherwise. A short probe restarts training after a shift in acceptance length or after T^* paused windows. c, H and T^* are estimated online.

AdaSpec still performs the per-position work after every step, but amortizes the per-weight work over the N drafted positions of an update:

$$\Delta T_{\text{AdaSpec}} = a\,m + b\,m/N. \tag{6}$$

Specifically, the learner stores the per-position factors of the weight gradients and multiplies them out once for many positions, and it steps the optimizer once per update. Hence the cost of learning grows with the number of trained positions, not with the number of decoding steps. Fused kernels and non-blocking scheduling reduce the remaining cost (Appendix A); §5 measures the overhead on all streams.

3.3 Cost-Aware Continual Learning

The in-engine learner makes learning cheap, but not free. We now describe what the learner optimizes and how AdaSpec decides when learning is worth its cost.

Learning objective. While serving, the only supervision comes from verification. At every drafted position j, the target computes its distribution p_j at the request's temperature, conditioned on the drafter's own proposals; p_j serves as the teacher. Since requests are sampled, the drafter also samples its proposals from $q_{\theta,j}$, and position j is accepted with probability α_j (Eq. 1). Speed thus depends on how much $q_{\theta,j}$ overlaps with p_j, not on whether their top tokens agree. AdaSpec therefore maximizes the expected acceptance length on the served drafts:

$$\max_{\theta}\ \mathcal{J}(\theta) = \mathbb{E}_{\text{served drafts}}\Big[\sum_{j=1}^{K}\prod_{i=1}^{j}\alpha_i(\theta)\Big], \qquad \alpha_i(\theta) = \sum_{v}\min\big(p_i(v),\, q_{\theta,i}(v)\big). \tag{7}$$

Unlike the offline objectives of Eq. 2, this objective raises the drafter's probability only for tokens that it proposes less often than the target, and it weights each position by the chance that all earlier positions are accepted; cross-entropy toward the target's token instead pushes the drafter toward a single token, which sampled verification does not reward. For a greedy drafter, $\alpha_j = p_j(d_j)$ for its top-1 token d_j, and we train it with cross-entropy toward the target's most likely token.

When to learn. Training costs throughput whenever the learner runs (Eq. 4), but its benefit is large only when the traffic is new to the drafter, and it shrinks as the drafter adapts. AdaSpec therefore uses a *gate* that switches the learner between training and pausing (Figure 3), following a simple cost–benefit rule. Suppose that training for n windows of W drafted positions each raises the acceptance length by a relative gain g and costs a fraction c of the throughput. The gain lasts as long as the current traffic does, say for H more windows, and a fraction κ of it turns into throughput. Training thus pays off if

$$\kappa\, g\, H \ \geq\ c\, n. \tag{8}$$

While training, the gate measures the gain with little noise by evaluating the current weights θ and the weights θ_0 from the start of the current training segment on the same drafts B_w of each window w, $d_w = \hat{\tau}_\theta(B_w) - \hat{\tau}_{\theta_0}(B_w)$, where $\hat{\tau}$ is the expected acceptance length of Eq. 1. Fitting $d_1, \ldots, d_n$ gives the gain $\hat{g}$ with standard error s, and the gate decides with confidence z:

$$\text{keep training if } \hat{g} - z\,s \geq \gamma, \qquad \text{pause if } \hat{g} + z\,s < \gamma, \qquad \text{where } \gamma = \frac{c\,n}{\kappa H}. \tag{9}$$

While paused, the learner costs nothing, and the gate restarts training for a short *probe*, judged by the same rule, when a CUSUM change detector (Page, 1954) signals new traffic, or periodically every $T^* = \sqrt{2cn/(\kappa\delta)}$ windows, where δ is the rate at which learnable gain reappears. None of c, H and δ is set by hand: the gate estimates them online from the time per drafted position, the rate of detected shifts, and the gains found by probes (Appendix C).

4 EXPERIMENTAL SETUP

Our experiments answer four research questions. **RQ1** (Table 1 and Figure 4): How much does ADASPEC improve the acceptance length and the end-to-end throughput on realistic serving streams? **RQ2** (Figures 5 and 6): Does learning pay for itself, and how does the gain depend on the serving load? **RQ3** (Figure 7): Which drafter parameters and which objective give the best trade-off between gain and cost? **RQ4** (Figure 8): How does learning evolve along a stream, i.e., when does the gate train, how does it react to traffic shifts, and how much does it specialize to individual users?

Datasets. We select diverse, real-world datasets with long streams of real traffic, from agents and real users to low-resource languages, which fall into seven workload groups (statistics in Table 3, Appendix D): (i) **agents**: software-engineering, terminal, and customer-service agents from SWE-Gym (Pan et al., 2025), OpenThoughts-Agent (Raoof et al., 2026), and the τ-bench-style (Yao et al., 2024) APIGen-MT (Prabhakar et al., 2025); (ii) **web**: browser and desktop agents from NNetNav-Live (Murty et al., 2024) and OSWorld (Xie et al., 2024); (iii) **low-resource languages**: translation from English into Swahili, Zulu, and Hindi with sources from WMT24++ (Deutsch et al., 2025) and XL-Sum (Hasan et al., 2021), and Swahili and Hindi news writing and summarization from XL-Sum; (iv) **coding users**: the coding-agent sessions of the five most active developers in SWE-chat (Baumann et al., 2026); (v) **chat users**: the conversations of four heavy WildChat (Zhao et al., 2024) users; (vi) **reasoning**: puzzle games from Reasoning Gym (Stojanovski et al., 2025); and (vii) **function calling**: BFCL v3 (Patil et al., 2025) and ToolACE (Liu et al., 2024a). Agent and user sessions are replayed in their recorded chronological order. Appendix D describes the construction.

Models. We study two target–drafter pairs, both starting from their publicly released drafters: (i) **Qwen3.8-27B with DFlash2** (Inco AI, 2026), a 5-layer block drafter with 1.92B parameters that drafts 7 tokens per step by sampling a path through its top-16 candidates, served with tensor parallelism 2; and (ii) **Qwen3.5-4B with DFlash** (Chen et al., 2026), a 6-layer block drafter with 634M parameters that drafts 7 tokens per step greedily, served on one GPU. All experiments run on NVIDIA H100-80GB GPUs.

Training. ADASPEC trains only the drafter; the target model stays frozen. By default, ADASPEC trains the last draft layers, together with the candidate selector of DFlash2, i.e., 701M of the 1.92B drafter parameters on Qwen3.8-27B and 142M of 634M on Qwen3.5-4B (Figure 7 compares other choices). The learner maximizes the objective of Eq. 7 for DFlash2 and its greedy counterpart for DFlash, with AdamW at a learning rate of 3×10^{-4} and 1.5×10^{-4}, respectively. Each optimizer update accumulates the gradients of 16K (Qwen3.8-27B) and 2K (Qwen3.5-4B) drafted positions, i.e., roughly 10K and 1K generated tokens. Both models share the gate hyperparameters of Table 2. Appendix B lists the full configuration.

Baselines. We compare ADASPEC with the following configurations: (i) **Frozen** serves the released drafter without learning, and is the reference for all gains; (ii) **Always** is ADASPEC without the gate, which trains after every verification step; (iii) **No speculation** decodes with the target alone; and (iv) **Aurora-DFlash** follows the decoupled design of Aurora (Wang et al., 2026a), re-implemented for DFlash (Appendix E). All configurations of a stream serve the same requests at the same time on identical, disjoint GPUs, with $C{=}8$ requests in flight and responses sampled at temperature 0.7.

Metrics. The *acceptance length* τ is the average number of tokens committed per verification step. The *throughput* is the number of output tokens divided by the wall-clock time, including all learning and the final drain. We report the relative gains of ADASPEC over the frozen drafter, where the gain of a workload group is the mean of the per-task gains. All metrics are computed on the prefix of each stream in which the frozen drafter generates 2M tokens; we also report the throughput gain on the second half of this prefix, after most of the adaptation.

5 EXPERIMENTAL RESULTS

Performance across workloads. Table 1 summarizes the gains of ADASPEC across workloads. In this setting, ADASPEC starts from the released drafter and learns online on each evaluation stream

Table 1: **Acceptance length** τ of the frozen drafter and AdaSpec (default configuration) within the first 2M generated tokens of each benchmark stream, served from its first request (learning included). Each column averages the tasks of a workload group; in parentheses: gain over the frozen drafter (%).

Method	**Agent** SWE-Gym, OH-Agent, τ-bench	**Web** NNetNav-Live, OSWorld	**Low-res.** WMT24++, XL-Sum	**Code** SWE-chat	**Chat** WildChat	**Reason.** Reasoning Gym	**Func.** BFCL, ToolACE	**Avg.**
Qwen3.8-27B + DFlash2								
Frozen	4.43	3.65	1.68	3.88	3.27	5.71	5.44	4.01
AdaSpec	4.77(+7.7)	3.83(+5.2)	2.31(+40.1)	4.05(+4.4)	3.39(+3.4)	5.99(+4.8)	5.90(+8.5)	4.32(+10.6)
Qwen3.5-4B + DFlash								
Frozen	3.69	3.27	1.78	3.36	3.23	4.82	4.50	3.52
AdaSpec	4.76(+32.0)	3.65(+11.7)	2.28(+28.2)	3.69(+9.9)	3.39(+4.6)	5.16(+7.0)	4.91(+9.1)	3.98(+14.7)

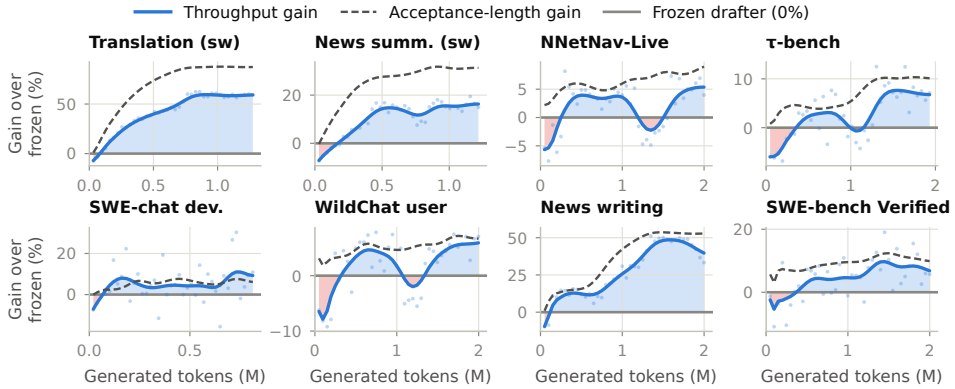

Figure 4: **The drafter gets progressively better while serving.** Gain of AdaSpec over the frozen drafter on the same requests along eight streams (Qwen3.8-27B + DFlash2). Learning first costs some throughput (red), after which AdaSpec overtakes the frozen drafter (blue).

while serving it, i.e., each request is served before the drafter learns from it, and all metrics include the time spent on learning. Our findings highlight that the drafter becomes a better drafter for the traffic it serves. First, AdaSpec consistently improves the acceptance length across all workload groups and both models, by **10.6%** on Qwen3.8-27B and **14.7%** on Qwen3.5-4B on average. Notably, these gains are obtained within only the first 2M generated tokens of each stream. Second, the gains are largest where the offline drafter fits the traffic worst. On low-resource languages, AdaSpec raises the acceptance length by **40.1%** and 28.2% for the two models, and the smaller drafter gains 32.0% on agent traffic and 11.7% on web traffic. In contrast, the gains on chat traffic are the smallest (3.4% and 4.6%). Chat is open-ended, with diverse questions and little shared structure, so the drafter improves more slowly.

Learning along the stream. Figure 4 shows how learning unfolds along a stream. The acceptance length of AdaSpec rises steadily above that of the frozen drafter. The throughput follows a different pattern. At the start of a stream, it is usually below that of the frozen drafter, because learning already costs GPU time while the drafter has not improved yet. As the drafter adapts, the throughput catches up and then overtakes the frozen drafter, e.g., by about 60% on Swahili translation. Overall, AdaSpec improves the end-to-end throughput by *4.3%* and *4.7%* on average for the two models (Table 10, Appendix F), and by 6.9% and 5.4% on the second half of each stream, as the gains keep growing with the stream.

Performance by concurrency. The results above serve $C{=}8$ requests at a time. To examine how the gain depends on the serving load, we serve two streams at concurrency levels up to $C{=}64$: news writing, a single-turn generation task, and OSWorld, an agent task. We report the GPU throughput and the per-user speed, i.e., the tokens each request receives per second. As shown in Figure 5, AdaSpec consistently improves over the frozen drafter at every concurrency level. For example, at $C{=}8$, it serves 43% (news writing) and 13% (OSWorld) more tokens per second at the same per-user

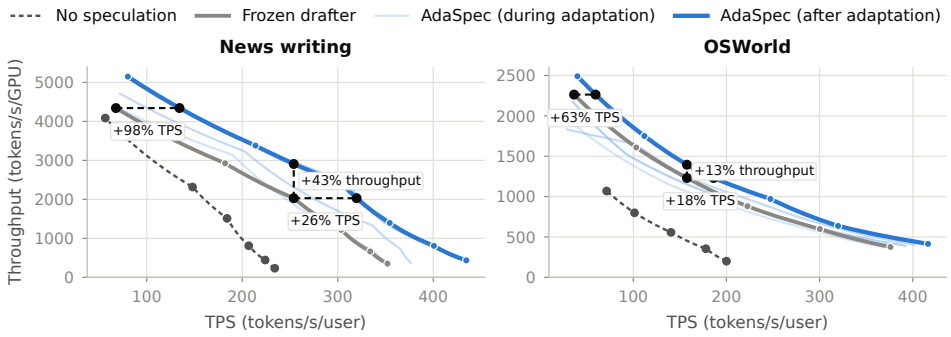

Figure 5: **Serving speed under load** (Qwen3.5-4B + DFlash). Each marker is one concurrency level C, and up-right is better. AdaSpec is shown after adaptation (second half of each stream; faint lines: during adaptation), and annotations give its gain at the frozen drafter's $C{=}8$ and $C{=}64$ operating points, at the same per-user speed and at the same throughput.

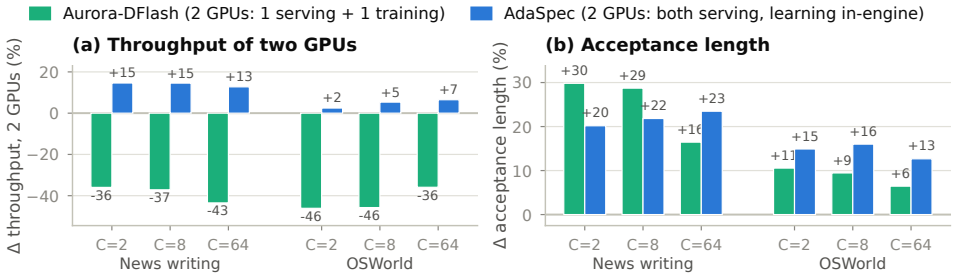

Figure 6: **In-engine vs. decoupled online learning** (Qwen3.5-4B + DFlash). Gain over the frozen drafter at the same concurrency C. Both use two GPUs: AdaSpec serves and learns on both, Aurora-DFlash serves on one and trains on the other.

speed, or raises the per-user speed by 26% and 18% at the same throughput. The gain holds at $C{=}64$, where the GPU is saturated and learning competes directly with decoding.

Comparison with decoupled learning. We next compare in-engine learning with the decoupled design of Aurora (Figure 6). Both designs use two GPUs: Aurora-DFlash serves on one GPU and trains on the other, while AdaSpec serves and learns on both. In terms of acceptance length, Aurora-DFlash is better on news writing at low and medium concurrency, since its dedicated GPU trains the full drafter (Appendix E), whereas AdaSpec trains only part of it. On OSWorld, however, AdaSpec achieves the higher acceptance gain at every concurrency, possibly because its objective directly targets the acceptance under sampled verification. The picture is different in throughput. Aurora-DFlash loses 36–46% of the throughput of the two GPUs: its trainer GPU is fully busy, but with training rather than serving (Appendix E). AdaSpec instead keeps both GPUs serving, covers the cost of learning, and still improves the throughput by up to 15%. This underscores the advantage of learning inside the engine.

What to learn. Which parameters to train is a trade-off: training more of the drafter raises the attainable acceptance gain, but it also raises the cost of every training step. Figure 7(a, b) shows this trade-off on news writing and OSWorld. For Qwen3.5-4B, training more parameters raises the acceptance gain on news writing from 11.5% (81M) to 29.8% (425M). However, the throughput gain peaks at a medium scope, our default of 142M, and the largest scope even loses throughput on OSWorld. The gate is also essential. Without it, always training yields a smaller throughput gain at larger scopes and loses throughput on OSWorld, because it keeps paying for updates after the gain has saturated. Figure 7(c) compares our objective with alternatives under sampled verification on Qwen3.8-27B. Maximizing the expected acceptance length (Eq. 7) raises the acceptance length by 1.6%, whereas the hard-label cross-entropy, sampled labels, forward KL, and one-sided KL objectives all lower it. These objectives imitate the target's top token or match the two distributions, which does not directly reward the overlap that sampled verification accepts. Without the small cross-entropy anchor, our objective keeps about half of its gain.

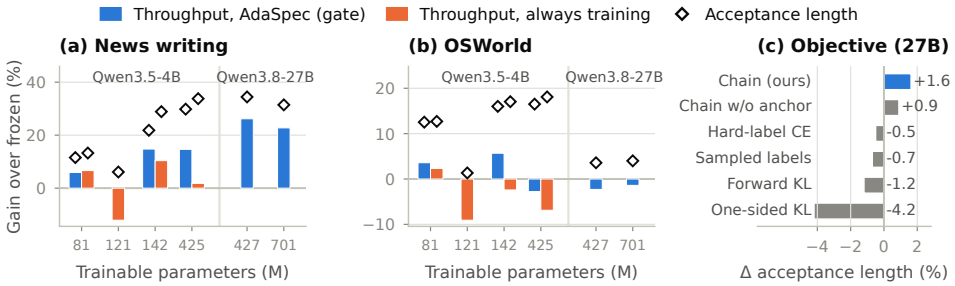

Figure 7: **What to train.** (a, b) Gain over the frozen drafter when training more of the drafter (C=8); the defaults are 142M (Qwen3.5-4B) and 701M (Qwen3.8-27B), and the 121M setting trains the native MTP head instead. (c) Acceptance gain of different objectives (Qwen3.8-27B); *Chain* maximizes the expected acceptance length (Eq. 7), and the anchor is a small cross-entropy term.

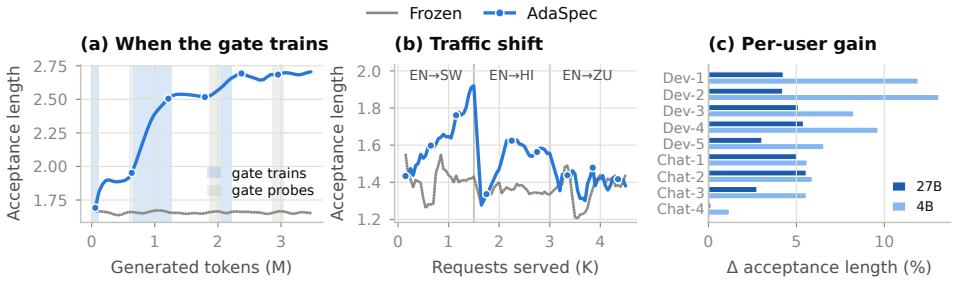

Figure 8: **Learning dynamics.** (a) The gate trains in bursts (shaded) and pauses once acceptance plateaus (Qwen3.8-27B, news writing). (b) A translation stream whose target language switches twice (Qwen3.5-4B). (c) Acceptance gain for each real user.

Learning dynamics. Figure 8(a) shows how the gate schedules learning along a news-writing stream. The gate trains in bursts: it trains while the acceptance length still rises, pauses once it plateaus, and periodically probes whether training pays off again. A probe at about 1.9M tokens finds further gain and restarts training, whereas the last probe finds none, and the gate stays paused. Overall, the acceptance length rises from 1.7 to 2.7, while the learner is paused for most of the stream. To examine how AdaSpec adapts when the traffic changes, we serve a translation stream whose target language switches from Swahili to Hindi and then to Zulu (Figure 8(b)). After each switch, the acceptance length first drops, because what the drafter has learned is specific to the previous language. AdaSpec then learns again and adapts to the new language within several hundred requests, e.g., from 1.3 to 1.6 on Hindi, while the frozen drafter stays at about 1.4. Finally, Figure 8(c) and Table 7 (Appendix F) break down the gain by real user. All users benefit, but the gain varies across users. Users with a focused topic gain the most: software developers, whose requests revolve around their own repositories, gain 6.6–13.1% with Qwen3.5-4B. Chat users, whose conversations span many unrelated topics, gain less (1.2–5.9%).

6 Conclusion

In this paper, we introduced **AdaSpec**, an inference engine in which the drafter learns continually from the traffic it serves. AdaSpec makes learning a by-product of serving: the learner back-propagates through the activations and target distributions that decoding already produces, amortizes the per-weight work over many positions, and runs concurrently with decoding. Coupled with cost-aware continual learning, which maximizes the expected acceptance length and trains only while the gain pays for its cost, AdaSpec turns deployment traffic into speed without extra GPUs. Experiments on real-world serving streams, from agents to low-resource languages, show that AdaSpec improves the acceptance length on every workload and the end-to-end throughput at two model scales, with the largest gains where offline drafters fall short. These results suggest that an inference server can become faster the longer it serves.

AI USE STATEMENT

In this work, we used generative AI tools to polish human-written drafts and check the validity of mathematical formulations. Additionally, we used generative AI tools to create analysis code and perform certain scaled evaluations. We reviewed all AI-assisted work, and LLM-generated code was verified and tested for correctness by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Ryan Prescott Adams and David J. C. MacKay. Bayesian online changepoint detection. *arXiv preprint arXiv:0710.3742*, 2007.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. SWE-chat: Coding agent interactions from real users in the wild. *arXiv preprint arXiv:2604.20779*, 2026.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *International Conference on Machine Learning (ICML)*, 2024.

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

Jian Chen, Yesheng Liang, and Zhijian Liu. DFlash: Block diffusion for flash speculative decoding. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2602.06036.

Qiaoling Chen, Zijun Liu, Peng Sun, Shenggui Li, Guoteng Wang, Ziming Liu, Yonggang Wen, Siyuan Feng, and Tianwei Zhang. ReSpec: Towards optimizing speculative decoding in reinforcement learning systems. *arXiv preprint arXiv:2510.26475*, 2025.

Xin Cheng, Xingkai Yu, Chenze Shao, Jiashi Li, Yunfan Xiong, Yi Qian, Jiaqi Zhu, Shirong Ma, Xiaokang Zhang, Jiasheng Ye, Qinyu Chen, Chengqi Deng, Jiping Yu, Damai Dai, Zhengyan Zhang, Yixuan Wei, Yixuan Tan, Wenkai Yang, Runxin Xu, Yu Wu, Zhean Xu, Xuanyu Wang, Muyang Chen, Rui Tian, Xiao Bi, Zhewen Hao, Shaoyuan Chen, Huanqi Cao, Wentao Zhang, Anyi Xu, Huishuai Zhang, Dongyan Zhao, and Wenfeng Liang. DSpark: Confidence-scheduled speculative decoding with semi-autoregressive generation. *arXiv preprint arXiv:2607.05147*, 2026.

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

Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Daniel Deutsch, Eleftheria Briakou, Isaac Caswell, Mara Finkelstein, Rebecca Galor, Juraj Juraska, Geza Kovacs, Alison Lui, Ricardo Rei, Jason Riesa, Shruti Rijhwani, Parker Riley, Elizabeth Salesky, Firas Trabelsi, Stephanie Winkler, Biao Zhang, and Markus Freitag. WMT24++: Expanding the language coverage of WMT24 to 55 languages & dialects. *arXiv preprint arXiv:2502.12404*, 2025.

Yasir Ghunaim, Adel Bibi, Kumail Alhamoud, Motasem Alfarra, Hasan Abed Al Kader Hammoud, Ameya Prabhu, Philip H. S. Torr, and Bernard Ghanem. Real-time evaluation in online continual learning: A new hope. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. In *International Conference on Machine Learning (ICML)*, 2024.

Moritz Hardt and Yu Sun. Test-time training on nearest neighbors for large language models. In *International Conference on Learning Representations (ICLR)*, 2024.

Tahmid Hasan, Abhik Bhattacharjee, Md. Saiful Islam, Kazi Mubasshir, Yuan-Fang Li, Yong-Bin Kang, M. Sohel Rahman, and Rifat Shahriyar. XL-Sum: Large-scale multilingual abstractive summarization for 44 languages. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 4693–4703, 2021.

Douglas M. Hawkins. Self-starting CUSUM charts for location and scale. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 36(4):299–316, 1987.

Qinghao Hu, Shang Yang, Junxian Guo, Xiaozhe Yao, Yujun Lin, Yuxian Gu, Han Cai, Chuang Gan, Ana Klimovic, and Song Han. Taming the long-tail: Efficient reasoning RL training with adaptive drafter. *arXiv preprint arXiv:2511.16665*, 2025.

Inco AI. DFlash 2: Keep drafting parallel. `https://inco.ai/blog/dflash2/`, 2026. Blog post.

Avinash Kumar, Sujay Sanghavi, and Poulami Das. Test-time speculation. *arXiv preprint arXiv:2605.09329*, 2026.

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

Haodi Lei, Yafu Li, Haoran Zhang, Shunkai Zhang, Qianjia Cheng, Xiaoye Qu, Ganqu Cui, Bowen Zhou, Ning Ding, Yun Luo, and Yu Cheng. Draft-OPD: On-policy distillation for speculative draft models. *arXiv preprint arXiv:2605.29343*, 2026.

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

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning (ICML)*, 2024.

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.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, et al. ToolACE: Winning the points of LLM function calling. *arXiv preprint arXiv:2409.00920*, 2024a.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Online speculative decoding. In *International Conference on Machine Learning (ICML)*, 2024b.

Xiaoxuan Liu, Jongseok Park, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Chen Zhang, Kuntai Du, Xiangxi Mo, Kaichao You, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. TurboSpec: Closed-loop speculation control system for optimizing LLM serving goodput. *arXiv preprint arXiv:2406.14066*, 2024c.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating large language model serving with tree-based speculative inference and verification. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

Shikhar Murty, Hao Zhu, Dzmitry Bahdanau, and Christopher D. Manning. NNetNav: Unsupervised learning of browser agents through environment interaction in the wild. *arXiv preprint arXiv:2410.02907*, 2024.

Gabriele Oliaro, Xupeng Miao, Xinhao Cheng, Vineeth Kada, Mengdi Wu, Ruohan Gao, Yingyi Huang, Remi Delacourt, April Yang, Yingcheng Wang, Colin Unger, and Zhihao Jia. FlexLLM: Token-level co-serving of LLM inference and finetuning with SLO guarantees. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2026.

E. S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *International Conference on Machine Learning (ICML)*, 2025.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley Function Calling Leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *International Conference on Machine Learning (ICML)*, 2025.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. APIGen-MT: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *arXiv preprint arXiv:2504.03601*, 2025.

Ameya Prabhu, Hasan Abed Al Kader Hammoud, Puneet Dokania, Philip H. S. Torr, Ser-Nam Lim, Bernard Ghanem, and Adel Bibi. Computationally budgeted continual learning: What does matter? In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

Yu-Yang Qian, Hao-Cong Wu, Chen Chen, Jiacheng Sun, Zhenhua Dong, Peng Zhao, and Zhi-Hua Zhou. AdaFlash: Adaptive speculative decoding via on-policy distilled diffusion drafters. *arXiv preprint arXiv:2607.19223*, 2026a.

Yu-Yang Qian, Hao-Cong Wu, Yichao Fu, Hao Zhang, and Peng Zhao. When drafts evolve: Speculative decoding meets online learning. In *International Conference on Machine Learning (ICML)*, 2026b.

Negin Raoof, Richard Zhuang, Marianna Nezhurina, Etash Guha, Atula Tejaswi, Ryan Marten, et al. OpenThoughts-Agent: Data recipes for agentic models. *arXiv preprint arXiv:2606.24855*, 2026.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *European Conference on Computer Systems (EuroSys)*, 2025.

Haizhou Shi, Zihao Xu, Hengyi Wang, Weiyi Qin, Wenyuan Wang, Yibin Wang, Zifeng Wang, Sayna Ebrahimi, and Hao Wang. Continual learning of large language models: A comprehensive survey. *arXiv preprint arXiv:2404.16789*, 2024.

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

Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. Reasoning Gym: Reasoning environments for reinforcement learning with verifiable rewards. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei A. Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International Conference on Machine Learning (ICML)*, 2020.

Junxiong Wang, Fengxiang Bie, Jisen Li, Zhongzhu Zhou, Zelei Shao, Yubo Wang, Yinghui Liu, Qingyang Wu, Avner May, Sri Yanamandra, Ce Zhang, Tri Dao, Percy Liang, Ben Athiwaratkun, Shuaiwen Leon Song, Chenfeng Xu, and Xiaoxia Wu. When RL meets adaptive speculative training: A unified training–serving system. *arXiv preprint arXiv:2602.06932*, 2026a.

Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

Zili Wang, Zhaopeng Qiu, Yuekai Zhang, Shuang Yu, and Junjie Lai. Online draft co-training for speculative decoding in large-scale, long-context RL post-training. *arXiv preprint arXiv:2609.07108*, 2026b.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. In *Findings of the Association for Computational Linguistics: ACL*, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Conference on Machine Learning and Systems (MLSys)*, 2025.

Euiin Yi, Taehyeon Kim, Hongseok Jeung, Du-Seong Chang, and Se-Young Yun. Towards fast multilingual LLM inference: Speculative decoding and specialized drafters. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-based generative models. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

Jiebin Zhang, Zhenghan Yu, Song Liu, Eugene J. Yu, Zheng Li, Dawei Zhu, Jiangshan Duo, Weimin Xiong, Yifan Song, Guanghua Yu, Jianchen Zhu, and Sujian Li. DFlare: Scaling up draft capacity for block diffusion speculative decoding. *arXiv preprint arXiv:2606.02091*, 2026.

Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. WildChat: 1M ChatGPT interaction logs in the wild. In *International Conference on Learning Representations (ICLR)*, 2024.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. DistillSpec: Improving speculative decoding via knowledge distillation. In *International Conference on Learning Representations (ICLR)*, 2024.

A Implementation Details of the In-Engine Learner

We implement the in-engine learner in SGLang for both setups. Both implementations follow the design of §3.2; they differ in how they schedule the backward pass.

Qwen3.5-4B. The learner runs on the serving stream right after each verification step. The serving CUDA graph exposes the activations of the drafter as graph outputs, and the backward pass borrows them before the next graph replay overwrites them. Only the residual inputs, which later fused operations modify in place, are copied. This implementation does not yet overlap the backward pass

with the next decoding step. The learner accumulates the gradients of 2,048 drafted positions before each optimizer update and computes the MLP weight gradients in factorized form.

Qwen3.8-27B. The learner runs on its own CUDA stream with its own NCCL communicator and overlaps with decoding. Because serving overwrites its activations before the learner runs, the learner uses gradient checkpointing: the capture path writes only the inputs of the trained layers into a ring buffer, and the learner recomputes these layers. A single verification step yields too few rows to use the GPU efficiently, so the learner groups the rows of several consecutive steps (at least 32 drafts, i.e., 224 drafted positions) into one backward pass, and accumulates the gradients of 16,384 drafted positions before each optimizer update. The serving stream never waits on the learner on the GPU; the host only checks, through CUDA events, that a ring slot has been consumed before serving writes into it again. New weights are published at the start of the next backward pass, so no gradient is computed with outdated weights.

CUDA graphs and kernels. On Qwen3.8-27B, each backward pass, including the loss and the gradient accumulation, is captured as one CUDA graph per batch size, with batch sizes rounded up to a multiple of four drafts; padded positions carry no loss. The learner uses fused kernels for the full-vocabulary cross-entropy under tensor parallelism, the sparse soft cross-entropy, and the attention over the cached context that the second-to-last-layer scope requires. Weight gradients are accumulated in FP32 directly from the BF16 serving weights, and the optimizer is a fused AdamW.

Tensor-parallel consistency. Under tensor parallelism, the gradients of replicated parameters (normalization layers, convolutions and the selector) differ in their last bits across ranks. Left alone, the replicated weights drift apart, the ranks make different drafting decisions, and their collective operations eventually deadlock. We average these gradients across ranks before every optimizer step, broadcast the training targets from rank 0, and check a fingerprint of the replicated weights after every update.

Memory. Optimizer state is preallocated at startup. To make room for it, we reduce the KV-cache pool of the server, identically for AdaSpec and for the frozen drafter.

B Training Configuration

Qwen3.8-27B with DFlash2. AdaSpec trains 700.6M of the 1.92B drafter parameters: the MLP, the post-attention and final normalization layers, the attention output projection, and the convolutions of the last draft layer; the MLP, the post-attention normalization, and the MLP convolution of the second-to-last layer; and the candidate selector (its hidden projection and its predecessor and successor codebooks). The learner maximizes Eq. 7 over the 16 candidates at each drafted position, plus a cross-entropy term toward the target's renormalized top-32 distribution with weight 0.1, which keeps the candidate set stable. It uses AdamW ($\beta = (0.9, 0.999)$, weight decay 0.01, gradient clipping 1.0) with a learning rate of 3×10^{-4} for the draft layers and 10^{-4} for the selector, 30 warmup updates; it accumulates the gradients of 16,384 drafted positions (roughly 10K generated tokens) before each optimizer update. The gate starts from a training-cost prior of 0.10 and does not pause before 32 optimizer updates.

Qwen3.5-4B with DFlash. AdaSpec trains 141.6M of the 634M drafter parameters: the MLPs and post-attention normalization layers of the last two draft layers, and the final normalization layer. The learner minimizes the cross-entropy toward the target's most likely token at every drafted position, using Adam ($\beta_1 = 0$, $\beta_2 = 0.95$) with a learning rate of 1.5×10^{-4}; it accumulates the gradients of 2,048 drafted positions (roughly 1K generated tokens) before each optimizer update. The gate starts from a training-cost prior of 0.025.

Frozen parameters. In both drafters, the parameters that produce the drafter's KV cache from the target's features, the embedding, and the LM head shared with the target stay frozen, so cached entries remain valid after every update.

C Details of the Gate

Algorithm 1 gives the gate, and Table 2 lists its hyperparameters. The same controller runs on both models with the same hyperparameters, except for the prior of the training cost and the minimum

Algorithm 1 The gate, run once per window of W drafted positions.

update the discounted estimates of σ, c, H and δ
if state is TRAIN or PROBE **then**
  if the level leaves the recent trend by more than $h\sigma$ **then**
    record a change point; start a PROBE with $\theta_0 \leftarrow \theta$
  else
    append the window's signal to the segment; $n \leftarrow$ segment length
    fit the segment gain $\hat{g}$ and its standard error s; $\gamma_n \leftarrow \max(g_{\min},\, c\,n/(\kappa H))$
    if $n \geq n_{\min}$ and $\hat{g} - z_e s \geq \gamma_n$ **then**
      keep training; start a new segment with $\theta_0 \leftarrow \theta$
    else if $n \geq n_{\min}$ and $\hat{g} + z_e s < \gamma_n$ **then**
      pause
    else if $n = n_{\max}$ **then**
      keep training with a new segment if $\hat{g} - zs \geq \gamma_n$, and pause otherwise
else ▷ paused: the learner is off
  update the two-sided CUSUM statistics of the level
  if a CUSUM statistic exceeds $h\sigma$ **then**
    record a change point; start a PROBE with $\theta_0 \leftarrow \theta$
  else if the gate has been paused for $T^* = \sqrt{2cn_{\min}/(\kappa\delta)}$ windows **then**
    start a periodic PROBE with $\theta_0 \leftarrow \theta$

number of updates before the first pause. It runs on the host once per window and reads the window statistics from the GPU one window late, so it never stalls decoding. Under tensor parallelism, all ranks take identical decisions: the wall-clock times that enter the cost estimate are broadcast from rank 0.

Signals. The *level* of a window is its expected acceptance, computed from the target's probabilities as in Eq. 1 and normalized by its maximum; it is a lower-variance version of the realized acceptance length. The *paired* signal of a window is the difference between the expected acceptance under the current weights and under the reference weights θ_0, the weights at the start of the current segment, evaluated on the same drafted positions every few verification steps. It is exactly zero when the reference is refreshed and is several times less noisy than the level. The gate uses the paired signal when the captured tensors allow replaying the reference, and the level otherwise. For the paired signal, the segment gain is the least-squares slope through the origin times the segment length; for the level, it is the ordinary least-squares slope times the segment length. The gain is converted into a relative gain in acceptance length. Its standard error pools the residuals of earlier segments, so that a four-window segment is not judged on two or three degrees of freedom.

Change detection. While training, a window whose level falls more than $h\sigma$ off the trend of the recent windows marks a change point and restarts the segment as a probe. While paused, a self-starting two-sided CUSUM (Page, 1954; Hawkins, 1987) standardizes each window against the mean of all windows since the pause and raises an alarm when either cumulative sum exceeds $h\sigma$, where σ is the window-to-window noise of the level.

Online estimates. All statistics are discounted by a factor $1 - 1/M$ per window. The cost c is measured once per switch between training and pausing: from the last windows on both sides of the switch (at least four each, at similar levels), $c = 1 - \mathrm{median}(\text{time per drafted position, paused})/\mathrm{median}(\text{time per drafted position, training})$, and the estimate is an exponential moving average of these measurements. The horizon is the reciprocal of the posterior-mean rate of change points, $H = (H_0 + S)/(1 + N)$, where S and N are the discounted numbers of windows and of detected change points. The drift rate is $\delta = (g_0 + \sum_k g_k)/(T_0 + \sum_k T_k)$, where g_k is the gain found by a periodic probe after T_k paused windows, and the prior (g_0, T_0) is chosen such that $T^* = T_0$ before any probe.

D Construction of the Evaluation Streams

Agent and web streams. Each stream replays recorded agent trajectories step by step: (i) SWE-Gym: the sampled OpenHands trajectories released with SWE-Gym (Pan et al., 2025); (ii) OpenThoughts-

Table 2: **Hyperparameters of the gate**, shared by both models.

Symbol	Meaning	Value
W	drafted positions per window	28,672
$n_{\min}, n_{\max}$	minimum and maximum segment length (windows)	4, 16
z_e, z	evidence for an early decision and at the end of a segment	2.5, 2
h	change-detection threshold (in units of σ)	8
κ	acceptance-length gain turned into throughput	0.5
M	memory of all statistics (windows)	1,024
H_0	prior of the horizon (windows)	24
T_0	prior of the periodic-probe interval (windows)	64
$g_{\min}$	minimum gain threshold	0.002
c (prior)	training cost before the first measurement	0.10 (27B), 0.025 (4B)
–	optimizer updates before the first pause (27B)	32

Table 3: **Evaluation streams.** Each stream is served by one always-on server. Agent and user streams replay recorded sessions step by step. #Req.: requests in the stream; #Sess.: recorded sessions; Out.: mean generated tokens per request on Qwen3.8-27B. Results use the prefix of each stream in which the frozen drafter generates 2M tokens.

Group	Stream	Thinking	#Req.	#Sess.	Out.
Agent	SWE-Gym (OpenHands)	✓	5,724	223	236
	OpenThoughts-Agent (Terminus-2)	✓	6,000	785	908
	APIGen-MT (τ-bench style)	✓	6,000	650	322
Web	NNetNav-Live (browser)	✓	5,876	622	639
	OSWorld (desktop)	✓	8,000	360	244
Low-res.	Translation en→sw / zu / hi	✗	9,959 each	–	128–149
	News writing sw (XL-Sum)	✗	4,000	–	874
	News summarization sw (XL-Sum)	✗	9,583	–	127
Code	SWE-chat (5 developers)	✓	13,318	235	381
Chat	WildChat (4 users)	✓	3,219	2,659	2,433
Reason.	Reasoning Gym (4 puzzle games)	✗	6,000	–	753
Func.	BFCL v3 + ToolACE	✗	15,107	–	62

Agent: terminal-task trajectories in the Terminus-2 harness (Raoof et al., 2026); (iii) APIGen-MT: the 5K customer-service trajectories of APIGen-MT (Prabhakar et al., 2025); (iv) NNetNav-Live: browser trajectories on live websites (Murty et al., 2024); and (v) OSWorld: trajectories of Claude Opus 5 with the computer-use tool on OSWorld-Verified tasks (Xie et al., 2024), replayed as text, with each screenshot replaced by a short text marker. Each request is one agent step together with the real history of its trajectory; the target generates the next step, and the following request continues with the recorded step. Tool calls and tool results are rendered in the target's native tool-call format, and tool outputs are truncated to 500 characters. Steps of a trajectory are sent strictly in order, and trajectories are interleaved to reach the target concurrency.

Low-resource languages. Translation streams contain the 959 English segments of WMT24++ (Deutsch et al., 2025) in document order, followed by 9,000 chunks of English XL-Sum articles (Hasan et al., 2021), translated into Swahili, Zulu, and Hindi. The news-writing stream asks the target to write a Swahili news article from the headline and summary of an XL-Sum article; the summarization stream summarizes Swahili XL-Sum articles.

User streams. *Coding users.* We select the five SWE-chat (Baumann et al., 2026) developers with the most sessions and choose whole sessions evenly over their timelines, up to 3,000 agent steps per developer. *Chat users.* We select heavy WildChat (Zhao et al., 2024) users (by hashed IP) and choose whole conversations evenly over each user's timeline, up to 1,000 user turns. *Replay.* Each request consists of one user turn or agent step together with the real history of its session; the target generates the next turn, and the following request contains the originally recorded response. When the history exceeds the context budget (3,900 tokens for chat, 15,500 for coding agents), it is compacted at once

to half the budget, as coding agents do, so that consecutive requests share a stable prefix and prefix caching behaves as in production. User identities are anonymized.

Other tasks. The reasoning stream contains 6,000 instances of four Reasoning Gym (Stojanovski et al., 2025) puzzle games (Tower of Hanoi, Sudoku, Game of Life, and N-Queens). The function-calling stream contains the single-turn categories of BFCL v3 (Patil et al., 2025) and ToolACE (Liu et al., 2024a) requests, 15,107 in total. Appendix F additionally reports XL-Sum (Japanese), MeetingBank, VCSUM, and Spider with WikiSQL on both models, and AgentNet and a multi-developer SWE-chat stream on Qwen3.5-4B.

Generation settings. Thinking is enabled for agent and user streams, with budgets of 2,048 thinking tokens and 2,048 answer tokens, and disabled for the other tasks, which generate at most 1,024 tokens. Responses are sampled at temperature 0.7, and each server runs at concurrency $C{=}8$ unless stated otherwise. The Qwen3.5-4B runs behind Figures 5, 6, 7, and 8(b) additionally end a request whose output repeats the same token block at least ten times (and at least 256 tokens), identically for every method; this bounds the degenerate repetition loops of the smaller target.

E AURORA-DFLASH BASELINE

Aurora (Wang et al., 2026a) couples an SGLang inference server with an asynchronous trainer on separate GPUs and supports EAGLE-3 drafters. We re-implement its pipeline for DFlash while following its recipe: the serving GPU streams completed requests to a trainer on a dedicated GPU; the trainer recomputes the target's context features and top-1024 next-token distribution, fine-tunes all drafter parameters (the embedding and LM head shared with the target stay frozen) with the forward KL divergence under the block-parallel teacher forcing used at inference, and hot-swaps the new weights into the server every 100 requests (about 1 ms on the serving path). Optimization follows Aurora's fine-tuning setting (AdamW, no weight decay, gradient clipping 0.5, global batch size 8, BF16 compute with FP32 master weights) except for the learning rate: Aurora's 10^{-5} learns little on our streams (news writing at $C{=}8$: $+11\%$ acceptance length, against $+29\%$ at 10^{-4}), so we report 10^{-4}, the same order as ours. We verified that the trainer's drafter reproduces the serving drafter's outputs on identical inputs (99% top-1 agreement; the remainder are BF16 near-ties).

Trainer utilization. To confirm that we do not weaken this baseline, Table 4 reports how busy its trainer GPU is. In 7 of the 8 runs, the trainer is busy 99% of the time and still cannot keep up with the served traffic: it trains on only 12–94% of the requests, and on long agent contexts (OSWorld) on at most 37%. Only in news writing at $C{=}2$ does the trainer wait for new requests (40% utilization), and there it trains on all of them. The decoupled design thus uses its trainer GPU fully, but spends it on training rather than serving.

Table 4: **Trainer GPU of Aurora-DFlash.** Utilization of the trainer GPU, time spent waiting for requests, and fraction of the served requests that the trainer trains on.

Stream	C	Utilization	Waiting time	Requests trained on
News writing	2	40%	944 s (≈60%)	100%
News writing	8 (LR 10^{-4} / 10^{-5})	99%	0	87% / 94%
News writing	64	99%	0	39%
OSWorld	2	99%	0	37%
OSWorld	8 (LR 10^{-4} / 10^{-5})	99%	0	22% / 22%
OSWorld	64	99%	0	12%

F ADDITIONAL RESULTS

Table 5: **Latency** (Qwen3.5-4B + DFlash, second half of each stream, i.e., after adaptation). Request latency (p50 / p99, seconds) and median per-request speed (output tokens/s) of the frozen drafter and AdaSpec at concurrency C=1, 8, and 64 (the runs of Figure 5). OSWorld requests are agent steps with thinking.

		C=1		C=8		C=64	
Stream	Metric	Frozen	AdaSpec	Frozen	AdaSpec	Frozen	AdaSpec
News writing	latency (s)	1.9 / 3.3	1.6 / 2.5	2.7 / 4.4	2.2 / 3.9	9.8 / 17.0	8.4 / 14.4
	speed (tok/s)	352	427	248	292	68	79
OSWorld	latency (s)	0.2 / 1.0	0.2 / 1.0	0.5 / 2.6	0.5 / 2.9	1.8 / 10.5	1.7 / 9.7
	speed (tok/s)	348	397	151	165	38	42

Table 6: **What to train** (C=8; with the gate unless noted). Acceptance-length and throughput gain (%) over the frozen drafter; throughput is the per-request serving rate on the same requests (as in Figure 7). The 121M MTP head has its own frozen baseline and trains without the gate. 27B news writing: first 2000 requests.

		News writing		OSWorld	
Model	Trained parameters	$\Delta\tau$	ΔTPS	$\Delta\tau$	ΔTPS
Qwen3.5-4B + DFlash	81M	+11.5	+5.7	+12.5	+3.6
	121M (MTP head, always)	+6.1	-12.1	+1.3	-9.1
	142M (default)	+21.8	+14.5	+16.0	+5.7
	425M	+29.8	+14.4	+16.5	-2.7
Qwen3.8-27B + DFlash2	427M	+34.4	+26.2	+3.6	-2.3
	701M (default)	+31.4	+22.7	+4.0	-1.4

Table 7: Per-user results (SWE-chat developers Dev-1–5, WildChat chat users Chat-1–4). $\Delta\tau$ / ΔTPS in %.

	Qwen3.8-27B + DFlash2		Qwen3.5-4B + DFlash	
User	$\Delta\tau$	ΔTPS	$\Delta\tau$	ΔTPS
Dev-1	+4.3	-2.4	+11.9	+6.0
Dev-2	+4.2	+1.5	+13.1	+5.3
Dev-3	+5.1	+2.5	+8.3	-3.3
Dev-4	+5.4	+4.3	+9.6	-0.3
Dev-5	+3.0	-2.9	+6.6	+3.1
Chat-1	+5.1	+1.7	+5.6	+2.5
Chat-2	+5.6	-0.7	+5.9	-0.1
Chat-3	+2.8	-1.6	+5.6	+2.0
Chat-4	+0.1	-1.6	+1.2	+1.0

Table 8: **Per-task results, Qwen3.8-27B + DFlash2** (TP2 per arm); same protocol as Table 1.

	Acceptance length τ			Throughput (tok/s)		
Dataset	Frozen	AdaSpec	Δ%	Frozen	AdaSpec	Δ%
Agent						
SWE-Gym (OpenHands)	4.18	4.68	+12.2	1099	1152	+4.8
OpenThoughts-Agent (Terminus-2)	4.62	4.83	+4.4	1617	1641	+1.5
APIGen-MT (τ-bench)	4.50	4.79	+6.5	1352	1387	+2.6
Web						
NNetNav-Live	3.36	3.57	+6.3	1223	1246	+1.9
OSWorld	3.94	4.09	+4.0	1163	1185	+1.9
Low-resource languages						
Translation en→sw	1.41	2.32	+64.9	490	677	+38.2
Translation en→zu	1.35	2.08	+54.2	486	646	+32.9
Translation en→hi	1.81	2.23	+23.0	606	668	+10.4
News writing sw	1.66	2.22	+33.8	682	853	+25.0
News summarization sw	2.19	2.72	+24.6	637	700	+9.9
Code						
SWE-chat, 5 developers	3.88	4.05	+4.4	810	817	+0.9
Chat						
WildChat, 4 users	3.27	3.39	+3.8	1281	1275	-0.5
Reasoning						
Reasoning Gym	5.71	5.99	+4.8	2079	2106	+1.3
Function calling						
BFCL v3 + ToolACE	5.44	5.90	+8.5	790	793	+0.4
Other workloads (not in Table 1)						
XL-Sum ja (summarization)	2.68	2.79	+4.0	941	920	-2.2
MeetingBank (meeting summarization)	3.34	3.48	+4.2	695	688	-1.0
VCSUM zh (meeting summarization)	2.88	2.94	+2.0	1070	1052	-1.7
Spider + WikiSQL (text-to-SQL)	6.45	7.03	+9.0	732	741	+1.2
Average (18 rows)			**+15.2**			**+7.1**

Table 9: **Per-task results, Qwen3.5-4B + DFlash** (one GPU per arm); same protocol as Table 1.

	Acceptance length τ			Throughput (tok/s)		
Dataset	Frozen	AdaSpec	Δ%	Frozen	AdaSpec	Δ%
Agent						
SWE-Gym (OpenHands)	3.12	5.39	+72.4	1496	1820	+21.6
OpenThoughts-Agent (Terminus-2)	3.96	4.56	+14.9	2415	2446	+1.3
APIGen-MT (τ-bench)	3.97	4.32	+8.8	1840	1856	+0.9
Web						
NNetNav-Live	3.16	3.45	+9.1	2190	2179	-0.5
OSWorld	3.37	3.86	+14.4	1216	1314	+8.1
Low-resource languages						
Translation en→sw	1.66	2.28	+37.3	1154	1352	+17.1
Translation en→zu	1.68	2.15	+28.2	1202	1379	+14.7
Translation en→hi	1.35	1.73	+28.1	937	1042	+11.3
News writing sw	2.25	2.82	+25.4	2032	2359	+16.1
News summarization sw	1.97	2.41	+22.2	999	1023	+2.4
Code						
SWE-chat, 5 developers	3.36	3.69	+9.9	1175	1198	+2.0
Chat						
WildChat, 4 users	3.23	3.39	+4.7	2807	2841	+1.2
Reasoning						
Reasoning Gym	4.82	5.16	+7.0	3854	3975	+3.1
Function calling						
BFCL v3 + ToolACE	4.50	4.91	+9.1	1002	1023	+2.1
Other workloads (not in Table 1)						
AgentNet (desktop agent)	3.55	3.76	+6.0	2191	2225	+1.6
SWE-chat, multi-developer stream	3.35	3.74	+11.6	1484	1443	-2.8
XL-Sum ja (summarization)	1.98	2.16	+9.4	1233	1220	-1.0
MeetingBank (meeting summarization)	2.52	2.69	+6.8	914	919	+0.5
VCSUM zh (meeting summarization)	2.22	2.30	+3.3	1748	1763	+0.9
Spider + WikiSQL (text-to-SQL)	5.25	5.93	+13.0	742	748	+0.8
Average (20 rows)			**+17.1**			**+5.1**

Table 10: **Throughput** (output tokens/s) of the same runs as Table 1. *Gain*: whole stream, end to end. *Gain, 2nd half*: the second half of each stream, after most of the adaptation (per-request serving rate on the same requests: output tokens over summed latency divided by the concurrency).

Method	**Agent** SWE-Gym, OH-Agent, τ-bench	**Web** NNetNav-Live, OSWorld	**Low-res.** WMT24++, XL-Sum	**Code** SWE-chat	**Chat** WildChat	**Reason.** Reasoning Gym	**Func.** BFCL, ToolACE	**Avg.**
Qwen3.8-27B + DFlash2								
Frozen	1356	1193	580	810	1281	2079	790	1155
AdaSpec	1393	1216	709	817	1275	2106	793	1187
Gain (%)	**+3.0**	**+1.9**	**+23.3**	**+0.6**	**-0.5**	**+1.3**	**+0.4**	**+4.3**
Gain, 2nd half	**+4.0**	**+0.2**	**+35.4**	**+3.0**	**+0.7**	**+3.6**	**+1.1**	**+6.9**
Qwen3.5-4B + DFlash								
Frozen	1917	1703	1265	1175	2807	3854	1002	1960
AdaSpec	2041	1747	1431	1198	2841	3975	1023	2036
Gain (%)	**+7.9**	**+3.8**	**+12.3**	**+2.2**	**+1.3**	**+3.1**	**+2.1**	**+4.7**
Gain, 2nd half	**+6.9**	**+4.2**	**+15.0**	**+0.9**	**+2.5**	**+5.2**	**+3.1**	**+5.4**