SPECRELAY: AMORTIZE OR AVOID THE ROUND TRIP IN EDGE–CLOUD SPECULATIVE DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

Edge–cloud speculative decoding moves autoregressive drafting to an edge device while a larger cloud model verifies draft tokens in parallel, so that edge drafting and cloud verification overlap under a synchronous draft–verify schedule. Beyond low-latency local networks, however, the next verification request of that schedule can be sent only after the previous reply arrives, so every round still waits out a full round trip, and throughput falls as the network RTT grows. To recover this loss, we present SpecRelay, an edge–cloud LLM inference system that addresses this cost at two levels: it amortizes round trips when cloud verification is necessary and avoids them when local reasoning suffices. For cloud-bound queries, an asynchronous verification pipeline keeps fixed-budget draft trees in flight while earlier replies are still outstanding, and the cloud rejects stale requests before the target-model forward pass (the *asynchronous schedule*). At the query level, an RTT-conditioned router (the *Latency-Aware Router*) estimates quality and completion latency for three on-device modes (direct generation, Chain-of-Draft, and full reasoning) and cloud speculative decoding, selecting among them under a configurable accuracy–latency objective without adding latency during inference, since the router runs concurrently with the draft-model prefill. The *asynchronous schedule* alone improves decoding throughput by up to 50% over a reproduced synchronous baseline across multiple benchmarks. At matched target-model accuracy and 250 ms of added round-trip latency, the full system with the router delivers $1.95\times$ and $1.55\times$ the throughput of the same baseline on SPROUT and OlympiadBench, respectively, while sending only 47% and 39% of the queries to the cloud.

1 INTRODUCTION

Edge–cloud speculative decoding splits generation between a small draft model on the edge device and a larger target model on the cloud server Leviathan et al. (2023); Chen et al. (2023). In each verification round, the edge proposes candidate tokens organized as a sequence or, more generally, a tree. The cloud server checks these candidates in parallel and returns the longest accepted path; unmatched speculative work is discarded. Draft length (the depth of the draft tree sent per round trip) therefore creates a tradeoff between round-trip frequency and wasted speculative computation. Short drafts limit the amount of work discarded after a mismatch, but they require frequent edge–cloud round trips. Longer drafts can amortize a round trip over more tokens if they are accepted, but an early mismatch can discard a larger portion of the speculative work.

In an edge–cloud setup, the impact of this tradeoff depends on network latency. SpecEdge demonstrated this design in a low-latency setting, with a round-trip time (RTT) of roughly 14 ms Park et al. (2025). In contrast, cellular, metropolitan Wi-Fi, and low-Earth-orbit satellite paths commonly incur RTTs of tens of milliseconds, with tail latencies of hundreds of milliseconds Charyyev et al. (2020); Xu et al. (2020); Narayanan et al. (2021); Michel et al. (2022); Mohan et al. (2024). In a synchronous schedule, the edge must wait for the cloud reply before it can safely advance, so every verification round adds another RTT. Slow and variable links therefore raise two questions: how to hide a longer RTT when the cloud is used, and when to avoid the cloud altogether.

We present SpecRelay, an edge–cloud LLM inference system that addresses both questions: *amortize* the round trip when cloud verification is necessary and *avoid* it when local reasoning suffices.

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.

For cloud-bound queries, an anchor-hedged asynchronous verification pipeline keeps multiple fixed-budget draft trees in flight. When the edge sends a tree, it tags the tree with the committed prefix it was drafted from, and the server verifies a tree only if that prefix is still current, discarding stale trees before any target-model execution. On each reply, the edge adopts the server's verdict and updates its draft trees to match: draft tokens that no longer align with the accepted path are demoted, and drafting continues from those that do (*lineage-preserving rebuild*). This overlaps successive verification rounds without modifying either language model. Between successive trees, the edge additionally hedges across the K most likely continuation anchors and grows a K-times wider tree from them, which raises the probability that the next tree is accepted; because the server forwards only the subtree under the correct anchor, its verification cost is unchanged. We evaluate this continuation hedging under both the synchronous and the asynchronous schedule.

At query granularity, an RTT-conditioned router selects among on-device direct generation, Chain-of-Draft Xu et al. (2025), full reasoning, and cloud speculative decoding. A frozen query encoder and nonparametric estimators predict per-mode quality and output length; measured hardware profiles and the current RTT estimate convert these predictions into completion-latency estimates. Because RTT affects only the cloud path, the same query can use remote verification on a fast link and additional local test-time compute on a slow one. One fitted router covers the whole RTT range without retraining. This extends content-based routing Ong et al. (2025); Chen et al. (2024); Ding et al. (2024); Somerstep et al. (2025) with runtime network state and a multi-level local reasoning action space, while leaving the language models unchanged.

In summary, our contributions are as follows:

- **Asynchronous cloud verification.** We introduce an asynchronous, prefix-gated protocol that overlaps successive verification rounds and prevents stale speculative requests from consuming a target-model forward pass. It needs no RTT-specific tuning—one configuration serves every network condition—and yields higher throughput.
- **Continuation hedging with lineage-preserving rebuild.** At each boundary between successive draft trees, the edge hedges across the K most likely continuation anchors at no extra verification cost, and lineage-preserving rebuild continues drafting from the surviving lineage.
- **RTT-conditioned test-time-compute routing.** A lightweight per-query router that selects among three on-device reasoning budgets and cloud speculative decoding by predicted per-mode quality and latency at the current RTT estimate: the lightest mode predicted to answer correctly. Its 0.6B encoder runs concurrently with the draft model's prefill and adds near-zero (<4 ms) latency.

2 Preliminaries and Related Work

Speculative decoding. Speculative decoding Leviathan et al. (2023); Chen et al. (2023) accelerates auto-regressive generation by using a fast draft model to propose multiple tokens that a slower target model verifies in parallel. Subsequent work has improved draft quality via self-speculation Cai et al. (2024); Zhang et al. (2024) and tree-structured drafts Miao et al. (2024); Li et al. (2024). These advances assume colocated draft and target models, so the network is not a first-order concern.

In edge–cloud speculative decoding, a draft model M_d runs on the edge and a target model M_t runs in the cloud Park et al. (2025): each round, the edge drafts a tree and sends it as a *verification request*; M_t verifies it in one forward pass and *replies* with the longest accepted path and one bonus token. We call this mode SPEC-CLOUD. The approximate latency per accepted token is

$$\frac{(k \cdot t_d + t_v + \tau)\, o}{\alpha\, k}, \tag{1}$$

where k is the draft length, t_d the per-token edge draft time, t_v the cloud verification time, τ the network round trip, α the per-round acceptance rate (so αk tokens are accepted per round), and $o \in [\frac{1}{3}, 1]$ the degree to which edge drafting, the round trip, and cloud verification overlap (1 strictly in sequence, $\frac{1}{3}$ fully overlapped). At small τ, the cost is compute-bound; as τ grows, the schedule becomes network-bound.

Speculative decoding systems. Existing systems act on individual terms of Eq. (1). Tree-structured drafts Miao et al. (2024); Li et al. (2024) raise α by verifying many continuations in

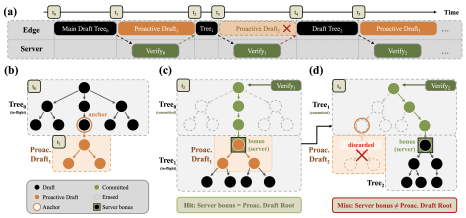

Figure 1: Baseline SpecEdge Park et al. (2025) draft tree. **(a)** edge and server timelines; **(b)** draft tree (edge) Tree_0; **(c-d)** Hit(c)/Miss(d): tree after the target (M_t) reply.

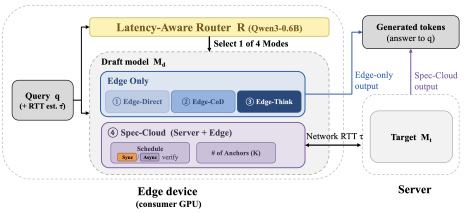

Figure 2: SpecRelay overview.

one pass. Other systems relax the lockstep draft–verify schedule in different settings: PipeSpec pipelines a hierarchy of progressively larger models with cascading rollback over a single draft chain McDanel et al. (2025); FlowSpec prunes and expands a draft tree across a distributed model pipeline Liu et al. (2025b); and PicoSpec overlaps edge drafting, communication, and cloud verification in a training-free edge–cloud design that drafts a short chain (four tokens) per round Zhang et al. (2026b). Of these, only FlowSpec drafts a tree, and none targets long round trips with tree drafts (Appendix D surveys further runahead and edge–cloud systems). SpecEdge Park et al. (2025), our baseline (Figure 1), drafts such a tree on the edge and lowers o with *proactive drafting*: while the cloud verifies the current tree (Tree_0 in Figure 1b, sent at t_1), the edge drafts the next one below its most likely leaf—the *anchor* from which the next tree continues—from a single guessed bonus token; if the guess matches the reply (Figure 1c, t_2), the proactive subtree is reused, and otherwise (Figure 1d, t_4) it is discarded and rebuilt from the verified prefix. This can hide part of the draft time $k\,t_d$ under the wait for the reply, but it rests on a single guess, and each round still waits $t_v + \tau$ in full (Figure 3a, top): at large τ, the drafting time discarded on a miss grows with the RTT, and o cannot fall below $\frac{1}{2}$. These limits are what SpecRelay addresses (§3–§4).

Routing and on-device reasoning modes. LLM routers decide, per query, which model or execution mode serves it, trading predicted accuracy against cost Ong et al. (2025); Chen et al. (2024); Ding et al. (2024); Somerstep et al. (2025): RouteLLM learns the choice between a strong and a weak model from preference data Ong et al. (2025); FrugalGPT cascades through models until an answer is accepted Chen et al. (2024); and CARROT estimates each model's accuracy and output length with nonparametric regressors Somerstep et al. (2025). Subsequent work widens the choice to test-time compute: BEST-Route selects a model together with the number of sampled responses Ding et al. (2025), Route-to-Reason jointly selects a model and a reasoning strategy Pan et al. (2026), and Thinkless trains a reasoning model to decide when to think at length Fang et al. (2025). Device–cloud systems split the same decision across the network: Hybrid LLM routes queries between a small and a large model Ding et al. (2024), and Minions has an on-device model collaborate with a cloud model on one task Narayan et al. (2025). Of these, only device–edge routers include a communication cost Bao et al. (2025), and none weighs the round trip τ of Eq. (1) against on-device reasoning budgets. SpecRelay conditions on both the query and the current RTT estimate, sending the same query to the cloud on a fast link but serving it on device when round-trip delay is high. Its on-device modes span reasoning budgets from direct generation to Chain-of-Draft Xu et al. (2025) and full reasoning Wei et al. (2022); Snell et al. (2025)(§5).

3 System Overview

Figure 2 shows the overall SpecRelay framework. The edge hosts a draft model M_d and router R, while the cloud hosts a larger target model M_t. Given query q and the current RTT estimate $\hat{\tau}$, R selects one of four modes: Edge-Direct, Edge-CoD, Edge-Think, or Spec-Cloud. The first three complete generation locally, trading more tokens for higher accuracy: Edge-Direct is cheapest; Edge-CoD is a concise middle ground; and Edge-Think uses full reasoning for higher accuracy on hard queries at the cost of verbose decoding. In Spec-Cloud, M_d drafts fixed-budget trees on the edge and M_t verifies them in the cloud. Neither model is modified.

Compared with SpecEdge, Spec-Cloud exposes two knobs (Figure 2): the verification schedule and the number of anchors K. The schedule is synchronous, as in SpecEdge, or asynchronous, in

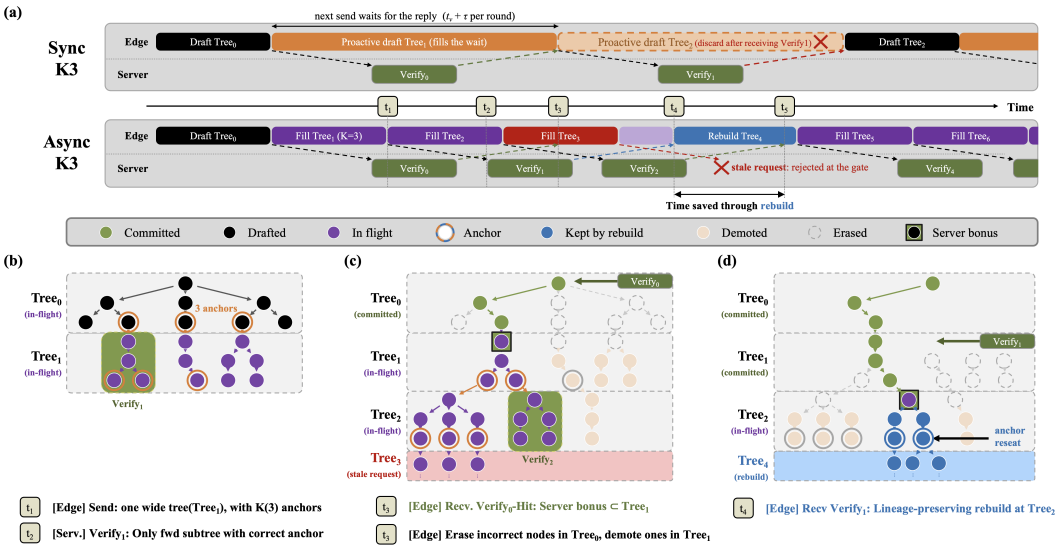

Figure 3: Anchor-hedged asynchronous verification pipeline. **(a)** Edge and server timelines under the synchronous and asynchronous schedules. **(b–d)** Edge KV state of Async-K3: **(b)** Tree_1 drafted under three anchors of Tree_0 and in flight; **(c)** reply to Tree_0 (Verify_0) received; **(d)** reply to Tree_1 (Verify_1) received, lineage-preserving rebuild at Tree_2.

which case the edge does not wait for the server's reply before drafting and sending the next tree: successive fixed-budget trees are issued back to back, so several verification requests are in flight at once (§4.1). At each boundary between trees, the next tree is grown from K *anchors*—the K most likely continuations of the tree still being verified—rather than from one guess, so that a subsequent tree is already in place for whichever continuation the reply confirms, and drafting continues from that lineage (*lineage-preserving rebuild*, §4.2). SpecRelay's full system (**R** in Table 1) runs on the asynchronous schedule with $K{=}3$; §6.2 evaluates the other combinations as ablations.

4 Cloud-Path Scheduling

This section presents the two mechanisms behind the SPEC-CLOUD knobs introduced in §3, which address the limits of proactive drafting noted in §2: the *asynchronous verification pipeline*, which overlaps the verification and round trips of successive requests so that both t_v and τ of a request are hidden behind the others in flight (lower o in Eq. (1); §4.1), and *continuation hedging*, which replaces the single guess at each boundary between trees by K anchors (higher α at the boundary; §4.2). Every combination of the two keeps the target model's verification rule and a fixed server-side tree budget. Together, the two mechanisms enable *lineage-preserving rebuild*: with several hedged trees in flight, a verdict is matched against them and drafting continues from the lineage it confirms instead of restarting from the verified prefix (§4.2).

4.1 Asynchronous Verification Pipeline

Under a synchronous schedule the edge issues the next verification request only after the previous reply has arrived, so the request rate is bounded by the round-trip time. Under the asynchronous schedule the edge continues drafting from the leaves it expects to be accepted and issues each fixed-budget tree as soon as it is complete, so several requests are in flight at any time (Figure 3a). The number of requests in flight is determined by the round-trip time and the drafting rate rather than by a parameter, and a longer round trip is covered by additional trees rather than by deeper ones. A single configuration (draft depth 5, $K{=}3$; K sweep in Appendix B.6) is therefore used for all RTTs, edge GPUs and model pairs in §6, whereas a synchronous draft has to be re-tuned to the wait in each setting (§6.2, Appendix B.3).

What crosses the network. As in SpecEdge, no KV cache, hidden state or logit is exchanged; each side keeps its own model's cache, and consistency rests on token IDs and positions. A *request*

carries the draft tree, its *base* (the committed position last seen by the edge and the tokens assumed accepted beyond it) and, for a query's first request, the prompt. A *reply* carries the accepted path, the bonus token and the committed position, from which the edge resynchronizes on rejection.

The j-th request of a session is $r_j = (A_j, T_j)$: its *base* A_j, the set of *anchors*—leaves of the previous tree, transmitted as the token paths from the committed position last seen by the edge to each anchor—and its draft tree T_j, one fixed-budget draft subtree rooted at each anchor. The edge does not guess the target's bonus token; the first level of each subtree holds the anchor's drafted candidates for that position. Each anchor is a bet, placed before the verdict of r_{j-1} arrives, that the accepted path will end at that anchor; T_j is usable only if one of these bets turns out to be correct. Since a tree is drafted before the verdict that fixes its place in the sequence, its coordinates—token positions and KV-cache slots—are relative to its anchor; the server rebases the matched subtree onto its committed position, and the edge rebases surviving drafts in the same way when it applies a verdict (§4.2), so a hit costs neither side any redrafting or renumbering.

On the server side, requests of a session are processed in the order received. When r_j reaches the front of the queue, let ℓ^*_{j-1} and z^*_{j-1} be the accepted leaf and the bonus token of the verdict on r_{j-1}. Before executing the target model, the server applies the *gate*, which classifies r_j as *valid* or *stale*:

$$\mathrm{valid}(r_j) \iff \mathrm{valid}(r_{j-1}) \ \wedge \ \ell^*_{j-1} \in A_j \ \wedge \ z^*_{j-1} \in T_j;$$

that is, r_j is valid if r_{j-1} was valid, one of its anchor bets was right, and the committed bonus token is in T_j (as a child of that anchor). The server checks this by comparing the declared anchor paths position by position with its committed tokens. A stale request is rejected without a target-model forward or state change (Figure 3a, t_4–t_5), and subsequent requests are rejected in turn until a rebuilt tree resets the chain. A valid request is verified only under its matching anchor (Figure 3b, Tree_1); the server commits and returns the accepted path and the bonus token as the reply.

The edge also applies replies in order. On each reply, it adopts the server's verdict and updates its draft trees to match: drafts that continue the committed prefix stay valid, together with the in-flight requests built on them, and the rest are dropped from the active tree (§4.2 describes which drafts survive and where the anchors are re-seated); replies to requests issued before a rebuild are ignored.

4.2 Continuation Hedging at Tree Boundaries

With a single anchor, a miss at a tree boundary invalidates every request in flight, and the longer the round trip, the more in-flight requests depend on each anchor. We therefore hedge across up to K anchors: in Figure 3b, Tree_1 is drafted under three anchors of Tree_0. The anchors are placed at the forks where the accepted path is most likely to leave the draft's most probable path, judged by the local margin at the fork rather than by cumulative probability, which favors the forks nearest the leaf; a subtree is grown under each anchor with an equal share of the beams, and the verdict keeps the surviving subtree (Appendix B.4 gives the rule).

The K-times wider tree does not burden the link: as in SpecEdge, a request carries only token IDs and tree structure (four integer vectors) plus the base fields the gate needs—about 3 KB for the $\approx$100-node K=3 tree, against $\approx$1 KB for a K=1 tree and a $\sim$0.1 KB reply—so the cloud path remains latency-bound rather than bandwidth-bound. The wider draft forward costs the edge about 10% per forward; Appendix B.5 shows that the fewer rejected forwards outweigh it.

Lineage-preserving rebuild. A verdict *hits* when the accepted path ends at one of the anchors and the bonus token is among that anchor's drafted children; drafting then simply continues under that anchor and the other hedges are demoted (Figure 3c, Verify_0). A verdict may also leave the predicted path before its tip. In Figure 3d the verdict of Verify_1 arrives while Tree_2 and Tree_3 are in flight. The edge keeps the draft tokens that continue the accepted path together with the server's bonus token and demotes the rest, which leaves Tree_3 stale; it then looks back, over at most a few levels, for the deepest level of the surviving tree that still offers K distinct continuations, re-seats the anchors there—here at the surviving leaves of Tree_2—and resumes drafting from them as Tree_4, so that the hedge is re-established rather than continued from a single survivor. Only when nothing survives does the edge draft a fresh tree from the bonus token.

Synchronous variant. The same hedge applies to a synchronous schedule—K proactive subtrees grown during the round trip, the matching one sent after the reply—which §6.2 uses (**S3**) to separate the contribution of hedging from that of asynchronous verification.

Correctness. With in-order processing per session and non-mutating rejection of stale requests, the request that advances the committed prefix is always verified from exactly that prefix, so its verdict is what serial speculative decoding would produce; hedging, the rebuild and the trees drafted ahead change only what the edge sends, never what the server commits. Asynchronous verification with hedging therefore yields output of the same quality as the synchronous schedule.

5 RTT-CONDITIONED ROUTING

Given query q and current RTT estimate $\hat{\tau}$, the router selects among EDGE-DIRECT, EDGE-COD, EDGE-THINK, and SPEC-CLOUD. The first three use the same edge model under different reasoning policies; SPEC-CLOUD uses the anchor-hedged asynchronous pipeline (**A3**; §4). The decision therefore jointly chooses execution placement and local test-time compute.

5.1 PER-MODE ESTIMATION

A frozen encoder maps q to $x = E(q)$. For each mode m, cosine-similarity k-nearest-neighbor regressors estimate correctness $\hat{A}_m(q)$ and decode length $\hat{O}_m(q)$:

$$\hat{A}_m(q) = \frac{1}{k_A} \sum_{i \in \mathcal{N}_{k_A}(x)} y_{m,i}, \qquad (2) \qquad\qquad \hat{O}_m(q) = \frac{1}{k_O} \sum_{i \in \mathcal{N}_{k_O}(x)} O_{m,i}, \qquad (3)$$

where $y_{m,i} \in [0,1]$ is the correctness label and $O_{m,i}$ is the realized decode-token count of mode m on fitting query i. The neighborhood sizes k_A and k_O are selected independently on validation data.

Let $P(q)$ be the known prompt length. We convert the predicted output length into request-completion latency $\hat{L}_m(q, \hat{\tau})$ using measured hardware profiles F_m for the edge modes and F_{sc} for the cloud path:

$$\hat{L}_m(q, \hat{\tau}) = \begin{cases} F_m\Big(P(q), \hat{O}_m(q)\Big), & m \in \mathcal{M}_{\mathrm{edge}}, \\ F_{\mathrm{sc}}\Big(P(q), \hat{O}_m(q), \hat{\tau}\Big), & m = \mathrm{sc}, \end{cases} \qquad (4)$$

where $\mathcal{M}_{\mathrm{edge}}$ contains the three local modes and sc denotes SPEC-CLOUD. Local profiles are linear in prompt and output length, $P(q)/\pi_m + \hat{O}_m(q)/\delta_m$, with per-mode prefill throughput π_m and decode throughput δ_m measured on the deployment. The cloud profile additionally includes both model prefills, communication, verification, pipeline fill and drain, and reconciliation. We obtain these profiles from calibration traces and interpolate across profiled RTTs, rather than approximating the asynchronous schedule with a constant overlap factor.

5.2 ROUTING OBJECTIVE

For trade-off parameter $\lambda \in [0,1]$, the router selects

$$g_\lambda(q, \hat{\tau}) = \arg\min_{m \in \mathcal{M}} (1-\lambda)\big(1 - \hat{A}_m(q)\big) + \lambda \frac{\hat{L}_m(q, \hat{\tau})}{L_{\mathrm{ref}}}, \qquad (5)$$

where L_{ref} is the validation-set mean completion latency. RTT enters only through F_{sc}, so the same query may use cloud verification on a fast link and local reasoning on a slow one.

Operating points are selected without test labels. For each target accuracy tolerance ϵ, we choose λ on validation data to minimize mean completion latency over a prespecified RTT grid, subject to accuracy remaining within ϵ of the cloud target. The matched-accuracy operating points use ϵ=0; varying ϵ traces a deployable accuracy–latency frontier. We then freeze λ, the neighborhood sizes, latency profiles, and normalization constants before the test sweep. Any test-set sweep of λ is reported only as an oracle frontier, not as a deployable operating point.

5.3 CONSTRUCTION AND OVERHEAD

We partition routing data into disjoint fitting, validation, and test sets. On the fitting split, we execute all four modes, record output lengths and completion traces, assign correctness labels with a single

Hardware		**Network** (edge↔cloud)		**Models**	
Edge device	RTX 4090	Base round trip†	12.5 ms	Draft (edge)	Qwen3-1.7B (FP16)
Cloud server	A100-40GB	Added latency swept	0–500 ms	Target (cloud)	Qwen3-14B (FP16)
		Effective RTT	measured per row	Router modes	EDGE-DIRECT, EDGE-COD, EDGE-THINK, SPEC-CLOUD

Variants

Up: released SpecEdge	**Ref**: our reproduction of **Up** (sync., $K=1$)	**S1**: **Ref** + edge draft optimizations (cuDNN attention, KV-length buckets)
S3: **S1** + hedging ($K=3$)	**A1**: **S1** + asynchronous verification	**A3**: **S3** + asynchronous verification
R: **A3** + router (full system)		

†Mean network path + gRPC handling on both ends, no added latency; effective RTT is measured the same way (Table 2).

Table 1: Experimental setup: hardware, models and the variants that appear in the experiments.

LLM judge (Qwen3.5-27B), embed the queries, and build the retrieval index. Validation data select k_A, k_O, the latency-profile parameters, L_{ref}, and λ. For out-of-distribution evaluation, the complete policy is applied without refitting or recalibration.

At runtime, the encoder forward (a Qwen3-Embedding-0.6B model; larger routers add latency without routing gains, Appendix C.3) overlaps the edge model's prompt prefill. Because this cost is common to all modes, it is omitted from Eq. (5); any non-overlapped residual counts in the reported end-to-end latency. The router requires no gradient training and modifies neither model.

6 EXPERIMENTS

The evaluation proceeds at two levels. §6.2 isolates the SPEC-CLOUD path and attributes its throughput gain across RTT to the three terms of Eq. (1): the draft time t_d (edge-draft optimizations), the acceptance α (continuation hedging), and the overlap o (asynchronous verification). §6.3 evaluates the complete system, whose per-query routing decisions shift with the RTT, and reports throughput at matched target-model accuracy together with the resulting server compute.

6.1 EXPERIMENTAL SETUP

Table 1 summarizes the hardware, models and network setup: an RTX 4090 edge drafting with Qwen3-1.7B, an A100 cloud verifying with Qwen3-14B, and gRPC between them; Appendix B.10 repeats the sweep on an RTX 3090 edge and Appendix B.9 on two further model pairs (Llama-3.2-1B/3.1-8B and Qwen3-1.7B/32B). The router chooses among the four modes in Table 1, using per-mode latency profiles (Eq. (4)) measured on this deployment. The added RTT is injected symmetrically between send and receive, from 0 to 500 ms, on top of the $\approx$12.5 ms base round trip and the server's verification time. All systems decode greedily (temperature 0); Appendix B.8 reports a temperature-0.7 spot check, and Appendix B.1 lists the remaining run settings.

Variants. **Up** is the released SpecEdge code Park et al. (2025) and **Ref** our reproduction of it—the same synchronous $K=1$ schedule with the four correctness fixes of Appendix A—and the reference for every speedup we report. Table 1 defines the variants that appear in the results.

Datasets. SPROUT Somerstep et al. (2025) is our mixed-difficulty routing benchmark, spanning GPQA, MuSR, MMLU-Pro, MATH, OpenHermes, and RAGBench. OlympiadBench He et al. (2024) evaluates out-of-distribution transfer on unseen Olympiad-level mathematics and physics without refitting or recalibration (Appendix C.2). SpecBench Xia et al. (2024) is the speculative-decoding benchmark SpecEdge reports on.

Metrics. For SPEC-CLOUD we report decoding throughput: the output tokens of a benchmark's 32-query run divided by its client wall time. Because every SPEC-CLOUD variant emits the target model's verified output for a query (the same text up to fp16 near-ties), a throughput ratio is also a wall-time ratio. For the full system, whose modes emit outputs of different lengths, speedup is defined on wall time: the summed per-query completion latencies (Eq. 4) of the routed workload against running every query on SPEC-CLOUD at matched target-model accuracy, with the accuracy–latency frontier; §6.3 defines the protocol. Because many edge devices share one server, we also compare the server-side runtime, the verifier's GPU time per 1k output tokens (Figure 5b–d, Appendix C.5).

			SPROUT (1024-tok)							**SpecBench** (256-tok)							**OlympiadBench** (1024-tok)						
Added (ms)	Eff. RTT (ms)	$d_{\max}$ (sync)	**Up** tok/s	**Ref** tok/s	**S1**	**S3**	**A1**	**A3**	**R**	**Up** tok/s	**Ref** tok/s	**S1**	**S3**	**A1**	**A3**	**R**	**Up** tok/s	**Ref** tok/s	**S1**	**S3**	**A1**	**A3**	**R**
							× **Ref**							× **Ref**							× **Ref**		
0	12	3	38.0	51.9	1.13	1.16	1.12	1.18	**1.24**	39.5	47.5	1.06	1.10	0.95	1.02	**1.12**	39.7	65.4	1.06	1.09	1.05	**1.15**	1.13
25	38	5	30.0	44.8	1.17	1.20	1.09	1.25	**1.31**	30.0	37.8	1.10	1.10	0.97	1.12	**1.14**	32.0	58.7	1.08	1.13	1.03	**1.17**	1.14
50	64	6	23.4	38.9	1.12	1.16	1.10	1.28	**1.38**	24.0	30.9	1.07	1.09	1.02	1.21	**1.25**	26.0	51.2	1.06	1.10	1.08	**1.24**	1.22
100	115	9	18.8	29.9	1.13	1.15	1.18	1.39	**1.59**	17.4	22.2	1.09	1.09	1.11	1.33	**1.33**	21.8	41.7	1.08	1.11	1.13	**1.32**	1.30
150	169	12	15.8	24.7	1.14	1.15	1.23	1.46	**1.71**	14.7	17.2	1.10	1.11	1.22	1.42	**1.45**	19.3	35.9	1.08	1.10	1.17	**1.35**	1.28
200	218	15	14.3	21.5	1.16	1.15	1.22	1.49	**1.84**	12.8	14.5	1.10	1.12	1.22	1.45	**1.50**	17.5	31.4	1.09	1.13	1.17	**1.41**	1.37
250	272	18	12.6	19.1	1.14	1.14	1.24	1.48	**1.95**	11.1	12.6	1.10	1.11	1.25	1.44	**1.50**	16.0	27.8	1.08	1.12	1.20	1.41	**1.55**
300	321	21	12.0	17.2	1.12	1.15	1.24	1.48	**1.97**	10.4	11.4	1.09	1.10	1.22	1.42	**1.50**	14.3	25.6	1.06	1.10	1.17	1.41	**1.65**
350	379	24	11.1	15.8	1.10	1.14	1.22	1.44	**2.02**	8.6	10.3	1.08	1.10	1.22	1.43	**1.50**	13.5	23.1	1.06	1.12	1.20	1.46	**1.78**
400	430	27	9.8	14.5	1.11	1.13	1.24	1.50	**2.13**	8.5	9.6	1.06	1.07	1.20	1.39	**1.47**	12.7	21.4	1.06	1.11	1.23	1.48	**1.98**
450	482	30	9.6	13.4	1.12	1.13	1.24	1.46	**2.16**	8.1	8.9	1.07	1.07	1.18	1.37	**1.47**	12.0	19.8	1.04	1.09	1.24	1.46	**2.13**
500	535	32	8.7	12.5	1.10	1.11	1.24	1.45	**2.21**	7.2	8.3	1.06	1.06	1.18	1.36	**1.47**	11.1	18.6	1.05	1.08	1.21	1.45	**2.22**

Table 2: Decoding throughput and speedup across added RTT on the RTX 4090 edge. **Up** and **Ref** in tok/s; the other columns are speedups over **Ref**; bold = best per benchmark and RTT.

6.2 SPEC-CLOUD ABLATIONS AND THROUGHPUT ACROSS RTT

Table 2 reports, per benchmark and added latency, the decoding throughput of **Up** and **Ref** in tok/s and that of every other variant as a speedup over **Ref**. **S1** adds two edge-side optimizations to **Ref**, cuDNN attention and CUDA graphs captured at KV-length buckets, which shorten the draft forward from 15.6 to 8.9 ms on the RTX 3090 without touching verification (Appendix B.2); every variant to its right includes them. **S3** adds continuation hedging (K=3) to the synchronous schedule (the variant of §4.2), **A1** is the asynchronous schedule without hedging and **A3** with it, so **S3** and **A1** each isolate one step of **A3**.

Draft depth. The synchronous variants follow SpecEdge's schedule but not its depth rule, verification time $\approx$ drafting time + RTT, which has no solution once the RTT exceeds the verification time (from 50 ms added latency). Following the released code, they instead cap main and proactive tree alike at $d_{\max} = 3 + 0.06 \cdot \mathrm{RTT}$ (Table 2), so that proactive drafting at **Ref**'s forward time spans the round trip; from 200 ms on the 32-node budget, not the cap, bounds the depth ($\approx$14 levels), and a larger budget does not help, since it lengthens every forward, above all the server's (Appendix B.3).

Re-tuned baselines. Since **S1**'s faster forward leaves part of the wait unused under this cap, Appendix B.3 re-tunes the synchronous variants with a shallow main tree and a proactive tree filling the wait: the hit rate rises, but the proactive tree still deepens with the RTT, the outcome varies by benchmark at 0 ms, and **A3**'s lead holds from 50 ms on (Table 3). The asynchronous schedule needs no such rule: one configuration (draft depth 5; K=1 for **A1**, 3 for **A3**, within 4% of the best K at every RTT, Appendix B.6) serves every RTT, both edge GPUs and all three model pairs, because a longer round trip is absorbed by more trees in flight rather than a deeper proactive tree.

Throughput across RTT. **A3** is the fastest SPEC-CLOUD variant on all three benchmarks at every added latency $\geq$ 25 ms, up to 1.50/1.45/1.48× over **Ref** on SPROUT/SpecBench/OlympiadBench; only at 0 ms on SpecBench, with no round trip to hide, does **S3** lead (1.10× vs. 1.02×). The steps compound rather than add: hedging alone changes a synchronous schedule by at most 6% (**S3** vs. **S1**) and asynchronous verification alone gives 1.11–1.25× at $\geq$ 100 ms (**A1**), yet hedging on top of it adds a further 15–22% (**A3** vs. **A1**), because a missed anchor costs a synchronous round one proactive draft but invalidates every request in flight under the asynchronous schedule. Appendix B.5 traces the gain to the draft forwards: hedged anchors and the lineage-preserving rebuild keep more of **A3**'s trees on the accepted path, so it spends 24–27% less on rejected drafting per committed token than the re-tuned **S1** despite a 12% longer forward. Under bursty latency the asynchronous schedule's lead widens further (Appendix B.7).

6.3 FULL-SYSTEM THROUGHPUT AT MATCHED ACCURACY

The **R** column of Table 2 reports the full system, the latency-aware router on **A3**, as a speedup over **Ref** at the matched target-model accuracy operating point: at 500 ms it reaches 2.21/1.47/2.22× on SPROUT/SpecBench/OlympiadBench, a further improvement over **A3** that grows with the RTT. On SPROUT and OlympiadBench, the two benchmarks with reference answers, these speedups do not trade quality for latency: the operating points match the always-cloud target accuracy of $\sim$79.6% and 51.6%, respectively.

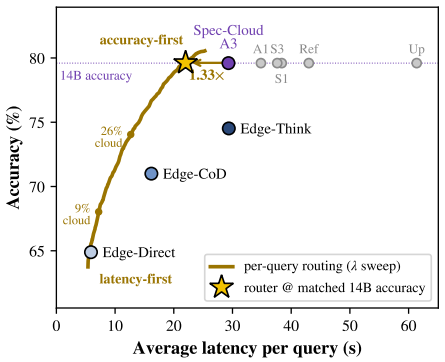

Figure 4: Accuracy vs. average latency per query on SPROUT at 300 ms added latency; curve: per-query routing over λ.

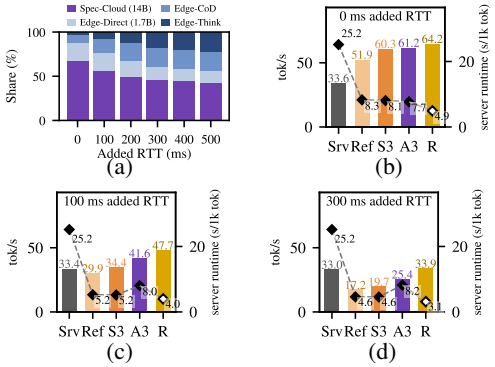

Figure 5: SPROUT, matched-accuracy operating point. **(a)** Routing shares vs. added RTT. **(b–d)** Throughput (bars) and server runtime (◆).

Figure 4 shows the accuracy–latency plane on SPROUT at 300 ms: each fixed mode is one point; the router's per-query decisions trace a curve as the latency weight λ moves from accuracy-first to latency-first (ticks: share of queries sent to the cloud); the star, the operating point matching the always-cloud 14B accuracy, completes the workload $1.33\times$ faster than **A3** alone. The other SPEC-CLOUD variants appear in grey. The max-accuracy operating points (Appendix C.4) are complementary: on SPROUT at 300 ms the accuracy-first end of the curve reaches 80.5%, 0.9 points above the always-cloud target, at a latency still below **A3**'s, so routing can improve accuracy and wall time at once relative to running every query on SPEC-CLOUD.

Routing adaptation across RTT. With the test set and the operating point held fixed, only the network latency varies. Figure 5a shows the routing distribution on SPROUT: at zero added latency the cloud path is cheap and receives 68.2% of the queries; as the added latency rises to 500 ms its share falls to 43.3%, while the combined share of EDGE-COD and EDGE-THINK rises from 12.4% to 44.0%, with accuracy at the always-14B 79.6% throughout. The full-system gain therefore grows with the RTT because a growing fraction of the queries is answered by on-device reasoning as the cloud path becomes more expensive, and cloud usage falls accordingly. The distribution also differs widely between benchmarks (Figure 7) and query groups (Appendix C.1), so the decisions depend on the queries as much as on the RTT; Appendix C.2 reports out-of-distribution behavior.

Server compute and cost. In realistic deployments many edge devices share one cloud server (two-client probe: Appendix B.10), so we also report the verifier's GPU time per 1k output tokens next to decoding throughput (Figure 5b–d; all RTTs and benchmarks in Appendix C.5); for Srv, server-only speculative decoding on the A100, the draft forwards count as server time, and **R**'s server time is **A3**'s per-token server time scaled by the share of output tokens routed to the cloud, since the edge modes use none. The synchronous variants use the server less as the RTT grows, because their proactive tree lengthens with the wait and verifies become rare, but they decode slowly for the same reason. **A3**'s server time per token is nearly constant across RTT: it keeps the verifier as busy at 500 ms as at 0 ms and turns that work into faster decoding. Routing lowers server time at the same accuracy wherever queries can be answered on the edge: 0.6–$0.8\times$ **Ref** on SPROUT across the sweep and $0.08\times$ on OlympiadBench at 500 ms, whereas on SpecBench, where over 90% of the queries stay on the cloud, it remains at **A3**'s level (1.1–$1.2\times$ **Ref**). Session rental cost per token, which also charges a slow session's idle time, is reported in Appendix C.5 (Table 14).

7 CONCLUSION

SpecRelay addresses variable RTT at two levels: SPEC-CLOUD adds continuation hedging and an asynchronous, prefix-gated verification pipeline that keep the edge drafting through the round trip, and an RTT-conditioned router decides per query whether to use the cloud or answer on-device. Over 0–500 ms added latency, the asynchronous schedule raises decoding throughput by up to $1.5\times$ over the reproduced synchronous baseline with one configuration, and a fitted router delivers up to $2.2\times$ speedup at matched target-model accuracy and cuts cloud usage by up to $2.3\times$.

ETHICS STATEMENT

This work studies systems efficiency of LLM inference (throughput, server usage, and accuracy on public benchmarks). It involves no human subjects, no personal data, and no new data release; all models and datasets are publicly available. We do not foresee ethical concerns beyond those common to LLM serving.

REPRODUCIBILITY STATEMENT

The system description (§3–§5) states every mechanism at the level needed to re-implement it; the experimental protocol, hardware, models, datasets, RTT emulation, and per-RTT draft-depth settings are given in §6 and the appendix (Appendix A onward). The appendix further reports the per-dataset, per-RTT measurements behind every figure. The supplementary material contains the anonymized source code of the edge client, the verification server and the router, the run configurations of every variant, the Table 2 measurements, and the per-query router outputs behind the router figures and tables. The full result logs and the trained router will be released with the camera-ready version.

AI USE STATEMENT

We used generative AI tools (large language models) to assist with code refactoring, log analysis scripts, and editing of the manuscript text. All AI-assisted code was reviewed and tested by the authors, and all AI-assisted text was reviewed and revised by the authors. We did not use generative AI tools to generate research ideas, experimental results, or figures. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Rui Bao, Nan Xue, Yaping Sun, and Zhiyong Chen. Dynamic quality-latency aware routing for LLM inference in wireless edge-device networks. In *2025 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*, pp. 1–6, 2025. doi: 10.1109/ICCCWorkshops67136.2025.11147210.

Payel Bhattacharjee, Fengwei Tian, Meiyu Zhong, Guangyi Zhang, Osvaldo Simeone, and Ravi Tandon. Conformal sparsification for bandwidth-efficient edge–cloud speculative decoding. In *NeurIPS 2025 Workshop: AI and ML for Next-Generation Wireless Communications and Networking*, 2025. URL https://openreview.net/forum?id=gZl2x7AN1F.

Branden Butler, Sixing Yu, Arya Mazaheri, and Ali Jannesari. PipeInfer: Accelerating LLM inference using asynchronous pipelined speculation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024. ISBN 9798350352917. doi: 10.1109/SC41406.2024.00046. URL https://doi.org/10.1109/SC41406.2024.00046.

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 *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Batyr Charyyev, Engin Arslan, and Mehmet Hadi Gunes. Latency comparison of cloud datacenters and edge servers. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pp. 1–6, 2020. doi: 10.1109/GLOBECOM42002.2020.9322406.

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.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=cSimKw5p6R. Featured Certification.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations (ICLR)*, 2024.

Dujian Ding, Ankur Mallick, Shaokun Zhang, Chi Wang, Daniel Madrigal, Mirian Del Carmen Hipolito Garcia, Menglin Xia, Laks VS Lakshmanan, Qingyun Wu, and Victor Rühle. BEST-route: Adaptive LLM routing with test-time optimal compute. In *Forty-second International Conference on Machine Learning*, 2025.

Gongfan Fang, Xinyin Ma, and Xinchao Wang. Thinkless: LLM learns when to think. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=ariVQf0KZx.

Luyao Gao, Jianchun Liu, Xichong Zhang, Guoju Gao, and Yunming Liao. CoSine: Enhancing LLM serving via collaborative and decoupled speculative inference. In *IEEE INFOCOM 2026 - IEEE Conference on Computer Communications*, pp. 1–10, 2026. doi: 10.1109/INFOCOM59046.2026.11571388.

Yunhe Han, Yunqi Gao, Bing Hu, Mahdi Boloursaz Mashhadi, Yitong Duan, Pei Xiao, and Yanfeng Zhang. PipeSD: An efficient cloud-edge collaborative pipeline inference framework with speculative decoding. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=1ebAvNphi7.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. OlympiadBench: A challenging benchmark for promoting AGI with olympiad-level bilingual multimodal scientific problems. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3828–3850, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.211. URL https://aclanthology.org/2024.acl-long.211/.

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

Xiangchen Li, Dimitrios Spatharakis, Saeid Ghafouri, Jiakun Fan, Hans Vandierendonck, Deepu John, Bo Ji, and Dimitrios S. Nikolopoulos. SLED: A speculative LLM decoding framework for efficient edge serving. In *Proceedings of the Tenth ACM/IEEE Symposium on Edge Computing*, SEC '25, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400722387. doi: 10.1145/3769102.3770608. URL https://doi.org/10.1145/3769102.3770608.

Xiangchen Li, Jiakun Fan, Qingyuan Wang, Dimitrios Spatharakis, Saeid Ghafouri, Hans Vandierendonck, Deepu John, Bo Ji, Ali R. Butt, and Dimitrios Nikolopoulos. WISP: Waste- and interference-suppressed distributed speculative LLM serving at the edge via dynamic drafting and SLO-aware batching. *Proc. ACM Meas. Anal. Comput. Syst.*, 10(2), May 2026. doi: 10.1145/3805653. URL https://doi.org/10.1145/3805653.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: speculative sampling requires rethinking feature uncertainty. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Tianyu Liu, Yun Li, Qitan Lv, Kai Liu, Jianchen Zhu, Winston Hu, and Xiao Sun. PEARL: Parallel speculative decoding with adaptive draft length. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=QOXrVMiHGK.

Xing Liu, Lizhuo Luo, Ming Tang, Chao Huang, and Xu Chen. FlowSpec: Continuous pipelined speculative decoding for efficient distributed LLM inference. *arXiv preprint arXiv:2507.02620*, 2025b.

Yuan Lyu, Bharath Irukulapati, and Jaya Prakash Champati. Speculation at a distance: Where edge-cloud speculative decoding actually pays off. *arXiv preprint arXiv:2606.25091*, 2026.

Bradley McDanel. AMUSD: Asynchronous multi-device speculative decoding for LLM acceleration. *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2024. URL https://api.semanticscholar.org/CorpusID:273532247.

Bradley McDanel, Sai Qian Zhang, Yunhai Hu, and Zining Liu. PipeSpec: Breaking stage dependencies in hierarchical LLM decoding. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 12909–12920. Association for Computational Linguistics, 2025.

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

François Michel, Martino Trevisan, Danilo Giordano, and Olivier Bonaventure. A first look at Starlink performance. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, pp. 130–136, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3517745.3561416.

Nitinder Mohan, Andrew E. Ferguson, Hendrik Cech, Rohan Bose, Prakita Rayyan Renatin, Mahesh K. Marina, and Jörg Ott. A multifaceted look at Starlink performance. In *Proceedings of the ACM Web Conference 2024*, WWW '24, pp. 2723–2734, New York, NY, USA, 2024. Association for Computing Machinery. doi: 10.1145/3589334.3645328.

Avanika Narayan, Dan Biderman, Sabri Eyuboglu, Avner May, Scott Linderman, James Zou, and Christopher Re. Cost-efficient collaboration between on-device and cloud language models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=qGDlzt3dKz.

Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, Feng Qian, and Zhi-Li Zhang. A variegated look at 5G in the wild: performance, power, and QoE implications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pp. 610–625, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383837. doi: 10.1145/3452296.3472923. URL https://doi.org/10.1145/3452296.3472923.

JIAHONG NING, Ce ZHENG, and Tingting Yang. DSSD: Efficient edge-device deployment and collaborative inference via distributed split speculative decoding. In *ICML 2025 Workshop on Machine Learning for Wireless Communication and Networks (ML4Wireless)*, 2025. URL https://openreview.net/forum?id=5vkXfhUmnn.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs from preference data. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=8sSqNntaMr.

Zhihong Pan, Kai Zhang, Yuze Zhao, and Yupeng Han. Adaptive model and strategy routing for cost-efficient LLM services. In *Proceedings of the ACM Web Conference 2026*, WWW '26, pp. 5568–5578, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400723070. doi: 10.1145/3774904.3792556. URL https://doi.org/10.1145/3774904.3792556.

Jinwoo Park, Seunggeun Cho, and Dongsu Han. Specedge: Scalable edge-assisted serving framework for interactive LLMs. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=4QVLKwgg3S.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4FWAwZtd2n.

Seamus Somerstep, Felipe Maia Polo, Allysson Flavio Melo de Oliveira, Prattyush Mangal, Mírian Silva, Onkar Bhardwaj, Mikhail Yurochkin, and Subha Maity. CARROT: A cost aware rate optimal router. In *ICLR 2025 Workshop on Foundation Models in the Wild*, 2025. URL https://openreview.net/forum?id=xEBOy2ze1U.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

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 Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 7655–7671, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.456. URL https://aclanthology.org/2024.findings-acl.456/.

Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pp. 479–494, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3405882. URL https://doi.org/10.1145/3387514.3405882.

Silei Xu, Wenhao Xie, Lingxiao Zhao, and Pengcheng He. Chain of draft: Thinking faster by writing less. *arXiv preprint arXiv:2502.18600*, 2025.

Guotao Yang, Hao Chen, Rui Guo, Xinyu Li, Liang Zheng, Sheng Chen, Yitao Hu, and Keqiu Li. Asymspec: Efficient cloud–edge speculative decoding over asymmetric networks. *arXiv preprint arXiv:2608.04974*, 2026.

Yong Yang, Ting Ting Yang, and Shao Shuai Gao. Communication-efficient speculative decoding for large language models inference. In *2025 IEEE 25th International Conference on Communication Technology (ICCT)*, pp. 1–5, 2025. doi: 10.1109/ICCT67417.2025.11374072.

Haofei Yin, Mengbai Xiao, Tinghong Li, Xiao Zhang, Dongxiao Yu, and Guanghui Zhang. Specpipe: Accelerating pipeline parallelism-based llm inference with speculative decoding. *arXiv preprint arXiv:2504.04104*, 2025.

Guangyi Zhang, Yunlong Cai, Guanding Yu, Petar Popovski, and Osvaldo Simeone. Quantize-sample-and-verify: LLM acceleration via adaptive edge-cloud speculative decoding. *IEEE Communications Letters*, 30:852–856, 2026a.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. Draft & verify: Lossless large language model acceleration via self-speculative decoding. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11263–11282, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.607. URL https://aclanthology.org/2024.acl-long.607/.

Yida Zhang, Zhiyong Gao, Shuaibing Yue, Jie Li, and Rui Wang. PicoSpec: A pipelined collaborative speculative decoding framework for efficient edge-cloud LLM inference. *arXiv preprint arXiv:2603.19133*, 2026b.

Ziyi Zhang, Ziheng Jiang, Chengquan Jiang, Menghan Yu, Size Zheng, Haibin Lin, Xin Liu, and Henry Hoffmann. SwiftSpec: Disaggregated speculative decoding and fused kernels for low-latency LLM inference. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pp. 2197–2211, New York, NY, USA, 2026c. Association for Computing Machinery. ISBN 9798400723599. doi: 10.1145/3779212.3790246. URL https://doi.org/10.1145/3779212.3790246.

A REPRODUCTION: FIXES TO THE RELEASED SPECEDGE CODE

`Up` is the released SpecEdge code, run unmodified apart from a network-delay hook in its server for the RTT sweep. On the release's own reproduction configuration (SpecBench, 60 queries, 256 output tokens, initial draft depth 4 with proactive depth 3, budget 32, RTT 0) it reproduces Table 1 of the paper (Park et al., 2025): 4.08/4.29/3.77/3.58/4.28/3.78 generated tokens per verify on the multi-turn/translation/summarization/QA/math/RAG tasks against the paper's 3.98/4.25/3.95/3.59/4.25/3.83, within 0.2 on every task.

`Ref` is the same $K{=}1$ schedule and tree budget in our codebase with four fixes to the release. **(1)** The KV-cache gather performs an in-place advanced-index copy with overlapping source and destination indices during accepted-path compaction—on *both* the edge drafter and the server verifier—which corrupts key/value entries and surfaces as repeated tokens; we fix it with a clone-before-scatter. **(2)** Under proactive depth-reduction the server's bonus token was committed to the prefix *and* simultaneously re-added as a tree candidate, so it was emitted twice. **(3)** The freshly built proactive subtree's KV was overwritten during prefix compaction; we snapshot the union of accepted and proactive slots and skip the tail-zeroing. **(4)** Result logging was non-atomic, corrupting the throughput accounting the official metric reads. These fixes, not further engineering, account for **`Ref`**'s higher throughput: the corrupted entries surface as repeated tokens that the target rejects, so with the fixes in place, on the release's own configuration and the same 60 queries, acceptance rises from 3.90 to 4.72 tokens per verify and throughput from 37.4 to 45.2 tok/s ($+21\%$); in the Table 2 grid **`Ref`** runs 20/37/65% faster than **`Up`** on SpecBench/SPROUT/OlympiadBench at RTT 0. With these fixes **`Ref`** realizes the schedule of the SpecEdge paper as intended (synchronous $K{=}1$ drafting with proactive expansion during the wait), so we adopt it, rather than **`Up`**, as the baseline against which every speedup in the paper is measured. **`Ref`** runs with a per-benchmark KV window sized to the workload (3072 tokens on SPROUT, 2048 on SpecBench and OlympiadBench) and one CUDA graph on the edge, and the server uses the same window for every variant; **`S1`** and the variants built on it allocate 4096 on the edge and dispatch each draft step to a length bucket (Appendix B.2). Its verdicts follow the target's greedy decode; outputs of different tree shapes differ only where fp16 near-ties resolve differently.

B SPEC-CLOUD: ABLATIONS AND EXTENSIONS

B.1 EXPERIMENTAL DETAILS

Runs. Every cell of Tables 2, 11, 9 and 10 is one run of the published code with one edge client: the same 32 prompts per benchmark, served one after another in a fixed order, with generation capped at 1024 output tokens on SPROUT and OlympiadBench and 256 on SpecBench (the SpecBench reproduction of Appendix A uses 60 prompts). Both models run in FP16 with Qwen3's thinking mode off and greedy decoding on both sides (temperature 0), seed 42. The draft tree is capped at 32 nodes and 32 beams, with at most 16 children per node; the synchronous variants apply the depth cap $d_{\max}$ of Table 2 to the main and the proactive tree, the asynchronous variants use depth 5 with $K{=}1$ (**`A1`**) or $K{=}3$ (**`A3`**). The server's KV window is 3072 tokens on SPROUT and 2048 on SpecBench and OlympiadBench for every variant; **`Ref`** uses the same per-benchmark window on the edge with one CUDA graph, while **`S1`** and every variant to its right allocate 4096 and capture the edge draft's CUDA graphs at KV-length buckets $512/1024/2048/4096$ and use cuDNN attention on the edge (Appendix B.2); the server keeps its full-width KV and default attention kernel throughout.

Network. Edge and cloud communicate over gRPC; the base round trip of 12.5 ms (Table 1) is the measured network path plus gRPC handling at zero added latency. Added latency is injected on the server as a one-way delay of half the added RTT on every receive and every send, so the physical link does not affect the sweep; the jitter runs of Appendix B.7 add a stochastic delay on top.

Server. The verifier serves one client with a static batch of one. As in the released SpecEdge code, the target model's prompt prefill of every prompt is computed before the run and loaded into the KV cache when the request arrives, so a run's wall time contains the draft model's prefill and the decoding only. A keepalive kernel (a background GEMM at about 50% duty, paused while a verify forward runs) holds the server's GPU clocks up between verifies; without it the GPU governor parks the clocks while a synchronous client waits out the round trip and the verify forward of the synchronous variants runs about $1.4\times$ slower at added RTT $\geq$ 50 ms, a penalty unrelated to the

schedule and specific to our single-client setting: a server shared by several clients would not idle long enough between verifies for its clocks to drop.

Measurement. Decoding throughput divides the output tokens of a run, counted by re-encoding the emitted text with the model tokenizer so that every variant is counted identically, by the run's wall time on the client (first to last client log record). Server-side runtime (Figure 5, Appendix C.5) is the sum of the verify-forward GPU times logged by the server; the server's CPU-side request handling (parsing, mask construction, acceptance, reply) is not counted and would add 22–25% for the synchronous variants and 31–35% for **A3**, whose hedged trees need more mask work.

B.2 Edge-Draft Acceleration via Length-Bucketed Graph Capture

Equation (1) treats the per-token edge draft time t_d as a fixed constant; on a consumer edge GPU it is not. The standard way to serve a CUDA-graphed model under varying shape is to capture graphs at a few bucket shapes and dispatch each step to the nearest—an established systems technique, which we do not claim as a contribution. We simply apply it to the *KV-length* axis: rather than one graph at the configured maximum length (3072 on SPROUT and 2048 elsewhere for **Ref**), whose attention reads the full window even when only the prefix is live, we capture a small set of graphs at length buckets (max-len $\in \{512, 1024, 2048, 4096\}$) and dispatch each draft step to the smallest covering bucket, so the attention contraction scales with the live window. What matters is that the base schedule leaves this unexploited—it treats edge draft time as fixed—so the optimization shortens the draft forward without touching verification, entirely edge-side (server forward unchanged). Together with cuDNN attention, which cuts the same attention cost, it forms **S1**. Figure 6 shows the effect on the RTX 3090 at 100 ms added latency: the mean draft forward falls from 13–16 ms (**Ref**) to 9 ms, and with it the edge wall time per 1k committed tokens on every benchmark (37 $\rightarrow$ 29 s on SPROUT, 47 $\rightarrow$ 41 s on SpecBench, 25 $\rightarrow$ 21 s on OlympiadBench); across the RTT grid **S1** is 1.10–1.17$\times$ **Ref** on SPROUT and 1.04–1.10$\times$ on SpecBench and OlympiadBench (Table 2).

B.3 Re-tuning the Synchronous Baselines

The synchronous baselines have two knobs that might be expected to close the gap to the asynchronous schedule: the proactive depth cap (with the rule that sets it) and the tree budget. This section re-tunes both and shows that neither does: the re-tuned depth gains only at 0 ms and **A3**'s lead holds from 50 ms on, and a larger budget saturates, since both forwards grow with the tree and **A3** stays 1.1–1.2$\times$ above the best budget of each cell.

Proactive depth cap. A shorter draft forward lets the synchronous schedule draft a deeper proactive tree within the same round trip. The cap $d_{\max}$ of Table 2 was set for **Ref**'s forward time, so with **S1**'s 9 ms forward the capped draft ends before the reply and the edge idles: at added 0 ms the $\approx$57 ms wait outlasts the three-level draft ($\approx$27 ms), and at 200 ms on the RTX 3090 **S1** idles for 116 of the 255 ms wait (**Ref** for 20 ms). The right cap is a hardware- and RTT-specific number, $\approx$ wait / draft-forward time ($\approx$9 ms: 8.9 ms on the 3090, 9.0 on the 4090, 9.4 ms for **S3**'s wider tree; Table 3 lists the resulting caps).

Holding the main tree at depth 3 is SpecEdge's own rule, a shallow initial tree with the wait left to proactive expansion, and it trades acceptance for hit rate: at added 200 ms on the RTX 4090 it raises the share of rounds whose proactive tree is confirmed from 16/6/27% to 35/27/46% on SPROUT/SpecBench/OlympiadBench (**S1**), while each verify accepts 26–29% fewer tokens.

Table 3 sweeps this re-tuning of **S1** and **S3** over the Table 2 grid, with the main tree held at depth 3, the proactive cap filled to the wait and the budget unchanged. The net effect depends on the RTT: the re-tuning gains only at low RTT, where the shallow main tree lets the proactive tree fill the wait; as the RTT grows, the deeper main tree of the original cap accepts more tokens per verify, and from 300 ms the re-tuned variants fall behind it. **A3**'s lead therefore holds from 50 ms on and grows from 3–11% to 34–70% at 300–500 ms; neither re-tuning the depth nor enlarging the budget (below) closes the gap, and both need the per-RTT, per-GPU tuning that the asynchronous schedule does without. Because the trend reverses with the RTT, the main results keep the single cap $d_{\max}$, which needs no per-benchmark tuning.

	depth cap			SPROUT						SpecBench						OlympiadBench					
Added (ms)	Tab. 2 $d_{\max}$	S1^{fill} proactive	S3^{fill} proactive	S1	S1^{fill}	S3	S3^{fill}	A1	A3	S1	S1^{fill}	S3	S3^{fill}	A1	A3	S1	S1^{fill}	S3	S3^{fill}	A1	A3
				tok/s						tok/s						tok/s					
0	3	5	5	58.6	**64.9**$^{+}$	60.2	**66.5**$^{+}$	58.1	61.2	50.4	**51.0**$^{+}$	52.3	**50.3**	45.1	48.5	69.3	**78.6**$^{+}$	71.3	**78.6**$^{+}$	68.7	75.2
50	6	9	9	43.6	45.5^{+}	45.1	47.7^{+}	42.8	49.8	33.1	33.6^{+}	33.7	35.6^{+}	31.5	37.4	54.3	59.4^{+}	56.3	61.6^{+}	55.3	63.5
100	9	15	14	33.8	35.5^{+}	34.4	37.1^{+}	35.3	41.6	24.2	24.9^{+}	24.2	25.8^{+}	24.6	29.5	45.0	47.9^{+}	46.3	49.8^{+}	47.1	55.0
150	12	21	20	28.2	28.1	28.4	30.2^{+}	30.4	36.1	18.9	19.4^{+}	19.1	20.2^{+}	21.0	24.4	38.8	39.0^{+}	39.5	40.9^{+}	42.0	48.5
200	15	27	26	24.9	23.7	24.7	24.6	26.2	32.0	16.0	16.0^{+}	16.2	16.5^{+}	17.7	21.0	34.2	33.0	35.5	34.1	36.7	44.3
300	21	32	32	19.3	17.6	19.8	18.6	21.3	25.5	12.4	11.8	12.5	12.1	13.9	16.2	27.1	24.1	28.2	25.5	30.0	36.1
500	32	32	32	13.8	11.5	13.9	12.2	15.5	18.1	8.8	7.7	8.8	8.0	9.8	11.3	19.5	15.9	20.1	16.9	22.5	27.0

$^{+}$ = faster than the Table 2 cell; bold = at or above **A3**.

Table 3: Synchronous variants re-tuned to fill the wait (RTX 4090, 32 queries, tok/s): main tree at depth 3, proactive cap ≈ wait / draft-forward time (listed; one level lower for **S3**), 32-node budget unchanged. **S1**, **S3**, **A1** and **A3** = the Table 2 rows.

Draft budget. The second knob is the tree budget: instead of a deeper proactive tree, the synchronous baseline could spend the wait on a bigger one, since with more nodes per level the proactive tree hedges more continuations at the same depth. We take the strongest synchronous variant (**S3**) at its RTT-matched draft depth and sweep the tree budget from 32 to 96 nodes on both edges (Table 4). The pattern is uniform: a larger tree *does* accept more tokens per verify forward (e.g. $9.1 \rightarrow 11.1$ on SPROUT at added 200 ms), but both forwards grow with it: the verifier processes the whole tree in one forward ($42 \rightarrow 49$ ms from budget 32 to 96) and each draft forward expands more nodes per level, i.e. more rows in its batch (forward time grows with the row count, if slowly, Appendix B.5); the gains and costs largely cancel: the best budget raises throughput by 9% on average (at most 12%) on the RTX 4090, where budget 96 is still the best, and by 4% (at most 7%) on the RTX 3090, where budget 64 is the optimum and 96 already falls back. Widening a single proactive tree therefore saturates, whereas at the same added latency **A3** (Tables 2 and 11) exceeds **S3** at budget 32 by $1.23\times$ on average (1.11–1.30) and **S3** at its best budget by $1.15\times$ (1.09–1.24) at 100–300 ms, by keeping many budget-32 trees in flight across the round trip instead of paying for one large one. The optimum is edge-dependent (64 on the RTX 3090, $\geq$96 on the RTX 4090), so the budget is a per-device knob rather than a fixed constant.

		SPROUT				SpecBench				OlympiadBench			
Edge	Budget	50	100	200	300	50	100	200	300	50	100	200	300
tok/s (bold = best budget per column)													
RTX 4090	32	**45.0**	34.4	24.8	19.7	33.7	24.3	16.2	12.5	**56.3**	46.3	35.4	28.1
	64	44.6	37.0	26.4	20.6	**34.6**	26.1	17.0	12.7	55.1	47.8	37.6	30.8
	96	43.9	**37.9**	**27.2**	**21.7**	34.0	**27.2**	**17.4**	**13.3**	53.8	**48.4**	**38.5**	**31.5**
RTX 3090	32	**43.6**	34.0	24.5	19.7	33.2	23.9	15.8	12.4	**55.4**	45.9	35.2	28.3
	64	42.8	**35.3**	**25.4**	**20.4**	**33.3**	**25.2**	**16.3**	12.5	53.6	**46.3**	**36.3**	30.2
	96	39.4	33.7	24.5	20.3	30.3	23.8	15.9	**12.7**	48.2	43.4	35.7	**30.4**
acc/verify | verify ms													
RTX 4090	32	7.4|42	8.3|42	9.1|42	9.3|42	5.5|39	5.8|39	6.0|39	6.0|39	9.1|39	10.8|39	12.4|39	12.7|39
	64	7.9|47	9.3|47	10.5|47	10.7|47	6.1|43	6.5|43	6.7|43	6.7|43	9.6|43	11.8|43	14.3|43	15.3|43
	96	8.2|49	9.9|49	11.1|49	11.8|49	6.3|45	7.0|45	7.1|45	7.2|45	9.8|45	12.3|45	15.1|45	16.4|45
RTX 3090	32	7.4|43	8.3|43	9.1|43	9.4|43	5.5|39	5.8|39	6.0|39	6.0|39	9.1|39	10.7|39	12.4|39	12.9|39
	64	7.9|47	9.3|47	10.4|47	10.9|47	6.1|43	6.5|43	6.7|43	6.6|43	9.6|43	11.8|43	14.2|43	15.3|43
	96	8.2|50	9.9|49	10.9|49	11.6|49	6.2|46	7.0|46	7.1|46	7.2|46	9.8|46	12.3|46	15.2|46	16.5|46

Table 4: Edge draft-budget sweep of **S3** (1 client, tok/s; budget 32 = the **S3** rows of Tables 2 and 11; bold = best budget per column). acc/verify = accepted tokens per verify forward; verify ms = mean server-side verify time.

B.4 Anchor Selection and Hedged Growth

Let $P^\star$ be the most probable root-to-leaf path of the current tree, $\mu(u) = \log p_d(c^\star(u) \mid u) - \log p_d(c'(u) \mid u)$ the margin at a fork $u \in P^\star$ between its on-path child $c^\star(u)$ and the runner-up $c'(u)$, and F_{K-1} the $K-1$ forks with the smallest μ among those with $\mu < 3$ nats. The anchor set is

$$A = \{\text{tip}(P^\star)\} \cup \big\{\text{tip}\big(P^\star(c'(u))\big) : u \in F_{K-1}\big\},$$

where $P^\star(c'(u))$ is the most probable path through the runner-up branch; if fewer than $K-1$ forks qualify, the next most likely leaves are added. The margin is a local, depth-independent measure

of where the draft hesitated; ranking alternatives by cumulative probability would favor the forks nearest the leaf, since an earlier fork carries a longer runner-up path. Growth is local as well: the K subtrees share one draft forward per level, each receives an equal quota of the beams and ranks candidates within itself, so a hedge subtree is not starved by the leading one (identical token IDs under different anchors remain distinct, since they represent different prefixes and KV states), and the tree may grow to K times the single-tree budget until the verdict selects the surviving anchor, after which the other subtrees are discarded and the survivor is trimmed to one budget. The next tree is drafted under anchors chosen on the surviving lineage in the same way, so the hedge carries across tree boundaries. The synchronous **S3** hedges with the same K but selects its anchors as the K most probable (leaf, next-token) continuations of the main tree, since its proactive tree is rebuilt every round.

B.5 Forwards per Token: Where A3's Gain Comes From

To attribute **A3**'s gain, every draft forward on the RTX 3090 was timed with CUDA events and assigned to the tree level it produced. A forward is *useful* when its level is on the accepted path; the count of useful forwards equals committed tokens minus bonus tokens to within 0.4% in every cell. Figure 6 gives the result per committed token at 100 ms added latency, with the synchronous baselines in their re-tuned form (**S1**^{fill}, **S3**^{fill}: main tree 3, proactive cap filled to the wait). The draft forward itself varies little across benchmarks: 13–16 ms for **Ref** (no cuDNN attention, no buckets; forward time follows the configured KV length: 15.8 ms at 3072 on SPROUT, 13.2–13.4 ms at 2048), 9.2 / 10.0 ms for **S1**^{fill} / **S3**^{fill}, 9.4 / 10.4 ms for **A1** / **A3**—the hedged forward carries ≈ 30 rows against 9 yet is only 12% longer than **S1**'s, because forward time is nearly flat in the row count. The variants differ instead in forwards per committed token (Table 5): useful forwards are nearly equal across variants, while rejected forwards—drafted but never accepted—are fewest for **A3** on every benchmark. Filling the wait with a deeper proactive tree does not help the synchronous edge, since the proactive tree is discarded in three rounds of four, so idle time turns into rejected drafting at the same wall time; **A3**'s hedging instead keeps more of its in-flight trees on the accepted lineage. Per 1k tokens the edge GPU therefore spends less time in draft forwards for **A3** than for **S1**^{fill} on every benchmark (Figure 6b–d), with a larger share of non-forward time on SpecBench, whose short answers make the per-query prefill and setup more visible.

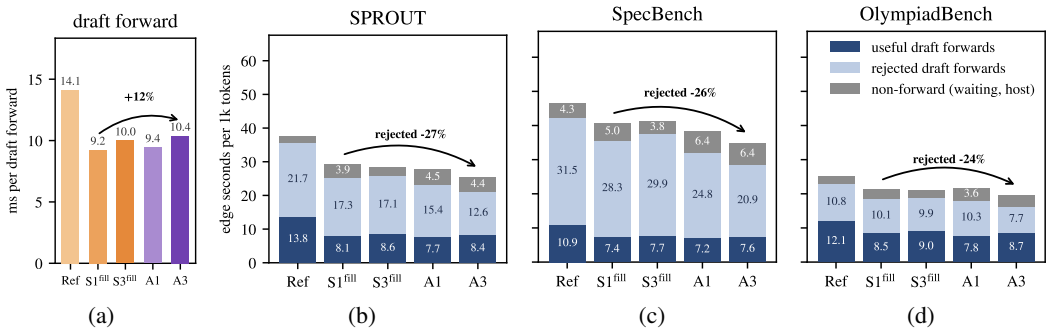

Figure 6: Edge time per committed token at 100 ms added RTT (RTX 3090). (a) Mean draft-forward time over the three benchmarks. (b–d) Edge seconds per 1k committed tokens.

Table 5: Draft forwards per committed token at 100 ms added RTT (RTX 3090): useful (level on the accepted path) and rejected.

Variant	SPROUT		SpecBench		OlympiadBench	
	useful	rejected	useful	rejected	useful	rejected
Ref	0.88	1.38	0.83	2.39	0.90	0.81
S1^{fill}	0.88	1.85	0.83	3.13	0.91	1.08
S3^{fill}	0.88	1.66	0.83	2.98	0.91	0.98
A1	0.81	1.62	0.79	2.66	0.82	1.08
A3	0.81	1.19	0.78	2.06	0.82	0.72

B.6 Sweep over the Number of Anchors

Table 6 sweeps the number of anchors K of the asynchronous schedule on both edges. Adding anchors has two opposing effects. Each additional anchor raises the chance that a verdict lands on a continuation already in flight, so more replies are hits and fewer trees are rebuilt from scratch; this gain grows with the RTT, since more trees ride on each bet. But every anchor carries its own fixed-budget subtree, so the draft forward widens with K ($\approx$30 rows per level for K=3 against $\approx$9 for K=1, Appendix B.5) and its cost rises, while each further anchor is placed on a less likely continuation and returns less. The two effects cross at K=3: it is within a few percent of the best K at every RTT, whereas more anchors cost throughput at low RTT and add almost nothing at high RTT.

tok/s		**SPROUT**					**SpecBench**					**OlympiadBench**				
Edge	Added (ms)	K=1	K=2	K=3	K=4	K=5	K=1	K=2	K=3	K=4	K=5	K=1	K=2	K=3	K=4	K=5
RTX 4090	0	58.2	**62.0**	61.2	57.9	56.3	45.0	**48.5**	48.3	45.7	46.5	68.6	74.6	**74.9**	69.9	67.3
	50	42.6	49.5	**49.9**	47.8	46.9	31.5	35.5	**37.4**	35.9	36.0	55.1	63.2	**63.6**	59.9	58.5
	100	35.3	40.2	**41.6**	39.0	36.6	24.7	28.2	**29.6**	29.4	28.6	47.1	**55.2**	55.0	52.4	51.5
	200	26.2	30.3	**32.1**	31.0	30.7	17.7	20.0	21.0	**21.1**	**21.1**	36.8	43.3	**44.3**	42.1	41.1
	300	21.3	24.0	**25.4**	25.3	25.3	13.9	15.6	16.2	**16.4**	**16.4**	30.0	35.3	**36.2**	35.7	35.2
	400	18.0	20.1	**21.8**	21.0	20.7	11.5	12.7	13.3	13.5	**13.7**	26.3	29.5	**31.6**	30.1	29.6
	500	15.5	17.3	18.1	18.0	**18.3**	9.8	10.7	11.3	11.4	**11.5**	22.5	25.6	**26.9**	26.5	26.6
RTX 3090	0	54.0	54.9	**55.0**	53.0	52.6	43.8	44.3	**44.9**	43.6	43.0	63.7	**65.6**	65.0	63.1	62.0
	50	43.2	45.6	**45.7**	44.5	44.6	32.2	34.0	**34.9**	34.5	34.6	54.9	**57.9**	57.3	55.8	54.8
	100	36.0	37.9	**38.8**	36.1	35.8	25.4	26.9	28.0	**28.1**	**28.1**	46.5	49.9	**50.8**	49.2	48.5
	200	26.7	29.2	**29.8**	29.7	29.2	18.2	19.5	20.2	20.4	**20.7**	36.7	40.0	**41.0**	40.2	40.0
	300	21.2	23.6	24.5	23.9	**24.7**	14.1	15.2	15.8	16.0	**16.2**	30.1	32.8	**34.3**	34.0	34.1
	400	18.0	19.7	**20.7**	20.5	20.4	11.6	12.4	13.1	13.3	**13.4**	25.9	28.8	29.2	**30.0**	29.0
	500	15.8	17.0	17.9	**18.0**	17.5	9.8	10.5	11.0	11.2	**11.4**	22.4	24.9	**26.1**	25.9	25.7

bold = best K per row and benchmark.

Table 6: Sweep over the number of anchors K of the asynchronous schedule.

B.7 Jitter Resilience at Added RTT 0

The steady-state crossover (§6.2) understates the asynchronous schedule's value under *bursty* links. We isolate this at added RTT 0 and inject server-side latency jitter of increasing intensity under two models, a de-meaned heavy-tailed Pareto (α=1.5) AR(1) process (scale s) and a Gilbert–Elliott on/off burst process (mean burst f ms). The proactive baseline sizes its draft to the *expected* round trip (here $\approx$0), so a latency spike leaves it idle; the asynchronous schedule keeps growing and sending draft trees while a reply is delayed, and is therefore expected to degrade more gracefully.

Table 7 runs this sweep on both edges with the bucketed synchronous baseline (**S1**), the synchronous K=3 baseline (**S3**) and the two asynchronous schedules (**A1**, **A3**), and **S3**^{until}, a variant of **S3** that keeps growing the proactive tree level by level until the reply arrives, finishes the level in flight, applies the verdict and commits. At zero jitter the asynchronous variants are at or below parity with the baselines; as the spikes intensify the ordering reverses, and **A3**'s margin over **S1** grows with the jitter, to up to 1.56$\times$ on the RTX 4090 and 1.39$\times$ on the RTX 3090 at the strongest bursts, with **A1** following the same curve at a smaller margin. **S3** tracks **S1** at every jitter level, so hedging alone does not hide latency spikes; the gain comes from asynchrony. **S3**^{until} recovers part of the gap on SPROUT and OlympiadBench, 1.03–1.28$\times$ **S3** at the same level and most at the strongest bursts, but not on SpecBench, whose short answers leave little wait to fill (0.95–1.07$\times$ **S3**). **A3** stays ahead wherever the jitter is substantial, by up to 1.23$\times$ at the strongest bursts; at the two mildest levels the two tie within $\pm$4%, and in the jitter-free control **S3**^{until} edges ahead of **A3** on the long-output benchmarks (1.03–1.07$\times$), where the proactive depth settles at 4–7 and no reply is late; with a steady added RTT the gap reopens, as the filled-depth sweep of Table 3 shows (**A3** ahead from 50 ms on). A synchronous edge cannot react to a delayed reply: even **S3**^{until} can only deepen one tree within the fixed budget while it waits, and commits only after the reply, so the remaining gap is structural rather than a tuning artifact.

	SPROUT						SpecBench						OlympiadBench					
Jitter (added RTT 0)	**S1**	**S3**	**S3**^{until}	**A1**	**A3**	**A3/S1**	**S1**	**S3**	**S3**^{until}	**A1**	**A3**	**A3/S1**	**S1**	**S3**	**S3**^{until}	**A1**	**A3**	**A3/S1**
RTX 4090 edge																		
none (control)	58.9	60.0	63.7	57.4	61.7	1.05	50.5	51.2	48.6	46.0	49.4	0.98	70.1	71.3	78.2	70.4	74.3	1.06
Pareto-AR(1), s=10	49.8	50.7	55.7	49.1	57.0	1.14	39.8	41.2	40.3	38.5	41.9	1.05	59.3	60.5	69.7	62.6	69.8	1.18
Pareto-AR(1), s=30	38.7	39.2	43.7	43.7	49.5	1.28	29.2	29.6	29.3	30.0	34.3	1.17	46.3	47.0	56.8	53.4	59.5	1.29
Pareto-AR(1), s=60	29.0	29.4	33.4	36.9	41.1	1.42	20.4	20.7	21.0	24.4	26.4	1.29	34.8	35.3	43.2	44.0	46.3	1.33
Gilbert, f=50	47.6	48.2	56.2	50.0	57.3	1.20	42.3	43.1	43.3	40.2	43.8	1.04	55.6	56.7	69.9	61.8	69.7	1.25
Gilbert, f=100	41.1	42.2	49.3	46.9	53.3	1.30	38.6	39.2	40.2	37.6	42.3	1.10	48.0	48.9	62.0	57.3	64.8	1.35
Gilbert, f=200	33.0	33.6	40.8	42.8	48.4	1.47	32.9	32.9	35.3	34.9	39.7	1.21	37.9	38.7	49.5	52.4	59.3	1.56
RTX 3090 edge																		
none (control)	57.8	59.0	58.8	54.0	55.0	0.95	50.3	50.7	46.2	43.9	44.9	0.89	68.2	69.0	70.0	64.3	65.4	0.96
Pareto-AR(1), s=10	49.6	50.6	52.0	49.4	51.4	1.04	40.8	40.9	38.9	39.1	40.1	0.98	58.7	59.4	63.5	60.2	61.2	1.04
Pareto-AR(1), s=30	39.0	39.7	41.9	42.4	45.1	1.16	29.7	29.9	29.1	31.0	30.9	1.04	46.2	47.1	53.0	52.0	54.0	1.17
Pareto-AR(1), s=60	29.3	29.9	32.4	35.7	36.6	1.25	21.0	21.2	21.0	24.3	23.1	1.10	35.0	35.8	41.9	42.0	44.2	1.26
Gilbert, f=50	47.7	48.4	51.2	49.2	51.4	1.08	42.9	43.3	41.7	40.4	41.9	0.98	55.3	56.3	63.6	58.9	62.6	1.13
Gilbert, f=100	41.4	42.6	46.9	46.8	49.0	1.18	39.2	39.5	38.7	39.2	39.7	1.01	48.2	49.4	56.6	55.9	58.1	1.21
Gilbert, f=200	33.1	34.2	38.3	–	44.0	1.33	33.1	33.5	34.0	36.7	37.4	1.13	38.3	39.2	46.5	49.8	53.3	1.39

Table 7: Jitter resilience at added RTT 0 (1 client, tok/s): server-side de-meaned Pareto-AR(1) delay (scale s) and Gilbert–Elliott bursts (mean bad-state delay f ms). **S1/S3** at the nominal depth 3; the blank cell hit a deterministic K=1 livelock in both attempts.

B.8 Temperature Spot Check

All throughput results in this paper use greedy decoding (temperature 0), whereas SpecEdge's original evaluation samples at temperature 0.7. Table 8 repeats three variants on SpecBench at added RTT 0 and 300 ms with the verifier sampling at temperature 0.7. Throughput changes by -2 to -11% and the ordering of the variants is unchanged on both edges, so temperature is not a confound for the comparisons above. One caveat remains: the asynchronous schedule predicts the next committed position assuming the verifier's bonus token is its argmax; under sampling we observed one stalled run (RTX 4090, added RTT 300 ms) that a retry cleared, so deployment at temperature >0 requires an explicit cursor re-synchronization step.

Throughput (tok/s)		**Added RTT 0 ms**		**Added RTT 300 ms**	
Edge	**Variant**	T=0	T=0.7	T=0	T=0.7
RTX 4090	**Ref**	47.5	44.9	11.4	11.0
	S3	52.1	49.9	12.5	11.1
	A3	48.3	45.7	16.2	15.3
RTX 3090	**Ref**	41.4	40.4	10.9	10.5
	S3	50.7	48.3	12.4	11.4
	A3	44.9	42.3	15.8	15.4

Table 8: Verifier temperature 0 vs. 0.7 on SpecBench (1 client, client-span tok/s).

B.9 Other Model Pairs

Two further model pairs test whether the SPEC-CLOUD results are specific to Qwen3-1.7B/14B. Table 9 swaps the family (Llama-3.2-1B-Instruct on the RTX 4090 edge drafting for Llama-3.1-8B-Instruct on the same A100), and Table 10 enlarges the target (Qwen3-1.7B on the edge drafting for Qwen3-32B on an H100-80GB server). In both cases the synchronous depths were re-tuned for the pair, since the 14B depths would otherwise leave the draft idle for part of the round trip: the 1B draft forward is $\approx$40% faster and the 32B verify takes 60 ms instead of 39 ms, so the 14B depths fill only two thirds of its window; the asynchronous schedule was run unchanged. With the 8B target the crossover lies between 50 and 100 ms, because its 24 ms verify leaves less to hide. The ordering matches Table 2: at zero added latency the schedules are within 6% of each other, and from 50–100 ms of added latency on **A3** leads with a margin that grows with the round trip.

Llama-3.2-1B → 3.1-8B (tok/s)		**SPROUT**						**SpecBench**						**OlympiadBench**					
Added (ms)	$d_{\max}$	**Ref**	**S1**	**S3**	**A1**	**A3**	**A3/S1**	**Ref**	**S1**	**S3**	**A1**	**A3**	**A3/S1**	**Ref**	**S1**	**S3**	**A1**	**A3**	**A3/S1**
0	5	75.7	88.4	**91.3**	75.4	87.7	0.99	68.2	73.4	**74.5**	62.0	70.0	0.95	95.6	101.9	**104.5**	88.5	99.1	0.97
50	10	55.6	63.6	**66.7**	53.8	66.0	1.04	46.1	48.7	49.3	42.2	**52.0**	1.07	72.4	78.3	**79.1**	65.8	78.4	1.00
100	15	40.8	46.1	46.8	45.1	**53.9**	1.17	33.6	34.8	35.6	33.0	**40.7**	1.17	53.2	56.3	58.5	54.6	**64.7**	1.15
200	21	28.6	31.2	32.4	31.8	**39.1**	1.25	22.6	23.3	23.6	22.8	**27.4**	1.18	35.7	37.5	38.5	40.2	**48.9**	1.30
300	27	22.1	24.0	24.6	25.3	**30.1**	1.25	17.1	17.6	17.8	17.8	**21.3**	1.21	27.4	28.7	28.8	32.3	**38.2**	1.33

bold = best per benchmark and RTT.

Table 9: Llama-3.2-1B-Instruct (edge draft) with Llama-3.1-8B-Instruct (cloud target); RTX 4090 edge, A100 server, 1 client, tok/s (Llama tokenizer). $d_{\max}$ = synchronous depth cap re-tuned for the pair.

Qwen3-1.7B → 32B (tok/s)		**SPROUT**						**SpecBench**						**OlympiadBench**					
Added (ms)	$d_{\max}$	**Ref**	**S1**	**S3**	**A1**	**A3**	**A3/S1**	**Ref**	**S1**	**S3**	**A1**	**A3**	**A3/S1**	**Ref**	**S1**	**S3**	**A1**	**A3**	**A3/S1**
0	5	42.8	51.8	**53.0**	46.9	51.0	0.98	37.3	41.6	**41.7**	36.8	40.6	0.98	60.9	67.6	**68.1**	61.5	68.0	1.01
50	8	32.3	36.8	37.7	37.5	**41.6**	1.13	24.8	26.8	26.9	26.0	**30.2**	1.13	47.8	51.3	53.1	49.6	**58.1**	1.13
100	13	24.3	28.6	30.1	30.3	**35.3**	1.23	17.6	19.5	19.6	21.0	**24.6**	1.26	40.6	44.6	45.2	42.1	**51.0**	1.14
200	20	18.7	21.6	22.2	23.2	**26.8**	1.24	12.6	13.7	13.8	15.4	**18.1**	1.32	30.4	33.1	34.0	34.0	**40.4**	1.22
300	27	15.5	17.1	17.2	18.9	**21.6**	1.26	10.1	10.9	11.0	12.4	**14.1**	1.29	24.8	26.3	27.1	28.1	**33.6**	1.28

bold = best per benchmark and RTT.

Table 10: Qwen3-1.7B (edge draft) with Qwen3-32B (cloud target) on an H100 server; RTX 4090 edge, 1 client, tok/s. $d_{\max}$ = synchronous depth cap re-tuned for the pair.

B.10 Replication on a Second Consumer Edge GPU

To test whether the behavior is a property of the schedule rather than of one edge accelerator, we repeated the current-state grid of Table 2 with an RTX 3090 as the edge GPU in place of the RTX 4090, against the same A100 server, models and configurations, at added RTT 0–500 ms (Table 11). The pattern replicates: **S1** and **S3** gain over **Ref**, and **A3** trails the best synchronous variant at zero added latency, crosses over at 50 ms and leads by up to 1.3× at 500 ms on all three benchmarks, with **A1** between the two. Absolute throughput is lower on the slower drafter at low RTT, and the difference vanishes at 300–500 ms, where the network hides the edge GPU.

		SPROUT							**SpecBench**							**OlympiadBench**						
	Added RTT (ms)	0	50	100	200	300	400	500	0	50	100	200	300	400	500	0	50	100	200	300	400	500
1 client	**Ref**	42.5	32.4	26.3	19.5	16.3	13.9	12.2	41.4	28.5	20.8	13.7	10.9	9.2	8.0	55.3	46.6	39.7	29.6	24.7	20.6	17.8
	S1	58.0	42.6	33.6	24.1	19.1	15.9	13.7	50.2	32.6	23.9	15.5	12.2	10.2	8.7	67.7	53.8	44.7	33.5	27.4	22.7	19.3
	S3	**59.0**	43.6	34.0	24.5	19.7	16.4	14.0	**50.7**	33.2	23.9	15.8	12.4	10.1	8.7	**68.9**	55.4	45.9	35.2	28.3	23.6	20.1
	A1	54.0	43.2	36.0	26.7	21.2	18.0	15.8	43.8	32.2	25.4	18.2	14.1	11.6	9.8	63.7	54.9	46.5	36.7	30.1	25.9	22.4
	A3	55.0	**45.7**	**38.8**	**29.8**	**24.5**	**20.7**	**17.9**	44.9	**34.9**	**28.0**	**20.2**	**15.8**	**13.1**	**11.0**	65.0	**57.3**	**50.8**	**41.0**	**34.3**	**29.2**	**26.1**
2 clients	**Ref**	85.3	64.8	52.2	39.1	32.4	27.7	24.4	82.3	56.9	41.5	27.2	21.7	18.2	15.9	110.1	93.2	78.4	58.8	49.1	41.0	35.6
	S1	94.5	84.2	66.0	47.5	37.9	31.8	27.3	84.7	65.0	47.3	31.0	24.2	20.1	17.3	116.6	105.8	88.0	67.2	53.7	44.7	38.4
	S3	**97.0**	**86.2**	67.1	49.0	38.8	32.3	28.1	**86.7**	**65.4**	47.5	31.2	24.7	20.3	17.5	**119.8**	**109.2**	**90.8**	69.2	55.8	47.4	39.9
	A1	88.3	76.2	66.1	50.7	41.7	35.0	30.5	78.2	60.6	49.0	35.7	28.1	22.9	19.6	107.4	95.8	85.7	69.3	58.3	50.2	43.9
	A3	87.9	78.5	**69.8**	**56.5**	**47.2**	**39.8**	**34.5**	79.2	64.4	**53.2**	**39.3**	**31.3**	**25.9**	**21.9**	108.0	96.5	89.6	**75.9**	**65.3**	**56.4**	**51.0**

bold = fastest per column within each group.

Table 11: **RTX 3090 edge**, 1 and 2 clients, with the server of Table 2. 1 client: tok/s; 2 clients: sum of both clients' tok/s (two edge GPUs sharing one A100).

Two-client shared-server load. Every sweep so far serves one edge client per server, whereas a central motivation for offloading verification to a shared cloud target is that one server amortizes its GPU across *many* edge clients. As a first probe of this regime, the RTX 3090 grid was repeated with two concurrent edge clients sharing one A100 server (two-client group of Table 11). Dividing the summed throughput by the matched single-client value gives ≈2.0× for **Ref** at every RTT and for **S1**/**S3** from 50 ms on (1.6–1.7× at 0 ms) (the server is mostly idle between their rounds) and 1.60–1.99× for **A3**, rising with the RTT (1.60–1.76× at added 0 ms, ≥1.9× from 300 ms) because

the asynchronous schedule already keeps the server busier with one client. The server verifies at batch 1 throughout, interleaving the two request streams; batching verifications across clients would let one server serve more edges at little cost in aggregate throughput.

C ROUTER: ABLATIONS AND EXTENSIONS

The router results here and in §6.3 use per-query latencies from the calibrated profiles of Eq. (4), applied to each query's prompt and to the output its chosen mode produced, rather than from a separate end-to-end run of the routed workload.

C.1 ROUTER BEHAVIOR

Server usage. An always-cloud system sends every query to the cloud, so its server-usage rate is 100% by definition. On SPROUT at the 500 ms tail, the router serves only 43.3% of queries from the cloud, 2.31× fewer, and those queries carry only 36.1% of the output tokens, so the verifier processes 2.8× fewer tokens (server time in Appendix C.5). Realized accuracy holds at 79.6%: the on-device reasoning modes recover the accuracy the cloud would have supplied, so the offloaded queries lose no accuracy.

Per-benchmark routing. The split depends on how much of a benchmark the edge can answer well (Figure 7). On SPROUT the cloud share falls gradually from 68% to 43% as the RTT grows and the two on-device reasoning modes absorb the difference. On SpecBench the 14B answers dominate the judge score, so 92–97% of the queries stay on the cloud at every RTT and routing changes little. On OlympiadBench the 1.7B chain-of-draft matches the non-thinking 14B target (54.8 vs. 51.6% correct), so once the round trip costs more than the longer on-device answer the router moves almost all queries to the edge (cloud share 95% at 0 ms, 6% at 500 ms). The router therefore adapts per query rather than per benchmark: the same rule yields three different regimes.

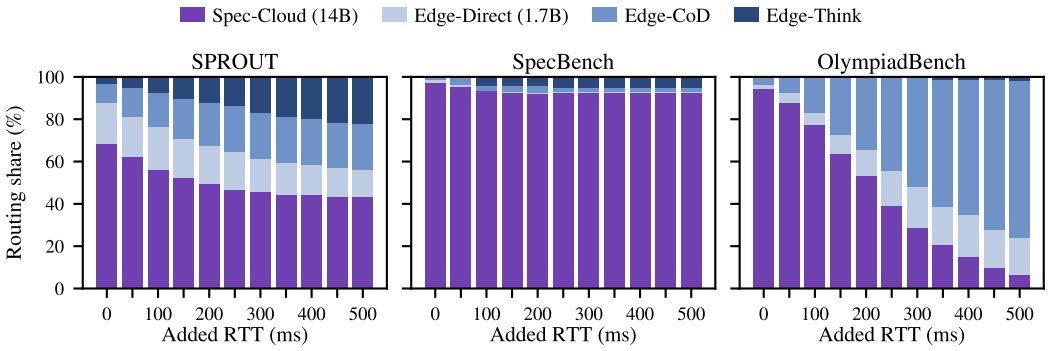

Figure 7: Routing shares vs. added RTT on the three benchmarks (50 ms steps).

Per-group behavior. The aggregate result conceals informative per-group structure, shown in Table 12 at 300 ms, reported at the *max-accuracy* operating point of the router on **`A3`**. At this point, overall accuracy is 80.5% (+0.9 pp over the always-14B target), and the aggregate throughput speedup is 1.72× over **`Ref`**. The router optimizes a *global* objective, so individual groups may sit above or below the always-cloud accuracy while the aggregate gains: it reallocates accuracy from where the edge is competitive to obtain latency savings where queries are edge-sufficient.

		Acc. (%)			Share (%)			
Group	n	**Ref**	**R**	Spd	Cl	Dir	CoD	Th
GPQA	94	42.6	41.5	1.54×	74	2	6	17
MuSR	123	59.3	61.8	2.09×	49	2	15	35
MMLU-Pro	1772	69.4	67.6	1.58×	78	3	9	10
MATH	1462	88.1	93.2	1.57×	11	3	61	25
OpenHermes	2912	82.9	83.5	1.89×	57	12	5	27
RAGBench	274	87.2	87.0	1.73×	70	1	7	22
All	6637	79.6	80.5	1.72×	53	7	19	22

Table 12: Per-group routing on SPROUT at 300 ms added latency, max-accuracy operating point (router on **A3**; overall 80.5%, +0.9 pp over the always-14B target; 1.72× **Ref** throughput). Acc: **Ref** (the 14B target) vs. **R**; Spd: speedup over **Ref** (as in Table 2); Share: routing fractions over SPEC-CLOUD, EDGE-DIRECT, EDGE-COD, EDGE-THINK.

Three patterns emerge. *(i) Edge-sufficient groups gain speed* and *accuracy.* OpenHermes—open instruction following, and the largest group—routes 12% of its queries to EDGE-DIRECT and 27% to EDGE-THINK, and outperforms the always-cloud baseline on both axes (82.9 → 83.5 at 1.89×, rising to 2.10× by 500 ms); MuSR gains +2.4 pp at 2.09×. *(ii) On-device reasoning gains accuracy where the edge outperforms the cloud, without a latency penalty.* On MATH, chain-of-draft is *more* accurate than the 14B cloud (93.5% vs. 88.1%), so the router sends 61% of MATH queries to EDGE-COD, lifting accuracy by 5.1 points *while* still running 1.57× faster at 300 ms (rising to 2.12× by 500 ms)—the concise drafts preserve the latency gain while adding accuracy. *(iii) Hard knowledge stays on the cloud.* GPQA and MMLU-Pro remain cloud-dominant (74%, 78%), where the 1.7B cannot match the target; their residual dips (−1.1, −1.8) are far outweighed by the MATH, MuSR, and OpenHermes gains, for an aggregate of 80.5%—+0.9 pp *above* the always-cloud baseline.

C.2 GENERALIZATION TO UNSEEN DATA

To test whether the routing function transfers beyond its fitting distribution, we evaluate on OlympiadBench He et al. (2024) ($n = 675$) without refitting the kNN index. OlympiadBench is both out-of-distribution—no Olympiad-level problems appear in SPROUT—and substantially harder: the always-cloud target reaches only 51.6%, versus ∼ 79.6% on SPROUT. The OlympiadBench columns of Table 2 already report the throughput sweep (up to 2.22× at 500 ms); here we describe how the routing distribution shifts to achieve it. At the fixed matched-accuracy operating point used in the main-text cost analysis, the measured SPEC-CLOUD fractions are 78%, 53%, and 29% of the queries (78/57/31% of the output tokens) at 100, 200, and 300 ms of added latency, respectively.

The router behaves as intended at both ends of the sweep. *At low RTT* it keeps most queries on the cloud (cloud share 95% at 0 ms, 53% at 200 ms and 29% at 300 ms) and stays at parity with running every query on SPEC-CLOUD (0.95–0.99× up to 200 ms): with hard, unfamiliar problems and a cheap network it does not over-offload to an edge that would lose accuracy. *At high RTT*, as the cloud's per-query latency climbs (to 84 s on average at 500 ms), the router shifts up to 74% of the queries to EDGE-COD and reaches 2.22× throughput at iso-accuracy by 500 ms (Table 2). This is not a quality sacrifice: router accuracy holds at 51–52% across the sweep, and on-device chain-of-draft alone scores 54.8% against the target's 51.6% (Figure 7), because on Olympiad-level mathematics, concise on-device reasoning outperforms the cloud target's direct answer. Traded for accuracy instead, the max-accuracy operating point reaches 55.9%, +4.3 pp above the target, while still running 1.87× faster than **Ref** at 500 ms (Table 13). The routing function transfers to a harder, unseen domain with no refitting: it neither over-commits to the edge when the network is cheap nor fails to exploit it when the network makes the cloud expensive.

We report SpecBench as an instructive negative result, because it delineates *when* embedding-based routing helps. On SpecBench the candidate qualities are sharply asymmetric—the 14B scores 0.92 (LLM-judge) while every 1.7B variant sits at 0.70–0.73—so the edge rarely substitutes for the cloud. The router recognizes this: its cloud share stays above 92% across the *entire* RTT sweep (from 97.2% at 0 ms to 92.7% at 500 ms), in contrast to SPROUT (68% → 43%) and OlympiadBench (95% → 6%). This is graceful degradation, not a failure mode: rather than over-offloading and losing accuracy, the router stays within 0.5 pp of 14B quality by staying on the cloud, so it never does *worse* than the SPEC-CLOUD variant it runs on in latency.

C.3 Router Model Size

The router's kNN scorer uses a Qwen3-Embedding-0.6B encoder rather than the 1.7B edge draft model. Two considerations motivate this. First, moving from 0.6B to 1.7B for the routing decision yields little accuracy change, so the larger model adds no routing quality. Second—and decisive for end-to-end latency—the 0.6B routing forward can be overlapped with the 1.7B draft model's prefill, whereas a 1.7B router would serialize behind it and add end-to-end overhead on every query.

C.4 Router Operating Points: Iso- vs. Max-Accuracy

Table 13 reports the router's two operating points side by side, both as speedups over **`Ref`** (as in Table 2): *iso-accuracy* (the smallest-latency λ whose overall accuracy matches the always-14B target; the **`R`** column of Table 2) and *max-accuracy* (the λ that maximizes accuracy). The accuracy headroom Δacc—how far the max-accuracy point exceeds the 14B target—is strongly dataset-dependent: largest on OlympiadBench ($+4.3$ pp, where the 14B target is weakest, $\approx 51.6\%$), modest on SPROUT ($+0.9$–1.1 pp), and absent on SpecBench (-0.3 to -0.5 pp, where 14B is near-ceiling and the 1.7B models cannot match it, cf. §C.2). On SPROUT and OlympiadBench the max-accuracy point remains a substantial speedup ($1.96\times$ and $1.87\times$ over **`Ref`** at 500 ms), so exceeding the target's accuracy costs no latency relative to the cloud.

Table 13: Router speedup over **`Ref`** at the iso-accuracy and max-accuracy operating points (router on the fastest SPEC-CLOUD variant of each cell: **`A3`**, or **`S3`** for SpecBench at added 0), with the max-accuracy point's accuracy change over the 14B target (Δacc, pp). The iso-accuracy row is the **`R`** column of Table 2; on SpecBench no operating point reaches the 14B accuracy, so both rows are the max-accuracy point.

Dataset	Operating point	Added RTT (ms)					
		0	**100**	**200**	**300**	**400**	**500**
SPROUT	Iso-acc ($\times$**`Ref`**)	1.24	1.59	1.84	1.97	2.13	2.21
	Max-acc ($\times$**`Ref`**)	0.92	1.31	1.56	1.72	1.86	1.96
	Δacc (pp)	$+1.1$	$+1.0$	$+0.9$	$+0.9$	$+0.9$	$+0.9$
SpecBench	Iso-acc ($\times$**`Ref`**)	1.12	1.33	1.50	1.50	1.47	1.47
	Max-acc ($\times$**`Ref`**)	1.13	1.33	1.50	1.50	1.47	1.47
	Δacc (pp)	-0.3	-0.3	-0.4	-0.5	-0.5	-0.5
OlympiadBench	Iso-acc ($\times$**`Ref`**)	1.13	1.30	1.37	1.65	1.98	2.22
	Max-acc ($\times$**`Ref`**)	0.62	0.94	1.09	1.32	1.62	1.87
	Δacc (pp)	$+4.3$	$+4.3$	$+4.2$	$+4.4$	$+4.4$	$+4.3$

C.5 Server Compute and Cost

The asynchronous overlap of drafting with verification costs additional server compute. Unlike the synchronous baselines, which deepen their proactive tree as the RTT grows and so spend the same verify budget on more depth (column $d_{\max}$ of Table 2), the asynchronous schedule keeps its tree shallow and instead issues *more* verification requests while a reply is in flight, so it uses 1.8–$2.3\times$ the verify-forward seconds of the synchronous schedules at matched RTT (Table 14: OlympiadBench, 100–300 ms added RTT), roughly constant as the RTT grows while the baselines idle more. Costs follow SpecEdge's cost-efficiency metric—tokens per dollar = throughput $\div$ hourly rental Park et al. (2025)—with the RTX 4090 edge counted identically for every schedule. On whole-session rental (billed by wall-clock time) the asynchronous schedule's faster completion makes the A100 rental per token 16–23% cheaper than **`S3`** at matched RTT; under a forward-only lower bound (a fully batched, multi-client server) its extra verifier work makes it $\sim 50\%$ more expensive per token. These cloud-side costs apply only to the queries routed to the cloud: at the matched-accuracy operating point the router sends 78/57/31% of the OlympiadBench output tokens to the cloud at 100/200/300 ms, and scaling the **`A3`** values by that share gives 149/111/61 s of verifier compute and \$17/16/10 per 1M tokens under rental—below every always-cloud baseline—and the lowest forward-only cost from 200 ms on.

Server-only speculative decoding. At long RTT one might instead keep both draft and target on the cloud and skip the round trip. Figure 8 includes this baseline (Srv: draft and target on one A100 in one process, the draft tree verified right after it is built; same models, budget and greedy decoding; its

draft forwards, 14.8 ms per 32-beam step, count as server time). It reaches 33.6–39.2 tok/s with the **`Ref`** draft stack (40.1/39.9/44.1 on SPROUT/SpecBench/OlympiadBench with the **`S3`** one, which is 0.59–0.76× the best edge–cloud schedule at zero added latency), and since it pays no round trip it overtakes **`A3`** in raw throughput from about 50 ms (SpecBench), 150 ms (SPROUT) and 200 ms (OlympiadBench) of added latency. The right axis of the figure shows the cost: the single A100 is busy for the whole run, so per 1k output tokens it spends about 3× the server time of the synchronous schedules at zero added latency and about 4–8× from 100 ms on (their verify work shrinks as the round trip grows), and 2.9–3.7× that of **`A3`**. Offloading the draft to the edge is what makes the server verify-only, which is the SpecEdge premise; our design trades a round trip—hidden by the asynchronous overlap or avoided outright by the router—for that server capacity.

Metric	Added	Eff. RTT	`Ref`	`S1`	`S3`	`A1`	`A3`	`R`
Server compute (s)	100	115	107	105	103	201	191	149
	200	218	92	90	90	207	195	111
	300	321	88	88	89	212	198	61
Rental ($/1M tok)	100	115	29	27	26	26	22	**17**
	200	218	39	36	35	33	28	**16**
	300	321	48	45	44	41	34	**10**
Compute ($/1M tok)	100	115	6.5	6.3	**6.2**	10.0	9.4	7.4
	200	218	6.7	6.4	6.3	10.8	10.0	**5.7**
	300	321	7.2	7.0	6.9	11.5	10.6	**3.2**

Table 14: Server compute and cost on OlympiadBench (RTX 4090 edge, one client): verifier-forward time per 32-query run, session rental cost per 1M tokens (A100 $4.05/h + RTX 4090 $0.35/h by wall clock) and the forward-only lower bound; **`R`** = **`A3`** scaled by the router's cloud token share (78/57/31% at 100/200/300 ms); bold = cheapest.

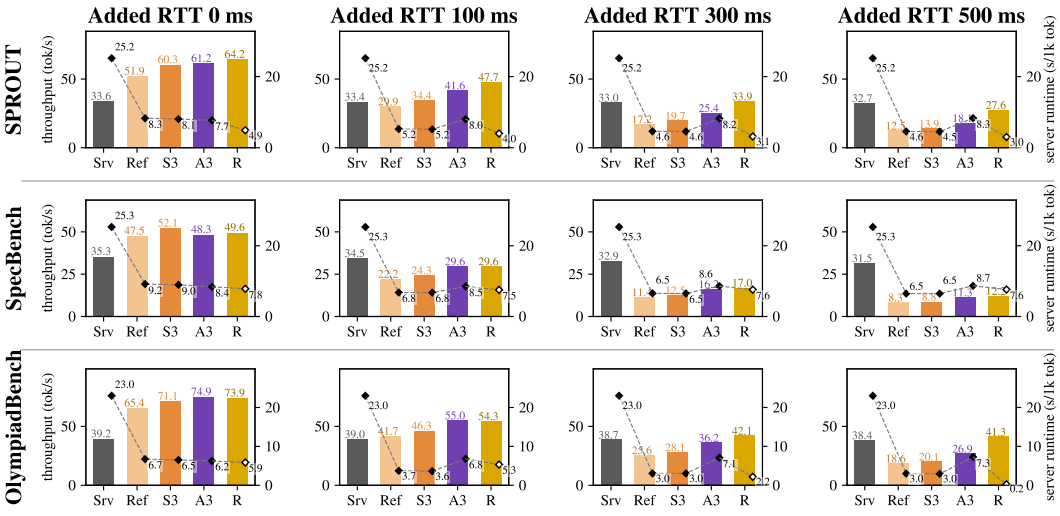

Figure 8: Decoding throughput (bars) and server runtime per 1k output tokens (♦). Srv = server-only speculative decoding, whose draft forwards also run on the server; **`R`**'s server time (◇) = **`A3`**'s per-token server time × the router's cloud token share.

D Extended Related Work

Same-request runahead—drafting past a block that is still being verified—originates in PipeInfer Butler et al. (2024), which keeps speculating while earlier micro-batches traverse a pipeline-parallel target and cancels invalidated work. AMUSD McDanel (2024) and PEARL Liu et al. (2025a) overlap drafting and verification on a single node, SpecPipe Yin et al. (2025) streams a fixed-width tree level by level into a pipeline-parallel target, and PipeSD Han et al. (2026) pipelines

edge drafting with cloud verification. In all of these the runahead depth is fixed by the hardware (a round or two) and rejected work is cancelled or flushed. SwiftSpec Zhang et al. (2026c) scales the idea over NVLink with a disaggregated draft model whose draft side keeps growing the tree while a batch is verified and then re-roots it at the verified token; none treats the WAN RTT as the variable or keeps several hedged trees in flight across a round trip.

Beyond SpecEdge, edge–cloud speculative decoding has been studied along two other axes. One shares or splits the verifier: DSSD NING et al. (2025) splits verification between device and edge, SLED Li et al. (2025) and WISP Li et al. (2026) serve many edge drafters from a shared verifier, and CoSine Gao et al. (2026) decouples drafting and verification across collaborating draft nodes. The other reduces the bytes per round: conformal sparsification Bhattacharjee et al. (2025), quantize-sample-and-verify Zhang et al. (2026a), and cooperative cache sharing with incremental synchronization Yang et al. (2025). All of these keep a synchronous draft–verify schedule per request. AsymSpec Yang et al. (2026) goes further and *forbids* same-request runahead—at most one open block per request—filling verification waits with other requests to raise multi-tenant throughput under asymmetric bandwidth, and Lyu et al. Lyu et al. (2026) show analytically and empirically that edge–cloud speculative decoding reduces latency only inside a bounded RTT window, outside of which co-located speculative decoding wins. SpecRelay targets that regime for a single request: it keeps 2–9 fixed-budget trees in flight as the added RTT grows from 0 to 500 ms, contains the wasted work with a committed-prefix gate, continuation hedging, and lineage-preserving rebuild, and its router falls back to on-device modes once the cloud path stops paying off.