SPECPILOT: GUIDING SPECULATION WITH ONLINE ADAPTIVE COST PREDICTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Speculative decoding speeds up autoregressive inference by having a cheap proposer draft several tokens that the target model then verifies in a single forward pass. Whether a step pays off depends on how far acceptance holds, what verifying a given tree costs on the machine at hand, and what drafting itself costs, and all three vary with the workload or the machine. Fixed policies therefore win on some workloads and lose on others: the draft length that is best on math and JSON for llama.cpp's speculative decoding is the worst on prose, where it runs $0.69\times$ as fast as not speculating at all. We present SpecPilot, a planner that decides at every step how many drafter levels to buy and which tree to verify, judging each choice against the throughput the run has actually achieved. SpecPilot learns the verify cost surface online, including kernel-switch discontinuities whose direction differs between GPUs, and can pad a tree past one when the larger tree is cheaper. It prices each node by a path probability calibrated against observed acceptance, and requests another drafter level only while its expected yield covers the measured cost of drafting. With a single untuned configuration on an RTX 3090, SpecPilot reaches a $1.78\times$ geometric-mean speedup over plain decoding across five workloads, 14.1% faster than llama.cpp and 3.7% faster than CAST, each tuned per cell in hindsight. CAST's best threshold changes with the workload and the GPU: across 23 cells on three GPUs, SpecPilot matches CAST's best fixed threshold (+0.5%) and is within 0.3% of CAST retuned for every cell, while CAST's worst threshold falls up to 22.7% below it.

1 INTRODUCTION

The autoregressive nature of decoder-only Transformers (Vaswani et al., 2017) makes efficient LLM inference difficult: each token requires a full forward pass of the model, and at small batch sizes that pass is bound by memory bandwidth rather than compute. Speculative decoding (Leviathan et al., 2023) exploits the idle compute: a cheap proposer drafts several tokens ahead, and the target model verifies them all in one forward pass, committing every token it agrees with.

A speculative step pays off only if the tokens it commits outweigh everything the step costs: proposal, planning, and verification. Neither side of that comparison is fixed. How long a drafter's chains stay correct depends on the text being generated: the same drafter and draft length that more than double throughput on one workload run slower than plain decoding on another. The drafter's confidence in each token is often close to the probability that the token is accepted, but not for every drafter on every text, and a path's probability is a product of these confidences, so a small error on each edge compounds with depth. Verify cost depends on the machine: moving from eight rows to nine makes verification cheaper on one GPU and more expensive on another. And drafting and planning often cost more than the extra verification itself. Existing planners simplify both sides: they fix a node budget or price trees with a smooth or offline cost curve, rank nodes by the product of raw draft confidences, and judge each tree in isolation rather than against the rate the run has actually achieved (Section 3).

We present SpecPilot, a speculation planner that treats every speculative step as an investment priced in units of the target's own forward pass. SpecPilot learns the cost of verification online, for the machine and model it runs on, and prices each tree by path probabilities built from confidences

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.

calibrated online rather than raw ones. It buys drafting one level at a time, and admits the tree that maximizes expected return minus cost, both weighted by the rate the run has actually achieved.

Our contributions are:

- **A run-level objective.** Judging each tree by its own return admits trees that dilute the rate the run has already banked. Following Dinkelbach's parametric reduction (Dinkelbach, 1967), SpecPilot instead admits, at every step, the tree maximizing $R - \rho L$, where ρ is the run's realized rate measured from wall-clock cost (Section 4).
- **An online, discontinuous cost model.** A cost surface learned from the run's own forwards that finds kernel-switch discontinuities without being told where they are, together with an admission step that can pad a tree past one when the larger tree is cheaper (Section 5.2).
- **Calibrated path probabilities.** Online per-edge calibration of drafter confidences, fit only on edges whose parent was accepted. It corrects a drafter that is miscalibrated on the current text and leaves a calibrated one alone, and with it the product of edge beliefs predicts realized path acceptance within 10% on most cells without a correlation correction (Section 5.1).
- **Priced drafting and an evaluation.** A planner that buys each level of drafting only when its expected value covers its measured cost (Section 5.3). With one untuned configuration on an RTX 3090, SpecPilot is 14.1% faster than llama.cpp and 3.7% faster than CAST, each tuned per cell in hindsight; against CAST it wins 5 of 9 cells, ties 3 and loses 1. Across three GPUs it matches CAST's best fixed threshold, while no single CAST threshold is best on every GPU (Section 6).

2 Related Work

Proposers and fixed-budget trees. Speculative decoding verifies tokens suggested by a cheaper proposer (Leviathan et al., 2023): a separate draft model, extra heads on the target (Cai et al., 2024; Li et al., 2025; DeepSeek-AI et al., 2025), or model-free matching (Saxena, 2023; Oliaro et al., 2025). SpecPilot is orthogonal to the choice of proposer; it decides how much of what a proposer offers to buy and verify. Most tree planners maximize expected accepted length under a fixed node budget (Li et al., 2024; Cai et al., 2024; Chen et al., 2024). Goose (Jin et al., 2026) lets an accurate proposer grow a deep spine and a less accurate one fill wide branches off it, and proves that the resulting tree accepts no fewer tokens per step than either proposer used alone. Like other fixed-budget planners, it counts every node as equally expensive; we show that giving up some yield to reduce the cost of verifying and drafting a tree is sometimes the better trade.

Cost-aware planners. SMART (Wang & Zhou, 2026) is the most similar to our design. It prunes EAGLE-3 trees with a marginal benefit-cost test: a node is kept only if its return per unit cost exceeds the return-to-cost ratio of the tree built so far, discounted by a fixed factor for draft over-confidence. SMART's bar is the ratio of the tree being built, computed from the planner's own estimates; ours is the rate the run has actually achieved, measured from wall-clock cost. Its cost is a smooth curve in tree size, fit offline, so it cannot represent a discontinuous cost landscape, and its admission stops at the first unprofitable node, which only works when cost is monotone. CAST (Hong et al., 2026) likewise prices a tree against its expected return, but with a table measured offline for each tree size and context length, under the assumption that cost increases strictly with size. It grows a tree while each increment clears a fixed threshold. Both rank nodes by the product of raw draft probabilities. Neither can find a larger tree that is cheaper than a smaller one, as happens on the RTX 3090 (Section 3.2). Table 1 summarizes these differences.

3 Motivation

Notation. The *target* is the model whose output speculation must reproduce. A *proposer* suggests tokens the target may produce next; our only proposer is a *drafter*, a small draft model. The *planner* decides at each step which proposed tokens to verify. For a node v at depth d with root path $x_1, \ldots, x_d$, let $c(v) = p_{\text{draft}}(x_d \mid x_{<d})$ be the drafter's confidence, $q(v) = \Pr[\text{target accepts } x_d \mid x_{<d} \text{ accepted}]$ the acceptance probability of its edge, and $J(v) = \Pr[\text{target accepts } x_1, \ldots, x_d]$ the

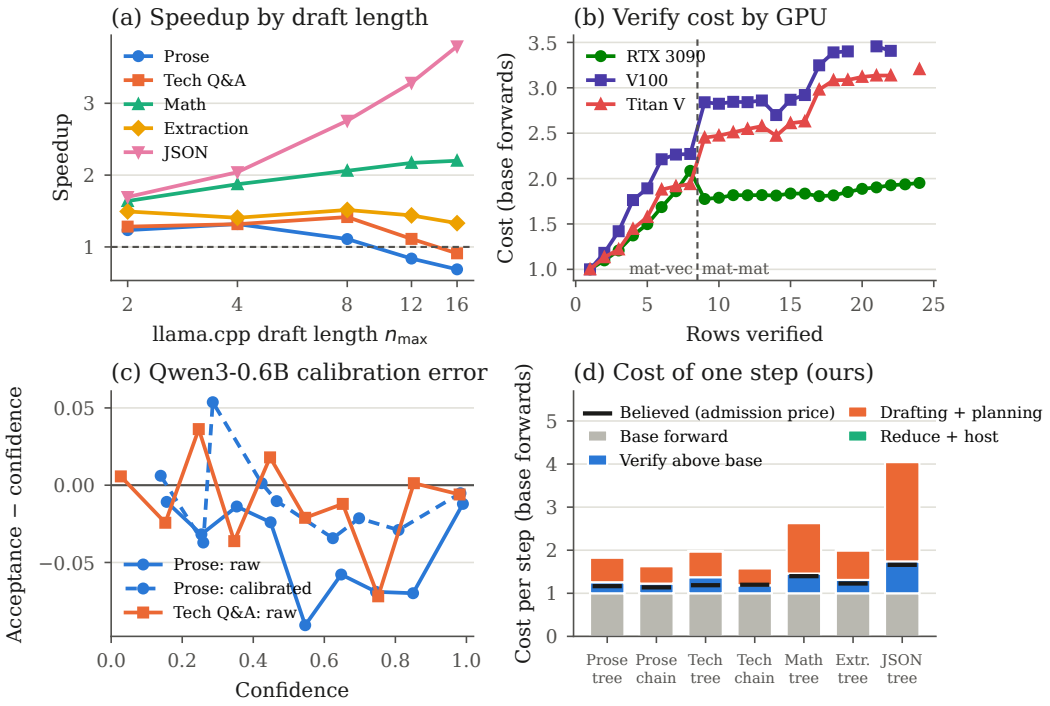


Figure 1: (a) Speedup of llama.cpp's speculative decoding by draft length $n_{\max}$, per workload. (b) Verify cost by rows verified, per GPU, over every verify pass of every planner in our runs; the dashed line marks llama.cpp's switch from a matrix-vector to a matrix-matrix kernel. (c) Calibration error of Qwen3-0.6B (Qwen3-14B target) on edges whose parent was accepted: raw confidence on prose and technical Q&A, and calibrated confidence on prose. (d) Per-step cost of our planner, split by component; black marks show the admission price it believed. Panels (a), (b) and (d): Qwen2.5-14B target, Qwen2.5-0.5B drafter; (a) and (d) on an RTX 3090.

probability of its path, taken over the planner's uncertainty before verification. A node contributes one committed token exactly when its whole path is accepted, so the expected yield of a tree S is $R(S) = 1 + \sum_{v \in S} J(v)$. Costs are measured in *base forwards*, the time of one plain decoding forward (Section 4).

3.1 Workload Dependence of Speculation Depth

The return on a speculative step depends on how far acceptance holds along the chain being proposed, and that depends on the text being generated. Figure 1(a) sweeps the speculative decoding of llama.cpp (Gerganov & ggml-org contributors, 2023) over its draft length $n_{\max}$ on five workloads. Its longest draft, $n_{\max}$=16, is the best setting on math (2.20×) and JSON (3.79×) and the worst on prose, where it runs at 0.69×, slower than not speculating at all. On JSON, at least 93% of drafted tokens are kept at every draft length; on prose, raising $n_{\max}$ from 4 to 16 quadruples the drafted tokens while the number accepted stays flat. No single setting is within 10% of the best on every workload, and choosing one per workload requires knowing the workload in advance, which a server handling mixed requests does not.

3.2 Discontinuities in Verify Cost

Verification cost is neither smooth nor monotone in tree size. llama.cpp multiplies a batch of at most eight rows with a matrix-vector kernel and switches to a matrix-matrix kernel above that, so the cost of a verify pass jumps between eight rows and nine, and the direction of the jump depends on the GPU (Figure 1(b)). On an RTX 3090, verifying nine rows of Qwen2.5-14B costs 15% less than eight (1.77 against 2.08 base forwards), and for Qwen2.5-32B nine rows cost less than seven or eight (50 against 55 and 62 ms); on a Tesla V100 and a Titan V, the same switch raises cost by 25% and

26%. A smooth cost curve such as SMART's (Wang & Zhou, 2026) cannot represent either jump, and an offline table indexed by size, as in CAST (Hong et al., 2026) and Sequoia (Chen et al., 2024), is correct only on the machine it was profiled on. On the 3090, a planner that grows a tree until the next node stops paying for itself never reaches the cheaper region beyond the jump. Reaching it requires pricing growth past the jump, with real nodes where they pay and with padding where they do not.

3.3 Drafter Calibration

A drafter's confidence $c(v)$ is a natural estimate of $q(v)$, and it is often a good one: over edges whose parent was accepted, the softmax of Qwen2.5-0.5B has an expected calibration error (ECE) (Guo et al., 2017) of 0.4–2.3% across our workloads on the RTX 3090, and where it errs it is slightly underconfident. But calibration is a property of the drafter *and the text*, not of the drafter alone (Figure 1(c)). On prose, tokens to which Qwen3-0.6B gives confidence between 0.5 and 0.9 are accepted 7.2 points less often than claimed (620 reached edges, $z = 3.7$), a systematic overconfidence that an ECE of 2.9% understates; on technical Q&A the same drafter has an ECE of 1.6%, with no consistent sign. A correction fitted offline for one drafter would be wrong on one of the two. Our online calibration (Section 5.1) lowers the prose ECE to 1.4% and the 0.5–0.9 gap to 2.4 points ($z = 1.2$).

Small per-edge errors matter because a node's value is its path probability $J(v)$, a product of per-edge estimates, which compounds their error: a bias of ε per edge becomes $(1+\varepsilon)^d$ at depth d. In an earlier version of our planner, the most confident decile of edges was valued 3–6% low, and realized depth-8 path acceptance exceeded the planner's estimate by 1.1–$1.3\times$; with that corrected, the two agree within 10% on seven of our 9 main cells (Section 5.1). Calibration must also be measured on the right population. An edge whose parent was rejected is never tested, and counting it as a miss deflates every estimate (Appendix A.1.3).

3.4 Proposal Cost

Drafting and planning cost more than the extra verification they buy. In every workload they exceed the verify time above the base forward, by 1.6–$3.1\times$ for our planner and 1.9–$2.5\times$ for CAST with the same drafter (Figure 1(d)). Each level of a drafter's tree is a full forward of the drafter, so drafting too deep, or drafting nodes that are later pruned, is paid for without return. These costs are also what a planner's own estimate leaves out: the verify-tree price understates the full wall-clock cost of a step by 1.3–$1.7\times$ on technical Q&A, 1.4–$1.6\times$ on prose, $1.6\times$ on extraction, $1.9\times$ on math, and $2.4\times$ on JSON. A planner that measures its progress by its own estimate therefore overrates it by the same factor.

3.5 Design Requirements

These effects give four requirements. **R1:** every tree must be judged against the rate the run has actually achieved, measured from wall-clock cost, so that how much to speculate follows the returns the workload is producing. **R2:** verify cost must be learned online, including discontinuities whose direction varies between GPUs, and the planner must be able to grow a tree past one. **R3:** edge beliefs must be calibrated online, on edges whose parent was accepted, because calibration varies with the workload and per-edge error compounds with depth. **R4:** each level of drafting must be bought only when its expected value covers its measured cost. Table 1 compares existing planners.

4 Problem Formulation

Cost unit. All costs are measured in *base forwards*: one forward pass of the target over a single token row, excluding logit reduction, timed as the median of the run's most recent 512 single-row forwards so that it follows the device. Plain decoding costs slightly more than one base forward per token, and all speedups are measured against plain decoding itself.

Objective. Step i verifies a tree S_i and commits its accepted tokens plus the target's bonus token. Step i costs L_i base forwards: its full wall-clock time from the end of the previous step, including

Table 1: How existing planners address each requirement. "Offline" means fit once before serving and never updated.

	R1: objective	R2: cost model	R3: belief	R4: depth
llama.cpp	none	none	raw confidence	fixed $n_{\max}$
EAGLE-2	expected length	none	product of softmax	fixed budget
Sequoia	per-tree ratio	offline, by size	offline rates	static tree
SMART	own-tree ratio	offline, smooth	product $\times\, \alpha$	per node
CAST	fixed threshold	offline, by size	raw product	per step
Ours	realized run rate	online, discontinuous	online, reach-conditioned	priced per level

drafting, planning, verification, and reduction. Over a run of n steps the planner commits $T = \sum_i R_i$ tokens at a cost of $C = \sum_i L_i$, and throughput relative to single-row decoding is the *rate* T/C. A rate of 1 means speculation gained nothing.

Maximizing R_i/L_i tree by tree does not maximize T/C, because it judges each tree in isolation. It admits the best-ratio tree even when that ratio is below the rate the run already achieves, diluting the rate already banked, and it prefers a small tree with a high ratio over a larger one that would add more tokens at a ratio still above the run's. The first matters more the higher that rate is. The quantity to maximize is the linear-fractional objective $\sum_i R_i / \sum_i L_i$. Dinkelbach's reduction (Dinkelbach, 1967) replaces maximizing a ratio with solving the linear problem $\max R - \rho L$ at a parameter ρ, and at the optimum ρ equals the best achievable ratio. At each planning step SpecPilot therefore admits the tree

$$S^\star \;=\; \arg\max_S \; R(S) - \rho\, \hat{L}(S),$$

where $R(S) = 1 + \sum_{v \in S} J(v)$ is the expected yield from Section 3 and $\hat{L}(S) = 1 + \hat{V}(S)$ is the predicted cost of verifying S: the base forward plus the verify cost above it (Section 5.2). Drafting is not part of $\hat{L}$. It is priced when levels are bought (Section 5.3), and it reaches admission through ρ, which is computed from measured cost. A tree is admitted over the empty tree only if $\sum_{v \in S} J(v) > \rho\, \hat{V}(S)$: its expected extra tokens must cost, on average, less than $1/\rho$ base forwards each. The better the run is doing, the more a tree must return to be worth its cost.

The rate must be realized. ρ is held fixed for the whole planning step, so every candidate tree is judged against the same bar. It is updated only after the step has run, from exponential moving averages of what the step actually committed and actually cost:

$$\bar{R} \leftarrow \gamma \bar{R} + (1-\gamma)\, R_i^{\text{real}}, \qquad \bar{L} \leftarrow \gamma \bar{L} + (1-\gamma)\, L_i^{\text{real}}, \qquad \rho = \bar{R}/\bar{L},$$

with $\gamma = 1 - 1/64$, long enough that one step cannot swing the decision it feeds back into and short enough to follow a workload change. ρ starts at 1, and steps with an empty tree are recorded too, so declined steps pull ρ back toward 1. L_i^{real} must be the step's measured wall-clock time: the predicted cost omits everything that is not verification and understates the true cost by 1.3–2.4× (Section 3.4), so a ρ computed from it would overstate the achieved rate by the same factor.

5 METHOD

SpecPilot plans each step in two stages. A priced beam search buys drafter levels while they can still pay (Section 5.3), and an exact dynamic program selects the tree to verify from the nodes it bought. Both price nodes with a path probability built from calibrated edge beliefs (Section 5.1, R3) and a verify cost surface learned during the run (Section 5.2, R2), and both judge against the realized rate ρ (Section 4, R1 and R4). Chain and tree geometries run the same code with different width limits and sequence budgets. Components summarized in one clause below are specified in Appendix A.2.

5.1 PATH PROBABILITY

Each drafter confidence is re-estimated online before it is used. Tokens fall into buckets by confidence decile and by whether they are their lane's most likely token, and each bucket's correction is

fit only on tokens whose parent was accepted. The correction rescales a confidence by its bucket's realized-to-claimed ratio, shrunk toward 1 while the bucket is sparse: a re-estimate of the same probability, not a second probability multiplied onto it, since two estimates of one event multiplied together deflate every edge and the error compounds with depth (Section 3.3). On a calibrated drafter it changes little; on Qwen3-0.6B's prose it halves the calibration error.

At each branch point u, SpecPilot estimates the probability $Z(u)$ that the target's next token is one of u's children, and splits it among them in proportion to calibrated confidence:

$$\hat{q}(v_i) = Z(u)\, w_i, \qquad w_i = \frac{\tilde{c}_i}{\sum_j \tilde{c}_j}, \qquad J(v_i) = J(u)\, \hat{q}(v_i).$$

With calibrated edges, this product predicts realized depth-8 path acceptance within 10% on seven of the 9 cells of our evaluation. It is slightly conservative (realized over predicted has a median of 1.06), and least accurate on Qwen2.5-14B Tech Q&A (tree), where it is 25% low over 48 reached paths. A correction for correlation between edges, fitted online, moves the depth-8 estimate by at most 0.2% (Appendix A.1.3).

5.2 Verify Cost and Admission

A tree is priced by its shape: rows r (its nodes plus the last committed token), sequences, and depth. Cost is smooth between kernel switches and jumps at them (Section 3.2), so the row axis is split into regions at changepoints found by a residual test. Within each region SpecPilot fits $\ln T = a + b \ln r + c\, r$, refitting jointly by ridge regression after every forward, and adds per-sequence and per-depth terms once the run's forwards show evidence for them. Admission prices against one Thompson sample (Thompson, 1933) $\hat{V}$ of this surface per step, so a size that looked expensive early is still retried; the beam prices on its mean $\bar{V}$. Across a downward cliff a larger tree can be cheaper, so the price is the lower envelope over padding the deepest path:

$$\hat{V}_{\mathrm{pad}}(S) = \min_{0 \le p \le P} \hat{V}\big(r + p,\ \mathrm{seq}(S),\ \mathrm{depth}(S) + p\big).$$

Because this price depends on the tree's whole shape, the admitted tree is chosen by an exact bottom-up tree knapsack over (nodes, leaves, depth) states. Each state is scored as $1 + \sum J - \rho\,(1 + \hat{V}_{\mathrm{pad}})$ against the empty tree's $1 - \rho$.

5.3 Buying Drafter Levels

Each drafter level costs a drafter forward, and these cost more than the verification they buy (Section 3.4), so every level is priced before it is requested. The drafter is opened when its measured gain exceeds ρ times the measured cost of opening it. Each further level is one batched drafter forward over up to eight lanes, and the planner times its cost $k(L)$ for each lane count L. With n nodes kept so far, a node is kept if

$$J(v) > \rho \min_{m > n} \frac{\bar{V}(m) - \bar{V}(n)}{m - n},$$

the cheapest per-node rate at which the tree could still grow, so a node just short of a downward cliff is not refused. Another level is requested only while some continuation can still pay:

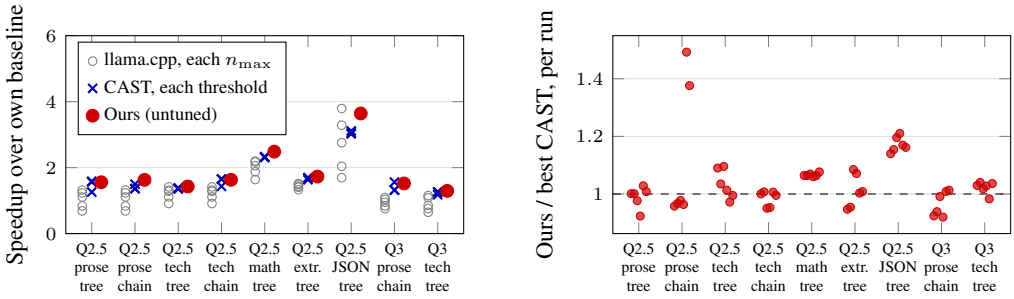
$$\max_{L,\, D,\, r} \Big[\textstyle\sum_{\mathrm{top}\ r} \hat{J}_D - \rho \big(\bar{V}(n + r) - \bar{V}(n)\big) - \rho\, k(L)\, D \Big] > 0,$$

where $\hat{J}_D$ projects a lane's path probability D levels deeper, pricing each further level like the one just seen, scaled per depth by how far past projections overshot the level actually kept.

6 Evaluation

6.1 Method

SpecPilot is built on llama.cpp (Gerganov & ggml-org contributors, 2023) revision `26394b4`. Our harness runs each planner over five chat workloads: prose, technical Q&A, math, information extraction and JSON generation, with three prompts each. For every prompt it decodes 1024 warmup tokens, measures 1024 more, and repeats the prompt twice, so every configuration is measured over six runs.



(a) Every setting of each system

(b) Paired runs against CAST's best threshold

Figure 2: (a) Speedup of every configuration we ran, per cell: llama.cpp at each draft length $n_{\max} \in \{2, 4, 8, 12, 16\}$, CAST at each threshold $\{2.5, 3, 4\}$, and our single untuned configuration. (b) Each of the six runs per cell (3 prompts $\times$ 2 repeats) as the ratio of our throughput to that of CAST at its best threshold for the cell, same prompt and repeat. One RTX 3090; targets as in Table 2.

Setup. Our main results run on one RTX 3090 (24 GB). To test how the planner carries across hardware, we repeat the Qwen2.5-14B cells, unchanged, on a Tesla V100 (16 GB) and a Titan V (12 GB), on which the kernel switch of Section 3.2 raises cost rather than lowering it. The targets are Qwen2.5-14B-Instruct (Qwen Team, 2024) with the Qwen2.5-0.5B-Instruct drafter (all GPUs) and Qwen3-14B (Qwen Team, 2025) with Qwen3-0.6B (RTX 3090), plus Qwen2.5-32B-Instruct with Qwen2.5-0.5B-Instruct for the padding ablation, all quantized to Q4_K_M. *Tree* cells allow up to 16 sequences per verify batch and 12 children per node; *chain* cells verify one sequence.

Teacher forcing. A speculative run's text can diverge from the baseline's at a near-tie between two tokens, after which the two predict different text. To compare planners on identical text, every speculative run is teacher-forced onto its baseline's output, the target's own greedy continuation. llama.cpp decodes greedily as well, so all three systems see the same text except after a near-tie.

Baselines. Our engine includes a reproduction of CAST's admission logic (Hong et al., 2026), with the same drafter, cost measurements and geometry as ours, at its published thresholds 2.5, 3 and 4. Our planner, the three CAST thresholds and a one-row (non-speculative) baseline run back to back in one run per cell, so they share one baseline and GPU drift between runs cancels out of every comparison between them. We also run llama.cpp's own speculative decoding (`llama-server`) over the same prompts at draft lengths $n_{\max} \in \{2, 4, 8, 12, 16\}$, in a separate run; it drafts chains only, so its numbers are the same for both geometries.

Metric. We report each system's speedup over its own one-row baseline. The three prompts of a workload differ far more than the planners do, so we compare our planner with CAST run by run (same prompt, same repeat) and report the geometric mean of the six paired ratios with the number of runs ours wins; a cell within 2% of parity is a tie. Absolute throughputs are in Table 5 (Appendix A.1.4).

6.2 Results

Table 2 and Figure 2 give the main results. Across its 9 cells, our planner reaches a geometric-mean speedup of 1.78$\times$ over plain decoding with one configuration, never tuned per workload.

Against llama.cpp. Even with llama.cpp tuned per cell in hindsight, our untuned planner is faster on 8 of 9 cells, and 14.1% faster in geometric mean. The exception is JSON, where llama.cpp's longest draft ($n_{\max}$=16) reaches 3.79$\times$ against our 3.64$\times$. That draft length is llama.cpp's worst on prose (0.69$\times$), and within a cell its worst setting falls up to 58% below our planner (Figure 2(a)).

Table 2: Speedup over each system's own one-row baseline on the RTX 3090. *Ours*: one untuned configuration. For CAST (our in-engine reproduction) and llama.cpp (`llama-server`): the best setting per cell in hindsight (n for llama.cpp) and the range over all settings. *Paired*: ours over the best CAST, run by run (6 runs), with the runs ours wins.

				CAST			llama.cpp	
Target	Workload	Geom.	Ours	best	range	paired	best (n)	range
Qwen2.5-14B	Prose	tree	1.56×	1.59×	1.26–1.59	−1.1% (4/6)	1.32× (4)	0.69–1.32
	Prose	chain	1.63×	1.50×	1.36–1.50	+10.2% (2/6)	1.32× (4)	0.69–1.32
	Tech Q&A	tree	1.43×	1.39×	1.36–1.39	+3.3% (4/6)	1.42× (8)	0.91–1.42
	Tech Q&A	chain	1.63×	1.65×	1.43–1.65	−1.5% (3/6)	1.42× (8)	0.91–1.42
	Math	tree	2.48×	2.33×	2.30–2.33	+6.6% (6/6)	2.20× (16)	1.64–2.20
	Extraction	tree	1.73×	1.70×	1.63–1.70	+1.0% (4/6)	1.52× (8)	1.33–1.52
	JSON	tree	3.64×	3.11×	3.02–3.11	+17.2% (6/6)	3.79× (16)	1.69–3.79
Qwen3-14B	Prose	chain	1.52×	1.56×	1.32–1.56	−3.5% (2/6)	1.09× (2)	0.76–1.09
	Tech Q&A	tree	1.30×	1.27×	1.18–1.27	+2.2% (5/6)	1.15× (2)	0.65–1.15

Against CAST. Our planner is 3.7% faster in geometric mean than CAST tuned per cell in hindsight, and 4.4% faster than CAST 2.5, its strongest single threshold. Run by run (Figure 2(b)), ours wins every run on JSON (+17.2%) and math (+6.6%), five of six on Qwen3 technical Q&A (+2.2%), and four of six on Qwen2.5 technical Q&A in the tree geometry (+3.3%). On Qwen2.5 prose (chain) it is +10.2%, all from one prompt (+38–49%), and 2–4% behind on the other two. Extraction (+1.0%), Qwen2.5 prose (tree, −1.1%) and Qwen2.5 technical Q&A (chain, −1.5%) are ties, and CAST wins Qwen3 prose (chain) by 3.5%. CAST's best threshold changes from cell to cell (2.5 on five cells, 3 on math, tree prose and tree technical Q&A, 4 on JSON), and its worst falls up to 19% below ours.

Where the gains come from. On math our planner verifies smaller trees than CAST at its best threshold (7.5 against 15.7 nodes per step) and commits fewer tokens per step (6.53 against 7.26), but each step takes 16% less time (37.5 against 44.5 ms). On JSON it verifies a narrower, deeper tree (14.9 nodes at mean depth 13.9, against 25.3 at 11.9) and commits 16% more tokens per step (14.55 against 12.55) for a step of the same cost (57.0 against 57.5 ms).

Where the losses come from. On Qwen3 prose (chain), our planner drafts deeper than CAST 2.5 (2.53 against 1.88 levels per step) and commits 2% more tokens per step, but the extra drafter forwards make each step 5% more expensive (22.2 against 21.1 ms). On Qwen2.5 technical Q&A (chain) both draft equally deep on average (2.42 against 2.44 levels) and our steps are 3% cheaper, but CAST commits 5% more tokens per step (2.69 against 2.57): ours spends its depth on steps where it pays off less often.

Across GPUs. Table 3 compares the two planners on all 23 cells over three GPUs, against each fixed CAST threshold and the per-cell best (per-cell Volta results are in Table 6, Appendix A.1.4). No CAST threshold is best everywhere (2.5 on 12 cells, 3 on 7, 4 on 4), and a deployed CAST must commit to one. Against its strongest fixed threshold, 2.5, our untuned planner is at parity (+0.5%); against 3 and 4 it is +2.5% and +10.7%. Only an oracle that retunes CAST for every cell is ahead, by 0.3%. In no cell do we fall more than 7.2% below any CAST threshold, while CAST's worst threshold falls up to 22.7% below us.

Table 3: Ours against CAST: geometric mean over cells of the paired ratio, with the cells ours wins. *Best*: CAST's threshold picked per cell in hindsight. *Worst case*: our largest shortfall against any threshold in any cell, and that of CAST's worst threshold against ours.

			Ours vs CAST at threshold				Worst case	
GPU	Cells	Ours	2.5	3	4	best	ours	CAST
RTX 3090	9	1.78×	+4.4% (7/9)	+8.2% (7/9)	+14.4% (9/9)	+3.7% (6/9)	−3.5%	−21.2%
Titan V	7	1.52×	−2.7% (1/7)	−1.6% (3/7)	+6.7% (5/7)	−3.6% (1/7)	−7.2%	−18.1%
V100	7	1.01×	−1.0% (3/7)	−0.5% (2/7)	+10.0% (5/7)	−2.0% (2/7)	−6.9%	−22.7%
All	23	1.43×	+0.5% (11/23)	+2.5% (12/23)	+10.7% (19/23)	−0.3% (9/23)	−7.2%	−22.7%

The comparison is not uniform across GPUs. On the RTX 3090 our planner is ahead of every fixed threshold in geometric mean; on the Volta GPUs it is 2.0%–3.6% behind CAST's per-cell best and behind CAST 2.5 on most cells; its largest per-cell shortfall is 7.2%. The losses include tree math and JSON, where our trees are shallower (4.7–5.1 tokens per step on tree math, against 5.8–6.4 for CAST's best threshold), and chain prose; on tree prose, ours is ahead on both. On the V100, where verification is expensive (Figure 1(b)), every planner is near or below $1\times$ on prose, technical Q&A and extraction.

6.3 Ablations

Table 4: Ablations, each run in the same run as our planner, pooled over every cell we ran (per cell: Table 7). Ours over the ablation, run by run: geometric mean and median over all runs, runs ours wins, and the range of the per-cell geometric means.

Req.	Ablation	Target	Cells	Mean	Median	Wins	Per-cell range
R1	per-tree ratio R/L	Qwen2.5-14B	2	+1.1%	+1.4%	9/12	+0.9% to +1.3%
R2	no padding past cliff	Qwen2.5-32B	2	−4.6%	−0.2%	5/12	−8.2% to −0.8%
R2	no padding past cliff	Qwen2.5-14B	6	−0.2%	−0.8%	15/36	−7.4% to +5.0%
R3	raw confidences	Qwen3-14B	3	+2.3%	+0.3%	9/18	−0.6% to +7.1%
R3	raw confidences	Qwen2.5-14B	2	−0.1%	+0.1%	6/12	−0.8% to +0.5%
R4	fixed 8 levels	Qwen2.5-14B	3	+18.8%	+11.6%	15/18	+4.5% to +46.8%

Table 4 removes one requirement at a time, pooled over every ablation cell we completed (per cell: Table 7, Appendix A.1.4). Pricing each drafter level (R4) matters most: drafting a fixed eight levels on prose is slower on all three GPUs, by 9.2% on the RTX 3090, 4.5% on the Titan V and 47% on the V100, where it runs at half the speed of plain decoding. Calibration (R3) helps only where the drafter is miscalibrated, and modestly: on Qwen3 prose the median run gains 3.5% on the RTX 3090, 1.3% on the Titan V and nothing on the V100, and on Qwen2.5, whose drafter is already calibrated, it changes nothing, as intended. The run-level objective (R1) is small at these rates: a per-tree ratio costs 0.9–1.3%, inside our tie band. Padding past the cost cliff (R2) is GPU-dependent and does not pay overall: it gains 3–5% on the Titan V, loses 4.5–8.2% on JSON on the RTX 3090 and V100, and ties elsewhere; we have not isolated why.

7 Conclusion

With one untuned configuration, SpecPilot is 14.1% faster than llama.cpp and 3.7% faster than CAST on an RTX 3090, each tuned per cell in hindsight, and across three GPUs it matches CAST's best fixed threshold. It trails CAST on the two Volta GPUs, where its trees on math and JSON are too shallow, and our results use one drafter family per target and teacher-forced workloads.

AI use statement

In this work, we used generative AI tools for implementing methods, designing and running experiments under the authors' direction, proposing and refining hypotheses about the causes of observed

behavior, providing feedback on experiments, formulating mathematical claims, supporting data analysis, interpreting results, and editing the text of the paper. We have not used generative AI tools for assisting in the writing of proofs, and assisting with translation is not applicable to this work. Additionally, we used generative AI tools for generating diagrams, figures, and data analysis scripts. We have reviewed all AI-assisted work. Code was manually reviewed for correctness, and invariant checks were manually inserted and verified. Analysis scripts were likewise validated through manual code review, and every number in the paper is generated from the experiment records by those scripts. Mathematical formulations were author-guided and never generated wholesale, and were checked against existing work. All AI-drafted text was reviewed, revised, and approved by the authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. Sequoia: Scalable, robust, and hardware-aware speculative decoding. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Werner Dinkelbach. On nonlinear fractional programming. *Management Science*, 13(7):492–498, 1967. doi: 10.1287/mnsc.13.7.492. URL https://doi.org/10.1287/mnsc.13.7.492.

Georgi Gerganov and ggml-org contributors. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017.

Yinrong Hong, Zhiquan Tan, and Kai Hu. Inference-cost-aware dynamic tree construction for efficient inference in large language models, 2026. URL https://arxiv.org/abs/2510.26577.

Tao Jin, Phuong Minh Nguyen, and Naoya Inoue. Goose: Anisotropic speculation trees for training-free speculative decoding, 2026. URL https://arxiv.org/abs/2604.02047.

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

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024.

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

Gabriele Oliaro, Zhihao Jia, Daniel Campos, and Aurick Qiao. Suffixdecoding: Extreme speculative decoding for emerging ai applications, 2025. URL https://arxiv.org/abs/2411.04975.

Qwen Team. Qwen2.5 technical report, 2024. URL https://arxiv.org/abs/2412.15115.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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

William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Lifu Wang and Pan Zhou. Smart: When is it actually worth expanding a speculative tree?, 2026. URL https://arxiv.org/abs/2604.09731.

A Appendix

A.1 Extended Motivation

A.1.1 Cost Units and the Rate of a Run

We measure all costs in *base forwards*. One base forward is the wall-clock time of the target's forward pass over a single row (the next token) at the current context, excluding the reduction of its logits: the forward that plain autoregressive decoding performs once per token. We estimate it online as the median duration of the target's own one-row forwards over a sliding window of recent steps, so it tracks the machine, the model, and the growing context without calibration. A verify pass over a tree S has $|S|+1$ rows (the drafted nodes plus the next token) and costs $1+V(S)$ base forwards, where $V(S)\geq 0$ is its cost above the base forward. A step's total cost, measured by wall clock from the end of one step to the end of the next, also includes logit reduction, drafting, planning, and host work, so even plain decoding costs slightly more than one base forward per token. After n steps, a run has committed $T=\sum_{i=1}^{n} R_i$ tokens at a total cost of $C=\sum_{i=1}^{n} L_i$ base forwards, where R_i is the number of tokens step i committed (its accepted drafts plus the target's own next token) and L_i is its total cost. We call T/C the *rate* of the run; speedups are reported against the rate of plain decoding measured in the same run.

A.1.2 Why Each Tree Must Be Judged Against the Run

Judging each tree by its own return and cost is the wrong objective. Suppose that at some point during inference we have committed $T = 150$ tokens and paid for them with $C = 100$ base forwards, a rate of 1.5. Suppose now that the planner considers a tree with expected return $R = 7$ tokens and predicted cost $L = 6$ base forwards. Judged on its own, the tree looks profitable: it returns more than it costs, so it is admitted. But even if it pays out exactly as predicted, our overall rate falls to $157/106 = 1.48$, because the tree earns less per base forward than the run already does. If it costs more than predicted, which Section 3.4 shows is the common case, the loss is larger: a realized cost of 9 base forwards drops the rate to $157/109 = 1.44$. In essence, we have risked our already gained speedup on a bet whose odds were worse than the ones we were already getting. Each speculative step is such a bet, and the stake grows with the speedup already banked. The right objective is therefore not any single tree's return but the rate of the run as a whole, $\sum_i R_i / \sum_i L_i$, and the bar a tree must clear is the rate the run has actually achieved so far. A planner that measures that rate from its own estimates rather than wall clock goes wrong in practice: in one of our early runs, such a rate settled at 1.79 while the run's actual rate was 1.32.

A.1.3 Calibration and Path Correlation

An estimate $\hat{p}$ of an event E is *calibrated* if $\Pr[E \mid \hat{p} = p] = p$ for every p: among all cases assigned probability p, the event occurs a fraction p of the time. We measure calibration by binning estimates into deciles and comparing each bin's mean estimate with its realized frequency. For edge estimates the event is conditional: E is "x_d is accepted", and only edges whose parent was accepted are counted. Under greedy verification the outcome of each edge is determined once its prefix is fixed; probabilities are the planner's, taken over its uncertainty about the target before verification.

The chain rule gives $J(v) = \prod_{j \le d} q(v_j)$ exactly, but the planner never observes q on a particular path. Even a perfectly calibrated confidence gives only its expectation, $\mathbb{E}[q \mid c]$, averaged over every edge that received the same confidence. Multiplying these expectations along a path is exact only if the edges are uncorrelated given their confidences; for two edges,

$$\mathbb{E}[q_1 q_2 \mid c_1, c_2] = \mathbb{E}[q_1 \mid c_1]\,\mathbb{E}[q_2 \mid c_2] + \mathrm{Cov}(q_1, q_2 \mid c_1, c_2).$$

When the covariance is positive, the product is biased low, and the bias compounds with depth.

The covariance term is what a mixture over chain regimes would model. We fit a two-regime mixture of this form online. On most of our configurations it finds no separation; where it does separate, it gives the locked regime a prior of only a few percent, and on every configuration it moves the depth-8 joint by at most 0.2%. Acceptance rises with depth, but the drafter's own confidence rises with it at the same rate, so the covariance given the confidences is negligible. Measured directly on the 9 cells of Table 2 (48–204 reached depth-8 paths per cell), the product of calibrated edges predicts realized depth-8 path acceptance within 7% on five cells and within 10% on seven, with a median realized-to-predicted ratio of 1.06. The worst cell, Qwen2.5-14B Tech Q&A (tree), is 25% low over 48 paths, and already 14% low at depth 4 over 364 paths, so its gap is not only sampling noise. In earlier versions of our planner whose edges were biased low, the same product was 1.2–2.3× low while both a confidence prior and a per-context miss rate deflated each edge, and 1.1–1.3× low with the miss rate alone (Section 3.3).

The conditioning population matters as much as the estimator. Under greedy verification an edge whose parent was rejected is never tested; counting it as a miss treats an untested edge as a failed one. On our drafter this deflates every calibrated estimate by about 35% relative to the reached-only fit.

A.1.4 Additional Measurements

Draft length. On prose, raising $n_{\max}$ from 4 to 16 quadruples the drafted tokens while the number accepted stays flat, and the kept fraction falls from 52% to 12%. On JSON each extra token is nearly free, since at least 93% of drafted tokens are kept at every draft length.

Verify cost. On the RTX 3090, trees of six and seven nodes are strictly dominated by a larger one, for both Qwen2.5-14B and Qwen2.5-32B: a nine-row tree is both cheaper and offers more drafted

nodes. Past the kernel switch, cost is nearly flat on the RTX 3090, while both Volta GPUs take a second, smaller step at 17 rows (11–13%). The switch happens at the same row count on every GPU we tested; only the direction of the jump differs.

Absolute throughput. Table 5 gives the committed tokens per second behind every speedup in Table 2, with each engine's own one-row baseline.

Table 5: Absolute throughput (committed tokens/s), same cells as Table 2. Baselines are each engine's own one-row decode. Ours/CAST and llama.cpp were measured in different runs; GPU throughput drifts by up to about 15% between runs, which is why speedup over each engine's own baseline is the primary comparison.

Target	Workload	Geom.	Ours base	Ours	CAST (best)	llama.cpp base	llama.cpp (best)
Qwen2.5-14B	Prose	tree	67.5	105.1	107.0	74.9	98.7
Qwen2.5-14B	Prose	chain	69.3	112.8	103.7	74.9	98.7
Qwen2.5-14B	Tech Q&A	tree	68.5	97.9	95.0	75.5	106.9
Qwen2.5-14B	Tech Q&A	chain	66.7	108.8	110.3	75.5	106.9
Qwen2.5-14B	Math	tree	70.0	174.0	163.1	73.2	161.0
Qwen2.5-14B	Extraction	tree	69.9	120.8	118.4	74.2	112.3
Qwen2.5-14B	JSON	tree	70.1	255.5	218.4	75.1	285.0
Qwen3-14B	Prose	chain	73.0	110.9	113.9	73.3	79.9
Qwen3-14B	Tech Q&A	tree	70.8	91.7	89.7	77.5	89.4

Volta GPUs. Table 6 gives the per-cell results behind the Titan V and V100 rows of Table 3. We ran llama.cpp's reference on one Volta cell, Titan V prose, where its best draft length ($n_{\max}$=4, 1.34×) is faster than our planner.

Table 6: Qwen2.5-14B with the Qwen2.5-0.5B drafter on the two Volta GPUs: speedup over the run's own one-row baseline for ours and each CAST threshold, all in one run per cell. *Paired* is ours over CAST's best threshold for the cell, run by run (6 runs), with the number of runs ours wins.

GPU	Workload	Geom.	Ours	CAST 2.5	CAST 3	CAST 4	Paired vs best
Titan V	Prose	tree	1.13×	**1.11**×	1.10×	1.05×	+2.3% (3/6)
	Prose	chain	1.24×	**1.29**×	1.21×	1.19×	−3.9% (1/6)
	Tech Q&A	tree	1.34×	1.37×	**1.41**×	1.17×	−4.6% (0/6)
	Tech Q&A	chain	1.53×	**1.56**×	1.46×	1.26×	−1.8% (2/6)
	Math	tree	1.84×	1.92×	1.91×	**1.92**×	−4.2% (0/6)
	Extraction	tree	1.42×	1.46×	**1.50**×	1.43×	−5.7% (0/6)
	JSON	tree	2.55×	2.71×	**2.77**×	2.39×	−7.2% (1/6)
V100	Prose	tree	0.77×	**0.71**×	0.69×	0.69×	+9.5% (6/6)
	Prose	chain	0.91×	0.95×	0.95×	**0.97**×	−5.9% (0/6)
	Tech Q&A	tree	0.82×	**0.81**×	0.79×	0.70×	+1.9% (2/6)
	Tech Q&A	chain	0.97×	0.97×	1.00×	**1.01**×	−4.3% (0/6)
	Math	tree	1.15×	**1.23**×	1.22×	1.03×	−6.9% (0/6)
	Extraction	tree	0.91×	0.93×	**0.93**×	0.72×	−4.1% (3/6)
	JSON	tree	1.79×	**1.87**×	1.84×	1.53×	−3.4% (1/6)

Ablations. Table 7 gives every ablation cell behind Table 4.

Table 7: Every ablation cell, each in the same run as our planner. *Paired*: ours over the ablation, run by run (6 runs): geometric mean, median run, and runs ours wins.

							Paired	
Req.	Ablation	Target	Workload	GPU	Ours	Ablated	mean	median (wins)
R1	per-tree ratio R/L	Qwen2.5-14B	Math (tree)	RTX 3090	2.54×	2.51×	+0.9%	+1.0% (3/6)
R1	per-tree ratio R/L	Qwen2.5-14B	JSON (tree)	RTX 3090	3.68×	3.63×	+1.3%	+1.4% (6/6)
R2	no padding past cliff	Qwen2.5-32B	Tech Q&A (tree)	RTX 3090	1.54×	1.56×	−0.8%	+0.2% (4/6)
R2	no padding past cliff	Qwen2.5-32B	JSON (tree)	RTX 3090	3.94×	4.32×	−8.2%	−3.2% (1/6)
R2	no padding past cliff	Qwen2.5-14B	Math (tree)	RTX 3090	2.44×	2.35×	+3.1%	−0.4% (3/6)
R2	no padding past cliff	Qwen2.5-14B	JSON (tree)	RTX 3090	3.59×	3.76×	−4.5%	−2.6% (0/6)
R2	no padding past cliff	Qwen2.5-14B	Math (tree)	Titan V	1.91×	1.85×	+3.3%	+3.1% (4/6)
R2	no padding past cliff	Qwen2.5-14B	JSON (tree)	Titan V	2.64×	2.51×	+5.0%	+4.8% (4/6)
R2	no padding past cliff	Qwen2.5-14B	Math (tree)	V100	1.10×	1.10×	−0.2%	+0.3% (3/6)
R2	no padding past cliff	Qwen2.5-14B	JSON (tree)	V100	1.63×	1.75×	−7.4%	−7.2% (1/6)
R3	raw confidences	Qwen3-14B	Prose (chain)	RTX 3090	1.56×	1.49×	+7.1%	+3.5% (4/6)
R3	raw confidences	Qwen3-14B	Prose (chain)	Titan V	1.14×	1.13×	+0.6%	+1.3% (4/6)
R3	raw confidences	Qwen3-14B	Prose (chain)	V100	0.80×	0.81×	−0.6%	−0.3% (1/6)
R3	raw confidences	Qwen2.5-14B	Prose (chain)	Titan V	1.25×	1.26×	−0.8%	+0.1% (3/6)
R3	raw confidences	Qwen2.5-14B	Prose (chain)	V100	0.90×	0.89×	+0.5%	−0.3% (3/6)
R4	fixed 8 levels	Qwen2.5-14B	Prose (tree)	RTX 3090	1.50×	1.35×	+9.2%	+11.6% (4/6)
R4	fixed 8 levels	Qwen2.5-14B	Prose (tree)	Titan V	1.14×	1.09×	+4.5%	+5.2% (5/6)
R4	fixed 8 levels	Qwen2.5-14B	Prose (tree)	V100	0.77×	0.52×	+46.8%	+48.2% (6/6)

A.2 Method Details

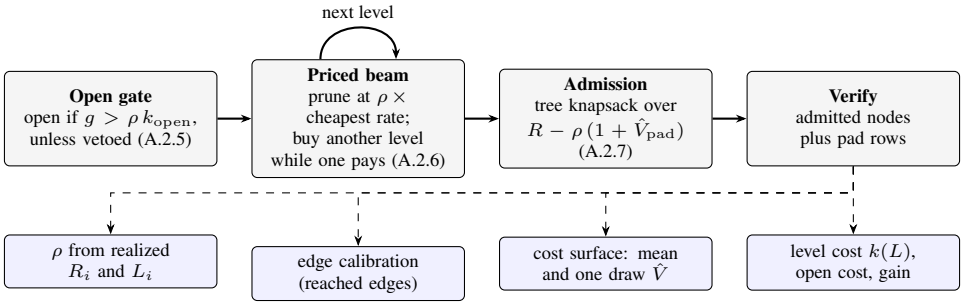


Figure 3: One planning step (top row) and what the verified step feeds back (bottom row, dashed). The beam prices drafter levels on the cost surface's mean and admission prices trees on one draw per step; both judge against the realized rate ρ, which is fixed for the step.

A.2.1 Base Forward and Measured Cost

The base forward's time is the median of the run's single-row target forwards over the most recent 512 forwards. Because logit reduction is excluded, plain decoding costs slightly more than one base forward per token; all reported speedups are measured against plain decoding itself. A step's measured cost L_i runs from the end of one step to the end of the next, so it includes drafting, planning, verification, logit reduction, and host work. ρ is updated only once the base forward has been measured. Steps with an empty tree are also recorded, so a run of declined steps pulls ρ back toward 1.

A.2.2 Geometry

Chain and tree runs use the same planner. A chain has width limit 1, so each level extends a single lane. A tree allows up to 12 alternatives per lane and up to eight lanes, under a sequence budget of 16. Each lane reports the drafter's top-k tokens with k equal to the width limit. The lane's most likely token is its *chain* token, and the others are *alternatives*.

A.2.3 EDGE CALIBRATION AND COVERAGE

The drafter's confidence $c(v)$ is its full-vocabulary softmax probability at temperature 1. Confidences fall into twenty buckets, by decile and by whether the token is a chain token or an alternative. For $c < 1/2$,

$$\tilde{c} = c\,\frac{H_b + k\bar{c}_b}{\sum_{u\in b} c(u) + k\bar{c}_b},$$

where H_b counts accepted tokens among the reached tokens of bucket b, the sum runs over the same tokens, $\bar{c}_b$ is their mean confidence, and $k = 2$ pseudo-observations shrink a sparse bucket toward the raw confidence. For $c \geq 1/2$ the same correction is applied to $1 - c$ using misses. Counts decay so the calibration follows the run. Only tokens whose parent was accepted are counted: an unreached token's outcome says nothing about its own edge, and counting it as a miss deflates every edge. This bucket correction is the drafter's only calibration. A finer re-estimate, per recent context and token, pools tokens of different confidence in one estimate and so pulls each toward their mean, lowering the drafter's most confident edges; we do not apply one to the drafter.

The fan coverage at branch point u is

$$Z(u) = \sigma\Big(\ln \textstyle\sum_i \frac{\tilde{c}_i}{1-\tilde{c}_i} + a_d(d) + a_w(W)\Big), \qquad W = \frac{(\sum_i \tilde{c}_i)^2}{\sum_i \tilde{c}_i^2}.$$

The log of summed odds responds far more to one confident child than to many weak ones. W, the effective number of children, varies continuously as near-zero children come and go. The corrections a_d and a_w are piecewise-linear in depth and in W and start at zero, so for a single child $Z = \tilde{c}_1$. They are fit online by an interval-censored logistic update against the target probability mass the fan actually covered, read from the target's output at u. At admission, Z is drawn from its fit together with the cost surface, from the same seed.

A.2.4 COST SURFACE

A changepoint between adjacent row counts is accepted only if all of the following hold:

- the step in log cost is at least 3σ of the residual noise and at least 0.05;
- each side's scatter is below half the step;
- the two-line residual sum of squares is at most 0.1 of the one-line fit's;
- each resulting region holds at least three distinct row counts.

Discovery runs up to four rounds per refit. Per-sequence and per-depth terms are added to the fit only once the run's forwards show evidence for them. Before any forward is measured, the surface is $T(r) = 1 + 0.3(r - 1)$ single-row forwards with a spread of $\times 2$.

Each Thompson draw perturbs a measured region's level and its slope about the region's centroid. Perturbations are Student-t with $\nu = 3$ until the per-forward noise has been measured, and Gaussian after that. An unmeasured region takes a Brownian-bridge prior between its measured neighbours, so its uncertainty grows with its distance from evidence. Each draw is truncated so that a tree of r rows costs between one and r single-row forwards.

The padding budget P is bounded by the allocation ceiling and by how deep the tree could still pay. Padding extends the deepest path and never widens the tree. Admitted pad rows are filled first with nodes the admission declined and then with filler tokens. Admission prices trees on one draw of this surface per step, and the beam prices drafter levels on its mean (Appendix A.2.6).

A.2.5 OPEN GATE AND REALIZED-RATE VETO

Opening the drafter costs k_{open} base forwards, the measured time to bring its cache up to date. The drafter opens when $g > \rho\, k_{\text{open}}$, where g is the mean increase in $R - \rho\hat{L}$ over recent steps on which it was opened while warm. It also opens while k_{open} or g is still unmeasured. A declined drafter is re-tested once its newest gain sample is 16 steps old.

Because the gain is itself a belief, a realized-rate veto can override an open. Opened and declined steps are kept in paired rings of 32 steps, with at least 8 required and a 64-step expiry. Each arm's

rate is $\hat{r} = \sum R_i / \sum L_i$, with delta-method standard error

$$\mathrm{se}^2 = \frac{\sum_i (R_i - \hat{r} L_i)^2}{n(n-1)\,\bar{L}^2}.$$

The veto fires when $\hat{r}_{\mathrm{open}} + 2\,\mathrm{se}_{\mathrm{open}} < \hat{r}_{\mathrm{decl}} - 2\,\mathrm{se}_{\mathrm{decl}}$. An unmeasured declined arm counts as plain decoding (rate 1). The veto lapses once its opened evidence is more than 16 steps old, and it never overrides a re-test.

A.2.6 Beam Search

The first level comes with the open at no extra cost. Each later level extends the kept frontier, best path probability first, up to eight lanes. The beam does not yet know the admitted tree's shape, so it prices growth as a chain of n nodes on the cost surface's mean, $\bar{V}(n)$, at the step's context length; the admission DP then reprices the shapes it considers exactly, on the step's draw $\hat{V}$. The prune uses the cheapest forward rate rather than the next node's marginal cost because, on a surface whose dearest row comes first, the marginal would refuse every tree; the stop rule uses the cumulative cost $\bar{V}(n+r) - \bar{V}(n)$ because every level it buys is a drafter forward actually paid. Both look no further than the largest tree size the run has measured: beyond it the surface is an extrapolation, and an invented cheap region must not talk a price down. The stop rule still prices at least one further level. The level cost $k(L)$ is the median over recent levels with L lanes, timed from the request to the end of that level's pricing and divided by the base forward. The stop rule prices every future level like the last one. Its projection for each depth is scaled by the ratio of the mean best path probability the next level actually kept to the mean projected one, over the last 128 continuations at that depth. The beam also stops at the depth limit (16) or the sequence budget, or when a level keeps nothing. While any level cost is unmeasured, the beam continues.

A.2.7 Admission

Each node's table maps (nodes, leaves, depth) of its subtree to the largest achievable $\sum J$, and children are folded in one at a time under the width limit. A virtual root folds the depth-1 nodes, and each root state is priced as a tree of $n{+}1$ rows. States that need more sequences than the budget allows, or more rows than the sampled surface says could pay at that depth, are never built. Ties go to the smaller tree. The DP is pseudo-polynomial in these bounds, and fast in practice: per step it takes 0.18–0.65 ms on the RTX 3090's host and 0.21–0.56 ms on the Titan V's, rising with the admitted tree from about 2 to 15 nodes, at most 1.2% of a step's median wall-clock time. The V100's host CPU is slower (0.46–1.56 ms, at most 3.8% of a step).