CADENCE: REGULATING THE SPECULATION EXTERNALITY IN LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Speculative decoding (SD) speeds up LLM inference by verifying draft tokens in parallel. SD applies to every request, but its effectiveness, how many draft tokens are accepted, varies across requests and over time. Requests in a batch share each step, so chasing mean throughput can slow requests that accept little: in one step at batch 112, SGLang's default SD (SGL-SD) raises throughput 13% while 26% of requests run slower than autoregressive (AR) decoding. CADENCE, our per-step controller, caps each request's delay, its time lost relative to AR steps. Every step commits at least one token per request, so the caps reduce to one allowed step latency, known before drafting. Within this latency, an all-or-none floor gives every request a draft token whenever the latency allows, and CADENCE picks the verification budget that an operator's utility scores highest. On Qwen3-32B under high load, CADENCE improves goodput by 1.29× over AR decoding and by 1.80× over SGL-SD, while 99.5% of its requests run no slower than AR decoding, against 69% under SGL-SD.

1 INTRODUCTION

Speculative decoding (SD) (Leviathan et al., 2023; Chen et al., 2023) speeds up large language model (LLM) inference without sacrificing model quality. In a draft-then-verify loop, a small drafter proposes draft tokens that the target model checks in one verification pass. The target model keeps those that rejection sampling accepts, then samples a bonus token. Better drafters (Li et al., 2025) raise acceptance, and adaptive verification (Liu et al., 2026b; Cheng et al., 2026) uses the drafter's confidence to set how many draft tokens each request verifies. An isolated request gets these gains almost free: verifying several draft tokens on spare compute takes about as long as producing one.

This advantage fades inside a shared server, and what speculation costs the other requests in a batch has received less attention. Under continuous batching (Yu et al., 2022), every request in the batch advances in the same decode step, and an autoregressive (AR) step gives each request exactly one token. A speculative step has a draft phase, which lasts as long as the batch's deepest draft, and a verification phase. Once a large batch has used up the spare compute, each draft token the batch verifies can lengthen that phase for every request. A request whose draft tokens are accepted commits several tokens in one step. A request whose draft tokens are all rejected waits through both phases for the single token an AR step would have given it. Every request pays the step's latency beyond one AR step, which we call the *speculation externality*. **The cost of speculation is socialized, but the benefit of acceptance is privatized.** Section 2 shows that at large batch sizes, verifying every draft token can speed up some requests while slowing others, and that one request's added verification can cost an unchanged co-batched request extra time.

Throughput masks these losses, and no simple policy bounds them and keeps the gains. Turning speculation off gives up every gain, while full-draft speculation, which verifies every request's full draft, slows requests whose draft tokens are rejected. A fixed limit on verified tokens helps only when tuned to the runtime, the request mix and the batch size, which moves with load. Even a throughput-maximizing allocation makes the least promising requests wait through both phases. A per-step controller should instead cap each request's *delay*, its time lost relative to AR steps, and let the operator decide who receives speculation within the cap. Each cap is the time AR steps would take to commit the request's tokens, scaled by the operator's delay tolerance, plus a fixed arrival allowance.

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.

Such a controller seems to need to know acceptance in advance. It must check each cap before drafting, but learns which tokens were rejected only after verification, once the draft phase is spent. One property of SD removes this need. **Every step commits at least one token per request, so the most delay a step can add to any request is known before it runs: its latency beyond one AR step.** The delay caps therefore reduce to one *allowed step latency*, set by the request closest to its cap. Up to the step's timing overshoot, no step within this latency can push a request past its cap, whichever draft tokens are accepted.

CADENCE, our controller, regulates speculation within this latency. Before drafting, it admits every *verification budget* (a draft depth and how many draft tokens to verify) that fits this latency. Whenever this latency allows, an all-or-none *floor* gives every request its first draft token, so no request waits through a speculative step with nothing to verify. CADENCE fills the rest with the draft tokens of highest expected benefit under one shared acceptance curve, and picks the budget an operator's utility scores highest. The allowed step latency grows while every request is far from its cap and shrinks as one nears it, down to an AR step. AR steps move every request away from its cap when the delay tolerance exceeds their relative timing overshoot.

We make two contributions:

- We characterize the speculation externality by modeling a speculative step as a draft phase and a verification phase, each relative to the AR step, and measure who gains and who loses. At batch 112, SGLang's default SD (SGL-SD) raises throughput 13% while 26% of requests run slower than AR decoding (Section 2.3).
- We design CADENCE, which turns the delay caps into one allowed step latency and picks within it the budget with the highest utility (Section 3). Under high load with Qwen3-32B and EAGLE-3, CADENCE improves goodput, the throughput of requests no slower than AR decoding, by 1.80× over SGL-SD and 1.29× over AR decoding, and cuts the share of requests slower than AR decoding from 31% to 0.5%, with none over its cap (Section 4).

Speculation in a batch is a shared resource that CADENCE regulates step by step.

Figure 1 and our evaluation serve Qwen3-32B with its EAGLE-3 drafter (Li et al., 2025) on one H200 in SGLang with its defaults. They use greedy chain drafting, in which the drafter proposes its likeliest token at each position.

2 CHARACTERIZING THE SPECULATION EXTERNALITY

SD applies to every request and raises throughput, but its effectiveness, how many draft tokens the target model accepts, varies across requests and over time. Requests share each step, so this variation becomes a disparity: some requests gain far more than others, and in a large batch some run slower than they would without SD. We show the gain (Section 2.1), the variation (Section 2.2) and the losses that mean throughput hides (Section 2.3).

2.1 SD RAISES THROUGHPUT

In a speculative step, every request drafts to the step's *draft depth*, and the target model verifies the draft tokens in one pass. SD raises throughput at every batch size in Figure 1a, if the draft depth suits the load. With every request verifying its whole draft, the panel compares *decoding speed*, output tokens over decoding time, with AR decoding, which runs no drafter. At batch 32, SD raises it 1.4 to 2.1×, depending on the draft depth. The gain shrinks as the batch grows, and the best draft depth falls from three to one. At batch 64, draft depth seven slows the batch.

A decode step advances all B requests at once. After every step, even an AR step, the drafter runs a *catch-up pass* (draft-extend in SGLang) over every verified position, accepted or not. The pass updates the drafter's cache and yields the next step's first draft token, so we count it in the draft phase. The draft depth $\ell_{\max}$ is at most $\gamma = 7$ in our implementation. Request r's *speculation length* $\ell_r \le \ell_{\max}$ counts its draft tokens the step verifies. The verification pass also covers each request's last committed token, so its *verification batch size* is $K = \sum_{r=1}^{B}(1 + \ell_r) = B + \sum_{r=1}^{B} \ell_r$. The *step latency* $L(B, \ell_{\max}, K)$ spans the whole step, catch-up pass included, and an AR step takes $L_{\mathrm{AR}}(B) = L(B, 0, B)$. A runtime's offline table of step latencies is its *latency profile*. Before the step runs, a scheduler chooses the speculation lengths, the step's *allocation*.

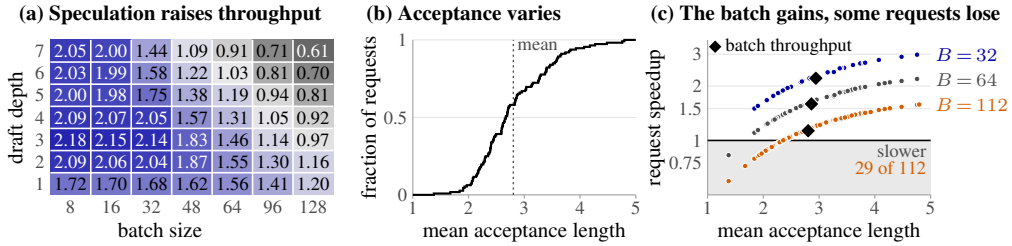


Figure 1: **SD raises throughput, but in a large batch it slows some requests.** Qwen3-32B on one H200 in SGLang with EAGLE-3, acceptance left to the model. Panels (b) and (c) mix chat, code, math and summarization prompts. Speedup is per-step decoding speed over AR decoding, prefill excluded. (a) Speedup when every request verifies its whole draft, by draft depth and batch size. Gray cells, below 1, are slowdowns. (b) Mean acceptance lengths of 112 requests at draft depth 7. (c) Their speedups under SGLang's default SD (SGL-SD: depth three, all three draft tokens verified), with the first 32, 64 and all 112 at batch 32, 64 and 112. Diamonds mark batch throughput, and shading slowdowns.

Every request waits through the whole step, which both phases lengthen. The draft phase chains small drafter passes, so its *drafting latency* grows with the draft depth. The target model's pass sets the *verification latency*, which grows with K on average once a large batch has used up the GPU's spare compute (Williams et al., 2009), and can jump where K crosses a kernel's tile boundary.

A paired intervention shows this cost directly: one request's added verification costs an unchanged co-batched request time. We run it on the H200 in bf16 at batch 48 in deterministic-inference mode, whose batch-invariant matrix multiplication makes step latency jump just past multiples of 128 in K, unlike the default one. Every request drafts to depth seven in both conditions, so only K differs. One request stays unchanged and verifies no draft tokens in either condition. When another request verifies seven more draft tokens in one step, K crosses 129, where step latency jumps, and the unchanged request takes a median of 31.9 ms longer to commit eight tokens. Two controls attribute this extra time to the larger K at the tested jump.

2.2 Effectiveness varies across requests and over time

The step is shared, but the gain is not: each request gains only from its own accepted draft tokens. Request r commits its accepted draft tokens and the bonus token, A_r tokens in all with $1 \leq A_r \leq 1 + \ell_r$. These *acceptance lengths* form the step's *verification outcome*, known only after the step. Acceptance varies widely across requests. Under full-draft speculation at draft depth seven, the mean acceptance lengths of the 112 mixed-workload requests in Figure 1b range from 1.38 to 4.80, with median 2.67. Such spreads appear across datasets and drafters, though narrow for learned drafters on some datasets (Liu et al., 2026c). Acceptance also changes over time: in the same study, EAGLE-3's mean acceptance length drifts along long reasoning outputs. A batch's mean acceptance therefore hides which requests gain little, and when.

2.3 The throughput trap

In a large batch, SD can raise throughput while slowing the requests whose draft tokens are least likely to be accepted. Each waits through a step lengthened for everyone and commits too few tokens to repay it. Figure 1c runs SGL-SD, where every request drafts and verifies three tokens. At batch 32, every request is faster than under AR decoding. At batch 64, throughput is 1.59× that of AR decoding, but one request is slower. At batch 112, throughput still rises 13%, yet 29 of the 112 requests (26%) run slower than AR decoding, the slowest at 0.60× its speed. A request gains only if it commits more tokens per step than the step lasts in steps of AR decoding, and this break-even acceptance rises with the batch: 1.22 at batch 32, 1.66 at batch 64 and 2.31 at batch 112. We call judging speculation by the batch's mean throughput the *throughput trap*. Throughput counts tokens, not who commits them.

We formalize the trap against the AR step, which gives each request the one token any step commits and lets none draft. Its catch-up pass makes it slower than AR decoding, a gap our evaluation reports.

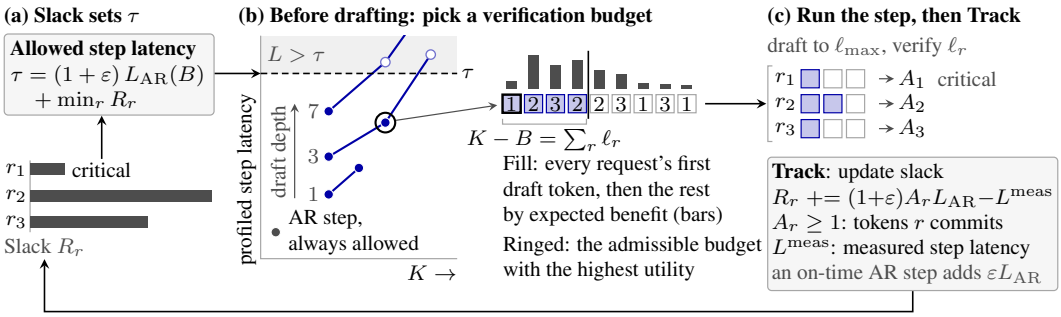


Figure 2: **Once per step, before drafting,** CADENCE **sets the draft depth and every speculation length.** (a) The critical request, with the least slack, sets the allowed step latency τ (drawn at pace $\rho = 1$). (b) CADENCE gives every request its first draft token in each admissible budget, fills the rest by expected benefit, and picks the budget with the highest utility. (c) After the step, `Track` updates slack.

Assuming the verification latency is independent of $\ell_{\max}$ (otherwise CADENCE reads the full latency profile), the drafting and verification latencies are increments over the AR step:

$$\Delta_{\mathrm{D}}(B, \ell_{\max}) = L(B, \ell_{\max}, B) - L_{\mathrm{AR}}(B), \qquad \Delta_{\mathrm{V}}(B, K) = L(B, \ell_{\max}, K) - L(B, \ell_{\max}, B).$$

Their sum is the speculation externality $L - L_{\mathrm{AR}} = \Delta_{\mathrm{D}} + \Delta_{\mathrm{V}}$, a cost every request in the step bears whether or not its own draft tokens pay. The deepest draft sets the drafting latency, and each verified token adds $\Delta_{\mathrm{V}}(B, K+1) - \Delta_{\mathrm{V}}(B, K)$. This is almost nothing within a *flat stretch*, a range of K between two jumps of the latency profile, and a whole jump where the stretch ends.

Measuring the step latency in AR steps gives the *relative step latency*

$$\theta \;=\; \frac{L(B, \ell_{\max}, K)}{L_{\mathrm{AR}}(B)} \;=\; 1 + \eta_{\mathrm{D}} + \eta_{\mathrm{V}},$$

with drafting overhead $\eta_{\mathrm{D}} = \Delta_{\mathrm{D}}/L_{\mathrm{AR}}$ and verification overhead $\eta_{\mathrm{V}} = \Delta_{\mathrm{V}}/L_{\mathrm{AR}}$. Committing A_r tokens would take A_r AR steps, so request r gains if and only if it meets the *break-even condition*

$$A_r \;>\; \theta, \tag{1}$$

that is, if its accepted draft tokens outnumber the two overheads combined. A request that fails this condition loses time. Its delay in the step is the externality net of its own gain,

$$h_r \;=\; L(B, \ell_{\max}, K) - A_r\, L_{\mathrm{AR}}(B) \;=\; \Delta_{\mathrm{D}} + \Delta_{\mathrm{V}} - (A_r - 1)\, L_{\mathrm{AR}}(B),$$

which is negative for a gain.

Processors face a similar problem in speculative execution, which SD generalizes (Leviathan et al., 2023): they throttle speculation that crowds out threads sharing a core or memory system (Tullsen et al., 1996; Ebrahimi et al., 2009) and estimate each thread's slowdown relative to running alone (Ebrahimi et al., 2010). Batched requests are coupled more directly: they advance in lockstep, so each waits through the whole step whether or not its draft tokens are accepted.

Even the allocation that maximizes throughput falls into the trap. At batch 112, it drafts to depth one and verifies 110 draft tokens for 112 requests, raising throughput 33% over AR decoding. The two requests it gives no draft token run at 0.74× the speed of AR decoding.

Avoiding the trap takes two decisions before drafting. How long may the step be, given each request's delay so far? Whose draft tokens should fill it?

3 CADENCE: REGULATING SPECULATION PER STEP

CADENCE makes the two decisions of Section 2.3 once per step (Figure 2). Delay caps set how long the step may be (Section 3.2), and an operator's utility scores whose draft tokens fill it (Section 3.3).

3.1 THE STEP AND ITS TWO DECISIONS

CADENCE picks each step's draft depth, 0 for an AR step or a registered depth such as 1, 3 or 7. A step runs one drafter pass for each draft position after the first, verifies, and ends with the catch-up pass. Under greedy chain drafting, a deeper draft leaves a request's first ℓ_r draft tokens unchanged, so the extra passes only make it wait. The allocation affects the step latency only through the draft depth and K, so a verification budget fixes it in advance. Once the deepest draft has run, its latency is spent and the AR step is no longer available, so CADENCE makes both decisions before drafting.

3.2 SLACK AND THE ALLOWED STEP LATENCY

After a step with profiled latency $L = L(B, \ell_{\max}, K)$ and measured latency L^{meas} commits A_r tokens for request r, CADENCE updates the request's cumulative delay H_r and its AR time T_r^{AR}, the time AR steps would take to commit the same tokens at the profiled AR latency:

$$H_r \leftarrow H_r + L^{\text{meas}} - A_r\, L_{\text{AR}}(B), \qquad T_r^{\text{AR}} \leftarrow T_r^{\text{AR}} + A_r\, L_{\text{AR}}(B). \tag{2}$$

The *delay cap* holds the delay to a delay tolerance $\varepsilon \geq 0$ times the AR time, plus an arrival allowance $\beta \geq 0$ in milliseconds:

$$H_r \;\leq\; \varepsilon\, T_r^{\text{AR}} + \beta \qquad \text{for every capped request } r. \tag{3}$$

Equivalently, the measured decoding time $T_r = T_r^{\text{AR}} + H_r$ is at most $(1+\varepsilon)\, T_r^{\text{AR}} + \beta$, that is, $1+\varepsilon$ times its AR time, plus β. Best-effort requests carry no cap. The delay a capped request may still take is its *slack* $R_r = \varepsilon\, T_r^{\text{AR}} + \beta - H_r$, which starts at β.

Slack turns the cap into a limit on the next step. Every step commits at least one token per request, so by Equation (2) it changes request r's slack by at least $(1+\varepsilon)\, L_{\text{AR}}(B) - L^{\text{meas}}$, whatever the acceptance. Measured latency is known only after the step, so CADENCE admits only budgets whose profiled latency is within the allowed step latency

$$\tau \;=\; (1+\varepsilon)\, L_{\text{AR}}(B) + \begin{cases} \rho\, \min_q R_q & \text{if } \min_q R_q \geq 0, \\ \min_q R_q & \text{otherwise,} \end{cases} \tag{4}$$

with the minimum over capped requests. A step may last $1+\varepsilon$ AR steps plus the *pace* $\rho \in (0, 1]$ times the least slack, and negative slack counts in full. The AR step is always allowed. A step's *overshoot* is how much its measured latency exceeded its profiled latency, or zero. A step within τ leaves request r's slack at least $(1-\rho)\max(R_r, 0)$ minus its overshoot. An AR step earns $\varepsilon\, L_{\text{AR}}(B)$ of slack, so if no step overshoots by more than that, the AR step stays within τ. By induction from β, every capped request then stays within its cap on the measured clock up to one step's overshoot, whatever the verification outcomes and arrivals (Appendix A.3). The cap bounds loss against profiled AR steps at the load each request saw, not against AR decoding (Section 2.3).

The capped request with the least slack, the *critical request*, sets τ. When the utility favors long steps, $\rho = 1$ lets one step spend all of its slack and can force AR steps after it. A smaller ρ, as in branch-parallel serving (Gandhi et al., 2026), prevents this and holds back a fraction $1-\rho$ of slack as a reserve against overshoot. Our runs fix $\rho = 0.8$ unless stated, and Appendix A.2 gives an option that adapts it online.

3.3 CHOOSING WHOSE TOKENS FILL THE STEP

In CADENCE, the operator supplies a utility u_r of each request's decoding speed and a batch objective W. The throughput utility, with u_r the speed and W their sum, is the default. Priority weights w_r and utilities clipped at a per-token deadline d_r belong to the same family (Appendix A.4).

Suppose first that each request's survival $s_{r,j}$, the probability that its first j draft tokens are accepted ($s_{r,0} = 1$), were known. It gives the expected acceptance length $a_r(\ell) = \sum_{j=0}^{\ell} s_{r,j}$ and the expected decoding speed $p_r = a_r(\ell_r)/(L+\delta)$. The margin δ covers timing variation and host work and never enters the cap. CADENCE fills each admissible budget with the draft tokens of highest *expected benefit*, the expected utility each adds at the budget's fixed latency: proportional to $s_{r,j}$ under the throughput utility and to $w_r s_{r,j}$ with priority weights. An all-or-none *floor* goes first: while τ admits a budget that covers every request, a step verifies either no draft token or every request's first, starting

Algorithm 1 CADENCE Per-Step Controller

Inputs. `requests` with slack R_r and optional deadline d_r; registered `depths` ≥ 1; latency profile `step_latency`; `utility` (objective W, margin δ); `tolerance` ε; `allowance` β; pace ρ; shared acceptance `curve`.

```
def CadenceStep(requests, step_latency, depths, utility, tolerance, rho, curve):
    B = len(requests)
    ar_latency = step_latency(B, 0, B)                    # AR: one token each
    least = min(r.slack for r in requests if r.capped)    # critical request
    spend = rho * least if least > 0 else least           # rho on positive slack
    allowed = (1 + tolerance) * ar_latency + spend        # allowed step latency
    best = Plan(0, [0] * B, ar_latency)                   # always allowed
    for depth in depths:                                  # registered, >= 1
        ranked = Rank(requests, depth, utility, curve)    # floor, shared curve
        for K in CandidateK(step_latency, B, depth):      # flat-stretch ends
            latency = step_latency(B, depth, K)           # fixed by depth and K
            if latency > allowed: continue                # not admissible
            lengths = Fill(ranked, K - B, latency)        # by rank, pad to K
            plan = Plan(depth, lengths, latency)
            if utility.better(plan, best): best = plan    # expected utility
    return best                                           # run it, then Track
def Track(requests, accepted, measured, ar_latency, tolerance):
    for r in requests:                                    # after verification
        r.slack += (1 + tolerance) * accepted[r] * ar_latency - measured
def Admit(request, allowance):                            # on arrival
    request.slack = allowance                             # start at the allowance
```

with the critical request's, so no request waits through a speculative step with nothing to verify. Priority weights may limit a budget to higher-weight requests. Since survival never rises with j, the fill is exact within each budget for both utilities, given the floor. CADENCE picks the budget scoring highest under W.

Survival is unknown before verification, so CADENCE reads it from one shared offline acceptance curve, the same for every request. With equal survival, the fill gives every request its first draft token, then its second, and so on, and the utility picks among the admissible budgets.

Algorithm 1's helpers (Appendix A.1) try only the largest K of each flat stretch of the profile, since verifying more never lowers expected acceptance (`CandidateK`), and pad each budget to exactly its K so the step runs at the checked latency (`Fill`).

3.4 WHY PER-STEP REGULATION WORKS

`Track` closes a loop on realized acceptance. Speculation that pays earns slack and can raise the next τ. Speculation that does not pay drains slack and can lower it, so CADENCE contracts by lowering the draft depth or K, whichever costs the utility least, down to AR steps. These rebuild slack when $\varepsilon\, L_{\mathrm{AR}}(B)$ exceeds their overshoot, but with $\varepsilon = 0$ a request at its cap can hold the batch to AR steps until it leaves. A request the utility passes over, a new arrival or one whose draft tokens are always rejected can become critical, and the cap bounds what it loses, not whether it receives speculation. The acceptance curve never touches τ, so a poor curve can waste speculation but cannot break a cap.

Deployment. CADENCE reads only the latency profile, one offline acceptance curve and each request's committed tokens, never model internals, so any chain drafter fits, and each decode instance runs its own controller (Appendix A.6).

Setting the knobs. At saturation the cap costs at most 0.2% of throughput for $\varepsilon \geq 0.05$, but $1 + \varepsilon$ must exceed measured over profiled latency (Appendices D.5 and D.9). The allowance β lets arrivals speculate before building slack (Appendix C). The pace sets when speculative steps run, not how many the cap allows (Appendix A.3), and priority weights trade throughput for deadlines (Section 4.3).

4 EVALUATION

Can CADENCE escape the throughput trap (Section 2.3) and keep the throughput of the best SD policy while slowing no request beyond its delay cap? We follow every policy through one trace whose load changes (Section 4.2) and show what the operator's utility contributes (Section 4.3).

4.1 EXPERIMENTAL SETUP

Model and workload. We serve Qwen3-32B with its EAGLE-3 drafter (Li et al., 2025) on one H200, in SGLang (Zheng et al., 2024) with its defaults (bf16, FlashAttention-3, CUDA graphs) and at most 112 requests per batch. Prompts mix chat, summarization, math and code in equal shares. Appendix C describes each request's acceptance level and spread, and their drift over its output.

Arrivals. Requests arrive on an Azure-derived trace (Microsoft, 2024; Patel et al., 2024), replayed for 600 s with a different scaling factor in each 200 s phase. The trace alternates bursts and quiet periods that last 8 s and 10 s on average. In multiples of the capacity of AR steps at batch 112 (33.8 requests/s), their rates are 0.6 and 0.06 under low load, 2.0 and 0.2 under high load, and 1.2 and 0.12 under moderate load. Every policy serves the same arrivals and requests with ten seeds. The slow-class runs of Section 4.2, the ablation and the appendix use steady-load runs, which hold one load for 100 s. *Saturation* holds the high-load rates (Appendix C).

Baselines. We compare CADENCE with four baselines, all run in SGLang (SGL). SGL-AR is AR decoding, with the drafter off. SGL-SD is SGLang's default: draft depth three, with every draft token verified. SGL-SD-D7 drafts to depth seven and verifies every draft token. SGL-SD-C320 drafts to depth seven but verifies at most 320 tokens per step, the best of nine such limits. CADENCE uses the throughput utility, $\varepsilon = 0.25$, $\beta = 40$ ms and $\rho = 0.8$.

Metrics. *Throughput* counts output tokens per second over the steps that start with requests waiting, relative to SGL-AR. *TPOT* is the time per output token from admission. A request's *slowdown* is its decoding time over the time AR decoding would take to commit its tokens at the batch sizes it saw, and the request is *slower than AR decoding* when this exceeds one. *Goodput* is throughput counting only the tokens of requests that are not slower than AR decoding. The *worst slowdown* is each run's largest, averaged over runs. A request is *over the cap* if its measured delay at departure breaks Equation (3). The cap is set against AR steps, which include the catch-up pass: at batch 112 an AR step takes 1.14× an SGL-AR step, so at $\varepsilon = 0.25$ the cap allows a slowdown of up to 1.43 before β. Throughput and TPOT time decode steps only (Appendix D.10 adds co-located prefill).

4.2 ESCAPING THE THROUGHPUT TRAP

Low load (0–200 s). Every policy keeps up, and no queue forms. CADENCE drafts three tokens per request (Figure 3i), as SGL-SD does, and both slow 0.02% of requests.

High load (200–400 s). Bursts at twice the capacity of AR steps fill the batch. CADENCE reaches 1.30× SGL-AR's throughput, 1.26× SGL-SD's, 2.08× SGL-SD-D7's and 1.14× SGL-SD-C320's. CADENCE's queue averages 65.6 requests, against 183.6 for SGL-SD, 214.6 for SGL-AR (Figure 3c) and 919 for SGL-SD-D7, whose requests wait 36.6 s on average. SGL-SD keeps verifying three draft tokens per request, though many fail the break-even condition (1), so its steps average 39.5 ms, against 27.8 ms for CADENCE, which drafts about 2.3 tokens per request (Figures 3d and 3i). SGL-SD reaches 1.03× SGL-AR's throughput and slows 31.4% of requests, against a 13% gain and 26% slowed in one step at batch 112 (Figure 1c). SGL-SD-C320 slows 14.7% and CADENCE 0.53% (Figure 3e). The 99th-percentile (p99) slowdown is 1.57 for SGL-SD and 0.96 for CADENCE (Figure 3f). Low-acceptance requests pay under SGL-SD: their p99 TPOT is 50.4 ms, against 29.8 ms under CADENCE and 29.1 ms under SGL-AR (Figure 3g).

Moderate load (400–600 s). Bursts at 1.2× the capacity of AR steps outpace SGL-AR and SGL-SD, whose queues average 19.0 and 9.6 requests, but not CADENCE (0.9 requests). SGL-SD still slows 7.8% of requests (p99 slowdown 1.31), and CADENCE 0.10%. Low-acceptance p99 TPOT is now 21.6 ms under CADENCE, below SGL-AR's 29.0 ms.

Summary. Under high load, CADENCE raises goodput 1.29× over SGL-AR (Appendix Table 8). Over SGL-SD, it raises throughput 1.26× and goodput 1.80× and cuts the share of requests slower

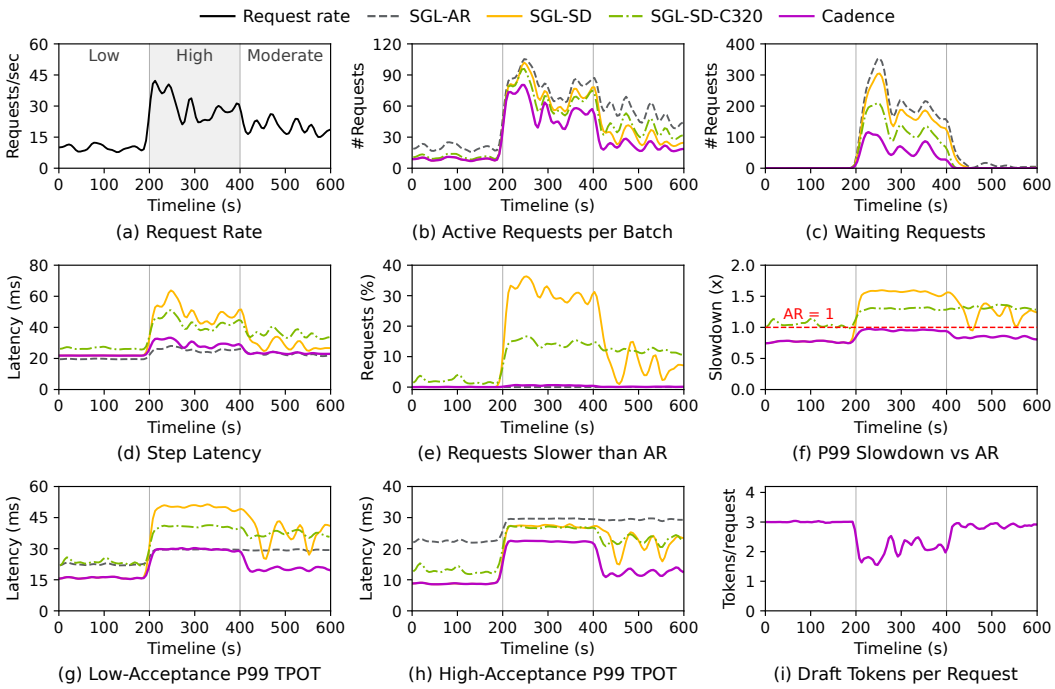


Figure 3: **As the load changes,** CADENCE **drains the queue as fast as any policy and slows almost no request.** Panels (a)–(d) and (i) show means over ten seeds, and the tail panels (e)–(h) pool the ten seeds' requests in each 5 s bin. Low and high acceptance are the quarters of requests with the lowest and highest mean acceptance lengths at draft depth seven. SGL-SD-C320 lengthens steps (d) and slows some requests (e) even under low load, and queues under high load (c). There, SGL-SD's steps lengthen (d) and about a third of its requests run slower than AR decoding (e), low-acceptance requests most of all (g). CADENCE drafts less as the batch fills (i), queues little (c) and keeps its p99 slowdown below AR decoding's level of 1 (f). In (g), its line lies on SGL-AR's under high load and drops below it under moderate load. Under high load, SGL-SD-D7 (not drawn) queues about 900 requests.

than AR decoding from 31.4% to 0.53%, the worst slowdown from 2.08 to 1.21 (2.17 to 1.25 for the slowest request of all ten trace runs) and the share over the cap from 2.25% to 0.

Whom the cap protects. On the trace, the gains come from the floor and the per-step choice of depth and budget, and the cap never binds. In the tolerance sweep, the cap's ablation, $\varepsilon = 0.25$ changes no step's plan (Appendix D.5, Table 16). The cap is the bound that holds when the floor and fill do not. In steady-load runs at saturation, we add a slow class, 10% of requests whose draft tokens are rarely accepted (Table 10). SGL-SD then puts 7.6% of requests over the cap (76.8% of the slow class). CADENCE puts none over, at 1.24× SGL-AR's throughput.

4.3 THE OPERATOR'S UTILITY

Table 1 changes CADENCE's utility on classed traffic. With equal weights, CADENCE meets a 25 ms deadline for 96.0% of tight requests and SGL-SD for 49.9% (Appendix Table 11). A priority weight w lets CADENCE withhold draft tokens from best-effort requests to shorten the steps tight requests wait through, a trade that SGL-SD cannot make. At $w = 2$ and 4, tight requests meeting a 20 ms deadline rise from 42.1% to 56.7% and 86.5%, for 0.9% and 5.4% less throughput, while best-effort p99 TPOT rises from 27.0 to 37.3 and 34.3 ms.

Overhead. CADENCE's per-step decision takes 0.4 ms at the median and 1.2 ms at the 99th percentile (1.2% and 3.6% of an AR step at batch 112), and the step latency includes it (Appendix D.11).

Table 1: **A priority weight trades throughput for a latency-sensitive class.** Steady-load runs at saturation. A quarter of requests are tight, with a per-token deadline d from admission, and the rest best effort, with no cap. A priority weight w on tight requests configures CADENCE's utility. Throughput is relative to CADENCE with equal weights.

	throughput	tight requests meeting d (%)				best-effort TPOT
policy	/ CADENCE	$d = 20$	$d = 25$	$d = 30$	$d = 35$	p50 / p99 (ms)
CADENCE	1.00×	42.1	96.0	99.8	100.0	19.7 / 27.0
priority weight $w = 2$	0.99×	56.7	97.8	99.8	100.0	19.1 / 37.3
priority weight $w = 4$	0.95×	86.5	99.6	100.0	100.0	18.8 / 34.3

Further experiments. Appendix D repeats the comparison on five more models, four of them mixture-of-experts (MoE), on A100, H100, H200 and B200 GPUs and in FP8, INT8 and MXFP4. It also varies the batch size, tolerance, deadline, acceptance spread, share of low-acceptance requests, drafter quality and profile bias, and adds larger MoE batches and co-located prefill.

5 Related Work

Neither offline profiling nor per-step selection is unique to CADENCE. D-cut (Liu et al., 2026b) jointly chooses a total budget and confidence-ranked prefixes using an end-to-end step-cost profile measured at startup. DSpark (Cheng et al., 2026) combines confidence with hardware capacity: it handles jagged capacity profiles, selects capacity by historical confidence and ranks candidates by current confidence. Bandit systems (Hou et al., 2025) learn speculation settings online. Nightjar (Li et al., 2026a) sets one batch-wide speculation length per step from the batch size alone and explores by running other lengths. CADENCE instead chooses the draft depth and per-request speculation lengths and tracks each request's delay. Deadline-aware systems tie speculation to service requirements. AdaServe (Li et al., 2026c) divides a hardware-profiled token budget among speculative trees using each request's elapsed decoding time, output count and per-token deadline, treating iteration time as constant within that budget. WISP's edge–cloud batch scheduler (Li et al., 2026b) tests estimated completion against every included request's deadline. For branch-parallel serving, Gandhi et al. (2026) admit branches when the batch's deadline slack absorbs the estimated extra latency, treating it as shared across co-batched requests. CADENCE caps each request's delay relative to AR steps instead. In short, D-cut and DSpark rank draft tokens by the drafter's confidence within one step, and AdaSpec (Huang et al., 2025b) tests each step's time against a per-token latency target, while CADENCE carries each request's delay across steps as slack, measured against AR decoding. We do not run these systems, so our comparisons do not rank complete serving systems.

6 Conclusion and Limitations

CADENCE treats speculative decoding in a shared batch as an externality to regulate, since one request's draft tokens lengthen every request's step. One allowed step latency limits each step, and an all-or-none floor gives every request its first draft token in a speculative step whenever that latency allows. Under high load with mixed traffic, CADENCE improves goodput by 1.29× over SGL-AR and 1.80× over SGL-SD. Over SGL-SD, it raises throughput 1.26×, cuts the share of requests slower than AR decoding from 31.4% to 0.53% and the slowest request's slowdown from 2.17 to 1.25, and puts none over its cap. Its gains depend on how widely acceptance varies across requests. We evaluate greedy chain drafting with one model on one GPU, and Appendix D.2 adds seven configurations. A request's measured delay can exceed its cap by one step's overshoot. In our runs at $\varepsilon = 0.1$, at most 0.02% of requests under CADENCE briefly pass it, by at most 1.3 ms, and depart within it. Even so, the results suggest that speculation need not trade the slowest requests for throughput: a per-step controller can raise throughput while bounding what each request pays.

Reproducibility Statement

We will release the code and data upon publication. Section 3 and Algorithm 1 specify CADENCE in enough detail to reimplement it, and Section 4.1 and Appendix C describe the experimental setup. Every model, drafter and dataset we use is public, as is the Azure LLM inference trace that drives the arrivals.

References

Talor Abramovich, Maor Ashkenazi, Izzy Putterman, Benjamin Chislett, Tiyasa Mitra, Bita Darvish Rouhani, Ran Zilberstein, and Yonatan Geifman. SPEED-Bench: A unified and diverse benchmark for speculative decoding. In *Proceedings of the 43rd International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR, 2026.

Amit Aggarwal, Stefan Savage, and Thomas Anderson. Understanding the performance of TCP pacing. In *Proceedings of IEEE INFOCOM 2000*, volume 3, pp. 1157–1165, 2000. doi: 10.1109/INFCOM.2000.832483.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134. USENIX Association, 2024a.

Sudhanshu Agrawal, Wonseok Jeon, and Mingu Lee. AdaEDL: Early draft stopping for speculative decoding of large language models via an entropy-based lower bound on token acceptance probability. In *Proceedings of The 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*, volume 262 of *Proceedings of Machine Learning Research*, pp. 355–369. PMLR, 2024b.

Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018. doi: 10.1609/aaai.v32i1.11797.

Ashwinkumar Badanidiyuru, Robert Kleinberg, and Aleksandrs Slivkins. Bandits with knapsacks. *Journal of the ACM*, 65(3):1–55, 2018. doi: 10.1145/3164539.

Martino Bernasconi, Matteo Castiglioni, Andrea Celli, and Federico Fusco. Bandits with replenishable knapsacks: The best of both worlds. In *The Twelfth International Conference on Learning Representations*, 2024.

Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Cilantro: Performance-aware resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 623–643. USENIX Association, 2023.

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*, volume 235 of *Proceedings of Machine Learning Research*, pp. 5209–5235. PMLR, 2024.

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

Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B. Gibbons. SLOs-Serve: Optimized serving of multi-SLO LLMs. *arXiv preprint arXiv:2504.08784*, 2025.

Wei Chen, Yajun Wang, Yang Yuan, and Qinshi Wang. Combinatorial multi-armed bandit and its extension to probabilistically triggered arms. *Journal of Machine Learning Research*, 17(50):1–33, 2016.

Wenyan Chen, Chengzhi Lu, Yanying Lin, and Dmitrii Ustiugov. FASER: Fine-grained phase management for speculative decoding in dynamic LLM serving. *arXiv preprint arXiv:2604.20503*, 2026a.

Yuansheng Chen, Yue Zhang, Xuan Mo, Weigang Wu, and Jialun Li. Beyond greedy chunking: SLO-aware sliding-window scheduling for LLM inference. *arXiv preprint arXiv:2606.05933*, 2026b.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. Sequoia: Scalable and robust speculative decoding. In *Advances in Neural Information Processing Systems*, volume 37, pp. 129531–129563, 2024. doi: 10.52202/079017-4116.

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

Rene L. Cruz. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991. doi: 10.1109/18.61109.

Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 316–326, 2009. doi: 10.1145/1669112.1669154.

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 335–346, 2010. doi: 10.1145/1736020.1736058.

Eray Can Elumar, Cem Tekin, and Osman Yağan. Bandits with anytime knapsacks. *arXiv preprint arXiv:2501.18560*, 2025.

Swapnil Gandhi, Siva Hari, William J. Dally, and Christos Kozyrakis. Regulating branch parallelism in LLM serving. *arXiv preprint arXiv:2605.06914*, 2026.

Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. QoServe: Breaking the silos of LLM inference serving. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS)*, pp. 1492–1507. ACM, 2026. doi: 10.1145/3779212.3790206.

Yunlong Hou, Fengzhuo Zhang, Cunxiao Du, Xuan Zhang, Jiachun Pan, Tianyu Pang, Chao Du, Vincent Y. F. Tan, and Zhuoran Yang. BanditSpec: Adaptive speculative decoding via bandit algorithms. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 24045–24079. PMLR, 2025.

Xinyi Hu, Yuhao Shen, Baolin Zhang, Hengxin Zhang, Jun Dai, Shuang Ge, Lei Chen, Yue Li, and Mingcheng Wan. ECHO: Elastic speculative decoding with sparse gating for high-concurrency scenarios. *arXiv preprint arXiv:2604.09603*, 2026.

Kaixuan Huang, Xudong Guo, and Mengdi Wang. SpecDec++: Boosting speculative decoding via adaptive candidate lengths. In *Second Conference on Language Modeling*, 2025a.

Kaiyu Huang, Hao Wu, Zhubo Shi, Han Zou, Minchen Yu, and Qingjiang Shi. AdaSpec: Adaptive speculative decoding for fast, SLO-aware large language model serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, pp. 361–374. ACM, 2025b. doi: 10.1145/3772052.3772239.

Abbas Kazerouni, Mohammad Ghavamzadeh, Yasin Abbasi-Yadkori, and Benjamin Van Roy. Conservative contextual linear bandits. *arXiv preprint arXiv:1611.06426*, 2017.

Linghao Kong, Megan Flynn, Michael Peng, Nir Shavit, Mark Kurtz, and Alexandre Marques. An interpretable latency model for speculative decoding in LLM serving. *arXiv preprint arXiv:2605.15051*, 2026.

Branislav Kveton, Csaba Szepesvári, Zheng Wen, and Azin Ashkan. Cascading bandits: Learning to rank in the cascade model. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 767–776. PMLR, 2015a.

Branislav Kveton, Zheng Wen, Azin Ashkan, and Csaba Szepesvári. Combinatorial cascading bandits. In *Advances in Neural Information Processing Systems*, volume 28, 2015b.

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

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

Rui Li, Zhaoning Zhang, Libo Zhang, Huaimin Wang, Xiang Fu, and Zhiquan Lai. Nightjar: Dynamic adaptive speculative decoding for large language models serving. *Journal of Systems Architecture*, 178:103889, 2026a. doi: 10.1016/j.sysarc.2026.103889.

Xiangchen Li, Jiakun Fan, Qingyuan Wang, Dimitrios Spatharakis, Saeid Ghafouri, Hans Vandierendonck, Deepu John, Bo Ji, Ali R. Butt, and Dimitrios S. Nikolopoulos. WISP: Waste- and interference-suppressed distributed speculative LLM serving at the edge via dynamic drafting and SLO-aware batching. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 10(2):55:1–55:31, 2026b. doi: 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*, volume 235 of *Proceedings of Machine Learning Research*, pp. 28935–28948. PMLR, 2024a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 7421–7432. Association for Computational Linguistics, 2024b. doi: 10.18653/v1/2024.emnlp-main.422.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Zikun Li, Zhuofu Chen, Remi Delacourt, Gabriele Oliaro, Zeyu Wang, Qinghan Chen, Shuhuai Lin, April Yang, Zhihao Zhang, Zhuoming Chen, Yi-Hsiang Lai, Xinhao Cheng, Xupeng Miao, and Zhihao Jia. AdaServe: Accelerating multi-SLO LLM serving with SLO-customized speculative decoding. In *Proceedings of the 21st European Conference on Computer Systems*, pp. 36–54. ACM, 2026c. doi: 10.1145/3767295.3769315.

Hongyi Liu, Jiaji Huang, Zhen Jia, Youngsuk Park, and Yu-Xiang Wang. Not-a-Bandit: Provably no-regret drafter selection in speculative decoding for LLMs. In *International Conference on Learning Representations*, 2026a.

Jiachen Liu, Jae-Won Chung, Zhiyu Wu, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. Andes: Defining and enhancing quality-of-experience in LLM-based text streaming services. *arXiv preprint arXiv:2404.16283*, 2024a.

Jiahao Liu, Qifan Wang, Jingang Wang, and Xunliang Cai. Speculative decoding via early-exiting for faster LLM inference with Thompson sampling control mechanism. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3027–3043. Association for Computational Linguistics, 2024b. doi: 10.18653/v1/2024.findings-acl.179.

Tianyu Liu, Yuhao Shen, Rui Cen, Junhan Shi, Jiebin Zhang, Guangshuo Qin, Hong Liu, Song Liu, Guanghua Yu, and Jianchen Zhu. D-cut: Adaptive verification depth pruning for batched speculative decoding. *arXiv preprint arXiv:2607.14647*, 2026b.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Online speculative decoding. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 31131–31146. PMLR, 2024c.

Xiaoxuan Liu, Jiaxiang Yu, Jongseok Park, Ion Stoica, and Alvin Cheung. Speculative decoding: Performance or illusion? *arXiv preprint arXiv:2601.11580*, 2026c.

David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 450–462. ACM, 2015. doi: 10.1145/2749469.2749475.

Hongtao Lyu, Boyue Liu, Mingyu Wu, and Haibo Chen. FairBatching: Fairness-aware batch formation for LLM inference. *arXiv preprint arXiv:2510.14392*, 2025.

Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 589–603. USENIX Association, 2015.

Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, pp. 132–141, 1998. doi: 10.1109/ISCA.1998.694769.

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)*, pp. 932–949. ACM, 2024. doi: 10.1145/3620666.3651335.

Microsoft. Azure LLM inference trace 2023. https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md, 2024.

Ahmadreza Moradipari, Christos Thrampoulidis, and Mahnoosh Alizadeh. Stage-wise conservative linear bandits. In *Advances in Neural Information Processing Systems*, volume 33, pp. 11191–11201, 2020.

Soowon Oh, Nam Cao, Yujin Kim, Hojung Jung, Huzama Ahmad, Sangmin Bae, and Se-Young Yun. Bastion: Budget-aware speculative decoding with tree-structured block diffusion drafting. *arXiv preprint arXiv:2605.29727*, 2026.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, 2024. doi: 10.1109/ISCA59077.2024.00019.

Haifeng Qian, Sujan Kumar Gonugondla, Sungsoo Ha, Mingyue Shang, Sanjay Krishna Gouda, Ramesh Nallapati, Sudipta Sengupta, Xiaofei Ma, and Anoop Deoras. BASS: Batched attention-optimized speculative sampling. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 8214–8224. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.489.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — A KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170. USENIX Association, 2025.

Jameson Sandler, Ahmet Üstün, Marco Romanelli, Sara Hooker, and Ferdinando Fioretto. The disparate impacts of speculative decoding. *arXiv preprint arXiv:2510.02128*, 2025.

Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 965–988. USENIX Association, 2024.

Qidong Su, Christina Giannoula, and Gennady Pekhimenko. The synergy of speculative decoding and batching in serving large language models. *arXiv preprint arXiv:2310.18813*, 2023.

Phuong Tran, Tzu-Hao Liu, Long Tan Le, Tung-Anh Nguyen, Van Quan La, Eason Yu, Han Shu, Choong Seon Hong, and Nguyen H. Tran. GoodSpeed: Optimizing fair goodput with adaptive speculative decoding in distributed edge inference. In *IEEE INFOCOM 2026 – IEEE Conference on Computer Communications*, pp. 1–10. IEEE, 2026. doi: 10.1109/INFOCOM59046.2026.11571335.

Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 191–202, 1996. doi: 10.1145/232973.232993.

Qinshi Wang and Wei Chen. Improving regret bounds for combinatorial semi-bandits with probabilistically triggered arms and its applications. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Qipeng Wang. Taming request imbalance: SLO-aware scheduling for disaggregated LLM inference. *arXiv preprint arXiv:2605.02329*, 2026.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Yifan Wu, Roshan Shariff, Tor Lattimore, and Csaba Szepesvári. Conservative bandits. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 1254–1262. PMLR, 2016.

Zhaoxuan Wu, Zijian Zhou, Arun Verma, Alok Prakash, Daniela Rus, and Bryan Kian Hsiang Low. TETRIS: Optimal draft token selection for batch speculative decoding. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, pp. 33329–33345, 2025. doi: 10.18653/v1/2025.acl-long.1598.

Minghao Yan, Saurabh Agarwal, and Shivaram Venkataraman. Decoding speculative decoding. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 6460–6473. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.328.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538. USENIX Association, 2022.

Ranran Haoran Zhang, Soumik Dey, Ashirbad Mishra, Hansi Wu, Binbin Li, and Rui Zhang. Correctness forensics for batch speculative decoding: Diagnosing the ragged tensor problem. In *Findings of the Association for Computational Linguistics: EMNLP 2026*, 2026. To appear. arXiv:2510.22876.

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

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210. USENIX Association, 2024.

Amir Ziashahabi, Hossein Entezari Zarch, Lei Gao, Murali Annavaram, and Salman Avestimehr. ASPIRE: Asynchronous batched self-speculative decoding for long-context LLM inference. *arXiv preprint arXiv:2609.17943*, 2026.

A THE CONTROLLER IN DETAIL

This appendix gives the detail behind Section 3: the helpers of Algorithm 1 (Appendix A.1), the pace and the price on slack (Appendix A.2), the proofs behind the delay cap (Appendix A.3), the utilities (Appendix A.4), a worked example (Appendix A.5) and serving with several instances (Appendix A.6).

Notation. A step serves B resident requests. A *candidate budget* pairs a registered draft depth $\ell_{\max}$ with a verification batch size K, where $B \le K \le B(1+\ell_{\max})$. Every request drafts to depth $\ell_{\max}$, and request r verifies its first $\ell_r \le \ell_{\max}$ draft tokens, its speculation length, so $K = B + \sum_r \ell_r$. We write $\mathcal{G}$ for the set of registered draft depths and J for the largest. A candidate budget fixes the step latency $L(B, \ell_{\max}, K)$, so all of its allocations pass or fail the allowed step latency τ together. Survival $s_{r,j}$ is the probability that request r's first j draft tokens are all accepted, with $s_{r,0} = 1$, and $a_r(\ell) = \sum_{j=0}^{\ell} s_{r,j}$ is its expected acceptance length at speculation length ℓ. CADENCE reads survival from one shared offline acceptance curve, the same for every request: $\bar{s}_j$ is the curve's survival at position j, read from the upper concave hull of its expected acceptance lengths, so it never increases with j. The curve both ranks draft tokens and scores plans. A step's profiled latency L is read from the profile before the step, and its measured latency L^{meas} is timed when it runs. Its *overshoot* is $o = \max(0, L^{\text{meas}} - L)$.

A.1 HELPERS OF ALGORITHM 1

`CandidateK`. This helper lists the values of K worth trying at depth $\ell_{\max}$. It forms the lower envelope of the step latency from the right, $\underline{L}(K) = \min_{K \le K' \le B(1+\ell_{\max})} L(B, \ell_{\max}, K')$, after smoothing dips that lie within profile noise, and returns the right end of each flat stretch of $\underline{L}$. On a rising stretch every profiled K is a stretch of its own, and the full draft at that depth, $K = B(1+\ell_{\max})$, always ends the last one. The right end of a flat stretch dominates every point inside it. It runs no longer, and verifying more draft tokens never lowers any request's expected acceptance length. Every utility is non-decreasing in expected decoding speed (Appendix A.4), so moving an allocation to the right end never lowers its score, and the step, no longer than before, stays within τ.

`Rank`. This helper orders the draft tokens (r, j) with $j \le \ell_{\max}$. The *floor* comes first: the first draft token of every request, capped or best effort, starting with the critical request's and then those of the other capped requests whose slack is below their allowance, $R_r < \beta$. The remaining first draft tokens, and after them the deeper ones, follow by weight times key. The key is the shared curve's survival $\bar{s}_j$. The weight is one under the throughput utility, w_r with priority weights, and $w_r d_r$ under a utility clipped at a deadline (Appendix A.4). Ties go to the request with less slack, then to the smaller j. Under the throughput utility every request has the same key at each position, so this tie rule orders the requests and the fill is breadth first. The key never increases with j, so a fill in rank order gives each request a prefix of its draft tokens, which is what verification accepts. When $K - B < B$, the fill takes the floor's first $K - B$ draft tokens. Such a budget leaves some request without a draft token and is admissible only when τ admits no speculative budget with $K - B \ge B$. Under priority weights the floor keeps its order but not this rule, so a budget may cover only the higher-weight requests.

The floor protects requests that the utility would pass over. Take each step at its profiled latency. At speculation length zero a request commits one token, so it loses slack in every step whose relative step latency $\theta = L/L_{\text{AR}}(B)$ exceeds $1 + \varepsilon$, and once it becomes critical it lowers τ for the whole batch. With its first draft token verified in every speculative step, its expected slack change per step is at least $[(1+\varepsilon)(1+s_{r,1}) - \theta]\, L_{\text{AR}}(B)$, so it stops losing slack in expectation once $\theta \le (1+\varepsilon)(1+s_{r,1})$. A floor for the critical request alone, or for capped requests below their allowance, would act only after a request had lost slack, and a best-effort request with no cap could be passed over without ever becoming critical. The floor therefore covers every request. It removes budgets that pass over some request while a covering budget is within τ, and it never removes the AR step, so the check against τ and the proofs of Appendix A.3 are unchanged.

`Fill`. This helper places $K - B$ draft tokens in rank order. Under a utility clipped at a deadline, a request's draft tokens add nothing once its expected decoding speed, from the shared curve, meets its deadline at the budget's step latency, which `Fill` receives for this purpose. `Fill` skips such draft

tokens, except those of the floor. If the draft tokens with positive expected benefit run out before K, it pads with the skipped ones, still in rank order, until the allocation reaches exactly K. The executed step then has exactly the step latency that was checked. A profile can dip, so a fill that stopped short of K could run longer than that latency. Each request takes at most $\ell_{\max}$ draft tokens and $K \le B(1+\ell_{\max})$, so the fill always reaches K. Under the throughput and priority utilities every draft token has positive expected benefit, and `Fill` simply takes the first $K - B$.

Plans and scores. A `Plan` holds a draft depth, the speculation lengths and the step latency. The AR step is the plan with depth zero and no draft tokens. `utility.better` compares two plans by the objective W at each plan's step latency. Each request's expected decoding speed, $p_r = a_r(\ell_r)/(L+\delta)$, uses the shared curve's survival $\bar{s}_j$. The margin δ enters only this speed, never τ. Ties go to the smaller K, then to the smaller draft depth. With the price on slack (Appendix A.2), the score also subtracts the price of the slack the plan spends. The step runs at the planned depth even if the largest speculation length is smaller, because a smaller depth is a different candidate budget. The runtime times each step over the span the profile measures, and `Track` charges this measured latency to every request's slack. `Admit` sets an arriving request's slack to the arrival allowance β. When every request in the batch is best effort, the minimum in τ runs over an empty set. Then τ is unbounded, every candidate budget is admissible, and `Rank`'s floor has no critical request to put first but still covers every request.

Cost of the search. Under the throughput and priority utilities the ranking does not depend on the step latency. One sort of the BJ draft tokens of the largest depth, and a prefix-sum sweep of survival in rank order at each depth, cost $O(BJ\log(BJ) + |\mathcal{G}|\,BJ)$ operations. At $B = 112$ and depths $\{1, 3, 7\}$, so that $J = 7$, this is one sort of 784 draft tokens and at most 3×784 candidate-budget evaluations. Under a clipped or deadline utility the fill depends on the step latency, so the scan costs $O(|\mathcal{G}|\,|\mathcal{K}|\,BJ)$, with $\mathcal{K}$ the candidate values of K. The search runs in every step, AR steps included, so its host time belongs to $L_{\mathrm{AR}}(B)$, and the profile includes it. Appendix D.11 reports the measured decision time.

A.2 The pace and the price on slack

By default ρ is fixed. While no capped request's slack is negative, Equation (4) lets a step spend at most the fraction ρ of the least slack, and every capped request keeps at least the fraction $1-\rho$ of its own slack, less the step's overshoot (Lemma A.1). Negative slack counts in full, because pacing it would be more lenient: $\rho R > R$ when $R < 0$ and $\rho < 1$. The pace plays two roles. First, the slack it holds back is a reserve against the profile's timing error, because a request whose reserve $(1-\rho)R_r$ covers a step's overshoot keeps nonnegative slack through that step. Second, it protects the batch when the utility favors long steps. With $\rho = 1$ one step can spend all of the critical request's slack. If that request then commits one token, the next τ is at most $(1+\varepsilon)\,L_{\mathrm{AR}}(B)$, which admits only budgets with $\theta \le 1+\varepsilon$. A budget of relative step latency θ then needs at least $\lceil(\theta - 1 - \varepsilon)/\varepsilon\rceil$ AR steps to rebuild that slack when $\varepsilon > 0$, and with $\varepsilon = 0$ it stays out of reach while that request remains. The pace changes which plans are admissible, never how `utility.better` ranks the plans it admits.

Adapting the pace with a price on slack. The pace can also adapt online, through a price $\mu \ge 0$ on slack. The price works with any fixed ρ, and with $\rho = 1$ it alone sets the pace. It enters `utility.better`, which then scores each admissible plan by

$$W - \mu\, W_{\mathrm{AR}}\, x, \qquad x = \frac{\big(L - (1+\varepsilon)\,L_{\mathrm{AR}}(B)\big)^{+}}{L+\delta}.$$

Here W_{AR} is the AR step's score, and x is the share of the step's time that every capped request is certain to lose from its slack, taking the step at its profiled latency. Scaling by W_{AR} makes μ a pure number. The price needs a single-number score with $W_{\mathrm{AR}} > 0$, as under the throughput, priority and clipped utilities. Under the sum of logarithms W_{AR} can be negative, and the deadline utility ranks plans in two stages, so neither uses the price. The price starts at zero. After each step, let R_{crit} be the slack, after the step, of the request that was critical before it. With a reserve $\bar{R} > 0$, a step size $\eta > 0$ and a ceiling $\mu_{\max}$, the price moves by

$$\mu \leftarrow \mathrm{clip}\big(\mu + \eta\,(\bar{R} - R_{\mathrm{crit}})/\bar{R},\ 0,\ \mu_{\max}\big),$$

where $\mathrm{clip}(v, 0, \mu_{\max})$ limits v to $[0, \mu_{\max}]$. The price rises while the critical request holds less slack than the reserve and falls while it holds more. A high price favors short steps, as a small ρ does, and $\mu = 0$ spends slack greedily within τ. The update is skipped when no request is capped, and when the critical request leaves during the step, since its slack after the step is then undefined. This is a dual update for the constraint that the critical request keep at least $\bar{R}$ of slack on average. The share x never falls as L rises, and the right end of a flat stretch runs no longer than any plan inside it, so `CandidateK`'s right ends still dominate. The price changes only which admissible plan scores highest, never τ, so the results of Appendix A.3 hold for every μ.

A.3 Proofs and properties

Lemma A.1 (Pace)**.** *Consider a step whose profiled latency L is within τ, and let o be its overshoot. After the step, every capped request r has slack $R'_r \geq (1-\rho)\max(R_r, 0) - o \geq -o$.*

Proof. The step commits $A_r \geq 1$ tokens for every capped request r, so by Equation (2) its slack becomes

$$R'_r = R_r + (1+\varepsilon)A_r L_{\mathrm{AR}}(B) - L^{\mathrm{meas}} \;\geq\; R_r + (1+\varepsilon)L_{\mathrm{AR}}(B) - \tau - o.$$

If no slack is negative before the step, Equation (4) gives $\tau = (1+\varepsilon)L_{\mathrm{AR}}(B) + \rho \min_q R_q$, and $R'_r \geq R_r - \rho \min_q R_q - o \geq (1-\rho)\, R_r - o$. Otherwise $\tau = (1+\varepsilon)L_{\mathrm{AR}}(B) + \min_q R_q$, and $R'_r \geq R_r - \min_q R_q - o \geq -o$. □

Proposition A.2 (The cap on the measured clock)**.** *Suppose that no step's overshoot exceeds $\varepsilon\, L_{\mathrm{AR}}(B)$ at the batch size B of the step after it. Then after every step, every capped request's slack is at least minus that step's overshoot, and its measured decoding time exceeds its cap $(1+\varepsilon)\, T_r^{\mathrm{AR}} + \beta$ by at most the overshoot of the last step it ran. This holds whatever the verification outcomes and arrivals, for any acceptance curve and any price on slack.*

Proof. By induction over steps. Suppose that before a step of batch size B every capped request has slack at least $-o'$, where o' is the previous step's overshoot, or zero before the first step. This holds for a request admitted since, because `Admit` sets $R_r = \beta \geq 0$. The condition gives $o' \leq \varepsilon\, L_{\mathrm{AR}}(B)$, so the least slack is at least $-\varepsilon\, L_{\mathrm{AR}}(B)$ and $\tau \geq L_{\mathrm{AR}}(B)$. The AR step is then within τ, and so is every step CADENCE runs. Lemma A.1 gives slack at least $-o$ after the step, with o its overshoot, which completes the induction. By the definitions of Section 3.2, a request's measured decoding time is $T_r = T_r^{\mathrm{AR}} + H_r = (1+\varepsilon)\, T_r^{\mathrm{AR}} + \beta - R_r$, so it exceeds the cap by at most the overshoot of its last step. The argument uses only realized acceptance lengths, the measured latencies of the steps that ran, the profiled latencies that admitted them, the profiled AR latency that credits each committed token, and the batch sizes that ran. It uses no estimate of acceptance. □

The bound is against profiled AR steps at the load each request saw, not against an AR-only server. Without the condition, the least slack can fall below $-\varepsilon\, L_{\mathrm{AR}}(B)$, and then $\tau < L_{\mathrm{AR}}(B)$. A plan within τ still obeys Lemma A.1, and when none is chosen the AR step runs anyway, because Algorithm 1 starts from it. That step changes every capped request's slack by at least $\varepsilon\, L_{\mathrm{AR}}(B) - o$. Let D be the largest amount by which any capped request's slack falls below zero after a step, and D' the same before it. The two cases give $D \leq o + \max\big(0,\, D' - \varepsilon\, L_{\mathrm{AR}}(B)\big)$ after every step. Each forced AR step repays $\varepsilon\, L_{\mathrm{AR}}(B)$ of lateness and adds its own overshoot. With $\varepsilon = 0$ and $\beta = 0$, a request therefore loses against profiled AR steps at most the summed overshoot of its steps. Appendix B.2 measures how often the condition fails, and Appendix D.5 shows what happens at $\varepsilon = 0$.

Proposition A.3 (Exactness of the fill)**.** *Fix a candidate budget, and with it the step latency L. Under the throughput and priority utilities with the sum, $W = \sum_r w_r\, a_r(\ell_r)/(L+\delta)$ with $w_r = 1$ for throughput, the floor's f draft tokens and the $K - B - f$ other draft tokens of largest $w_r \bar{s}_j$ maximize the score under the shared curve over every allocation of the budget that contains the floor.*

Proof. Each term of W is concave in its speculation length, since the gain from position j, $w_r \bar{s}_j/(L+\delta)$, does not increase with j. The standard exchange argument for separable concave allocation then gives the claim. When $K - B < B$, the same holds with the floor's first $K - B$ draft tokens in place of the floor. □

The scan therefore maximizes the score over the AR step and every admissible allocation, on the grid of candidate budgets it reads, that contains the floor. The fill is not exact under a clipped utility, because the draft token that reaches a request's deadline counts at its unclipped benefit, nor under the deadline indicator, whose threshold makes a draft token's value depend on the request's other draft tokens. Under the minimum, which does not split into one term per request, and under the sum of logarithms, whose gains are not proportional to survival, it is not exact either. Between profiled points on rising stretches the scan is not exhaustive, since no profile resolves every integer K.

How much speculation the cap allows. Apart from overshoot, a request whose draft tokens are always rejected gains $\varepsilon\, L_{\mathrm{AR}}$ of slack at each AR step and loses $(\theta - 1 - \varepsilon)\, L_{\mathrm{AR}}$ at each speculative step of relative step latency θ. In the long run, at most a fraction $\varepsilon/(\theta - 1)$ of steps can then be speculative, plus a burst worth β. For example, $\varepsilon = 0.05$ and $\theta = 1.25$ allow one step in five. The fraction does not depend on the pace, which only changes how the speculative steps are spread out. With the AR step as the known safe action and one cap per request, the cap resembles the constraint of conservative bandits (Wu et al., 2016; Kazerouni et al., 2017). Under the condition of Proposition A.2, it holds here on every step, up to one step's overshoot, rather than with high probability.

How speculation contracts. As slack shrinks, CADENCE lowers the draft depth or K, whichever costs the utility least (Section 3.4). Lowering the draft depth removes drafting latency for every request and gives up only the acceptance of the dropped positions. Lowering K removes verification latency: almost nothing inside a flat stretch, a whole jump when K drops back below a cliff. Under the floor, the shortest budget that covers every request, at the smallest registered depth, is the last covering budget to go. Once it exceeds τ, budgets with $K - B < B$ become admissible, and they cover the critical request first.

The warm-drafter premium. The AR step still runs the catch-up pass, so its latency $L_{\mathrm{AR}}(B)$ exceeds the latency $L_{\mathrm{off}}(B)$ of AR decoding with the drafter off. An EAGLE-3 drafter needs that pass even after an AR step, because it reads the target model's hidden states. We call $\omega = L_{\mathrm{AR}}/L_{\mathrm{off}} - 1$ the *warm-drafter premium*. It is the same for every request, so it is the price of keeping speculation available, not part of the externality. Against AR decoding with the drafter off, a delay tolerance ε becomes $\varepsilon + \omega + \varepsilon\omega$, with ω taken at its largest over the batch sizes that ran. At batch 112 of the main configuration, $1 + \omega = 1.14$.

The output distribution. Rejection sampling commits tokens with the target model's distribution for any speculation length chosen without inspecting the step's draft tokens. CADENCE chooses every length before drafting, from slack, the latency profile, the utility and the shared acceptance curve. None of these depends on the current step's draft tokens, so CADENCE's choices leave the output distribution unchanged. Under greedy decoding, each committed token is the target model's greedy choice in that step's verification pass. Kernels can round differently for different verification batch shapes, so this is a statement about distributions, not about the bits of a run.

A.4 UTILITY FUNCTIONS

CADENCE separates what the operator values from its estimate of acceptance, following performance-aware resource allocation (Bhardwaj et al., 2023). For a candidate plan, it computes each request's expected decoding speed $p_r = a_r(\ell_r)/\big(L(B, \ell_{\max}, K) + \delta\big)$ from the shared curve's survival. The operator supplies a per-request utility u_r, non-decreasing in p_r, and an objective W that combines the batch's utilities. The delay cap of Equation (3) is independent of both: it restricts which plans are admissible, and W ranks the admissible ones.

Throughput and priority weights. The *throughput* utility, $u_r = p_r$ with W the sum, values every expected committed token equally. *Priority weights* scale it, $u_r = w_r p_r$, so that the draft tokens of a request with a larger weight rank higher. At a fixed candidate budget, the expected benefit of draft token (r, j) is $\bar{s}_j/(L + \delta)$ under the throughput utility and $w_r \bar{s}_j/(L + \delta)$ with priority weights. For both, `Fill` is exact given the floor (Proposition A.3).

Utilities clipped at a deadline. A request may instead carry a per-token deadline d_r, the milliseconds it may take on average per output token, as in deadline-aware speculative scheduling (Li

et al., 2026c;b). Following the target-clipped utilities of Bhardwaj et al. (2023), its utility can rise with expected decoding speed until the request keeps up with its deadline and stay flat beyond, $u_r = w_r \min(p_r d_r, 1)$. A draft token's expected benefit is then $w_r d_r \bar{s}_j/(L+\delta)$ while the request's expected decoding speed is below $1/d_r$, and zero once it meets it. `Rank` therefore weights keys by $w_r d_r$, and `Fill` skips a request's draft tokens once its expected decoding speed meets its deadline at the budget's step latency.

The deadline utility. The sharpest member of the family counts only whether a request keeps up. Its utility is its weight w_r when its expected decoding speed meets the deadline, and zero otherwise:

$$u_r \;=\; w_r\,\mathbf{1}[\,p_r \geq 1/d_r\,] \;=\; w_r\,\mathbf{1}[\,d_r\,a_r(\ell_r) \geq L(B, \ell_{\max}, K) + \delta\,]\,, \tag{5}$$

with W their sum. A request keeps up in a step only if it commits at least $L(B, \ell_{\max}, K)/d_r$ tokens, a condition of the same form as break-even (Section 2.3) with the deadline in place of the AR step. No fill in a fixed order is exact for it. CADENCE fills in the clipped utility's order, and `utility.better` ranks plans by the weighted count of requests that keep up, then by weighted expected decoding speed.

Objectives. The *sum*, $W = \sum_r u_r$, maximizes total value and favors requests that turn verification into value cheaply. The *minimum*, $W = \min_r u_r$, maximizes the value of the worst-off request. The *sum of logarithms*, $W = \sum_r \log u_r$, lies between the two and rewards proportional gains. The fill's exactness needs the sum, but under every objective `utility.better` scores each candidate budget by W itself.

Relation to other policies. The throughput utility with the sum is throughput-maximizing allocation, the objective of the confidence-ranked schedulers of Section 5, which rank draft tokens by the drafter's confidence. CADENCE ranks them by the shared curve's survival instead, so the throughput fill is breadth first: every request's first draft token, then every second, and so on. It departs from breadth first only where priority weights or deadlines order the requests. Full-draft speculation is the plan at the largest registered depth with every draft token verified, run with no cap. A fixed cap on verified tokens fixes K instead of choosing a budget in each step.

Table 2: Allocations chosen by different utilities on an illustrative batch. Three requests share a 33 ms per-token deadline. Drafting time is taken as negligible, so the step takes $L(3, K) = 40 + 2(K-3)$ ms at any draft depth, and $L(3,3) = 40$ ms is the AR step. The margin is $\delta = 0$. Committed tokens at speculation lengths 0 to 7 are $(1, 1.9, 2.7, 3.4, 4.0, 4.5, 4.9, 5.2)$ for H, $(1, 1.8, 2.3, 2.6, 2.75, 2.8, 2.825, 2.85)$ for M and $(1, 1.3, 1.45, 1.52, 1.55, 1.57, 1.58, 1.59)$ for Lo. The ratio $a_r(\ell_r)/\theta$, with $\theta = L(3,K)/L(3,3)$, is a request's decoding speed relative to AR steps, and below one the step slowed the request. The row within τ has Lo as the critical request with 6 ms of slack, $\varepsilon = 0.1$ and $\rho = 1$, so $\tau = 50$ ms.

utility, objective	$(\ell_{\mathrm{H}}, \ell_{\mathrm{M}}, \ell_{\mathrm{Lo}})$	K	L (ms)	tokens/ms	on deadline	a_{H}/θ	a_{M}/θ	a_{Lo}/θ
Throughput, sum	$(7,3,1)$	14	62	0.1468	2	3.35	1.68	0.84
Throughput, sum, within τ	$(3,1,1)$	8	50	0.1300	2	2.72	1.44	1.04
Priority, $w_{\mathrm{Lo}} = 4$, sum	$(5,2,2)$	12	58	0.1422	2	3.10	1.59	1.00
Clipped at deadline, sum	$(1,1,3)$	8	50	0.1044	3	1.52	1.44	1.22
Deadline indicator, sum	$(1,1,3)$	8	50	0.1044	3	1.52	1.44	1.22
Clipped at deadline, minimum	$(1,1,3)$	8	50	0.1044	3	1.52	1.44	1.22
Full draft, no regulation	$(7,7,7)$	24	82	0.1176	2	2.54	1.39	0.78

An illustrative comparison. Table 2 compares the utilities on three co-batched requests whose survival is taken as known. Request H's drafts are usually accepted, M's less often and Lo's rarely. Every row but the full draft is the exact optimum over all 8^3 allocations, or, for the row within τ, over those within τ that give every request its first draft token. The full draft shows the externality: H decodes at 2.54 times its speed under AR steps and Lo at 0.78 times. Throughput-maximizing allocation gives most verification to H and still leaves Lo 16% slower than under AR steps. Within τ, the floor gives every request its first draft token and expected benefit fills the rest, which yields $(3,1,1)$: Lo decodes 4% faster than under AR steps, at 11% less throughput than the unregulated optimum. Even if Lo's draft token is rejected, its slack after the step is $6 + 1.1 \times 40 - 50 = 0$ ms.

A weight of four on Lo moves verification to Lo until it is no slower than under AR steps, at 3.1% of throughput. The clipped, indicator and minimum utilities all let every request meet the deadline, at a larger cost. An objective that rewards converting verification into tokens directs it away from requests that convert it poorly, and under the condition of Proposition A.2 the delay cap bounds what they pay, up to timing overshoot, whatever the objective.

A.5 A WORKED EXAMPLE

Table 3: The worked example: each candidate plan of one step with two requests, its speculation lengths, profiled latency and expected committed tokens under the stated survival ($\delta = 0$), and whether it is within τ at pace $\rho = 1$ ($\tau = 30$ ms) and $\rho = 0.5$ ($\tau = 26$ ms).

	draft	(ℓ_A, ℓ_B)	L	expected	tokens	within τ	
plan	tokens		(ms)	tokens	per ms	$\rho = 1$	$\rho = 0.5$
AR step	0	(0, 0)	20	2.00	0.100	yes	yes
$K = 4$	2	(1, 1)	24	3.44	0.143	yes	yes
$K = 7$	5	(3, 2)	29	4.83	0.167	yes	no
$K = 8$	6	(3, 3)	33	4.98	0.151	no	no

Two requests, A and B, share a step with $\varepsilon = 0.1$ and one registered depth, three. The AR step takes $L_{\mathrm{AR}}(2) = 20$ ms. At depth three the profile is flat at 24 ms for K from 2 to 4 and at 29 ms for K from 5 to 7, and the full draft, $K = 8$, takes 33 ms, so `CandidateK` returns 4, 7 and 8. B is the critical request with 8 ms of slack, and A holds 30 ms. To show how `Rank` orders requests whose survival differs, the example takes survival as known, as Table 2 does. A's survival at positions one to three is 0.92, 0.72 and 0.43, and B's is 0.52, 0.24 and 0.15. The shared curve would give both requests the same survival, and the fill would be breadth first. `Rank` puts B's first draft token first, because B is critical, then A's first, A's second and third, and B's second and third. Table 3 scores each plan. At $\rho = 1$, $\tau = 1.1 \times 20 + 8 = 30$ ms admits $K = 7$, which the throughput utility picks. If B's first draft token is then rejected, B's slack after the step is $8 + 1.1 \times 20 - 29 = 1$ ms, still within its cap. At $\rho = 0.5$, $\tau = 26$ ms admits only $K = 4$, and B keeps at least 6 ms whatever it accepts, more than the 4 ms Lemma A.1 requires. Without the floor, $K = 4$ would give both draft tokens to A.

A.6 SEVERAL INSTANCES

A request's KV cache and drafter state live on one decode instance, and every step of the request runs there. CADENCE therefore runs one controller per instance, on that instance's own profile, and no step spans instances. Under tensor or expert parallelism one instance spans several GPUs and runs one controller, as in the multi-GPU rows of Appendix D.2. Under prefill-decode disaggregation the controller runs on the decode instance, whose steps it times. A router that spreads requests across instances changes each instance's batch, and each instance enforces the cap for the requests it serves.

B THE LATENCY PROFILE

B.1 PROFILING

CADENCE reads step latency from a profile $L(B, \ell_{\max}, K)$ measured offline. Each point times whole steps over the span that `Track` charges: the drafter passes, verification, the catch-up pass and the host's decision. The AR step is $L_{\mathrm{AR}}(B) = L(B, 0, B)$. For the main configuration, the profile covers batch sizes 1, 8, 16, 32, 48, 64, 96, 112, 128, draft depths 0, 1, 3 and 7, and K in steps of 8 tokens up to $B(1 + \ell_{\max})$. Each point averages 20 timed steps after 5 warm-up steps. The 714 points take 16.0 minutes on one H200. Between profiled batch sizes, step latency is read linearly in B. `CandidateK` smooths dips within the profile's noise before it takes the right ends of flat stretches.

B.2 ACCURACY OF THE PROFILE

Table 4 checks the profile against the trace's measured steps. The error is 1.9% in every load regime and under every policy. The median step runs 0.05 ms over its profile, and the p99 step 1.2 ms under

Table 4: Profiled against measured step latency on the trace of Section 4.2: mean absolute percentage error (MAPE) in each phase, the overshoot of each step (measured minus profiled latency, or zero), and the share of steps whose overshoot exceeds $\varepsilon L_{\mathrm{AR}}(B)$ at $\varepsilon = 0.25$, the condition of Proposition A.2.

	error of the profile (MAPE, %)				overshoot per step (ms)			steps over
policy	low	high	moderate	all	median	p99	max	$\varepsilon L_{\mathrm{AR}}(B)$ (%)
CADENCE	1.9	1.9	1.9	1.9	0.05	1.2	20.5	0.18
SGL-SD	1.9	1.9	1.9	1.9	0.05	2.1	20.5	0.18
SGL-AR	1.9	1.9	1.9	1.9	0.04	1.1	20.6	0.17

CADENCE. The largest overshoots, about 20.5 ms, are host stalls, and they break the condition of Proposition A.2 on 0.18% of CADENCE's steps. The slack each request held absorbed every one of them: at $\varepsilon = 0.25$, no request under CADENCE ended over its cap (Section 4.2).

C EXPERIMENTAL SETUP DETAILS

This appendix gives the full setup behind Section 4.1.

Hardware and model. One NVIDIA H200 (141 GB) serves Qwen3-32B (revision `9216db57`) with thinking mode off, paired with the EAGLE-3 drafter `AngelSlim/Qwen3-32B_eagle3` (revision `74789e1a`), without tensor parallelism. Qwen3-32B has 64 layers and grouped-query attention with 8 KV heads of dimension 128, so one token of KV cache takes 256 KiB in bf16. A full batch of 112 requests holds about 90k tokens of KV cache, 23 GB, and no request is preempted.

Serving engine. We use SGLang 0.5.14 with its defaults: bf16 weights and KV cache, the FlashAttention-3 backend, CUDA graphs for decoding, overlap scheduling and the radix cache, with deterministic inference off. At most 112 requests run in a batch, and waiting requests are admitted in arrival order. Decoding is greedy. Every policy runs on the same engine, weights and arrivals, and policies differ only in how many draft tokens each step verifies and for which requests.

Workload. Each request draws its dataset with probability 0.25 each: chat from ShareGPT, summarization from CNN/DailyMail, math from GSM8K and code editing from InstructCoder (Table 5). Chat prompts keep the earlier turns of the conversation, summarization prompts hold the article and a one-line instruction, math prompts carry eight worked examples before the question, and code prompts hold the instruction and the code to edit. Prompts average about 700 tokens. Each request keeps its dataset's acceptance level and spread, with drift over its output. A request's output length is drawn from one lognormal law, whatever its dataset, with a median of 84 tokens, clipped to 16–1024 tokens (mean 100.6), and the request decodes exactly that many tokens. Output length is therefore unrelated to a request's acceptance.

Table 5: Prompt and output lengths per dataset, in tokens.

	prompt		output	
dataset	P50	P90	P50	P90
ShareGPT (chat)	640	1,780	84	181
CNN/DailyMail (summarization)	820	1,335	84	181
GSM8K (math)	832	867	84	181
InstructCoder (code)	150	324	84	181
All	759	1,169	84	181

Arrivals. Arrivals follow an Azure-derived trace, the Azure LLM Inference Trace 2023 (Microsoft, 2024; Patel et al., 2024). Its bursts and quiet periods last 8 s and 10 s on average, and a scaling factor sets its rate. Rates are multiples of λ, 33.8 requests/s, the rate at which a full batch of 112 requests of mean length 100.6 tokens completes under AR steps (32.9 ms each). Table 6 gives each load. A

steady-load run replays the trace at one of these loads. The trace of Section 4.2 replays it for 600 s and changes the scaling factor at 200 and 400 s. This gives three load regimes: low load, then high load at saturation's rates, then moderate load. A request belongs to the regime in which it arrives.

Table 6: Arrival rates of each load, in multiples of λ and in requests/s.

load	burst / quiet (multiple of λ)	mean	burst / quiet (requests/s)	mean (requests/s)
low load	0.6 / 0.06	0.3	20.3 / 2.03	10.2
near-capacity load	1.8 / 0.18	0.9	60.9 / 6.09	30.5
saturation and high load	2 / 0.2	1	67.7 / 6.77	33.8
moderate load	1.2 / 0.12	0.6	40.6 / 4.06	20.3

Runs and seeds. Each configuration runs ten seeds. A seed fixes the arrival times and every request's dataset, prompt, output length and class, so all policies serve the same requests. A steady-load run takes arrivals for 100 s, measures steps and arrivals after a 10 s warm-up and follows every measured request to its departure. The trace follows every request to its departure. Intervals are 95% t-intervals over seeds, paired by seed for ratios.

Tolerance and deadline. The tolerance $\varepsilon = 0.25$ lets a capped request take at most 25% longer than AR steps would take for the same tokens, plus $\beta = 40$ ms. At batch 112 an AR step takes 32.9 ms, so a capped request averages at most 41.1 ms per token, plus β spread over its output. This is well under the per-token targets of 100 to 200 ms that DistServe (Zhong et al., 2024) sets for chat, code and summarization. In units of AR decoding, whose step takes 28.8 ms at batch 112, the cap is a slowdown of 1.43. The allowance β is about 1.2 AR steps at batch 112. It lets a newly admitted request join speculative steps before it has built up slack. We also run the tighter $\varepsilon = 0.1$, a slowdown of 1.26. The deadline $d = 25$ ms for tight requests is chosen from the baselines alone: of the grid 20, 25, 30 and 35 ms, it is the value at which the best speculative baseline beats SGL-AR by the most (48.8 points). It lies between the step of AR decoding with one request, 19.0 ms, and with 112 requests, 28.8 ms. With a full batch, SGL-AR misses it and SD can meet it, so the deadline measures what speculation delivers under load. A deadline of five times the single-request step, 95 ms, the multiple Mooncake uses for its time-between-tokens threshold (Qin et al., 2025), would be met by at least 99% of tight requests under every policy at saturation and would separate none of them.

Metrics. *Throughput* is the output tokens committed in steps that start with requests waiting, divided by the summed measured latency of those steps. At saturation it averages nine of the ten runs, leaving out the run in which a policy we ran has such steps in under 5% of its steps, and under the trace's high load all ten. *TPOT* divides a request's time to departure by its output length, and each table states its clock: from arrival, queueing included, or from admission, the clock of the deadline. A request's *slowdown* is its decoding time over the time AR decoding would take to commit its tokens at the batch sizes it saw. The *worst slowdown* averages each run's largest. A request is *over the cap* if its measured decoding time at departure exceeds $(1 + \varepsilon)$ times its AR time plus β. Uncapped policies are judged against the cap at $\varepsilon = 0.25$.

Configurations. CADENCE uses the throughput utility, one shared offline acceptance curve, $\varepsilon = 0.25$, $\beta = 40$ ms and $\rho = 0.8$, with draft depths 0, 1, 3 and 7. SGL-AR turns the drafter off. SGL-SD runs SGLang's EAGLE-3 chain with `speculative_num_steps` 3, `speculative_eagle_topk` 1 and `speculative_num_draft_tokens` 4, verifying all $4B$ tokens. SGL-SD-D7 sets 7, 1 and 8 and verifies all $8B$. SGL-SD-CK^* drafts to depth seven and verifies at most $\min(K^*, 8B)$ tokens per step, filled breadth-first: every request's first draft token, then every second, and so on. We tuned K^* at saturation over nine caps (Table 7), and SGL-SD-C320 gave the highest throughput, with SGL-SD-C224 and SGL-SD-C256 within 0.2%.

D ADDITIONAL EXPERIMENTS

This appendix complements Section 4. Each subsection pairs its tables with the paragraphs that read them. Unless stated, runs serve Qwen3-32B on one H200 at saturation with ten seeds, and every

Table 7: Throughput of SGL-SD-CK^* at saturation for each cap K^*.

cap K^*	144	160	176	192	224	256	320	384	448
tokens/s	3,580	3,910	4,090	4,220	4,440	4,440	4,440	4,210	3,970

policy runs as Appendix C configures it. Apart from the tolerance sweep of Appendix D.5 and the per-step plans of Appendix D.4, CADENCE's variants appear only in Appendix D.9.

D.1 FULL RESULTS

Table 8: Every policy in each phase of the trace of Section 4.2, means over ten seeds. Slower is the share of requests slower than AR decoding. Goodput counts only the tokens of requests not slower than AR decoding. The high-load columns give throughput, goodput, mean step latency, mean queue, p99 slowdown and the p99 TPOT, from admission, of the quarter of requests with the lowest acceptance. Step latency averages every step of the phase, including those that run with a partly filled batch between bursts.

	low load	high load							moderate load
policy	slower (%)	throughput / SGL-AR	goodput / SGL-AR	step (ms)	queue length	slower (%)	p99 slowdown	low-acc. p99 TPOT (ms)	slower (%)
SGL-AR	0.0	1.00×	1.00×	24.9	215	0.0	1.00×	29.1	0.0
SGL-SD	0.02	1.03×	0.72×	39.5	184	31.4	1.57×	50.4	7.8
SGL-SD-D7	2.9	0.63×	0.03×	103.3	919	94.4	2.75×	87.9	93.2
SGL-SD-C320	2.0	1.14×	1.00×	38.4	128	14.7	1.30×	40.9	12.1
CADENCE	0.02	1.30×	1.29×	27.8	66	0.53	0.96×	29.8	0.10

The trace. Table 8 gives every policy in each phase. Under low load, the policies that draft to depth seven slow the most requests, SGL-SD-D7 2.9% and SGL-SD-C320 2.0%, against 0.02% under each of the others except SGL-AR. Under high load, SGL-SD's steps average 39.5 ms against 27.8 ms under CADENCE, and its queue holds nearly three times as many requests. Under moderate load, SGL-SD-D7 still slows 93.2% of requests and SGL-SD 7.8%, against 0.10% under CADENCE. Under high load, high-acceptance requests reach a p99 TPOT of about 22 ms under CADENCE. Over the whole trace, SGL-SD slows 18.6% of requests and CADENCE 0.31%.

Main comparison. For the steady-load runs, Table 9 compares CADENCE with the baselines, with each result's 95% interval over runs. Their TPOT runs from arrival, queueing included. Every share over the cap printed as 0 is zero in every run, and CADENCE also never passes its cap between steps.

Table 9: Main comparison in steady-load runs at $\varepsilon = 0.25$, with 95% intervals over runs in brackets. TPOT, from arrival, is under low load and near-capacity load. The other columns are at saturation. The worst slowdown averages each run's slowest request. Policies without a cap are judged against the cap at $\varepsilon = 0.25$.

	throughput	TPOT from arrival (ms)		slower than	worst	over the
policy	/ SGL-AR	low load	near capacity	AR (%)	slowdown	cap (%)
SGL-AR	1.00×	20.5 [20.1, 20.9]	88.5 [52.4, 124.6]	0	1.00×	0
SGL-SD	1.03× [1.02, 1.03]	9.1 [9.0, 9.2]	71.8 [39.1, 104.5]	29.0 [23.4, 34.6]	1.96× [1.89, 2.02]	2.0
SGL-SD-D7	0.63× [0.62, 0.63]	11.4 [10.9, 11.8]	345.5 [213.6, 477.3]	89.6 [83.3, 95.9]	3.38× [3.26, 3.51]	56.0
SGL-SD-C320	1.14× [1.14, 1.15]	11.3 [10.9, 11.8]	52.9 [34.0, 71.9]	14.2 [12.9, 15.5]	1.65× [1.58, 1.72]	0.2
CADENCE	1.30× [1.30, 1.30]	9.1 [9.0, 9.2]	29.3 [19.1, 39.5]	0.4 [0.3, 0.5]	1.14× [1.11, 1.18]	0

Slow class. Table 10 adds a slow class, 10% or 25% of requests with mean acceptance length 1.39 at draft depth seven, at $\varepsilon = 0.25$, which sets the cap for every policy. CADENCE puts no request over the cap in either cell. Judged against the tighter cap of $\varepsilon = 0.1$, SGL-SD puts 9.0% and 21.9% of requests over it.

Table 10: The slow class at saturation, at $\varepsilon = 0.25$. The first column gives its share of requests and the tolerance ε. Shares over the cap are of all requests and of the slow class.

slow class		throughput	over the cap (%)		slower	worst
(ε)	policy	/ SGL-AR	all	slow class	than AR (%)	slowdown
10% (0.25)	SGL-SD	0.95×	7.6	76.8	34.9	2.14×
	CADENCE	1.24×	0	0	5.1	1.24×
25% (0.25)	SGL-SD	0.86×	19.9	80.8	45.3	2.21×
	CADENCE	1.18×	0	0	13.5	1.28×

Classed traffic. In Table 11, a quarter of the requests are tight, capped at $\varepsilon = 0.25$ and given a per-token deadline d, measured from admission. The rest are best effort, with no cap. Under low load, every tight request meets $d = 25$ ms under CADENCE and SGL-SD alike.

Table 11: Classed traffic at saturation: throughput relative to CADENCE, the share of tight requests meeting each per-token deadline d in milliseconds, and the best-effort TPOT from admission at the 50th and 99th percentiles.

	throughput	tight requests meeting d (%)				best-effort TPOT
policy	/ CADENCE	$d = 20$	$d = 25$	$d = 30$	$d = 35$	p50 / p99 (ms)
SGL-AR	0.77×	7.1	14.4	99.9	100.0	28.5 / 29.1
SGL-SD	0.79×	24.4	49.9	77.8	91.6	24.2 / 44.1
SGL-SD-D7	0.48×	5.5	8.0	13.3	24.0	42.7 / 77.3
SGL-SD-C320	0.88×	20.1	63.2	91.4	98.0	23.4 / 36.4
CADENCE	1.00×	42.1	96.0	99.8	100.0	19.7 / 27.0

D.2 MODELS, HARDWARE AND PRECISION

Table 12: Configurations. Parameters are total and active. MoE rows give the routed experts and how many of them each token uses. DeepSeek-V3 and Llama-4-Maverick also run one shared expert for every token. Max batch is the batch each configuration runs at (Table 13).

model	total / active	type	attention	drafter	precision	GPUs	max batch
Qwen3-32B	32.8B / 32.8B	Dense	GQA	EAGLE-3	BF16	1×H200	112
Qwen3-32B	32.8B / 32.8B	Dense	GQA	EAGLE-3	FP8	1×H200	128
Qwen3-32B	32.8B / 32.8B	Dense	GQA	EAGLE-3	BF16	1×B200	160
Llama-3.3-70B	70.6B / 70.6B	Dense	GQA	EAGLE-3	INT8	2×A100	96
Qwen3-235B-A22B	235B / 22B	MoE 128/8	GQA	EAGLE-3	BF16	8×H100	304
gpt-oss-120b	116.8B / 5.1B	MoE 128/4	GQA, sliding window	EAGLE-3	MXFP4	1×H200	240
DeepSeek-V3	671B / 37B	MoE 256/8	MLA	MTP	FP8	8×H200	512
Llama-4-Maverick	400B / 17B	MoE 128/1	chunked local + global	EAGLE-3	FP8	8×B200	512

Table 13: Throughput relative to SGL-AR at saturation, with the share of requests slower than AR decoding in percent in brackets. Each configuration runs at the largest batch, in multiples of 16 up to its memory limit and 512, at which SGL-SD still gains at least 2% over SGL-AR. SGL-SD-CK^* uses the best cap K^* for each configuration. CADENCE puts no request over its cap in any configuration.

configuration	SGL-SD	SGL-SD-D7	SGL-SD-CK^*	K^*	CADENCE
Qwen3-32B, BF16, 1×H200	1.03× (29.0)	0.63× (89.6)	1.14× (14.2)	320	1.30× (0.4)
Qwen3-32B, FP8, 1×H200	1.03× (28.7)	0.66× (87.0)	1.08× (19.1)	368	1.22× (4.0)
Qwen3-32B, BF16, 1×B200	1.04× (27.0)	0.65× (87.5)	1.13× (13.7)	464	1.26× (0.6)
Llama-3.3-70B, INT8, 2×A100	1.03× (27.1)	0.64× (89.2)	1.09× (17.6)	272	1.25× (0.3)
Qwen3-235B-A22B, BF16, 8×H100	2.19× (0.0)	1.40× (4.5)	2.09× (0.0)	1216	2.19× (0.0)
gpt-oss-120b, MXFP4, 1×H200	1.02× (30.9)	0.63× (89.5)	1.13× (14.6)	688	1.28× (0.8)
DeepSeek-V3, FP8, 8×H200	1.56× (1.4)	0.98× (43.1)	1.50× (0.3)	1456	1.67× (0.6)
Llama-4-Maverick, FP8, 8×B200	1.53× (2.2)	1.21× (14.7)	1.46× (3.1)	2560	1.53× (2.1)

Table 13 repeats the comparison on seven more configurations (Table 12): Qwen3-32B in FP8 and on a B200, a dense 70B model in INT8 on two A100s, and four MoE models from 117B to 671B

parameters, with sliding-window, chunked and latent attention and, on DeepSeek-V3, its built-in MTP layer as the drafter. DeepSeek-V3 runs with SGLang's limits on running requests and on CUDA-graph batch size raised from 48 to 512. Acceptance levels differ by drafter, so rows are not comparable to each other. Batch sizes follow the caption's rule. On Qwen3-32B, Llama-3.3-70B and gpt-oss-120b, it puts SGL-SD where it barely gains in aggregate: SGL-SD gains 2 to 4% and slows 27.0 to 30.9% of requests. On Qwen3-32B in FP8 on the H200, CADENCE slows 4.0% of requests. FP8 makes each verified token cheaper against an AR step, so the fill chooses deeper budgets, and at batch 128 it sometimes fills depth three part way. The requests left with one draft token in these longer steps, those with low acceptance, pay for the depth that the others get. None of them passes the cap.

The three largest MoE models tell a different story. Each token uses a small share of their weights (22B of 235B, 37B of 671B and 17B of 400B parameters), and at their batch sizes SGL-SD's depth-three step costs far less over an AR step than on the other rows. gpt-oss-120b uses a small share too (5.1B of 117B), but SGL-SD slows 30.9% of requests there, so the share alone does not decide it. On the three largest models verification stays cheap, so few requests accept too little to pay for it, and SGL-SD's disparity nearly vanishes, as the externality argument of Section 2 expects: SGL-SD slows no request on Qwen3-235B-A22B, 1.4% on DeepSeek-V3 and 2.2% on Llama-4-Maverick. On Qwen3-235B-A22B the step latency starts to grow with K only past about 1,246 verified tokens, so at its memory-limited batch of 304, SGL-SD's depth-three plan fits below that point and is the plan CADENCE also chooses. Both reach 2.19 and slow no request. On Llama-4-Maverick the two also share one plan and reach 1.53. At these batch sizes the trap appears only under the full draft: SGL-SD-D7 slows 4.5%, 43.1% and 14.7% of requests on Qwen3-235B-A22B, DeepSeek-V3 and Llama-4-Maverick (Appendix D.4 shows it at larger batches). SGL-SD and CADENCE each slow about 2% of requests on Llama-4-Maverick (2.1% under CADENCE). Llama-4-Maverick's AR step, which runs the drafter's catch-up pass, costs 1.51 steps of AR decoding, a warm-drafter premium (Appendix A.3) far above the main configuration's 1.14. Slowdown is measured against AR decoding with the drafter off and the cap against AR steps, so these requests are slower than AR decoding and still within their caps.

On DeepSeek-V3, with its MTP drafter at a batch of 512, the knee lies near 1,271 tokens, between the 1,024 and 2,048 verified tokens of depths one and three for every request. CADENCE's per-request lengths can stop in between. CADENCE fills depth three part way and reaches 1.67, and it slows 0.6% of requests against SGL-SD's 1.4%.

D.3 BATCH SIZE

Table 14: Throughput relative to SGL-AR (thr.) and the share of requests slower than AR decoding, by the maximum batch B. Arrivals are at twice the capacity of AR steps at each B. Rows end at 128, the largest batch the profile covers.

	SGL-SD		SGL-SD-D7		CADENCE	
B	thr.	slower (%)	thr.	slower (%)	thr.	slower (%)
1	2.09×	0.0	1.89×	0.0	2.10×	0.0
8	2.07×	0.0	1.92×	0.0	2.09×	0.0
16	2.06×	0.0	1.88×	0.0	2.07×	0.0
32	2.07×	0.0	1.36×	5.7	2.07×	0.0
64	1.43×	2.4	0.86×	63.1	1.75×	0.0
96	1.12×	17.1	0.68×	85.1	1.42×	0.0
112	1.03×	29.0	0.63×	89.6	1.30×	0.4
128	0.96×	40.4	0.58×	91.2	1.21×	1.4

Table 14 caps the batch at B and scales saturation's arrivals to each B. Up to $B = 32$, SGL-SD and CADENCE make the same depth-three plan, about 2.07 times SGL-AR's throughput, and no request is slower than AR decoding. SGL-SD-D7 already slows 5.7% at $B = 32$, where its $8B = 256$ verified tokens pass the point, about 163 tokens, past which each verified token adds step latency. At $B = 64$, CADENCE fills depth three part way, between $2B$ and $4B$ verified tokens, and reaches 1.75. SGL-SD's slowed share grows with the batch, from 2.4% at $B = 64$ to 40.4% at 128, while CADENCE's stays at or below 1.4%.

D.4 LARGER BATCHES ON MOE MODELS

Table 15: MoE configurations as the batch B grows, on a closed batch of B requests at each configuration's acceptance level. Rows stop where the KV cache no longer holds B requests of 1,279 tokens, the 90th percentile of prompt plus output. The knee is the number of verified tokens past which each added token lengthens the step, and SGL-SD verifies $4B$ tokens per step. SGL-SD's step over SGL-AR's is the mean acceptance length below which a request decodes slower than under AR decoding. Throughput is relative to SGL-AR at the same B, with the share of requests slower than AR decoding in percent in brackets.

	SGL-AR	SGL-SD		throughput / SGL-AR (slower, %)	
B	tokens/ms	$4B$ / knee	step / SGL-AR's	SGL-SD	CADENCE
Qwen3-235B-A22B, BF16, 8×H100: knee 1,246 tokens, cache holds 193 requests					
64	2.4	0.21	1.15	2.35× (0.0)	2.41× (0.0)
128	4.6	0.41	1.16	2.32× (0.0)	2.40× (0.0)
Qwen3-235B-A22B, FP8, 8×H200: knee 885 tokens, cache holds 1,507 requests					
64	5.1	0.29	1.29	2.09× (0.0)	2.09× (0.0)
128	9.4	0.58	1.33	2.04× (0.0)	2.04× (0.1)
256	16.5	1.16	1.49	1.82× (0.3)	1.87× (0.3)
512	26.6	2.32	2.07	1.31× (8.5)	1.36× (9.7)
1,024	36.5	4.63	2.63	1.03× (41.6)	1.08× (38.8)
gpt-oss-120b, MXFP4, 1×H200: knee 417 tokens, cache holds 646 requests					
64	3.6	0.61	1.37	1.82× (0.2)	1.82× (0.2)
128	5.9	1.23	1.49	1.68× (1.1)	1.84× (0.3)
256	9.8	2.46	2.35	1.06× (38.8)	1.24× (1.6)
512	13.2	4.91	3.11	0.81× (89.1)	0.94× (55.1)
DeepSeek-V3, FP8, 8×H200: knee 1,271 tokens, cache holds 469 requests					
64	2.7	0.20	1.35	2.00× (0.0)	2.01× (0.1)
128	4.7	0.40	1.25	2.17× (0.0)	2.17× (0.0)
256	8.3	0.81	1.27	2.14× (0.0)	2.14× (0.1)
Llama-4-Maverick, FP8, 8×B200: knee 2,724 tokens, cache holds 3,433 requests					
64	9.7	0.09	2.01	1.28× (12.0)	1.28× (14.3)
128	14.8	0.19	1.75	1.46× (3.9)	1.46× (3.9)
256	23.5	0.38	1.59	1.62× (1.4)	1.62× (1.4)
512	39.5	0.75	1.66	1.55× (2.2)	1.55× (2.2)
1,024	66.3	1.50	2.20	1.17× (22.3)	1.20× (20.1)
2,048	101.3	3.01	3.14	0.82× (87.5)	0.85× (75.5)

Table 15 asks whether larger batches bring the trap to MoE models. Every request in a batch waits for the same step, so each one pays for the tokens the others verify, at any batch size. That cost becomes a loss for a request once SGL-SD's step over SGL-AR's exceeds the request's mean acceptance length. On a dense model this happens early: past Qwen3-32B's knee of 163 tokens, SGL-SD slows more than 10% of requests from $B = 96$ (Table 14). A sparse MoE model reads many expert weights per step for little computation per token, so its knee lies far out, at 417 to 2,724 tokens. Past the small-batch effect described below, more than 10% of requests become slower once B reaches 1.25 to 2.6 times a quarter of the knee. Such batches fit in memory on gpt-oss-120b, on Llama-4-Maverick and on Qwen3-235B-A22B in FP8 on eight H200s. They do not fit on Qwen3-235B-A22B in BF16 on eight H100s, whose 10% point, $B = 800$, lies past the 193 requests its cache holds, or on DeepSeek-V3, whose first slower requests appear at $B = 336$ but whose 10% point, $B = 768$, lies past the 469 requests it holds when every GPU stores the full latent cache. Sized at the mean length of about 790 tokens, as in Table 13, every cache holds about 1.6 times as many requests, which still leaves Qwen3-235B-A22B in BF16 short of its 10% point and DeepSeek-V3 at about it. MoE models also slow some requests at small batches: 19.4% on gpt-oss-120b and 10.6% on DeepSeek-V3 at $B = 16$, and 10.5 to 15.6% on Llama-4-Maverick up to $B = 64$, under every speculative policy. The verify pass's extra tokens reach experts that the batch alone would not, a cost that vanishes once the batch reaches nearly every expert ($B = 95$ on gpt-oss-120b, 382 on Llama-4-Maverick). Larger batches keep raising MoE throughput, but speculation's gain over SGL-AR shrinks past the knee and turns into a loss. On Llama-4-Maverick, SGL-AR grows from 9.7 tokens/ms at $B = 64$ to 101 at $B = 2{,}048$, where it beats every speculative policy and CADENCE reaches 86. Where the loss first appears, CADENCE drafts one token per request, which on gpt-oss-120b at $B = 256$ cuts the slowed share from 38.8% to 1.6%. Past the knee on Llama-4-Maverick and Qwen3-235B-A22B in FP8, it fills depth three part way, gains 3 to 4% over SGL-SD and slows a similar share. There the drafter's work per request makes even one draft token costly, and an AR step with the drafter running

costs 1.55 to 1.81 SGL-AR steps at $B = 1{,}024$. CADENCE's column is its per-step plan with each request's acceptance known, not the controller on a trace. Its delay cap moves no cell by more than about one point, except gpt-oss-120b at $B = 512$, where speculation already loses and the cap raises the slowed share to 76.4%.

D.5 TOLERANCE AND DEADLINE

Table 16: CADENCE at saturation for each tolerance ε: throughput, the share of steps where the allowed step latency τ changes the plan, the share of requests slower than AR decoding and over the cap, and the worst slowdown. The τ cell is blank at $\varepsilon = \infty$, where no request is capped.

tolerance ε	throughput / SGL-AR	τ binds (% of steps)	slower than AR (%)	over the cap (%)	worst slowdown
0	1.04×	48.9	47.3	21.8	1.15×
0.05	1.297×	1.0	0.4	0.0	1.13×
0.1	1.299×	0.3	0.4	0.0	1.14×
0.25	1.30×	0.0	0.4	0.0	1.14×
0.5	1.30×	0.0	0.4	0.0	1.14×
∞	1.30×		0.4	0.0	1.14×

Table 16 varies the tolerance. From $\varepsilon = 0.05$ up, the cap costs at most 0.2% of throughput, τ binds on at most 1.0% of steps, and no request ends over the cap. From $\varepsilon = 0.25$ up, τ never binds, and CADENCE matches the uncapped row. At $\varepsilon = 0$ an AR step earns no slack, so a GPU that runs slower than its profile drains the requests' slack for good, the case Proposition A.2 excludes. The mean hides two groups of runs. Runs whose GPU is 0.5% to 2.9% slower than its profile fall to 0.87–0.93 times SGL-AR's throughput and put up to 83% of requests over the cap, and the others keep 1.26–1.29. The main evaluation therefore runs $\varepsilon = 0.25$, with $\varepsilon = 0.1$ as the tight setting. At $\varepsilon = 0.25$, τ changes no step's plan, so the main-trace results do not depend on the cap.

The deadline grid of Table 11 runs from 20 to 35 ms. Between $d = 25$ and 30 ms, SGL-AR's share of tight requests meeting the deadline jumps from 14.4% to 99.9%, because its step with a full batch, 28.8 ms, lies between them. At 20 ms no policy meets the deadline for half of the tight requests, and at 35 ms SGL-AR and CADENCE meet it for every tight request. The rule of Appendix C therefore picks 25 ms, where speculation decides whether tight requests keep up.

D.6 ACCEPTANCE SPREAD

Table 17: Throughput relative to SGL-AR (thr.), the share of requests slower than AR decoding and the share over the cap at $\varepsilon = 0.25$ (both in percent), by how widely acceptance varies across requests. The spread is the coefficient of variation (CV) of each request's mean acceptance length at draft depth seven.

	spread	SGL-SD			CADENCE		
workload	(CV)	thr.	slower	over	thr.	slower	over
narrow	0.10	1.02×	27.5	0.0	1.28×	0.0	0.0
evaluation mix	0.24	1.03×	29.0	2.0	1.30×	0.4	0.0
10% slow class	0.18	0.95×	34.9	7.6	1.24×	5.1	0.0
25% slow class	0.26	0.86×	45.3	19.9	1.18×	13.5	0.0
50% slow class	0.34	0.74×	62.6	42.0	1.09×	29.0	0.0

Table 17 varies how widely acceptance differs across requests. The narrow workload has little spread, the evaluation mix is the workload of Section 4, and the slow-class workloads give 10, 25 or 50% of requests a mean acceptance length of 1.39 at depth seven. CADENCE's throughput over SGL-SD's is lowest on the narrow workload, 1.25, and grows with the slow-class share, to 1.48 with a 50% slow class, and SGL-SD puts more requests over the cap as the slow class grows, while CADENCE puts none. Under CADENCE, slow-class requests can still be slower than AR decoding, which the cap allows: it bounds how much slower a request runs, not whether.

D.7 SHARE OF LOW-ACCEPTANCE REQUESTS

Table 18: The trace of Section 4.2 with a share of requests in the slow class (mean acceptance length 1.39 at draft depth seven). The other requests are the evaluation mix. Cells give throughput relative to SGL-AR, over steps that start with requests waiting, and in brackets the share of requests slower than AR decoding. Throughput is printed where such steps make up at least 5% of the regime in at least eight of ten runs, and as – elsewhere. Over the cap is at $\varepsilon = 0.25$. CADENCE puts no request over it in any cell. The last two columns give the share of evaluation-mix requests slower than AR decoding.

		throughput / SGL-AR (slower, %)		over the cap (%)	others slower (%)	
low-acc. share	regime	SGL-SD	CADENCE	SGL-SD	SGL-SD	CADENCE
0%	Low load	– (0.0)	– (0.0)	0.0	0.0	0.0
	Moderate load	– (7.8)	– (0.1)	0.4	7.8	0.1
	High load	1.03× (31.4)	1.30× (0.5)	2.3	31.4	0.5
	Aggregate	1.03× (18.6)	1.30× (0.3)	1.3	18.6	0.3
5%	Low load	– (0.1)	– (0.1)	0.0	0.0	0.0
	Moderate load	– (14.3)	– (0.6)	2.2	12.0	0.1
	High load	0.99× (35.2)	1.28× (3.3)	6.3	32.2	0.6
	Aggregate	0.99× (22.7)	1.27× (1.9)	3.9	20.4	0.3
10%	Low load	– (0.2)	– (0.1)	0.0	0.0	0.0
	Moderate load	– (19.8)	– (1.3)	4.6	14.7	0.1
	High load	0.95× (39.1)	1.25× (6.2)	10.6	33.1	0.6
	Aggregate	0.95× (26.5)	1.25× (3.6)	6.9	21.8	0.3
25%	Low load	– (0.5)	– (0.4)	0.0	0.0	0.0
	Moderate load	– (35.4)	– (3.7)	14.7	21.0	0.1
	High load	0.86× (49.6)	1.19× (15.4)	23.8	34.5	0.6
	Aggregate	0.86× (37.0)	1.19× (9.2)	17.0	24.5	0.4
50%	Low load	– (1.2)	– (0.9)	0.1	0.1	0.0
	Moderate load	– (61.5)	– (7.7)	40.9	32.0	0.1
	High load	0.73× (67.3)	1.10× (31.5)	47.7	37.0	0.7
	Aggregate	0.73× (54.7)	1.10× (18.8)	37.8	29.4	0.4
75%	Low load	– (2.4)	– (1.6)	0.1	0.1	0.0
	Moderate load	0.64× (82.5)	1.02× (19.3)	69.4	37.2	0.3
	High load	0.64× (83.5)	1.02× (48.6)	71.8	38.1	0.7
	Aggregate	0.64× (69.9)	1.02× (31.6)	59.3	31.6	0.4

Table 18 replays the trace while a growing share of requests joins the slow class of Table 17. The other requests keep their acceptance, and every policy sees the same arrivals. Under low load no queue forms and the policies differ little. Under high load the gap between SGL-SD and CADENCE widens with the share. CADENCE's throughput over SGL-SD's grows from 1.26 at 0% to 1.59 at 75%, and SGL-SD's share over the cap grows from 2.3% to 71.8%, while CADENCE puts no request over it. The gap is already open at 0%, because the evaluation mix alone spans a wide range of acceptance, and there CADENCE slows 30.8 points fewer requests. The last two columns show the externality directly. Slow requests lengthen SGL-SD's shared steps, so the other requests slow down as well: under high load, the share of evaluation-mix requests slower than AR decoding rises from 31.4% to 38.1%. Under CADENCE it stays between 0.5 and 0.7%. Under moderate load, from a share of 25% on, SGL-SD no longer keeps up with the bursts. Its mean queue grows from 9.6 requests at 0% to 949 at 75%. The allowed step latency τ binds on no step, so CADENCE's zero over the cap comes from the plans it chooses, not from the cap.

D.8 DRAFTER QUALITY

Table 19 changes the drafter instead of the mix. Each request's acceptance probability at every draft position is raised to one common power, chosen to reach each mean acceptance length, so every request keeps its place in the mix while the whole mix moves, as with a stronger or weaker drafter. Read from the top, the gap to SGL-SD is nearly absent when acceptance is high and widens as it falls. At a mean of 4.50, SGL-SD's fixed depth of three is close to the best plan, and CADENCE's throughput is 1.01 times SGL-SD's. At 1.50 it is 1.67 times, and no speculative policy beats AR decoding. CADENCE stays at 0.98 times SGL-AR's throughput and never takes an AR step, because its own AR step still runs the drafter's catch-up pass (Appendix A.3), so a step with one draft token per request costs little more.

Table 19: The trace of Section 4.2 with every request's acceptance raised or lowered together, by the mean acceptance length at draft depth seven over all requests. The 2.78 row is the evaluation mix as it is. Throughput relative to SGL-AR and, in brackets, the share of requests slower than AR decoding are under high load. The middle columns give the slowed share over all three regimes, and the last the p99 TPOT from admission under high load, against 29.1 ms for SGL-AR. CADENCE puts no request over the cap in any row.

mean acc.	high load throughput / SGL-AR (slower, %)		all regimes slower (%)		p99 TPOT, high load from admission (ms)	
length	SGL-SD	CADENCE	SGL-SD	CADENCE	SGL-SD	CADENCE
4.50	1.39× (0.4)	1.40× (0.0)	0.2	0.0	27.0	22.4
3.50	1.21× (7.0)	1.36× (0.0)	3.6	0.0	34.9	24.4
2.78	1.03× (31.4)	1.30× (0.5)	18.6	0.3	45.2	27.6
2.50	0.94× (48.4)	1.26× (1.6)	32.5	1.0	50.3	29.6
2.00	0.77× (79.4)	1.15× (10.4)	63.2	6.6	60.8	34.6
1.50	0.59× (96.6)	0.98× (47.2)	84.5	37.7	69.1	39.3

D.9 PROFILE BIAS AND THE DELAY LEDGER

Table 20: CADENCE at saturation with a biased profile, at pace $\rho = 0.8$: throughput relative to SGL-AR, with the requests over the cap in brackets, counted against the profiled AR step and against the true one, out of 28,171 requests. The profile states every latency, the AR step's included, as $1 + b$ times the true one. Delay is tracked on measured latency, as in CADENCE, or on profiled latency.

		delay	profile bias b				
workload	ε	tracked on	-10%	-5%	0	$+5\%$	$+10\%$
evaluation mix	0.1	measured	1.00 (10,455/0)	1.30 (0/0)	1.30 (0/0)	1.30 (0/0)	1.30 (0/0)
evaluation mix	0.1	profiled	1.30 (4/0)	1.30 (1/0)	1.30 (0/0)	1.30 (0/0)	1.30 (0/0)
evaluation mix	0.05	measured	0.88 (24,981/0)	1.01 (7,227/0)	1.30 (0/0)	1.30 (0/1)	1.30 (0/1)
evaluation mix	0.05	profiled	1.30 (27/0)	1.30 (4/0)	1.30 (0/0)	1.30 (0/0)	1.30 (0/0)
25% slow class	0.1	measured	0.96 (11,348/0)	1.17 (6/0)	1.18 (0/0)	1.18 (0/3)	1.18 (0/3)
25% slow class	0.1	profiled	1.18 (325/2)	1.18 (45/2)	1.18 (2/2)	1.18 (0/3)	1.18 (0/2)

Table 20 plans on a profile that is off by a factor $1 + b$, so $b < 0$ means that steps run slower than the profile says. We call the accounting of Equation (2) the delay ledger. The measured ledger charges each step's measured latency, as CADENCE does, and the profiled ledger its profiled latency. Rows that track delay on profiled latency run CADENCE with the profiled ledger. The cap is counted twice: against the profiled AR step, which is the cap the operator writes down, and against the true AR step. At $\varepsilon = 0.25$, every bias leaves throughput at 1.30 and no request over either cap. At $\varepsilon = 0.1$ and $b = -10\%$, an AR step's credit is $(1 + \varepsilon)(1 + b) = 0.99$ times its true cost, so the measured ledger only drains and holds CADENCE at AR steps. Throughput falls to SGL-AR's, and no request passes the true cap. More than a third pass the profiled cap, which even AR steps cannot meet on a GPU slower than its profile. Tracking delay on profiled latency never sees the error, so it keeps throughput and puts fewer requests over the profiled cap. With a pessimistic profile ($b > 0$) the measured ledger over-credits, and a few requests pass the true cap, by at most 76 ms. The pace matters little here. Setting ρ to 0.5 or 1 changes throughput by at most 0.012 in every cell. An operator should pick ε above the profile's optimism: with the measured ledger, $(1 + \varepsilon)(1 + b)$ must exceed one.

D.10 CO-LOCATED PREFILL

The main evaluation times decode steps only. Here one server also prefills arriving prompts. SGLang 0.5.14 does not mix prefill chunks into an SD step. It runs prefill as its own iteration between decode steps. CADENCE's ledger times decode steps only, so a prefill iteration enters no request's slack, and AR decoding would wait through the same iteration. A 690-token prompt takes 71 ms to prefill, and prefill takes 73% of GPU time. A co-located SGL-AR completes 10.3 requests/s at batch 112, so the trace's rates are multiples of that capacity, with the same three phases. Under low load no queue forms, and TTFT is set by prefill: a P50 of 100 ms under SGL-AR and up to 105 ms under the SD policies, whose drafter also prefills each prompt. Prefill takes most of the GPU's time, so SD's gain

Table 21: One server that also prefills arriving prompts (Qwen3-32B, one H200, the trace's phases at rates relative to this server's capacity): throughput relative to SGL-AR in steps that start with requests waiting, prefill included, and time to first token (TTFT) from arrival.

	throughput	TTFT, low load (ms)		TTFT, high load (s)	
policy	/ SGL-AR	P50	P99	P50	P99
SGL-AR	1.00×	100	516	24.6	80.1
SGL-SD	1.01×	104	556	24.0	78.2
SGL-SD-D7	0.87×	105	561	41.4	115.4
SGL-SD-C320	1.02×	105	561	23.0	75.6
CADENCE	1.04×	105	552	21.5	71.1

shrinks to 1.04 under CADENCE and 1.01 under SGL-SD, and SGL-SD-D7 falls to 0.87. Under high load, TTFT is queueing and follows throughput: a P50 of 21.5 s under CADENCE, against 24.6 s under SGL-AR and 41.4 s under SGL-SD-D7.

D.11 PER-STEP OVERHEAD

Table 22: CADENCE's decision time per step, and its share of the AR step at each configuration's max batch.

	max	decision time (ms)		AR step	share of an AR step (%)	
configuration	batch	median	p99	(ms)	median	p99
Qwen3-32B, BF16, 1×H200	112	0.4	1.2	32.9	1.2	3.6
Qwen3-32B, FP8, 1×H200	128	0.4	1.2	23.8	1.7	5.0
Qwen3-32B, BF16, 1×B200	160	0.5	1.3	23.1	2.2	5.6
Llama-3.3-70B, INT8, 2×A100	96	0.4	1.3	45.3	0.9	2.9
Qwen3-235B-A22B, BF16, 8×H100	304	0.6	1.5	37.2	1.6	4.0
gpt-oss-120b, MXFP4, 1×H200	240	0.5	1.3	32.6	1.5	4.0
DeepSeek-V3, FP8, 8×H200	512	0.7	1.6	45.5	1.5	3.5
Llama-4-Maverick, FP8, 8×B200	512	0.7	1.6	19.5	3.6	8.2

CADENCE's decision time grows with the batch and the candidate grid, not with the GPU. On the main configuration it takes 0.4 ms at the median and 1.2 ms at p99, 1.2% and 3.6% of an AR step. At batch 512 it reaches 0.7 ms and 1.6 ms. The largest share is Llama-4-Maverick's 8.2% at p99, because its AR step takes only 19.5 ms. The decision time is host time within the step, so the profile includes it (Appendix A.1).

E EXTENDED RELATED WORK

This appendix extends Section 5 by topic. Where the difference is clear, each paragraph ends with how CADENCE differs.

SD methods and drafters. SD drafts several tokens cheaply and verifies them in one target-model pass, and rejection sampling keeps the target model's output distribution (Leviathan et al., 2023; Chen et al., 2023). Medusa adds decoding heads to the target model (Cai et al., 2024). EAGLE drafts at the feature level (Li et al., 2024a), EAGLE-2 grows context-dependent draft trees from the drafter's confidence (Li et al., 2024b), and EAGLE-3 drafts tokens from fused features of several target layers (Li et al., 2025). SpecInfer verifies a token tree in one pass (Miao et al., 2024), Sequoia sizes the tree for a token budget and a hardware platform (Chen et al., 2024), and BASTION grows each query's tree while the marginal gain, estimated from path confidence, covers the added verification cost (Oh et al., 2026). Online speculative decoding updates the drafter from verification feedback (Liu et al., 2024c), and HedgeSpec chooses among drafters online by scoring each on the committed tokens (Liu et al., 2026a). Yan et al. (2025) find that the speedup depends heavily on the drafter's latency. The speedup also differs across tasks (Sandler et al., 2025) and shrinks as the batch grows, while acceptance varies within requests, across requests and across datasets (Liu et al., 2026c).

SPEED-Bench finds that the best draft length depends on the batch size (Abramovich et al., 2026). CADENCE takes the drafter as given and, for each shared step, sets the draft depth and how many draft tokens each request verifies.

Adaptive speculation length for one request. SpecDec++ treats stopping the draft as a Markov decision process, shows that it is optimal to stop once the probability that some draft token is rejected passes a threshold, and estimates that probability with a trained acceptance head (Huang et al., 2025a). AdaEDL stops drafting when an entropy-based lower bound on acceptance falls below a threshold (Agrawal et al., 2024b). EESD samples from a Beta posterior to decide whether to keep drafting and updates it only with the draft tokens that verification reaches (Liu et al., 2024b). ASPIRE runs drafting and verifying requests in one forward pass, and each request chooses when to verify from a moving average of its verified acceptance, which starts from a prior, and a batch-aware cost model (Ziashahabi et al., 2026). FASER gives each request the length that minimizes the step latency per token that request commits, searching online with a Gaussian-process lower confidence bound, and notes that each trial affects serving latency (Chen et al., 2026a). CADENCE instead reads one shared offline acceptance curve and chooses every request's length within one allowed step latency that the whole batch shares.

Adaptive speculation for the batch. Su et al. (2023) find that the best speculation length depends on the batch size and choose it by batch size. In its serving experiment, BanditSpec picks one length per step for the batch as a bandit arm (Hou et al., 2025). Nightjar learns a batch-wide length per batch size and notes that once speculation is off, no new speculative observations arrive (Li et al., 2026a). Other allocators rank draft tokens across the batch. Under a fixed capacity, TETRIS verifies the draft tokens with the highest cumulative acceptance, which it proves optimal for one step given the true acceptance rates and no drafting time, and over a whole run when all tokens of a request share one rate (Wu et al., 2025). D-cut and DSpark choose the total budget by weighing the drafter's confidence against a step cost profiled at startup, and fill it in order of confidence (Liu et al., 2026b; Cheng et al., 2026). ECHO treats the batch as one draft tree under a global verification cap (Hu et al., 2026). CADENCE also fills a budget in order of expected benefit, but admits only budgets whose profiled latency fits the allowed step latency that the least slack among capped requests sets.

Deadline-aware speculation. AdaSpec rejects any plan whose step time exceeds the per-token latency target, because some requests may accept only one token (Huang et al., 2025b). AdaServe first serves the requests furthest behind their per-token deadlines, then fills the rest of a profiled token budget by path probability (Li et al., 2026c). SLOs-Serve sets one speculation length per service tier and counts expected accepted tokens against each tier's target latency per token (Chen et al., 2025). WISP batches verification from edge drafters under every included request's deadline (Li et al., 2026b). GoodSpeed splits one verifier's capacity among draft servers by the gradient of a concave utility of smoothed goodput (Tran et al., 2026). CADENCE admits each step by counting only the one token it is sure to commit per request, and carries each request's delay across steps, so slack that accepted tokens earn can lengthen later steps.

Batched SD mechanics and correctness. BASS handles the ragged tensors that variable acceptance creates with custom attention kernels and adjusts the draft length each step (Qian et al., 2024). Zhang et al. (2026) find that several batched SD implementations silently corrupt outputs because variable acceptance desynchronizes position IDs, attention masks and KV caches, and they state the invariants that batched inference must keep. Kong et al. (2026) model SD latency in serving and explain why speedups often shrink as load rises. D-cut and DSpark each show that a rule reading the sampled draft token's own confidence can change the output distribution, and each avoids this with a rule that does not read that confidence, the condition DSpark names the non-anticipating property (Liu et al., 2026b; Cheng et al., 2026). BanditSpec proves that any rule choosing speculation settings from the history keeps the output distribution (Hou et al., 2025). CADENCE chooses every length before drafting and claims output preservation in distribution only (Appendix A.3).

Serving with latency targets, fairness and slack. Under continuous batching (Yu et al., 2022; Kwon et al., 2023; Zheng et al., 2024), every running request advances in one decode step. Sarathi-Serve chunks prefills and sets, by one-time profiling, a per-iteration token budget that keeps decode steps within a time-between-tokens target (Agrawal et al., 2024a). DistServe places prefill and

decoding on different GPUs to meet both latency targets (Zhong et al., 2024). FairBatching sets each iteration's time budget from its decode requests' next-token deadlines and gives prefill the time they do not need (Lyu et al., 2025). QoServe picks the largest prefill chunk that misses no deadline (Goel et al., 2026), and Kairos packs requests whose step time fits the least cumulative slack in the batch (Wang, 2026). SlidingServe plans two iterations at once and keeps slack for the second, reporting that spending all of it in one iteration severely limits the next (Chen et al., 2026b). TAPER admits the extra branches of branch-parallel requests while a fixed fraction of the batch's least deadline slack absorbs their estimated latency (Gandhi et al., 2026). Andes counts tokens delivered ahead of a user's reading speed as slack (Liu et al., 2024a), and VTC divides service fairly among clients by counting the tokens each has received (Sheng et al., 2024). CADENCE's slack is instead the delay each request may still take relative to AR steps at the load it saw, and accepted draft tokens refill it.

Online learning and utility-driven allocation. Cilantro learns each job's performance online and allocates resources to maximize a user-defined welfare of per-job utilities, using optimism under uncertainty (Bhardwaj et al., 2023). CADENCE's operator utilities follow this form. Bandits with knapsacks stop when a budget runs out (Badanidiyuru et al., 2018). Replenishable knapsacks allow negative costs and a void action that consumes no budget (Bernasconi et al., 2024). Anytime knapsacks require the budget to hold at every round, and SUAK slightly under-uses it to skip fewer rounds (Elumar et al., 2025). Conservative bandits keep cumulative reward above a fraction of a known baseline's reward (Wu et al., 2016; Kazerouni et al., 2017), the stage-wise version applies that constraint at every round (Moradipari et al., 2020), and a shield blocks unsafe actions while a reinforcement learner explores (Alshiekh et al., 2018). Cascading bandits observe a ranked list up to the first click (Kveton et al., 2015a), the conjunctive variant observes a chosen tuple up to its first failed item (Kveton et al., 2015b), and bandits with probabilistically triggered arms observe a base arm only when the round's outcomes trigger it (Chen et al., 2016; Wang & Chen, 2017). Verification likewise reveals a request's draft tokens up to its first rejection, while speculation's reward counts the accepted prefix. CADENCE does not learn acceptance online. It reads one shared offline curve, and its allowed step latency depends on the latency profile and on slack, never on the curve, so a poor curve can waste speculation but cannot break a cap.

Shared-resource interference in systems. Processors throttle work that crowds out other threads. ICOUNT gives fetch priority to the simultaneous-multithreading threads with the fewest instructions in decode, rename and the instruction queues (Tullsen et al., 1996), pipeline gating stops fetch while too many low-confidence branches are unresolved (Manne et al., 1998), and prefetcher control throttles prefetchers that harm other cores (Ebrahimi et al., 2009). Source throttling estimates each application's slowdown relative to running alone and throttles the cores that cause unfairness (Ebrahimi et al., 2010). Heracles grows best-effort tasks only while a latency-critical service keeps slack against its tail-latency target (Lo et al., 2015), and Retro throttles tenants whose use of a shared resource slows others (Mace et al., 2015). In networks, a token-bucket envelope bounds a flow's traffic in any interval (Cruz, 1991), and TCP pacing can lower throughput because it delays congestion signals (Aggarwal et al., 2000). CADENCE measures each request's slowdown against AR steps at the load it saw rather than against running alone, because co-batched requests share every step.

AI USE STATEMENT

LLMs play two roles in this work. They are the workload: the models we serve (Qwen3-32B, Llama-3.3-70B, Qwen3-235B-A22B, gpt-oss-120b, DeepSeek-V3 and Llama-4-Maverick) and their drafters serve the requests we schedule, and CADENCE is a model-agnostic controller that does not modify or depend on model internals. LLMs were used only for editing assistance (grammar, spelling, word choice) and did not impact the core methodology, scientific rigor, or originality of the research.