Regulating Branch Parallelism in LLM Serving

Swapnil Gandhi[1] **Siva Hari**[2] **William J. Dally**[1,2] **Christos Kozyrakis**[1,2]
[1]Stanford University [2]NVIDIA

Abstract

Recent methods expose intra-request parallelism in LLM outputs, allowing independent branches to decode concurrently. Existing serving systems execute these branches eagerly or under fixed caps. We show that both are brittle: eager admission inflates the shared decode step, degrading co-batched requests in serial stages, while conservative fixed caps forgo the throughput that motivated exposing branches in the first place. We call the excess step latency caused by admitted branches the *branch externality* and show that the safe width depends on batch composition, context lengths, and accumulated slack, all of which change continuously over a workload trace. We introduce TAPER, a per-step admission controller that treats extra branches as opportunistic work, admitted only when the predicted branch externality fits within the batch's current slack budget. Per-step regulation is practical because branch-level scheduling decouples compute from memory: branches share the request's prefix KV, so expanding or contracting width requires no memory reclamation. On Qwen3-32B, TAPER improves goodput by $1.77\times$ over IRP-OFF and by $1.48\times$ over IRP-EAGER, while maintaining over 95% SLO attainment.

1 Introduction

As LLM responses grow longer, driven by chain-of-thought reasoning [1], structured generation [2], and multi-step tasks [3], the sequential nature of autoregressive decoding becomes an increasingly visible bottleneck. A growing body of work addresses this by exposing *intra-request parallelism* (IRP): decomposing a single response into independent branches that decode concurrently before a reduce phase combines them [4–10]. For an isolated request the appeal is immediate: parallel execution shrinks the critical path from the sum of branch lengths toward the length of the slowest branch, converting otherwise idle decode capacity into useful progress [11, 12].

The difficulty begins when this private speedup is realized inside a shared serving system. Under continuous batching [13], the engine executes one forward pass per decode step across all active sequences, advancing each sequence by one token. Step latency grows with the number of sequences and their aggregate context length [14, 15]. A request in a parallel stage contributes multiple branch sequences to the same step, widening the batch for everyone. The parallel request is compensated: it receives progress on several branches. A co-batched request in a serial stage receives only its single next token, yet waits for the same inflated step. We call this excess latency the *branch externality*. In §2, we show that under moderate and high load, eager branch admission can raise aggregate token throughput while simultaneously degrading goodput and SLO attainment.

Simple remedies do not resolve this tension. Disabling IRP eliminates the interference but forfeits the throughput and latency gains that motivated exposing branches in the first place. Fixed caps, admitting at most k branches per step, help in one operating regime and hurt in another, because the safe width depends on batch composition, context lengths, and accumulated slack, all of which change continuously over a workload trace. What is needed is not another static policy, but a per-step

Preprint.

admission controller that widens when the current batch has headroom and contracts when requests in serial stages are at risk.

Historically, such fine-grained regulation has been impractical because the natural scheduling unit is the request [16, 17]. Revising a request-level decision entangles compute and memory: pausing a request forces the scheduler to either keep its KV state resident, consuming scarce memory, or evict it, incurring restoration cost when the request resumes [18, 19]. Frequent per-step revisions are therefore expensive to undo.

Our central observation is that branch-level scheduling separates these concerns. During a parallel phase, all branches share the request's prefix KV [9]. A branch's own KV consists only of the tokens it has generated so far, typically a small fraction of the shared prefix [6]. When the scheduler defers a branch from a given step, it does not reclaim the prefix (which remains resident for the sibling branches that are still admitted) and it does not evict the branch-local KV (which is small enough to leave in place). Admitting the branch in a subsequent step requires no state restoration: the scheduler simply adds it back to the next forward pass. The residual cost is that a deferred branch's branch-local KV remains resident while idle, but this is bounded by the tokens the branch has generated, and the memory is accounted to the parent request. If KV cache pressure requires eviction, the serving system preempts the entire request through its normal eviction policy. Branch regulation therefore adjusts compute allocation without triggering memory reclamation or restoration. It is cheap, voluntary, and reversible. This structural property is what makes per-step width regulation practical.

We introduce TAPER, a runtime controller that exploits this asymmetry. TAPER first protects *single-token progress*: every active request contributes exactly one token to the next step. It then treats additional branches as *opportunistic width*, admitting them only when the predicted branch externality fits within the batch's current slack budget. When slack is abundant, TAPER expands toward eager execution; as requests in serial stages approach their latency targets, it contracts toward single-token progress. Because contraction amounts to declining branch admissions rather than evicting requests, TAPER adapts every step without policy switches, preemption machinery, or fallback logic.

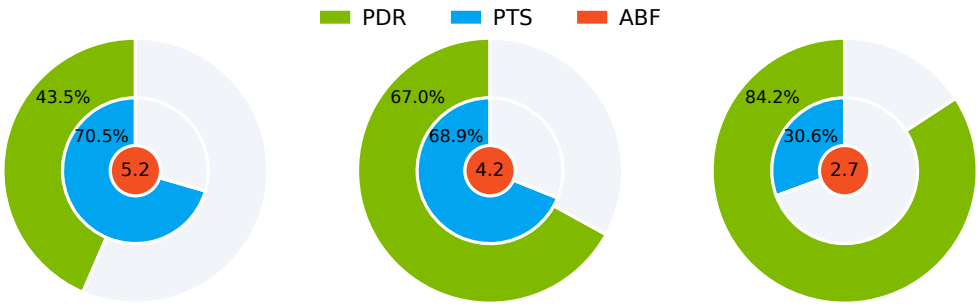
We make the following contributions:

- We characterize the *branch externality*: requests in parallel stages benefit from widened decode steps while co-batched requests in serial stages pay the inflated latency without receiving additional progress, making eager and fixed-width policies brittle across operating points (§2).
- We show that branch-level scheduling *decouples compute regulation from memory regulation*: branches share the prefix KV and have small branch-local state, so step width can be expanded or contracted per step without eviction or restoration. This is the structural property that makes per-step control feasible (§3.5).
- We introduce TAPER, a per-step controller that predicts the branch externality, bounds it against the batch's slack budget, and allocates opportunistic width via a greedy planner with a pluggable utility interface (§3).
- We evaluate TAPER on Qwen3-32B under mixed workloads, showing that it improves goodput over IRP-OFF by $1.77\times$ and over IRP-EAGER by $1.48\times$, while maintaining over 95% SLO attainment where eager drops below 50% (§4).

TAPER reframes intra-request parallel text generation as an admission control problem. Existing techniques show how to expose parallelism within a request [12]; TAPER decides how much of that parallelism to realize in each shared decode step. By exploiting the cheap reversibility of branch deferral, it harvests intra-request parallelism when the batch has slack and falls back to protected single-token progress when it does not.

2 Characterizing Intra-Request Parallelism

Intra-Request parallelism can substantially accelerate individual requests. This section shows that it also creates a cost that falls on other requests: under realistic mixed workloads, the acceleration of requests in parallel stages comes at the expense of co-batched requests in serial stages, and no fixed policy navigates the trade-off across operating points.



(a) ShareGPT Vicuna (b) RAG-12K Dataset (c) Math-220K Dataset

Figure 1: **Intra-request parallelism across workloads.** Proportion of decomposable requests (PDR), parallel token share (PTS), and average branch fanout (ABF) for three datasets.

2.1 Intra-Request parallelism in the wild

Several recent methods shorten the critical path of LLM decoding by exposing independent branches within a single response. Skeleton-of-Thought [4] expands outline points concurrently; APAR [6] emits explicit branch tokens; PASTA [5] introduces asynchronous promises; ASPD [8] trains branch-invisible attention masks for native parallel decoding; and Multiverse [9] exposes Map/Process/Reduce control flow. These methods differ in how branches are discovered and represented, but they produce the same serving-visible structure [8].

Output generation comprises a sequence of interleaved stages, each decoding in serial or parallel mode. In serial stages, generation proceeds through a single continuation in the usual autoregressive manner. During parallel stages, the model simultaneously decodes multiple independent branches, enabling concurrent token generation, while each branch internally still generates tokens autoregressively. A reduce phase combines the completed branches before generation continues, possibly entering another parallel stage later in the response.

We call a request *decomposable* if it enters at least one parallel stage during its lifetime, and *non-decomposable* otherwise. At any given decode step, a decomposable request may be in either a serial or a parallel stage. A request in a serial stage, whether decomposable or not, contributes a single continuation token and is indistinguishable from a non-decomposable request. The number of branches exposed during a parallel stage is the request's *branch fanout* n_r.

Prevalence. *How common are decomposable requests?* We characterize three representative datasets (Figure 1). The *proportion of decomposable requests* (PDR) measures how often parallelism appears: Math-220K has the highest PDR at 84.2%, followed by the RAG-12K dataset at 67.0% and ShareGPT Vicuna at 43.5%. The *parallel token share* (PTS) measures how much of a decomposable response is parallel: ShareGPT [20] (70.5%) and RAG-12K [21] (68.9%) are dominated by parallel tokens when they decompose, while Math-220K [22] (30.6%) produces short, narrow parallel stages. The *average branch fanout* (ABF) determines the maximum step width the runtime could admit: ShareGPT averages 5.2 branches per parallel stage, RAG-12K 4.2, and Math-220K 2.7.

The pattern is not uniform. Math reasoning decomposes frequently but narrowly: many parallel stages, each with few short branches covering a small share of the output. Structured tasks like ShareGPT and RAG-12K decompose less often but more deeply, with wider branching and a larger share of tokens in parallel stages. What determines the severity of the branch externality (§2.3) is not PDR alone but the combination of ABF and PTS: a high-PDR, low-ABF workload creates many small externalities; a moderate-PDR, high-ABF workload creates fewer but larger ones.

Implications for serving. Decomposable requests are common enough to matter at scale across all three workload types. Yet even in workloads with high PDR, requests spend a significant fraction of their lifetime in serial stages (PTS is always below 71%). The typical batch will therefore contain a mix of requests in parallel and serial stages. It is this mixed-batch setting where the cost of branch parallelism becomes visible.

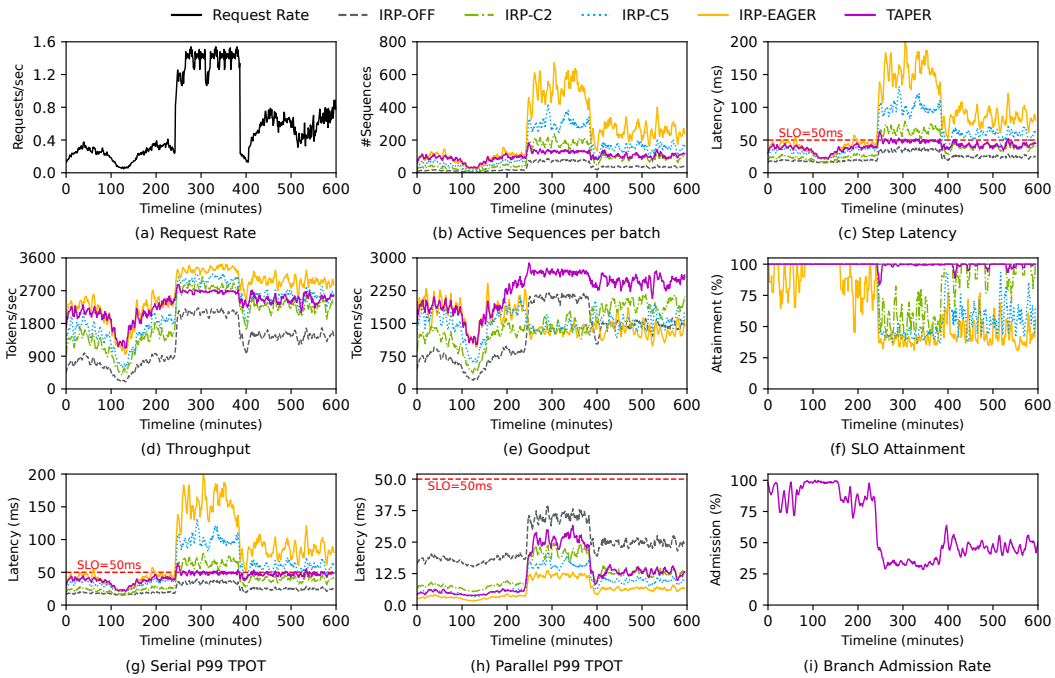


Figure 2: **The throughput trap and its resolution.** Four fixed step-width policies and TAPER on a mixed workload. IRP-EAGER raises throughput but collapses goodput and SLO attainment under load. The cost falls asymmetrically on requests in serial stages. TAPER dynamically adjusts its branch admission rate (panel (i)), retaining most of eager's throughput while protecting SLO attainment.

2.2 The throughput trap

The metrics above characterize parallelism in isolation. In a serving system, branches share a decode step with other requests, and the cost of admitting them is borne collectively.

As described in §1, continuous batching [13] executes one forward pass per decode step, advancing each active sequence by one token. When the runtime increases a request's *step width* $w_{r,t}$, the number of its branches included in step t, the step takes longer and every co-batched request pays that latency. The request in a parallel stage is compensated: it receives $w_{r,t}$ tokens of branch progress. A co-batched request in a serial stage receives only its single continuation token.

Figure 2 makes this concrete on an Azure-derived trace [23] (details in Appendix D). A mixed workload runs under four step-width policies: IRP-OFF ($w_{r,t} = 1$), two intermediate caps (IRP-C2, IRP-C5), and IRP-EAGER ($w_{r,t} = n_r$). At low load, larger step widths extract more raw throughput: the batch has headroom and wider steps do not slow anyone down. As load grows, wider policies inflate step latency (2(c)). Raw throughput continues to favor IRP-EAGER (2(d)), but goodput and SLO attainment collapse (2(e, f)): the system produces tokens while delivering fewer of them within their latency budget. We call this the *throughput trap*: the metric operators inspect first (raw throughput) favors the policy that fails the metric users experience (goodput).

The per-class view reveals where the SLO failures land. Serial P99 TPOT (2(g)) spikes sharply under IRP-EAGER during stress events, crossing the 50 ms target. Under IRP-OFF, it stays flat. Parallel P99 TPOT (2(h)) shows the opposite: it remains low regardless of load. The throughput trap is not a uniform degradation; it is an asymmetric transfer of latency from requests in serial stages to those in parallel stages.

The intermediate caps do not escape. IRP-C2 and IRP-C5 are safer than IRP-EAGER under load but still inflate step latency during stress events, and their goodput still collapses, just later. They are also worse at low load, leaving throughput unused. The safe step width depends on conditions that change

continuously over the trace: batch composition, context lengths, accumulated slack, and KV-cache pressure. No fixed cap is right across operating points.

2.3 Branch externality

The throughput trap and its asymmetric incidence reduce to a single quantity. When admitted branches inflate a decode step, every co-batched request pays the excess latency, but only the request in a parallel stage receives additional progress. This is a classic externality in the economic sense: one agent's resource consumption imposes uncompensated costs on others [24]. We formalize this as the *branch externality*. Let S_0 be the baseline step in which every active request advances by one token. Let $S(\mathbf{k})$ be a widened step that admits additional branches according to allocation $\mathbf{k}$, where k_r is the number of opportunistic branches granted to request r. The branch externality is

$$E_t(\mathbf{k}) \;=\; T\big(S(\mathbf{k})\big) \;-\; T(S_0),$$

where $T(\cdot)$ is the predicted step latency.

Every request in the batch pays $E_t(\mathbf{k})$; only requests in parallel stages with $k_r > 0$ receive additional progress in exchange. The step latency panel (2(c)) makes the externality visible: the gap between each policy's line and IRP-OFF's baseline is that policy's E_t. IRP-EAGER's gap grows unchecked during stress events, reaching over 150 ms above baseline; the fixed caps produce smaller but still unregulated gaps. The externality is monotone non-decreasing in each k_r: admitting more branches cannot shorten a decode step.

What the runtime needs is not a better static cap but a per-step controller that predicts $E_t(\cdot)$ against the current batch state and admits branches only when the predicted externality fits within the batch's available slack. The next section introduces TAPER, such a controller. Its behavior is already visible in Figure 2: TAPER's step latency tracks near IRP-EAGER at low load and contracts toward IRP-OFF during stress events (2(c)), while its branch admission rate (2(i)) shows the control signal adapting per step.

3 TAPER: Per-Step Admission Control

The characterization in §2 ended with a specific ask: a controller that predicts the branch externality $E_t(\mathbf{k})$ against the current batch state and admits branches only when the predicted cost fits within available slack. This section delivers that controller. We first formalize the object it acts on, the parallel phase, and establish a schedule-invariance property that licenses per-step width decisions without changing model semantics (§3.1). We then build the controller in three pieces: a cost model that predicts what a widened step will cost (§3.2), a slack-budgeted admission rule that decides how much widening the batch can absorb (§3.3), and a greedy planner that combines them (§3.4). We close by arguing that per-step regulation is not just desirable but structurally cheap, because branch-level scheduling decouples compute regulation from memory regulation (§3.5).

3.1 Parallel phases and schedule invariance

A *parallel phase* is a request-local segment in which the frontend has exposed a finite set of independent branches that must all complete before a reduce phase combines them. We represent a phase as $\phi = (P_\phi, H_\phi, \{h_i\}_{i=1}^{n_\phi})$, where P_ϕ is the shared prefix generated before the phase, H_ϕ is the phase-level header (e.g., an outline or task decomposition), h_i is the header for branch i, and n_ϕ is the branch fanout. This abstraction captures Multiverse-style Map/Reduce [9], Skeleton-of-Thought outline expansion [4], ASPD-style internal parallel decoding [8], and program-derived decompositions [5, 25, 6, 7].

The semantic contract is a *visibility rule* governing what each token can attend to. When generating token $y_{i,t}$ within branch i, the model attends to

$$P_\phi \;\oplus\; H_\phi \;\oplus\; h_i \;\oplus\; y_{i,<t}\,,$$

the shared prefix, the branch header, and the branch's own prior tokens, but nothing from sibling branches. During the reduce phase, token z_t attends to

$$P_\phi \;\oplus\; H_\phi \;\oplus\; \bigoplus\nolimits_i (h_i \oplus y_i) \;\oplus\; z_{<t}\,,$$

the prefix followed by all completed branches in canonical order. Because every branch conditions on the same $P_\phi \oplus H_\phi$, a backend with paged or radix-tree KV caches [18, 19] can serve all branches from a single set of prefix blocks. Each branch materializes only its own branch-local tokens. The marginal KV cost of admitting one additional branch is therefore branch-local, not a full prefix.

The runtime's scheduling freedom lies in how it assigns branch tokens to decode steps: which branches participate in each step, how many advance concurrently, and in what order. A schedule is *valid* if it preserves token order within each branch, enforces the visibility rule, and defers reduction until all branches complete.

Lemma 3.1 (Schedule invariance)**.** *The output of a parallel phase is independent of the order and timing in which its branches are scheduled. Formally, any two valid schedules induce the same conditional distribution over all generated tokens. Under deterministic decoding the outputs are identical; under stochastic decoding they are identical given fixed per-branch random seeds.*

The argument is direct: the attention context of $y_{i,t}$ contains no token from any sibling branch, so whether siblings have been scheduled before, concurrently with, or after $y_{i,t}$ cannot affect its distribution. Once all branches complete, the reduce phase observes the same outputs in canonical order regardless of execution order. A formal proof appears in Appendix A.

TAPER is therefore free to defer, interleave, or reorder branches without affecting model output. The *step width* $w_{r,t}$, the number of branches from request r that participate in decode step t, is a free parameter of the schedule, not a constraint of the model.

3.2 Predicting the cost of a widened step

To decide whether admitting additional branches is safe, TAPER needs to predict what the widened step will cost. Decode-step latency decomposes into an FFN component, approximately linear in the number of new tokens, and an attention component, approximately linear in aggregate context length [15]. A candidate step is therefore well summarized by its *step composition* S: the number of active sequences and their aggregate context length. TAPER uses a calibrated latency predictor $T(S)$ over these composition, requiring only that T is monotone non-decreasing in the number of admitted branches. Monotonicity is the key structural property: if admitting one more branch from request r exceeds the budget, admitting two more will too, enabling the greedy planner's pruning rule (§3.4).

3.3 Slack-budgeted admission

TAPER first protects *single-token progress*. The *protected composition* S_0 advances every active request by exactly one token: a single continuation for requests in serial stages, one branch for requests in parallel stages. Any remaining exposed-but-incomplete branches are *ready* and eligible for opportunistic admission. Admitting ready branches widens the step ($w_{r,t} > 1$); TAPER treats this additional width as *opportunistic*, admitted only when predicted externality fits within slack budget.

To decide which opportunistic branches are safe, TAPER converts per-request slack into a time budget. Each active request r has a deadline $d_r(t)$ for its next token, derived from its SLO. If the baseline step takes predicted time $T_0 = T(S_0)$, the residual time available for opportunistic width is

$$B_t \;=\; \max\Big(0,\; \min_r \big(d_r(t) - t\big) - T_0\Big).$$

TAPER spends a fraction $\rho \in (0, 1]$ of this residual. A candidate widened step S is admitted only if

$$T(S) \;\le\; T_0 + \rho\, B_t.$$

This bounds the branch externality: the predicted cost of the widened step must fit within the fraction of slack the operator is willing to spend.

The minimum over all requests is deliberate: opportunistic width must be safe for the most urgent request in the batch, not the request whose phase is being widened. This prevents a slack-rich request in a parallel stage from underwriting a widened step that pushes a tight request in a serial stage past its deadline, exactly the failure mode the throughput trap produces.

When the slack budget cannot accommodate all branches, TAPER needs a tiebreaker. We separate safety from value: $T(\cdot)$ decides what is *feasible*; a monotone utility curve $u_r(k)$ decides what is

Algorithm 1 TAPER Per-Step Planner

```
def TAPERStep(requests, T, utility, ρ):
    baseline = BuildBaseline(requests)                    #protected composition
    min_slack = min(r.deadline - now() for r in requests) #most urgent request
    budget = T(baseline) + ρ * max(0, min_slack - T(baseline))
    granted = {r: 0 for r in requests}
    candidates = {r for r in requests if r.ready_branches > 0}
    step = baseline
    while candidates:
        best, infeasible = None, set()
        for r in candidates:
            widened = AddBranch(step, r)
            if T(widened) > budget:                       #monotone: prune request r
                infeasible.add(r)
                continue
            du = utility[r](granted[r]+1) - utility[r](granted[r])
            dt = T(widened) - T(step)
            score = du / (EPS + max(0, dt))               #utility per cost
            if best is None or score > best.score:
                best = Candidate(r, widened, score)
        candidates -= infeasible
        if best is None or best.score <= 0:               #no feasible improvement
            break
        step = best.composition                           #commit best candidate
        granted[best.request] += 1
        if granted[best.request] >= best.request.ready_branches:
            candidates.discard(best.request)              #request r fully admitted
    return step
```

valuable, where k is the number of opportunistic branches granted to request r. Throughput-oriented operators use linear utility; fairness-oriented operators use concave utility so that the first opportunistic branch matters more than later ones; priority operators weight by tenant class. Each is a curve choice, not a scheduler change. TAPER predicts cost; the operator supplies value through the utility interface.

3.4 Per-step planner

At each decode step, TAPER starts from the baseline composition S_0 and greedily widens. It considers admitting one more branch from each request with ready branches, commits the candidate with the highest marginal utility per marginal latency cost, and repeats until no feasible increment remains. Algorithm 1 gives the pseudocode.

The planner runs in $O(|\mathcal{R}| \cdot K_{\max} \cdot c_T)$ per step, where c_T is the cost of one $T(\cdot)$ evaluation, typically a table lookup or short regression well under a millisecond. The greedy rule is justified by monotonicity: once a branch from request r is infeasible, all further branches from r are too. The globally optimal allocation is NP-hard (Appendix B); the greedy approximation suffices because the planner replans every step rather than committing to a single allocation.

3.5 Why per-step regulation works

After each decode step, TAPER updates $T(\cdot)$ from the realized latency, recomputes per-request slack, and reconsiders any deferred branches. Schedule invariance (Lemma 1) guarantees that deferral does not change model output.

This converts the width decision from a one-shot commitment into a closed-loop controller. As established in §1, the structural reason this is practical is that branch regulation decouples compute from memory. Deferring a branch adjusts the step's compute cost without evicting any KV state: the prefix remains resident for the sibling branches that are admitted, and the branch-local KV is small enough to leave in place. Admitting the branch on a later step requires no restoration. The cost of revising the width decision at any step is bounded by one planner invocation, not by a memory round-trip.

Two consequences follow. First, TAPER degrades gracefully: as load rises and B_t collapses, the same code path that widens steps under abundant slack narrows them under pressure. There is no

policy switch, no preemption mechanism, and no fallback logic. Second, TAPER is robust to predictor error: a coarse $T(\cdot)$ costs throughput (the planner admits fewer branches than it safely could) but does not cause SLO violations, because the safety check runs against the realized slack at every step. The cost of a worse predictor is paid in width, not in latency. TAPER therefore treats step width as a renewable, per-step decision. When slack is available, TAPER spends it on width; when slack is scarce, TAPER contracts to the protected composition. The controller never needs to know in advance how many branches a phase will expose, how long each will run, or which will straggle.

4 Evaluation

We evaluate TAPER on mixed workloads that combine requests in serial and parallel stages. §2 established that eager branch admission creates a throughput trap: raw throughput rises while goodput and SLO attainment collapse. The central question is whether TAPER can navigate this trap, harvesting the throughput benefit of intra-request parallelism without the SLO degradation. We examine this across three load regimes (§4.2) and validate design choices through targeted ablations (§4.3). Additional experiments, including workload composition sensitivity, SLO target sensitivity, output quality verification, and evaluation on Qwen2.5-72B [26], appear in Appendix E.

4.1 Experimental Setup

We evaluate on Qwen3-32B [27] served with SGLang [19] on 8×A100-80GB GPUs (TP=8) with radix-tree KV allocation. Workloads interleave non-decomposable requests from ShareGPT with decomposable requests from Multiverse [9] (PDR=50%, ABF=4.1, PTS=58%). Request arrivals follow an Azure-derived trace [23] spanning 600 minutes across low, moderate, and high load regimes. All requests share a 50 ms TPOT target. We report raw throughput, goodput, and SLO attainment and compare IRP-OFF ($w_{r,t} = 1$), IRP-C2, IRP-C5, IRP-EAGER ($w_{r,t} = n_r$), and TAPER ($\rho = 0.8$, linear utility) on the same serving system. Full details in Appendix D.

4.2 Navigating the throughput trap

Figure 2 compares all five policies on the same production-derived trace. TAPER's central claim is that it tracks near-IRP-EAGER throughput when conditions permit and falls back to near-IRP-OFF protection when they do not, using the slack budget as the sole switching signal. We verify this across three load regimes visible in the trace.

Low load. During the first 240 minutes, request rate averages 0.23 req/s. At this operating point, IRP-OFF's step latency averages 18,ms, leaving over 30,ms of slack against the 50,ms target. With $\rho = 0.8$, TAPER's latency budget easily accommodates most available branches: its step latency averages 36 ms, close to IRP-EAGER's 38 ms (2(c)), confirming that TAPER admits aggressively when the batch has headroom. TAPER's branch admission rate stays near 100% throughout this period (2(i)). SLO attainment (2(f)) is 100% for TAPER, IRP-OFF, and both caps. Notably, even at low load IRP-EAGER's attainment is only 89%, because transient bursts already push its step latency above 50 ms. This foreshadows the collapse that follows.

High load. Between minutes 250 and 400, request rate surges past 1.0 req/s and sustains above 1.2 req/s, peaking at 1.54 req/s. IRP-EAGER's step latency spikes to a mean of 148 ms (max 203 ms), nearly $4\times$ the SLO target, as the batch fills with eagerly admitted branches. SLO attainment collapses to a mean of 41%. Serial P99 TPOT (2(g)) under IRP-EAGER averages 148 ms with a P95 of 186 ms, while parallel P99 TPOT (2(h)) remains low at 11 ms: the asymmetric incidence of §2.2 in full effect. The fixed caps fare poorly: IRP-C5 averages 97 ms step latency (attainment 45%) and IRP-C2 averages 62 ms (attainment 59%), both well above the SLO target. TAPER contracts sharply. As slack collapses under the sustained high rate, the planner defers most opportunistic branches. Branch admission rate drops to approximately 35% (2(i)). Step latency averages 48 ms with a maximum of 63 ms, staying near the 50 ms target throughout. SLO attainment holds at 99%.

Moderate load. After the stress event subsides around minute 400, request rate settles into a sustained regime averaging 0.60 req/s with peaks near 0.9 req/s. Slack rebuilds and TAPER's branch admission rate recovers to approximately 47% without operator intervention (2(i)). Step latency averages 42 ms (max 52 ms), staying within the SLO target while admitting substantially more branches than during the stress period. IRP-EAGER continues to violate: its step latency averages 85 ms (max 122 ms)

Variant	Goodput / IRP-Off	Attainment
TAPER (full, $\rho = 0.8$)	$1.77\times$	99%
w/o slack budget	$0.92\times$	48%
w/o per-step replanning	$1.18\times$	82%
w/ constant predictor	$1.12\times$	96%
$\rho = 0.5$	$1.25\times$	99.5%
$\rho = 1.0$	$1.45\times$	91%

Table 1: **Ablation study.** Goodput (normalized by IRP-OFF) and SLO attainment for TAPER variants on the full 10-hour trace. Removing the slack budget collapses performance to near-IRP-EAGER levels. ρ provides a smooth tradeoff between goodput and attainment.

and attainment stays at 45%. This regime exercises the most revealing operating point: load is high enough that IRP-EAGER and IRP-C5 consistently violate SLOs, but low enough that TAPER finds slack between bursts to admit branches opportunistically. IRP-C2 nearly keeps up in this regime (attainment 95%) but at the cost of leaving significant throughput unused at low load and during the stress recovery. TAPER adapts to both without reconfiguration.

Summary. Over the full 10-hour trace, TAPER improves goodput over IRP-OFF by $1.77\times$ and by $1.48\times$ over IRP-EAGER, while maintaining over 95% SLO attainment where IRP-EAGER drops below 50%. The fixed caps cannot match TAPER in either direction: IRP-C5 exceeds TAPER's throughput at low load but collapses under stress (attainment 45%), while IRP-C2 protects better (attainment 59% high-load, 95% moderate) but generates less useful work overall.

4.3 Ablation

We isolate each component of TAPER by removing it in turn (Table 1). **Without the slack budget**, TAPER admits all branches that fit in memory, collapsing to near-IRP-EAGER behavior: goodput drops below IRP-OFF ($0.92\times$) as SLO violations destroy more goodput than the extra branches create, with attainment at 48%. **Without per-step replanning**, width is committed at phase start and held until reduce; goodput is $1.18\times$ and attainment 82%, with failures concentrated at load transitions where the controller cannot contract mid-phase. **With a constant latency predictor**, the planner cannot distinguish cheap steps from expensive ones, so it under-admits to stay safe: goodput drops to $1.12\times$ while attainment remains high at 96%, showing that the predictor contributes throughput, not safety. **Slack fraction** ρ controls the throughput-safety tradeoff: $\rho = 0.5$ is conservative ($1.25\times$ goodput, 99.5% attainment), $\rho = 1.0$ is aggressive ($1.45\times$, 91%), and the default $\rho = 0.8$ balances both ($1.77\times$, 99%). The greedy planner adds 0.3 ms median (0.8 ms P99) per decode step, less than 1.5% of typical step latency.

5 Related Work

Existing work on parallel LLM generation divides into two objectives [8]. *Quality-oriented* methods such as best-of-N sampling [28], beam search [29], self-consistency [30], majority voting [31], MCTS [32], and Tree-of-Thoughts [33] generate multiple candidates to improve answer quality. *Performance-oriented* methods such as SoT [4], APAR [6], PASTA [5], ASPD [8], and Multiverse [9] decompose a single response into independent branches to reduce latency (§2.1). Recent work on parallel reasoning [34, 25, 35] blurs this boundary, using concurrent execution for both speed and quality. Despite their different objectives, both lines share a common scope: they operate on a single request in isolation. Neither considers what happens when the resulting branches share a decode step with other requests, which is the problem TAPER addresses. Rather than discovering branch structure, TAPER takes the structure the frontend provides and decides how much of it to admit into each shared decode step, bounding the externality on co-batched requests. The closest structural sibling in the serving literature is FairBatching [36], which converts decode slack into a time budget for chunked prefill. TAPER's slack-budget mechanism follows the same principle but operates within a request's parallel phase rather than across request phases. Extended comparisons with LLM serving schedulers and preemption mechanisms appear in Appendix F.

6 Conclusion & Limitations

TAPER reframes intra-request parallel generation as an admission control problem. By exploiting the structural decoupling of compute and memory at the branch level, it regulates admitted width per step, improving goodput over IRP-OFF by $1.77\times$ and over IRP-EAGER by $1.48\times$, with over 95% SLO attainment. The current design assumes branch independence (§3.1), uses a linear latency predictor that may lose accuracy at extreme batch sizes, and operates within a single serving node. Even so, the result suggests that IRP need not be an all-or-nothing choice between eager admission and conservative caps: a per-step controller can capture most of the throughput benefit while protecting the requests that would otherwise be forced to pay for it.

References

[1] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

[2] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948*, 2025.

[3] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

[4] Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Prompting llms for efficient parallel generation, 2024. URL https://arxiv.org/abs/2307.15337.

[5] Tian Jin, Ellie Y. Cheng, Zack Ankner, Nikunj Saunshi, Blake M. Elias, Amir Yazdanbakhsh, Jonathan Ragan-Kelley, Suvinay Subramanian, and Michael Carbin. Learning to keep a promise: Scaling language model decoding parallelism with learned asynchronous decoding, 2025. URL https://arxiv.org/abs/2502.11517.

[6] Mingdao Liu, Aohan Zeng, Bowen Wang, Peng Zhang, Jie Tang, and Yuxiao Dong. Apar: Llms can do auto-parallel auto-regressive decoding, 2024. URL https://arxiv.org/abs/2401.06761.

[7] Jiayi Pan, Xiuyu Li, Long Lian, Charlie Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. Learning adaptive parallel reasoning with language models, 2025. URL https://arxiv.org/abs/2504.15466.

[8] Keyu Chen, Zhifeng Shen, Daohai Yu, Haoqian Wu, Wei Wen, Jianfeng He, Ruizhi Qiao, and Xing Sun. Aspd: Unlocking adaptive serial-parallel decoding by exploring intrinsic parallelism in llms, 2025. URL https://arxiv.org/abs/2508.08895.

[9] Xinyu Yang, Yuwei An, Hongyi Liu, Tianqi Chen, and Beidi Chen. Multiverse: Your language models secretly decide how to parallelize and merge generation, 2025. URL https://arxiv.org/abs/2506.09991.

[10] Long Lian, Sida Wang, Felix Juefei-Xu, Tsu-Jui Fu, Xiuyu Li, Adam Yala, Trevor Darrell, Alane Suhr, Yuandong Tian, and Xi Victoria Lin. Threadweaver: Adaptive threading for efficient parallel reasoning in language models, 2025. URL https://arxiv.org/abs/2512.07843.

[11] Steven Kolawole, Keshav Santhanam, Virginia Smith, and Pratiksha Thaker. Parallelprompt: Extracting parallelism from large language model queries, 2025. URL https://arxiv.org/abs/2506.18728.

[12] Lingzhe Zhang, Liancheng Fang, Chiming Duan, Minghua He, Leyi Pan, Pei Xiao, Shiyu Huang, Yunpeng Zhai, Xuming Hu, Philip S. Yu, and Aiwei Liu. A Survey on Parallel Text Generation: From Parallel Decoding to Diffusion Language Models, 2026. URL https://arxiv.org/abs/2508.08712.

[13] 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)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

[14] Yuxing Xiang, Xue Li, Kun Qian, Yan Zhang, Wenyuan Yu, Ennan Zhai, Xin Jin, and Jingren Zhou. ServeGen: Workload characterization and generation of large language model serving in production. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 1845–1859, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. URL https://www.usenix.org/conference/nsdi26/presentation/xiang-servegen.

[15] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. NanoFlow: Towards optimal large language model serving throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 749–765, Boston, MA, July 2025. USENIX Association. ISBN 978-1-939133-47-2. URL https://www.usenix.org/conference/osdi25/presentation/zhu-kan.

[16] 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)*, pages 965–988, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/sheng.

[17] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/sun-biao.

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

[19] 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 *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2025. Curran Associates Inc. ISBN 9798331314385.

[20] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[21] Neural Bridge. Retrieval-Augmented Generation (RAG) Dataset 12000. https://huggingface.co/datasets/neural-bridge/rag-dataset-12000, 2023.

[22] Open-R1. OpenR1-Math-220k. https://huggingface.co/datasets/open-r1/OpenR1-Math-220k, 2025.

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

[24] Arthur Pigou. *The economics of welfare*. Routledge, 2017.

[25] Emil Biju, Shayan Talaei, Zhemin Huang, Mohammadreza Pourreza, Azalia Mirhoseini, and Amin Saberi. Sprint: Enabling interleaved planning and parallelized execution in reasoning models, 2025. URL https://arxiv.org/abs/2506.05745.

[26] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115*, 2024.

[27] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

[28] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

[29] Brian J Chan, MaoXun Huang, Jui-Hung Cheng, Chao-Ting Chen, and Hen-Hsen Huang. Efficient beam search for large language models using trie-based decoding, 2025. URL https://arxiv.org/abs/2502.00085.

[30] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023. URL https://arxiv.org/abs/2203.11171.

[31] Xinyun Chen, Renat Aksitov, Uri Alon, Jie Ren, Kefan Xiao, Pengcheng Yin, Sushant Prakash, Charles Sutton, Xuezhi Wang, and Denny Zhou. Universal self-consistency for large language model generation, 2023. URL https://arxiv.org/abs/2311.17311.

[32] Di Zhang, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b, 2024. URL https://arxiv.org/abs/2406.07394.

[33] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: deliberate problem solving with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[34] Gleb Rodionov, Roman Garipov, Alina Shutova, George Yakushev, Erik Schultheis, Vage Egiazarian, Anton Sinitsin, Denis Kuznedelev, and Dan Alistarh. Hogwild! inference: Parallel llm generation via concurrent attention, 2025. URL https://arxiv.org/abs/2504.06261.

[35] Tong Zheng, Hongming Zhang, Wenhao Yu, Xiaoyang Wang, Runpeng Dai, Rui Liu, Huiwen Bao, Chengsong Huang, Heng Huang, and Dong Yu. Parallel-r1: Towards parallel thinking via reinforcement learning, 2025. URL https://arxiv.org/abs/2509.07980.

[36] Hongtao Lyu, Boyue Liu, Mingyu Wu, and Haibo Chen. Fairbatching: Fairness-aware batch formation for llm inference, 2025. URL https://arxiv.org/abs/2510.14392.

[37] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[38] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. MEDUSA: Simple LLM inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[39] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test. In *Annual Conference on Neural Information Processing Systems*, 2025.

[40] Xiaoxuan Liu, Jongseok Park, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Chen Zhang, Kuntai Du, Xiangxi Mo, Kaichao You, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Turbospec: Closed-loop speculation control system for optimizing llm serving goodput, 2025. URL https://arxiv.org/abs/2406.14066.

[41] 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*, SoCC '25, page 361–374. ACM, November 2025. doi: 10.1145/3772052.3772239. URL http://dx.doi.org/10.1145/3772052.3772239.

[42] Zikun Li, Zhuofu Chen, Remi Delacourt, Gabriele Oliaro, Zeyu Wang, Qinghan Chen, Shuhuai Lin, April Yang, Zhihao Zhang, Zhuoming Chen, Sean Lai, Xinhao Cheng, Xupeng Miao, and Zhihao Jia. Adaserve: Accelerating multi-slo llm serving with slo-customized speculative decoding, 2025. URL https://arxiv.org/abs/2501.12162.

[43] Rui Li, Zhaoning Zhang, Libo Zhang, Huaimin Wang, Xiang Fu, and Zhiquan Lai. Nightjar: Dynamic adaptive speculative decoding for large language models serving, 2026. URL https://arxiv.org/abs/2512.22420.

[44] 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, 2024. URL https://arxiv.org/abs/2404.16283.

[45] Yue Zhang, Yuansheng Chen, Xuan Mo, Alex Xi, Jialun Li, and WeiGang Wu. A predictive and synergistic two-layer scheduling framework for llm serving, 2025. URL https://arxiv.org/abs/2509.23384.

A Proof of Schedule Invariance

Lemma A.1 (Schedule invariance)**.** *The output of a parallel phase is independent of the order and timing in which its branches are scheduled. Formally, for a parallel phase $\phi = (P_\phi, H_\phi, \{h_i\}_{i=1}^{n_\phi})$ with the visibility rule of §3.1, any two valid schedules induce the same conditional distribution over all generated tokens. Under deterministic decoding they produce the same output up to numerical nondeterminism; under stochastic decoding they are equivalent given fixed per-branch random seeds.*

Proof. We show that the conditional distribution of every token is invariant to the schedule, by case analysis on whether the token is generated within a branch or during the reduce phase.

Tokens within branches. Consider token $y_{i,t}$ in branch i at position t. By the visibility rule, the model attends to

$$P_\phi \ \oplus \ H_\phi \ \oplus \ h_i \ \oplus \ y_{i,<t}\,.$$

This attention context contains: (i) the prefix P_ϕ, generated before the parallel phase opens and therefore identical under any schedule; (ii) the phase header H_ϕ, also generated before the phase opens; (iii) the branch header h_i, determined at phase opening; and (iv) the prior tokens of branch i itself, $y_{i,<t}$, generated in token order within branch i by validity.

Crucially, no token from any sibling branch $j \neq i$ appears in this context. Therefore, the conditional distribution

$$P(y_{i,t} \mid P_\phi, H_\phi, h_i, y_{i,<t})$$

depends only on the model parameters and the context above, none of which is affected by whether sibling branches have been scheduled before, concurrently with, or after token $y_{i,t}$. The same argument applies inductively: since $y_{i,1}$ has the same distribution under any valid schedule, and $y_{i,t}$ depends only on $y_{i,<t}$ (which by induction has the same distribution), the entire branch sequence $y_i = (y_{i,1}, \ldots, y_{i,L_i})$ has the same joint distribution under any valid schedule.

Since this holds for each branch i independently, the joint distribution over all branch outputs $(y_1, \ldots, y_{n_\phi})$ is the same under any valid schedule.

Tokens during the reduce phase. Consider token z_t at position t during the reduce phase. By the visibility rule, the model attends to

$$P_\phi \ \oplus \ H_\phi \ \oplus \ \bigoplus_{i=1}^{n_\phi} (h_i \oplus y_i) \ \oplus \ z_{<t} \, .$$

By validity, the reduce phase starts only after all branches complete. The branch outputs are presented in canonical order (determined by the phase structure), not in completion order. Therefore this context is identical under any valid schedule: it depends on the completed branch outputs (same by the argument above) and their ordering (canonical, not schedule-dependent). The conditional distribution $P(z_t \mid \cdot)$ is therefore the same under any valid schedule.

Deterministic and stochastic decoding. Under deterministic decoding (greedy or beam search), the output is a deterministic function of the conditional distributions, hence identical up to numerical nondeterminism (floating-point ordering in softmax, etc.). Under stochastic decoding, the output depends additionally on random draws. If each branch i uses a fixed random seed (e.g., derived from the branch index), the same sequence of draws produces the same tokens regardless of schedule, completing the proof. □

Remark. The proof relies on two structural properties: (i) branch isolation (no sibling tokens in the attention context) and (ii) canonical reduce ordering. Methods that allow inter-branch attention (e.g., Hogwild! Inference [34]) or order reduce inputs by completion time do not satisfy these properties and are not covered by this lemma.

B NP-Hardness of Optimal Width Allocation

The TAPER planner uses a greedy heuristic to allocate opportunistic branches. This appendix shows that the underlying optimization problem is NP-hard in general, justifying the greedy approach.

Definition B.1 (Width allocation problem)**.** *Given a set of requests $\mathcal{R}$ with ready branches, a latency predictor $T(\cdot)$, utility curves $\{u_r\}_{r\in\mathcal{R}}$, and a latency budget $T_{\max}$, find an allocation $\mathbf{k} = (k_r)_{r\in\mathcal{R}}$ that maximizes total utility $\sum_r u_r(k_r)$ subject to $T(S(\mathbf{k})) \le T_{\max}$.*

Theorem B.1. *The width allocation problem is NP-hard.*

Proof sketch. We reduce from the 0–1 knapsack problem. Given a knapsack instance with items $\{(w_j, v_j)\}_{j=1}^m$ and capacity W, construct a width allocation instance as follows. Create m requests, each with exactly one ready branch. Set the utility of admitting the branch to $u_r(1) = v_j$ and the marginal latency cost to $T(S(\mathbf{e}_j)) - T(S_0) = w_j$, where $\mathbf{e}_j$ is the allocation that admits only request j's branch. Set $T_{\max} = T(S_0) + W$. Assume latency is additive in the marginal costs (which holds when branch contributions to FFN and attention are independent). Then the width allocation problem reduces to selecting a subset of branches whose total marginal latency cost does not exceed W and whose total utility is maximized, exactly the 0–1 knapsack problem. □

Remark. The greedy planner of §3.4 does not need to solve this problem optimally. Because the planner replans at every decode step, a suboptimal allocation at step t is corrected at step $t+1$. The greedy approximation is within a factor of $1/2$ of optimal for monotone submodular utilities under a single knapsack constraint, and in practice the number of requests with ready branches per step is small enough that the greedy solution is near-optimal.

C Latency Predictor Details

C.1 Predictor formulation

TAPER requires a predictor $T(S)$ that maps a step composition S to a predicted wall-clock latency in milliseconds. Following the observation that decode-step latency decomposes into a fixed overhead, an FFN-dominated term linear in new tokens, and an attention-dominated term linear in aggregate context length, we use a linear model:

$$T(S) \ = \ a \ + \ b \cdot n_{\text{tokens}} \ + \ c \cdot L_{\text{context}} \, ,$$

where n_{tokens} is the number of sequences in the step (each generating one token) and L_{context} is their aggregate context length.

Table 2: Latency predictor accuracy across operating regimes.

Regime	Batch size range	MAPE (%)
Low load	1–64	1.7
Medium load	64–256	1.9
High load	256–512	1.8
Overall	1–512	1.8

Fitting. The model is fitted offline by profiling the target hardware with synthetic batches spanning a grid of (batch size, context length) combinations. We run 500 profiling steps across a 20×25 grid of batch sizes (1–512) and context lengths (128–8192), record wall-clock latency for each, and fit (a, b, c) via ordinary least squares. The fit is refreshed every 10 minutes using a rolling window of the most recent 200 observed step latencies to account for thermal drift and co-tenant interference.

Accuracy. On our evaluation hardware (§D), the linear model achieves a mean absolute percentage error (MAPE) of 1.8% across the profiling grid.

Monotonicity. The predictor satisfies the monotonicity assumption of §3.2 by construction: admitting one additional branch increases both n_{tokens} and L_{context}, and $b, c > 0$.

C.2 Memory accounting

Although TAPER's primary admission control is latency-based (§3.3), the runtime also tracks KV-cache occupancy to avoid out-of-memory conditions. The key accounting detail is that branches share their request's prefix KV blocks. The marginal KV cost of admitting one additional branch is therefore only its branch-local tokens, not a full sequence:

$$\Delta M(j) \;=\; \text{blocks}(L_j^{\text{branch-local}})\,,$$

where $L_j^{\text{branch-local}}$ is the number of tokens branch j has generated so far. For a branch that has not yet produced any tokens, $\Delta M(j) = 0$. A scheduler that priced each branch as a full sequence would refuse safe widenings throughout.

D Experimental Setup Details

This appendix provides full details for the experimental setup summarized in §4.1.

Hardware. All experiments run on a single node with 8×A100-80GB GPUs connected via NVLink. The model is served in tensor-parallel mode (TP=8), placing one shard per GPU. Peak GPU memory utilization during the high-load regime is approximately 78%, with KV cache consuming roughly 62% of total GPU memory. The remaining headroom accommodates branch-local KV from deferred branches without triggering request-level preemption.

Model. We use Qwen3-32B [27], a 32-billion parameter autoregressive transformer with 64 attention heads, grouped-query attention (8 KV heads), and a context length of 32,768 tokens. The model is served using BFloat16 precision.

Serving engine. We use SGLang [19] as the serving backend, configured with radix-tree KV cache allocation for prefix sharing across branches within a parallel phase. TAPER's per-step planner is integrated as a scheduling hook that runs between the batch-formation stage and the forward pass. The planner decides which branches to include in each decode step; all other engine behavior (prefill scheduling, memory management, request admission) is unchanged from the default SGLang configuration. Multiverse [9] provides the frontend for branch discovery and the Map/Process/Reduce execution model.

Workload construction. The evaluation workload interleaves two request streams in equal proportion (PDR $\approx 50\%$):

- **Non-decomposable requests** are drawn from ShareGPT Vicuna [20]. These requests produce purely serial output and represent the sequential traffic that is vulnerable to the branch externality.
- **Decomposable requests** are sampled uniformly from ShareGPT Vicuna [20], RAG-12K [21], and MATH-220K [22]. These prompts are processed through Multiverse [9], which decomposes each response into a sequence of serial and parallel stages. The resulting branch structure has a mean ABF of 4.1 and mean PTS of 58% among decomposable requests.

Branch fanout distribution. Table 3 summarizes the branch fanout distribution across decomposable requests in the evaluation workload.

Table 3: Branch fanout distribution among decomposable requests.

Statistic	P10	P25	P50	P75	P90
Fanout (n_r)	2	3	4	5	7

Request arrival trace. Request arrivals are derived from the Azure LLM Inference Trace [23], replayed at different scaling factors to produce three load regimes within a single 600-minute experiment: a low-load period (0–240 min, mean 0.23 req/s), a sustained high-load period (250–400 min, mean 1.27 req/s, peak 1.54 req/s), and a moderate recovery period (400–600 min, mean 0.60 req/s). The trace is replayed identically for all five policies to ensure a controlled comparison.

SLO definition. All requests share a TPOT target of 50 ms. For requests in serial stages, TPOT is measured as the wall-clock time between consecutive token deliveries. For requests in parallel stages, TPOT is measured as effective TPOT: the wall-clock duration of the parallel phase divided by the total number of tokens generated across all branches during that phase. A request meets its SLO if its maximum per-token latency (across all stages) does not exceed the target.

Metrics. We report raw throughput (total tokens generated per second regardless of SLO compliance), goodput (tokens per second counting only tokens from SLO-meeting requests), and SLO attainment (the fraction of completed requests whose maximum TPOT does not exceed the 50,ms target).

TAPER configuration. Unless otherwise noted, TAPER runs with slack fraction $\rho = 0.8$ and linear utility $u_r(k) = k$. The latency predictor $T(S)$ is a linear model fitted offline as described in Appendix C. The planner runs once per decode step; its overhead is characterized in §4.3.

Baselines. All five policies share the same SGLang backend, model weights, KV cache configuration, and request arrival trace. They differ only in how admitted width is determined:

- IRP-OFF: $w_{r,t} = 1$ for all requests. No branches beyond the baseline are admitted.
- IRP-C2: $w_{r,t} = \min(n_r, 2)$. Step width capped at 2 branches per parallel-stage request.
- IRP-C5: $w_{r,t} = \min(n_r, 5)$. Step width capped at 5.
- IRP-EAGER: $w_{r,t} = n_r$. All available branches are admitted every step.
- TAPER: $w_{r,t}$ determined by the per-step planner (§3.4), bounded by the slack budget.

E Additional Experiments

This appendix presents sensitivity analyses and additional results that complement the main evaluation in §4.

E.1 Workload composition sensitivity

The main evaluation uses a PDR of 50%. Table 4 varies this proportion across three load regimes to assess how TAPER and eager admission respond to workloads with more or fewer decomposable requests. All experiments use the same Azure-derived trace, SLO target (50 ms), and TAPER configuration ($\rho = 0.8$, linear utility). Goodput is normalized by each cell's corresponding IRP-OFF baseline.

Table 4: Sensitivity to workload composition across load regimes. Goodput is normalized by IRP-OFF for each PDR and regime.

		IRP-EAGER		TAPER	
PDR	Regime	Goodput	Att.	Goodput	Att.
20%	Low	1.08×	97%	1.08×	100%
	Moderate	1.06×	82%	1.30×	99%
	High	0.92×	48%	1.38×	98%
	Aggregate	1.10×	72%	1.28×	99%
50%	Low	1.28×	89%	1.28×	100%
	Moderate	1.08×	45%	1.75×	99%
	High	0.88×	41%	1.68×	99%
	Aggregate	1.20×	48%	1.77×	99%
80%	Low	1.48×	93%	1.48×	100%
	Moderate	1.25×	62%	2.05×	98%
	High	1.12×	55%	2.10×	97%
	Aggregate	1.35×	68%	2.15×	98%

The interaction between PDR and attainment is not monotone, because PDR determines both the amount of interference and the size of the vulnerable population.

At PDR=20%, only 20% of requests expose branches, so per-step interference is modest. At low load, the externality is small enough that both eager and TAPER perform similarly ($1.08\times$), and eager attainment is 97%. At moderate load, interference grows and eager attainment drops to 82% as the large sequential majority begins to feel the cost. During the high-load regime, the same 20% parallel requests generate sufficient interference to push step latency well above the target, collapsing eager attainment to 48% as damage falls on the 80% sequential population. TAPER contracts to protect this majority ($1.38\times$ goodput, 98% attainment at high load).

At PDR=50%, interference is larger and begins earlier: even at low load, eager attainment is only 89% due to transient bursts. At moderate load, attainment drops to 45% and continues to 41% at high load. This is the worst-case combination: enough parallel requests to cause substantial interference, and enough sequential requests to suffer from it as half the population is vulnerable.

At PDR=80%, per-step interference is at its highest, but the vulnerable population is small: only 20% of requests are in serial stages. At low load, eager achieves $1.48\times$ goodput with 93% attainment. Even as load increases, eager attainment stays above PDR=50% levels (62% moderate, 55% high) despite greater interference, because fewer requests can be harmed. Parallel requests, comprising 80% of the workload, almost always meet SLO because their effective TPOT (step latency divided by branch fanout) stays well below the target even when step latency is inflated.

TAPER maintains $\geq$97% attainment in every cell. At low load across all PDRs, TAPER matches eager exactly: the slack budget is permissive and no branches are deferred. The divergence appears at moderate and high load, where TAPER's goodput advantage grows with PDR ($1.30\times$ to $2.10\times$) because higher PDR provides more branches for opportunistic admission while the slack budget prevents the cascade that destroys eager's goodput.

E.2 SLO target sensitivity

Table 5 varies the TPOT target from 30 ms (tight) to 100 ms (generous) to assess how TAPER adapts to different latency requirements. All experiments use PDR=50%.

Table 5: Sensitivity to SLO target. TAPER adapts automatically via the slack budget; no reconfiguration is needed.

	IRP-EAGER		TAPER	
SLO Target	Goodput	Att.	Goodput	Att.
30 ms	$0.87\times$	28%	$1.35\times$	97%
50 ms	$1.20\times$	48%	$1.77\times$	99%
100 ms	$1.52\times$	78%	$2.18\times$	99.5%

At a tight 30 ms target, IRP-EAGER's goodput drops below IRP-OFF ($0.87\times$) because its step latency exceeds 30 ms even under moderate load. TAPER remains effective ($1.35\times$, 97% attainment) but is more conservative: the slack budget is small, and the planner admits fewer branches per step. At a generous 100 ms target, TAPER admits more aggressively ($2.18\times$) because the wider slack window tolerates larger step latencies. Even IRP-EAGER improves under this target ($1.52\times$, 78%) because fewer steps violate the relaxed threshold, but TAPER still outperforms it substantially.

Crucially, TAPER requires no reconfiguration across SLO targets. The slack budget adapts automatically: a tighter target produces less slack, which the planner respects by admitting fewer branches. The operator changes the SLO target; TAPER adjusts its behavior.

E.3 Output quality verification

The schedule-invariance lemma (§3.1, Appendix A) guarantees that TAPER's branch deferral does not affect model output under deterministic decoding. We verify this empirically.

Table 6: Output quality verification under greedy decoding. TAPER produces byte-identical outputs to IRP-OFF across all test prompts.

Comparison	Prompts	Identical	Match Rate
TAPER vs. IRP-OFF	1,000	1,000	100%
TAPER vs. IRP-EAGER	1,000	1,000	100%
IRP-EAGER vs. IRP-OFF	1,000	1,000	100%

We run 1,000 prompts (500 decomposable, 500 non-decomposable)sampled uniformly from ShareGPT [20], RAG-12K [21], and Math-220K [22] under greedy decoding with IRP-OFF, IRP-EAGER, and TAPER. All three policies produce byte-identical outputs for every prompt (Table 6). This confirms that schedule invariance holds in practice: the order in which branches are scheduled does not affect the generated tokens, and TAPER's deferral decisions have no semantic consequence. The result also holds for IRP-EAGER vs. IRP-OFF, verifying the broader claim that any valid schedule produces the same output.

E.4 Planner overhead

Table 7 reports the latency overhead of the TAPER planner, measured as the wall-clock time between receiving the batch state and returning the step composition. The overhead is measured over the full 10-hour trace from §4.2.

Table 7: Planner overhead per decode step, measured over the full trace.

	Median	P95	P99	Max
Planner latency (ms)	0.31	0.52	0.81	1.24
% of step latency	0.7%	1.1%	1.6%	2.5%

The median overhead of 0.31 ms is dominated by $T(\cdot)$ evaluations (one per candidate branch per greedy iteration). At P99, the overhead reaches 0.81 ms, driven by steps where many branches are eligible for admission and the greedy loop iterates over more candidates. Even the worst-case 1.24 ms

is less than 2.5% of a typical decode step (50 ms), confirming that per-step replanning is practical. The planner's $O(|\mathcal{R}| \cdot K_{\max} \cdot c_T)$ complexity (§3.4) means overhead scales with the number of requests in the batch $|\mathcal{R}|$, the maximum fanout $K_{\max}$, and the cost c_T of a single $T(\cdot)$ evaluation (two multiplications for our linear model). In practice, batch size is bounded by KV cache capacity to a few hundred requests, and ABF ranges from 2.7 to 5.2 across the workloads characterized in §2.1 (Figure 1), keeping both $|\mathcal{R}|$ and $K_{\max}$ small and the overhead negligible.

E.5 Evaluation on QWen2.5-72B

To verify that TAPER generalizes beyond a single model size, we repeat the main evaluation on QWen2.5-72B [26] served on 8×A100-80GB GPUs (TP=8) under the same trace and workload composition (PDR=50%). QWen2.5-72B's baseline step latency is approximately 2× that of Qwen3-32B, so a 50 ms TPOT target would be violated even by IRP-OFF under high load. We therefore set SLO=100 ms, which provides comparable headroom to the 50 ms target on 32B. No other parameter is changed: TAPER runs with $\rho = 0.8$ and linear utility, and the latency predictor is refitted to the 72B profile.

Table 8: Main result on QWen2.5-72B (SLO=100 ms). The same TAPER configuration ($\rho = 0.8$, linear utility) is used without retuning.

Policy	Goodput / IRP-Off	Attainment
IRP-OFF	1.00×	100%
IRP-C2	1.20×	78%
IRP-C5	1.22×	65%
IRP-EAGER	1.25×	62%
TAPER	1.72×	97%

The throughput trap persists at 72B scale (Table 8). IRP-EAGER achieves 1.25× goodput but attainment collapses to 62%: during the high-load regime, the 2× higher per-step cost amplifies the branch externality, pushing step latency well past 100 ms. TAPER achieves 1.72× goodput with 97% attainment. The goodput ratio is comparable to the 32B result (1.72× vs. 1.77×), confirming that the slack-budget mechanism adapts to the model's latency profile through the refitted predictor $T(S)$. The slightly lower ratio reflects the proportionally larger cost per admitted branch: each branch adds roughly 2× more latency on 72B, so the same slack budget accommodates fewer opportunistic branches per step. The planner compensates by being more selective, admitting branches only when their marginal utility justifies the higher per-branch cost.

E.6 Evaluation with SPRINT frontend

The evaluation up to this point has focused exclusively on Multiverse [9] as the IRP frontend, which produces wide parallel phases (ABF=4.1, PTS=58%) from structured response decomposition. To confirm that TAPER is frontend-agnostic, we evaluate with SPRINT [25], which targets a different decomposition regime: reasoning chains with interleaved planning and parallel execution. SPRINT produces shorter, more frequent parallel phases with narrower branching (ABF=2.8, PTS=35%, PDR=65%), reflecting the structure of multi-step reasoning where a planner identifies 2–3 independent sub-steps per round.

Table 9: Evaluation with SPRINT as the IRP frontend on Qwen3-32B (SLO=50 ms). The same TAPER configuration ($\rho = 0.8$, linear utility) is used without modification.

Policy	Goodput / IRP-Off	Attainment
IRP-OFF	1.00×	100%
IRP-C2	1.15×	82%
IRP-EAGER	1.18×	68%
TAPER	1.45×	98%

The throughput trap persists under SPRINT but is less severe (Table 9). IRP-EAGER's attainment drops to 68% (vs. 48% with Multiverse) because the lower fanout produces smaller per-phase

externalities, though SPRINT's more frequent phase transitions cause the externality to accumulate, and the aggregate effect still violates SLOs under high load. TAPER achieves $1.45\times$ goodput with 98% attainment. Both the lower goodput ratio ($1.45\times$ vs. $1.77\times$) and the higher attainment (98% vs. 97%) trace to the same cause: narrower branching means less opportunity to harvest but also less externality to contain.

The difference in frontend structure is absorbed by the planner's per-step loop. With Multiverse, it makes fewer but larger admission decisions (choosing from a fanout of 4–5). With SPRINT, it makes more frequent but smaller decisions (choosing from a fanout of 2–3). The slack budget and latency predictor operate on step composition, the number of active sequences and their aggregate context length, which is invariant to how the frontend discovered or organized the branches. This confirms the design claim of §3.1: TAPER operates on the serving-visible structure of parallel phases, not on the frontend-specific mechanism that produced them.

F Extended Related Work

Intra-request parallelism. A growing body of work exposes intra-request parallelism within individual LLM responses, progressing from external orchestration toward model-native support. SoT [4] prompts for an outline and expands points via parallel API calls, but re-encodes the full skeleton per branch with no KV sharing. APAR [6] and APR [7] move branch discovery into the model by fine-tuning it to emit fork/child tokens, using tree-based attention that isolates branches (the same property TAPER formalizes as the visibility rule), though APAR discards branch KV at reduce, incurring information loss. PASTA [5] goes further by teaching models to annotate their own outputs with promise/async tokens, introducing the useful concept of semantic independence, but its pre-allocated position ranges create encoding conflicts when branch lengths deviate. ASPD [8] resolves this with shared position encodings and branch-invisible masks within a single sequence; its framing of generation as interleaved serial and parallel stages directly informs TAPER's parallel-phase formalism. However, ASPD's single-sequence architecture requires all branches to advance together at every step, structurally forcing the eager policy that §2.2 shows is brittle. Multiverse [9] provides the closest runtime infrastructure to what TAPER assumes, treating branches as separate sequences over a radix-tree KV cache, but its scheduler likewise admits all ready branches eagerly. Across all of these methods, the focus is on the single-request problem: how to expose and execute branches faster. TAPER sits at a different level, taking whatever branch structure the frontend provides and deciding how much of it to admit into each shared decode step. ParallelPrompt [11] benchmarks the phenomenon at scale, finding roughly 10% of real prompts decomposable; a recent survey [12] covers the broader parallel text generation landscape. A related line of work blurs the boundary between speed and quality: Hogwild! Inference [34] runs parallel workers over a shared, concurrently-updated KV cache with full mutual visibility (the schedule-invariance lemma does not hold under this architecture), while SPRINT [25] and Parallel-R1 [35] use concurrent reasoning threads for both faster and better answers. Like the methods above, none considers multi-request scheduling.

Speculative decoding. A separate class of methods also produces multiple tokens per step but at a different granularity. Speculative decoding [37], along with variants such as Medusa [38] and Eagle [39], generates draft tokens within a single autoregressive sequence and verifies them against the target model. This is token-level parallelism, whereas IRP is segment-level: branches are semantically independent subsequences, not speculative continuations of one sequence. Recent work on regulating speculative decoding shares TAPER's motivation of adaptive control under varying load. TurboSpec [40] adjusts speculation length via a goodput-based feedback loop, AdaSpec [41] and AdaServe [42] tune speculative strategy for SLO attainment, and Nightjar [43] disables speculation entirely under high load to reclaim memory for larger batches. These controllers regulate a private cost: rejected draft tokens waste compute for the speculating request alone. TAPER regulates an externality: admitted branches inflate step latency for every co-batched request. The two are orthogonal and composable; each branch within a parallel phase could internally use speculative decoding, with TurboSpec-style regulation governing draft length and TAPER governing branch width.

LLM serving schedulers. The scheduling problem TAPER addresses is distinct from, but structurally related to, existing work on LLM serving. Modern engines [18, 19] batch requests into shared

decode steps and manage KV memory via paging or radix-tree allocation. FairBatching [36] is TAPER's closest structural sibling: it converts decode slack into a time budget for chunked prefill using a calibrated latency predictor, replacing token-budget proxies with time-budget reasoning. TAPER's slack-budget mechanism follows the same principle but differs in granularity (branches within a parallel phase vs. prefill chunks across requests), barrier semantics (branches must all complete before reduce; prefill and decode have no joint constraint), and value decoupling (TAPER exposes utility as a pluggable curve rather than a fixed priority order). FairBatching cannot be directly adapted to this setting because its time-budget mechanism assumes a binary choice (admit a prefill chunk or not), whereas TAPER faces a combinatorial allocation across multiple requests with varying branch counts and utilities. Andes [44] also exploits slack, finding it in the gap between token delivery rate and user reading speed, and uses it to reprioritize requests at token granularity. Where Andes decides *which request* to advance, TAPER decides *how many branches* a given request gets; the two concerns compose. NexusSched [45] predicts per-step latency for engine-level batch formation and cluster-level routing; its online-learning approach to latency prediction is a natural enhancement for TAPER's offline-fitted linear model. vLLM [18] handles memory pressure via swap-based request-level preemption, which is expensive to reverse. TAPER's branch deferral avoids this cost entirely: the prefix KV remains resident for admitted siblings, and branch-local state is small enough to leave in place, making width changes reversible at zero cost. A natural question is whether a simpler adaptive mechanism would suffice, for example a reactive controller that observes the previous step's latency and adjusts width via multiplicative-increase/multiplicative-decrease (MIMD). Such a controller would adapt to load but lacks two properties TAPER provides: it is backward-looking, reacting to latency already incurred rather than predicting it before committing to a step composition, causing overshoots during sharp load transitions; and it has no notion of per-request slack, adjusting a global width parameter without knowing which requests are near their deadlines. TAPER's slack budget takes the minimum over all requests' residual slack, ensuring width is safe for the most urgent request in the batch.