Pacing Branch Parallelism in LLM Serving

Swapnil Gandhi[1] **Siva Kumar Sastry 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 PACE, a per-step admission controller that treats extra branches as opportunistic width, 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, PACE improves goodput by $1.77\times$ over serving without branch parallelism and by $1.48\times$ over eager admission while maintaining 99% SLO attainment. Comparable gains (1.72–$2.15\times$) hold on five additional dense and mixture-of-experts models with up to 671B parameters. Code is available at `https://github.com/Swapnil-Gandhi/PACE`.

1 Introduction

As LLM responses grow longer, driven by chain-of-thought reasoning [1, 2], structured generation, 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. That 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 raw 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, which bound how many branches each request may decode per step, help in one operating regime and hurt in another, because the safe width depends on batch composition, context lengths, and

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

accumulated slack, all of which change continuously over a workload trace. Reactive controllers that adjust a global cap from observed SLO violations fare better, but they respond only after violations accumulate and apply one width to every request, regardless of how close each is to its deadline. What is needed is a per-step admission controller that predicts the cost of a wider step before committing to it, widening when the current batch has headroom and contracting when requests in serial stages are at risk. It must also be cheap enough to revise its decision at every step.

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 either to keep its KV state resident, consuming scarce memory, or to 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 stage, 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 is therefore cheap, voluntary, and reversible. This decoupling is what makes per-step width regulation practical.

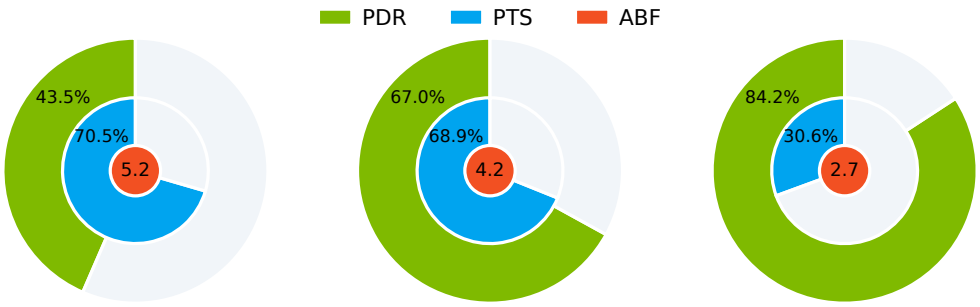
We introduce PACE (Per-step Admission Control for Externalities), a runtime controller that exploits this decoupling. PACE first protects *single-token progress*: every active request is guaranteed one token in the next step. It then treats additional branches as *opportunistic width*, admitting them only when the predicted branch externality fits within the batch's current slack budget. When slack is abundant, PACE 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, PACE 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, fixed-cap, and reactive 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 (§3.1, §3.5).
- We introduce PACE, 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 PACE under mixed workloads on six models, two dense and four mixture-of-experts. On Qwen3-32B, it improves goodput by $1.77\times$ over disabling IRP and by $1.48\times$ over eager admission while maintaining 99% SLO attainment, versus below 50% for eager admission and 79% for a reactive width controller (§4).

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 neither fixed nor reactive policies navigate 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 (SoT) [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 autoregressive continuation; in parallel stages, the model decodes multiple independent branches concurrently, each still autoregressive. 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 [20] has the highest PDR at 84.2%, followed by RAG-12K [21] at 67.0% and ShareGPT Vicuna [22] at 43.5%. The *parallel token share* (PTS) measures how much of a decomposable response is parallel: ShareGPT (70.5%) and RAG-12K (68.9%) are dominated by parallel tokens when they decompose, while Math-220K (30.6%) produces short, narrow parallel stages. The *average branch fanout* (ABF) indicates the typical 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 its combination with 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; the branch externality already erodes SLO attainment when only one request in ten is decomposable (Appendix E.1). 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 in this mixed-batch setting that the cost of branch parallelism becomes visible.

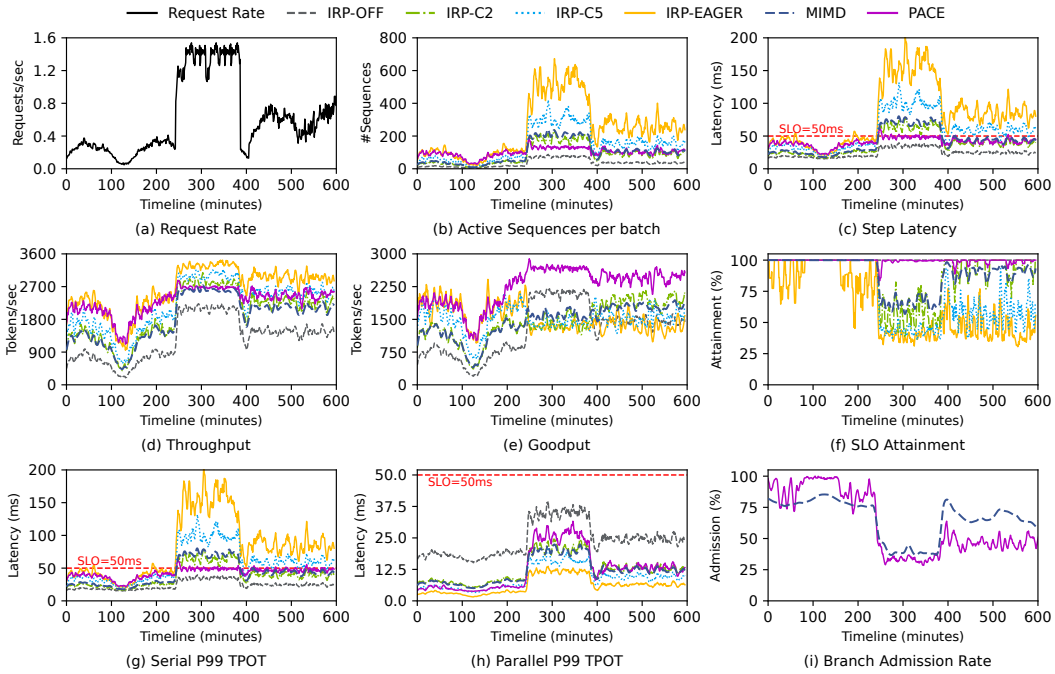


Figure 2: **The throughput trap and its resolution.** Four fixed step-width policies, a reactive width controller (MIMD), and PACE on a mixed workload. IRP-EAGER raises throughput but collapses goodput and SLO attainment under load, and the cost falls asymmetrically on requests in serial stages. PACE adjusts its branch admission rate every step (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. 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 fixed step-width policies: IRP-OFF ($w_{r,t} = 1$), IRP-C2 and IRP-C5 (which cap $w_{r,t}$ at 2 and 5), and IRP-EAGER ($w_{r,t} = n_r$). A fifth policy, MIMD, adapts a global width cap reactively from observed SLO violations. At low load, larger step widths extract more raw throughput: the batch has headroom to absorb wider steps. 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 more 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 but an asymmetric transfer: requests in serial stages absorb the latency, while requests in parallel stages see their lowest P99 TPOT under IRP-EAGER.

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 also leave throughput unused at low load. 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.

Reactive adaptation narrows the gap but does not close it. MIMD grows its width multiplicatively while steps meet the target and cuts it when violations accumulate. It reaches the best attainment of any branch-admitting baseline (79%; Appendix E.8), yet it errs in both directions (2(i)): at low load it admits fewer branches than the batch can absorb, and under moderate load it admits more than the most urgent requests can tolerate. Both errors are structural. A windowed controller acts only after violations have occurred, and a single global width cannot distinguish a request near its deadline from one with ample slack.

2.3 Branch externality

The throughput trap and its asymmetric incidence reduce to a single quantity. The asymmetry 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 $S(\mathbf{a})$ be a widened step that admits additional branches according to an allocation $\mathbf{a} = (a_r)_{r\in\mathcal{R}}$ over the active requests $\mathcal{R}$, where $a_r \geq 0$ is the number of additional branches granted to request r, which then decodes with step width $w_{r,t} = 1 + a_r$ (IRP-OFF sets $a_r = 0$; IRP-EAGER sets $a_r = n_r - 1$). The branch externality is

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

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

Every request in the batch pays $E_t(\mathbf{a})$; only requests in parallel stages with $a_r > 0$ 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 a_r: admitting more branches cannot shorten a decode step.

What the runtime needs is neither a tighter cap nor a faster reaction, but a per-step controller that predicts $E_t(\cdot)$ against the current batch state before committing to a step, and admits branches only when the predicted externality fits within the batch's available slack. The next section introduces PACE, such a controller. Its behavior is already visible in Figure 2: PACE's step latency tracks near IRP-EAGER at low load and stays near the SLO target during stress events (2(c)), while its branch admission rate (2(i)) shows the control signal adapting per step.

3 PACE: Per-Step Admission Control for Externalities

This section builds the controller that §2 calls for. We formalize the parallel phase and show that per-step width decisions do not change model output (§3.1), then develop a cost model (§3.2), a slack-budgeted admission rule (§3.3), and a greedy planner that combines them (§3.4). We close by showing why per-step regulation is structurally cheap (§3.5).

3.1 Parallel phases and schedule invariance

A *parallel phase*, the formal counterpart of a parallel stage (§2.1), 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, both headers, 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 up to numerical nondeterminism; 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.

The step width $w_{r,t}$ is therefore a free parameter of the schedule, not a constraint of the model: PACE may defer, interleave, or reorder branches without affecting model output.

3.2 Predicting the cost of a widened step

To decide whether admitting additional branches is safe, PACE 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. PACE uses a calibrated latency predictor $T(S)$ over this composition, requiring only that T is monotone non-decreasing in the number of admitted branches. Our implementation fits a linear model in these two quantities offline and refits it periodically from observed step latencies, reaching 1.8% mean absolute percentage error across batch sizes from 1 to 512 (Appendix C). Monotonicity is the key structural property: if admitting one more branch from request r exceeds the slack budget, admitting two more will too, enabling the greedy planner's pruning rule (§3.4).

3.3 Slack-budgeted admission

PACE 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 admission. Admitting ready branches widens the step ($w_{r,t} > 1$), so PACE treats this additional width as *opportunistic*: it is admitted only when the predicted externality fits within the slack budget.

To decide which opportunistic branches are safe, PACE 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 protected composition takes predicted time $T_0 = T(S_0)$, the residual time available for opportunistic width is

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

PACE spends a *slack fraction* $\rho \in (0, 1]$ of B_t. A candidate widened step S is admitted only if

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

This bounds the branch externality, $E_t(\mathbf{a}) \le \rho\, B_t$: the predicted cost of widening must fit within the fraction of slack the operator is willing to spend. The reserve $1 - \rho$ absorbs prediction error. A small ρ leaves usable slack idle, while ρ near 1 lets underestimates push steps past the deadline. We use $\rho = 0.8$ throughout; §4.3 sweeps ρ, and Appendix E.9 adapts it online from observed attainment.

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

Algorithm 1 PACE Per-Step Planner

```
def PACEStep(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
```

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, PACE needs a tiebreaker. We separate safety from value: $T(\cdot)$ decides what is *feasible*; a monotone utility curve $u_r(a_r)$ decides what is *valuable*, where a_r 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. PACE predicts cost; the operator supplies value through the utility interface.

3.4 Per-step planner

At each step, PACE seeks the allocation that maximizes total utility within the slack budget:

$$\max_{\mathbf{a}} \sum_{r \in \mathcal{R}} u_r(a_r) \quad \text{s.t.} \quad T\big(S(\mathbf{a})\big) \le T_0 + \rho\, B_t, \qquad a_r \in \{0, 1, \ldots, n_r - 1\}.$$

The step then decodes $|\mathcal{R}| + K$ tokens, where $K = \sum_r a_r$ is the number of opportunistic branches admitted. Finding the optimal allocation is NP-hard (Appendix B), so PACE widens greedily from the protected composition S_0: 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.

Example. Three requests are active: r_1 and r_2 are in parallel stages with fanouts $n_{r_1} = 3$ and $n_{r_2} = 2$, and r_3 is in a serial stage. S_0 advances each by one token, leaving two ready branches for r_1 and one for r_2. The planner admits a branch from r_1, then r_2's only ready branch; both pass the budget check. Admitting r_1's last branch would push the predicted latency past $T_0 + \rho\, B_t$, so the planner prunes r_1 and stops with $\mathbf{a} = (1, 1, 0)$: r_1 and r_2 each decode two tokens, r_3 decodes one, and the step decodes $|\mathcal{R}| + K = 5$ tokens. The deferred branch stays ready for the next step, with no KV eviction or restoration (§3.5).

The planner runs in $O(|\mathcal{R}| \cdot K_{\max} \cdot c_T)$ per step, where $K_{\max}$ bounds the number of opportunistic branches K admitted in one step and c_T is the cost of one $T(\cdot)$ evaluation, typically a table lookup or short regression well under a millisecond. Monotonicity justifies the pruning rule: once a branch from request r is infeasible, all further branches from r are too. The greedy allocation 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, PACE recomputes per-request slack from the realized step latency and reconsiders any deferred branches; the predictor itself is refit periodically from observed latencies (Appendix C). Schedule invariance (Lemma 3.1) ensures that deferral never changes model output.

Replanning turns the width decision from a one-shot commitment into closed-loop control. As established in §1, the loop is cheap because branch regulation decouples compute from memory: deferring a branch evicts no KV state, and readmitting it 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, PACE 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, PACE tolerates predictor error. A pessimistic $T(\cdot)$ costs width rather than latency: the planner admits fewer branches than it safely could. An optimistic $T(\cdot)$ can over-admit, but each overshoot consumes slack and shrinks B_t at the next step, so the error corrects itself while the reserve $1-\rho$ absorbs it; even a constant predictor holds attainment at 96% (§4.3). In short, PACE treats step width as a renewable, per-step decision. When slack is available, PACE spends it on width; when slack is scarce, PACE 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

The central question is whether PACE can escape the throughput trap of §2.2, harvesting the throughput benefit of intra-request parallelism without degrading SLO attainment. We examine this across three load regimes (§4.2), validate design choices through targeted ablations (§4.3), and test generality across models, frontends, and deployments (§4.4). Appendix E adds sensitivity studies (workload composition, SLO target, batch size) and output-quality verification.

4.1 Experimental setup

We evaluate on Qwen3-32B [26] served with SGLang [19] on 8×A100-80GB GPUs (TP=8) with radix-tree KV allocation. Prefill and decode run on separate instances [27, 28], and PACE runs on the decode instance. Workloads interleave non-decomposable requests from ShareGPT with decomposable requests drawn from the three datasets of §2.1 and decomposed by 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, five times the single-request decode-step latency, following Mooncake [29]. We report raw throughput, goodput, and SLO attainment and compare IRP-OFF ($w_{r,t}=1$), IRP-C2, IRP-C5, IRP-EAGER ($w_{r,t}=n_r$), MIMD, and PACE ($\rho=0.8$, linear utility); Appendix D gives full details.

4.2 Navigating the throughput trap

Figure 2 compares all six policies on the same production-derived trace. PACE'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, the 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 $\rho=0.8$, PACE's slack budget easily accommodates most ready branches: its step latency averages 36 ms, close to IRP-EAGER's 38 ms (2(c)), confirming that PACE admits aggressively when the batch has headroom. PACE's branch admission rate stays near 100% outside transient bursts (2(i)). SLO attainment (2(f)) is 100% for PACE, 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. PACE matches IRP-EAGER's 1.28× goodput without those violations.

High load. Between minutes 250 and 400, the request rate surges past 1.0 req/s and stays 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 3× the SLO target, as the batch fills with eagerly admitted branches. SLO attainment collapses

Table 1: **Ablation study.** Top: removing one component at a time ($\rho = 0.8$). Bottom: varying the slack fraction ρ. The default configuration is in bold.

Variant	**PACE (full)**	w/o slack budget	w/o replanning	constant $T(\cdot)$
Goodput ($\times$ IRP-OFF)	**1.77**	0.92	1.18	1.12
Attainment (%)	**99**	48	82	96

Slack fraction ρ	0.05	0.1	0.2	0.3	0.4	0.5	0.6	0.7	**0.8**	0.9	1.0
Goodput ($\times$ IRP-OFF)	1.00	1.05	1.09	1.13	1.19	1.25	1.43	1.59	**1.77**	1.61	1.45
Attainment (%)	100	100	100	100	100	99.5	99.4	99.2	**99**	94.7	91.3

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. MIMD cuts its width only after violations accumulate, and its step latency stays above the target throughout the stress period. PACE contracts sharply. As slack collapses under the sustained high rate, the planner defers most opportunistic branches. Its 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%, with goodput at $1.68\times$ IRP-OFF versus $0.88\times$ for IRP-EAGER.

Moderate load. After the stress event subsides around minute 400, the request rate settles into a sustained regime averaging 0.60 req/s with peaks near 0.9 req/s. Slack rebuilds, and PACE's branch admission rate recovers to approximately 47% without operator intervention (2(i)). Step latency averages 42 ms (max 52 ms), staying near the SLO target with substantially more branches admitted than during the stress period. IRP-EAGER continues to violate: its step latency averages 85 ms (max 122 ms) and attainment is only 45%. MIMD regrows its width faster than slack rebuilds, admitting more branches than PACE (2(i)) and violating intermittently. 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 PACE 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 and moderate load. PACE adapts to every regime without reconfiguration.

Summary. Over the full 10-hour trace, PACE improves goodput over IRP-OFF by $1.77\times$ and over IRP-EAGER by $1.48\times$ at 99% SLO attainment, while IRP-EAGER drops below 50%. Neither fixed caps nor reactive adaptation matches it: IRP-C5 collapses under stress (attainment 45%), IRP-C2 leaves throughput unused at low load, and MIMD, the best branch-admitting baseline on attainment (79%), converts its $1.44\times$ throughput into only $1.14\times$ goodput (Appendix E.8).

4.3 Ablation

We isolate each component of PACE by removing it in turn (Table 1, top). **Without the slack budget**, PACE admits every ready branch that fits in memory and inherits IRP-EAGER's failure mode: attainment falls to 48% and goodput drops below IRP-OFF ($0.92\times$), as SLO violations destroy more goodput than the extra branches create. **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 trade-off (Table 1, bottom). While attainment stays at or above 99%, goodput rises with ρ, peaking at $\rho = 0.8$ ($1.77\times$). Beyond that point, the remaining headroom no longer absorbs prediction error: attainment falls to 94.7% at $\rho = 0.9$ and 91.3% at $\rho = 1.0$, and goodput falls with it. At the conservative end, $\rho \leq 0.2$ trails even IRP-C2 ($1.11\times$). We therefore recommend $\rho = 0.8$ as the default; when load characteristics drift, a simple outer loop can adapt ρ online, converging from 1.0 to 0.8 in 10 minutes (Appendix E.9). The default holds 97–99.5% attainment for 30–100 ms SLOs (Appendix E.2).

Table 2: **Generality across models.** Goodput over IRP-OFF (SLO attainment); PACE uses $\rho = 0.8$ throughout. MoE setup in Appendix E.7.

Model	Params (total / active)	SLO	IRP-EAGER	MIMD	PACE
Qwen3-32B [26]	32B / 32B	50 ms	1.20× (48%)	1.14× (79%)	**1.77**× (99%)
GPT-OSS-120B [30]	116.8B / 5.1B	100 ms	1.26× (61%)	1.18× (86%)	**1.91**× (99%)
DeepSeek-V4-Flash [31]	284B / 13B	100 ms	1.33× (68%)	1.22× (91%)	**2.15**× (99%)
Llama 4 Maverick [32]	400B / 17B	100 ms	1.30× (65%)	1.20× (88%)	**2.05**× (99%)
DeepSeek-R1 [2]	671B / 37B	100 ms	1.24× (56%)	1.16× (82%)	**1.88**× (99%)

4.4 Generality

Models. Table 2 adds four mixture-of-experts (MoE) models spanning 116.8B to 671B total parameters, three attention mechanisms, three numerical formats, and two GPU generations; their higher single-request step latency sets a 100 ms TPOT target. With the same configuration and a refit predictor, PACE achieves 1.88–2.15× goodput at 99% attainment on all four, while IRP-EAGER reproduces the throughput trap and MIMD again recovers attainment only partially. Planner overhead stays below 1 ms at P99 on every model, as on Qwen3-32B (0.81 ms P99; Appendix E.4), since its cost scales with the batch's branch structure rather than model size. On the dense Qwen2.5-72B, PACE achieves 1.72× goodput at 97% attainment (Appendix E.5).

Frontends and deployment. With SPRINT [25] as the frontend, whose reasoning-oriented decomposition yields narrower branching (ABF=2.8), PACE achieves 1.45× goodput at 98% attainment (Appendix E.6). PACE also leaves time-to-first-token (TTFT) intact. Under disaggregation, branch admission never touches the prefill instance; in a co-located deployment with chunked prefill [33], PACE's P99 TTFT stays within 6 ms of IRP-OFF, while IRP-EAGER's more than doubles (Appendix E.11).

5 Related Work

Existing work on parallel LLM generation divides into two objectives [8]. *Quality-oriented* methods such as best-of-N sampling [34], beam search [35], self-consistency [36], majority voting [37], MCTS [38], and Tree-of-Thoughts [39] 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 [40, 25, 41] blurs this boundary, using concurrent execution for both speed and quality. All of these operate on a single request in isolation; none considers what happens when the resulting branches share a decode step with other requests. PACE takes the branch 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 [42], which converts decode slack into a time budget for chunked prefill. PACE's slack-budget mechanism follows the same principle but operates within a request's parallel phase rather than across the prefill and decode phases. Extended comparisons with LLM serving schedulers and preemption mechanisms appear in Appendix F.

6 Conclusion & Limitations

PACE 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× and over IRP-EAGER by 1.48× at 99% SLO attainment on Qwen3-32B, with comparable gains on five additional dense and MoE models. The current design assumes branch independence (§3.1), uses a linear latency predictor that may lose accuracy at extreme batch sizes, and regulates one serving instance at a time; because a request's branches share its prefix KV and stay on its instance, a multi-instance deployment simply runs one controller per instance (Appendix E.11). Learned latency predictors and richer online control of ρ are natural extensions. More broadly, 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.

Acknowledgments and Disclosure of Funding

We are grateful to the anonymous reviewers whose comments have greatly helped improve this paper. We also thank NVIDIA for providing the computational resources used in early experiments for this work. This research was partly supported by the Stanford Platform Lab and its affiliates, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

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*, NeurIPS '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, 2025. URL https://arxiv.org/abs/2501.12948.

[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, Tian Tang, Qinyu Xu, 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, pages 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*, NeurIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

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

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

[22] 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. URL https://arxiv.org/abs/2306.05685.

[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, 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.

[27] 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)*, pages 118–132, 2024. doi: 10.1109/ISCA59077.2024.00019.

[28] 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)*, pages 193–210, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

[29] 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)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/qin.

[30] OpenAI. Introducing gpt-oss. https://openai.com/index/introducing-gpt-oss/, 2025.

[31] DeepSeek-AI. DeepSeek-V4-Flash. https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash, 2026.

[32] Meta. Llama 4 Maverick 17B-128E Instruct. https://huggingface.co/meta-llama/Llama-4-Maverick-17B-128E-Instruct, 2025.

[33] 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)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/agrawal.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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*, NeurIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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, 2024. URL https://arxiv.org/abs/2412.15115.

[45] Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling, 2025. URL https://arxiv.org/abs/2501.19393.

[46] 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.

[47] 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.

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

[49] 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.

[50] 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, pages 361–374. ACM, November 2025. doi: 10.1145/3772052.3772239. URL http://dx.doi.org/10.1145/3772052.3772239.

[51] 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.

[52] 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.

[53] 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; restatement of Lemma 3.1)**.** *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 produce identical outputs 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

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

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 on the schedule only through $y_{i,<t}$ (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 and no branch conditions on another's tokens, 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 has the same distribution under any valid schedule: it depends on the completed branch outputs (same by the argument above), their ordering (canonical, not schedule-dependent), and the prior reduce tokens $z_{<t}$ (same by induction on t). The conditional distribution $\Pr(z_t \mid \cdot)$ 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 [40]) 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 PACE 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 $\mathbf{a} = (a_r)_{r\in\mathcal{R}}$ that maximizes total utility $\sum_r u_r(a_r)$ subject to $T(S(\mathbf{a})) \le T_{\max}$, with each a_r at most the number of ready branches of request r.*

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 of size s_j and value v_j ($j = 1, \dots, m$) and capacity C, construct a width allocation instance as follows. Create m requests, each with exactly one ready branch. For request j, set the utility of admitting its branch to $u_j(1) = v_j$ and the marginal latency cost to $T(S(\mathbf{e}_j)) - T(S_0) = s_j$, where $\mathbf{e}_j$ is the allocation that admits only request j's branch. Set $T_{\max} = T(S_0) + C$. Assume latency is additive in the marginal costs (which holds when branch contributions to FFN and attention are independent). The instance then amounts to selecting a subset of branches whose total marginal latency cost does not exceed C and whose total utility is maximized, which is 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 revisited at step $t + 1$ against the updated batch state and slack budget.

C Latency Predictor and Memory Accounting

C.1 Predictor formulation

PACE requires a predictor $T(S)$ that maps a step composition S to a predicted wall-clock latency in milliseconds. Following the observation [15] 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:

$$T(S) \;=\; \theta_0 \;+\; \theta_1 \cdot n_{\text{tokens}} \;+\; \theta_2 \cdot L_{\text{context}}\,,$$

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

Table 3: Latency predictor accuracy across operating regimes.

Regime	Batch size range	MAPE (%)
Low load	1–64	1.7
Moderate 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 $(\theta_0, \theta_1, \theta_2)$ 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 (Appendix D), the linear model achieves a mean absolute percentage error (MAPE) of 1.8% across the profiling grid (Table 3).

Monotonicity. The predictor satisfies the monotonicity assumption of §3.2 by construction: admitting one additional branch increases both n_{tokens} and L_{context}, and $\theta_1, \theta_2 > 0$.

Robustness to predictor bias. Random error and systematic bias affect PACE differently. Random error is absorbed by the reserve $1 - \rho$ of the slack budget (§3.3). Systematic bias, for example from thermal throttling or clock variation, persists until the next refit. During that window, an underestimating predictor over-admits, but the error is self-correcting: each overshoot consumes deadline slack, shrinking B_t (and therefore $\rho\, B_t$) at subsequent steps until the refit absorbs the bias. The constant-predictor ablation (§4.3) bounds the cost of a stale predictor: even a predictor frozen at a single operating point maintains 96% attainment, because the deadline slack in B_t is recomputed from realized latency at every step, independently of predictor quality.

C.2 Memory accounting

Although PACE'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.

Deployment and hardware. We use a prefill-decode disaggregated deployment [27, 28]: a prefill instance processes prompts and populates their KV caches, and a decode instance runs all decode steps. Because IRP is a decode-phase technique (no branch participates in a prefill forward pass), PACE runs exclusively on the decode instance. The decode instance is a single node with 8×A100-80GB GPUs connected via NVLink, serving the model in tensor-parallel mode (TP=8) with one shard per GPU. Qwen3-32B's BF16 weights occupy about 64 GB (8 GB per GPU), leaving about 576 GB across the node for KV cache and activations. KV cache is the binding constraint: at peak load it consumes roughly 62% of total GPU memory, and overall memory utilization reaches approximately 78%. This sizing sustains the batch sizes in our evaluation, up to roughly 650 active sequences at peak (Figure 2(b)), without KV eviction, and the remaining headroom accommodates branch-local KV from deferred branches without triggering request-level preemption.

Model. We use Qwen3-32B [26], 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 in BF16 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. PACE'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 [22]. These requests produce purely serial output and represent the serial traffic that is vulnerable to the branch externality.
- **Decomposable requests** are sampled uniformly from ShareGPT Vicuna [22], RAG-12K [21], and Math-220K [20]. These prompts are processed through Multiverse [9], which decomposes each response into a sequence of serial and parallel stages. The resulting branch structure has an ABF of 4.1 and a PTS of 58% among decomposable requests.

Context lengths. Table 4 summarizes the context lengths of the three datasets, which cover qualitatively different regimes. RAG-12K pairs long inputs with short outputs: high per-step attention cost raises T_0 and tightens the slack budget B_t, pushing PACE toward conservative admission. Math-220K pairs short inputs with long outputs: low T_0 leaves ample slack throughout a long decode, so PACE admits aggressively. ShareGPT Vicuna lies in between. Blending the three datasets exercises PACE across a wide range of per-step costs and slack conditions.

Table 4: Context lengths of the evaluation datasets (tokens).

Dataset	Input P50	Input P90	Output P50	Output P90
ShareGPT Vicuna	105	840	363	1,372
RAG-12K	860	1,439	242	827
Math-220K	58	118	1,720	5,180

Branch fanout distribution. Table 5 summarizes the branch fanout distribution across decomposable requests in the evaluation workload.

Table 5: 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 six policies to ensure a controlled comparison. It supplies only arrival timestamps; prompts and output lengths come from the datasets above.

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 TPOT (across all stages) does not exceed the target. We set the target following the methodology of Mooncake [29]: five times the single-request decode-step latency, measured as the P50 step latency when each request in the workload is replayed in isolation (batch size 1, no co-tenants). This latency is about 10 ms for Qwen3-32B, giving 50 ms, and about 19 ms for Qwen2.5-72B, giving 95 ms, which we round to 100 ms. The 50 ms target also matches the tight tier used by prior SLO-aware serving work [43]. The single-request latency anchors only the target; admission decisions use the context-dependent predictor $T(\cdot)$, so the slack budget tightens on long-context steps and widens on short ones.

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 TPOT target).

PACE configuration. Unless otherwise noted, PACE runs with slack fraction $\rho = 0.8$ and linear utility $u_r(a_r) = a_r$. 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 Appendix E.4.

Baselines. All six 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 protected composition are admitted.
- IRP-C2: $w_{r,t} = \min(n_r, 2)$. Each request in a parallel stage decodes at most 2 branches per step.
- IRP-C5: $w_{r,t} = \min(n_r, 5)$. At most 5 branches per step.
- IRP-EAGER: $w_{r,t} = n_r$. All ready branches are admitted every step.
- MIMD: $w_{r,t} = \min(n_r, W)$, where W is a global width cap adapted by multiplicative increase and multiplicative decrease. Every N decode steps, the controller computes the fraction of steps whose realized latency exceeded the SLO target. If the fraction exceeds 1%, matching the 99% attainment target, it sets $W \leftarrow \max(1, \lfloor \beta W \rfloor)$; otherwise it sets $W \leftarrow \min(W_{\max}, \lceil \alpha W \rceil)$, where $W_{\max}$ is an upper bound on W. We use $N = 10$, $\alpha = 1.25$, and $\beta = 0.5$, the best setting from the sweep in Appendix E.8.
- PACE: $w_{r,t}$ 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 6 varies this proportion from 5% to 80% across three load regimes to assess how PACE 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 PACE configuration ($\rho = 0.8$, linear utility). Goodput is normalized by each cell's corresponding IRP-OFF baseline.

The relationship between PDR and IRP-EAGER's attainment is not monotone, because PDR determines both the amount of interference and the size of the vulnerable population.

At low PDR, the externality shrinks but persists until decomposable requests are rare. At PDR=5%, with one request in twenty decomposable, it is negligible and the two policies converge ($1.01\times$ aggregate goodput for eager vs. $1.02\times$ for PACE, both at $\geq$99% attainment). By PDR=10%, the attainment gap opens: eager's aggregate attainment falls to 83%, and to 65% in the high-load regime, because even a small decomposable population concentrates its externality in high-load periods. A fixed cap could prevent these violations only by throttling width globally, forfeiting goodput at low load, when B_t is ample and branches could be admitted safely.

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 PACE perform similarly ($1.08\times$), and eager attainment is 97%. At moderate load, interference grows and eager attainment drops to 82% as the large non-decomposable majority begins to feel the cost. During the high-load regime, the same decomposable 20% of requests generate sufficient interference to push step latency well above the target, collapsing eager attainment to 48% as the damage falls on the other 80%. PACE 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 falls further to 41% at high load. This is the worst-case combination: enough decomposable requests to cause substantial interference, and enough non-decomposable requests to suffer from it.

At PDR=80%, per-step interference is at its highest, but the vulnerable population is small: only 20% of requests are non-decomposable. At low load, eager achieves $1.48\times$ goodput with 93%

Table 6: Sensitivity to workload composition across load regimes. Goodput is normalized by IRP-OFF for each PDR and regime.

		IRP-EAGER		PACE	
PDR	Regime	Goodput	Att.	Goodput	Att.
5%	Low	1.02×	100%	1.02×	100%
	Moderate	1.02×	100%	1.02×	100%
	High	0.99×	97%	1.03×	100%
	Aggregate	1.01×	99%	1.02×	100%
10%	Low	1.04×	99%	1.04×	100%
	Moderate	1.03×	87%	1.08×	99%
	High	0.96×	65%	1.13×	99%
	Aggregate	1.03×	83%	1.08×	99%
15%	Low	1.06×	98%	1.06×	100%
	Moderate	1.05×	83%	1.19×	99%
	High	0.94×	57%	1.24×	99%
	Aggregate	1.04×	79%	1.19×	99%
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%
35%	Low	1.18×	93%	1.18×	100%
	Moderate	1.07×	64%	1.52×	99%
	High	0.91×	45%	1.53×	98%
	Aggregate	1.09×	67%	1.42×	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%

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. Decomposable requests, comprising 80% of the workload, are partly shielded: during parallel stages, their effective TPOT (step latency divided by branch fanout) stays well below the target even when step latency is inflated.

PACE maintains $\geq$97% attainment in every cell. At low load across all PDRs, PACE matches eager's goodput exactly while holding 100% attainment: the slack budget is permissive, so PACE defers only the few branches that would push transient bursts past the target. The divergence appears at moderate and high load, where PACE's goodput gain over IRP-OFF grows with PDR (from 1.08–1.13× at PDR=10% to 2.05–2.10× at PDR=80%) because higher PDR provides more branches for opportunistic admission while the slack budget prevents the SLO violations that destroy eager's goodput. At low PDR, fewer branches are eligible per step, so the planner's greedy loop terminates earlier and its overhead (Appendix E.4) is smaller still.

E.2 SLO target sensitivity

Table 7 varies the TPOT target from 30 ms (tight) to 100 ms (generous) to assess how PACE adapts to different latency requirements. All experiments use PDR=50%.

Table 7: Sensitivity to SLO target. PACE adapts automatically via the slack budget; no reconfiguration is needed.

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

At a tight 30 ms target, IRP-EAGER's goodput drops below IRP-OFF (0.87×) because its step latency exceeds 30 ms even at low load. PACE remains effective (1.35×, 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, PACE admits more aggressively (2.18×) because the larger slack budget tolerates larger step latencies. Even IRP-EAGER improves under this target (1.52×, 78%) because fewer steps violate the relaxed threshold, but PACE still outperforms it substantially.

Crucially, PACE 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.

E.3 Output quality verification

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

Table 8: Output quality verification under greedy decoding. PACE produces byte-identical outputs to IRP-OFF across all test prompts.

Comparison	Prompts	Identical	Match Rate
PACE vs. IRP-OFF	1,000	1,000	100%
PACE 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 Vicuna [22], RAG-12K [21], and Math-220K [20] under greedy decoding with IRP-OFF, IRP-EAGER, and PACE. All three policies produce byte-identical outputs for every prompt (Table 8). This confirms that schedule invariance holds in practice: the order in which branches are scheduled does not affect the generated tokens, and PACE's deferral decisions have no semantic consequence. The result also holds for IRP-EAGER vs. IRP-OFF, supporting the broader claim that any valid schedule produces the same output.

E.4 Planner overhead

Table 9 reports the latency overhead of the PACE 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 9: 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 the 50 ms TPOT target, 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 bound $K_{\max}$ on opportunistic branches admitted per step, 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), so each request contributes only a few candidates and the slack budget further limits K, keeping the overhead negligible.

E.5 Evaluation on Qwen2.5-72B

To verify that PACE generalizes beyond a single model size, we repeat the main evaluation on Qwen2.5-72B [44] served on 8×A100-80GB GPUs (TP=8) under the same trace and workload composition (PDR=50%). Qwen2.5-72B's single-request decode-step latency is about 19 ms, roughly twice that of Qwen3-32B, so a 50 ms TPOT target would be violated even by IRP-OFF under high load. Following the methodology of Appendix D, we set SLO=100 ms, which provides comparable headroom to the 50 ms target on 32B. No other parameter is retuned: PACE runs with $\rho = 0.8$ and linear utility, and the latency predictor is refitted to the 72B profile.

Table 10: Main result on Qwen2.5-72B (SLO=100 ms). The same PACE 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%
PACE	1.72×	97%

The throughput trap persists at 72B scale (Table 10). 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. PACE 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 twice as much latency on 72B as on 32B, so the same slack budget accommodates fewer opportunistic branches per step. The planner compensates automatically, admitting branches only while the widened step's predicted latency stays within $T_0 + \rho\, B_t$.

E.6 Evaluation with SPRINT frontend

All other experiments use Multiverse [9] as the IRP frontend, which produces wide parallel phases (ABF=4.1, PTS=58%) from structured response decomposition. To confirm that PACE 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 SPRINT's planner identifies 2–3 independent sub-steps per round.

Table 11: Evaluation with SPRINT as the IRP frontend on Qwen3-32B (SLO=50 ms). The same PACE 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%
PACE	1.45×	98%

The throughput trap persists under SPRINT but is less severe (Table 11). IRP-EAGER's attainment drops only to 68% (vs. 48% with Multiverse): the lower fanout produces smaller per-phase exter-

nalities, but SPRINT's more frequent phase transitions let them accumulate, so the aggregate effect still violates SLOs under high load. PACE achieves $1.45\times$ goodput with 98% attainment. The lower goodput ratio ($1.45\times$ vs. $1.77\times$) reflects narrower branching: fewer ready branches per step leave less parallelism to harvest. Attainment remains comparable (98% vs. 99%).

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: PACE operates on the serving-visible structure of parallel phases, not on the frontend-specific mechanism that produced them.

E.7 Mixture-of-experts models

Our dense-model results use two GQA models in BF16: Qwen3-32B (§4) and Qwen2.5-72B (Appendix E.5). Table 12 lists four additional mixture-of-experts (MoE) models. Together they span three attention mechanisms, three numerical formats, two GPU generations, and, with Qwen3-32B, a $21\times$ range of total parameters.

Table 12: Configuration of the MoE models. GQA: grouped-query attention; MLA: multi-head latent attention; CSA+HCA: hybrid of compressed sparse attention and heavily compressed attention.

Model	Params (total / active)	Attention	Precision	Hardware
GPT-OSS-120B [30]	116.8B / 5.1B	GQA, local/global	BF16	8×A100
DeepSeek-V4-Flash [31]	284B / 13B	CSA+HCA	INT8	8×H200
Llama 4 Maverick [32]	400B / 17B	GQA	BF16	8×H200
DeepSeek-R1 [2]	671B / 37B	MLA	FP8 (E4M3)	8×H200

Fine-tuning. We adapt each model to produce parallel generation following the supervised fine-tuning procedure of Multiverse [9]. Both training sets derive from s1K-1.1 [45]: the 1K Multiverse parallel examples, prompted with "Think step by step and in parallel," and the original sequential examples, prompted with "Think step by step." Following Multiverse, the mixing ratio is annealed linearly from all sequential to all parallel over eight epochs. Fine-tuning takes 8 hours per model on 8×H200 GPUs with DeepSpeed (12 hours for DeepSeek-R1).

Results. Table 13 reports all policies under a 100 ms TPOT target, which reflects the higher single-request step latency of these models. PACE achieves 1.88–$2.15\times$ goodput at 99% attainment on all four models. The fixed caps and MIMD improve goodput modestly but forfeit attainment, and IRP-EAGER reaches the highest goodput among the baselines (1.24–$1.33\times$) at the lowest attainment (56–68%), reproducing the throughput trap of §2.2 on every model. Planner overhead is 0.33 ms at the median and 0.83 ms at P99 across the four models, in line with Qwen3-32B (Table 9): the planner's cost tracks the number of active requests and admitted branches, not model depth or parameter count.

Table 13: Results on MoE models. Goodput normalized to IRP-OFF, with SLO attainment in parentheses; SLO=100 ms.

Policy	GPT-OSS-120B	DeepSeek-V4-Flash	Llama 4 Maverick	DeepSeek-R1
IRP-OFF	$1.00\times$ (100%)	$1.00\times$ (100%)	$1.00\times$ (100%)	$1.00\times$ (100%)
IRP-C2	$1.15\times$ (82%)	$1.18\times$ (88%)	$1.16\times$ (85%)	$1.13\times$ (79%)
IRP-C5	$1.22\times$ (67%)	$1.28\times$ (74%)	$1.25\times$ (71%)	$1.20\times$ (62%)
MIMD	$1.18\times$ (86%)	$1.22\times$ (91%)	$1.20\times$ (88%)	$1.16\times$ (82%)
IRP-EAGER	$1.26\times$ (61%)	$1.33\times$ (68%)	$1.30\times$ (65%)	$1.24\times$ (56%)
PACE	$1.91\times$ (99%)	$2.15\times$ (99%)	$2.05\times$ (99%)	$1.88\times$ (99%)

E.8 Reactive width control

A natural, simpler alternative to PACE is a controller that adapts width from observed latency rather than predicting it. MIMD (Appendix D) instantiates this idea in the same policy space as the fixed

caps, adapting the global cap W online rather than holding it constant. We select the window $N = 10$ from a sweep over $N \in \{1, 5, 10, 25, 50, 100\}$, fix the target violation rate at 1%, sweep $\alpha \in \{1.1, 1.25, 1.5, 1.75, 2\}$ and $\beta \in \{0.25, 0.5, 0.7, 0.9\}$, and report the best pair ($\alpha = 1.25$, $\beta = 0.5$).

Table 14 reports raw throughput (all completions) alongside goodput (SLO-compliant completions only) on the main trace. The fixed caps and IRP-EAGER fall into the throughput trap at increasing depth. MIMD partially avoids it: throttling width when violations accumulate lifts attainment to 79%, the best among branch-admitting baselines, but a throughput–goodput gap persists ($1.44\times$ vs. $1.14\times$). PACE escapes the trap: 99% of its $1.79\times$ throughput is SLO-compliant, yielding $1.77\times$ goodput.

Table 14: Throughput, goodput, and attainment on the main trace (Qwen3-32B, PDR=50%, SLO=50 ms). Throughput and goodput are normalized to IRP-OFF.

Policy	Throughput	Goodput	Attainment
IRP-OFF	$1.00\times$	$1.00\times$	100%
IRP-C2	$1.56\times$	$1.11\times$	71%
IRP-C5	$2.19\times$	$1.18\times$	54%
IRP-EAGER	$2.54\times$	$1.20\times$	48%
MIMD ($\alpha = 1.25$, $\beta = 0.5$)	$1.44\times$	$1.14\times$	79%
PACE ($\rho = 0.8$)	$1.79\times$	$1.77\times$	99%

The residual gap reflects two structural limitations of any purely reactive controller with a global width. First, the window imposes a detection lag: W is held constant for N steps regardless of ongoing violations, so a sudden load spike can cause up to N consecutive violating steps before any adjustment; after a decrease, W regrows one multiplicative step per window regardless of how much slack has actually rebuilt, so it can trail the recovery or, as under moderate load (§4.2), overshoot it. Second, a single global W cannot distinguish requests close to their deadline from those with ample slack. PACE addresses both by predicting step latency before each step and calibrating admission against the slack budget B_t, which is set by the most urgent request in the batch (§3.3).

E.9 Online adaptation of the slack fraction

Table 1 sweeps ρ statically. For deployments whose load characteristics shift over time, ρ can also be adapted online. Algorithm 2 is one simple approach: an outer loop that observes rolling SLO attainment every T_{adapt} minutes and adjusts ρ by a fixed step δ. It operates on a timescale of minutes, decoupled from the per-step planner, and a tolerance band ε above the attainment target A^* (`A_target`) prevents oscillation when attainment hovers near the target.

Algorithm 2 Online Adaptation of ρ

```
def AdaptRho(ρ, A_obs, A_target, δ, ε, ρ_min, ρ_max):
    #called every T_adapt minutes with the attainment A_obs of that window
    if A_obs < A_target:                #below target: tighten
        return max(ρ_min, ρ - δ)
    if A_obs > A_target + ε:            #comfortably above: relax
        return min(ρ_max, ρ + δ)
    return ρ                            #within tolerance band: hold
```

To verify convergence, we initialize Algorithm 2 at the aggressive end, $\rho = 1.0$, with $A^* = 0.99$, $\delta = 0.1$, $\varepsilon = 0.005$, $[\rho_{\min}, \rho_{\max}] = [0.2, 1.0]$, and $T_{\text{adapt}} = 5$ min. The controller converges to $\rho = 0.8$ within two adaptation windows (Table 15), recovering attainment from 91.3% to 99% and matching the static $\rho = 0.8$ result. Gradient-based controllers, PID regulators, or Bayesian optimization could improve convergence speed or stability under non-stationary load; we leave these to future work.

E.10 Batch-size sensitivity

Table 16 sweeps batch size on Qwen3-32B (PDR=50%, SLO=50 ms) over the range validated for the latency predictor (Table 3).

Table 15: Convergence of Algorithm 2 from $\rho = 1.0$.

Elapsed time	ρ	Attainment	Goodput / IRP-OFF
0 min	1.0	91.3%	1.45×
5 min	0.9	94.7%	1.61×
10 min	0.8	99.0%	1.77×
15 min	0.8	99.0%	1.77×

Table 16: Batch-size sensitivity. Goodput normalized to IRP-OFF (which attains 100% at every batch size), with SLO attainment in parentheses.

Batch size	IRP-C2	IRP-C5	MIMD	IRP-EAGER	PACE
1	1.09× (100%)	1.14× (100%)	1.18× (100%)	1.24× (100%)	1.24× (100%)
4	1.12× (100%)	1.20× (100%)	1.22× (100%)	1.24× (100%)	1.24× (100%)
8	1.16× (100%)	1.21× (100%)	1.23× (100%)	1.24× (100%)	1.25× (100%)
16	1.18× (100%)	1.21× (99%)	1.22× (100%)	1.24× (99%)	1.27× (100%)
32	1.16× (97%)	1.20× (92%)	1.19× (98%)	1.23× (91%)	1.32× (100%)
64	1.14× (86%)	1.18× (77%)	1.17× (91%)	1.21× (74%)	1.45× (99%)
128	1.12× (76%)	1.15× (62%)	1.15× (84%)	1.17× (56%)	1.62× (99%)
192	1.10× (70%)	1.12× (52%)	1.14× (78%)	1.08× (47%)	1.68× (99%)
256	1.09× (64%)	1.10× (47%)	1.13× (72%)	0.99× (43%)	1.72× (99%)
384	1.07× (57%)	1.07× (42%)	1.12× (66%)	0.91× (39%)	1.73× (99%)
512	1.05× (53%)	1.05× (37%)	1.10× (62%)	0.88× (37%)	1.74× (99%)

At batch size 1, there is no inter-request externality: PACE admits every ready branch and matches IRP-EAGER (1.24×, 100% attainment). MIMD comes closest among the remaining policies (1.18×): with no violations, its width grows toward the maximum, but the windowed increase leaves it short. The fixed caps bind regardless of load (1.14× for IRP-C5, 1.09× for IRP-C2). As batch size grows, externalities compound (§2.3) and every baseline sacrifices attainment, goodput, or both. IRP-EAGER's uncapped admission pushes step latency past the target, collapsing to 0.88× goodput and 37% attainment at batch size 512; the fixed caps degrade more gradually but converge near 1.05× (37% attainment for IRP-C5, 53% for IRP-C2); and MIMD, the best branch-admitting baseline on attainment at 62%, settles at 1.10×. PACE alone sacrifices neither: a larger working set exposes more branch candidates per step, and spending the slack budget precisely converts them into monotonically rising goodput (1.24× to 1.74×) at ≥99% attainment throughout.

E.11 Prefill interaction and multi-instance deployment

TTFT under disaggregation. IRP is purely a decode-phase technique: prefill processes all input tokens in parallel, and no branch participates in it. Under the disaggregated deployment used throughout (Appendix D), TTFT is determined entirely by the prefill instance and is unchanged by branch admission under every policy. Disaggregation and PACE are complementary: disaggregation isolates prefill from decode pressure, while PACE maximizes branch parallelism within the decode SLO.

TTFT under co-location. We also evaluate a co-located deployment with chunked prefill on the same Qwen3-32B setup, with a 1000 ms TTFT target. We use SGLang's chunked-prefill scheduler unmodified; following Sarathi-Serve [33], it prioritizes decode tokens and fills the remainder of each step's token budget with a prefill chunk. To the scheduler, every branch token is a decode token. Under the fixed caps and IRP-EAGER, the entire admitted width therefore claims decode priority and narrows the prefill chunk, so prompts need more steps to finish prefilling and TTFT grows (Table 17). The effect scales with aggressiveness: IRP-C2 adds 118 ms at P50 and 190 ms at P99, IRP-C5 adds 291 ms and 826 ms, and IRP-EAGER more than doubles TTFT at both percentiles. PACE is the only policy that splits decode into two classes. Tokens in the protected composition S_0, which guarantees single-token progress for every active request (§3.3), keep decode priority; opportunistic branches are admitted only from the slack that remains after the prefill chunk. Under prefill pressure, that slack collapses and admission pauses, resuming once the prefill queue drains. PACE's TTFT consequently

tracks IRP-OFF to within 1 ms at P50 and 6 ms at P99. All policies run against the same unmodified scheduler, so these differences come from branch admission alone.

Table 17: TTFT in a co-located deployment with chunked prefill (Qwen3-32B, TTFT target 1000 ms).

Policy	TTFT P50 (ms)	TTFT P99 (ms)
IRP-OFF	520	987
IRP-C2	638	1177
IRP-C5	811	1813
IRP-EAGER	1062	2528
PACE ($\rho = 0.8$)	521	993

Multi-instance deployment. In a multi-instance cluster, one might expect to route a request's branches to different instances. For IRP, however, keeping a request's branches on one instance is a structural consequence of the technique rather than a design choice. Branches share their parent's prefix KV blocks, which under radix-tree allocation are resident on the instance serving the parent. Routing a branch elsewhere would require either recomputing the shared prefix on the remote instance, which adds compute proportional to prefix length, or transferring the prefix KV blocks over the network, which adds latency that grows with context size. Either way, the core appeal of IRP, branches sharing an already-resident prefix at no additional compute or transfer cost, is lost. An IRP-aware serving system therefore co-locates a request's branches by construction. Because no request's decode spans multiple instances, each decode instance runs an independent PACE controller on its local batch, with no cross-instance coordination.

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 PACE 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 from those allocations. 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 PACE'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 PACE 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. PACE 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 intra-request parallelism 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 [40] 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 [41] 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 [46], along with variants such as Medusa [47] and EAGLE-3 [48], 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 PACE's motivation of adaptive control under varying load. TurboSpec [49] adjusts speculation length via a goodput-based feedback loop, AdaSpec [50] and AdaServe [43] tune speculative strategy for SLO attainment, and Nightjar [51] 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. PACE 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 PACE governing branch width.

LLM serving schedulers. The scheduling problem PACE 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. Chunked prefill [33] and prefill-decode disaggregation [27–29] manage interference between prefill and decode; PACE is complementary, regulating interference within the decode phase (Appendix E.11). FairBatching [42] is PACE'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. PACE'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 (PACE 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 PACE faces a combinatorial allocation across multiple requests with varying branch counts and utilities. Andes [52] 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, PACE decides *how many branches* a given request gets; the two concerns compose. NexusSched [53] 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 PACE's offline-fitted linear model. vLLM [18] handles memory pressure via swap-based request-level preemption, which is expensive to reverse. PACE'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 negligible cost. A simpler adaptive mechanism, a reactive controller that adjusts a global width by multiplicative increase and multiplicative decrease (MIMD), serves as a baseline throughout (§2.2, Appendix E.8). It adapts to load but lacks two properties PACE provides: it is backward-looking, reacting to violations already incurred rather than predicting latency before committing to a step composition; and it has no notion of per-request slack, applying one width to every request regardless of how close each is to its deadline. PACE's slack budget takes the minimum over all requests' residual slack, ensuring width is safe for the most urgent request in the batch.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction claim that (1) eager branch admission creates a branch externality that degrades co-batched requests, and neither fixed caps nor reactive width control resolves it (§2); (2) branch-level scheduling decouples compute from memory (§3.1, §3.5); and (3) PACE improves goodput by $1.77\times$ over IRP-OFF and $1.48\times$ over IRP-EAGER at 99% SLO attainment on Qwen3-32B (§4.2), with comparable gains (1.72–$2.15\times$) on five additional dense and MoE models (§4.4, Appendices E.5 and E.7). All claims are supported by the corresponding sections.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: §6 discusses three limitations: the branch-independence assumption (excluding inter-branch attention methods), the linear latency predictor's accuracy at extreme batch sizes, and per-instance scope, with multi-instance deployment discussed in Appendix E.11. The sensitivity to the slack fraction ρ is characterized in §4.3 and Appendix E.9, and Appendix E.1 shows that the benefit narrows as fewer requests are decomposable, vanishing at PDR=5%.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: The schedule-invariance lemma (§3.1) states its assumptions (visibility rule, valid schedule definition) and provides a proof sketch in the main text with a full proof in Appendix A. The NP-hardness result (Appendix B) is shown via a proof sketch that reduces from 0–1 knapsack, assuming latency is additive in the marginal costs.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: §4.1 specifies the model, hardware, serving engine, deployment, workload composition, SLO target, and baselines. Appendix D provides extended details, including hardware sizing, branch fanout and context-length distributions, trace characteristics, the SLO methodology, metric definitions, and the configuration of every policy, including MIMD. Appendix E.7 specifies the MoE models and their fine-tuning, and Appendix E specifies the settings of each additional experiment. Algorithm 1 fully specifies the planner, and Algorithm 2 the online adaptation of ρ.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The PACE implementation is released at `https://github.com/Swapnil-Gandhi/PACE` under the Apache 2.0 license, with instructions for installation and for serving under each admission policy. All datasets and the Azure LLM inference trace are publicly available, and Appendix D describes how the workload is constructed from them.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: PACE itself has no training phase. Its hyperparameters ($\rho = 0.8$, linear utility $u_r(a_r) = a_r$) are specified in §4.1, and §4.3 and Appendix E.9 show how ρ was chosen. The latency predictor's fitting procedure is described in Appendix C, MIMD's hyperparameter sweep in Appendix E.8, and the supervised fine-tuning that adapts the MoE models to parallel generation (data, prompts, mixing schedule, and epochs) in Appendix E.7.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Experiments use a single deterministic trace replayed identically across all policies, ensuring a controlled comparison. Variability across runs is negligible because the workload, model, and hardware are held constant. We report time-series results across a 10-hour trace, which capture temporal variability directly, rather than only aggregate statistics, and add sensitivity sweeps over workload composition, SLO target, batch size, and ρ (§4.3, Appendix E).

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: §4.1 specifies 8×A100-80GB GPUs with TP=8, which also serve Qwen2.5-72B (Appendix E.5). Appendix D provides memory utilization details, and Appendix E.7 specifies the hardware for the MoE models (8×A100 or 8×H200) and their fine-tuning cost (8–12 hours on 8 H200 GPUs). Each trace-driven experiment replays the 10-hour trace; wall-clock time is approximately equal to trace duration per policy.

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: This work addresses LLM serving efficiency and does not involve human subjects, personal data, or dual-use concerns. All datasets used are publicly available and appropriately cited.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: PACE is a scheduling mechanism for LLM serving infrastructure. It improves resource efficiency without affecting model capabilities, training data, or output content. We do not foresee direct negative societal impacts from a decode-step scheduling algorithm.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: PACE is a scheduling algorithm, not a model or dataset, and we release only its implementation. It does not introduce new capabilities for language generation and poses no risk of misuse beyond that of the underlying serving infrastructure.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All datasets (ShareGPT Vicuna, RAG-12K, Math-220K, s1K-1.1), the models (Qwen3-32B, Qwen2.5-72B, GPT-OSS-120B, DeepSeek-V4-Flash, Llama 4 Maverick, DeepSeek-R1), the serving engine (SGLang, extended through the Multiverse Engine), the IRP frontends (Multiverse, SPRINT), and the Azure LLM inference trace are cited with their original publications or model cards. URLs for RAG-12K, Math-220K, and the Azure trace are provided in the references. All assets are used under their published licenses.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We release the PACE implementation at `https://github.com/Swapnil-Gandhi/PACE` under the Apache 2.0 license. It extends the Multiverse Engine, an SGLang fork; the README documents installation, configuration flags, and usage, and PACE is opt-in, reproducing the stock engine when disabled. The paper does not release new datasets or model weights.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This work does not involve crowdsourcing or human subjects.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This work does not involve human subjects research.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs are used as the workload (the six evaluated models serve requests, and four MoE models are fine-tuned to produce parallel generation), not as a methodological component. PACE is a model-agnostic scheduling algorithm 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.