CoServe: Preserving Model Choice under GPU Memory Pressure

Anonymous authors
Paper under double-blind review

Abstract

Multi-model LLM services often offer more models than GPU memory can hold simultaneously. Selecting one model for a request can trigger loading and eviction, even when an already resident alternative could answer it. We present CoServe, which preserves predicted acceptable model alternatives and coordinates request routing with model residency. For each proposed resident set, CoServe recomputes request assignments and compares estimated batch-service costs, waiting penalties, and activation costs. Our structural analysis shows that committing requests to single model identifiers can discard information needed to select a coverage-optimal resident set. On a single B200 deployment with eleven configured model footprints totaling $3.05\times$ nominal GPU memory, CoServe increases the fraction of requests completed within 100 seconds by at least 16.1 and 3.4 percentage points over CARROT on a Prism-based backend in two arrival realizations. Mean arrival-to-completion latency decreases by 14–17% in the first realization, while tail latency remains higher in both. Numerical ranges reflect timestamp reconstruction uncertainty, not confidence intervals. These results support preserving model alternatives when coordinating routing and GPU residency, with benefits that depend on the workload and latency objective.

1 Introduction

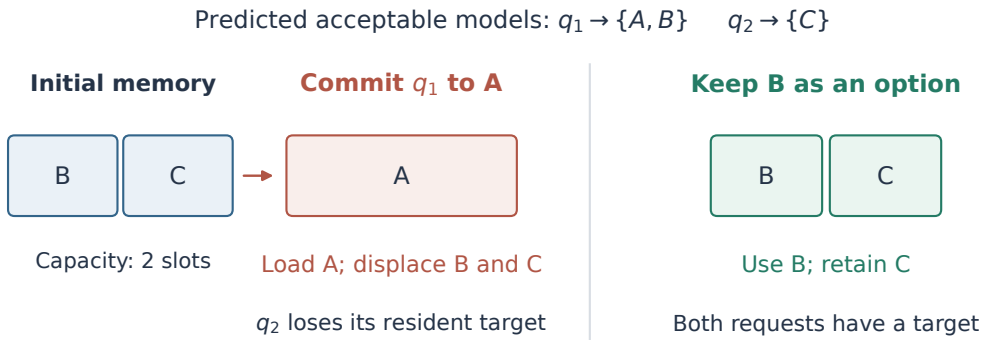
A multi-LLM service selects models to meet different requests' needs, but its model pool may exceed GPU memory. Selecting an absent model then requires loading it, which can evict a model needed by another request. If a router commits each request to one model first, the memory controller cannot tell which requests could instead use a resident alternative. The service loses information needed to decide which models to keep in memory, potentially making requests wait for avoidable loads and displacing useful models.

CoServe preserves each request's predicted acceptable models and **recomputes request assignments for every proposed set of GPU-resident models**. This lets the controller evaluate the demand its memory decision would create, rather than treating previous routing decisions as fixed. Keeping a candidate list is only the interface: the controller uses those alternatives to compare the batch-service costs, waiting penalties, and activation costs of different memory configurations. Requests continue to route using observed backend availability while residency changes take effect.

Figure 1 illustrates the lost information. Resident models B and C fit together, while absent model A needs their combined space. Request q_1 has predicted acceptable models A and B; request q_2 has only C. Committing q_1 to A can trigger a load that displaces C. Using B instead retains C and leaves both requests with a resident target. Once the router emits only A, the memory controller cannot recover this substitution from the committed identifier. Our deployment makes such decisions consequential: eleven configured model footprints total 586 GB before KV pools, against 192 GB of nominal HBM, and the largest models have observed median activation intervals of 44.1–50.4 s.

Learned routers optimize model selection using capability and cost estimates (Ong et al., 2025; Somerstep et al., 2025); modern serving systems share scarce GPU memory across models (Duan et al., 2024; Yu et al., 2026). Composing these strong components does not by itself preserve request alternatives at their interface. MuxServe highlights unequal model popularity (Duan et al., 2024), Prism reports changing groups of active models (Yu et al., 2026), and ServerlessLLM reduces inference startup costs (Fu et al., 2024). These observations motivate memory management and faster

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



Memory pressure (configuration)	Activation delay (existing telemetry)
11 model footprints: 586 GB	Math-72B / Qwen-72B: median 44.1 s
Nominal GPU HBM: 192 GB	R1-70B: median 50.4 s

Figure 1: **A local model choice can displace another request's only target.** A occupies both memory slots; B and C fit together. Preserving B as an alternative for q_1 avoids loading A and retains C for q_2. The schematic is illustrative. The evidence strip reports configured footprints and observed activation medians; Appendix B gives measurement details.

loading. CoServe addresses the complementary question of how request substitutions should change the working set itself.

The central principle is that **a resident set must be evaluated through the request assignments it enables**. Three difficulties determine how CoServe applies it:

Overlapping alternatives. Two popular models may serve the same requests, so adding their candidate counts can double-count demand. CoServe evaluates which distinct requests each proposed model set can serve, retaining each request's alternatives throughout planning.

Unequal serving costs. Assigning more requests to an available model can overload its queue. Coverage alone therefore cannot determine which set offers better service. For each proposal, CoServe recomputes model assignments and batch positions, then estimates service costs using model-specific batch widths and decode rates. Requests left without a candidate receive age-sensitive waiting penalties.

Slow changes in availability. Loading takes time and can displace another model. CoServe includes activation costs when comparing proposals and submits the selected set as an advisory wishlist. The backend checks memory and transitions; routing continues against observed resident and loading models. This feedback separates a proposed configuration from what the backend can currently serve.

These choices go beyond selecting an already loaded model: the planner must also decide when an absent model is worth admitting and which requests a switch would leave uncovered. Relative to joint deployment/routing and adapter caching/routing (Kasnavieh et al., 2026; Li & Li, 2026), CoServe targets repeated full-model activation and eviction with heterogeneous footprints, queue-dependent service, and pending requests with different alternatives. The contribution is this coordination under changing residency, not a replacement quality predictor or the first use of routing inside a configuration search.

Our contributions are threefold. First, we formalize the information lost by single-model commitment and the coverage structure revealed by candidate sets. These structural results motivate the interface; they do not establish latency optimality for the practical controller. Second, we implement residency planning that recomputes assignments and batch-service estimates for every proposed working set. Third, saved deployment records show higher 100-second deadline attainment than CARROT on a Prism-based backend in two arrival realizations, by at least 16.1 and 3.4 percentage points. Mean

latency improves by 14–17% in the first realization; tails remain higher in both. A separate same-build ablation tests the effect of recomputing assignments during residency scoring.

2 THE STRUCTURE OF ROUTING–RESIDENCY COUPLING

Setting and decisions. We consider services authorized to select a model for each query; requests that require a named model have a singleton choice set. Let $\mathcal{M}$ be the model pool and $s_t = (R_t, L_t, b_t, Q_t)$ summarize resident models, loading models, observed per-model load, and pending queries. Predicted quality $\widehat{Q}(q, m)$ defines

$$C(q) = \{m \in \mathcal{M} : \widehat{Q}(q, m) \geq \tau\}, \tag{1}$$

with a highest-score singleton fallback if the threshold set is empty. These are predicted acceptable alternatives; the fallback may fall below τ. The decisions are a destination for each dispatched query and a target resident set. The backend's memory and transition checks determine whether and when that target can be realized.

Why model counts lose decision-relevant information. For a snapshot, temporarily ignore execution and transition costs. The demand a set can cover is

$$H_t(R) = \sum_{q \in Q_t} \omega_q \, \mathbf{1}\{C(q) \cap R \neq \varnothing\}, \qquad \omega_q \geq 0. \tag{2}$$

Here ω_q is a request weight used for this structural analysis, not a measured quality score. If routing commits to $g(q) \in C(q)$ first, the backend instead sees $n^g_m = \sum_q \omega_q \mathbf{1}\{g(q) = m\}$. Those counts describe one assignment, not the alternatives available under another resident set.

Proposition 1 (Information lost by committing routes)**.** *There exist two workloads with identical committed model identifiers, request weights, model footprints, and memory budget, but different unique maximizers of $H_t(R)$. Therefore a residency controller observing only that projection cannot always recover a coverage-optimal resident set.*

The construction uses the three-model example in Figure 1; Appendix C gives the proof. It establishes a limitation of the interface, rather than a claim that every state-aware router fails.

The loss can grow with the model pool. Proposition 2 strengthens this separation: for every $k \geq 2$ and every possibly randomized controller restricted to committed-route observations, there is a k-query, $2k$-model instance with one resident slot for which

$$\mathbb{E}[H_t(R_{\text{committed}})] \leq \frac{1}{k} \max_{|R| \leq 1} H_t(R). \tag{3}$$

Each query commits to its own model while all share one hidden alternative. Identical committed routes hide which alternative is shared. Candidate sets reveal it and permit full coverage. Thus even an exact downstream optimizer cannot guarantee a pool-independent coverage factor through this restricted interface. The claim concerns information available to the controller, not the approximation quality of CoServe's search.

Proposition 3 additionally quantifies the value of observing serving state before choosing a model. These results motivate retaining alternatives; evaluating their shared-queue consequences requires the coupled controller below.

Why preserving alternatives does not make selection trivial. Even with all candidate sets known, maximizing (2) under additive model-memory costs is the classical budgeted maximum coverage problem (Khuller et al., 1999). Each model covers the queries for which it is a candidate; these sets overlap. The marginal value of loading a model depends on the models retained alongside it. Appendix C gives the reduction and the diminishing-returns identity. This worst-case hardness concerns growing pools; our eleven-model deployment permits bounded local search. The practical difficulty remains evaluating a set correctly.

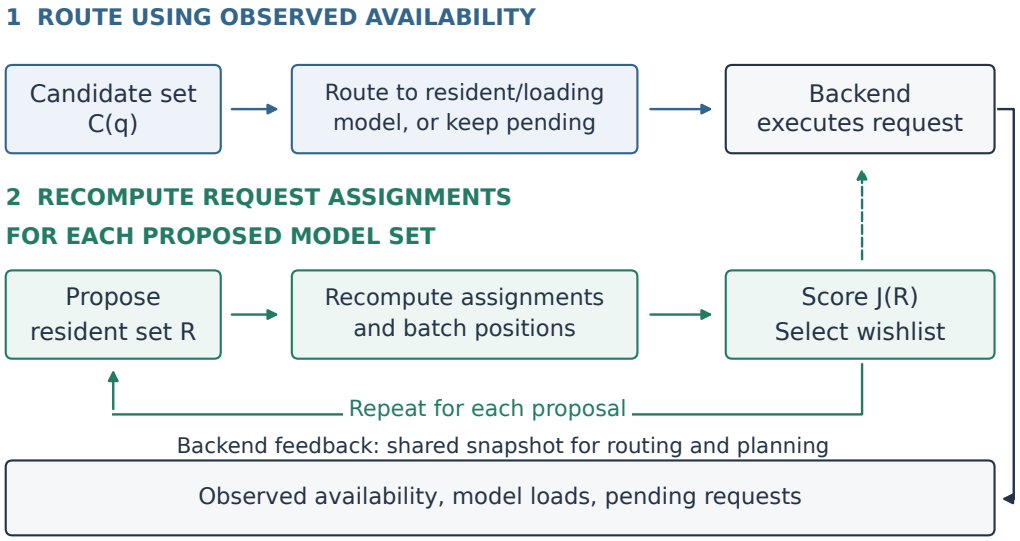


Figure 2: **Preserve candidates across routing and residency planning.** The per-request path uses observed resident and loading models. The periodic path reassigns pending/projected queries under each proposed resident set, recomputes batch positions, and scores service, age, and activation cost. The dashed arrow submits an advisory wishlist; the backend validates transitions. Backend telemetry updates the serving snapshot used by both paths.

From coverage to timely service. Coverage is necessary to dispatch to a predicted suitable resident model, but it does not determine delay. Under a candidate-conditioned route $\rho(q; R, s_t)$, the induced assignment count is

$$n_m(R) = \sum_{q \in Q_t} \mathbf{1}\{\rho(q; R, s_t) = m\}. \tag{4}$$

Changing R changes these counts, batch waves, and uncovered requests. Loading also consumes time and can displace other models. The service must balance useful coverage, congestion, and transition costs, while quality predictions restrict substitutions. CoServe addresses this interaction with a candidate-conditioned cost surrogate; it does not claim to solve the global dynamic optimum. User-visible latency is measured from arrival through completion, including admission and availability waiting. The measured deployment scope remains a single GPU.

3 CoServe

The analysis identifies two requirements: preserve query-level alternatives, then evaluate the service those alternatives enable under each proposed residency. CoServe implements these requirements through candidate-aware routing and residency planning (Figure 2). Shared telemetry connects their decisions at different timescales.

3.1 Candidate-aware routing

The historical deployment builds $C(q)$ from CARROT quality predictions with $\tau = 0.7$ in Equation (1). It reuses this quality estimator (Somerstep et al., 2025). The contribution of the interface is to retain multiple destinations until their serving costs can be considered.

For an available set A, the implemented rule is

$$\rho(q; A, s) = \begin{cases} \arg \operatorname*{lex\,min}_{m \in C(q) \cap A} \big(\kappa_m, b_m(s), -\widehat{Q}(q, m)\big), & C(q) \cap A \neq \varnothing, \\ \arg \operatorname*{lex\,min}_{m \in C(q)} (\kappa_m, v_m), & C(q) \cap A = \varnothing. \end{cases} \tag{5}$$

Here $b_m(s)$ is observed waiting-plus-running load, v_m is the router's model-size estimate, and κ_m is a deployment-specific slow-model tier: 1 for `r1-70b`, 0 otherwise. Lexicographic selection prefers a lower slow tier, then lower load, then higher predicted quality. This explicit heuristic is the route resolver used by the evaluated implementation.

At runtime, $A = R_s \cup L_s$: requests may target resident or loading models. If this set contains no candidate, dispatch can be deferred and the query remains visible to planning. During planning, A is replaced by a proposed resident set R. The same resolver then exposes the assignments that proposal would induce. If no candidate is covered, the second branch names an absent target and scoring applies an uncovered-request penalty. Actual execution and counterfactual evaluation therefore use the same candidate interface, with different availability sets.

3.2 Recompute assignments for each residency proposal

Candidate sets and memory feasibility. The planner considers the current resident set, curated configurations, and local additions, removals, and one-for-one swaps. The addition/swap filter accounts for configured model footprints, reserved memory pools, and activation headroom. Curated configurations are predefined and the current set remains a stay proposal; the backend revalidates transitions. The search compares a bounded neighborhood rather than optimizing all future schedules.

Service and uncovered demand. For every R, CoServe computes $m_R(q) = \rho(q; R, s_t)$ for pending queries Q_t and a short demand projection $\widehat{Q}_t$. It then recomputes assigned-query positions separately in these two lists. The default scoring branch is

$$c(q,R) = \begin{cases} \left(\left\lfloor \dfrac{p_R(q)-1}{B_m} \right\rfloor + 1 \right) \dfrac{\widehat{o}_q}{T_m}, & m = m_R(q) \in R, \\ \eta(1 + a_q/a_0), & m_R(q) \notin R. \end{cases} \tag{6}$$

$p_R(q)$ is the one-based position among queries assigned to m, B_m is profiled batch width, T_m is profiled decode rate, and $\widehat{o}_q$ is an output-length hint. The first branch estimates service in batch waves. The second penalizes leaving a query uncovered, increasingly with its nonnegative age a_q; the implementation uses $\eta = 600$ and $a_0 = 60$ seconds. It is a control penalty, not a calibrated completion-time prediction. Projected queries have age zero, and the replay uses fixed token hints.

This resolves a limitation of coverage alone. Two sets can cover the same queries yet assign them to models with different batch widths, service rates, and queue positions. Recomputing routes and positions allows the cost to distinguish those sets. Live queue counts guide the route resolver; the default batch-wave model approximates positions in the scored workload rather than simulating the entire backend queue.

Transition-aware choice. CoServe minimizes the combined surrogate

$$J_t(R) = w_q \, \mathcal{A}\{c(q,R) : q \in Q_t\} + w_f \, \mathcal{A}\{c(q,R) : q \in \widehat{Q}_t\} + \lambda_{\text{load}} L(R_t, R) + D_t(R), \tag{7}$$

where $\mathcal{A}$ is the run-configured sum, mean, or 95th percentile. L is the maximum profiled activation cost among added models, or zero if none are added. D_t penalizes prolonged residence of large models. The short demand projection uses past arrivals grouped by the replay's domain metadata. Its purpose is to inform imminent residency choices, not to supply future arrivals.

The dependence is explicit:

$$R \;\longrightarrow\; \{m_R(q), p_R(q)\}_q \;\longrightarrow\; J_t(R) \;\longrightarrow\; R^\star. \tag{8}$$

Holding model assignments fixed would remove the first step and treat routing-induced demand as given. This candidate-conditioned evaluation is the central algorithmic coupling.

Age and backend-queue urgency can restrict the search to sets containing an urgent absent model. The scarcity-weighted option favors older requests with fewer alternatives. Algorithm 1 summarizes the controller; Appendix B details urgency. These controls do not establish a starvation guarantee. For K proposed sets, N pending/projected requests, maximum candidate size d, and M models, sum/mean scoring takes $O(K(Nd + M))$ operations after enumeration.

Algorithm 1 CoServe: route for observed availability; plan for candidate demand

Require: Candidate sets $C(q)$, quality scores, profiles, backend state

procedure DISPATCHATTEMPT(q, s)
  if $C(q) \cap (R_s \cup L_s) = \varnothing$ **then**
    Keep q pending and visible to residency planning
  else
    Dispatch $(q, \rho(q; R_s \cup L_s, s))$ to the backend
  end if
end procedure
procedure PLANTICK(s_t)
  if a load is in flight or no pending/projected demand exists **then**
    return
  end if
  Form $Q_t, \widehat{Q}_t$ from pending queries and past arrivals
  $\mathcal{N}_t \leftarrow$ current set, curated seeds, and single removals
  Add memory-filtered single additions and swaps to $\mathcal{N}_t$
  $\mathcal{E}_t \leftarrow$ URGENCYFILTER$(\mathcal{N}_t, Q_t, s_t)$
  for each $R \in \mathcal{E}_t$ **do**
    $m_R(q) \leftarrow \rho(q; R, s_t)$ for each $q \in Q_t \cup \widehat{Q}_t$
    Compute assigned-query positions separately in Q_t and $\widehat{Q}_t$
    Evaluate $J_t(R)$ using Equation (7)
  end for
  $R^\star \leftarrow \arg\min_{R \in \mathcal{E}_t} J_t(R)$
  if dwell and drain guards permit the proposal **then**
    Submit $R^\star$ as an advisory wishlist; backend validates transition
  end if
end procedure

3.3 REALIZING THE DECISIONS IN A SERVING BACKEND

The evaluated controller polls telemetry nominally once per second and plans every 15 s. Loading, dwell, and drain guards regulate wishlist submission to the Prism-based backend (Yu et al., 2026); requests continue to route against observed availability and load. The backend executes batches and checks activation and eviction, and its actual state feeds the next decisions.

A wishlist does not make its target immediately available. Telemetry can be stale and transitions can be deferred, so dispatch and control are coordinated through feedback rather than one atomic optimization step. Admission and availability waiting remain part of arrival-to-completion latency. These execution semantics connect the formal choice to the service users actually receive.

4 EVALUATION

We test whether preserving model alternatives improves timely completion when routing and residency share a constrained GPU. The primary comparison composes learned routers with a capable residency-aware backend, so any gain must survive the backend's existing memory management. We combine historical deployment records with a new controlled ablation of proposal-conditioned scoring (Appendix D).

4.1 SETUP AND MEASUREMENT

Deployment and workload. The historical deployment uses one NVIDIA B200 with 192 GB nominal HBM and an eleven-model pool spanning 0.5B–72B parameters. It includes Qwen general, math, and code models, Llama, Mistral, and DeepSeek-R1 distillations; Appendix B specifies the pool and profiles. Configured model footprints total 586 GB before KV reservations.

The saved workload contains 5,000 rows, balanced across code, math, reasoning, and general domains, with a 3,500/500/1,000 row-stratified training/validation/test split. Its actual sources are HumanEval

Table 1: Existing runs on the same backend family. All latency columns are seconds. Ranges are arrival-metric bounds, rounded outward; HTTP is post-dispatch. Proxy is offline routing correctness with labelled n. Each row has 1,000 issued requests.

System	Proxy (n)	HTTP	Arrival mean	Arrival p95	SLO (%)	Fails
CoServe A	0.756 (980)	45.7	107.4–111.0	319.7–323.3	59.6–60.4	0
CARROT A	0.746 (1000)	128.7	129.0	315.3	43.5	0
RouteLLM A	0.741 (1000)	134.5	134.5	317.4	42.7	0
Round-robin A	0.444 (1000)	173.8	174.5	386.5	34.2	0
Static single A	0.718 (883)	69.3	69.3	172.8	72.9	0
CoServe B	0.743 (930)	64.6	136.3–141.8	412.9–418.4	51.5–53.3	1
CARROT B	0.746 (1000)	139.5	139.5	350.6	48.1	0

and MBPP variants (Chen et al., 2021; Austin et al., 2021), MATH and GSM-Hard (Hendrycks et al., 2021b; Gao et al., 2023), BBH (Suzgun et al., 2023), and MMLU/MMLU-Pro (Hendrycks et al., 2021a; Wang et al., 2024). The saved split has 76 test prompts that exactly match training prompts; variants were not grouped by source problem. Thus the quality results characterize this workload and do not establish generalization to unseen problems.

Arrivals follow a constructed session-clustered process, using an Azure-style rate-envelope shape rather than replaying production request contents. Realization A contains 64 sessions, with about 80% domain/difficulty purity and roughly 6.9 requests/min at the evaluated $0.1\times$ rate. Realization B redraws session composition and arrival times for the same 1,000 queries. Within each main comparison, offered schedules match within 1 ms. Generation lengths vary with model and request; token throughput is therefore not treated as equivalent useful work.

Baselines and variants. Our principal baseline is **CARROT + Prism**: CARROT is a cost-aware learned router with statistical rate-optimality results (Somerstep et al., 2025), and Prism is a modern multi-LLM serving system with GPU memory ballooning (Yu et al., 2026). RouteLLM (Ong et al., 2025) supplies a preference-trained routing baseline on the same backend family; round-robin supplies a nonlearned reference. CoServe also uses the Prism-based backend, isolating the architectural question from a comparison against an unaware execution engine. A static baseline pins `r1-70b` and avoids model switching. These comparisons hold the backend family fixed, although historical builds are not byte-identical across all runs. CARROT uses the recorded $\mu = 0.1$ configuration. The available variants remove forecasting, truncate candidates to top-1, or route without residency awareness; a lower-penalty full-system run provides an additional reference. Exact provenance and the limits of component attribution are given below. We do not use older, different-backend comparisons to establish the central claim.

Metrics and timing boundary. Table 1 separates HTTP-dispatch latency from arrival-to-completion latency. HTTP timing starts after client admission and concurrency waiting, so it does not measure full user delay. Baseline records retain relative send times and permit arrival reconstruction. CoServe records omit the exact replay epoch. We bound it using a server snapshot received before replay and the earliest send-minus-schedule timestamp, producing two-sided latency intervals where startup logs survive (Appendix A). Intervals reflect timestamp uncertainty, not confidence intervals.

For Table 1, latency means and p95 use successful requests; SLO attainment counts successful completion within 100 s over all 1,000 issued requests, including failures in the denominator. The quality proxy is an offline correctness lookup for the selected query/model pair, with its labelled denominator shown. It is not correctness scored on the actual generated response, and is reported separately from service success.

4.2 Timely completion and the static tradeoff

Deadline attainment improves across both arrival realizations. On A, CoServe completes 59.6–60.4% of issued requests within 100 s, compared with CARROT's 43.5% and RouteLLM's 42.7%. Relative to CARROT, the gain is 16.1–16.9 percentage points: *at least 161 additional timely completions* in the recorded run. On B, the gain remains positive across the entire timestamp-uncertainty

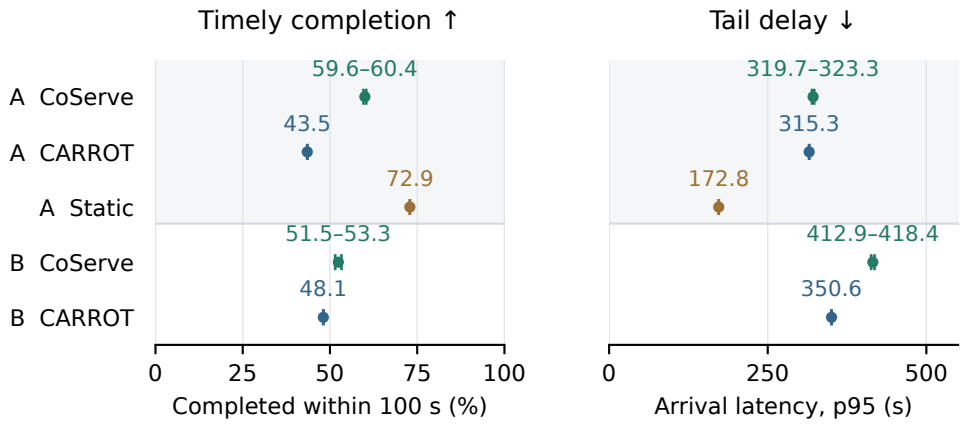


Figure 3: **More timely completions, with a tail-delay tradeoff.** CoServe improves 100 s deadline attainment over CARROT on both arrival realizations; p95 latency is higher. Static residency is a strong latency alternative on A. Points and horizontal intervals show reconstructed values and replay-epoch bounds, with endpoints printed explicitly; these are not confidence intervals. A and B denote arrival realizations. Full completion distributions appear in Appendix A, Figure 4.

interval: 51.5–53.3% versus 48.1%, or 3.4–5.2 percentage points. The conservative improvement is therefore positive on both offered schedules, including failed requests in the denominator.

The mean benefit depends on the arrival realization. On A, CoServe's arrival mean is 107.4–111.0 s versus CARROT's 129.0 s, a 14–17% reduction. On B, the 136.3–141.8 s interval overlaps CARROT's 139.5 s. Deadline attainment is consequently the supported cross-realization result. Arrival time is essential to this result: on A, CoServe's post-dispatch mean is 45.7 s, but pre-dispatch admission/concurrency waiting adds at least 61.7 s. This waiting remains in every arrival-based result; moving a queue upstream does not remove user delay.

The tail exposes a cost of sharing the changing model pool. CoServe's arrival p95 is at least 319.7 s on A and 412.9 s on B, exceeding CARROT's 315.3 and 350.6 s (Figure 3). A higher fraction of timely completions can coexist with longer waits for the remaining requests. Age-sensitive controls do not eliminate this effect.

The static deployment is a strong alternative: 69.3 s mean, 172.8 s p95, and 72.9% SLO attainment on A. It outperforms CoServe on these latency metrics. Its offline quality proxy is 0.718 on 883 labelled rows, compared with CoServe's 0.756 on 980. Different missing-label patterns, prompt overlap, and the use of offline labels prevent interpreting this as a guaranteed live-response quality advantage. The practical result is a workload-specific tradeoff between model choices and serving delay, not uniform domination by orchestration.

Candidate lists expose substantial routing flexibility. In the saved CARROT candidate artifact associated by filename with the main run, 63.6% of requests have at least two candidates; the mean set size is 3.085. Excluding the 76 exact training-prompt matches leaves 61.5% with multiple candidates. Every saved set reconstructs from the saved quality scores using $\tau = 0.7$ and the singleton fallback. This is descriptive evidence of predicted substitutability, not a guarantee of candidate correctness or a new inference experiment. Appendix B records artifact provenance.

Controlled evidence for proposal-conditioned scoring. On a separate three-model B200 deployment, a same-build ablation freezes assignments only during residency scoring. CoServe lowers p95 by 9.2–16.1 s on three of four matched schedules; on the fourth, p95 rises by 1.7 s. All eight runs complete every request. Both policies already attain the 100 s deadline for 99.5–100% of requests; graded outputs do not establish a correct-and-timely advantage. Appendix D reports the complete

comparison and faster static reference. This isolates the scoring intervention in a reduced setting; historical variants remain diagnostic (Appendix B.1).

5 Positioning relative to prior work

From routing quality to realizable service. RouteLLM and CARROT establish learned model selection using quality and cost (Ong et al., 2025; Somerstep et al., 2025); HW-Router incorporates live hardware and queue signals (Kabir et al., 2026). CoServe carries predicted model alternatives into the decision that changes future availability. Its advance concerns the interaction with residency control.

What routing benchmarks expose. LLMRouterBench (Li et al., 2026) compares methods including RouterDC, EmbedLLM, GraphRouter, and Avengers-Pro, and documents both model complementarity and failures to recover rare correct experts. Its token/provider-based latency analysis motivates a service dimension. Our distinction is endogenous availability: selecting a model can change the latency of other requests through shared-memory eviction. Benchmark accuracy or provider-latency estimates do not measure that effect.

From serving a model stream to choosing its working set. MuxServe optimizes multi-model colocation and batch scheduling (Duan et al., 2024), ServerlessLLM reduces startup costs through loading, migration, and locality-aware scheduling (Fu et al., 2024), and Prism manages shared GPU memory dynamically (Yu et al., 2026). These mechanisms determine how model demand is realized. CoServe additionally exposes query-dependent substitutions that can change the demand itself. Its implementation builds on Prism, making comparisons on the same backend family essential to evaluating the combined controller.

Closest joint optimization approaches. RouterWise searches deployment setups and computes routing for each setup under an SLO (Kasnavieh et al., 2026). Thus, evaluating routing inside a configuration search is shared territory. Its formulation fits all deployed shards; CoServe targets repeated activation and eviction of a pool that cannot remain resident, carrying pending-query age and transition costs across decisions. POLAR jointly learns utility and controls LoRA-adapter caching/routing (Li & Li, 2026). Its adapter-count constraint and per-adapter paging penalties differ from our heterogeneous full-model footprints and assignment-dependent batch service. CoServe uses offline quality estimates and heuristic control; POLAR studies online learning guarantees. Our structural analysis uses classical coverage theory (Khuller et al., 1999) to explain the candidate interface; it does not introduce a new coverage approximation algorithm.

6 Discussion and conclusion

Model alternatives are a serving resource. Committing to one destination before residency planning can discard useful coverage information. CoServe preserves predicted alternatives and recomputes request assignments for each proposed working set, while a Prism-based backend retains execution and memory checks.

The historical single-GPU records show higher 100-second deadline attainment than CARROT on both tested arrival realizations and a mean-latency benefit on the first. A controlled reduced-pool ablation lowers p95 on three of four schedules, supporting the scoring mechanism without establishing a timely-correctness gain. Longer tails in the historical comparison and faster static baselines motivate better protection of scarce alternatives. Timing bounds, incomplete offline quality labels, and prompt overlap limit broader inference. The proofs explain the coverage information lost by early commitment; they do not guarantee latency optimality. Together, the interface, controller, and measured deadline gains support reconsidering request assignments when deciding which models to keep in GPU memory.

AI USE STATEMENT

We used AI coding assistants for implementation and debugging, preparation and analysis of saved research records, literature discovery, and manuscript drafting and editing. Assistance also included refining the conceptual framework and hypotheses, formulating mathematical claims and proof arguments, drafting proofs, providing methodological feedback, and interpreting results. Verification included code tests, checks against recorded measurements and source references, and examination of proof assumptions and constructions. The authors are responsible for verifying all AI-assisted content and for the final research decisions, claims, code, and results.

ETHICS STATEMENT

This work studies serving policies for existing language models using benchmark-derived requests. A residency policy can distribute delays unequally across query types, particularly when suitable models are scarce or large. We therefore report failures and tails and do not claim that age-based controls guarantee fairness. The method inherits the limitations and potential harms of the served models; offline routing labels do not certify individual responses, and live-output grading in the controlled experiment has unresolved cases. Energy use and environmental impact were not measured.

REPRODUCIBILITY STATEMENT

Section 4.1 specifies the evaluated setting and metrics. Appendix A derives the timing intervals and identifies their assumptions; Appendix B provides the model pool, controller settings, and variant provenance. Appendix C contains complete structural proofs. The manuscript source includes an exported request-level measurement table, run metadata, and a deterministic rebuilding script for the reported tables and CDFs. Appendix D documents the new ablation; its exported request timings and grades reproduce Table 4 using `tools/rebuild_controlled.py`. These artifacts reproduce the analysis without rerunning inference. They are an analysis package, not a claim that every historical serving binary has been reconstructed.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. MuxServe: Flexible spatial-temporal multiplexing for multiple LLM serving. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11905–11917. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/duan24a.html.

Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-Latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 135–153, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/fu.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10764–10799. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/gao23f.html.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021a. URL https://openreview.net/forum?id=d7KBjmI3GmQ.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021b. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/be83ab3ecd0db773eb2dc1b0a17836a1-Paper-round2.pdf.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Ahasan Kabir, Jiaqi Xue, Mengxin Zheng, and Qian Lou. Hw-router: Hardware-aware routing for scalable multi-llm serving. In *Proceedings of the 63rd Design Automation Conference (DAC)*, 2026.

Hossein Hosseini Kasnavieh, Gholamreza Haffari, Christopher Leckie, and Adel N Toosi. Routerwise: Joint resource allocation and routing for latency-aware multi-model llm serving. *arXiv preprint arXiv:2604.10907*, 2026.

Samir Khuller, Anna Moss, and Joseph (Seffi) Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, 1999. ISSN 0020-0190. doi: 10.1016/S0020-0190(99)00031-9. URL https://www.sciencedirect.com/science/article/pii/S0020019099000319.

Hao Li, Yiqun Zhang, Zhaoyan Guo, Chenxu Wang, Shengji Tang, Qiaosheng Zhang, Yang Chen, Biqing Qi, Peng Ye, Lei Bai, Zhen Wang, and Shuyue Hu. LLMRouterBench: A massive benchmark and unified framework for LLM routing. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 37733–37754, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1881. URL https://aclanthology.org/2026.findings-acl.1881/.

Shaoang Li and Jian Li. Polar: Online learning for lora adapter caching and routing in edge llm serving. *arXiv preprint arXiv:2604.16583*, 2026.

Mistral AI Team. Mistral small 3. https://mistral.ai/news/mistral-small-3/, January 2025.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, Mohammed Kadous, and Ion Stoica. Routellm: Learning to route llms from preference data. In *International Conference on Learning Representations*, volume 2025, pp. 34433–34448, 2025.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Seamus Somerstep, Felipe Maia Polo, Allysson Flavio Melo de Oliveira, Prattyush Mangal, Mírian Silva, Onkar Bhardwaj, Mikhail Yurochkin, and Subha Maity. CARROT: A Cost Aware Rate Optimal Router. In *ICLR 2025 Workshops: FM-Wild*, 2025. URL https://mlanthology.org/iclrw/2025/somerstep2025iclrw-carrot/.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, and Jason Wei. Challenging BIG-bench tasks and whether chain-of-thought can solve them. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 13003–13051, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.824. URL https://aclanthology.org/2023.findings-acl.824/.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 95266–95290. Curran Associates, Inc., 2024. doi: 10.52202/079017-3018. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/ad236edc564f3e3156e1b2feafb99a24-Paper-Datasets_and_Benchmarks_Track.pdf.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*, 2024.

Shan Yu, Yifan Qiao, Mingyuan Ma, Yangmin Li, Shuo Yang, Xinyuan Tong, Yang Wang, Zhiqiang Xie, Yuwei An, Shiyi Cao, Ke Bao, Deepak Vij, Xiaoning Ding, Yichen Wang, Qingda Lu, Zhong Wang, Gao Gao, Harry Xu, Junyi Shu, Jiarong Xing, and Ying Sheng. Prism: Cost-Efficient Multi-LLM serving via GPU memory ballooning. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 75–93, Seattle, WA, July 2026. USENIX Association. ISBN 978-1-939133-55-7. URL https://www.usenix.org/conference/osdi26/presentation/yu-shan.

A MEASUREMENT RECONSTRUCTION AND UNCERTAINTY

For request i, let a_i be the offered arrival relative to the replay epoch t_0, s_i the absolute HTTP send time, and f_i completion time. The desired latency is $\ell_i = f_i - t_0 - a_i$; recorded HTTP latency is $h_i = f_i - s_i$. Their difference is admission/concurrency waiting, $s_i - t_0 - a_i$. Thus a policy can improve h_i by delaying dispatch without improving ℓ_i.

Baseline reconstruction. The baseline client saves $d_i = s_i - t_0$, so $\ell_i = (d_i - a_i) + h_i$. Its saved schedule is unscaled and is divided by the rate multiplier, 0.1 in these comparisons. CoServe's saved schedule is already scaled. The reconstructed baseline epochs agree within 1 ms within each run, and the paired main offered schedules agree within 1 ms.

Two-sided bounds for CoServe. Sending after the offered arrival implies $t_0 \le U = \min_i(s_i - a_i)$, using successful rows with complete timestamps. A startup log also records the server snapshot timestamp, rounded to integer seconds, before the replay epoch is set. If that value is b, then $B = b - 0.5 \le t_0$. The server snapshot and client use the same host wall clock; the derivation assumes coherent timestamps and no clock jumps. Consequently,

$$f_i - U - a_i \;\le\; \ell_i \;\le\; f_i - B - a_i. \tag{9}$$

The upper-minus-lower width is constant within a run: $U - B$. It is 3.496 s on A and 5.373 s on B. The bounds are not statistical intervals and do not assume the earliest dispatch had zero admission wait.

The ordering follows the replay path: a successful readiness snapshot is received before initialization, and the first request coroutine sets the replay epoch afterward. The snapshot time is generated by the backend's wall-clock call. We use the readiness timestamp, not an assumed ordering between concurrent controller heartbeats and request dispatch.

Means and nearest-rank p95 are bounded by aggregating the lower and upper per-request endpoints. SLO lower bounds count upper latency endpoints within 100 s; SLO upper bounds count lower endpoints within 100 s. Failures count in neither numerator and remain in the issued-request denominator. The CDF band is formed in the same way at every latency threshold. Table intervals round outward.

When the startup log is unavailable. The older top-1 and no-awareness variants retain the CSV-derived upper bound on t_0 but no usable startup lower bound. Their arrival latency is therefore reported only as a lower bound, and their SLO only as an upper bound. Comparing two lower bounds cannot establish their ordering. The lower-penalty reference has a valid but wider 42.231 s epoch interval.

Quality and success. The `correct` field is an offline query/model lookup. Its labelled denominator may include a chosen route whose live request fails; it is therefore explicitly a routing proxy, not successful quality-constrained goodput. Missing labels are excluded and counted, not replaced by zeros. Live-generation scoring, generation caps, and offline model labels are not interchangeable. No equivalence or non-inferiority test is claimed.

Scope of uncertainty. The intervals address the missing epoch only. They do not quantify variability across workloads, seeds, model quality, or deployment environments. Requests interact through queues, so per-request resampling would not turn these two traces into independent deployment replications. HTTP TTFT and backend TTFT also use different instrumentation boundaries; a harmonized arrival-based TTFT comparison is not claimed.

B IMPLEMENTATION, WORKLOAD, AND VARIANT DETAILS

Model pool and cost profiles. Table 2 lists the historical configured pool and the saved v10 serving parameters used in the main replay logs. These fitted service parameters are approximations for scoring proposed sets, not guarantees for every batch. Footprints are the planner's configured backend estimates, which can differ from parameter-count-based weight estimates; KV reservations and transient initialization overhead are additional.

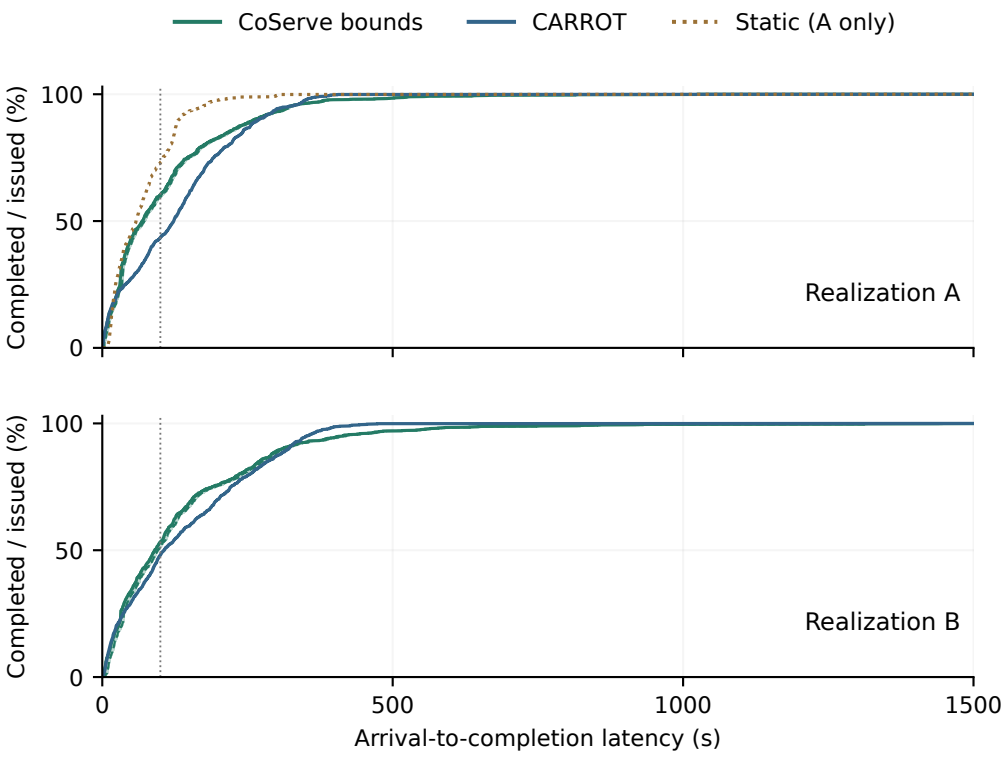


Figure 4: **Complete arrival-based completion distributions.** CoServe's bands enclose the CDFs permitted by the replay-epoch bounds; their actual width is preserved. Baselines are reconstructed point curves. All issued requests, including failures, remain in the denominator. The common axis includes every observed successful completion; the vertical line marks the 100 s deadline.

Table 2: Configured model footprints and saved service profiles. T_m: decode rate (tokens/s); B_m: batch width; L_m: profiled activation cost (s).

Model	Footprint (GB)	T_m	B_m	L_m
qwen05b	1	75	32	1.00
math-1b5	3	71	17	2.00
coder-1b5	3	76	32	2.01
r1-1b5	4	137	21	2.00
llama-3b	6	71	13	2.58
llama-8b	15	56	5	6.01
mistral-24b	88	82	87	15.05
coder-32b	62	48	124	21.07
math-72b	136	33	48	46.68
r1-70b	132	33	39	45.17
qwen-72b	136	29	30	46.40

The pool comprises Qwen2.5 general, Math, and Coder models (Qwen et al., 2025; Yang et al., 2024; Hui et al., 2024), Llama-3.2-3B and Llama-3.1-8B (Grattafiori et al., 2024), Mistral-Small-24B (Mistral AI Team, 2025), and DeepSeek-R1-Distill-Qwen-1.5B and DeepSeek-R1-Distill-Llama-70B (Guo et al., 2025). Short identifiers in the implementation refer to these fixed choices, not families searched during deployment.

Motivation evidence. The Figure 1 footprint sum uses planner configuration estimates, excluding KV pools and activation headroom. Activation medians use complete observed `activating`-to-`activated` intervals from an existing run at nominal 1 Hz resolution: $n = 6$, 13, and 18 for

Math-72B, Qwen-72B, and R1-70B. Intervals include backend activation work, not only storage transfer. This evidence establishes the cost regime; it is not a measured latency benefit for the schematic routing choice.

Controller settings and execution. The main logs identify candidate-list routing, a causal demand projection with a 120 s horizon, a 180 s trailing window, forecast weight 0.3, 12 curated starting configurations, and the saved v10 batch-wave parameters. Planning runs on a 15 s period with nominal 1 Hz telemetry. The restored controller supports different aggregations, activation penalties, and urgency options; these should not be conflated into a single globally optimal policy. Loading, dwell, and drain checks can defer the selected set, and the backend revalidates requested residency changes.

For old queries without a resident candidate, the scarcity-weighted urgency option uses $a_q/|C(q)\setminus R_t|$. Each query nominates its smallest absent candidate; a model receives the maximum priority among its nominators. Backend waiting queues provide an additional urgency signal. When feasible candidates contain the highest-priority urgent model, the search is restricted to those sets. The restored defaults trigger query-age urgency after 1,800 s or a backend waiting count of at least five; enabled options can differ across historical builds. These mechanisms protect against some prolonged waits without proving a bounded response time.

Workload construction. The 5,000 saved rows comprise 1,086 MBPP variants, 164 HumanEval, 1,179 MATH, 71 GSM-Hard, 1,250 BBH, 812 MMLU, and 438 MMLU-Pro rows. Some hard code entries reuse MBPP tasks with long canonical solutions; they are relabelled rather than independent new problems. Rows are divided by domain/difficulty strata, not by underlying problem identity. Exact prompt matching finds 76 test rows also present in training; related variants can share a source even without identical text. This overlap is retained and disclosed because the evaluation uses the existing trained routers and saved test records.

The session generator uses sizes 15–20, a domain/difficulty purity target of 0.8, gaps of 5–15 seconds, and a periodic rate envelope before the $0.1\times$ replay scaling. The metadata describes the Azure reference as the rate-envelope shape only. Therefore the workload is a controlled, constructed mixture, not an unmodified production trace, and its purity and difficulty assignments are properties of the generator.

B.1 Available variant records

Table 3: Available variants, with the same metric definitions as Table 1. One-sided bounds indicate missing startup logs. The two full-system references are both retained.

System	Proxy (n)	HTTP	Arrival mean	Arrival p95	SLO (%)	Fails
CoServe A	0.756 (980)	45.7	107.4–111.0	319.7–323.3	59.6–60.4	0
CoServe, lower penalty	0.751 (970)	41.6	98.9–141.3	321.3–363.6	50.4–62.4	0
No forecast	0.737 (938)	47.1	135.1–139.5	423.5–427.9	53.9–54.9	0
Top-1 candidates	0.758 (1000)	50.1	$\geq$ 190.9	$\geq$ 465.6	$\leq$ 31.4	0
No residency awareness	0.757 (1000)	168.4	$\geq$ 273.8	$\geq$ 762.7	$\leq$ 30.0	5

The top-1 and no-residency-awareness runs use earlier builds than the main CoServe A run and the lower-penalty reference. Top-1 restricts model alternatives. The no-awareness variant also replaces the load/slow-tier routing rule with highest predicted quality, so it does not isolate the implementation of a shared state store. The lower-penalty run does not establish byte-identical matching to these variants. Consequently, the table is evidence about observed configurations, not a controlled additive decomposition of the full system's gain.

No-forecast and full-system results differ substantially once admission is included, but configuration differences and interval width limit causal attribution. We do not infer significance by comparing one-sided lower bounds, treat nonsignificant quality differences as equivalence, or attribute every change to the named component alone.

Candidate-artifact provenance. The candidate multiplicity analysis uses the saved `step9/candidate_lists_carrot_test.json`, whose basename appears in the main A replay log. All 1,000 sets reconstruct from the saved CARROT scores at threshold 0.7 with the highest-score fallback. The analysis export retains candidate counts, exact-training-prompt overlap flags, and source hashes; its portable rebuilding script reproduces the histogram. The historical replay log does not record a candidate-file content hash, so the association establishes the named artifact rather than byte identity at execution time. A separate step10 candidate variant is excluded from this analysis. Removing exact prompt matches is a sensitivity check, not a source-grouped resplit or router retraining.

C STRUCTURAL ANALYSIS OF THE CANDIDATE INTERFACE

This analysis isolates why request alternatives matter to residency. It concerns the snapshot coverage objective in Equation (2), not a performance guarantee for CoServe's dynamic policy. Candidate sets are taken as given; predicted membership does not imply actual answer correctness.

C.1 PROOF OF PROPOSITION 1

Proof. Let the memory budget be 2, with model costs $w_A = 2$ and $w_B = w_C = 1$. There are two queries with weights $\omega_1 = 2$ and $\omega_2 = 1$. Consider the following candidate sets and committed routes:

Workload	$C(q_1)$	$C(q_2)$	$(g(q_1), g(q_2))$
I	$\{A, B\}$	$\{C\}$	(A, C)
II	$\{A\}$	$\{C\}$	(A, C)

Both projections contain exactly the same identifiers and weights, so both have $n_A^g = 2$, $n_C^g = 1$, and $n_B^g = 0$. The feasible resident sets are $\varnothing$, $\{A\}$, $\{B\}$, $\{C\}$, and $\{B, C\}$. In workload I, their respective coverage values are $0, 2, 2, 1, 3$, making $\{B, C\}$ uniquely optimal. In workload II the values are $0, 2, 0, 1, 1$, making $\{A\}$ uniquely optimal. A controller restricted to the identical projection must make the same deterministic choice in both cases and hence fail in at least one. A randomized controller likewise cannot choose both distinct unique optima with probability one. All other observed state may be held fixed because this snapshot objective excludes transitions and execution costs. □

The result does not require uncertainty or an inaccurate quality estimator. It arises from discarding alternatives. A controller receiving candidate sets, full quality scores, or equivalent information is outside the proposition's restricted observation model. Unit-weight queries give the same construction by duplicating q_1.

C.2 A QUANTITATIVE SEPARATION FOR THE COMMITTED-ROUTE INTERFACE

Proposition 2 (No pool-independent coverage factor from committed routes)**.** *For every integer $k \geq 2$ and every possibly randomized residency rule observing only committed routes, request weights, model footprints, and the budget, there is an instance with $2k$ unit-footprint models, k unit-weight queries, and capacity for one model such that*

$$\frac{\mathbb{E}[H(R_{\text{committed}})]}{\max_{|R|\leq 1} H(R)} \leq \frac{1}{k}. \tag{10}$$

The expectation is over the rule's randomization. A controller that observes the candidate sets can cover every query in this instance.

Proof. Use models $a_1, \ldots, a_k, b_1, \ldots, b_k$. For each $j \in \{1, \ldots, k\}$, construct workload W_j with $C_j(q_i) = \{a_i, b_j\}$ and committed route $g(q_i) = a_i$ for every i. Every workload exposes the same projection: one request per a_i, zero per b_i, and identical footprints and budget. Consequently the rule has the same selection probabilities in all W_j. Let p_A be its total probability of choosing an a_i, and

p_j its probability of choosing b_j; then $p_A + \sum_j p_j \le 1$. In W_j, an a_i covers one query, b_j covers k, and any other b covers zero. Hence

$$\frac{1}{k}\sum_{j=1}^{k}\mathbb{E}[H_j(R)] = p_A + \sum_{j=1}^{k} p_j \le 1. \tag{11}$$

At least one workload has expected coverage at most one. Its candidate-aware optimum selects b_j and has value k, proving the result. □

The separation is an information bound: even an exact optimizer downstream cannot recover alternatives erased upstream. It permits the downstream rule to select models with zero committed demand, so its force is not an artifact of excluding such models from the search. It assumes the current candidate sets cannot be reconstructed from additional signals or prior knowledge. A router that communicates alternatives, or jointly chooses commitments after evaluating residency, is outside this observation model. The construction establishes a worst-case opportunity for the interface; it is not a k-fold latency claim or a guarantee that CoServe's heuristic attains the candidate-aware optimum.

C.3 The value of delaying a model commitment

Let C be the fixed, nonempty candidate set of one request and let S be the backend state revealed before dispatch. Write $\ell_m(S)$ for a finite, integrable serving cost of choosing $m \in C$; it can include waiting for availability. A state-blind commitment chooses one model before observing S, whereas a delayed decision can choose after observing S. Define

$$V_{\text{early}} = \min_{m\in C}\mathbb{E}[\ell_m(S)], \qquad V_{\text{late}} = \mathbb{E}\left[\min_{m\in C}\ell_m(S)\right]. \tag{12}$$

Proposition 3 (Value of retained alternatives)**.** *For a finite candidate set, $V_{\text{late}} \le V_{\text{early}}$, with the exact gap*

$$V_{\text{early}} - V_{\text{late}} = \min_{m\in C}\mathbb{E}\left[\ell_m(S) - \min_{u\in C}\ell_u(S)\right]. \tag{13}$$

Equality holds exactly when some fixed candidate minimizes cost almost surely.

Proof. Subtract the same finite quantity $\mathbb{E}[\min_u \ell_u(S)]$ from every term in the finite minimum defining V_{early}. Each resulting random variable is nonnegative, proving the identity and inequality. The minimum expectation is zero exactly when at least one of these nonnegative variables is zero almost surely. □

For example, suppose exactly one of k candidates is resident, each equally likely, with cost 1 for that candidate and $D > 1$ for the others. Then $V_{\text{late}} = 1$ and $V_{\text{early}} = 1 + (1 - 1/k)(D - 1)$. Thus the opportunity grows with the availability penalty and disappears when a single model is always best. This is a one-request decision result with exogenous state and an exact cost resolver. CoServe's lexicographic resolver approximates a serving preference; coupled queues, future state changes, and prediction error prevent treating this identity as its end-to-end guarantee.

C.4 Coverage structure and its limits

Proposition 4 (A standard coverage subproblem)**.** *For fixed candidate sets and nonnegative request weights, H_t is monotone and submodular. Maximizing it over R subject to $\sum_{m\in R} w_m \le B$ is NP-hard in general.*

Proof. For $m \notin R$, the marginal gain is

$$H_t(R\cup\{m\}) - H_t(R) = \sum_{q\in Q_t}\omega_q \mathbf{1}\{m \in C(q),\ C(q)\cap R = \varnothing\}. \tag{14}$$

Every term is nonnegative, proving monotonicity. If $R \subseteq S$, every query uncovered by S is also uncovered by R, so the marginal gain of m under S is no greater than under R. This is diminishing returns, hence submodularity.

For hardness, take a budgeted maximum coverage instance with weighted elements, sets, and positive set costs (Khuller et al., 1999). Create one query per element, one model per set, and put a model in a query's candidate set exactly when its set contains the element. Use element weights as ω_q, set costs as w_m, and retain the budget. Elements in no set can be removed without affecting the optimum, ensuring nonempty candidate sets. Feasible set choices and objective values are identical, giving a polynomial reduction. This restricted problem excludes queues, loading, and transient memory; hardness already holds without them. □

This is a specialization of classical coverage theory. It explains why summing per-model candidate counts double-counts shared demand, and why memory-aware selection needs set-dependent values. The standard coverage approximation results do not automatically apply to CoServe's full objective. The implemented cost includes assignment-dependent batch waves, control penalties, and transition costs; we do not assume it is monotone or submodular.

A constructive guarantee for the coverage surrogate. If footprints are equal and memory admits at most $b \geq 1$ models, candidate information also enables a simple certified solution to the snapshot problem. Starting with $R_0 = \varnothing$, repeatedly add the model of largest uncovered marginal gain in Equation (14). After b additions (or exhausting the pool),

$$H_t(R_b) \geq \left[1 - (1 - 1/b)^b\right] H_t(R^\star) \geq (1 - 1/e) H_t(R^\star), \tag{15}$$

where $R^\star$ is a coverage-optimal set of at most b models. To verify the bound, submodularity gives

$$H_t(R^\star) - H_t(R_i) \leq \sum_{m \in R^\star \setminus R_i} [H_t(R_i \cup \{m\}) - H_t(R_i)].$$

At most b terms appear, so the greedy increment is at least $1/b$ of the remaining optimality gap. Iterating from $H_t(R_0) = 0$ proves Equation (15); exhausting the pool covers every coverable query. This classical cardinality-constrained coverage guarantee contrasts with Proposition 2: preserving alternatives permits a pool-independent approximation for the simplified objective. It applies to this greedy coverage procedure, not to the evaluated controller, heterogeneous footprints, or latency optimization.

C.5 Search scope and computational cost

Let $M = |\mathcal{M}|$, $r = |R_t|$, S be the number of curated seeds, and K the number of distinct sets considered before urgency filtering. The implemented neighborhood satisfies

$$K \leq S + 1 + (M - r) + r + r(M - r) = S + 1 + M + r(M - r). \tag{16}$$

This counts seeds, staying, additions, removals, and swaps; duplicate removal and memory checks can reduce the count. It is enumeration of a local neighborhood, not the full 2^M space.

Let N count pending and projected queries, and d be their maximum candidate-set size. With a fixed load snapshot, route selection costs $O(Nd)$ per resident set. Batch positions and costs take $O(N)$; transition and residence penalties take $O(M)$. Thus default sum/mean scoring costs $O(K(Nd + M))$ after candidates are built. A conservative bound for percentile aggregation adds $O(KN \log N)$. Straightforward feasibility checking and canonicalization during enumeration cost at most $O(K_0 M \log M)$ for $K_0 = S + 1 + M + r(M - r)$ generated proposals. These are operation-count bounds, not newly measured controller timings. They do not provide an approximation guarantee or establish multi-node serving scalability.

D Controlled ablation of proposal-conditioned scoring

This experiment tests the specific step $R \mapsto \{m_R(q), p_R(q)\}$ in Equation (8). It complements the historical deployment with a controlled intervention on a different model pool. **FULL** recomputes assignments for each residency proposal. **FIXED** assigns each pending/projected request once per planning tick using observed availability and retains that destination when scoring all proposals. A destination absent from a proposal incurs the same uncovered-request penalty, even if another candidate would be present. Runtime dispatch, candidate scores, proposal-enumeration procedure, forecasting, urgency, and cost parameters are unchanged. Thus this is an ablation of proposal-conditioned scoring, not of candidate-aware dispatch.

Deployment and provenance. Each run uses one B200 and BF16 Llama-3.1-8B-Instruct, Gemma-2-9B-it, and Granite-3.3-8B-instruct, with model revisions `0e9e39f`, `11c9b30`, and `51dd4bc`, respectively. The Prism-based activation planner enforces a configured 72 GB budget: weights plus a 10 GB KV reservation per resident model, with 15 GB additional activation headroom. Any pair is feasible; activating the triple is not. This is a controller-enforced serving budget, not a physical GPU partition. Telemetry shows at most two resident models and 12–18 measured activations per primary run. Models load from host memory; observed activation intervals are approximately 4–14 s. This reduced setting does not reproduce the historical eleven-model deployment or its large-model loading costs.

Within every primary pair, the logged replay, configuration, launcher, prompts, candidate scores, and trace hashes match. Four isolated single-GPU replicas run two schedule pairs concurrently, then swap policy/device assignments for two further schedules. Each run starts fresh, warms every model before timing, and starts from the same resident set, {Gemma, Granite}, selected by calibration candidate coverage. The common backend adds Granite support and Gemma asynchronous-loading hooks. Granite was smoke-checked on calibration responses; equivalence to a reference inference implementation was not established. Shared fixes include per-replica state isolation and a 60 s minimum observation span for the causal forecast, avoiding an unstable initial rate estimate. A startup cache-isolation fix changed the launcher between waves, identically within each pair.

Queries and predicted alternatives. We use released per-query response labels from LLMRouterBench (Li et al., 2026), aligning the three models by source index, original question, and ground truth. Across MATH500, MMLU-Pro, GPQA, and BBH, 2,779 binary-labelled records are partitioned by normalized question identity into 1,675 training, 546 calibration, and 558 test records; four duplicate training questions are removed before fitting. No normalized question crosses these partitions. A TF-IDF and dataset-feature logistic predictor is fit per model on training data, with regularization selected by calibration log-loss. Calibration AUC ranges from 0.67 to 0.70. This lightweight predictor is separate from the historical CARROT estimator.

The threshold is the highest of $\{0.7, 0.6, 0.5\}$ yielding at least 20% multi-candidate requests on calibration, which selects $\tau = 0.5$; an empty threshold set uses the highest-score singleton fallback. This rule selects sufficient alternatives to exercise the scoring intervention, rather than certifying a quality constraint. No test labels enter candidate construction or routing. A fixed, dataset-stratified sample of 200 test requests contains 74 BBH, 72 MMLU-Pro, 35 MATH500, and 19 GPQA requests; candidate sizes are one for 100 requests, two for 63, and three for 37. Four session-shaped schedules replay these same queries over 1,740 s, approximately 6.9 requests/min. They are four arrival schedules, not four independent query sets.

Timing and response grading. Scheduled-arrival and completion wall timestamps are recorded directly in the replay client; all reported delays include admission waiting. Each model uses its own chat template, temperature 0.7, and a 1,024-token generation cap with context length 4,096. The common drain cap is 480 s after the last arrival. A local grader scores generated answers against ground truth; published model scores are not substituted for live correctness. On training/calibration responses, grader agreement with released labels is 94.9–99.8% by dataset. Most BBH disagreements arise because the local grader accepts an answer letter followed by option text. Agreement with a published scorer is not an independent correctness guarantee. Unresolved extractions remain unknown; primary runs have 8–15 such cases each and 5–13 truncated responses. Correct-and-timely bounds count unresolved timely answers as wrong/right, over all 200 issued requests. These are missing-grade bounds, not confidence intervals; they do not capture grader error or generation variability.

Primary result: a delay effect under a saturated deadline. Every primary run completes all 200 requests. FULL lowers p95 on three schedules by 16.1, 9.2, and 10.6 s, while increasing it by 1.7 s on S2 (Table 4). The mean of these four paired p95 differences is -8.5 s; it is not a pooled percentile or a significance claim. Mean arrival delay falls on the same three schedules. Mean admission delay is 2.7/5.0, 5.3/4.7, 6.7/8.5, and 2.7/4.1 s for FULL/FIXED across S1–S4. Proposal-level traces confirm the intervention is exercised: on 91–100% of FIXED planning ticks, at least one proposal assigns a request differently under the FULL resolver.

Table 4: **Controlled scoring ablation and reference policies.** All rows issue 200 requests. Timings start at scheduled arrival; p95 uses linear interpolation between ordered latencies. Correct-and-timely intervals bound unresolved grades. Primary rows have no failures or censoring. †: load-on-miss has four failed requests; exploratory FULL S2 has one censored request. Their timing summaries include observed failure/censor times and are not uncensored successful-completion summaries.

Schedule	Policy	Deadline (%)	Correct & timely (%)	Mean (s)	p95 (s)
Primary, approximately 6.9 requests/min					
S1	FULL	100.0	47.5–53.5	7.3	28.4
S1	FIXED	99.5	50.0–55.5	11.0	44.5
S2	FULL	100.0	52.0–57.5	9.4	43.1
S2	FIXED	100.0	53.5–60.0	8.7	41.4
S3	FULL	100.0	54.0–61.0	11.2	47.7
S3	FIXED	100.0	54.5–61.5	13.6	56.9
S4	FULL	100.0	51.0–58.5	7.4	27.6
S4	FIXED	100.0	50.5–54.5	8.2	38.2
Schedule-1 references, same offered rate					
S1	Static	100.0	49.5–52.0	3.6	10.6
S1	Load-on-miss†	97.0	45.0–53.0	10.0	25.5
Exploratory, three times the offered rate					
S1	FULL	100.0	53.5–59.0	7.4	23.9
S1	FIXED	100.0	49.0–55.5	9.8	62.3
S2	FULL†	99.5	53.0–58.0	11.7	44.8
S2	FIXED	100.0	53.5–57.5	12.0	64.7

Both policies already attain the 100 s deadline for 99.5–100% of requests, so these runs establish no material deadline advantage. Correct-and-timely intervals overlap on every schedule. They establish neither a quality improvement nor equivalence: for bounds $[L_F, U_F]$ and $[L_X, U_X]$, the paired difference lies in $[L_F - U_X, U_F - L_X]$, not the difference of corresponding endpoints. Averaging these bounds across primary schedules yields $[-6.75, +5.50]$ percentage points, addressing missing grades only. One stochastic realization per policy and schedule does not quantify sampling variability.

References and exploratory load. On S1, the fixed resident set {Gemma, Granite} is faster than either dynamic policy (p95 10.6 s). It routes out-of-set requests to the highest-predicted-quality resident model, relaxing the candidate restriction; its correct-and-timely bounds overlap FULL's. The load-on-miss reference uses candidate-aware dispatch and LRU eviction without proposal scoring: it has four timeouts and 20 activations, versus 13 for FULL. These references do not establish a quality advantage for dynamic residency. First attempts at the static/load-on-miss runs failed to save request CSVs because shutdown tried to join an unstarted planner thread; only the complete S1 reruns after the join guard are tabulated.

After inspecting the first primary pair, a logged protocol amendment added two exploratory schedules at three times the offered rate. FULL's p95 is lower in both, but its S2 run leaves one response censored, and deadline attainment remains 99.5–100%. These observations are reported separately from the primary comparison. Boot failures and cancelled premeasurement attempts are retained in the campaign records. Taken together, the primary runs support the practical relevance of recomputing assignments during residency scoring in this setting, while leaving its quality tradeoff and general latency benefit unresolved.