POLARIS: Policy-Orchestrated Adaptive LLM Inference Across Heterogeneous

Somesh Rahul[1], Krishna Suman Dara[2], Raghuram Malpe Pai[2]

IEEE Independent Researcher, Fremont, CA, USA[1], IEEE Independent Researcher, Sunnyvale, CA, USA[2]

Corresponding author:Somesh Rahul, Email: someshrahul@ieee.org

Production language-model serving now composes retrieval, multimodal encoders, speculative decoding, quantization, edge–cloud partitioning, privacy isolation and fault recovery, and most serving frameworks tune those mechanisms one at a time inside a fixed pipeline. POLARIS treats serving as a constrained online composition problem instead: it derives a policy vector per request, prunes a capability graph down to plans that satisfy the request's hard constraints, and scores what survives against a published objective. We evaluate it in a discrete-event simulation of 12,000 requests across six workload classes, against six independently implemented baseline policies and an offline oracle, with 24 seeds and bootstrap intervals throughout. The headline result isn't a speedup. Against a latency-greedy adaptive controller, the strongest deployable comparator here, POLARIS is 11.3% slower at P95 (95% CI [10.0, 12.7]), and that penalty holds in all 72 one-at-a-time parameter sweeps. What it buys is compliance and cost: violations of the per-request output-confidence floor drop from 18.68% to 6.60% (64.7% lower, CI [62.9, 66.4]), serving cost drops 14.7% and energy 10.2%, putting realized utility within 2.7% of the offline oracle. Planning costs 0.841 ms of measured CPU per decision. Two mechanisms the design argues for don't pay under this workload: the carbon term in the objective produces no measurable utility change, though the workload carries no binding carbon budget and a single grid-intensity pair, so that result bounds the mechanism rather than refuting it; and recovery re-planning cuts mean time to recovery 19.3% without improving utility at all. The ablation also shows most of the end-to-end gain comes from speculative decoding being available rather than from orchestration. Policy-aware composition is a compliance-and-cost mechanism, and it should be argued for on those terms. The architectural claim behind it is separable from the simulation: hard constraints are enforced structurally, by pruning the plan space before anything is scored, because a constraint expressed as a weighted term is tradeable by construction.

Keywords: Large language model serving, adaptive inference, edge–cloud computing, constrained optimization, runtime verification, simulation study.

2026. In Mukesh Saraswat & Harish Sharma (eds.), *Emerging Technologies and Intelligent Systems: Theory, Applications and Innovations*, 88–108. Computing & Intelligent Systems, SCRS, India. DOI: https://doi.org/10.56155/978-81-975670-4-9-5

1 Introduction

A production language-model request is no longer one model on one GPU. It may need an image encoder, a retrieval subsystem, a large decoder, speculative verification, quantized execution, a trusted execution environment, and a hop between an edge node and a data center. At the same time the request carries a tail-latency target, a cost budget, a privacy label and a minimum acceptable output confidence.

That combination changes the question the scheduler is answering. It isn't only where to run a model. It's which inference capabilities should be switched on for this particular request, and that answer differs request to request. A product lookup benefits from retrieval. A conversational rewrite doesn't, and paying for retrieval anyway spends prefill budget and adds distracting context. A personalization request carrying user data may require attested execution, which removes most of the cheap placements. Speculative decoding needs a resident draft model, so it competes for the same memory that governs how many requests a GPU can hold.

Existing systems optimize these mechanisms in separate control loops. Each loop is locally sensible and they conflict. Deeper speculation raises memory pressure and forces a worse placement. Privacy isolation eliminates otherwise cheap GPUs. Retrieval consumes the latency slack a long generation needed.

We built POLARIS to test one claim: that making governance, privacy, retrieval need, modality and confidence into first-class scheduling inputs, and reasoning over alternative pipeline topologies rather than parameters of one pipeline, is worth the control-plane complexity.

What came out is more limited than we expected, and that's the contribution. A latency-greedy adaptive controller beats POLARIS on latency at every operating point we tested. POLARIS wins on the things the greedy controller doesn't model, and the size of that win is measurable. It refuses requests it can't serve compliantly rather than serving them badly. Its ablation shows two of its own mechanisms contributing nothing measurable.

Our contributions are:

1. A policy-vector abstraction and a capability-graph planner that changes pipeline topology per request rather than tuning a fixed path, with the objective weights published rather than described.

2. A simulation study of 12,000 requests over six workload classes against six independently implemented baselines and an offline oracle, with the unit of analysis the seed and bootstrap intervals on every reported quantity.

3. A quantified answer to what policy orchestration actually buys, including a measured latency penalty and two mechanisms that come out null.

4. A sensitivity analysis over all 38 cost-model parameters, one at a time, reporting whether each qualitative finding survives.

Everything reported here is simulated except planner overhead, which is measured wall-clock of the planner code. We say which is which throughout, because the distinction is the difference between a result and a claim.

2 Related Work

Serving and scheduling. Modern serving systems expose several optimization surfaces. Orca introduced iteration-level scheduling and selective batching for autoregressive models [1]. vLLM's PagedAttention reduced KV-cache fragmentation and raised achievable concurrency [2]. FlexGen studied high-throughput inference under constrained GPU memory [3], and AlpaServe applied model parallelism to statistical multiplexing across bursty arrivals [4]. A more recent line separates the prefill and decode phases outright: DistServe disaggregates them [5], Splitwise maps them onto different hardware classes [6], and Sarathi-Serve uses chunked prefills to avoid decode stalls [7]. Since 2025 the same problem has been posed as an explicit service objective rather than a latency target: GoodServe schedules for goodput under per-request SLOs [8], Laser moves the decision to layer granularity across concurrent SLO classes [9], and AdaServe customises speculative decoding per class [10]. Punica addresses multi-tenant LoRA serving [11], Keyformer reduces KV pressure by retaining high-value tokens [12], and FlashAttention attacks the attention I/O bottleneck underneath all of it [13,14]. KV pressure has since been attacked from both ends, by caching context across requests [15] and by compressing the cache within one at near-zero cost [16].

These systems tune one pipeline well. None of them decides whether retrieval should run for a given request, and that is the gap POLARIS is aimed at.

Speculation and quantization. Speculative decoding accelerates generation with a smaller draft model while preserving the target distribution [17]. Medusa replaces the separate draft model with multiple decoding heads [18], and EAGLE performs feature-level speculative prediction [19]. We use Leviathan et al.'s analytical form directly in the cost model rather than assuming a fixed speedup constant, which matters because the speedup that emerges depends on acceptance rate and draft cost rather than on a number we chose. Post-training quantization reduces memory and arithmetic cost with different calibration strategies across SmoothQuant [20], AWQ [21] and GPTQ [22].

Retrieval. Retrieval-augmented generation supplies an orthogonal grounding path [23], with dense first-stage retrieval from DPR [24] and late-interaction ranking from ColBERT [25]. Large-scale vector search rests on FAISS [26] and HNSW-style approximate nearest-neighbor indexing [27]. In POLARIS retrieval is a decision with a real price, not a fixed stage.

Privacy, agents, sustainability. Differential privacy provides the formal model [28] and DP-SGD operationalizes it for deep learning [29]. Agentic systems combine reasoning and tool use in ways that raise the cost of an unverified action, as in ReAct [30], Toolformer [31], Reflexion [32] and Constitutional AI [33]. Serving those agents carries its own memory profile, which recent architecture work addresses by tiering across HBM and high-bandwidth flash [34]. Green AI argued for efficiency-aware reporting [35], and carbon-accounting work showed that hardware and geography move emissions materially [36]. POLARIS treats these as scheduling inputs rather than after-the-fact checks, and our carbon result below is a caution about how much that buys.

What's different here. A conventional framework asks what parameters a pipeline should use. POLARIS asks which policy-compliant pipeline should exist for this request, and when it should change. The closest recent neighbour is cost-aware model orchestration [37], which routes between models on price and quality; POLARIS differs in that the unit of choice is a whole pipeline topology under hard constraints, not a model. The contribution of this paper is less the architecture than the measurement of what that reframing is worth.

3 System Model

3.1 Request policy vector

For a request r, POLARIS constructs a policy vector

$$\mathbf{p}_r = [\, m_r,\ q_r,\ s_r,\ \rho_r,\ L_r,\ B_r,\ C_r,\ \kappa_r \,] \tag{5.1}$$

where m is modality, q estimated retrieval need, s privacy sensitivity, rho a minimum reliability, L the tail-latency target, B a monetary budget, C a carbon budget and kappa a minimum acceptable output confidence.

Some fields are hard. A privacy policy requiring attested execution can't be traded for latency. Others are soft preferences the optimizer may spend. The separation is the reason the planner has two stages rather than one weighted score, and it's what makes the system refusable: if no plan satisfies the hard fields, POLARIS declines the request instead of serving it out of policy.

Retrieval need is estimated, not known. We model that with a classifier of stated accuracy (82% here) rather than assuming the planner is told, because assuming it is told is what turns an online controller into an oracle.

3.2 Capability graph

Serving mechanisms are vertices in a directed graph whose edges are valid compositions. The operators cover text, vision and audio encoding, retrieval gating and reranking, local

or cloud or split execution, quantization and speculative configurations, DP and TEE privacy, and verification and recovery. An execution plan is a path through that graph satisfying the request's dependencies. Unlike a parameter-only optimizer, a plan can omit or insert whole operators.

Each operator carries a state-dependent cost vector

$$\mathbf{c}_v(x_t) = [\,\text{latency},\ \text{cost},\ \text{energy},\ \text{carbon},\ \text{reliability},\ \text{utility}\,] \tag{5.2}$$

evaluated under system state

$$x_t = [\,g_t,\ n_t,\ w_t,\ h_t,\ f_t,\ z_t\,] \tag{5.3}$$

covering accelerator state, network state, workload, cache warmth, component health and regional cost and carbon.

3.3 The objective, in full

For each candidate plan the controller predicts outcomes and minimizes

$$\begin{aligned} J(\pi) = {} & w_L\cdot(\hat{L}/L_r) + w_B\cdot(\hat{B}/B_{\text{ref}}) + w_C\cdot(\hat{C}/C_{\text{ref}}) \\ & + w_E\cdot(\hat{E}/E_{\text{ref}}) + w_R\cdot(1-\hat{R}) + w_U\cdot(1-\hat{U}) \end{aligned} \tag{5.4}$$

subject to $\hat{U} \geq$ kappa_r, $\hat{R}$ $\geq$ rho_r, $\hat{B}$ $\leq$ B_r where the budget is hard, and pi $\in \Pi$(p_r), the set of policy-compliant plans.

Weights are fixed per service class, never per request. They're in Table 1 rather than described in prose, because an objective you can't recompute isn't an objective, it's an assertion. The reference constants are B_ref = 4.0 mUSD, C_ref = 0.45 gCO_2 and E_ref = 0.55 Wh per request.

Table 1: Objective weights by service class (Eq. 4). Fixed per class, never per request.

Service class	w_L	w_B	w_C	w_E	w_R	w_U
text	0.40	0.25	0.08	0.05	0.10	0.12
rag	0.30	0.20	0.07	0.05	0.10	0.28
multimodal	0.38	0.22	0.08	0.05	0.10	0.17
privacy	0.28	0.18	0.07	0.05	0.20	0.22
agentic	0.24	0.18	0.06	0.04	0.22	0.26
edge_interactive	0.48	0.18	0.08	0.06	0.10	0.10

Realized utility, the quantity reported in every results table, is

$$U(r) = \mathbf{1}\{\text{compliant}\} \cdot \left(0.40\,\hat{u}_L + 0.25\,\hat{u}_B + 0.25\,\hat{u}_U + 0.10\,\hat{u}_C \right) \tag{5.5}$$

with $\hat{u}$_L = max(0, 1 − latency/L_r), $\hat{u}$_B = max(0, 1 − cost/B_ref), $\hat{u}$_C = max(0, 1 − carbon/C_ref), and $\hat{u}$_U = (confidence − kappa)/(1 − kappa) clipped to [0,1]. The indicator is zero when the request failed or finished below its confidence floor. A refused request scores zero and is also counted separately, so refusals can't be hidden inside a good average.

Equation 5 is an evaluation metric, not the planning objective. The planner minimises Equation 4; Equation 5 is how a served request is scored afterwards. The two carry different terms and different weights deliberately, so that the system is not graded by the same function it optimises. That does not make Equation 5 neutral between policies, and §6.1 says where that bites.

3.4 Runtime verification and failing closed

A verifier checks execution against guard conditions: latency-slack collapse, memory pressure, retrieval failure, attestation failure, network degradation and shard failure. A guard violation triggers a re-plan under an updated capability graph with the failed component removed.

Re-planning never removes a mandatory control. A TEE failure can reroute to another attested region, or it can fail closed. It can't silently fall back to unprotected execution. That behavior shows up directly in Section 6.5, where POLARIS refuses 1.42% of requests during an attestation outage while every baseline queues and waits.

3.5 Algorithm

Algorithm 1 Policy-orchestrated planning for one request.

```
Require: request r, state x_t, capability graph G
 1  p_r ← ExtractPolicy(r)
 2  G' ← PruneInfeasible(G, p_r, x_t)                          // hard constraints only
 3  P ← GenerateCandidates(G', p_r, x_t)                                  // bounded
 4  for π in P: ŷ_π ← PredictOutcomes(π, x_t)
 5  π* ← arg min J(π) over plans meeting all hard constraints
 6  if no such π*: degrade or fail closed, record reason
 7  s ← Execute(π*)
 8  while s not terminal:
 9    o ← ObserveTelemetryAndVerification()
10    if GuardViolation(o, p_r): x_t ← Update(x_t, o); π* ← SafeReplan(π*, p_r, x_t)
11  UpdateController(p_r, x_t, π*, o)
12  return output, policy trace, execution trace
```

Separating line 2 from line 5 is the whole design. Feasibility is enforced structurally,

and learning only ever happens among plans already proven compliant. A monolithic reinforcement-learning agent over the same action space would have to learn the constraints, and a learned constraint is one that can be violated.

4 Experimental Methodology

4.1 What this study is

This study runs entirely in simulation. Every latency, cost, energy and carbon figure comes from the cost model described here, not from a deployed serving system. The one exception is planner overhead in Section 6.3, which is measured wall-clock of the planner function running on a named CPU.

We are explicit about this because of what the alternative does. Reporting modeled numbers in the register of measurements is the failure mode that makes serving papers unreproducible. A simulation can answer whether a control policy dominates another under a stated cost model. It can't tell you what P95 you'll see in production.

4.2 Cost model and calibration

There are 38 cost-model parameters. 9 are anchored to published sources and 29 are stated assumptions. Every one is listed with its source and its sweep range in the reproducibility package, and every one is swept in Section 6.7.

Two modeling choices matter more than the rest. Speculative decoding isn't a speedup constant: I compute expected accepted tokens per verification round as $(1 - \alpha^{\wedge}(\gamma+1))/(1 - \alpha)$ following Leviathan et al. [17] and charge γ draft passes plus one target pass, so the realized speedup emerges from acceptance rate and draft cost. And model weights are resident per GPU rather than per request, so a GPU's capacity is KV cache plus draft share, which is why speculation costs headroom and can change a placement.

GPUs are processor-sharing servers: a request co-resident with n others has its service time inflated by a factor $1 + 0.10(n - 1)$, reflecting that continuous batching is efficient but not free. Admission requires free KV memory, and requests queue when none is available.

4.3 Workload

Six classes with distinct policy vectors, summarised in Table 2: text generation, retrieval-heavy question answering, multimodal requests, privacy-sensitive personalization, agentic tool use, and edge-interactive requests. Ground-truth retrieval need is set per request and hidden from every online policy.

SLOs follow the shape production SLOs take, a time-to-first-token budget plus a per-output-token budget, because a flat per-request SLO makes every long generation a violation no matter what the scheduler does.

Arrivals are Poisson at 12 req/s, 500 requests per seed, 24 seeds. That operating point loads the system without saturating it, and Section 6.4 sweeps load from 6 to 26 req/s.

Table 2: Workload classes. Retrieval need is ground truth and is hidden from every online policy. SLO is a time-to-first-token budget plus a per-token budget.

Class	Mix	Privacy	TTFT (ms)	ms/token	Conf. floor κ
text	32%	none	420	18.0	0.55
rag	22%	none	620	18.0	0.72
multimodal	14%	none	700	18.5	0.62
privacy	12%	TEE and/or DP	560	19.0	0.60
agentic	11%	none	820	19.5	0.78
edge_interactive	9%	none	300	15.0	0.55

4.4 Baselines

Six deployable baselines plus a bound, all charged by the same cost model and run against the same simulator. None of them is produced by degrading POLARIS output, which is the failure that makes a comparison meaningless no matter how correct the metric is.

Cloud-only and Edge-only: fixed placement, retrieval for declared RAG requests.

Fixed-RAG, Fixed-Speculative, Fixed-Split: one configuration applied to everything.

Adaptive-Latency: adapts placement and mechanisms per request to minimize predicted latency.

It respects mandatory privacy placement and memory feasibility, and ignores cost, carbon and the confidence floor. This is the strongest latency-oriented comparator and the one that matters.

Offline oracle: true retrieval need, unbiased predictions, exact occupancy. Not deployable, reported as a bound.

All online policies share one estimator with 12% prediction error. Prediction error is a property of the (request, plan) pair rather than a fresh draw per evaluation, so a policy can't re-roll until a candidate looks good.

4.5 Statistics

Unit of analysis is the seed, not the request: generated requests inflate n without adding information. Every interval is a percentile bootstrap over 24 seeds with 4,000 resamples, and every ratio is bootstrapped as a ratio, paired by seed, rather than divided after the fact.

5 Hypotheses

Declared before the results, with thresholds:

H1. POLARIS achieves higher realized utility than every deployable baseline. Supported.

H2. POLARIS reduces confidence-floor violations relative to the latency-greedy controller by more than 25% relative. Supported.

H3. POLARIS matches or beats the latency-greedy controller on P95 latency. Not supported. It's 11.3% slower.

H4. Every mechanism in the ablation contributes measurably to utility. Not supported. Two don't.

H5. Runtime re-planning reduces mean time to recovery across all injected failure modes. Not supported. It helps in one, ties in one, and hurts in another.

Three of five fail. We report them in the abstract because a study that only reports its wins isn't a study.

6 Results

6.1 End-to-end comparison

Table 3 gives the full comparison at 12 req/s. Against cloud-only serving POLARIS cuts P95 latency 59.1% (CI [58.6, 59.5]), cost 68.0% and energy 70.2%. Those numbers are large and they're also the least interesting ones in the paper, because the ablation in Section 6.2 shows most of that gap is speculative decoding being switched on at all rather than anything to do with policy orchestration.

Isolating orchestration means comparing against Adaptive-Latency, which has the same mechanisms available and optimizes latency alone:

P95 latency: 11.3% higher for POLARIS (CI [10.0, 12.7]). P50 6.8% higher (CI [5.8, 7.7]).

Confidence-floor violations: 6.60% against 18.68%, a 64.7% relative reduction (CI [62.9, 66.4]).

Cost: 14.7% lower (CI [13.8, 15.5]). Energy 10.2% lower.

Realized utility: 0.542 against 0.473, 14.7% higher (CI [14.0, 15.4]), and within 2.7% of the oracle's 0.557.

Plan mixes explain the trade. POLARIS runs retrieval on 32% of requests and the greedy controller on 56%. POLARIS places 62% of requests off the pure-cloud path against 23%. POLARIS spends latency to buy confidence and cost, and it spends it deliberately.

One caveat on realized utility itself. Equation 5 weights latency, cost, confidence and carbon, and Adaptive-Latency ignores three of those four by construction, so H1 is partly definitional: a metric that scores confidence and cost will favour the policy that optimises them. The direction of the result is not an artefact of that, because the latency penalty and the violation reduction are each measured on their own axis and each survives all 72

sweeps. The size of the utility gap does depend on weights we chose. The experiment that would settle it is a sweep of Equation 5's weights reporting how far the latency weight has to rise before the ranking flips, which is the single number that tells a practitioner when to build which system. We have not run it, and §7.2 keeps it on the list.

One caveat on the SLA-violation column: at this operating point it's zero for every policy that uses speculative decoding, so it doesn't discriminate. Section 6.6 tightens the SLO until it does.

Table 3: End-to-end comparison at the nominal operating point, 24 seeds. Latency, cost and energy are simulated. Lower is better except utility. Intervals are 95% percentile bootstrap over seeds. The offline oracle is a bound, not a deployable policy.

Policy	P50 (ms)	P95 (ms)	Cost (mUSD)	Energy (mWh)	Conf. viol. (%)	Utility [95% CI]
Cloud-only	4066	7314	3.51	541	11.64	0.211 [0.208, 0.214]
Edge-only	18638	34984	3.83	510	11.64	0.200 [0.197, 0.203]
Fixed-RAG	4090	7374	3.89	546	5.32	0.216 [0.214, 0.219]
Fixed-Speculative	2141	3861	1.90	286	11.64	0.439 [0.435, 0.444]
Fixed-Split	4837	8641	2.92	439	11.64	0.228 [0.225, 0.231]
Adaptive-Latency	1478	2688	1.32	179	18.68	0.473 [0.468, 0.478]
POLARIS	1578	2992	1.12	161	6.60	0.542 [0.539, 0.546]
Offline-Oracle	1630	3129	1.13	166	5.32	0.557 [0.554, 0.560]

6.2 Where the gains actually come from

Table 4 removes one mechanism at a time.

Full POLARIS appears in both Table 3 and Table 4 with slightly different values: P95 2992 against 2988 ms, confidence violations 6.60% against 6.58%, utility 0.542 against 0.541. The two rows come from separate invocations of the experiment driver rather than from one. The estimator's prediction error is a deterministic function of the (request, plan) pair, but it is keyed on a hash that the interpreter salts per process, so the realised noise differs between invocations even though the seeds do not. The differences sit well inside the seed-level bootstrap interval for that row, [0.537, 0.545] on utility, so these are two draws of one quantity rather than two quantities. It is still a reproducibility defect rather than a rounding artefact, and §7.3 states the fix: the package now pins the hash seed, which makes repeated runs bit-identical.

Speculative decoding dominates. Without it utility falls from 0.541 to 0.358, a 33.8% drop, and P95 rises from 2988 ms to 5473 ms. Most of what POLARIS delivers end-to-end is the availability of that one mechanism.

Adaptive routing and edge–cloud partitioning behave the way the design says they shouldn't. Removing either lowers P95 (to 2777 ms and 2778 ms against 2988 ms) and

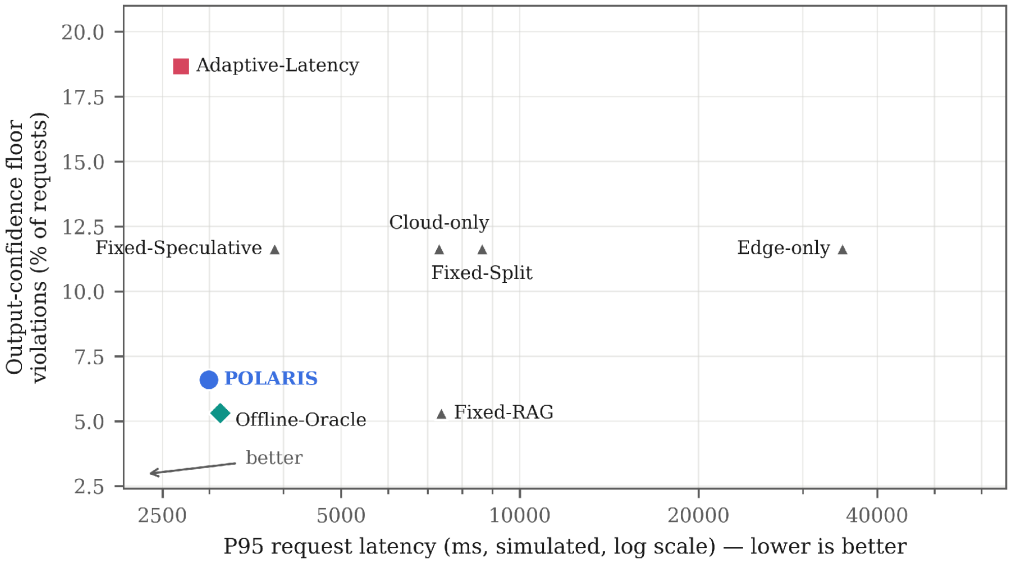


Figure 1: The trade POLARIS makes. No policy is in the bottom-left corner. The latency-greedy controller is fastest and violates the confidence floor most often; POLARIS gives up P95 latency to approach the offline oracle's compliance. Latency is simulated and on a log scale.

raises cost (to 1.326 and 1.326 mUSD against 1.118). They aren't latency mechanisms in this system. They're cost mechanisms, and their utility contribution sits inside the interval.

Carbon-aware scoring is null. Removing it gives utility 0.542 (CI [0.538, 0.546]) against 0.541 (CI [0.537, 0.545]). The intervals overlap almost completely. Under this cost model, carbon-aware scoring changes nothing you can measure, and we'd rather say that than keep a term in the objective because it sounds responsible.

Removing the cost term raises cost from 1.118 to 1.358 mUSD, which is the one ablation row that does exactly what the design predicts.

6.3 Planner overhead — measured, not modeled

Real timings, finally. Over 600 planning decisions on an Intel(R) Xeon(R) Processor @ 2.10GHz, mean planning time was 0.841 ms (CI [0.828, 0.855]), P95 1.209 ms and P99 1.397 ms, over a candidate set averaging 35 plans and never exceeding 36.

That's 0.028% of the simulated P95 request latency. The control plane isn't the bottleneck, and bounded candidate generation is why. An unbounded capability graph would not have this property.

Table 4: Ablation, one mechanism removed at a time. Rows whose utility interval contains the full-system mean are not distinguishable from it. Recovery re-planning is inactive without faults and is isolated in Table 6.

Configuration	P95 (ms)	Cost (mUSD)	Conf. viol. (%)	Utility [95% CI]
Full POLARIS	2988	1.12	6.58	0.541 [0.537, 0.545]
− speculative decoding	5473	1.96	6.46	0.358 [0.354, 0.362]
− adaptive routing	2777	1.33	6.57	0.540 [0.536, 0.543]
− edge-cloud partitioning	2778	1.33	6.57	0.540 [0.537, 0.543]
− cost objective	2927	1.36	6.58	0.541 [0.537, 0.545]
− carbon objective	2985	1.13	6.58	0.542 [0.538, 0.546]
− recovery re-planning	2988	1.12	6.58	0.541 [0.537, 0.545]

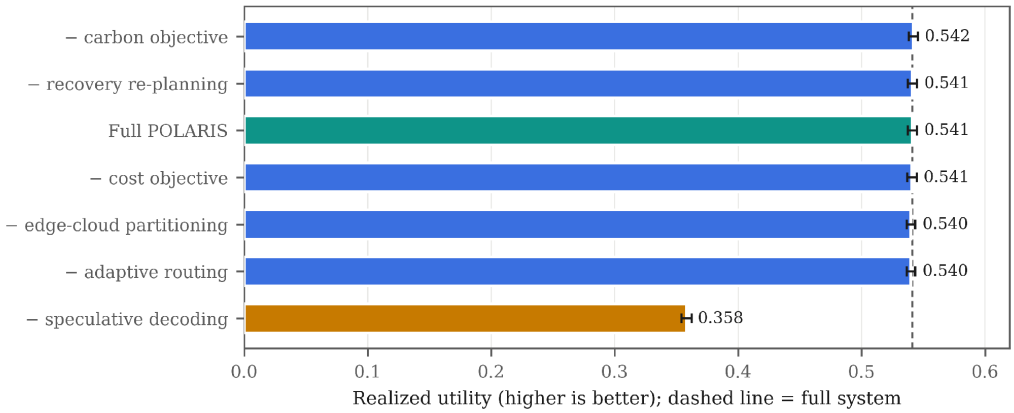


Figure 2: Ablation by realized utility with 95% bootstrap intervals. Only speculative decoding moves the result. Bars whose interval spans the full-system mean (dashed line) are not distinguishable from it.

6.4 Load scaling

Between 6 and 26 req/s, POLARIS P95 rises 22% (from 2820 to 3442 ms) while cloud-only rises 66% (to 10462 ms, with 98% of requests missing their SLO). Planning time stays flat at 0.89 ms, which follows from the candidate set being bounded by the policy vector rather than by system size.

6.5 Fault recovery

H5 fails here, and the pattern is worth more than an average.

Retrieval timeout: POLARIS recovers in 2466 ms against 7030 ms for the greedy controller, 64.9% faster. It re-plans onto a no-retrieval path for requests whose confidence floor still permits it.

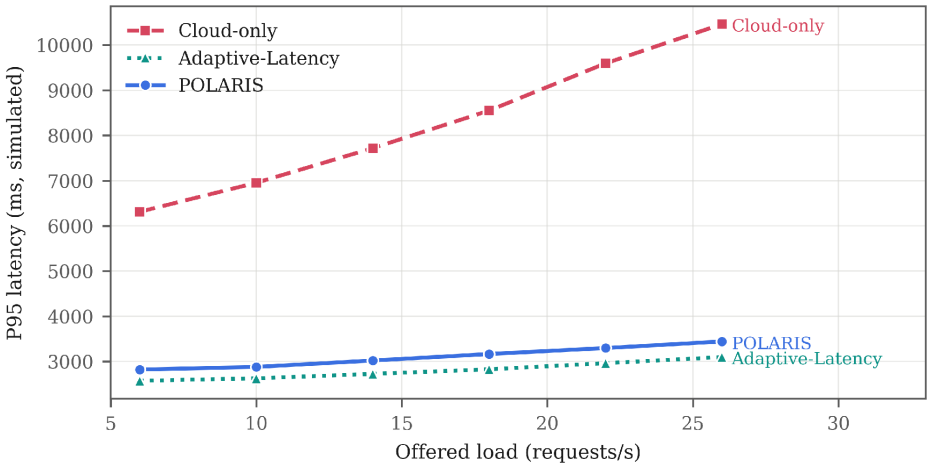


Figure 3: P95 latency against offered load. POLARIS keeps a roughly constant latency penalty against the greedy controller across the range, while cloud-only serving degrades steeply. Series carry distinct markers and dash patterns so identity survives greyscale printing.

Edge node failure: 1960 ms against 2038 ms. A tie.

GPU pool degradation: POLARIS is 76% worse, 5450 ms against 3095 ms. Policy-compliant alternatives are scarcer than latency-minimal ones, so constrained re-planning has fewer places to go. That's the direct cost of the constraint, and it's real.

TEE attestation loss: POLARIS refuses 1.42% of requests outright rather than queueing them, so its recovery time isn't comparable to the baselines' 5773 ms. We are not reporting that as a win. Refusing a request and recovering one are different outcomes, and a system that reports zero recovery time because it declined the work has not recovered anything.

Network degradation produced no failed requests under this model, so recovery time isn't defined for it. It degrades throughput rather than failing requests.

Isolating re-planning under faults (Table 6): it cuts mean recovery from 4876 ms (CI [4515, 5194]) to 3933 ms (CI [3694, 4199]), a 19.3% reduction with non-overlapping intervals. Utility is unchanged: 0.489 (CI [0.485, 0.493]) with re-planning against 0.491 (CI [0.487, 0.495]) without. Re-planning is a recovery-latency mechanism and nothing else, and if you're building one you should size the engineering effort against that alone.

One row in Table 6 deserves a note it did not get in the previous version. Re-planning raises the failed-closed rate from 1.34% to 1.51%. That is the mechanism behaving as specified rather than a regression. Without re-planning, a request whose plan fails is re-queued against the same plan and retried; with re-planning it is re-planned against a capability graph with the failed component removed, and when no policy-compliant plan survives that pruning it is refused rather than retried into a non-compliant path.

More of the affected requests therefore end as explicit refusals. It is also part of why recovery time improves while utility does not, since a refusal scores zero utility and is counted separately.

Table 5: Mean time to recovery by injected failure, 24 seeds. POLARIS is faster in one scenario, level in one and slower in one. Refusing a request and recovering one are different outcomes and are not averaged together.

Failure scenario	Cloud-only (ms)	Adaptive-Latency (ms)	POLARIS (ms)	Note
Retrieval timeout	9749	7030	2466	-65% vs greedy
Edge node failure	n/a	2038	1960	-4% vs greedy
GPU pool degradation	7219	3095	5450	+76% vs greedy
TEE attestation loss	9612	5773	refused	POLARIS failed closed on 1.42% — not comparable
Network degradation	n/a	n/a	n/a	no requests failed

Table 6: Recovery re-planning isolated under the fault schedule. It cuts recovery time with non-overlapping intervals and leaves utility unchanged.

Configuration	MTTR (ms) [95% CI]	P95 (ms)	Failed closed (%)	Utility [95% CI]
Full POLARIS	3933 [3694, 4199]	5713	1.51	0.489 [0.485, 0.493]
— recovery re-planning	4876 [4515, 5194]	5944	1.34	0.491 [0.487, 0.495]

6.6 When does the tail SLO actually bind?

At the nominal SLO shape, no speculative policy misses it. Scaling the per-token budget down, the SLO starts to bind for POLARIS at 0.45× nominal, where POLARIS violates 4.7%, the greedy controller 1.9%, fixed-speculative 64.2% and cloud-only 100.0%. POLARIS violates more than the greedy controller here, which is the same latency trade appearing in a different column.

We are reporting this because a metric that reads 0.00% for four policies is reporting a definition, not a result.

6.7 Sensitivity

Each of the 38 cost-model parameters except the two accelerator counts, which set the size of the system rather than its cost model, was swept to its low and high bound one at a time: 36 parameters, 72 configurations. All three qualitative findings hold in every one:

POLARIS retains a utility advantage over the greedy controller in 72/72 sweeps (smallest advantage 0.029).

POLARIS is slower at P95 in 72/72 sweeps (penalty between 3.6% and 20.3%).

POLARIS has fewer confidence violations in 72/72 sweeps.

No conclusion in this paper rests on a single point value of any cost-model parameter. Two constants outside that set are not swept and should be read as assumptions: the estimator's 12% prediction error and the 82% retrieval-need classifier accuracy. The second is load-bearing, since the retrieval decision is where the cost win comes from. §7.2 says what that leaves open. The latency penalty in particular holds everywhere we looked, and that's why we put it in the abstract.

7 Discussion

7.1 What policy orchestration is for

If you want the lowest P95, don't build this. Build the greedy controller, it's simpler and it's faster.

Build POLARIS when a request carries a constraint the greedy controller can't see. The 64.7% reduction in confidence-floor violations is what a policy-aware planner is for, and the 11.3% latency penalty is what it costs. Those two numbers together are the finding.

Here's the general rule under the specific case: a controller optimizing a scalar can't protect a constraint it doesn't represent, and adding the constraint as another weighted term doesn't fix it, because a weighted term is tradeable by construction. Hard constraints have to be enforced structurally, before scoring. Most adaptive-serving designs we've read put everything in the weighted sum.

7.2 Threats to validity

This is a simulation, and the cost model is the study's main assumption. 29 of 38 parameters are assumptions rather than published measurements. The sensitivity analysis covers one-at-a-time variation and does not cover interactions, so a correlated shift in several parameters could move things the sweep doesn't reach.

Both nulls are bounded claims, not proofs. The carbon term produces no measurable utility change under a workload with no hard carbon budgets and one grid-intensity pair. With a binding carbon constraint or a wider regional spread it would very likely matter. The honest statement is that we didn't test the condition under which the mechanism is supposed to pay.

Processor sharing with a linear inflation factor is a coarse model of continuous batching, and it's the assumption most likely to be responsible for the latency ordering between POLARIS and the greedy controller.

The confidence model is the weakest part. Output confidence is modeled as an additive function of retrieval relevance, quantization and DP noise. Real confidence isn't additive and isn't observable at plan time. A production system would need a calibrated estimator, and that estimator's error would change these results.

Single-region, single model family. No multi-tenant interference, no LoRA adapters, no heterogeneous accelerator classes within a tier.

Three chosen constants are not covered by that sweep and each is load-bearing. The first is Equation 5's weight vector: realized utility is not a neutral yardstick between a policy that optimises four terms and one that optimises one, so the utility ranking is partly a statement about those weights. The second and third are the estimator's 12% prediction error and the 82% retrieval-need classifier accuracy. A worse classifier would erode the cost advantage directly, because the retrieval decision is where that advantage comes from, and a worse estimator would hurt POLARIS more than the greedy controller because POLARIS scores more candidate plans against it. Sweeping all three is the first thing we would add, and until it is done those three numbers are assumptions with results resting on them rather than results.

7.3 Reproducibility

Simulator, cost model with per-parameter provenance, experiment driver and `results.json` all sit in the reproducibility package, submitted with this paper as `polaris_reproducibility.tar.gz` and archived at the DOI given in the camera-ready version. Every table in this paper is generated from that file. The README states what is measured, what is modeled, and what isn't claimed. One defect found during revision is fixed there: the estimator's per-(request, plan) error was keyed on a hash the interpreter salts per process, so two invocations of the driver produced slightly different numbers from identical seeds. The driver now pins that seed, and repeated runs are bit-identical. §6.2 notes where the unpinned behaviour is visible in this paper's tables.

7.4 Practical implications

The decision this paper is meant to inform is which controller to build, and it reduces to one question: does a request carry a constraint the scheduler cannot see? If every request is interchangeable and the only target is tail latency, the greedy controller is the right answer, and this paper's contribution is to say so with a number attached, 11.3% at P95. If requests carry per-request confidence floors, privacy labels or budgets, the greedy controller cannot protect them at any weight setting, and 11.3% of P95 is what protection costs here.

Three practical consequences follow for anyone building the second kind of system. Enforce hard constraints by pruning the plan space, not by adding a term to the objective, because a weighted term is tradeable and a pruned plan is not. Bound candidate genera-

tion by the policy vector rather than by system size, which is why planning stays at 0.841 ms and flat under load. And size the engineering effort on recovery re-planning against recovery time alone, at 19.3%, because it buys nothing on utility. The mechanisms worth building are not the ones with the largest end-to-end numbers next to them.

8 Conclusion

POLARIS treats language-model serving as a constrained composition problem: derive a policy vector, prune a capability graph to plans that satisfy the hard constraints, then score what survives.

Measured against a latency-greedy adaptive controller with the same mechanisms available, it's 11.3% slower at P95 and cuts confidence-floor violations 64.7%, cost 14.7% and energy 10.2%, landing within 2.7% of an offline oracle's utility, at 0.841 ms of measured planning cost per request. The latency penalty holds across all 72 parameter sweeps.

Two of its mechanisms don't earn their place. Carbon-aware scoring changes nothing measurable under this workload, which carries no binding carbon budget and one grid-intensity pair, so that null bounds the mechanism rather than refuting it; and recovery re-planning improves recovery time without improving utility. Most of the end-to-end gain over cloud-only serving comes from speculative decoding rather than from orchestration.

What we'd want next is the experiment this study couldn't run: the same policy comparison on a deployed serving stack, with a calibrated confidence estimator and a genuinely binding carbon budget. Those are the two conditions under which the design's remaining claims either pay or don't, and simulation can't settle either one.

Acknowledgments and Disclosures

Corporate disclaimer. Views expressed are the authors' own and not those of their employers.

Conflicts of interest. The authors declare no competing financial or non-financial interests.

Funding. This work received no external funding.

Author contributions. S.R. designed the system, built the simulator and cost model, ran and audited the experiments, and drafted the manuscript. All authors contributed to the framing of the problem and to the review of the cost model, the analysis and the manuscript.

Data and code availability. The simulator, cost model, experiment driver, figure code and every results file accompany this submission as `polaris_reproducibility.tar.gz`. Every table and every percentage in Section 6 is generated from those result files rather

than retyped, and the driver's random seeds and hash seed are pinned so that a re-run reproduces them exactly.

Use of AI assistance. Generative AI tools were used for drafting and editing assistance and for reviewing code. Every experiment, result and claim was produced and verified by the authors, who take full responsibility for the content.

References

[1] Yu, G.-I., Jeong, J.S., Kim, G.-W., Kim, S., Chun, B.-G.: Orca: A distributed serving system for transformer-based generative models. USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2022).

[2] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C.H., Gonzalez, J.E., Zhang, H., Stoica, I.: Efficient memory management for large language model serving with PagedAttention. ACM Symposium on Operating Systems Principles (SOSP) (2023). https://doi.org/10.1145/3600006.3613165

[3] Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., Zhang, C.: FlexGen: High-throughput generative inference of large language models with a single GPU. International Conference on Machine Learning (ICML) (2023).

[4] Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J.E., Stoica, I.: AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2023).

[5] Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., Zhang, H.: DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2024).

[6] Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., Bianchini, R.: Splitwise: Efficient generative LLM inference using phase splitting. ACM/IEEE International Symposium on Computer Architecture (ISCA) (2024).

[7] Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B.S., Tumanov, A., Ramjee, R.: Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2024).

[8] Zhao, D., Xian, C., Zong, Z., Jin, Y., Zeng, H., Zhou, J., Li, K., Zhai, J.: GoodServe: A SLO-aware scheduling framework for improving goodput in LLM inference serving.

IEEE Transactions on Services Computing, pp. 1–18 (2026). https://doi.org/10.1109/TSC.2026.3732701

[9] Liao, J., Dong, Q., Liang, Y., Zhou, Z., Chen, X.: Laser: Unlocking layer-level scheduling for efficient multi-SLO LLM serving. Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 509–521 (2026). https://doi.org/10.1145/3774934.3786413

[10] Li, Z., Chen, Z., Delacourt, R., Oliaro, G., Wang, Z., Chen, Q., Lin, S., Yang, A., Zhang, Z., Chen, Z., Lai, Y., Cheng, X., Miao, X., Jia, Z.: AdaServe: Accelerating multi-SLO LLM serving with SLO-customized speculative decoding. Proceedings of the 21st European Conference on Computer Systems (EuroSys), pp. 36–54 (2026). https://doi.org/10.1145/3767295.3769315

[11] Chen, L., Ye, Z., Wu, Y., Zhuo, D., Ceze, L., Krishnamurthy, A.: Punica: Multi-tenant LoRA serving. Conference on Machine Learning and Systems (MLSys) (2024).

[12] Adnan, M., Arunkumar, A., Jain, G., Nair, P.J., Soloveychik, I., Kamath, P.: Keyformer: KV cache reduction through key tokens selection for efficient generative inference. Conference on Machine Learning and Systems (MLSys) (2024).

[13] Dao, T., Fu, D.Y., Ermon, S., Rudra, A., Ré, C.: FlashAttention: Fast and memory-efficient exact attention with IO-awareness. Advances in Neural Information Processing Systems (NeurIPS) (2022). https://doi.org/10.52202/068431-1189

[14] Dao, T.: FlashAttention-2: Faster attention with better parallelism and work partitioning. International Conference on Learning Representations (ICLR) (2024).

[15] Yu, Y., Gan, Y., Sarda, N., Tsai, L., Shen, J., Zhou, Y., Krishnamurthy, A., Lai, F., Levy, H., Culler, D.: IC-Cache: Efficient large language model serving via in-context caching. Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP), pp. 375–398 (2025). https://doi.org/10.1145/3731569.3764829

[16] Zhang, B., Zhou, D., Huang, Y., Song, S., Feng, H., Jia, J., Zhang, C., Zhang, Z.: Near-zero cost KV cache compression for large language model inference. 2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 612–625 (2026). https://doi.org/10.1109/IPDPS65963.2026.00058

[17] Leviathan, Y., Kalman, M., Matias, Y.: Fast inference from transformers via speculative decoding. International Conference on Machine Learning (ICML) (2023).

[18] Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J.D., Chen, D., Dao, T.: Medusa: Simple LLM inference acceleration framework with multiple decoding heads. International Conference on Machine Learning (ICML) (2024).

[19] Li, Y., Wei, F., Zhang, C., Zhang, H.: EAGLE: Speculative sampling requires rethinking feature uncertainty. International Conference on Machine Learning (ICML) (2024).

[20] Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., Han, S.: SmoothQuant: Accurate and efficient post-training quantization for large language models. International Conference on Machine Learning (ICML) (2023).

[21] Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., Gan, C., Han, S.: AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. Conference on Machine Learning and Systems (MLSys) (2024).

[22] Frantar, E., Ashkboos, S., Hoefler, T., Alistarh, D.: GPTQ: Accurate post-training quantization for generative pre-trained transformers. International Conference on Learning Representations (ICLR) (2023).

[23] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems (NeurIPS) (2020).

[24] Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., Yih, W.: Dense passage retrieval for open-domain question answering. Conference on Empirical Methods in Natural Language Processing (EMNLP) (2020). https://doi.org/10.18653/v1/2020.emnlp-main.550

[25] Khattab, O., Zaharia, M.: ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. ACM SIGIR Conference on Research and Development in Information Retrieval (2020).

[26] Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with GPUs. IEEE Transactions on Big Data (2021). https://doi.org/10.1109/TBDATA.2019.2921572

[27] Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence (2020). https://doi.org/10.1109/TPAMI.2018.2889473

[28] Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. Foundations and Trends in Theoretical Computer Science, Now Publishers (2014). https://doi.org/10.1561/9781601988195

[29] Abadi, M., Chu, A., Goodfellow, I., McMahan, H.B., Mironov, I., Talwar, K., Zhang, L.: Deep learning with differential privacy. ACM Conference on Computer and Communications Security (CCS) (2016). https://doi.org/10.1145/2976749.2978318

[30] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: ReAct: Synergizing reasoning and acting in language models. International Conference on Learning Representations (ICLR) (2023).

[31] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., Scialom, T.: Toolformer: Language models can teach themselves to use tools. Advances in Neural Information Processing Systems (NeurIPS) (2023). https://doi.org/10.52202/075280-2997

[32] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems (NeurIPS) (2023). https://doi.org/10.52202/075280-0377

[33] Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A. et al.: Constitutional AI: Harmlessness from AI feedback. arXiv:2212.08073 (2022).

[34] Baek, J., Ji, W., Yoo, S., Kim, J.: Hot-cold tiering of HBM and high bandwidth flash for agentic LLM serving. IEEE Computer Architecture Letters, pp. 1–4 (2026). https://doi.org/10.1109/LCA.2026.3729099

[35] Schwartz, R., Dodge, J., Smith, N.A., Etzioni, O.: Green AI. Communications of the ACM (2020).

[36] Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., Dean, J.: Carbon emissions and large neural network training. arXiv:2104.10350 (2021).

[37] Smirnova, D., Nasiri, H., Adamska, M., Yu, Z., Garraghan, P.: Cost-aware model orchestration for LLM-based systems. Proceedings of the Sixth European Workshop on Machine Learning and Systems (EuroMLSys), pp. 417–425 (2026). https://doi.org/10.1145/3805621.3807624