Transactions on Computer Science and Intelligent Systems Research
ISSN: 2960-1800, eISSN: 2960-2238 | Volume 14
TPCEE 2026

HiPlan: Tail-Latency-Aware Hybrid Embedded–Remote Placement for Conditional Inference Pipelines

Jin Wang [1], Bingjie Zi [2, *]

[1] University of California, Los Angeles, USA

[2] Northeastern University, Boston, USA

* Corresponding Author Email: zi.bi@northeastern.edu

Abstract. Production machine-learning services execute directed acyclic graphs of models rather than single models, and every stage of such a graph can be deployed in one of two ways: as an independently scaled remote model server, or embedded in the orchestrator process. We profile eight real ONNX Runtime operators spanning 0.005–58 ms of service time and 0.1–291 MB of resident footprint, and show that the choice is not global: remote invocation inflates the latency of cheap operators by 10–40× while costing under 1% for expensive ones, and the tail latency saved per megabyte of orchestrator memory varies across operators by three orders of magnitude. We formalise per-stage placement as tail-latency minimisation over a conditional DAG under a memory budget and a core budget, prove it NP-hard, and identify the structural property that makes it tractable: all stages couple only through one scalar, the in-process load imposed on the orchestrator tier. Conditioning on that scalar makes the objective separable, so each load level yields a two-constraint knapsack solved exactly by dynamic programming; a DAG-aware local search then corrects the non-critical-path branches that a separable objective over-values. HiPlan lands within 0.09% of exhaustive search on average while planning a 20-stage pipeline 66× faster, and an online controller re-plans under load shift while charging the measured model-load cost. Driven by two production Azure LLM inference traces and measured service-time distributions, HiPlan cuts end-to-end p95 latency by 32–58% against always-remote serving and keeps SLO violations at 3.0% where memory-greedy embedding reaches 55.5%.

Keywords: Model serving, inference pipelines, tail latency, Kubernetes, operator placement, autoscaling.

1. Introduction

A production prediction service is rarely one model. A travel or e-commerce ranking request passes through query understanding, several feature transforms, a tree ensemble, an occasionally invoked neural re-ranker and a set of business-policy stages; an agent request fans out to parallel retrieval scorers before a conditional cross-encoder. The orchestration layer that walks this graph is, in current Kubernetes-native platforms, separate from the models themselves: each stage is a model server behind its own service endpoint, autoscaled independently, and reached over REST or gRPC. This is the deployment idiom of KServe, Seldon Core and TorchServe, and it is what systems such as Clipper and InferLine assume when they provision pipelines [1], [2]. Recent work demonstrates that adaptive intelligence can be combined effectively with explicit optimization guarantees: LLMSched couples foundation-model reasoning with an ILP refinement layer and reports substantial reductions in job completion time and SLA violations on production cluster traces [3].

The idiom has an obvious cost. A stage whose arithmetic takes five microseconds still pays a full remote procedure call. We measure this directly: a logistic-regression policy stage runs in 0.005 ms in process and 0.183 ms through a loopback HTTP model server, a 37× inflation, while a six-layer transformer encoder at 57.9 ms pays 1% for the same call. Real pipelines are full of the former. The alternative — loading the model artifact into the orchestrator process and invoking it in process, which we call embedded serving — removes the call entirely, but is not free either: the model's weights occupy the orchestrator's memory, its arithmetic runs on the orchestrator's CPU, and it can no longer be scaled independently of the graph walker.

Content from this work may be used under the terms of CC BY-NC 4.0 licence (https://creativecommons.org/licenses/by-nc/4.0/).
Published by Warwick Evans Publishing.

Neither extreme is right. Embedding everything is infeasible under any realistic memory budget — our eight operators total 469 MB of marginal resident set — and, more subtly, it destabilises the orchestrator: embedding a single 2.3 ms encoder into a pipeline running at 2400 req/s drives p95 latency from 4.6 ms to 444 ms because the orchestrator tier saturates. Embedding nothing leaves 32–58% of the tail on the table. The decision is per stage, and it is load dependent. A related hardware-aware inference line reaches the same broader systems conclusion that static execution choices leave significant performance untapped: Neuro-Elastic jointly adapts precision and sparsity to input difficulty and device state, cutting latency and energy while preserving high predictive accuracy [4].

What makes the per-stage decision interesting is that the benefit and the cost are almost uncorrelated. The quantity that matters to a memory-constrained planner is the tail latency saved per megabyte embedded. Across our operators this spans three orders of magnitude: a 300-tree random forest occupies 291 MB and saves 0.19 ms, while a two-layer feature MLP occupies 1.8 MB and saves 0.20 ms. A planner that orders stages by latency saved, or by model size, or by the classical knapsack ratio, gets this wrong — and gets it catastrophically wrong once the orchestrator's own queue is taken into account, because the ratio rule is blind to the load a stage imports into the shared tier.

This paper makes four contributions.

A measurement study of the embedded–remote trade-off. We build eight real ONNX graphs — tree ensembles and an MLP trained on UCI Adult, a convolutional text encoder, and three transformer encoder stacks — and measure, on real hardware, in-process latency, remote latency over binary and JSON transports, marginal resident footprint, and cold-load time (Section II). The marginal footprint, not the standalone one, is what a placement plan must charge, and it differs by up to 2.3×.

A tractable formulation. We cast placement as tail-latency minimisation over a conditional DAG under a memory budget and a core budget, show it is NP-hard by reduction from the knapsack problem, and prove that all stages interact only through the scalar in-process load offered to the orchestrator tier (Section III).

The HiPlan planner. Conditioning on that scalar makes the objective separable, so every load level becomes a two-constraint 0/1 knapsack solved exactly by dynamic programming; a bounded DAG-aware local search then repairs the error a separable objective makes inside fan-out groups, where embedding a non-critical branch buys nothing. An online controller re-plans under load shift using a burst-aware rate predictor and charges the measured model-load time of newly embedded stages (Section IV).

An evaluation on production traces. Driven by the Azure LLM inference traces of Patel et al. [5] and by the measured service-time distributions, HiPlan reduces p95 latency by 32–58% relative to always-remote serving, reaches the latency of full embedding with 32% less memory, stays within 0.09% of the exhaustive optimum, and holds SLO violations at 3.3% through a traffic surge that drives a static plan to 44.4% (Section V).

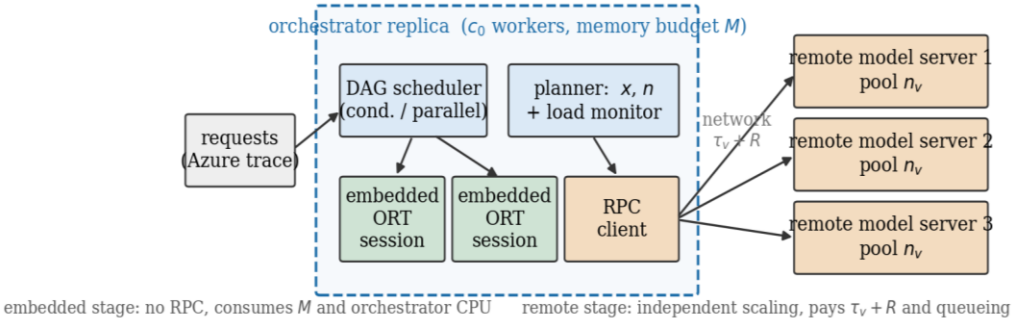

Figure 1. Hybrid orchestration. Each stage is either embedded in the orchestrator replica, where it consumes the memory budget M and the shared worker pool, or served remotely by its own pool, where it pays τ_v + R and its own queueing delay.

2. The Embedded–Remote Trade-Off

2.1 Hybrid Orchestration

Fig. 1 shows the execution model. A request enters an orchestrator replica, which walks the pipeline DAG. For each stage the orchestrator either runs the model in process on a loaded ONNX Runtime session, or issues a non-blocking remote call to a pool of model servers dedicated to that stage. Remote calls do not hold an orchestrator worker, so the orchestrator tier is loaded only by graph bookkeeping and by whatever arithmetic has been embedded into it. Two budgets bound the system: each orchestrator replica has a memory budget M available for embedded model artifacts, and the deployment as a whole has a core budget C shared between the orchestrator tier and the remote pools.

This is exactly the degree of freedom that current serving stacks leave unexploited. KServe, Seldon and TorchServe place every stage remotely; monolithic in-process serving places every stage locally. The serverless literature has recognised the same tension — Faastlane runs workflow functions as threads of one process [6], SAND and Nightcore short-circuit calls between co-located functions [7], and SONIC picks data-passing methods per edge [8] — but these systems fuse functions to avoid data movement, not to trade a fixed per-replica memory budget against a shared CPU tier under a tail-latency objective.

2.2 Methodology

All numbers in this section are measured on one platform: a container with one dedicated CPU core and 3 GB of memory running ONNX Runtime 1.24 with intra- and inter-operator parallelism pinned to one thread. We build eight operators covering the shapes that appear in production pipelines:

a logistic regression, a 100-tree depth-8 random forest, a 300-tree depth-16 random forest and a 512–256 MLP, all trained on the UCI Adult census data set; a token-embedding plus 1-D convolutional text encoder whose vocabulary is the TF-IDF vocabulary fitted on the UCI SMS Spam corpus; and three pre-LayerNorm transformer encoder stacks of (d, L, seq) = (256, 2, 96), (384, 4, 128) and (512, 6, 192), emitted directly as ONNX graphs.

Embedded latency is the distribution of 400 timed in-process invocations after 30 warm-up runs. Remote latency is the distribution of 400 invocations of the same session hosted by a separate HTTP model server on the loopback interface, over a persistent connection with Nagle's algorithm disabled, measured for a binary tensor body and for a KServe-v2-style JSON body. Marginal footprint is the resident-set increment caused by adding the operator to a process that already hosts an ONNX Runtime session, measured in a fresh process per operator so that allocator high-water marks cannot mask it. Cold-load time is the session-construction time in that same process. This operator-aware methodology is consistent with CAGR, which shows that graph-level inference optimization can exploit hardware-specific operator characteristics to deliver 1.8–3.2× speedups across heterogeneous accelerators while retaining strong performance portability [9].

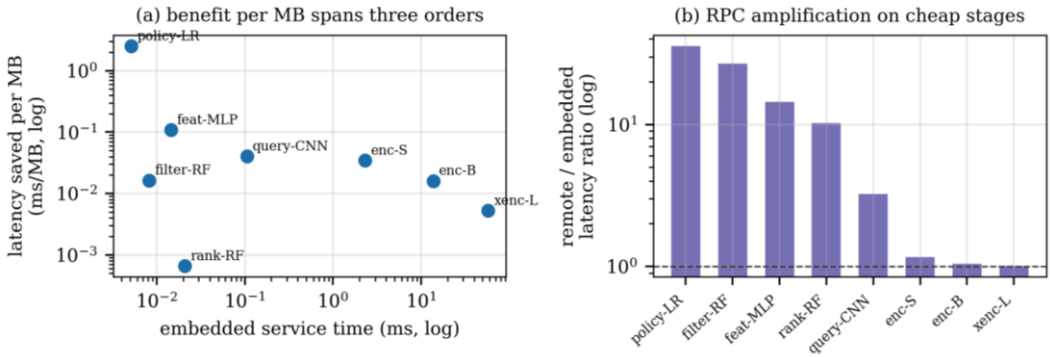

Figure 2. Measured operator trade-off. (a) Tail latency saved per marginal megabyte against in-process service time; the spread across operators is three orders of magnitude. (b) Ratio of remote to embedded latency: 37× for the cheapest stage, 1.01× for the most expensive.

Table I. Measured operator profiles (latencies in ms; marginal footprint is the resident-set increment inside a process already hosting an ONNX Runtime session)

Operator (architecture / training data)	File MB	Marg. MB	Load ms	Emb. p50	Emb. p95	Rem. bin	Rem. JSON
policy-LR (logistic reg. / Adult)	0.00	0.1	0.5	0.005	0.006	0.183	0.196
filter-RF (RF 100×d8 / Adult)	0.98	12.8	17.9	0.008	0.010	0.219	0.258
feat-MLP (512-256 / Adult)	0.53	1.8	2.7	0.015	0.023	0.210	0.252
rank-RF (RF 300×d16 / Adult)	27.61	290.9	1171.3	0.021	0.028	0.214	0.246
query-CNN (emb+Conv1D / SMS)	4.76	5.9	9.7	0.106	0.132	0.344	0.376
enc-S (transf. 256/2/96)	6.01	10.8	24.0	2.325	2.485	2.698	21.497
enc-B (transf. 384/4/128)	27.03	36.1	107.8	13.769	15.012	14.348	53.552
xenc-L (transf. 512/6/192)	72.05	110.6	352.6	57.886	63.917	58.470	137.301

Table II. Transport cost against payload size (ms)

Tensor	Bytes	Binary p50	Binary p99	JSON p50	JSON encode
16 floats	64 B	0.138	0.192	0.139	0.010
256	1.0 kB	0.140	0.453	0.151	0.155
4096	16 kB	0.167	0.820	0.173	2.596
16384	64 kB	0.166	0.874	0.205	8.064
65536	256 kB	0.177	0.224	0.748	31.738
262144	1.0 MB	0.587	0.721	2.279	137.395

2.3 Findings

Table I collects the profiles. Five observations drive the rest of the paper.

F1: remote invocation costs a near-constant 0.18–0.6 ms. Across five orders of magnitude of model cost the measured overhead of a remote call is flat, which is what one expects of a transport-bound path. Table II isolates it on an echo server: 0.138 ms for a 64 B body, still only 0.587 ms at 1 MB.

F2: the same overhead is 37× or 1% depending on the stage. The policy stage pays 0.183 ms on 0.005 ms of arithmetic; the large cross-encoder pays 0.584 ms on 57.9 ms. Fig. 2(b) shows the ratio.

F3: benefit density spans three orders of magnitude. Fig. 2(a) plots latency saved per marginal megabyte. The 300-tree forest is the worst possible embedding candidate — 291 MB of weights for 0.19 ms — and would be selected early by any size-blind rule and late by a ratio rule, while the 1.8 MB MLP saves the same 0.20 ms.

F4: marginal footprint is not model size. The 27.6 MB serialised forest expands to 291 MB of in-memory nodes, a 10.5× inflation, while the 72 MB transformer occupies 110.6 MB. Planning against file size would over-commit memory by an order of magnitude on one operator and under-commit on another.

F5: JSON transport is not a viable default for tensor stages. Serving the 384 kB cross-encoder input as a KServe-v2 JSON body raises the call from 58.5 ms to 137.3 ms, and client-side encoding alone costs 137 ms at 1 MB (Table II). All remaining experiments use the binary transport, which is the favourable case for remote serving.

3. Problem Formulation

3.1 Pipeline and Execution Model

A pipeline is a DAG G = (V, E). Stage v runs operator o(v) and executes conditionally with probability c_v given that at least one predecessor executed; π_v denotes its unconditional reach probability, and $\Pi = \Sigma_v \pi_v$ the expected number of stages traversed. Let s_v be the service-time random variable of operator o(v) measured in Section II, m_v its marginal footprint, τ_v its measured RPC overhead and R an additional network round trip that depends on how far the pool is placed. The orchestrator tier has c_0 workers and charges ε per stage dispatch. The value of treating dependency structure as first-class scheduling information is reinforced by GART, whose GNN-based scheduler models workflows as DAGs and demonstrates strong adaptive and robust performance under heterogeneous resources, load spikes, and node failures [10].

The decision variables are the placement $x \in \{0, 1\}^{|V|}$, where $x_v = 1$ means stage v is embedded, and the replica counts n_v of the remote pools. Stage duration is

$d_v = s_v + \varepsilon + W_0(x)$ if $x_v = 1$, $d_v = s_v + \varepsilon + \tau_v + R + W_v(n_v) + W_0(x)$ otherwise,

Where W_0 is the waiting time at the orchestrator tier and W_v that at pool v. End-to-end latency L(x, n) is the critical path of the realised sub-DAG, with concurrent branches taking a maximum and the conditional configurations weighted by their probabilities; the objective is its q-th percentile L_q.

Both queues are modelled as M/G/c using the Allen–Cunneen approximation [11], with second moments taken from the measured service-time samples rather than assumed; the waiting-time percentile uses the standard exponential approximation of the conditional wait, $W_q = (\bar{W} / P_{wait}) \ln(P_{wait} / (1-q))$. Using the measured second moment matters: the orchestrator's task mix after embedding is a mixture of 0.005 ms and 2.3 ms tasks whose squared coefficient of variation exceeds 30, and a model that assumes exponential service badly under-predicts the penalty of embedding an expensive stage.

3.2 The Optimisation Problem And Its Hardness

$$\text{minimise } L_q(x, n) \text{ subject to } \Sigma_v\, x_v\, m_v \le M,\ c_0 + \Sigma_v\, (1-x_v)\, n_v \le C.$$

Proposition 1. The problem is NP-hard. Take a chain DAG with unit conditional probabilities, a single orchestrator worker so lightly loaded that $W_0 = 0$, and pools provisioned so that $W_v = 0$. The objective collapses to $\Sigma_v\, (s_v + \varepsilon) + \Sigma_v\, (1-x_v)(\tau_v + R)$; minimising it under the memory budget is exactly the 0/1 knapsack problem with profits $\tau_v + R$ and weights mv.

Proposition 2 (scalar coupling). Fix the aggregate in-process load $\theta(x) = \Sigma_v\, \pi_v\, x_v\, \bar{s}_v$. Then W_0 depends on x only through θ, and conditional on θ the remaining per-stage terms of the objective are mutually independent.

The orchestrator's offered load is $\lambda(\Pi\varepsilon + \theta)/c_0$, a function of θ alone, and no other term of d_v refers to x_u for $u \neq v$. The coupling that makes the problem non-separable is therefore one-dimensional, which is what Section IV exploits. Two caveats survive: W_0 is shared by all stages, so θ must be optimised jointly rather than eliminated; and the critical-path maximum over concurrent branches means the marginal value of embedding a stage off the critical path is zero, which no separable objective can express.

4. The Hiplan Planner

4.1 Decomposition Over Orchestrator Load Levels

HiPlan discretises θ into K levels in $[0, \Sigma_v\, \pi_v\, \bar{s}_v]$. For each level it solves

$$\max\ \Sigma_v\, x_v\, g_v \quad \text{s.t.} \quad \Sigma_v\, x_v\, m_v \le M, \quad \Sigma_v\, \pi_v\, x_v\, \bar{s}_v \le \theta,$$

With gain $g_v = \pi_v(\tau_v + R + W_v)$, the per-execution latency a remote placement would pay. This is a two-constraint 0/1 knapsack, solved exactly by dynamic programming over a grid of memory cells and load cells in $O(|V| \cdot M/\Delta_m \cdot \theta/\Delta_s)$ time. Each level's solution is then evaluated under the full model — including the W_0 that its own θ induces — and the best level is kept. The second constraint is what distinguishes HiPlan from a classical knapsack: it prices the CPU a stage imports into the shared tier, which is precisely the term that causes the ratio-greedy baseline to embed a 2.3 ms encoder and lose two orders of magnitude of tail latency.

4.2 DAG-aware Refinement

A separable objective is exact on series regions and optimistic inside fan-out groups: it credits a branch stage with its full RPC saving even when a sibling branch determines the join time. HiPlan therefore follows the dynamic program with a bounded local search over the true critical-path evaluator, sweeping all single flips and all pairwise flips among the twelve highest-gain stages until no move improves the modelled tail. The cost is a few dozen model evaluations; the benefit is measured in Section V-F, where refinement cuts the mean optimality gap from 0.76% to 0.09% and the worst case from 12.7% to 1.6%.

4.3 Replica Allocation

Given x, the remaining cores are distributed over the remote pools. Each pool first receives the smallest replica count that keeps its utilisation below 0.8; the surplus is then assigned one core at a time to the pool with the largest marginal reduction $\pi_v(W_v(n) - W_v(n+1))$. Because W_v is convex and decreasing in n, this greedy allocation is optimal for the separable sum.

4.4 Online Control

A plan is only as good as the rate it was made for, and re-planning is not free: re-materialising the 300-tree forest in a replica takes 1.17 s, during which that replica serves at reduced capacity. HiPlan's controller therefore (i) predicts the next window's rate with an exponentially smoothed level, a one-step trend extrapolation so that a ramp is anticipated rather than followed, and a safety margin proportional to the smoothed deviation; (ii) treats the two directions asymmetrically — shedding embedded stages when load rises is urgent and happens on the first window, while adding embedded stages when load falls is speculative and requires three consecutive confirming windows and a 1.6× rate margin; and (iii) accepts a new plan only if the model predicts at least a 5% tail improvement, charging the summed cold-load time of newly embedded operators against one replica of the tier.

The same predictor is used for static planning. Because a plan must survive bursts rather than averages, HiPlan sizes a static plan at the 90th percentile of the offered rate measured over 100 ms windows rather than at the mean; Section V-C quantifies what this is worth on a bursty trace. Predictive elastic scheduling provides complementary evidence that anticipatory control matters under bursty workloads: PRESAGE propagates calibrated workload forecasts into resource decisions and reports 3–5× fewer SLO violations at lower resource cost across cloud and GenAI traces [12].

5. Evaluation

5.1 Setup

Three pipelines are built from the eight profiled operators. P1-rank (9 stages) is a search-ranking graph: a query encoder, two parallel feature paths, a tree ranker, a neural re-ranker invoked on 3% of queries, and two policy stages. P2-cascade (7 stages) is a moderation cascade in which light pre-filters guard a 3% encoder and a further 15% cross-encoder. P3-agent (8 stages) fans out to four parallel scorers before a 2% cross-encoder. All three mix sequential, parallel and conditional execution.

Arrival processes come from the Azure LLM inference traces released with Splitwise [5]: the conversation trace (19366 requests, near-Poisson, index of dispersion 1.39) and the coding trace

(8819 requests, extremely bursty, index of dispersion 13.2 with second-level peaks 26× the mean). Each trace is mapped onto the simulated horizon so that its empirical burst structure is preserved, and the mean rate is set by independent thinning or by superposing randomly shifted copies. A discrete-event simulator resolves the orchestrator tier and the per-stage pools as FCFS queues and samples service times from the measured 400-point empirical distributions. Unless stated otherwise $c_0 = 8$, $C = 40$, $M = 24$ MB, $R = 0.5$ ms and the SLO is 8 ms for P1 and P3 and 2 ms for P2.

Baselines: All-Remote is the KServe idiom; All-Embedded embeds greedily by ascending footprint until M is exhausted; Greedy-Lat orders by absolute latency saved; Greedy-Ratio is the classical knapsack ratio rule and is the strongest load-oblivious competitor. Two ablations isolate HiPlan's components: HiPlan-DP omits the DAG-aware refinement, and HiPlan-mean sizes the plan at the mean rate instead of the burst-aware rate. All policies receive the same replica-allocation procedure, so they differ only in placement.

Table III. Model validation against the real orchestrator (ms)

Placement	p50 meas	p50 mdl	p95 meas	p95 mdl	p99 meas	p99 mdl
All-Remote	7.87	8.00	27.32	23.28	29.85	24.22
All-Embedded	4.82	4.86	23.66	19.33	25.07	20.27
Embed small-footprint	5.43	5.38	23.80	19.86	24.11	20.80
Embed large-footprint	7.61	7.47	26.95	22.76	27.38	23.70

5.2 Model Validation

We built a real hybrid orchestrator that executes P1 with an arbitrary placement, running embedded stages on in-process ONNX Runtime sessions and remote stages against live HTTP model servers, and measured its end-to-end latency for four placements. The measured ordering is the one the model predicts: 7.87 ms (All-Remote), 7.61 ms (embedding only the large-footprint stages), 5.43 ms (embedding only the small-footprint ones) and 4.82 ms (All-Embedded) at p50. Embedding the five cheapest stages recovers 80% of the benefit of embedding everything, at 31% of the memory.

The model has two deployment constants — per-stage orchestration overhead ε and an RPC overhead scale κ — which we fit once on the four measured medians, obtaining $\varepsilon = 0.47$ ms and $\kappa = 2.60$ for this reference orchestrator. Table III reports the fit and the held-out tail predictions: medians land within 1.8% and p95/p99 within 13–19%, with a uniform negative bias that preserves the ranking of placements. The evaluation that follows uses the directly measured primitives ($\varepsilon = 0.039$ ms from a no-op DAG walk, $\kappa = 1$) and treats R as a swept parameter, so the reported gains do not depend on the fitted constants.

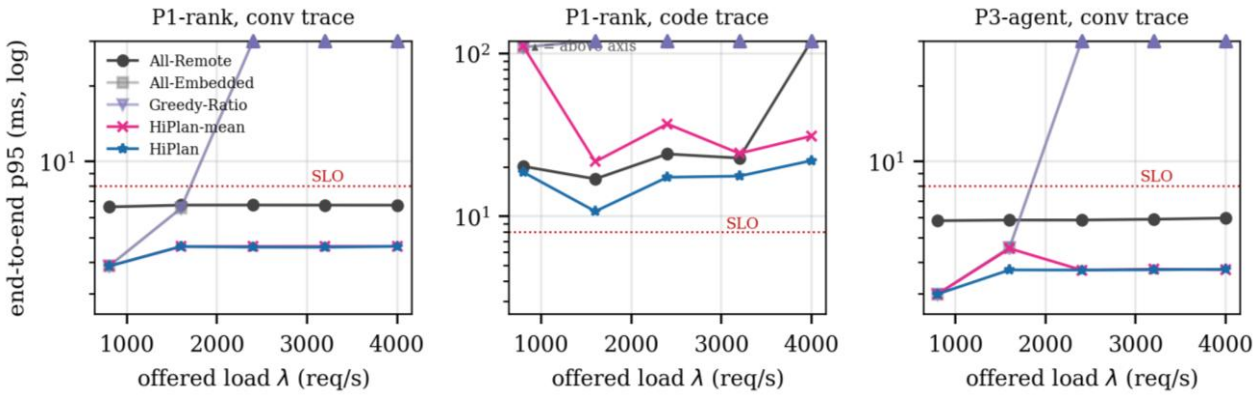

Figure 4. End-to-end p95 against offered load at $M = 24$ MB, $R = 0.5$ ms. Triangles mark points above the axis. The embedding baselines track HiPlan at low load and diverge by two to three orders of magnitude once the orchestrator tier saturates.

5.3 Latency and SLO Under Load

Fig. 4 sweeps the offered rate. On P1 with the conversation trace, All-Remote is flat at 6.6–6.7 ms p95 because its orchestrator tier stays nearly idle, while HiPlan holds 3.86–4.62 ms — a 31–42% reduction. The embedding baselines are the interesting failure: identical to HiPlan at 800 req/s, they cross All-Remote at 1600 req/s and reach 5040 ms with 100% SLO violation at 4000 req/s, because all three select the 2.3 ms encoder and saturate the shared tier. HiPlan selects the same encoder at low load and drops it as soon as the load constraint binds; the transition is visible in the plan itself, which moves from {qu, ft1, se, ft2, cal, biz} to {qu, ft1, pf, ft2, cal, biz}.

On the bursty coding trace every policy degrades, but the ranking is unchanged and the margin widens: at 4000 req/s All-Remote cannot fit its minimum pools within the core budget at all, while HiPlan delivers 21.96 ms p95. The HiPlan-mean ablation shows why burst-aware sizing matters — at 800 req/s mean rate it plans as if the peak did not exist and reaches 111.10 ms against HiPlan's 18.64 ms, a 6× difference attributable entirely to the planning rate.

Table IV reports all three pipelines at 2400 req/s. The cascade P2 is the clearest case for embedding: its light path is 0.12 ms of arithmetic spread over five stages, so remote placement multiplies its p95 by 9.9× and makes a 2 ms SLO unattainable, while embedding it costs 20.7 MB. P3 shows the value of the DAG-aware step: HiPlan and Greedy-Ratio embed the same number of stages, but HiPlan embeds the cheap filter on the critical path instead of the 2.3 ms encoder on a parallel branch, giving 3.73 ms against 269.84 ms.

Table IV. All pipelines at 2400 req/s, conversation trace (ms)

Pipeline	Policy	p50	p95	p99	SLO %	MB
P1-rank	All-Remote	6.33	6.73	20.73	3.2	0.0
	All-Embedded	13.58	443.88	456.38	55.5	20.5
	Greedy-Ratio	13.58	443.88	456.38	55.5	20.5
	HiPlan	4.22	4.59	19.31	3.0	22.5
P2-cascade	All-Remote	3.12	3.16	18.33	100.0	0.0
	Greedy-Ratio	0.28	0.32	15.52	2.8	20.7
	HiPlan	0.28	0.32	15.52	2.8	20.7
P3-agent	All-Remote	5.57	5.87	64.17	2.0	0.0
	Greedy-Ratio	6.12	269.84	278.90	46.4	18.7
	HiPlan	3.46	3.73	61.74	1.9	20.7

5.4 Memory Budget

Fig. 5(a–b) sweeps M at 2400 req/s on P1. HiPlan reaches its floor of 4.59 ms at 24 MB and does not consume more memory when more is offered; Greedy-Ratio keeps spending, taking 33 MB at M = 48, 69 MB at M = 128 and 360 MB at M = 512, and its p95 rises monotonically from 4.55 ms to 1098 ms as it does so. The comparison at M = 512 is the sharpest statement of the paper's thesis: given sixteen times the memory that HiPlan uses, a ratio-greedy planner produces a system 239× slower at the tail. Additional memory is only useful if the planner knows when to stop using it.

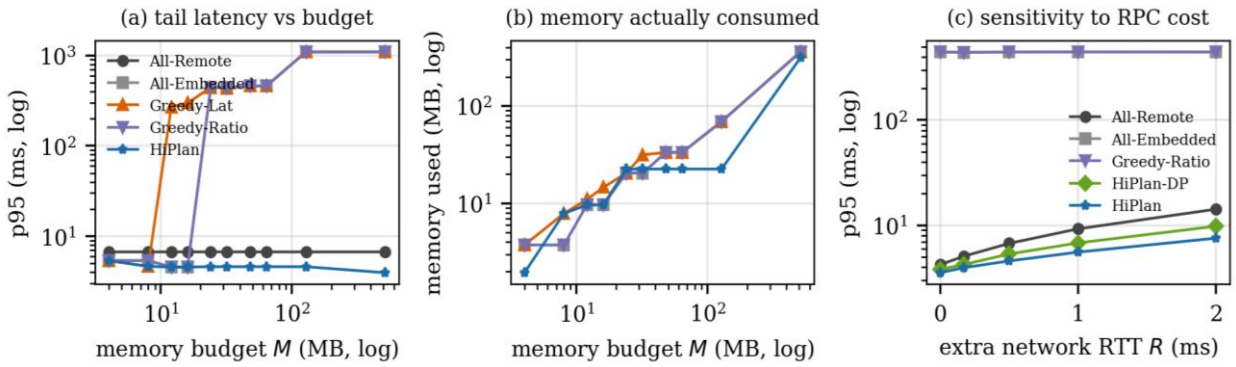

Figure 5. (a) Tail latency against the per-replica memory budget and (b) the memory actually consumed, on P1 at 2400 req/s. HiPlan stops spending at 22.5 MB; the greedy rules keep filling the budget and keep getting slower. (c) Sensitivity to the extra network round trip R.

5.5 Sensitivity to RPC Cost

Fig. 5(c) varies the extra network round trip R from 0 (same node) to 2 ms (cross-zone). HiPlan's advantage over All-Remote grows from 16% at R = 0 to 47% at R = 2 ms, which is the expected behaviour: the further apart the pools are placed, the more a per-stage placement decision is worth. The HiPlan-DP ablation also widens with R, from 7% to 30% behind full HiPlan, because a larger R raises the gain of every stage and so increases the error a separable objective makes on non-critical branches.

5.6 Optimality and Planning Cost

Over 60 random instances spanning the three pipelines, rates from 600 to 3600 req/s, budgets from 6 to 120 MB and R from 0 to 2 ms, HiPlan is 0.092% from the exhaustive optimum on average, 1.03% at the 95th percentile and 1.62% at worst (Table V). The dynamic program alone averages 0.761% with a 12.71% worst case, confirming that the refinement step earns its cost. Fig. 7 shows planning time against pipeline size on synthetic DAGs: at |V| = 20 HiPlan takes 97 ms against 6.46 s for exhaustive search, and it continues to 48 stages where enumeration is out of reach.

Table V. Distance from the exhaustive optimum over 60 instances

Planner	Gap mean	Gap p95	Gap max	Plan time
HiPlan	0.092 %	1.03 %	1.62 %	21.3 ms
HiPlan-DP (no refine)	0.761 %	3.94 %	12.71 %	8.1 ms
Exhaustive 2^|V|	0 %	0 %	0 %	44.0 ms

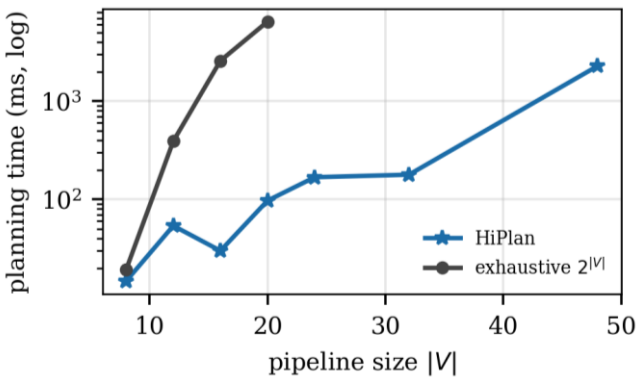

Figure 7. Planning time against pipeline size on synthetic DAGs.

5.7 Online Adaptation

Fig. 6 drives P1 through a traffic surge that rises from 420 to 2800 req/s and falls again, with the sub-second arrival structure inside each window taken from the conversation trace. Static plans are sized, as an operator would size them, from the off-peak rate. Table VI gives the outcome. The static HiPlan plan embeds the encoder and is correct until the surge arrives, after which it reaches 228.02 ms p95 and 44.43% SLO violation. All-Remote survives the surge but pays 6.33 ms p50 throughout. The adaptive controller gets both: 4.22 ms p50, 5.12 ms p95 and 3.34% SLO violation, using a single re-plan that costs 17.9 ms of model loading. The controller's asymmetry is what keeps it safe — it sheds the encoder on the first window in which the predictor indicates a rise, and does not re-embed it during the trough.

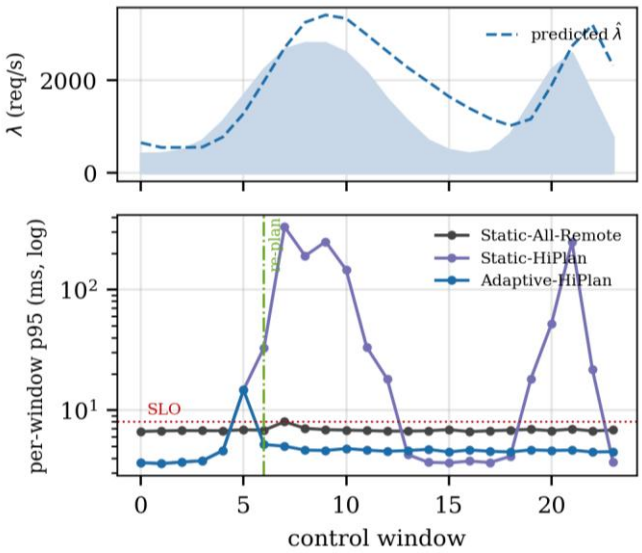

Figure 6. Traffic surge on P1. Top: offered rate and the controller's causal prediction. Bottom: per-window p95. The dash-dotted line marks the single re-plan, which sheds the encoder from the embedded set.

Table VI. Traffic surge on P1 (ms, SLO = 8 ms)

Policy	p50	p95	p99	SLO %	Re-plans
Static All-Remote	6.33	6.92	20.74	3.52	0
Static Greedy-Ratio	5.29	228.02	310.30	44.43	0
Static HiPlan (off-peak)	5.29	228.02	310.30	44.43	0
Adaptive HiPlan	4.22	5.12	19.25	3.34	1

6. Related Work

Prediction-serving systems. Clipper introduced latency-aware batching and caching for single models [1]; InferLine provisions pipeline stages to meet end-to-end tail objectives using stage-wise profiling and discrete-event simulation [2]; Clockwork obtains predictable tails by removing choice from the bottom of the stack [13]. INFaaS and Cocktail select model variants and autoscale them [14], [15], Llama auto-tunes video-analytics pipelines [16], MArk trades cloud service types against SLOs [17], and SHEPHERD exploits request-level multiplexing [18]. All of these keep each model behind its own serving endpoint; HiPlan instead asks which stages should have no endpoint at all, and shows that the answer changes with load.

Serverless fusion. Faastlane executes functions of a workflow as threads of one process [6], Nightcore short-circuits calls between co-located microservice functions [7], SONIC chooses per-edge data-passing methods [8], and Faasm provides lightweight in-process isolation [19]. These works fuse to eliminate data movement and cold starts; none faces a per-replica memory budget against model artifacts whose footprint varies 3000-fold, and none models the tail-latency cost of importing service-time variance into a shared tier. The workload characterisations of Shahrad et al. [20] and Patel et al. [5] supply the traces used here.

Conditional execution. Early-exit networks [21] and Apparate [22] adapt how much computation a request receives. They are complementary: HiPlan takes the conditional structure as given and decides where each conditionally executed stage runs. LLM-serving systems such as Orca [23] and vLLM [24] optimise within a single large model, a regime where our F2 finding says placement barely matters; the pipelines that matter here are the ones built mostly of cheap stages. Recent within-model systems continue to push this frontier: SW-SpeedDLM combines sliding-window denoising, KV compression, and speculative acceptance to enable substantially longer diffusion-language-model contexts with higher throughput and lower peak memory [25].

7. Conclusion

Whether a model stage should run inside the orchestrator or behind a service endpoint is a per-stage, load-dependent question, and the measurements show why: a remote call is 37× the cost of a cheap stage and 1% of an expensive one, while the tail saved per megabyte varies a thousandfold across operators. Formulating the choice as tail-latency minimisation under memory and core budgets yields an NP-hard problem whose stages couple through a single scalar; conditioning on it gives an exact dynamic program per load level, and a DAG-aware refinement repairs what separability costs on parallel branches. The result stays within 0.09% of optimal, plans a 20-stage pipeline in 97 ms, cuts p95 latency by 32–58% against always-remote serving, and — with an asymmetric online controller that charges the measured cost of reloading models — holds SLO violations at 3.3% through a surge that takes a static plan to 44.4%.

1. References

[1] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., & Stoica, I. (2017). Clipper: A low-latency online prediction serving system. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (pp. 613–627). USENIX Association.

[2] Crankshaw, D., Sela, G.-E., Mo, X., Zumar, C., Stoica, I., Gonzalez, J., & Tumanov, A. (2020). InferLine: Latency-aware provisioning and scaling for prediction serving pipelines. In Proceedings of the 11th ACM Symposium on Cloud Computing (pp. 477–491). Association for Computing Machinery.

[3] Ding, G., Yang, S., Lin, H., Chen, Z., & Yang, J. S. (2026). LLM-driven adaptive cloud resource scheduling: Bridging reasoning intelligence with optimization guarantees. IEEE Open Journal of the Computer Society, 7, 560–573. https://doi.org/10.1109/OJCS.2026.3667549

[4] Shang, W., Teng, D., Chen, T., Zhang, F., & Ding, J. (2026). Neuro-Elastic: A unified framework for hardware-aware adaptive quantization and dynamic sparsity in real-time edge intent prediction. IEEE Access, 14, 111168–111187. https://doi.org/10.1109/ACCESS.2026.3715276

[5] Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., & Bianchini, R. (2024). Splitwise: Efficient generative LLM inference using phase splitting. In Proceedings of the ACM/IEEE 51st Annual International Symposium on Computer Architecture (pp. 118–132). Institute of Electrical and Electronics Engineers.

[6] Kotni, S., Nayak, A., Ganapathy, V., & Basu, A. (2021). Faastlane: Accelerating function-as-a-service workflows. In Proceedings of the USENIX Annual Technical Conference (pp. 805–820). USENIX Association.

[7] Jia, Z., & Witchel, E. (2021). Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In Proceedings of the 26th ACM International Conference on Architectural

Support for Programming Languages and Operating Systems (pp. 152–166). Association for Computing Machinery.

[8] Mahgoub, A., Shankar, K., Mitra, S., Klimovic, A., Chaterji, S., & Bagchi, S. (2021). SONIC: Application-aware data passing for chained serverless applications. In Proceedings of the USENIX Annual Technical Conference (pp. 285–301). USENIX Association.

[9] Shen, Z., Zhao, W., Wang, B., Wang, Z., & Shang, W. (2026). CAGR: A cross-accelerator graph optimization framework for efficient recommender system inference. IEEE Access, 14, 38544–38562. https://doi.org/10.1109/ACCESS.2026.3671798

[10] Yang, S., Ding, G., Chen, Z., & Yang, J. S. (2025). GART: Graph neural network-based adaptive and robust task scheduler for heterogeneous distributed computing. IEEE Access, 13, 200196–200216. https://doi.org/10.1109/ACCESS.2025.3633290

[11] Allen, A. O. (1990). Probability, statistics, and queueing theory with computer science applications (2nd ed.). Academic Press.

[12] Wang, Y., Zou, X., Lu, Z., Wei, K., & Rhee, M. (2026). PRESAGE: Uncertainty-aware predictive elastic scheduling for cloud analytics and GenAI inference. IEEE Access. https://doi.org/10.1109/ACCESS.2026.3734498

[13] Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., & Mace, J. (2020). Serving DNNs like Clockwork: Performance predictability from the bottom up. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (pp. 443–462). USENIX Association.

[14] Romero, F., Li, Q., Yadwadkar, N. J., & Kozyrakis, C. (2021). INFaaS: Automated model-less inference serving. In Proceedings of the USENIX Annual Technical Conference (pp. 397–411). USENIX Association.

[15] Gunasekaran, J. R., Mishra, C. S., Thinakaran, P., Sharma, B., Kandemir, M. T., & Das, C. R. (2022). Cocktail: A multidimensional optimization for model serving in cloud. In Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation. USENIX Association.

[16] Romero, F., Zhao, M., Yadwadkar, N. J., & Kozyrakis, C. (2021). Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In Proceedings of the ACM Symposium on Cloud Computing (pp. 1–17). Association for Computing Machinery.

[17] Zhang, C., Yu, M., Wang, W., & Yan, F. (2019). MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In Proceedings of the USENIX Annual Technical Conference (pp. 1049–1062). USENIX Association.

[18] Zhang, H., Tang, Y., Khandelwal, A., & Stoica, I. (2023). SHEPHERD: Serving DNNs in the wild. In Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (pp. 787–808). USENIX Association.

[19] Shillaker, S., & Pietzuch, P. (2020). Faasm: Lightweight isolation for efficient stateful serverless computing. In Proceedings of the USENIX Annual Technical Conference (pp. 419–433). USENIX Association.

[20] Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., & Bianchini, R. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In Proceedings of the USENIX Annual Technical Conference (pp. 205–218). USENIX Association.

[21] Teerapittayanon, S., McDanel, B., & Kung, H. T. (2016). BranchyNet: Fast inference via early exiting from deep neural networks. In Proceedings of the 23rd International Conference on Pattern Recognition (pp. 2464–2469). Institute of Electrical and Electronics Engineers.

[22] Dai, Y., Pan, R., Iyer, A., Li, K., & Netravali, R. (2024). Apparate: Rethinking early exits to tame latency-throughput tensions in ML serving. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (pp. 607–623). Association for Computing Machinery.

[23] Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., & Chun, B.-G. (2022). Orca: A distributed serving system for transformer-based generative models. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (pp. 521–538). USENIX Association.

[24] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). Efficient memory management for large language model serving with PagedAttention. In Proceedings of the 29th ACM Symposium on Operating Systems Principles (pp. 611–626). Association for Computing Machinery.

[25] Teng, D., Rhee, M., Qin, Y., Zi, B., & Liu, W. (2026). SW-SpeedDLM: Sliding window speculative decoding for diffusion language models under long context constraints. Mathematics, 14(12), Article 2137. https://doi.org/10.3390/math14122137

[26] Dean, J., & Barroso, L. A. (2013). The tail at scale. Communications of the ACM, 56(2), 74–80. https://doi.org/10.1145/2408776.2408794

[27] Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., & Soyke, J. (2017). TensorFlow-Serving: Flexible, high-performance ML serving. arXiv preprint arXiv:1712.06139.

[28] Whitt, W. (1993). Approximations for the GI/G/m queue. Production and Operations Management, 2(2), 114–161.

[29] Becker, B., & Kohavi, R. (1996). Adult. UCI Machine Learning Repository. https://doi.org/10.24432/C5XW20

[30] Almeida, T. A., Hidalgo, J. M. G., & Yamakami, A. (2011). Contributions to the study of SMS spam filtering: New collection and results. In Proceedings of the 11th ACM Symposium on Document Engineering (pp. 259–262). Association for Computing Machinery.

[31] Microsoft. (2023). Azure public dataset: Azure LLM inference trace 2023. https://github.com/Azure/AzurePublicDataset

[32] ONNX Runtime. (n.d.). https://onnxruntime.ai

[33] KServe. (n.d.). Highly scalable and standards-based model inference platform on Kubernetes. https://kserve.github.io