Dyserve: Dynamic Strategy Generation for Agent Serving

Jiayi Qian [1] **Zishen Wan** [2 3] **Hanchen Yang** [1] **Souvik Kundu** [4] **Tushar Krishna** [1]

Abstract

Agentic workflows orchestrate optional operators (verifiers, routers, retries, escalations) but commit to a fixed pipeline per task type. This is structurally suboptimal: different requests, models, and runtime conditions favor different operator subsets. Additionally, no single hand-written workflow is uniformly Pareto-optimal. We present **Dyserve**, a framework that generates the serving strategy as a *structured subgraph* over a workflow abstraction by solving an integer linear program (ILP). The ILP jointly selects per-node execution options (model, verification, speculative execution, retry) weighted by offline-profiled coefficients for service quality and latency performance. On LiveCodeBench, the generated strategies match the best-verify accuracy with $\sim 10\times$ latency reduction. Additionally, we describe an event-driven suffix-repair extension that resolves a smaller ILP over the residual workflow when high-impact runtime events fire.

1. Introduction

Large language models (LLMs) are increasingly deployed across domains such as code generation, scientific reasoning, and tool-augmented task completion. For complex tasks, however, a single LLM call is rarely sufficient: hallucinations and reasoning errors propagate without correction, the model has no native access to external state such as filesystems or live data, and a single weight set must serve as generator, verifier, and planner alike. *Agentic workflows* address these limitations by decomposing a task into a coordinated sequence of LLM calls, tool invocations, and verification steps. Such a workflow may be specified ahead of execution, either manually (Wu et al., 2024; Hong et al., 2024) or by an LLM planner (Zhang et al., 2025a; Wang et al., 2025b; Zhang et al., 2025b), in which case it takes the form of a directed acyclic graph (DAG) over operators. On the other hand, it may be constructed incrementally at runtime, as in ReAct (Yao et al., 2023) or AutoGPT (Significant Gravitas, 2023), where each step is decided only after the output of the previous step has been observed. We focus on the former setting, where the workflow's structure is known ahead of execution and is therefore amenable to optimization.

Within an agentic workflow, each node performs a distinct role (planning, retrieval, code generation, or tool invocation), and an error at any single node propagates downstream and contaminates the final output (Ro et al., 2025). Moreover, per-node accuracy is non-uniform across these roles: a model strong on one task family may underperform on another, so even a workflow that uses a single model end-to-end exhibits substantial accuracy variation across its nodes. Selective verification has accordingly emerged as the standard countermeasure, with methods such as self-refine (Madaan et al., 2023), self-consistency (Wang et al., 2023a), multi-agent debate (Du et al., 2024), and LLM-as-a-Judge (Zheng et al., 2023) each detecting and correcting errors at a node before they propagate downstream. Their cost, however, is substantial: a single verification call can increase per-node latency by as much as $29\times$ and monetary cost by as much as $53\times$ on standard benchmarks (Ro et al., 2025). Applying verification at every node compounds this overhead, while restricting it to the terminal node forfeits opportunities for earlier correction.

The accuracy and efficiency of an agent workflow are therefore intrinsically coupled: operators that improve quality (additional verifiers, larger models, longer reasoning chains) also consume more latency and more compute. The set of achievable (quality, latency cost) operating points forms a Pareto frontier, and the central problem an agent-serving system must solve is, for each request, to select and execute at a favorable point on this frontier given the available operators and the current system state.

Existing work approaches this problem from three directions, each constrained by what it omits from its decision space. *Workflow-generation methods*, including MaAS (Zhang et al., 2025a), EvoAgentX (Wang et al., 2025b), and AFlow (Zhang et al., 2025b), embed verification operators into the workflow during synthesis, with

[1]Georgia Institute of Technology, Atlanta, USA [2]Harvard University, Cambridge, USA [3]Columbia University, New York, USA [4]Intel Labs, USA. Correspondence to: Jiayi Qian <jiayiqian@gatech.edu>, Tushar Krishna <tushar@ece.gatech.edu>.

Proceedings of the 43rd International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

accuracy as the primary objective; they do not model system state, and any efficiency gains stem from a more parsimonious verification design rather than from systems-level co-optimization. *Resource-efficient serving* systems such as Murakkab (Chaudhry et al., 2025), alongside model-routing approaches (Ong et al., 2025; Wang et al., 2023b; Chen et al., 2024), address the trade-off from the systems side by providing load-aware model selection and per-call resource configuration; verification, however, is treated as fixed (often absent) and therefore lies outside their decision space, and the strategy is not revised in response to runtime events that affect output quality, such as verifier disagreement or tool failure. *Verification-aware serving*, exemplified by Sherlock (Ro et al., 2025), places verifiers selectively across the workflow and overlaps them with speculative execution (Ye et al., 2025); the placement, however, is computed offline by training a per-task vulnerability model, which limits scalability to dynamically generated workflows. Furthermore, the runtime does not monitor live system state, and the model axis (which model executes each node) is fixed by the developer. None of these directions alone reaches the Pareto frontier: accuracy and efficiency must be co-optimized over a single decision space spanning verification policy, model selection, and runtime adaptation to load and failure events.

We present **Dyserve**, a framework that addresses these gaps by treating the serving strategy as a structured optimization variable, organized around three components. ❶ **Workflow Abstraction**. Dyserve provides an abstraction for agent workflows that formalizes the model choice, verification strategy, speculation depth, and retry mechanism at each node as a co-optimizable per-node policy. ❷ **Profile-based ILP Policy Generator**. To trade per-node accuracy against latency, we cast the selection as an integer linear program (ILP) over the workflow abstraction whose objective is a Lagrangian relaxation of the accuracy-latency frontier, weighted by **offline-profiled coefficients** for accuracy and GPU-time. ❸ **Event-driven Re-planning**. At runtime, salient events such as tool failures or shifts in system load trigger *suffix repair*, in which a smaller ILP re-solves the residual workflow (without re-doing committed work) and adapts the remaining policy to the new conditions. Together, these components allow Dyserve to operate close to the accuracy-efficiency frontier, maximizing efficiency while maintaining the target accuracy under both static and dynamic workloads.

Our contributions instantiating these components are:

- A workflow abstraction that decouples per-node policy optimization from workflow synthesis and exposes explicit prerequisite, exclusivity, and action-budget constraints (Section 3.1).
- A call-structure-decomposed multi-model profiler that tags each sub-call within a verifier policy so the ILP coefficients account for per-stage GPU-time accurately, rather than treating each policy as a single black-box cost (Section 3.3).
- A residual-ILP construction for suffix repair that re-solves a smaller ILP over the not-yet-executed portion of the workflow when high-impact runtime events fire, preserving committed work and respecting prerequisite structure (Section 3.4).

Evaluated on the LiveCodeBench benchmark, Dyserve nearly matches the strongest fixed baseline's accuracy (0.74 vs. 0.76 for `27B verify-all`) at $\sim 10\times$ lower GPU-time (37.2 s vs. 383.4 s) and lies on a more favorable accuracy-latency trade-off than any single-axis approach; ablations show that topology-aware vulnerability weighting is the dominant lever and that joint (model, verifier) optimization is required to reach this frontier.

2. Motivation

2.1. How Big Is the Oracle Gap?

For a given request, an oracle that knows the per-(model, verifier) accuracy and latency profile at each node can identify the configuration that lies on the accuracy-latency Pareto frontier. Two questions follow: how large is the configuration space, and how much does the oracle gain over fixed baselines?

Configuration space. With K choice nodes per workflow, M candidate execution models, and V verification policies per node, the number of joint configurations is $(M \cdot V)^K$. With $M{=}2$, $V{=}8$, and a typical workflow of $K{=}6$ choice nodes, this evaluates to $16^6 \approx 1.7 \times 10^7$ configurations. Each configuration must be evaluated by a full workflow execution; at $\sim$2 minutes per execution on our profiling benchmark, exhaustive enumeration is infeasible even for a single workflow.

The oracle gap. To quantify the headroom over fixed baselines, we profile all per-node configurations on a representative subset of LiveCodeBench (medium and hard) and compare the per-node oracle against four single-model baselines. Figure 1 reports the (accuracy, latency) operating points across five configurations: `4B base` and `27B base` pin a single model at every node with no verification; `4B verify-all` and `27B verify-all` attach the per-node-type accuracy-maximizing verifier to each node; and `Ground Truth` selects (model, verifier) per node from the full configuration space. The oracle matches the highest baseline accuracy (0.76, achieved by `27B verify-all`) at roughly $10\times$ lower latency (35.7 s vs. 383.4 s), and dominates each of the other three baselines on both axes. The gap is large: there is substantial headroom between any fixed strategy and the per-request optimum.

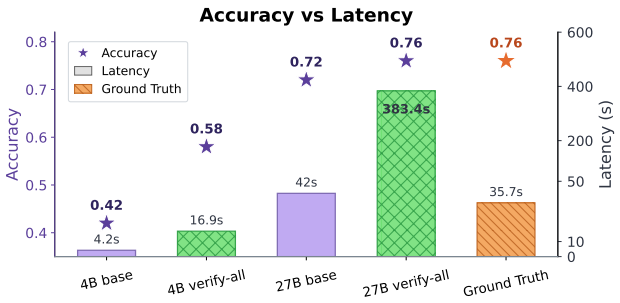

Figure 1. Accuracy and latency on LiveCodeBench (hard) for four baselines and the per-node oracle. `4B base` and `27B base` pin LLM nodes with no verification; `4B verify-all` and `27B verify-all` pin the model and attach to each node the verification policy that maximizes accuracy at that node's type; `Ground Truth` selects (model, verifier) per node from the full configuration space. Ground Truth matches `27B verify-all`'s accuracy at $\sim$10$\times$ lower latency (35.7 s vs. 383.4 s).

2.2. Why Do We Need Coupled Optimization?

The oracle in Figure 1 makes two kinds of per-node decisions: which model to assign, and which verifier (if any) to attach. These two decisions cannot be made independently since not every node in an agentic workflow contributes equally to final-output correctness. We adopt Sherlock's notion of node *vulnerability* (Ro et al., 2025): the marginal impact on terminal accuracy of an error at that node. High-vulnerability nodes (for example, the synthesis step in a code-generation workflow) demand strong correctness; low-vulnerability ones (for example, a retrieval rewrite that downstream stages can re-issue) can tolerate cheaper, less reliable execution.

Two operator-level levers raise a node's effective accuracy: route it to a more capable model, or attach a verifier. Both consume latency, and both target the same per-node objective. The two axes therefore act as *substitutes* on the accuracy side and as *complements* on the latency-cost side. A high-vulnerability node served by a strong model alone may yield accuracy comparable to the same node served by a smaller model with multi-round debate, at very different latency-cost profiles, and the better choice depends on the per-(model, verifier, task) profile. Conversely, a low-vulnerability node may be served sufficiently by a small model alone, where any verifier is wasted overhead. A model-only optimizer cannot recognize either substitution and tends to pin a strong model on every node; a verifier-only optimizer attaches verifiers where no correction is needed. The two axes must therefore be co-optimized per node. The co-optimization ablation in Section 4.4 confirms this empirically: single-axis ILP variants leave a measurable gap to the joint formulation (Table 3).

The argument above holds for a request in isolation. In a deployed system, the conditions under which an optimum is computed change during execution.

2.3. Why Do We Need Event-Driven Replanning?

A deployed serving system processes many requests over shared backends, and the conditions at admission rarely persist for the duration of execution. Two classes of events invalidate an admission-time strategy: load shifts that change effective per-device throughput, and node-level events (tool timeouts, verifier disagreement) that alter the cost-quality calculus on the not-yet-executed suffix. Both feed back into the ILP coefficients: throughputs $\theta_{\text{pre}}, \theta_{\text{dec}}$ enter the GPU-time coefficients $\tau_{\text{b}}, \tau_{\text{v}}$ directly (Section 3.3), and node-level events change the residual cost-quality trade-off on the suffix. Two scenarios illustrate the gain.

- **Strong-model congestion.** A concurrent surge saturates the strong-model backend; its effective throughput drops sharply while the small model's is unchanged. The residual ILP shifts remaining nodes to the small model with a verifier, recovering throughput at comparable accuracy.
- **Verifier disagreement.** A verifier flags a high-vulnerability node's output as suspect. The residual ILP allocates a stronger model or additional verification to the suffix to compensate, in places where the admission-time strategy did not schedule them.

In both cases, the runtime decision is a fresh solve under new inputs rather than a correction of the original. We adopt *suffix repair* as the mechanism (Section 3.4): at high-impact events, a smaller residual ILP re-solves the not-yet-executed portion of the workflow while committed work is preserved. The same coefficient pipeline that supplies the admission-time ILP supplies the residual ILP, now reading current system state.

Implications. The oracle gap (Section 2.1) shows the headroom is large; the axis coupling (Section 2.2) shows it requires joint per-node optimization across the model and verifier axes; and the runtime state dependence (Section 2.3) rules out a single one-shot solve at admission. A structured optimizer that solves an ILP at admission and re-solves a residual ILP on runtime events follows directly. We instantiate it in Section 3.

3. Method

Figure 2 summarizes the framework. A planner emits a draft workflow as a directed acyclic graph (DAG); we wrap it in a *workflow abstraction* that exposes optional operators as decision axes; an ILP selects a request-specific subgraph; the runtime executes the chosen subgraph. Coefficients flow into the ILP from an offline profiler and (when available) per-device throughput readings that capture current backend load.

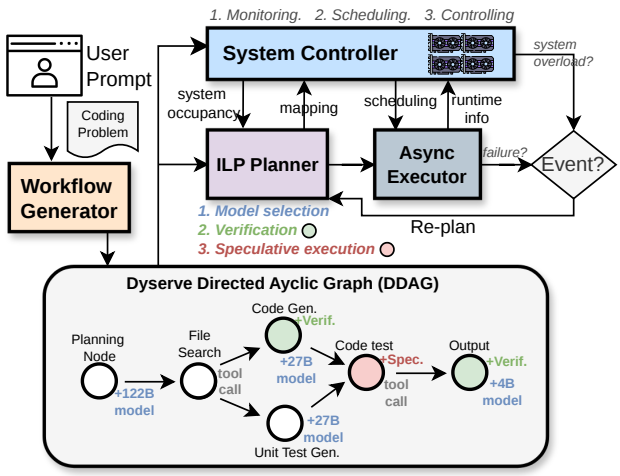

Figure 2. End-to-end architecture of Dyserve. The workflow abstraction is generated once per request from the planner-generated draft DAG. The ILP selects a serving strategy using offline profiling coefficients combined with per-device throughput readings. When high-value events are triggered, a residual ILP re-solves the remaining suffix, shown as the dashed in-progress path.

3.1. Workflow Abstraction

The workflow abstraction is a directed acyclic graph $G = (V, E)$ over operator nodes; the edges E encode the dependency ordering enforced at runtime. Each node $i \in V$ is either an *LLM node* or a *tool-call node*, and exposes four policy axes that the ILP can choose over: an execution-option set $\mathcal{O}_i$ (a (model, backend) candidate for an LLM node, or the singleton tool invocation for a tool-call node), a verification-policy set $\mathcal{P}_i$, a retry-policy set $\mathcal{R}_i$, and a speculation-policy set $\mathcal{S}_i$ (Section 3.2). LLM nodes additionally carry a *node type* $t_i \in \{\mathsf{code}, \mathsf{math}, \mathsf{reasoning}\}$ that classifies the subtask handled at the node.

The abstraction is not a hand-authored library. The planner emits a Flow-style DAG (Niu et al., 2025) per request and, at workflow-generation time, tags each LLM node with its type t_i according to the subtask it solves. The node type is the join key into the per-(model, policy, type) profile that supplies the ILP coefficients (Section 3.3). A single lookup at admission time yields the per-node accuracy and GPU-time coefficients used by the ILP.

3.2. Policy Design

A node's strategy is determined by three policy axes: a verification policy that decides how to validate the node's output, a speculation policy that decides whether to overlap downstream computation with that validation, and a retry policy that decides how to recover from failure. The ILP in Section 3.3 jointly optimizes the verification policy together with the model assignment; speculation and retry are exposed by the abstraction and handled by the runtime/executor, but are not part of the evaluated ILP in this version (Section 3.4).

Verification policies. We expose eight standard policies (`none`, self-refine and its iterative variant, parallel critique-and-rewrite, k-sample self-consistency, LLM-as-a-judge gating, gated refine, and multi-round debate) at every applicable node. Each is described by a *call structure*, a sequence of stages with one or more sub-calls per stage, that determines how token cost aggregates across the policy; the full taxonomy is in Section A. Treating a policy as a single black-box cost loses per-(model, stage) accounting, so the profiler tags each sub-call to keep per-stage statistics recoverable.

Speculation policies. A speculation policy controls whether to overlap a node's verifier with downstream execution. The simplest form runs the verifier on the critical path; richer policies fire downstream nodes speculatively in parallel with the verifier and roll back on a negative verdict.

Retry policies. A retry policy describes how to recover when a node's call fails or its output is rejected, ranging from aborting, to re-issuing the same execution option, to escalating to a stronger option from $\mathcal{O}_i$.

3.3. Profile-based ILP

We cast strategy generation as an *integer linear program* (ILP) with binary decision variables for per-node (model, verifier) assignments and continuous auxiliaries that track GPU-time makespans along the workflow DAG. ILP is a natural fit because per-node decisions are inherently discrete, the constraints induced by the workflow abstraction are linear in those binary variables, and per-call GPU-seconds (composed from offline-profiled token counts and device throughput) accumulate linearly along DAG-aligned critical paths. The formulation scales without retraining: extending the system to a new model or verification policy amounts to adding variables and linear rows to the program. Off-the-shelf solvers (Dunning et al., 2011) dispatch the program in milliseconds at the workflow sizes we encounter.

Offline profiling. The offline profile measures, for every (model, policy, node type) triple, the per-item accuracy and the device-independent token counts an LLM node of that type would consume; per-call GPU-time is composed at plan time from a separate per-device throughput sweep that records prefill and decode tok/s. Decomposing the measurement this way means adding a new serving device requires only a fresh throughput sweep, not a re-run of the full policy grid. We treat code, math, and $\mathsf{reasoning}$ as the three node-type categories and assemble datasets from eight tasks: HumanEval+ and MBPP+ for code; MATH-500, AIME24, and AIME25 for math; and GPQA-Diamond plus MMLU-Pro for $\mathsf{reasoning}$. The profile covers all eight verification policies of Section A.

Coefficient construction. For each (model m, policy p, node type t) the profile yields a measured accuracy $q(m, p, t)$ and the token counts of the base call and the verifier fan-out $(\text{base}_{\text{pre}}, \text{base}_{\text{dec}}, \text{ver}_{\text{pre}}, \text{ver}_{\text{dec}})$. A separate per-device throughput sweep yields prefill and decode tok/s $\theta_{\text{pre}}(m, d)$ and $\theta_{\text{dec}}(m, d)$, which differ by an order of magnitude on modern serving stacks. Per-node GPU-seconds for the base and verifier components compose as

$$\tau_{\text{b}}(n, m, p) = \frac{\text{base}_{\text{pre}}}{\theta_{\text{pre}}(m, d)} + \frac{\text{base}_{\text{dec}}}{\theta_{\text{dec}}(m, d)},$$
$$\tau_{\text{v}}(n, m, p) = \frac{\text{ver}_{\text{pre}}}{\theta_{\text{pre}}(m, d)} + \frac{\text{ver}_{\text{dec}}}{\theta_{\text{dec}}(m, d)}.$$

Verifier tokens are summed across all sub-calls of policy p, since each call adds its GPU-seconds to the verifier pool independently of the others. Adding a new device requires only a fresh $(\theta_{\text{pre}}, \theta_{\text{dec}})$ sweep on d'; the profile-side counts are device-independent. At admission the ILP uses the offline-measured θ; suffix repair (Section 3.4) re-reads the current effective throughput so that load-induced shifts on a backend propagate to $\tau_{\text{b}}, \tau_{\text{v}}$ and into the residual solve.

Variables. For each node n, candidate model $m \in \mathcal{M}$, and candidate verifier $p \in \mathcal{P}$:

$x_n \in \{0, 1\}$: node n activated,
$z_{n,m,p} \in \{0, 1\}$: joint (m, p) chosen at n,
$s^{\text{b}}_n,\ s^{\text{v}}_n \geq 0$: start times of n's base / verify GPU-time,
$T_{\text{b}},\ T_{\text{v}} \geq 0$: workflow makespans of base / verify pools.

Constraints. Non-optional nodes are pinned, the DAG enforces prerequisite chains, and each activated node carries exactly one (m, p) assignment:

$$x_n = 1\ (n \notin V_{\text{opt}}),\quad x_n \leq x_{\pi(n)},\quad \textstyle\sum_{m,p} z_{n,m,p} = x_n, \tag{1}$$

where $\pi(n)$ is each parent of n. A verification budget caps the fraction of nodes that may verify:

$$\textstyle\sum_{n,\,m,\,p\neq\text{none}} z_{n,m,p} \ \leq\ \lfloor |V| \cdot \rho \rfloor. \tag{2}$$

Critical-path precedence couples the binary choices to the two continuous makespans: with $\hat{\tau}_{\text{b}}(n) := \sum_{m,p} \tau_{\text{b}}(n, m, p)\, z_{n,m,p}$ (and analogously $\hat{\tau}_{\text{v}}$),

$$s^{\text{b}}_n \geq s^{\text{b}}_{\pi(n)} + \hat{\tau}_{\text{b}}(\pi(n)), \quad s^{\text{v}}_n \geq s^{\text{v}}_{\pi(n)} + \hat{\tau}_{\text{v}}(\pi(n)), \tag{3}$$

$$T_{\text{b}} \geq s^{\text{b}}_\ell + \hat{\tau}_{\text{b}}(\ell), \quad T_{\text{v}} \geq s^{\text{v}}_\ell + \hat{\tau}_{\text{v}}(\ell) \quad \forall \text{ leaves } \ell. \tag{4}$$

Objective. We maximize a topology-weighted log-quality gain minus separate penalties on the base- and verifier-pool makespans:

$$\max\ \lambda_{\text{q}} \sum_{n,m,p} w(n)\ \log\big(\max(q(m, p, t_n), \varepsilon)\big)\, z_{n,m,p} \ -\ \lambda_{\text{b}}\, T_{\text{b}}\ -\ \lambda_{\text{v}}\, T_{\text{v}}. \tag{5}$$

Algorithm 1 Strategy generation for one request.

Require: Abstraction $G{=}(V, E)$; profile Φ; throughputs $\theta_{\text{pre}}, \theta_{\text{dec}}$; weights $\lambda_{\text{q}}, \lambda_{\text{b}}, \lambda_{\text{v}}$; $w(\cdot), \rho, \varepsilon$
Ensure: Serving strategy π
for $n \in V,\ m \in \mathcal{M},\ p \in \mathcal{P}$ **do**
  $\tau_{\text{b}}, \tau_{\text{v}}, q \leftarrow \textsc{Coef}(n, m, p, \Phi, \theta_{\text{pre}}, \theta_{\text{dec}})$
end for
Build ILP with vars $\{x_n, z_{n,m,p}, s^{\text{b}}_n, s^{\text{v}}_n, T_{\text{b}}, T_{\text{v}}\}$
Add constraints (1)–(4)
Solve objective (5) via PuLP (Dunning et al., 2011)
Decode the binary z to per-node (m, p) choices π
return π

where λ_{q} scales the accuracy gain, λ_{b} and λ_{v} are per-second latency penalties on the base-model and verifier-pool GPU time respectively, and the Sherlock-style topology weights $w(n) \in \{w_{\text{root}}, w_{\text{mid}}, w_{\text{leaf}}\}$ (Ro et al., 2025) weight nodes by their topological role (root and leaf nodes carry higher vulnerability than intermediate nodes, in the sense of Section 2.2); ε floors $\log 0$. Defaults $\lambda_{\text{q}}{=}20$, $\lambda_{\text{b}}{=}0.02$, $\lambda_{\text{v}}{=}0.5$, $\rho{=}0.25$, $(w_{\text{root}}, w_{\text{mid}}, w_{\text{leaf}}){=}(0.5,\ 0.1,\ 0.4)$, $\varepsilon{=}0.01$ are fixed once by grid search over a held-out workload and reused across all experiments (Section 4.3).

Scale. The program has $\mathcal{O}(|V|{\cdot}|\mathcal{M}|{\cdot}|\mathcal{P}|)$ binary variables and $\mathcal{O}(|V|)$ continuous variables. For the workflows we generate, $|V| \leq 20$ and the ILP solves in < 100 ms per request.

3.4. Event-Driven Suffix Repair

The framework's second half is local revision at high-value runtime events. We describe the design and the validated mechanics; a full empirical evaluation is ongoing work.

When a high-value event fires (a load shift exceeding a latency-pressure threshold, a verifier disagreement, or a tool timeout), the runtime computes a *residual workflow* by removing already-executed nodes and their satisfied prerequisites, then re-solves a smaller ILP over the residual graph using the updated system state. Already-computed artifacts are preserved; the new strategy decides only the future suffix.

The residual mechanics are implemented and unit-tested: residual workflow construction respects prerequisite dependencies, frozen-node coefficients are excluded, and the suffix ILP solves correctly under the same constraint structure as the global program. What remains is the runtime integration: subscribing the executor to the event bus, populating system state from a live monitor, scoring verifier disagreement, and detecting tool failures. The evaluation for this phase is still in-progress.

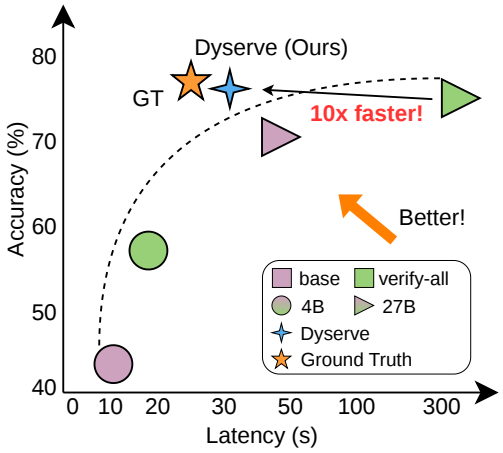

Figure 3. Accuracy versus latency on 50 LiveCodeBench problems for four single-model baselines and Dyserve. `4B base` and `27B base` pin the model with no verification; `4B verify-all` and `27B verify-all` attach the per-node-type accuracy-maximizing verifier; `Dyserve` jointly optimizes (model, verifier) per node. Dyserve nearly matches `27B verify-all`'s accuracy at $\sim 10\times$ lower latency.

3.5. Running Example

Figure 2 (lower panel) shows the strategy the ILP selects on a six-node coding workflow. Two tool nodes (*File Search*, *Code test*) are pinned; the four LLM nodes vary across models (4B/27B/122B) and verifiers. *Planning* (highest vulnerability) gets 122B alone; *Code Gen.* takes 27B with a verifier (the substitution from Section 2.2); *Unit Test Gen.* uses 27B without a verifier since *Code test* downstream checks it; and *Output* uses 4B with a verifier as a cheap last-line check. *Code test* is wrapped in speculation (red), firing *Output* in parallel to hide its latency. No single-axis optimizer reaches this strategy: jointly mixing high-tier (122B alone) and low-tier (4B + verifier) endpoints across nodes requires per-node selection over both axes.

4. Evaluation

We evaluate Dyserve on a subset of LiveCodeBench spanning medium and hard difficulty. Three experiments isolate (i) end-to-end accuracy and latency gains over single-model baselines (Section 4.2); (ii) the local sensitivity to the Lagrangian and topology weights (Section 4.3); and (iii) the necessity of joint optimization across the model and verification axes (Section 4.4).

4.1. Setup

Workflow and benchmark. We evaluate on 50 LiveCodeBench problems of medium and hard difficulty. For each problem the planner emits a per-request workflow over the abstraction of Section 3.1, and Dyserve generates its strategy on that workflow. Per-request latency cost is the total GPU-time, computed as the sum of execution times across all sub-calls (verifier calls included); accuracy is pass@1. Experiments run on GH200 machines.

Configuration	Model	Verifier
`{4B,27B} base`	pinned	none
`{4B,27B} verify-all`	pinned	per-type best
`Verify-ILP@{4B,27B}`	pinned	ILP
`Model-ILP`	ILP	per-type best
`Dyserve`	ILP	ILP

Table 1. Configurations compared. *pinned*: fixed model. *ILP*: per-node Dyserve optimization. *per-type best*: the per-node-type accuracy-maximizing verifier from the offline profile.

Models and verification policies. The execution-option pool consists of two models, gemma-4B and Qwen-27B, and all eight verification policies described in Section A, matching the configuration used in profiling (Section 3.3).

Configurations compared. Four single-model baselines, three single-axis ILP variants, and Dyserve; Table 1 summarizes how each fixes or optimizes the two design axes.

4.2. Main Results

Figure 3 reports the (accuracy, latency) trade-off across the four single-model baselines and Dyserve (Table 3 lists exact numbers). The baselines establish the corners: `4B base` is fastest (4.2 s) but lowest accuracy (0.42), while `27B verify-all` reaches the highest baseline accuracy (0.76) at the slowest latency (383.4 s). Dyserve sits inside this envelope and lies on a more favorable accuracy-latency trade-off than every fixed baseline at matched accuracy regimes: it nearly matches `27B verify-all`'s accuracy (0.74 vs. 0.76) at $\sim 10\times$ lower latency (37.2 s vs. 383.4 s) by replacing strong-model calls with small-model+verifier substitutes wherever the substitution is profitable, and Pareto-dominates `27B base` (0.72, 44.0 s) by gaining 2 pp of accuracy at slightly lower latency through verifier placement only on high-vulnerability nodes.

4.3. Lambda Sweep

The Lagrangian weights $(\lambda_q, \lambda_b, \lambda_v)$ and the topology weights $w(n)$ trade Dyserve's accuracy against its latency in the objective (5). Table 2 reports sensitivity around the default configuration (top row). Perturbing λ_b or λ_v traces a local Pareto curve: small perturbations cost ~ 6 pp of accuracy for lower latency, while large excursions (λ_b=0.1 or λ_v=5.0) collapse accuracy to 0.38 and 0.50. The most informative row is the uniform-topology variant: holding the Lagrangians at the default and replacing the topology weights with uniform $(1, 1, 1)$ drops accuracy by 10 pp (0.74→0.64)

λ_q	λ_b	λ_v	Topology	Accuracy	Latency (s)
20	**0.02**	**0.5**	**Sherlock**	**0.740**	**37.2**
20	0.005	0.5	Sherlock	0.625	26.1
20	0.1	0.5	Sherlock	0.375	13.2
20	0.02	0.1	Sherlock	0.625	18.7
20	0.02	5.0	Sherlock	0.500	35.0
20	0.02	0.5	$(1, 1, 1)$	0.640	37.1
20	0.05	0.5	Sherlock	0.650	27.0
30	0.02	0.3	Sherlock	0.600	32.5

Table 2. Lambda and topology sensitivity around the default Dyserve configuration (top row, bold). Rows 2–5 also relax the verification budget to ρ=1.0 (the default uses ρ=0.25), so the perturbations are not strict ceteris paribus. Rows 7–8 are alternative operating points from earlier sweeps.

at near-identical latency, isolating topology-aware vulnerability weighting (Section 3.3) as the lever behind the joint formulation's gains.

4.4. Co-Optimization Ablation

To isolate where Dyserve's gain comes from, we ablate three single-axis ILP variants alongside the full joint optimization (Table 3; configurations defined in Table 1). Both `Verify-ILP` variants reduce latency relative to their fixed-model `verify-all` counterparts (12.2 s vs. 16.9 s at 4B; 195.5 s vs. 383.4 s at 27B) but plateau in accuracy at 0.56 and 0.72 respectively, since neither can promote the hardest nodes to a stronger model. `Model-ILP` reaches only 0.62 at latency comparable to Dyserve (35.4 s vs. 37.2 s): with the verifier locked to the per-type accuracy-maximizing policy, the ILP can buy latency only by demoting nodes to the smaller model, sacrificing the accuracy that the joint formulation recovers by selectively dropping verifiers on low-vulnerability nodes. Only the joint formulation captures the substitution between model strength and verifier strength that Section 2.2 identified.

4.5. Discussion

Three observations follow from Figure 3. First, Dyserve closes most of the headroom that Section 2.1 identified between any single-model baseline and the per-node oracle, while the ILP itself solves in < 100 ms per request. Second, the sensitivity sweep shows the default sits at the highest-accuracy operating point in its neighborhood, and that topology-aware weighting (not the Lagrangians alone) is the dominant lever. Third, the co-optimization ablation confirms the structural claim of Section 2.2: neither model-only nor verifier-only optimization reaches the joint frontier, because the better verifier depends on which model produced the candidate and vice versa.

5. Related Work

Model routing (RouteLLM (Ong et al., 2025), Tabi (Wang et al., 2023b), FrugalGPT (Chen et al., 2024), SLM-MUX (Wang et al., 2025a)) optimizes the model axis but fixes verification. **Verifier selection** (Self-Refine (Madaan et al., 2023), Self-Consistency (Wang et al., 2023a), debate (Du et al., 2024), LLM-as-judge (Zheng et al., 2023)) does the converse. **Workflow generators** (MaAS (Zhang et al., 2025a), EvoAgentX (Wang et al., 2025b), Flow (Niu et al., 2025)) synthesize a workflow per request or task family but do not select operators inside it. **Agent-aware serving runtimes** (vLLM (Kwon et al., 2023), Sherlock (Ro et al., 2025), Murakkab (Chaudhry et al., 2025), MoA serving (Wang et al., 2025c)) address engine-level batching and verification-aware scheduling. Dyserve closes the gap by combining model selection, verification, and runtime in a single ILP keyed by node-type-aware profile coefficients.

Configuration	Accuracy	Latency (s)
Fixed-model baselines		
`4B base`	0.42	4.2
`4B verify-all`	0.58	16.9
`27B base`	0.72	44.0
`27B verify-all`	0.76	383.4
Single-axis ILP		
`Verify-ILP@4B`	0.56	12.2
`Verify-ILP@27B`	0.72	195.5
`Model-ILP`	0.62	35.4
`Dyserve` (joint)	**0.74**	**37.2**

Table 3. Co-optimization ablation on LiveCodeBench. Configuration definitions are in Table 1.

6. Conclusion

Dyserve treats the agent serving strategy as a structured optimization variable: per-node (model, verifier) selection over a workflow abstraction is formulated as a profile-coefficient-driven ILP, and an event-driven suffix-repair mechanism re-solves the residual workflow when runtime conditions change. On LiveCodeBench, the resulting strategies nearly match the strongest single-model verify-all baseline at substantially lower latency.

Two design choices deserve emphasis. First, *joint* (model, verifier) selection captures a substitution that single-axis methods cannot: the right verifier depends on which model produced the candidate, and the right model depends on which verifier will check it downstream. Second, modeling each verification policy as a sequence of call stages, rather than as a black box, is necessary for the ILP coefficients to honestly account for per-stage GPU-time.

Future work wires runtime triggers into the suffix-repair ILP (a live system-state monitor, verifier disagreement scoring, and tool-failure detection) and lifts retry/fallback/escalation from planning-time to runtime. We plan to fold speculation into the ILP as an explicit axis and extend the profiler to capture operator-interaction effects.

Impact Statement

This paper presents work toward more efficient agentic LLM serving by jointly optimizing per-node model selection and verifier placement. By reducing the GPU-time required at matched accuracy, the work contributes to lower compute and energy costs for LLM-based agentic systems. We do not foresee specific ethical risks beyond those generic to LLM deployment.

Acknowledgements

This work was supported in part by CoCoSys, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

Chaudhry, G. I., Choukse, E., Qiu, H., Goiri, Í., Fonseca, R., Belay, A., and Bianchini, R. Murakkab: Resource-efficient agentic workflow orchestration in cloud platforms, 2025. URL https://arxiv.org/abs/2508.18298.

Chen, L., Zaharia, M., and Zou, J. Frugalgpt: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=cSimKw5p6R.

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., and Mordatch, I. Improving factuality and reasoning in language models through multiagent debate. In Salakhutdinov, R., Kolter, Z., Heller, K., Weller, A., Oliver, N., Scarlett, J., and Berkenkamp, F. (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11733–11763. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/du24e.html.

Dunning, I., Mitchell, S., and O'Sullivan, M. PuLP: A linear programming toolkit for python. Technical report, Department of Engineering Science, The University of Auckland, September 2011. URL https://optimization-online.org/2011/09/3178/.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Schmidhuber, J. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626. Association for Computing Machinery, 2023. doi: 10.1145/3600006.3613165.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023.

Niu, B., Song, Y., Lian, K., Shen, Y., Yao, Y., Zhang, K., and Liu, T. Flow: Modularized agentic workflow automation. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=sLKDbuyq99.

Ong, I., Almahairi, A., Wu, V., Chiang, W.-L., Wu, T., Gonzalez, J. E., Kadous, M. W., and Stoica, I. RouteLLM: Learning to route LLMs from preference data. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=8sSqNntaMr.

Ro, Y., Qiu, H., Goiri, Í., Fonseca, R., Bianchini, R., Akella, A., Wang, Z., Erez, M., and Choukse, E. Sherlock: Reliable and efficient agentic workflow execution, 2025. URL https://arxiv.org/abs/2511.00330.

Significant Gravitas. AutoGPT: An autonomous GPT-4 experiment. https://github.com/Significant-Gravitas/AutoGPT, 2023. Software.

Wang, C., Wan, Z., Kang, H., Chen, E., Xie, Z., Krishna, T., Reddi, V. J., and Du, Y. SLM-MUX: Orchestrating small language models for reasoning, 2025a. URL https://arxiv.org/abs/2510.05077.

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=1PL1NIMMrw.

Wang, Y., Chen, K., Tan, H., and Guo, K. Tabi: An efficient multi-level inference system for large language models. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pp. 233–248. Association

for Computing Machinery, 2023b. doi: 10.1145/3552326.3587438.

Wang, Y., Liu, S., Fang, J., and Meng, Z. EvoAgentX: An automated framework for evolving agentic workflows. In Habernal, I., Schulam, P., and Tiedemann, J. (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 643–655, Suzhou, China, November 2025b. Association for Computational Linguistics. ISBN 979-8-89176-334-0. doi: 10.18653/v1/2025.emnlp-demos.47. URL https://aclanthology.org/2025.emnlp-demos.47/.

Wang, Z., Qi, Y., Chen, H., Wan, Z., Sun, G., Li, D., Pei, S., and Hao, C. Efficient mixture-of-agents serving via tree-structured routing, adaptive pruning, and dependency-aware prefill-decode overlap, 2025c. URL https://arxiv.org/abs/2512.18126.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Awadallah, A., White, R. W., Burger, D., and Wang, C. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *Conference on Language Modeling*, 2024. URL https://www.microsoft.com/en-us/research/publication/autogen-enabling-next-gen-llm-applications-via-multi-agent-conversation-framework/.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Ye, N., Ahuja, A., Liargkovas, G., Lu, Y., Kaffes, K., and Peng, T. Speculative actions: A lossless framework for faster agentic systems, 2025. URL https://arxiv.org/abs/2510.04371.

Zhang, G., Niu, L., Fang, J., Wang, K., Bai, L., and Wang, X. Multi-agent architecture search via agentic supernet. In *International Conference on Machine Learning*, 2025a. URL https://openreview.net/forum?id=imcyVlzpXh.

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X.-H., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., Zheng, B., Liu, B., Luo, Y., and Wu, C. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=z5uVAKwmjf.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36: 46595–46623, 2023.

Policy	Behavior	Calls
`none`	pass-through	1
`self_refine`	feedback → rewrite	2
`self_refine_iter`	multi-round, halt on `STOP:`	up to K
`advanced_refine`	parallel critique + rewrite	k par. + 1
`self_consistency`	k samples + vote	k par. + 1
`judge_gate`	pointwise verdict only	1
`gated_refine`	gate; escalate on REJECT	1 or 3
`debate`	opponent/proponent + select	$2K+1$

Table 4. Verification policies and their call structures. The profiler decomposes each policy by stage so latency and token cost enter the ILP as honest coefficients under concurrency.

A. Verification Policies

This appendix lists the eight verification policies exposed at every applicable node, together with their call structures (Table 4). Each policy is described by a sequence of stages, each containing one or more parallel sub-calls; the call structure determines how token cost and wall latency aggregate. For example, a 3-sample `self_consistency` policy issues three calls in parallel and one selection call serially, so its expected wall latency is approximately $\max(t_1, t_2, t_3) + t_{\text{sel}}$, while its token cost is the sum of all four. Treating any of these policies as a black box overstates wall latency under concurrency by up to $\sim 3\times$; our profiler tags each sub-call so per-stage statistics are recoverable.