ORACLE: AGENTIC AI ORCHESTRATOR ROUTING VIA ADAPTIVE VERIFIER CALIBRATION FEEDBACK

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern enterprise agent deployments consist of a heterogeneous pool of large language models (LLMs) having diverse capabilities and cost. Existing model routing strategies optimize the quality-cost trade-off, while providing request-level *static* decisions. More recent solutions address agentic routing as a task-level selection with a serial verifier based router feedback loop. However, their fixed verifier suitable for homogeneous workloads may not generalize to heterogeneous batches of agentic tasks (example: coding, general conversational). Additionally, due to the verifier placement in the critical path of the loop, serving quality may be affected during multiple concurrent requests routing. To mitigate these issues, we present ORACLE. It is a concurrency-aware online routing mechanism that aligns adaptive routing with adaptive verification for feedback. ORACLE acts as a *training-free* drop-in 'feedback loop' on top of any model-selection policy to first classify the task type and then dynamically assigns a task-appropriate verifier. Further, we develop a *delayed feedback* strategy for concurrent requests that largely removes verifier latency from the dispatch critical path. We then present a post-routing dispatch scheduler, namely DISC. DISC reserves each task's peak KV footprint at admission and dispatches to an alternate backend when the reward gain from reduced wait exceeds the reward loss from lower accuracy. Extensive evaluation on SWE-bench, τ^2-bench, and Terminal-Bench 2.0 shows that ORACLE improves the accuracy–cost frontier by up to 7 percentage points over state-of-the-art routing baselines, while ORACLE with DISC improves program throughput by up to $1.8\times$.

1 INTRODUCTION

Modern enterprise agent deployment involves a pool of large language model (LLM) choices. For example, Datadog (2026) reports that over 70% of the organizations use three or more models. In such deployments, larger models generally offer higher accuracy; however, they are costly and slow. Smaller models trade accuracy for lower cost and faster inference. Model serving APIs from different vendors may include models of similar sizes that provide different accuracy–cost trade-offs, with relative performance varying across benchmarks. Efficient and effective routing has therefore become an integral part of the agent orchestration layer, for optimized serving of agentic multi-turn ReAct (Yao et al., 2022) style tasks. For example, LLMRouterBench (Li et al., 2026) reports that routing across heterogeneous models can reduce cost by up to 31.7% without sacrificing accuracy.

The majority of routing works such as different variants of the semantic router (Liu et al., 2026b; Xue et al., 2026; Tian et al., 2026) make model-selection decisions from request features without adapting to execution outcomes. This problem escalates in multi-turn agentic use cases, as the router's ability to select can have a domino effect on both cost and efficacy of the agent-harness. We thus argue that *routing principle needs to learn from the execution feedback*. A recent work, ACRouter (Zhou et al., 2026), is consistent with this principle, incorporating execution-grounded feedback into subsequent decisions through a *sequential* context–action–feedback (CAF) loop. However, this work faces two key challenges. First, the lack of task classification can corrupt online learning because heterogeneous tasks, such as enterprise tool use and software engineering, require different feedback mechanisms. Second, its sequential CAF loop can place verification (Sec. 2) on the critical path under concurrent request handling.

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

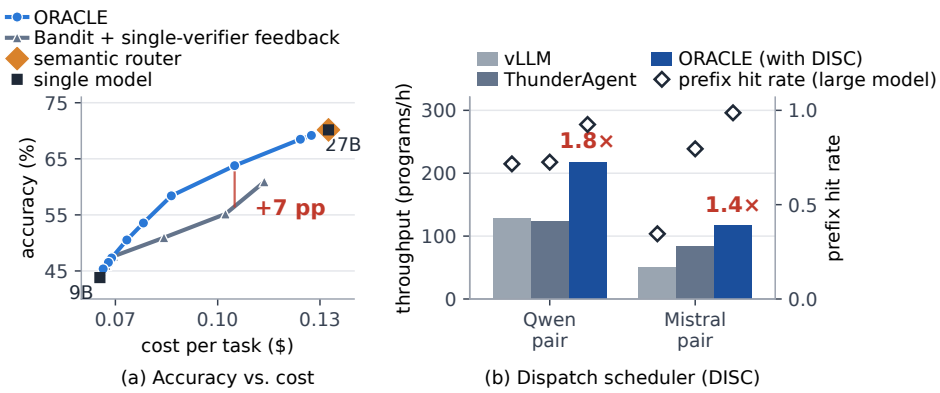


Figure 1: ORACLE comparison with routing and scheduling baselines using Qwen3.6-27B (FP8) and Qwen3.5-9B; panel (b) adds Devstral-Small-2-24B and Ministral-3-14B. (a) Accuracy–cost trade-off on a heterogeneous stream of SWE-bench Verified, τ^2-bench, and Terminal-Bench 2.0. ORACLE uses the same bandit selector as the baseline but replaces single-verifier feedback with adaptive verifier feedback, achieving up to 7 pp higher accuracy at matched cost. (b) Program-throughput comparison on concurrent SWE-bench Verified tasks, where ORACLE with DISC achieves 1.4–1.8× higher throughput than ThunderAgent.

Beyond feedback adaptation, heterogeneous routing also introduces a systems challenge: *load imbalance*. Routing decisions can concentrate requests on one backend (a serving instance hosting a particular model) while leaving another underutilized. The resulting increase in backend concurrency can degrade KV-cache efficiency and throughput: ThunderAgent (Kang et al., 2026) reports that, even for homogeneous models, scaling from 24 to 96 concurrent agentic tasks reduces prefix-cache hit rate from roughly 98% to 32% and throughput from about 220 to 140 steps/min.

Our Contributions. We present ORACLE, a concurrency-aware online routing mechanism that provides a *training-free*, drop-in feedback loop over existing model-selection policies. ORACLE first identifies each task's type and assigns a corresponding verifier, whose outcome feedback guides subsequent routing decisions. At each task's first request, ORACLE stores the selected model and verifier in a program-ID-indexed lookup table; subsequent requests reuse this binding. This abstraction decouples ORACLE's feedback loop from the underlying routing policy (e.g., Agent router, complexity router) and verifiers (e.g., LLM-as-a-judge, unit tests), enabling seamless integration.

Building on this lookup-table abstraction, ORACLE supports concurrent heterogeneous tasks through two mechanisms: *adaptive verification* and *delayed feedback*. Adaptive verification is driven by task classification: ORACLE infers each task's type from its initial request and selects a corresponding verifier for outcome feedback. With delayed feedback, ORACLE dispatches new tasks using the current routing policy without waiting for prior verification. Feedback from completed tasks asynchronously updates the policy for subsequent decisions, removing verification from the dispatch critical path.

We further present DISC, a post-route dispatch scheduler that mitigates load imbalance across heterogeneous backends. It reserves each task's expected peak KV-cache footprint in a per-backend ledger, admitting tasks only when capacity permits. At task boundaries, DISC selects an alternate backend only when the predicted reward gain from reduced admission wait exceeds the estimated reward loss from lower accuracy. ORACLE includes DISC by default.

We evaluate ORACLE on a heterogeneous stream of SWE-bench Verified (Jimenez et al., 2024), τ^2-bench (Barres et al., 2026), and Terminal-Bench 2.0 (Merrill et al., 2026), and on concurrent SWE-bench Verified tasks in Fig. 1. With the same bandit selector, ORACLE's adaptive verifier feedback achieves up to **7 percentage points (pp)** higher accuracy than single-verifier feedback at matched cost (Fig. 1a); with DISC, ORACLE achieves **1.4–1.8**× higher program throughput than ThunderAgent across the two evaluated model pairs (Fig. 1b).

2 Related Works

Feedback-aware routing. SWE-Router (Son et al., 2026) uses a partial execution trajectory to decide whether to continue with a cheaper model or restart with a stronger model. ACRouter accumulates execution feedback across coding tasks through a sequential context–action–feedback loop, using task-dependent verification signals. ORACLE targets concurrent, heterogeneous task streams through a *training-free* feedback loop over existing model-selection policies. Its delayed feedback strategy incorporates verified outcomes into subsequent routing decisions without blocking new task dispatch.

Verification of agentic tasks. Software-engineering tasks are evaluated through test execution (Jimenez et al., 2024; Xia et al., 2024), while LLM-based judges assess open-ended responses (Zheng et al., 2023) and agent-based judges inspect execution artifacts against task requirements (Zhuge et al., 2025). JudgeBench (Tan et al., 2025) examines judge reliability on difficult responses, and work on noisy verification rewards (Cai et al., 2026) studies how verifier errors affect reinforcement learning. RewardAgent (Peng et al., 2025) selects verification agents and combines factuality and instruction-following checks with preference scores for training and inference-time response selection. ORACLE applies adaptive verification to online model selection: it infers each task's type from the initial request, selects an appropriate verifier, and uses the resulting outcome feedback to update the routing policy.

Load-aware routing and scheduling. ThunderAgent (Kang et al., 2026) schedules agentic programs using their KV-cache footprints and tool-resource lifecycles. Aragog (Dai et al., 2025) identifies workflow configurations predicted to preserve accuracy and adapts stage-level model assignments to serving capacity. DISC, the post-route scheduler in ORACLE, uses per-backend reservations of each task's expected peak KV-cache footprint to control admission and predict waiting time. Before a task starts, DISC selects an alternate backend only when the predicted reward gain from reduced wait exceeds the estimated reward loss from lower accuracy.

3 Challenges in Routing Concurrent Agentic Workloads

We examine three challenges in feedback-aware concurrent agentic routing: verifier mismatch, feedback handling under concurrency, and load imbalance across model backends. Our routing experiments use a LinUCB learner in the feedback loop of a released feedback-aware router (Zhou et al., 2026), with replayed outcomes for Qwen3.5-9B and Qwen3.6-27B-FP8. We also profile a program KV-aware serving system (Kang et al., 2026) on two GH200 GPUs with 64 concurrent SWE-bench Verified tasks.

3.1 Verifier Mismatch Limits Online Learning

Feedback-aware routing uses the outcomes of completed tasks to improve model selection. Applying the same verifier across different task types, however, can give the learner incorrect feedback. In our replay (Fig. 2a), a verifier designed for software patches reports every τ^2-bench task as a failure because it produces no patch, regardless of whether the task succeeds. We compare this patch-only setup with adaptive verification, which selects a verifier based on the inferred task type. Both use the same learner, task orders, and mixed stream of τ^2-bench tasks and SWE-bench background tasks. We measure τ^2-bench success using its benchmark evaluator in both settings. After eight passes over the stream, adaptive verification improves τ^2-bench success by approximately 9.1 percentage points (pp) over patch-only verification. Since only the verification feedback changes, this comparison shows that task-appropriate verification improves routing with the same learner.

3.2 Feedback Handling Under Concurrency

We examine two implementations of concurrent feedback-aware routing. *Batch-synchronous dispatch* uses a shared learner but waits for every task in a batch to finish and be verified before dispatching the next batch. *Independent per-slot learners* allow each slot to proceed independently but divide the available feedback: with C slots and a fixed total task budget, each learner receives approximately $1/C$ of the stream.

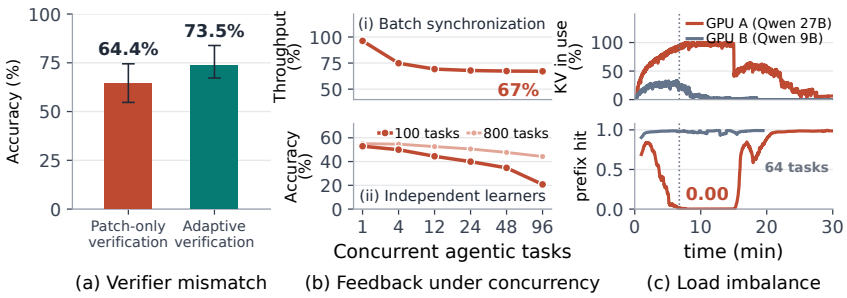


Figure 2: Challenges in feedback-aware routing for concurrent agentic workloads. (a) Verifier mismatch: τ^2-bench accuracy after eight passes over the same mixed task stream, changing only verification feedback. (b) Feedback under concurrency on SWE-bench Verified: batch-synchronous dispatch loses throughput and independent per-slot learners lose accuracy. (c) Load imbalance under routing of 64 concurrent SWE-bench Verified tasks across Qwen3.6-27B (FP8) and Qwen3.5-9B backends on two GH200 GPUs.

Fig. 2b evaluates both implementations on SWE-bench Verified. We measure that batch-synchronous dispatch achieves 67% of ideal throughput at 96 concurrent tasks. The ideal reference starts each slot's next task as soon as execution finishes, without waiting for verification or other slots. With independent learners, increasing concurrency from 1 to 96 concurrent tasks reduces task success by approximately 32 pp for a 100-task stream and 11 pp for an 800-task stream. These results motivate *delayed feedback* with a shared routing policy: new tasks proceed without waiting for prior verification, while completed-task feedback updates subsequent routing decisions.

3.3 Load Imbalance Across Heterogeneous Backends

Model-selection decisions can concentrate concurrent tasks on one backend without accounting for its available KV-cache capacity. The preferred backend may then become memory-constrained while another retains spare capacity. Under high concurrency, this imbalance can disrupt prefix reuse and reduce serving efficiency.

Fig. 2c shows this imbalance when 64 concurrent SWE-bench Verified tasks are routed across Qwen3.6-27B (FP8) and Qwen3.5-9B backends without accounting for backend load. The 27B backend approaches full KV-cache capacity after approximately 6.8 minutes, and its prefix-cache hit rate falls to near zero. Meanwhile, the 9B backend reaches at most 33% KV-cache use and maintains a prefix-cache hit rate of roughly 98% while active. These observations motivate DISC's use of per-backend KV-cache reservations to guide admission and initial task dispatch.

4 ORACLE: A Concurrency-aware Agentic Routing System

To address the challenges described in Sec. 3, we present ORACLE (Fig. 3), a concurrency-aware online routing mechanism for heterogeneous agentic workloads. Sec. 4.1 formalizes the framework for routing and scheduling. Secs. 4.2.1 and 4.2.2 describe its two primary mechanisms: adaptive verification and delayed feedback. Finally, Sec. 4.3 presents DISC, a post-route dispatch scheduler that mitigates load imbalance across heterogeneous model backends.

4.1 Routing Abstraction

We abstract agentic routing using a lookup table that binds each program to a model and a verifier. The program serves as the common unit of concurrency, routing, and feedback. For an agentic program $\mathcal{P}_{\mathrm{ID}}$ with identifier ID, we define the routing abstraction as a tuple:

$$\mathcal{R} = \langle \mathrm{ID}, \mathcal{P}, \mathcal{M}, \mathcal{V} \rangle, \tag{1}$$

where $\mathcal{M} = \{1, \ldots, K\}$ and $\mathcal{V} = \{1, \ldots, L\}$ index the candidate models and verifiers, respectively.

At a program's first request, ORACLE assigns its unique identifier ID and records the binding $\mathcal{P}_{\mathrm{ID}} \mapsto (m, v)$ for the selected model $m \in \mathcal{M}$ and verifier $v \in \mathcal{V}$. Subsequent requests from

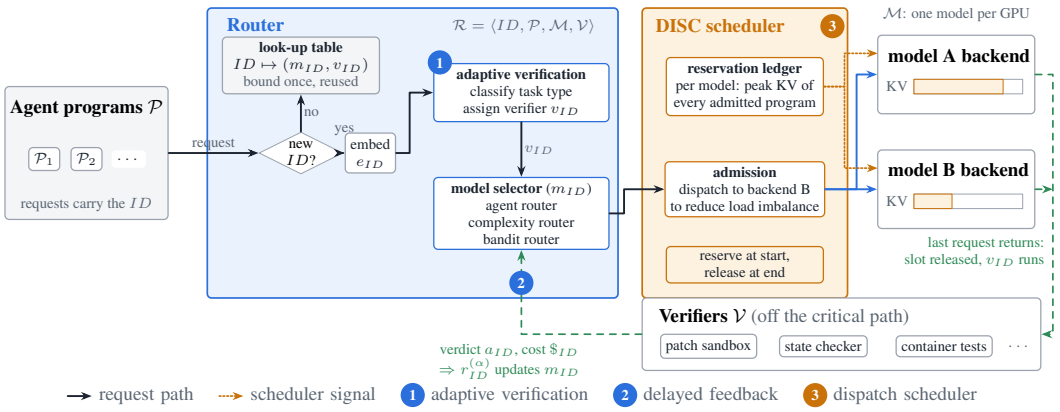


Figure 3: **Overview of ORACLE.** The routing abstraction $\mathcal{R} = \langle \mathrm{ID}, \mathcal{P}, \mathcal{M}, \mathcal{V} \rangle$ binds each program to a model and verifier. Task classification selects the verifier from the initial request (1), and subsequent requests reuse the binding stored in the lookup table. When execution completes, the slot is released and verification runs asynchronously; delayed feedback updates the model-selection policy (2). Before execution, DISC (3) controls admission through per-backend KV-cache reservations and selects an alternate backend only when the predicted reward gain from reduced admission wait exceeds the estimated reward loss from lower accuracy.

$\mathcal{P}_{\mathrm{ID}}$ use the same model, while the assigned verifier evaluates the completed program and supplies feedback for future routing decisions.

This abstraction decouples ORACLE's feedback loop from the underlying routing policy and verifier implementation. Tool calls, host-side computation, retrieval, and environment interactions remain part of program execution but do not receive separate routing-table entries.

4.2 Concurrency-Aware Routing Mechanism

Rapid growth in LLM usage motivates efficient concurrent program routing: Google reports that daily requests to Gemini (Gemini Team et al., 2023) grew more than sevenfold between May 2025 and May 2026 (Pichai, 2026). Building on this lookup-table abstraction, ORACLE supports online model selection for concurrent heterogeneous agentic programs through two mechanisms: *adaptive verification* and *delayed feedback*.

4.2.1 Adaptive Verification

Different task types require different verification of success. Software patches can be evaluated through test execution, whereas tool-use tasks may require verification of the resulting environment state. Applying the same verifier across these tasks can produce incorrect feedback, as shown in Fig. 2a. ORACLE therefore identifies each program's task type from its initial request and selects a corresponding verifier.

Let e_{ID} denote the embedding of the initial request of program $\mathcal{P}_{\mathrm{ID}}$. For each verifier $v \in \mathcal{V}$, we construct a task-type prototype μ_v by averaging the embeddings of k exemplar requests known to require that verifier. We infer the task type by matching the incoming request to these prototypes and select the corresponding verifier:

$$v_{\mathcal{P}_{\mathrm{ID}}} = \arg\max_{v \in \mathcal{V}} \mathrm{sim}\left(e_{\mathrm{ID}}, \mu_v\right), \tag{2}$$

where $\mathcal{V} = \{1, \ldots, L\}$ indexes the candidate verifiers and sim measures embedding similarity. This selection rule requires labeled exemplars but no additional model training. A new verifier can be added by constructing its prototype from the corresponding exemplar requests.

The selected verifier is recorded in the program's lookup-table entry and evaluates its outcome upon completion. Its verdict provides feedback for subsequent routing decisions. Incorrect task classification can select an unsuitable verifier and give the learner misleading feedback. We therefore evaluate verifier-selection accuracy on held-out tasks (Appendix B.3).

4.2.2 DELAYED FEEDBACK

ORACLE combines concurrent program dispatch with online adaptation through *delayed feedback*. When a program finishes execution, ORACLE releases its slot for the next program and runs the assigned verifier asynchronously. Incoming programs use the current routing policy, while verified outcomes update the policy as they become available.

For each completed program, ORACLE computes a reward combining its verified success score and execution cost:

$$r^{(\alpha)}_{\mathcal{P}_{\mathrm{ID}}} = (1-\alpha)a_{\mathcal{P}_{\mathrm{ID}}} - \alpha\widetilde{\$}_{\mathcal{P}_{\mathrm{ID}}}, \tag{3}$$

where $\alpha \in [0,1]$ controls the accuracy–cost trade-off, $a_{\mathcal{P}_{\mathrm{ID}}} \in [0,1]$ is the score returned by the assigned verifier, and $\widetilde{\$}_{\mathcal{P}_{\mathrm{ID}}} = \min\left(\$_{\mathcal{P}_{\mathrm{ID}}}/\$_0, 1\right)$ is the final program cost normalized by a reference cost $\$_0 > 0$ and clipped at one.

Each reward is associated with the program's original routing context and selected model through its identifier. The shared routing policy incorporates feedback in arrival order, allowing programs to complete and be verified in a different order from their dispatch. Once the update is complete, ORACLE removes the program's lookup-table entry.

4.3 DISPATCH SCHEDULER (DISC)

Model quality varies across tasks. In Qwen's reported evaluations (Qwen Team, 2026a), Qwen3.6-27B leads Qwen3.5-397B-A17B by 6.8 percentage points (pp) on Terminal-Bench 2.0 (Merrill et al., 2026) (59.3% vs. 52.5%) but trails it by 4.7 pp on Humanity's Last Exam (Phan et al., 2025) (24.0% vs. 28.7%). A router that learns such preferences can concentrate tasks on one model, filling its backend's KV cache while another remains underutilized (Fig. 2c). Addressing this imbalance requires coordinating model selection with backend capacity. We therefore present DISC, a post-route dispatch scheduler that adjusts the backend assignment to reduce load imbalance.

DISC maintains a *reservation ledger* for each backend. When a program $\mathcal{P}_{\mathrm{ID}}$ is admitted to model m, the ledger reserves its expected peak KV-cache footprint and releases the reservation when execution completes. A program is admitted only when sufficient unreserved capacity is available. The admission component uses these reservations to forecast the wait w_m before backend m has sufficient capacity and exposes this forecast to the router at the program boundary.

Given the routing policy's proposed model m_{ID} and an alternative m' with a shorter forecast wait, DISC dispatches the program to m' only when the reward gain from reduced admission wait exceeds the estimated reward loss from lower accuracy:

$$\beta\frac{w_{m_{\mathrm{ID}}} - w_{m'}}{w_0} > (1-\beta)\big(\hat{a}_{m_{\mathrm{ID}}} - \hat{a}_{m'}\big), \tag{4}$$

where $\beta \in [0,1]$ controls the accuracy–wait trade-off, $\hat{a}_m$ is the routing policy's current estimate of model m's accuracy, and $w_0 > 0$ is a reference wait used for normalization. Eq. 4 is the expected-reward-maximizing choice under a wait-priced reward (Appendix C.1).

The dispatch decision is made once per program, before its first request, and the final assignment is recorded in the lookup table. Subsequent requests use the same backend, preserving the binding in Sec. 4.1 without splitting the program's prefix cache across backends.

5 EVALUATION

In this section, we evaluate ORACLE on three agentic workloads (enterprise tool calling, SWE, and terminal tasks) on GH200 and H200 GPUs. Furthermore, we conduct ablation studies (Sec. 5.5) to explore hyperparameters and to test ORACLE on a second model pair and other GPU deployments.

5.1 EXPERIMENTAL SETUP

Benchmarks and workflows. We evaluate mini-swe-agent (Yang et al., 2024) on SWE-bench Verified (108 instances for live experiments; 100 for accuracy–cost evaluation), τ^2-bench (Barres et al., 2026) on 276 retail, airline, and telecom tasks, and Terminus-2 on 89 Terminal-Bench 2.0

tasks (Merrill et al., 2026). SWE-bench and terminal tasks have 900 s execution budgets. Their respective verifiers are the SWE-bench harness, benchmark state checker, and container tests. For τ^2-bench, the selected model serves both the agent and user simulator.

Models and deployments. We use Qwen3.6-27B (FP8) (Qwen Team, 2026c) and Qwen3.5-9B (Qwen Team, 2026b), one model per GPU across two GH200 nodes. Additional experiments use a two-H200 node and a second pair, Devstral-Small-2-24B (Rastogi et al., 2025) and Ministral-3-14B (Liu et al., 2026a). We price input/output tokens at \$0.32/\$3.20 per million for the 27B model and \$0.10/\$0.15 for the 9B model, using DeepInfra reference rates (DeepInfra). Costs include agent and user-simulator calls but exclude classification, verification, and orchestrator routing calls.

Model selectors. We evaluate LinUCB (Li et al., 2010) and ACRouter's orchestrator (Zhou et al., 2026). LinUCB uses exploration coefficient $\alpha_{\text{UCB}} = 0.6$ and a context combining benchmark/domain indicators from metadata with 16 principal components of the bge-m3 initial-request embedding. ACRouter uses Qwen3.5-4B with the released prompts and outcome memory, evaluated at two cost settings over eight passes and eight seeds. Memory starts empty per seed and persists across passes, allowing retrieval of earlier outcomes for repeated tasks (Appendix E.6).

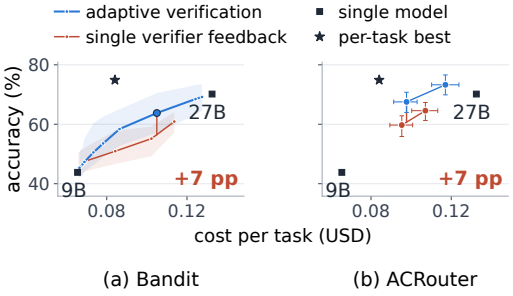


Figure 4: Adaptive verification versus single-verifier feedback on (a) bandit and (b) ACRouter.

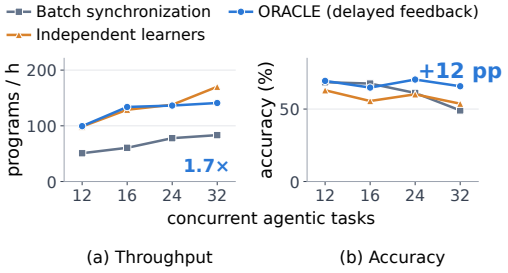


Figure 5: Comparison of feedback handling on 108 SWE-bench Verified tasks at 12–32 concurrent tasks using two GH200 GPUs.

Verification configurations. Adaptive verification uses $k = 5$ exemplars per verifier. Single-verifier feedback applies the released patch verifier to all tasks, assigning failure feedback to τ^2-bench and Terminal-Bench tasks. Within each selector, only verification feedback changes; evaluation always uses benchmark-specific success signals.

Concurrency configurations. We compare batch synchronization, independent per-slot learners, and delayed feedback using the same LinUCB algorithm and serving stack.

Dispatch scheduling. DISC reserves expected peak KV-cache footprints within 85% of the KV pool, with a 95% per-call ceiling, $\beta = 0.5$, and $w_0 = 600$ s. Baselines include vLLM with prefix caching, ThunderAgent's released proxy (Kang et al., 2026) ($\Delta t = 5$, $f(t) = 2^{-t}$), and a static cap. The static cap admits at most 27 programs on the 27B backend and 42 on the 9B backend, queuing additional programs. These limits are rounded from DISC's mean admitted counts during admission waiting at 64 concurrent tasks and remain fixed thereafter. The static cap uses no footprint model or backend reassignment.

Matched-cost comparisons. We interpolate single-verifier accuracy at the adaptive configuration's cost and report paired task-bootstrap confidence intervals. For the bandit, we select the adaptive point nearest the midpoint between single-model costs.

5.2 Adaptive Verification Evaluation

Accuracy–cost trade-off. Fig. 4 compares adaptive verification with single-verifier feedback using a bandit selector (a) and ACRouter's agent selector (b). Both route between Qwen3.5-9B and Qwen3.6-27B (FP8). The single-model baselines achieve 44% accuracy at \$0.066 per task and 70% at \$0.133, respectively. Adaptive verification improves accuracy by approximately 7 percentage points (pp) at the highlighted matched-cost comparisons in both panels.

Effect of verifier selection. Within each panel, both configurations use the same learner and task stream; only verification feedback changes. Single-verifier feedback applies the same verifier to all tasks, whereas adaptive verification selects a verifier based on task type. The task classifier selects the appropriate verifier on 97–99% of held-out tasks (Appendix B.3). The gains across both routing implementations show that task-appropriate verification improves model selection without replacing the underlying routing algorithm.

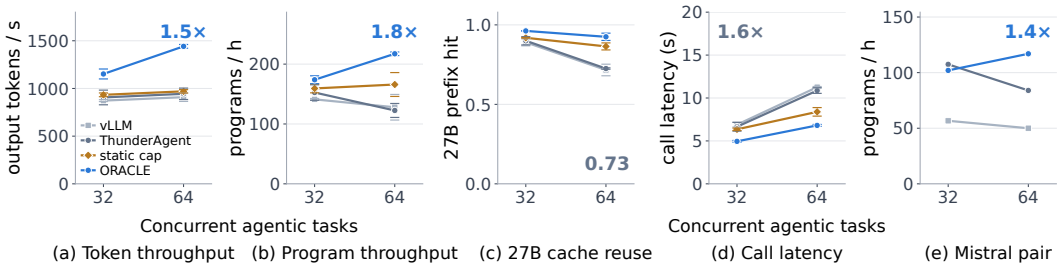


Figure 6: DISC evaluation on 108 SWE-bench Verified programs using two GH200 GPUs. Panels (a)–(d) average two plan seeds for the Qwen pair; panel (e) uses one seed for the Mistral pair.

5.3 Delayed Feedback Evaluation

Higher throughput without batch synchronization. Fig. 5 evaluates one pass over 108 SWE-bench Verified tasks on the GH200 pair at 12–32 concurrent tasks. All configurations use the same routing algorithm and serving stack, differing in how they handle feedback. By releasing each slot when program execution finishes, ORACLE allows subsequent tasks to proceed without waiting for other tasks or verification. At 32 concurrent tasks, it achieves $1.7\times$ the throughput of batch-synchronous dispatch. Its throughput is comparable to independent learners at 12–24 concurrent tasks, but lower at 32.

Higher accuracy through shared feedback. Independent per-slot learners each receive only a fraction of the completed-task feedback, whereas ORACLE uses every verified outcome to update a shared routing policy. Across concurrency levels, ORACLE resolves 7–13 additional tasks per 108-task run compared with independent learners. Relative to batch synchronization, it resolves 10 and 18 additional tasks at 24 and 32 concurrent tasks, respectively. Batch synchronization delays policy updates until the entire batch completes, reducing opportunities to adapt during a fixed task stream. Delayed feedback instead incorporates outcomes as they become available, improving accuracy while sustaining concurrent dispatch.

5.4 Dispatch Scheduler Evaluation

Higher throughput under concurrency. Fig. 6 evaluates 108 SWE-bench Verified programs on two GH200 GPUs at 32 and 64 concurrent tasks. Panels (a)–(d) show the Qwen pair, averaged over two plan seeds. At 64 concurrent tasks, ORACLE achieves $1.8\times$ the program throughput and $1.5\times$ the output-token throughput of ThunderAgent. Its 27B prefix-cache hit rate remains approximately 0.93, compared with 0.73 for ThunderAgent, 0.72 for vLLM, and 0.87 for the static cap. Mean call latency falls to 6.8 s, compared with 10.9 s, 11.3 s, and 8.4 s, respectively.

KV-aware admission across model pairs. DISC reserves each program's expected peak KV-cache footprint and can assign it to an alternate backend before execution. In the tested Qwen configuration, ThunderAgent's smaller per-call KV estimate admits nearly as many calls as vLLM, providing little protection against cache contention. Its admission control is more effective on the Mistral pair: the approximately 44K-token per-call estimate limits the Devstral-24B backend to 24 simultaneous calls at 32 concurrent tasks, compared with 31 under vLLM. Preemptions decrease from 21 to 7, and prefix-cache hit rate rises from 0.54 to 0.87. DISC's program-level reservations maintain a hit rate of 0.99 at both concurrency levels. At 64 concurrent tasks, ORACLE achieves 117 programs/h, compared with 84 for ThunderAgent and 50 for vLLM, yielding $1.4\times$ and $2.3\times$ higher throughput, respectively (panel (e), one plan seed).

Admission overhead. Reservations introduce waiting at lower concurrency. At 32 concurrent tasks on the Qwen pair, ORACLE incurs approximately two minutes of mean admission wait. On the

Mistral pair at 32 concurrent tasks, its throughput is slightly below ThunderAgent's (102 vs. 107 programs/h), although it leads at 64 concurrent tasks. The benefits of reservation-based admission are therefore most pronounced under higher cache contention.

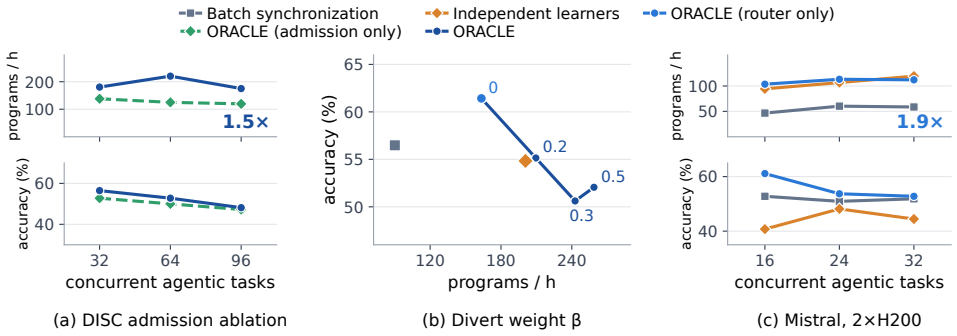


Figure 7: Ablations on SWE-bench Verified. Panels (a)–(b) use the Qwen pair on two GH200 GPUs; (b) aggregates 972 task executions. Panel (c) uses the Mistral pair on two H200 GPUs.

5.5 Ablation Study

Verification across task types. Table 1 reports routing gains by task type. Patch-only feedback yields almost no improvement on τ^2-bench or Terminal-Bench 2.0, whereas adaptive verification yields gains of 8.9 and 9.9 pp, respectively. It also matches the 17.9 pp gain on SWE-bench. Across all three task types, adaptive verification is within 0.5 pp of the best fixed verifier chosen separately for each type. Additional Terminal-Bench 1 results appear in Appendix E.3. **Contribution of backend reassignment.** Fig. 7(a) disables alternate-backend dispatch while retaining DISC's reservation-based admission. Admission alone resolves within four tasks of full ORACLE at every concurrency level, including 51 versus 52 tasks at 96 concurrent tasks. However, full ORACLE achieves $1.5\times$ its throughput at that concurrency. Admission control therefore preserves comparable accuracy, while backend reassignment increases throughput by using spare capacity rather than waiting at the preferred backend. **Dispatch-weight sensitivity.** Fig. 7(b) varies β at 64 concurrent tasks over three stream seeds. At $\beta = 0$, ORACLE achieves the highest accuracy but only $0.83\times$ the throughput of independent learners. Setting $\beta = 0.2$ gives comparable accuracy and throughput to independent learners. Values of 0.3 and 0.5 increase throughput to 1.2–$1.3\times$ their rate, at the cost of 27–41 fewer resolved tasks across 972 executions. We use $\beta = 0.5$ by default. Sensitivity to the LinUCB exploration coefficient α_{UCB} is reported in Appendix E.7. **Additional models and hardware.** Fig. 7(c) evaluates delayed feedback with Devstral-Small-2-24B and Ministral-3-14B on two H200 GPUs. At 24 and 32 concurrent tasks, ORACLE (router only) resolves 6 and 9 additional tasks compared with independent learners. At 32 concurrent tasks, it achieves $1.9\times$ the throughput of batch synchronization, although its throughput is slightly below that of independent learners. Additional DISC evaluations on the Mistral pair, the Qwen pair on H200 GPUs, and both models sharing one GH200 appear in Appendix E.2.

Table 1: Task-success gain over equal-probability model selection at $\alpha = 0$ (pp; 95% task-bootstrap intervals).

Task type	Patch-only	Adaptive	Best fixed
SWE-bench	+17.9	+17.9 [+13.0, +22.2]	+18.0
τ^2-bench	−0.2	+8.9 [+4.1, +16.4]	+9.4
TB 2.0	+0.1	+9.9 [+4.2, +13.8]	+10.0

6 Conclusion

We introduced ORACLE, a concurrency-aware routing framework for agentic programs. Its lookup-table abstraction binds each program to a model and verifier, providing a common unit for routing, feedback, and scheduling. Adaptive verification selects task-appropriate feedback, delayed feedback removes verification from the dispatch critical path, and DISC uses per-backend KV-cache reservations to balance accuracy against admission delay. Across SWE-bench, τ^2-bench, and Terminal-Bench 2.0, ORACLE improves the accuracy–cost frontier by up to 7 pp. With DISC, ORACLE improves program throughput by up to $1.8\times$ over ThunderAgent.

AI USE STATEMENT

In this work, we used generative AI tools for designing and giving feedback on experiments (pre-registering the experimental blocks, their readings, stop rules and sweeps), for implementing methods (the replay and figure scripts), for refining hypotheses about the mechanisms behind the measured effects, for interpreting results, and for assisting in the writing of the derivation in Appendix C.1, which formalizes a rule the authors specified. We have not used generative AI tools to generate synthetic datasets, to clean or reformat datasets, or for translation, and qualitative or thematic data analysis is not applicable to this work. Additionally, we used generative AI tools to draft the experimental and appendix sections from the measurement logs, to identify and summarize related work, to source the public API price lists behind the cost axis, to create the figures from the logged results, and to edit for readability the sections whose first drafts the authors wrote. We have reviewed all AI-assisted work: every number in the paper is read from result files by scripts the authors ran; the code used in the experiments was exercised against pre-registered readings before any reported cell ran; the derivation in Appendix C.1 was checked line by line by the authors; and AI-identified related work and prices were verified against the cited papers and providers' pages by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

All benchmarks (SWE-bench Verified, τ^2-bench, Terminal-Bench 2.0) and all model weights (Qwen3.6-27B-FP8, Qwen3.5-9B, Devstral-Small-2-24B, Ministral-3-14B) are publicly available; Section 5.1 names the harnesses, the price table, the pools and the baselines, and Appendix B gives the serving stack with version numbers, the single header an agent harness needs to adopt ORACLE, the ledger's calibration priors and learned allowances, the verifier selector and its exemplar protocol (Appendix B.3), the replay protocol with its context features, reward and bootstrap, and the grading pipeline for every benchmark. Every constant of ORACLE and of the baselines was fixed before the first cell ran and is listed in Table 2; each experimental block was pre-registered in a dated file with its cells, its reading and its stop rule, and blocks whose reading was not met are reported (Appendices E.8). The seeds, task counts and random orders behind every figure are stated in its caption, in Section 5.1 or in Appendix B, and the paired bootstrap that produces every interval is described in Appendix B. The hypotheses and the complete proof behind the divert rule are in Appendix C.1.

REFERENCES

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam. Automix: Automatically mixing language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2310.12963.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2506.07982.

Xin-Qiang Cai, Wei Wang, Feng Liu, Tongliang Liu, Gang Niu, and Masashi Sugiyama. Reinforcement learning with verifiable yet noisy rewards under imperfect verifiers. arXiv:2510.00915, 2026. URL https://arxiv.org/abs/2510.00915.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. arXiv:2305.05176.

Qiaoling Chen, Zhisheng Ye, Tian Tang, Peng Sun, Boyu Tian, Guoteng Wang, Shenggui Li, Yonggang Wen, Zhenhua Han, and Tianwei Zhang. Concur: High-throughput agentic batch inference of llm via congestion-based concurrency control. arXiv:2601.22705, 2026. URL https://arxiv.org/abs/2601.22705.

Jiaming Cheng, Duong The Do, and Duong Tung Nguyen. Robust kv cache management for llm serving under output token length uncertainty. arXiv:2607.16892, 2026. URL https://arxiv.org/abs/2607.16892.

Yinwei Dai, Zhuofu Chen, Anand Iyer, and Ravi Netravali. Aragog: Just-in-time model routing for scalable serving of agentic workflows. arXiv:2511.20975, 2025. URL https://arxiv.org/abs/2511.20975.

Datadog. State of AI engineering. https://www.datadoghq.com/state-of-ai-engineering/, 2026. Accessed: September 22, 2026.

DeepInfra. Simple pricing, deep infrastructure. https://deepinfra.com/pricing. Accessed September 23, 2026.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2404.14618.

Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: Flexible spatial-temporal multiplexing for multiple llm serving. In *International Conference on Machine Learning (ICML)*, 2024. arXiv:2404.02015.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.06770.

Pooria Joulani, András György, and Csaba Szepesvári. Online learning under delayed feedback. In *International Conference on Machine Learning (ICML)*, 2013.

Hao Kang, Ziyang Li, Weili Xu, Xinyu Yang, Yinfang Chen, Junxiong Wang, Beidi Chen, Tushar Krishna, Chenfeng Xu, and Simran Arora. Thunderagent: A simple, fast and program-aware agentic inference system. arXiv:2602.13692, 2026. URL https://arxiv.org/abs/2602.13692.

Tor Lattimore and Csaba Szepesvári. *Bandit Algorithms*. Cambridge University Press, 2020.

Hao Li, Yiqun Zhang, Zhaoyan Guo, Chenxu Wang, Shengji Tang, Qiaosheng Zhang, Yang Chen, Biqing Qi, Peng Ye, Lei Bai, et al. Llmrouterbench: A massive benchmark and unified framework for llm routing. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 37733–37754, 2026.

Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pp. 661–670, 2010.

Shaoang Li and Jian Li. Near-optimal online deployment and routing for streaming llms. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2506.17254.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023. arXiv:2302.11665.

Alexander H Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, Adrien Sadé, Alan Jeffares, Albert Jiang, Alexandre Cahill, Alexandre Gavaudan, et al. Ministral 3. *arXiv preprint arXiv:2601.08584*, 2026a.

Xunzhuo Liu, Huamin Chen, Samzong Lu, Yossi Ovadia, Guohong Wen, Hao Wu, Zhengda Tan, Jintao Zhang, Senan Zedan, Yehudit Kerido, et al. vllm semantic router: Signal driven decision routing for mixture-of-modality models. *arXiv preprint arXiv:2603.04444*, 2026b.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. arXiv:2601.11868, 2026. URL https://arxiv.org/abs/2601.11868.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2406.18665.

Hao Peng, Yunjia Qi, Xiaozhi Wang, Zijun Yao, Bin Xu, Lei Hou, and Juanzi Li. Agentic reward modeling: Integrating human preferences with verifiable correctness signals for reliable reward systems. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. arXiv:2502.19328.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025.

Sundar Pichai. I/O 2026: Welcome to the agentic Gemini era. Google Blog, May 2026. URL https://blog.google/intl/en-in/company-news/technology/sundar-pichai-io-2026/. Published May 19, 2026; accessed September 24, 2026.

Qwen Team. Qwen3.6-27B model card. Hugging Face, 2026a. URL https://huggingface.co/Qwen/Qwen3.6-27B. Accessed September 23, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026b. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026c. URL https://qwen.ai/blog?id=qwen3.6-27b.

Abhinav Rastogi, Adam Yang, Albert Q Jiang, Alexander H Liu, Alexandre Sablayrolles, Amélie Héliou, Amélie Martin, Anmol Agarwal, Andy Ehrenberg, Andy Lo, et al. Devstral: Fine-tuning language models for coding agent applications. *arXiv preprint arXiv:2509.25193*, 2025.

Marija Šakota, Maxime Peyrard, and Robert West. Fly-swat or cannon? cost-effective language model choice via meta-modeling. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM)*, 2024. doi: https://doi.org/10.1145/3616855.3635825. URL https://arxiv.org/abs/2308.06077.

Seongho Son, Sangwoong Yoon, Jiahua Tang, Shuhan Wang, Lorenz Wolf, and Ilija Bogunovic. Swe-router: Routing in multi-turn agentic software engineering tasks. arXiv:2607.00053, 2026. URL https://arxiv.org/abs/2607.00053.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024. arXiv:2406.03243.

Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Y. Tang, Alejandro Cuadron, Chenguang Wang, Raluca Ada Popa, and Ion Stoica. Judgebench: A benchmark for evaluating llm-based judges. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.12784.

Jiayi Tian, Yupeng Su, Ryan Solgi, Souvik Kundu, and Zheng Zhang. Rankguide: Tensor-rank-guided routing and steering for efficient reasoning. *Conference on Language Modeling*, 2026.

Xinyuan Wang, Yanchi Liu, Wei Cheng, Xujiang Zhao, Zhengzhang Chen, Wenchao Yu, Yanjie Fu, and Haifeng Chen. Mixllm: Dynamic routing in mixed large language models. In *Proceedings of NAACL-HLT*, 2025. arXiv:2502.18482.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. arXiv:2407.01489, 2024. URL https://arxiv.org/abs/2407.01489.

Yuxing Xiang et al. Aegaeon: Effective GPU pooling for concurrent LLM serving on the market. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2025. doi:10.1145/3731569.3764815.

Jiaqi Xue, Qian Lou, Jiarong Xing, and Heng Huang. R2-router: A new paradigm for llm routing with reasoning. *arXiv preprint arXiv:2602.02823*, 2026.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shan Yu, Yifan Qiao, Mingyuan Ma, Yangmin Li, Shuo Yang, Xinyuan Tong, Yang Wang, Zhiqiang Xie, Yuwei An, Shiyi Cao, Ke Bao, Deepak Vij, Xiaoning Ding, Yichen Wang, Qingda Lu, Zhong Wang, Gao Gao, Harry Xu, Junyi Shu, Jiarong Xing, and Ying Sheng. Prism: Cost-efficient multi-llm serving via gpu memory ballooning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2026. arXiv:2505.04021.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2023. arXiv:2306.05685.

Pengfei Zhou, Zhiwei Tang, Yixing Ma, Jiasheng Tang, Yizeng Han, Zhenglin Wan, Fanqing Meng, Wei Wang, Bohan Zhuang, Wangbo Zhao, et al. Agent-as-a-router: Agentic model routing for coding tasks. *arXiv preprint arXiv:2606.22902*, 2026.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-judge: Evaluate agents with agents. In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2410.10934.

A Extended Study

A.1 Request-level routing

Existing LLM routers primarily make routing decisions at the request level. Cascade-based methods initially use cheaper backends and invoke a more capable one when confidence is low (Chen et al., 2024; Aggarwal et al., 2024), while learned routers predict the appropriate backend from the prompt, difficulty labels, preference data, or backend-performance models (Ding et al., 2024; Ong et al., 2025; Šakota et al., 2024), with LLMRouterBench providing a common evaluation framework (Li et al., 2026). Other approaches use contextual bandits that update routing decisions from per-request feedback (Wang et al., 2025) or jointly optimize backend selection and routing (Li & Li, 2026). Despite these differences, these methods treat each request as an independent routing unit and observe its reward when the request completes. This assumption does not hold for agentic tasks, where a single task may generate tens to hundreds of requests and its final outcome is known only after the entire task completes. Consequently, request-level routers do not receive direct task-level feedback to learn from earlier routing decisions.

A.2 Bandits with delayed feedback

ORACLE uses a contextual UCB policy (Lattimore & Szepesvári, 2020), where the reward for a routing decision becomes available only when the corresponding agentic task completes. Since the feedback delay is determined by task execution time, ORACLE operates in the delayed-feedback bandit setting, where prior work shows that stochastic bandits can tolerate delayed rewards with an additive cost (Joulani et al., 2013). We make no new regret claim; instead, this result motivates a single learner that can continue making routing decisions while previous tasks are still running, enabling learning across concurrent tasks. In contrast, ACRouter's alternatives, such as closed-loop batching or maintaining a separate learner for each concurrency slot, either reduce throughput or limit learning across tasks.

A.3 Multi-Model Serving

Multi-model serving on a shared GPU pool requires both model placement and memory management. Prior systems address these challenges through model multiplexing, KV-cache and weight sharing, request migration, token-level switching, and dynamic memory allocation (Li et al., 2023; Duan et al., 2024; Sun et al., 2024; Xiang et al., 2025; Yu et al., 2026), while recent work also considers KV-cache-based admission control for long-lived agentic sessions (Chen et al., 2026; Cheng et al., 2026). These approaches generally treat the routing decision as an input and focus on managing the workload after requests are assigned to a backend. In contrast, ORACLE incorporates backend memory pressure directly into routing: its reservation ledger tracks KV-cache occupancy, estimates the resulting waiting time at each backend, and uses this forecast together with model accuracy to select a backend.

B Implementation Details

B.1 Serving stack and interfaces

Each model runs in its own vLLM engine behind a lightweight HTTP forwarder that the agents call in place of the engine. The forwarder logs every call's prompt, cached and completion tokens, and passes the program's key, an `X-Session-ID` header that the agent harness sets once per program, to the scheduler, so ThunderAgent's proxy and DISC's ledger see one stable key per program. Adopting ORACLE therefore changes one header in the agent harness and nothing in the agent or the engine. The router runs in the experiment driver: it binds a program to a model at the program boundary, hands the program to that model's admission queue, and receives the verdict from the grader when it lands. ThunderAgent's released proxy runs in front of each engine with $\Delta t = 5\,\mathrm{s}$ and priority decay on, the configuration of its paper; DISC adds the reservation ledger to the same proxy. ThunderAgent's result does not depend on the decay setting: with decay off, at 32 concurrent SWE-bench tasks on two plan seeds on the calibration pool (27B at 8 GiB of KV), the 27B's hit rate is 0.010 and 0.023 against 0.006 and 0.021 with decay on, call p95 106 and 77 s against 80 and 85 s, patches 9 and 8 against 10 and 6, preemptions 54 and 47 against 67 and 47.

B.2 Reservation ledger and divert rule

The ledger of engine m keeps, per admitted program p, a reservation of $\max(\text{tokens}_p, \hat{c}_m) + \hat{\delta}_m$ tokens, where $\hat{c}_m$ is the running mean of finished programs' peak context, started from the calibration prior c_0 with five pseudo-observations ($\hat{c}_m = (5c_0 + \sum \text{peaks})/(5 + n)$; $c_0 = 22.6$k tokens on the 27B and 44.7k on the 9B), and $\hat{\delta}_m$ is a learned per-request allowance: on the hybrid-attention engines each in-flight request holds a fixed KV state regardless of its token count, measured offline at 3.6–4.7k token-equivalents per running request (usage = tokens/capacity + this constant, R^2 0.79–0.91), which a token-sum capacity model such as ThunderAgent's does not see. The allowance is identifiable only while the engine is not saturated: added to ThunderAgent's proxy without the ledger, the estimator clamps to zero under the saturated pool and the arm matches ThunderAgent (27B hit rate 0.02–0.10 at 32 concurrent tasks on two plan seeds), whereas under DISC's admission control it converges to 3.0–3.4k tokens; it is a component of DISC, not a standalone fix. A new program is admitted, strictly FIFO, when the reserved total plus its own reservation fits within 85% of the engine's KV capacity; a 95% ceiling per call remains as the hard limit. The forecast wait is $w_m = 0$ if the queue is empty and the program fits now, otherwise $w_m = (q_m + 1)\, D_m / \max(K_m, 1)$, with q_m programs queued ahead, K_m admitted and D_m the mean run time of finished programs (prior 642 s and 639 s from the calibration passes). The divert rule of Eq. 4, derived in Appendix C.1, is evaluated once, at the boundary, with $\beta = 0.5$ and $w_0 = 600\,\mathrm{s}$; a diverted program starts on the other model from scratch, no KV state moves, and no program is moved after it starts. The reservation is released when the program's last call returns.

B.3 Verifier selector

Every task's admission context is embedded once by bge-m3 (sentence-transformers CLS pooling, normalized, on CPU). A verifier class is represented by the mean embedding of $k = 5$ exemplars drawn from tasks of that class and removed from the evaluation set; the selector returns the class of the nearest prototype by cosine. Terminal-Bench and Terminal-Bench 2.0 share the container-tests class. Over 200 random exemplar draws the per-family accuracy is 98.0% [95, 100] on SWE-bench, 99.0% [87, 100] on τ^2-bench, 93.6% [85, 99] on Terminal-Bench and 97.4% [92, 100] on Terminal-Bench 2.0 (n = 100, 200, 80, 89). With $k = 1$ the two terminal families fall to 76%; with $k = 10$ every family is at or above 96%. In the replays a task meets the wrong verifier with probability $1 - q$ for its family's measured q and is then credited 0.

B.4 Replay protocol

The frontier and verifier figures replay the router over measured per-task outcomes and costs of both models: τ^2-bench from single-model runs of all 276 tasks, SWE-bench Verified from single-model runs over 100 instances, Terminal-Bench 2.0 from four runs per model over 89 tasks (success is the resolved rate over runs, cost the mean tokens per run at the price table). The learner is LinUCB with

width $\alpha_{\mathrm{UCB}} = 0.6$; its context is a bias, a family one-hot and a τ^2 domain one-hot, plus 16 principal components of the task's bge-m3 embedding where one exists (the 89 Terminal-Bench 2.0 tasks and 23 SWE instances). The reward is $r = \text{success} - a \cdot \text{cost}/\text{scale}$, with scale the stream's mean 27B cost per task and $a \in \{0, 0.1, 0.25, 0.5, 0.75, 1, 1.5, 2, 3\}$. This is equivalent to the unclipped form of the reward in Eq. 3 with $\alpha = a/(1+a)$ and $\$_0 = \text{scale}$, divided by the positive constant $1 - \alpha$, which leaves the ordering of the two models unchanged; the live loops learn at $\alpha = 0$, where the clip is inactive. Each point is the mean over 100 random stream orders; the steady state (Fig. 4) runs eight passes over the stream and scores the last, the cold start (Appendix E.8) one pass. Intervals are task bootstraps stratified by family, 200 draws. Random mixing is the line between the two single models; the per-task best picks the cheaper successful model for each task. Fig. 4(a) and Fig. 9(a) use this replay with LinUCB; panel (b) of each replaces LinUCB with ACRouter's orchestrator (Appendix E.6) and keeps everything else.

B.5 Harnesses and grading

SWE-bench programs run mini-SWE-agent 2.2.8, one container per program with a 900 s budget, on the node that hosts the 9B (arm64 instance images under Docker) or, on 2xH200, under rootless Podman with x86 images. Verdicts come from the SWE-bench harness (swebench 5.0.2), run after the cell for the scheduling experiments and inline by a thread pool for the loop experiments, where the verdict's arrival time is the delayed feedback under study; on 2xH200 a gold-patch guard grades two known instances before any cell is trusted. τ^2-bench runs the benchmark's own simulation with its user simulator and judge served by the local engines, the routed model playing the user of its own conversation, and its state checker as the verdict. Terminal-Bench 2.0 runs Terminus-2 through harbor 0.22.0, one harbor run per task, 24 concurrent tasks, 900 s per task. Program throughput counts the programs that submit their result (a patch, for SWE-bench) inside their budget, per hour of the cell's makespan. Accuracy counts a task as resolved only if its program finished inside its budget and the verifier accepted it.

B.6 Pre-registration

Every block, the frontier, the loop sweep, the scheduler comparison and each ablation, was written up in a dated file before its first cell ran, with the cells, the reading that would count as met, and a stop rule. Every constant in Table 2 was fixed before the first cell and never changed after. Blocks whose reading was not met are reported as such, for example the cold-start frontier (Appendix E.8).

Table 2: Constants, all fixed before the first cell.

Component	Constant	Value
router	LinUCB width α_{UCB}	0.6
router	tie	cheaper model
verifier selector	exemplars per class k	5
ACRouter orchestrator	memory neighbors / similarity floor	10 / 0.5
ACRouter orchestrator	cost weights	-0.1, -50
ledger	admission fraction / per-call ceiling	0.85 / 0.95
ledger	prior pseudo-observations	5
ledger	peak-context prior c_0 (27B / 9B)	22.6k / 44.7k tokens
ledger	run-time prior D_0 (27B / 9B)	642 / 639 s
divert rule	β, w_0	0.5, 600 s
ThunderAgent proxy	Δt, decay	5 s, 2^{-t}
static cap	admitted programs (27B / 9B)	27 / 42
agents	episode budget	900 s

C Extended Theoretical Analysis

C.1 Derivation of the Divert Rule

This appendix shows that the divert rule in Eq. 4 is the choice with the highest expected reward when the admission wait is priced in the same way that Eq. 3 prices cost. We first state three assumptions and check each one against our measurements. We then give the proof and two corollaries.

Assumption C.1 (Wait-priced reward)**.** *Let w be the admission wait of a program: the time from its arrival at the task boundary until its first request is served. We divide the wait by a reference wait $w_0 > 0$ and clip the result at one,*

$$\tilde{w} = \min\left(w/w_0,\ 1\right), \tag{5}$$

and define the wait-priced reward as

$$r_{\text{wait}} = (1-\beta)\, a - \beta\, \tilde{w}, \qquad \beta \in [0,1], \tag{6}$$

where $a \in [0,1]$ is the graded accuracy of the program. The clip keeps the penalty bounded for very long waits, as the clip on cost in Eq. 3 does. It has no effect while $w < w_0$.

Assumption C.2 (Forecast waits below the reference)**.** *At the task boundary, the reservation ledger of Appendix B.2 gives a forecast wait w_m for each candidate model m, and both forecasts are below the reference wait, $w_m < w_0$. In our runs $w_0 = 600$ s and the mean admission waits in Table 3 are between 86 and 130 s, so the assumption holds on average. Corollary C.2 covers the case $w_m \geq w_0$.*

Assumption C.3 (Selection estimate)**.** *For each model m, the router keeps an estimate $\hat{a}_m$ of the accuracy it expects from m on the current program. We use the bandit's selection index as $\hat{a}_m$. Because the bandit chose m_{ID} by this index, $\hat{a}_{m_{\text{ID}}} \geq \hat{a}_{m'}$ for every other model m'.*

Remark C.1 (Run time)**.** *The reward in Assumption C.1 prices the wait only. It does not price the run time of the program. Run time depends on the model and, under load, on the state of the engine (Section 5.4), and we do not model it here. The result below is therefore a statement about the wait-priced reward, not about end-to-end latency.*

Proposition C.1 (Divert rule)**.** *Let Assumptions C.1 to C.3 hold. Let m_{ID} be the model the bandit chose, and let m' be the other model, which has the shorter forecast wait. The model with the higher expected wait-priced reward is m' if and only if*

$$\beta\, \frac{w_{m_{\text{ID}}} - w_{m'}}{w_0} \ >\ (1-\beta)\left(\hat{a}_{m_{\text{ID}}} - \hat{a}_{m'}\right), \tag{7}$$

which is Eq. 4.

Proof. By Assumption C.2, the clip is inactive for both models, so $\tilde{w}_m = w_m/w_0$. Taking the expectation of r_{wait} with the estimates of Assumption C.3 gives, for each candidate m,

$$\mathbb{E}\left[r_{\text{wait}} \mid m\right] = (1-\beta)\,\hat{a}_m - \beta\,\frac{w_m}{w_0}. \tag{8}$$

The router should choose m' when $\mathbb{E}[r_{\text{wait}} \mid m'] > \mathbb{E}[r_{\text{wait}} \mid m_{\text{ID}}]$. Writing out both sides,

$$(1-\beta)\,\hat{a}_{m'} - \beta\,\frac{w_{m'}}{w_0} \ >\ (1-\beta)\,\hat{a}_{m_{\text{ID}}} - \beta\,\frac{w_{m_{\text{ID}}}}{w_0}. \tag{9}$$

Moving the wait terms to the left and the accuracy terms to the right gives Eq. (7). The left side is the wait the program saves, priced at β/w_0. The right side is the accuracy it gives up, priced at $1-\beta$. □

Corollary C.1 (Endpoints)**.** *At $\beta = 0$, Eq. (7) reduces to $\hat{a}_{m'} > \hat{a}_{m_{\text{ID}}}$. This never holds under Assumption C.3, so the router never diverts and acts as a quality-only bandit. At $\beta = 1$, the rule reduces to $w_{m'} < w_{m_{\text{ID}}}$, so the router always diverts to the shorter wait, whatever the accuracy. The sweep in Fig. 7(b) follows this path: $\beta = 0$ resolves the most tasks at the lowest rate, and a larger β trades resolved tasks for speed.*

Corollary C.2 (Saturated wait)**.** *If $w_{m_{\text{ID}}} \geq w_0$, the wait penalty of m_{ID} is clipped at β, and the rule becomes*

$$\beta\left(1 - \frac{w_{m'}}{w_0}\right) \ >\ (1-\beta)\left(\hat{a}_{m_{\text{ID}}} - \hat{a}_{m'}\right). \tag{10}$$

The largest accuracy gap the rule can accept is then $\beta/(1-\beta)$. At the deployed $\beta = 0.5$ this bound equals one, so the clip does not restrict the rule. It only keeps the penalty bounded.

Remark C.2 (Choice of w_0)**.** *Below saturation, the rule depends on β and w_0 only through the ratio $(1-\beta)\,w_0/\beta$: a wait of w_0 seconds trades against an accuracy gap of $\beta/(1-\beta)$. Sweeping β at a fixed w_0, as in Fig. 7(b), therefore also covers the choice of w_0.*

D CACHE REUSE AND LATENCY STATISTICS

Table 3 lists, for plan seed 912 of Fig. 6, the quantities behind panels (a), (c) and (d) together with the two the figure does not show: the admission wait the ledger imposes and the net change in a program's submit-to-done time against stock vLLM. Three points follow. First, output tokens per second rise under ORACLE at every concurrency level, 1.2–1.7× stock vLLM, because reserved programs decode at a high prefix hit rate instead of recomputing prefixes: the 27B's hit rate is 0.96, 0.90 and 0.75 against 0.87, 0.75 and 0.28 as concurrency grows. Second, the call latency that agents see is 1.4–1.8× lower on the mean and 1.5–2.5× lower at the tail. Third, the reservation's cost is the admission wait, 1.4–2.2 minutes per program; at 32 concurrent tasks nothing is contended, so the wait is pure delay and submit-to-done grows by 5.3 minutes, whereas at 64 concurrent tasks the faster calls repay it exactly and at 96 the program finishes 37 s sooner despite waiting 101 s at admission.

Table 3: Cache reuse and latency on the GH200 pair (SWE-bench Verified, 108 tasks, plan seed 912). Call latency is the mean and p95 over every model call; wait is the mean admission wait; Δ is the mean change in submit-to-done time against stock vLLM (95% task bootstrap).

concurrent tasks	system	27B hit	tok/s	call mean	call p95	wait	Δ submit→done
32	stock vLLM	.87	863	7.2 s	27.8 s	–	–
32	ThunderAgent	.87	829	7.2 s	28.9 s	–	−3 s
32	ORACLE	.96	1204	5.1 s	18.4 s	130 s	+316 s [+255, +379]
64	stock vLLM	.75	864	11.3 s	43.0 s	–	–
64	ThunderAgent	.72	884	11.2 s	40.7 s	–	−1 s
64	ORACLE	.90	1461	6.9 s	25.2 s	86 s	+6 s [−55, +66]
96	stock vLLM	.28	1049	15.5 s	72.0 s	–	–
96	ThunderAgent	.29	1063	15.5 s	64.6 s	–	+17 s
96	ORACLE	.75	1237	8.8 s	28.6 s	101 s	−37 s [−92, +18]

E Additional Results and Ablations

E.1 Decomposing DISC

Table 4 lists the five systems behind Fig. 7(a) (GH200 pair, plan seed 912), adding the two baselines and the static per-model cap of Fig. 6, which admits at most 27 programs on the 27B and 42 on the 9B, values fixed from plan seed 911. All three reservation-based systems keep the 27B's prefix hit rate above 0.75 at 96 concurrent tasks, where stock vLLM and ThunderAgent fall below 0.3, and all three resolve two to three times as many tasks. They differ in speed: DISC resolves 130 tasks per hour against 94 with admission only and 100 for the cap, at a similar resolved count, because a program forecast to wait on the 27B is dispatched to the idle 9B instead. The cap also needs its two values chosen per pool and workload; the ledger learns each program's peak context online.

Table 4: DISC decomposed on the GH200 pair, SWE-bench Verified, 108 tasks, plan seed 912.

	64 concurrent tasks			96 concurrent tasks		
	resolved	res./h	27B hit	resolved	res./h	27B hit
stock vLLM	40	82	.75	22	66	.28
ThunderAgent	41	89	.72	20	61	.29
static cap (27×42)	56	112	.84	57	100	.86
ORACLE (admission only)	54	99	.80	51	94	.80
ORACLE	57	159	.90	52	130	.75

E.2 Other Pools and Model Families

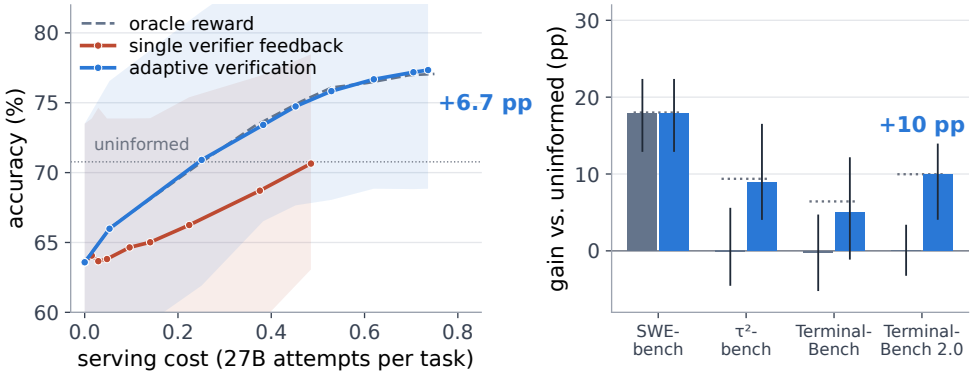
Table 5 extends the comparison of Fig. 6 to 96 concurrent tasks, in resolved tasks per hour rather than program throughput, on a node with two H200s (141 GB each), with the Qwen pair and with a second family, Devstral-Small-2-24B and Ministral-3-14B, and on a single GH200 hosting both models. The direction holds on every pool. The Mistral pair resolves 29 tasks per seed under ORACLE against 17 under ThunderAgent and 16 under stock vLLM (+27 of 216 [+12, +42] over stock), at 1.2–1.3× their rate and with only 70 s of added submit-to-done time. On the single GPU, where the two models share one KV cache, the gap is largest and appears at 32 concurrent tasks already: 41 resolved against 22 and 23. On the H200 pair, whose larger cache contends least, the gap is smallest, 1.3–1.5×. The gain scales with the memory pressure the workload creates.

Table 5: ORACLE against ThunderAgent and stock vLLM at 96 concurrent tasks on other pools and models: resolved tasks per hour, mean of two plan seeds (single-GPU row: 32 concurrent tasks, one seed).

	resolved / h		ORACLE vs.	
Pool and models	ThunderAgent	ORACLE	ThunderAgent	stock
2×GH200, Qwen (Fig. 6)	60	127	2.1×	1.9×
2×H200, Qwen	198	265	1.3×	1.5×
2×H200, Devstral-24B / Ministral-14B	56	69	1.2×	1.3×
1×GH200, both models (32 concurrent tasks)	24	41	1.7×	1.6×

E.3 Verifier per Family

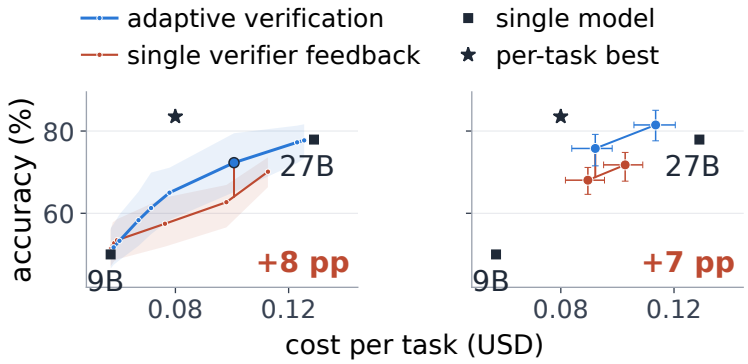
Fig. 8(b) is the replay behind Table 1, each family alone at cost weight 0. With the single patch verifier the gain over uninformed routing is -0.2, -0.2 and $+0.1$ pp on τ^2-bench, Terminal-Bench and Terminal-Bench 2.0: the router receives the same verdict for every outcome and learns nothing. The adaptive verifier gains +8.9 [+4.1, +16.4], +5.1 [-1.0, +12.1] and +9.9 [+4.2, +13.8] pp, within 1.5 pp of the best fixed verifier for each family, and matches it on SWE-bench (+17.9 pp), the one family the patch verifier reads. Panel (a) shows the same effect on a mixed τ^2+SWE stream against the share of tasks sent to the 27B: single-verifier feedback tops out at 71% success, adaptive verification reaches 77% and stays above it at every share, within 1 pp of oracle reward.



(a) Routing quality (τ² + SWE-bench) (b) Gain by family, α = 0

Figure 8: **A single patch verifier loses every family it cannot read.** Choosing the verifier per task recovers +5 to +10 pp on τ^2-bench and both Terminal-Bench versions and loses nothing on SWE-bench. Both routed curves use the bandit selector.

E.4 Adaptive Verification on the τ^2 + SWE Stream



(a) Bandit (b) ACRouter

Figure 9: Adaptive verification versus single-verifier feedback on the τ^2+SWE stream with (a) the bandit selector and (b) ACRouter's agent selector, in the layout of Fig. 4.

Fig. 9 repeats Fig. 4 on the stream of 276 τ^2-bench and 100 SWE-bench Verified tasks, without Terminal-Bench 2.0. The single models reach 50.0% accuracy at \$0.057 per task (9B) and 77.9% at \$0.129 (27B); the per-task best is 83.5%. With the bandit selector, adaptive verification is +8.3 pp [+2.7, +15.1] above single-verifier feedback at the matched cost of \$0.101. With ACRouter's orchestrator, adaptive verification reaches 75.8% [71.5, 79.2] at \$0.092 and 81.5% [77.6, 85.0] at \$0.114 at the two cost settings, against 68.1% [64.6, 71.2] at \$0.090 and 71.8% [67.8, 74.8] at \$0.103 with single-verifier feedback, a gain of +7.0 pp [+2.7, +10.4] at matched cost. Both gains are slightly larger than on the three-family stream of Fig. 4, where τ^2-bench tasks, the ones a patch verifier cannot read, make up a smaller share of the stream.

E.5 Frontier on the SWE + Terminal-Bench 2.0 Stream

Fig. 10 is the third stream, the two families whose verifiers run in containers; Fig. 4 and Fig. 9 show the other two. The single models span 20% at \$0.127 per task and 52% at \$0.235; ORACLE spans 25–51% for \$0.130–\$0.227 and sits +6.4 pp [+2.9, +12.5] above random mixing at the midpoint cost. Single-verifier feedback trails by 2–5 pp at the same cost weight.

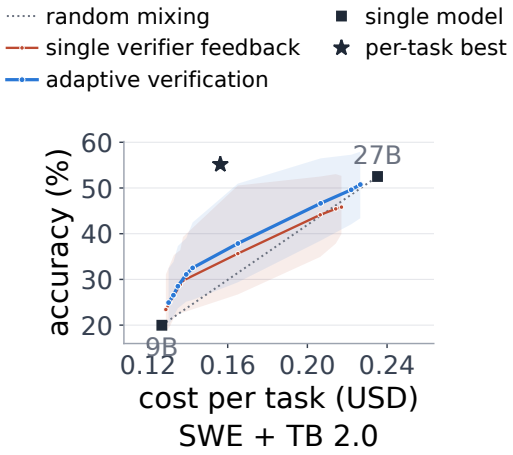


Figure 10: **Accuracy–cost frontier on the SWE + Terminal-Bench 2.0 stream.** The bandit selector with adaptive verification sits +6.4 pp above random mixing at the midpoint cost; both routed curves use the bandit selector.

E.6 ACROUTER'S OWN POLICY

Fig. 4(b) and Fig. 9(b) run ACRouter's own model selector. This section gives its setup, then two further measurements of the released system.

Hosted orchestrator. ACRouter's orchestrator is an LLM that chooses the model for each task from a prompt. We serve Qwen3.5-4B with vLLM on a GH200 and build the prompt as the released code does: the task's initial request, the cost-tiered model descriptions, a complexity tier from LiteLLM's rule-based classifier, and the nearest past outcomes from ACRouter's memory (up to 10 neighbors with cosine similarity of at least 0.5 between bge-m3 embeddings of the initial requests). The orchestrator answers with a JSON decision; a decision costs about 660 prompt and 7 completion tokens, and no call fell back to a default choice. Each run replays eight passes over the stream for eight seeds, one orchestrator call per task and pass. The memory starts empty for each seed and persists across passes, so from the second pass a repeated task finds its own earlier verdict among its neighbors; this recall is what lifts the orchestrator's accuracy at its released cost setting above the 27B alone. The two cost settings are the released cost weight (-0.1) and a cost-penalized one (-50): the released weight multiplies a few cents against a score in $[0, 1]$ and barely moves the decision, so the second setting is needed for a second point. Adaptive verification and single-verifier feedback differ only in the verdict written to the memory, so the comparison within each panel (b) is paired.

Released memory policy in the verifier replay. Substituting ACRouter's released `route()` rule (per-dimension mean of past rewards, arg-max, ties to the cheap chain) for LinUCB in the replay of Fig. 4, on the τ^2+SWE pool of 65 tasks it reaches 56.0% task success with its fixed verifier and 67.0% with the adaptive verifier, against 64.4% and 72.7% for LinUCB under the same two verifiers and 64.6% for uninformed routing. On two larger pools (100 τ^2 + 100 SWE tasks each) it stays within 1 pp of uninformed routing under either verifier. The memory rule needs many verdicts per dimension before its means separate, which these streams do not supply.

Earlier reconstruction as one operating point. An earlier reconstruction of the complete system, an LLM orchestrator prompted with the memory's nearest neighbors and cost-tiered model descriptions, warm-started on a 30% probing split as in its paper, lands on the per-domain accuracy–cost frontiers at one point each: 77.3% at \$0.035 on retail, 82.7% at \$0.082 on telecom, 82.5% at \$0.032 on airline, 61.6% resolved at \$0.31 on SWE-bench. Each point is on or near the frontier that ORACLE traces, but it is a single point: the published cost weight is inert over its released range, which is why the hosted runs above add a cost-penalized setting. Its routing decision costs 209 ms at the median against 0.56 ms for ORACLE's, on the critical path of every program. Two caveats: the orchestrator is a prompted 4B model rather than their fine-tuned 0.8B adapter, and the points are

scored on the held-out 70% split, which is 2.5–3 pp more favorable than the full pools the other curves use.

Released fine-tuned router. Running the published 0.8B LoRA router itself, greedily, with its prompt and its eight backend names mapped to our models by its own tier labels, on 274 τ^2 tasks gives -5.1 pp $[-8.8, -1.5]$ against the best single model; it uses two or three of its eight names. The published adapter does not reproduce the decisions in the paper's own traces (7% per-task agreement on its 112 code tasks), so this bounds the release, not the paper.

E.7 Exploration Width

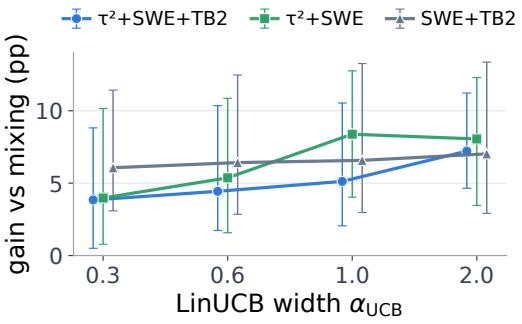


Figure 11: **The gain over random mixing survives every exploration width.** Replay of Fig. 4 at four LinUCB widths, ORACLE minus random mixing at the midpoint cost (95% task bootstrap); the main figures use the pre-registered $\alpha_{\mathrm{UCB}} = 0.6$.

Fig. 11 sweeps the LinUCB width α_{UCB} in the frontier replay of Fig. 4, and Table 6 gives the intervals. Every interval excludes zero. Wider exploration helps on the two τ^2-containing streams (+3.8 to +7.2 pp on τ^2+SWE+TB 2.0, +4.0 to +8.4 pp on τ^2+SWE) and is flat on SWE + TB 2.0 (+6.1 to +7.0 pp). The main figures keep the pre-registered $\alpha_{\mathrm{UCB}} = 0.6$.

Table 6: ORACLE minus random mixing at the midpoint cost (pp, 95% task bootstrap) by LinUCB width α_{UCB}.

stream	$\alpha_{\mathrm{UCB}} = 0.3$	$\alpha_{\mathrm{UCB}} = 0.6$	$\alpha_{\mathrm{UCB}} = 1.0$	$\alpha_{\mathrm{UCB}} = 2.0$
τ^2 + SWE + TB 2.0	+3.8 [+0.5, +8.8]	+4.4 [+1.7, +10.4]	+5.1 [+2.1, +10.5]	+7.2 [+4.6, +11.2]
τ^2 + SWE	+4.0 [+0.8, +10.2]	+5.4 [+1.6, +10.9]	+8.4 [+4.0, +12.8]	+8.1 [+3.5, +12.3]
SWE + TB 2.0	+6.1 [+3.1, +11.4]	+6.4 [+2.9, +12.5]	+6.6 [+3.0, +13.3]	+7.0 [+2.9, +13.4]

E.8 Cold Start

Fig. 4 reports the router's steady state after eight passes over the stream. Replayed cold, in a single pass with family and domain context only, ORACLE coincides with random mixing of the two models: +0.9 pp $[-1.4, +4.3]$ at the midpoint cost on the τ^2+SWE+TB 2.0 stream, +1.6 $[-1.3, +5.7]$ on τ^2+SWE and -0.8 $[-2.1, +2.6]$ on SWE+TB 2.0. The frontier gain therefore emerges with accumulated task-level feedback; a single cold pass over the stream is insufficient to reach the steady-state improvement reported in Fig. 4.

E.9 Pending Term in the Width

Delayed credit counts every decision made so far in the confidence width, landed or not; the alternative counts only landed verdicts. Live on the GH200 pair (108 tasks, stream seed 921, DISC stack), removing the pending term costs 7 resolved tasks at 32 concurrent tasks (64 versus 71) and 4 at 64 (54 versus 58) at the same rate, with intervals of $[-3, +17]$ and $[-4, +12]$ that include zero: on one seed the term is not separable. Its effect is visible in the routing rather than the totals: without it the

width collapses while verdicts are in flight and the router commits earlier to the cheaper 9B (27B share 0.65 against 0.76 at 32 concurrent tasks, 0.28 against 0.53 at 64).