LIFTING CHEAPER MODELS: SELF-EVOLVING ROUTING FOR COST-EFFICIENT AGENT SERVICES

Anonymous authors
Paper under double-blind review

ABSTRACT

As agent services built on large language models (LLMs) move into production, their cost accrues with every request, much of it the premium for top models. Model routing lowers this premium by serving a request with a cheaper model whenever one matches the model the user asked for, but it is confined to the models already in the pool. When every cheaper model fails a recurring task, the router can only pay for the expensive one or accept worse results, and its savings stop at the capability boundary of the cheaper models. We observe that these failures are rarely random: a cheaper model tends to break at the same step across tasks of the same kind, and the execution history already records how a stronger model gets past it. We introduce `RouteSage`, a self-evolving router that edits its own routing table through capability patches, validated entries that let a cheaper action serve a family of tasks on behalf of the requested model. A direct patch reroutes the family to a cheaper model that already suffices. A guided patch pairs a cheaper model that falls short with an Experience, a reusable procedure distilled from past executions, creating an action the pool never had without retraining any model. Direct patches recover savings that existing models allow, and guided patches lift the models that fall short. Across web interaction, tool use, and code generation, `RouteSage` exceeds requested-model quality in every domain while lowering total online cost, including the cost of building and validating every patch, and on MiniWoB++ it cuts online cost by $\downarrow$29.6% and raises success by $\uparrow$3.4 points. Served from a frozen state, it delivers the highest quality of any method in every setting and cuts cost by $\downarrow$57.4% when GPT-5.6-sol is requested on MiniWoB++, whereas no baseline saves more than 5.2% without losing quality.

1 INTRODUCTION

Agent services built on large language models (LLMs) are moving from research prototypes to production, filling in forms, orchestrating tools, and writing code on users' behalf (Yao et al., 2023; Schick et al., 2023; Zhou et al., 2024; Jimenez et al., 2024). Each request pays for a multi-step loop of model calls, and the strongest models charge a steep premium on each: on our tool-use workload, serving with GPT-5.6-sol (OpenAI, 2026) costs nearly 20 times as much as with Qwen3.5-35B-A3B (Qwen Team, 2026). Model routing cuts this premium by sending each request to the cheapest model that suffices (Chen et al., 2024a; Ding et al., 2024; Ong et al., 2025; Chen et al., 2024b; Feng et al., 2025). Yet routing treats what each model can do as fixed, even though a deployed service watches its models succeed and fail on thousands of recurring tasks and could learn from each. We ask whether a service can become cheaper the longer it runs, without retraining any model.

We study services in which each request names the model that should serve it, and that model sets the quality the user expects. Consider a reimbursement workflow (Figure 1). A cheap model fills in every field correctly but never confirms the submission, while the requested model checks the confirmation page and repairs what is missing. Seeing this, a router sends the next reimbursement to the expensive model, and the next, and the next. For every task type, what the cheaper models can do on their own sets a floor on cost. We call this limit the capability boundary of the cheaper models. Near the boundary, routing becomes a trade. A router can hold quality by paying for the expensive model, as cascades and exploratory bandits do (Chen et al., 2024a; Aggarwal et al., 2024; Hu et al., 2025), or cut cost by accepting worse results, as threshold routers do (Ding et al., 2024; Ong et al., 2025). A service-level quality target (Woisetschläger et al., 2025; 2026) only picks where on this

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.

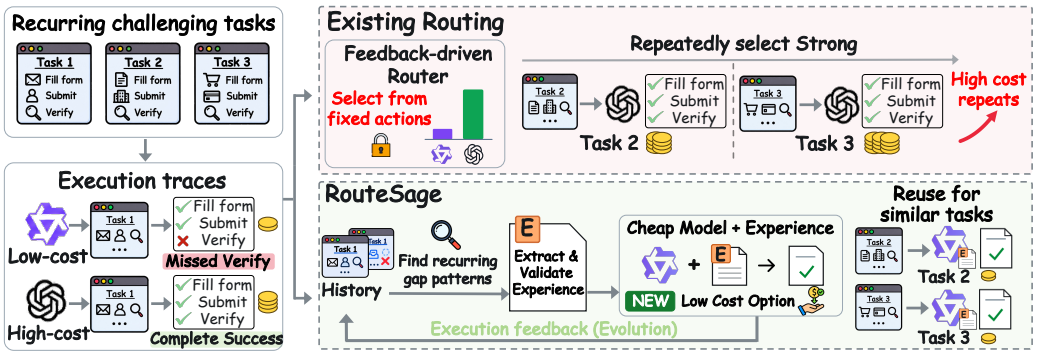


Figure 1: **Motivation of RouteSage.** Recurring failures lock a task family into expensive execution. Left: on tasks from one family, a low-cost model fails where a high-cost model succeeds. Top right: a fixed-pool router is reinforced toward the strong model and pays the high cost on every later task. Bottom right: RouteSage finds the recurring failure pattern, distills it into an Experience, and after validation adds the cheap model with it as a new low-cost action, a guided patch.

trade to sit. The cheaper models set the trade, not the router: a better router finds a better point among the same models but never makes a cheaper model stronger. That takes lifting the model.

Must the cheap model stay locked out? Its failures suggest otherwise. Across reimbursements, the employee, the amount, and the receipts change, yet the cheap model keeps skipping the same final check, and the requested model's trace shows exactly what that check involves. Written down as a procedure, the missing step becomes guidance for every future reimbursement. Recurring failures therefore carry reusable, model-specific knowledge, which current routers use only to lower a model's score. Turning this knowledge into savings raises two questions: ❶ *Where does a procedure carry over, and when may a cheaper action serve on behalf of the requested model?* Tasks that look alike can require different procedures, and a substitute that falls short costs the user quality. ❷ *How can a router create a cheaper action that its pool does not have?*

Building on this view, we present RouteSage, a router that lifts cheaper models past their boundaries while serving (Figure 2). The router executes actions, each a model optionally paired with a procedure, and RouteSage patches the routing table that decides which action serves which request. A capability patch is a validated entry stating that, for one family of tasks, a cheaper action may replace the requested model. A direct patch points to an action the pool already has. A guided patch points to a new one: the cheaper model equipped with an Experience, a procedure distilled by contrasting its failures with the requested model's successes. For ❶, RouteSage groups tasks into Capability Regions that share a procedure, so that, as in case-based reasoning (Kolodner, 1992; Aamodt & Plaza, 1994; Guo et al., 2024), Experiences are retained and reused only where they apply, and it admits a patch only after audits on fresh tasks find its action within the statistical resolution of the requested model and cheaper. For ❷, it builds guided patches where recurring demand can repay the effort, and a guided patch must also beat the same model without its Experience. Because patches live in a model's context rather than its weights, the approach applies equally to API-only models.

We evaluate RouteSage on a mixed stream of web interaction on MiniWoB++ (Shi et al., 2017; Liu et al., 2018)[1], multi-turn tool use on T1 (Chakraborty et al., 2025)[2], and iterative code generation on CodeFlowBench (Wang et al., 2026b)[3]. With every construction and validation call billed, RouteSage exceeds requested-model quality in all three domains while lowering total online cost. On MiniWoB++, it cuts cost by ↓29.6% and raises success by ↑3.4 points. It is cheaper and better than FrugalGPT and PAK-UCB everywhere, and MESS+, the only baseline that spends less, drops T1 tool-calling F1 by ↓21.2 points. Served from a frozen state, it delivers the highest quality in all twelve domain–tier settings and cuts cost by ↓57.4% when GPT-5.6-sol is requested on MiniWoB++, while no baseline saves more than 5.2% without losing quality. As the service runs, direct patches level off early, and the savings keep growing with guided patches. Without guided patches,

[1] https://github.com/Farama-Foundation/miniwob-plusplus
[2] https://github.com/CapitalOne-Research/T1
[3] https://github.com/Rise-1210/codeflow

T1 savings fall by 12.5 points and quality drops below the requested model on two domains: the trade returns. Our main contributions are summarized as follows:

- ***Capability-Boundary Perspective.*** We identify the capability boundary of cheaper models as what forces fixed-pool routers to trade cost against quality, and recast routing as the growth of the routable action set, with the requested model as the quality bar.
- ***Self-Evolving Routing Framework.*** We present `RouteSage`, which discovers where procedures transfer, routes under a quality constraint, and turns recurring failures into validated capability patches without touching model weights.
- ***Empirical Validation.*** Across three domains, `RouteSage` attains the best held-out quality in every setting at lower cost, with savings that no fixed-pool router matches at requested quality.

2 Related Work

Model and Agent Routing. FrugalGPT, Hybrid LLM, RouteLLM, and MixLLM learn cascades or query-dependent choices between models of different cost (Chen et al., 2024a; Ding et al., 2024; Ong et al., 2025; Wang et al., 2025a), and later routers score models with contrastive embeddings or graphs (Chen et al., 2024b; Feng et al., 2025), or escalate on self-verification and answer consistency (Aggarwal et al., 2024; Yue et al., 2024). PAK-UCB, MESS+, and PILOT learn online from prompt-aware utility estimates, service-level quality targets, or budget constraints (Hu et al., 2025; Woisetschläger et al., 2025; Panda et al., 2025; Poon et al., 2026). For multi-step agents, EvoRoute and SkillOrchestra bring execution experience and skill-level competence into selection, and recent routers also choose strategies or act as agents themselves (Zhang et al., 2026; Wang et al., 2026a; Pan et al., 2026; Zhou et al., 2026). These methods refine the choice while the actions stay fixed. They differ in where they settle on the cost–quality trade, not in whether they face it. `RouteSage` instead makes a model-specific procedure a routable action, validated against the requested model.

Execution Experience and Procedural Reuse. Reflexion keeps verbal feedback, ExpeL extracts cross-task insights, and Agent Workflow Memory induces reusable workflows (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2025b). Voyager and Synapse store skills or trajectories for later retrieval (Wang et al., 2024; Zheng et al., 2024), Dynamic Cheatsheet and Trace2Skill distill lessons into reusable guidance (Suzgun et al., 2026; Ni et al., 2026), and ReMe and SkillOpt refine procedural memory or external skills through validation feedback (Cao et al., 2026; Yang et al., 2026), in the spirit of the retrieve–reuse–revise–retain cycle of case-based reasoning (Aamodt & Plaza, 1994). This work shows that guidance distilled from traces can change how well a model executes, with the goal of helping an agent succeed more often. A routing platform must also know which model the guidance is written for, whether the guided cheaper model can stand in for a more expensive one, and whether the savings repay its cost. `RouteSage` answers these questions inside the routing decision, and what it retains is a patch with a defined scope, measured quality, and measured cost. Appendix E discusses further related work.

3 The Proposed Framework: RouteSage

3.1 Problem Formulation

Our setting differs from standard routing in two ways: quality is judged relative to the model each request names, and the action set grows over time at a cost charged to the same objective as serving.

Requests arrive as (x_t, r_t), where x_t is the task and $r_t \in \mathcal{M}$ is the requested model. An action $a = (m, E)$ pairs an executor m with an optional Experience E, which is supplied throughout the agent loop, including any intermediate model or tool calls. Raw actions carry no Experience, $\mathcal{A}^{\text{raw}} = \{(m, \varnothing) : m \in \mathcal{M}\}$, and $a^r = (r, \varnothing)$ executes the requested model. We index models by fixed base-price tiers, $c^0_{m_1} \geq \cdots \geq c^0_{m_M}$. The action set starts with raw actions only, and a request may use either the requested model or a registered action whose executor is strictly cheaper:

$$\mathcal{A}_t = \mathcal{A}^{\text{raw}} \cup \mathcal{A}^{\text{exp}}_t, \quad \mathcal{A}^{\text{exp}}_0 = \varnothing, \qquad \mathcal{D}_{r,t} = \{a^r\} \cup \{a = (m, E) \in \mathcal{A}_t : c^0_m < c^0_r\}. \tag{1}$$

Base prices set eligibility, while measured cost also reflects tokens, Experience length, and fallback.

Each execution records a task identifier, action version, trajectory, feedback, and cost. Feedback gives completion signals or diagnostics such as compiler errors, never target solutions. Terminal

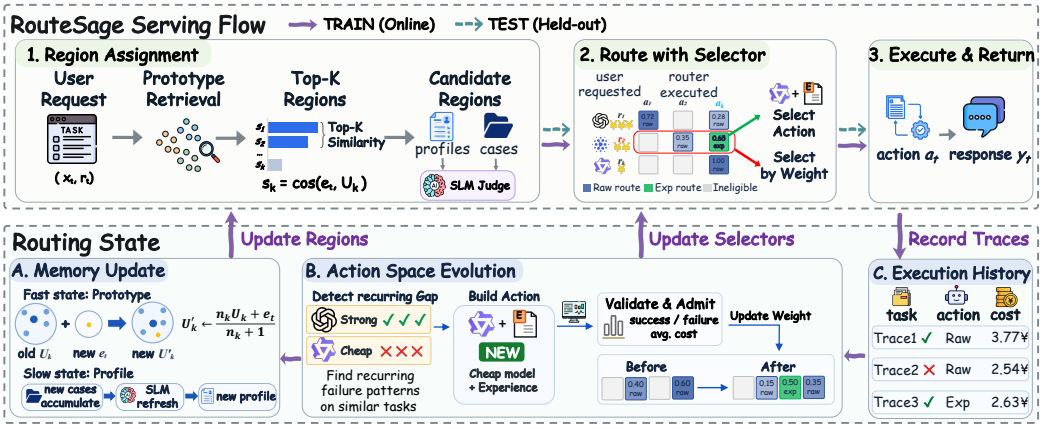


Figure 2: **Overview of RouteSage.** Top: a request (x_t, r_t) is matched to a Region by prototype retrieval and SLM verification, and the Region's selector samples an action from row r_t. Bottom: traces update Region memory on two timescales, refresh selector statistics, and supply the gap evidence for building new actions and validating them as patches. Solid arrows form the online path, which also updates the state, and dashed arrows form the held-out path, which only serves.

outcomes are scores $z \in [0,1] \cup \{\perp\}$, where $\perp$ marks unresolved feedback, and intermediate errors serve as diagnostics. Repeated executions of one task yield linked traces, each with its own outcome.

For a task population $\mathcal{X}$ and requested model r, let $Q_{\mathcal{X},r}(\pi)$ and $Q^{\text{req}}_{\mathcal{X},r}$ denote the expected delivered quality under routing policy π and under always executing the requested model. The policy controls both routing and the growth of the action set so as to minimize total cost:

$$\begin{aligned} \min_{\pi} \quad & \mathbb{E}\left[\frac{C^{\text{serve}}_{1:T} + C^{\text{evolve}}_{1:T}}{T}\right] \\ \text{s.t.} \quad & Q_{\mathcal{X},r}(\pi) \geq Q^{\text{req}}_{\mathcal{X},r} \quad \forall \text{ evaluated } (\mathcal{X}, r), \qquad a_t \in \mathcal{D}_{r_t,t}. \end{aligned} \tag{2}$$

The requested model is the quality bar itself, not a reference to trade against: a cheaper action may serve a request only if it does as well on average. Serving cost covers assignment, execution, and fallback, and evolution cost covers memory maintenance, construction, and validation, including candidates never admitted. Online service updates the routing state from feedback, whereas frozen held-out service runs on the accumulated state without updates or recovery.

Overview of RouteSage. RouteSage addresses Eq. (2) with three mechanisms in one service loop (Figure 2). Capability Regions decide where a procedure applies, by grouping tasks that require the same procedure. A quality-constrained selector decides which action serves each request, spending as little as possible at the requested model's estimated quality. Capability expansion decides what the routing table gains next: it reads execution gaps from regional traces, builds direct or guided patches, and admits them only after paired audits on fresh tasks.

3.2 Capability Regions: Discovering Where Experience Transfers

Regions and Retrieval. Tasks that require the same procedure, with the same prerequisites, operations, and completion conditions, form a Region, and an Experience is reused only within its Region. A Region $R_k = (U_k, P_k, \mathcal{H}_k)$ consists of an embedding prototype U_k for fast retrieval, a profile P_k describing the shared execution conditions, and a set of linked execution traces $\mathcal{H}_k$. Membership depends on what a task requires, not on which model was requested, so traces from cheap and expensive executors on the same procedure accumulate side by side, ready to be contrasted when new actions are built. Given $e_t = \text{Embed}(x_t)$ from a frozen encoder, retrieval proposes Regions by

$$\mathcal{K}_t = \text{TopK}_{k \in \mathcal{R}^{\text{active}}_t} \cos(e_t, U_k). \tag{3}$$

Embedding similarity is a useful first filter, but requests that look alike can still require different procedures. A frozen small language model (SLM) therefore checks procedural compatibility against each candidate's profile and representative cases. Unmatched requests go to the requested model.

Two-Timescale Memory. Each new task updates the prototype at once,

$$U_k \leftarrow U_k + \frac{e_t - U_k}{n_k + 1}, \qquad n_k \leftarrow n_k + 1, \tag{4}$$

and its linked traces enter $\mathcal{H}_k$. The profile P_k is rewritten only when the Region's trace count doubles, consolidating recurring execution conditions and diagnostics. This lets the memory absorb new traffic quickly, while the scope in which Experiences are reused shifts only as evidence accumulates.

Region Activation. A Region starts as a provisional group and becomes active once the evidence shows, at the 95% level, that its profile accepts most subsequent requests. Until then its traffic stays on the requested model, and requests that later prove incompatible return to provisional memory instead of inheriting existing actions. Frozen service uses only active Regions (Appendix B.1).

3.3 Quality-Constrained Adaptive Routing

Routing Table. The selector decides which action serves each request. It spends as little as possible while keeping average quality no lower than the requested model's. Within each active Region k and for every requested model r, it keeps retained actions $\mathcal{Q}_{k,r}$ and trial actions $\mathcal{U}_{k,r}$, the actions of candidate patches still under audit, all within the cost-feasible set $\mathcal{D}_{r,t}$. Initially $\mathcal{Q}_{k,r} = \{a^r\}$ and $\mathcal{U}_{k,r} = \varnothing$, so every request runs on the model it names. We view the selector of a Region as a routing table: rows are requested models, columns are executors in decreasing price order, and raw and Experience actions hold separate weights within each column. The table starts as the identity and stays upper triangular (Lemma C.3). As cheaper actions earn trust, traffic leaves the diagonal.

Action Statistics. For each action, the selector tracks executions, resolved outcomes, quality, and cost. With summed outcome s_a over n_a resolved direct outcomes, the smoothed quality estimate is $\widehat{q}_a = (s_a + 1/2)/(n_a + 1)$, and unresolved feedback updates history and cost only. The mean serving cost $\widehat{c}_a$ includes Experience context, assignment, and fallback. Audits initialize the estimates of new actions, and the requested model's estimate comes from the traffic it serves and from audits. Below, we suppress the indices (k, r, t) and write $\mathcal{F} = \mathcal{Q}$ for the actions that serve regular traffic.

Quality-Constrained Allocation. Each row solves an empirical surrogate of serving in Eq. (2):

$$\min_{\mathbf{w} \in \Delta(\mathcal{F})} \quad \sum_{a \in \mathcal{F}} w_a \frac{\widehat{c}_a}{\widehat{c}_r} - \tau_t H(\mathbf{w}) \qquad \text{s.t.} \quad \sum_{a \in \mathcal{F}} w_a \widehat{q}_a \geq \widehat{q}_r. \tag{5}$$

Here $\Delta(\mathcal{F})$ is the probability simplex, $H(\mathbf{w}) = -\sum_a w_a \log w_a$ is the entropy, and the temperature $\tau_t = 1/\sqrt{1 + n_{k,r}}$ decays with the $n_{k,r}$ resolved outcomes of the row, so exploration fades with evidence. A multiplier $\lambda \geq 0$ on the quality constraint yields an action score and weights within a group $\mathcal{G}$:

$$u_a = \lambda \widehat{q}_a - \frac{\widehat{c}_a}{\widehat{c}_r}, \qquad w(a \mid \mathcal{G}) = \frac{\exp(u_a/\tau_t)}{\sum_{b \in \mathcal{G}} \exp(u_b/\tau_t)}. \tag{6}$$

Regular traffic uses $\mathcal{G} = \mathcal{Q}$, and audit tasks use $\mathcal{G} = \mathcal{U}$ to pick the next candidate. The multiplier is the price of quality in units of requested-model cost, set to the smallest $\lambda \geq 0$ at which the weighted quality meets the target, or to ∞ when no finite price does, which keeps the row on its best-estimated actions (Appendix B.2). Either way, the weights solve Eq. (5) exactly (Proposition C.1).

Execution and Freezing. Online, the selector samples from the retained actions of the row. When a cheaper action ends in a detected terminal failure, the request falls back to the requested model. The two traces are linked but keep their own outcomes, so a successful recovery never inflates the failed action's direct quality. Fallback needs a restorable state or safe replay, and tasks without one run on the requested model from the start. Frozen held-out service executes, without updates or recovery, the cheapest retained action with $\widehat{q}_a \geq \widehat{q}_r$, which always exists because a^r qualifies.

3.4 Capability Expansion: Turning Execution Gaps into Capability Patches

The routing table grows only through patches. A patch for Region k and requested model r is a validated entry stating that an action $a = (m, E)$ with $c^0_m < c^0_r$ may replace the requested model there. A direct patch uses a raw action ($E = \varnothing$) that the pool already contains, while a guided patch uses an Experience action, which the pool does not contain until `RouteSage` builds it.

Reading Regional Evidence. Traces of cheaper executors come from the requests that name them, so a Region shared across tiers gathers evidence on several executors for free. At each profile refresh, `RouteSage` compares the Region's raw, Experience, and requested-model traces by count, resolved quality, cost, and recurring diagnostics. A cheaper executor whose estimated quality already matches the requested model yields a candidate direct patch. One that keeps failing in the same way, skipping the same verification step or missing the same tool call, yields a candidate guided patch with an Experience for that gap. The two kinds respond to different evidence and follow no fixed order.

Building an Experience. An LLM builder distills the procedure from the Region profile and the execution history available at version cutoff t_v:

$$E^{(v)} = \mathrm{Build}\big(m, P_k, \mathcal{H}^{\mathrm{construct}}_{k,\le t_v}\big), \qquad a^{(v)} = (m, E^{(v)}). \tag{7}$$

Linked traces, where a cheap and an expensive executor attempted the same instance, expose execution differences directly, and traces from related tasks show which differences recur across the Region. The builder turns this evidence into applicability conditions, ordered operations, conditional recovery steps, and completion checks, replacing instance-specific entities with task roles so the procedure generalizes. Each Experience is bound to its target executor, Region, and version.

Validating and Admitting a Patch. A candidate patch enters the selector as a trial action that runs only on audit tasks: requests the selector has already sent to the requested model while the row holds candidates, so each audit is chosen before its outcome is known. Audits thus concentrate on rows that still rely on the requested model, where most savings remain. On them the candidate and its comparators start from the same state. The separation matters because construction evidence is selected by failure: evaluated on the tasks that motivated it, almost any fix would look effective. Let $\mathcal{T}_{a,r}$ denote the audits so far and $\ell_\alpha(a,b)$ the lower 95% posterior bound ($\alpha = 0.05$) on the difference in expected direct quality between actions a and b, and $u_\alpha(a,b)$ the upper one. Let h_r be half the gap between the lower and upper 95% posterior bounds of q_r in the row, the resolution at which the requested model's own quality is known. A patch with $a = (m, E)$ and raw counterpart $a^{\mathrm{raw}} = (m, \varnothing)$ is admitted if

$$\ell_\alpha(a, a^r) \ge -h_r, \qquad \widehat{c}^{\,\mathrm{dir}}_a < \widehat{c}^{\,\mathrm{dir}}_r, \qquad E \neq \varnothing \implies \ell_\alpha(a, a^{\mathrm{raw}}) > 0. \tag{8}$$

The first condition keeps any shortfall the audits still allow within the resolution of the bar itself. Since h_r shrinks as the requested model is observed, the test tightens toward exact parity (Proposition C.2). Quality is enforced on estimates where traffic is allocated (Eq. (5)): an action estimated below $\widehat{q}_r$ serves traffic only where better actions offset it. The second condition requires lower direct cost, Experience context included. The last applies to guided patches only: the Experience must make its executor better than the same model without it, or a direct patch would do the same job without the extra context. The test is sequential. After each audit the bounds are recomputed, and the patch is admitted as soon as Eq. (8) holds and rejected as soon as $u_\alpha(a, a^r) < -h_r$ or, for a guided patch, $u_\alpha(a, a^{\mathrm{raw}}) \le 0$, so the number of audits adapts to how clear the evidence is. Once admitted, a patch moves its action from $\mathcal{U}_{k,r}$ to $\mathcal{Q}_{k,r}$, scoped to the tested Region and requested model. It is suspended if its online estimate later shows, with 95% posterior probability, that it falls below the requested model (Appendix B.3).

Paying for Patches. `RouteSage` builds a candidate patch only when its projected saving on past demand exceeds its building and audit cost, which enters C^{evolve} in Eq. (2) (Appendix B.3).

4 Experiments

4.1 Experimental Setup

Benchmarks and Backbones. We evaluate on web interaction (MiniWoB++ (Shi et al., 2017; Liu et al., 2018), Success), multi-turn tool workflows (T1 (Chakraborty et al., 2025), mean tool-calling F1), and iterative code generation (CodeFlowBench (Wang et al., 2026b), Pass@1), whose tasks follow recurring procedures and return automatic feedback. The executor pool holds five backbones in increasing price order: Qwen3.5-4B, Qwen3.5-35B-A3B, Qwen3.5-122B-A10B (Qwen Team, 2026), GLM-5.2 (GLM-5 Team, 2026), and GPT-5.6-sol (OpenAI, 2026). Qwen3.5-4B also serves as the SLM and Qwen3.5-35B-A3B as the builder (Eq. (7), Appendix D).

Protocol. Online (Table 2), every method serves an interleaved stream of 2,000 requests per benchmark, each naming one of the five backbones, and learns from feedback after each execution.

Table 1: Held-out comparison by requested model, with every method served from its frozen state (Qwen3.5-4B has no cheaper executor). **Q** (↑): quality (%), **Cost** (↓): serving cost (¥), **Save** (↑): reduction from Raw Request (%). Gray marks quality below Raw Request and negative savings. Only `RouteSage` saves cost without losing quality in every setting.

Method	Qwen3.5-35B-A3B			Qwen3.5-122B-A10B			GLM-5.2			GPT-5.6-sol		
	Q (↑)	Cost (↓)	Save (↑)	Q (↑)	Cost (↓)	Save (↑)	Q (↑)	Cost (↓)	Save (↑)	Q (↑)	Cost (↓)	Save (↑)
MiniWoB++ (Success, %)												
Raw Request	30.5	3.49	0.0	40.6	7.16	0.0	47.3	30.49	0.0	50.3	69.47	0.0
FrugalGPT	30.5	4.87	−39.5	40.6	12.03	−68.0	46.9	40.78	−33.7	47.8	93.34	−34.4
RouteLLM	15.8	1.42	59.3	15.8	1.42	80.2	38.3	17.36	43.1	50.3	65.83	5.2
PAK-UCB	27.5	3.26	6.6	35.9	5.19	27.5	43.5	25.95	14.9	49.4	51.75	25.5
MESS+	29.0	2.16	38.1	33.9	2.71	62.2	35.4	5.07	83.4	36.2	11.06	84.1
`RouteSage`	**35.1**	**2.05**	**41.3**	**44.4**	**3.73**	**47.9**	**49.0**	**13.54**	**55.6**	**51.5**	**29.59**	**57.4**
T1 (Tool-calling F1, %)												
Raw Request	46.8	5.40	0.0	53.7	10.75	0.0	75.7	45.90	0.0	85.7	104.60	0.0
FrugalGPT	46.8	7.41	−37.2	53.7	18.16	−68.9	70.9	52.75	−14.9	76.9	94.20	9.9
RouteLLM	35.5	2.01	62.8	37.1	2.79	74.0	75.6	45.54	0.8	85.7	104.60	0.0
PAK-UCB	44.6	4.63	14.3	53.5	10.50	2.3	73.1	39.53	13.9	84.5	92.75	11.3
MESS+	37.9	2.06	61.9	37.9	2.06	80.8	75.7	45.90	0.0	75.7	45.90	56.1
`RouteSage`	**48.0**	**3.58**	**33.7**	**61.9**	**7.37**	**31.4**	**76.5**	**30.24**	**34.1**	**86.1**	**60.25**	**42.4**
CodeFlowBench (Pass@1, %)												
Raw Request	25.1	6.38	0.0	39.5	11.56	0.0	45.0	11.61	0.0	57.7	27.06	0.0
FrugalGPT	25.1	8.21	−28.7	39.5	19.76	−70.9	45.0	31.37	−170.2	57.7	58.27	−115.3
RouteLLM	22.1	1.83	71.3	22.1	1.83	84.2	22.1	1.83	84.2	57.6	26.65	1.5
PAK-UCB	24.0	4.67	26.8	39.4	11.00	4.8	42.9	11.44	1.5	54.7	23.88	11.8
MESS+	22.8	2.97	53.4	39.5	11.56	0.0	39.5	11.56	0.4	39.5	11.59	57.2
`RouteSage`	**26.2**	**3.77**	**40.9**	**43.6**	**8.34**	**27.9**	**47.2**	**9.53**	**17.9**	**59.3**	**21.02**	**22.3**

`RouteSage` learns from the quality score of each benchmark, with tool-calling F1 as a graded outcome on T1. Online cost covers routing, execution, recovery, construction, and validation. Held-out (Table 1), each method is frozen and serves 1,000 fresh tasks per benchmark and tier without fallback. Costs are totals in ¥, and savings are $100(1 - C_{\text{method}}/C_{\text{Raw}})$.

Baselines. We compare `RouteSage` with Raw Request, which always executes the requested model, and with representative routers from three categories: (1) *cascade routing*: FrugalGPT (Chen et al., 2024a), (2) *learned routing*: RouteLLM (Ong et al., 2025), and (3) *online routing*: PAK-UCB (Hu et al., 2025) and MESS+ (Woisetschläger et al., 2025). All baselines follow their original papers, and every method uses the same task cohorts, model prices, and cost accounting and never invokes a model more expensive than the requested one (Eq. (1), Appendix D.3). RouteLLM trains its router in batch, so we train it on the complete online split and report it in the held-out evaluation only.

4.2 Performance Comparison

Table 2 reports the online stream, where construction, validation, and recovery are billed together with serving. `RouteSage` lowers total cost by ↓29.6%, ↓20.3%, and ↓11.9% on MiniWoB++, T1, and CodeFlowBench while raising quality by ↑3.4, ↑3.2, and ↑1.9 points, so ***Obs. ❶ the savings of `RouteSage` hold with every patching cost billed.*** Among the baselines, FrugalGPT matches requested quality but costs more than Raw Request in every domain, about twice as much on CodeFlowBench, whereas PAK-UCB and MESS+ save by giving quality away, losing ↓3.0 and ↓21.2 points on T1. We observe that ***Obs. ❷ each fixed-pool router only chooses a side of the cost–quality trade***, since no baseline is both cheaper and as good in any domain.

Table 2: Online comparison on the mixed stream. **Q** (↑): delivered quality (%), including recovery, **Cost** (↓): total cost (¥), including patch construction and validation, **Save** (↑): reduction from Raw Request (%). RouteLLM trains in batch and has no online result (–). Gray follows Table 1. Only `RouteSage` is both cheaper and better than Raw Request in every domain.

Method	MiniWoB++			T1			CodeFlowBench		
	Q (↑)	Cost (↓)	Save (↑)	Q (↑)	Cost (↓)	Save (↑)	Q (↑)	Cost (↓)	Save (↑)
Raw Request	48.2	42.34	0.0	59.5	67.45	0.0	38.6	21.94	0.0
FrugalGPT	48.2	59.20	–39.8	59.5	74.14	–9.9	38.6	44.31	–102.0
RouteLLM	–	–	–	–	–	–	–	–	–
PAK-UCB	44.6	32.97	22.1	56.5	56.93	15.6	36.8	19.90	9.3
MESS+	40.0	17.48	58.7	38.3	8.71	87.1	34.0	18.39	16.2
`RouteSage`	**51.6**	**29.81**	**29.6**	**62.7**	**53.76**	**20.3**	**40.5**	**19.33**	**11.9**

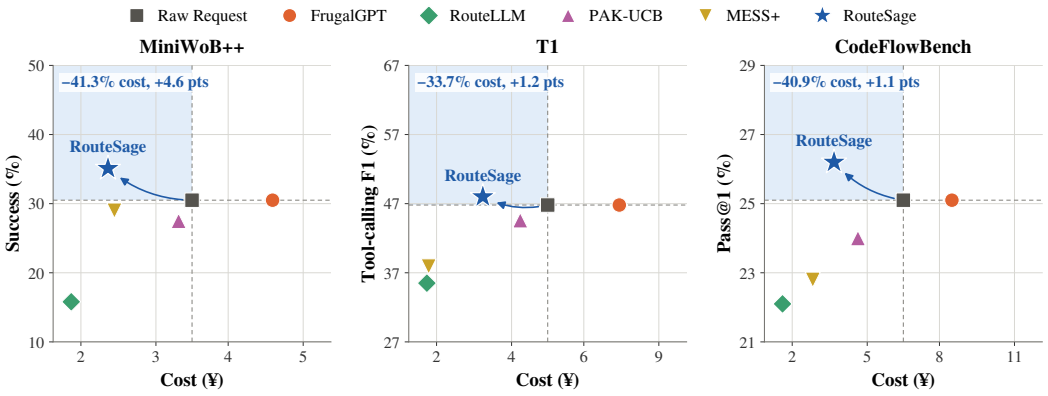


Figure 3: **Held-out quality versus serving cost.** Qwen3.5-35B-A3B is the requested model, whose only cheaper executor is Qwen3.5-4B. Dashed lines mark Raw Request, and the shaded region is cheaper and at least as good. FrugalGPT matches requested quality at a higher cost, while RouteLLM, PAK-UCB, and MESS+ are cheaper but fall below it. Only `RouteSage` enters it.

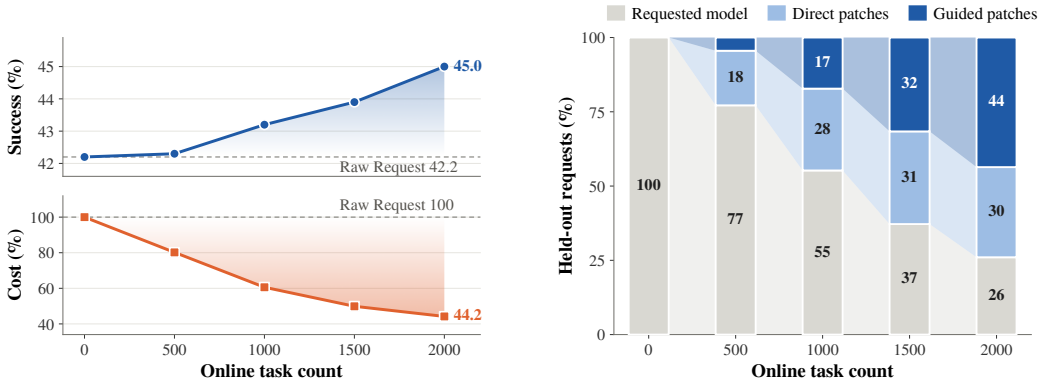
Table 1 serves fresh tasks from the frozen state, and Figure 3 plots its Qwen3.5-35B-A3B tier. `RouteSage` lowers serving cost in all twelve domain–tier settings and delivers the highest quality of any method in each. The baselines split along the same trade: FrugalGPT costs more than Raw Request in eleven of twelve settings, and the other three save mainly by losing quality, RouteLLM by ↓24.8 points at the Qwen3.5-122B-A10B tier of MiniWoB++. Of all 48 baseline results, only one saves without losing quality, RouteLLM's 5.2% at the GPT-5.6-sol tier of MiniWoB++, where `RouteSage` cuts cost by ↓57.4%. At the same tier of T1, it cuts cost by ↓42.4% while raising F1 from 85.7 to 86.1, and at the Qwen3.5-122B-A10B tier it raises F1 by ↑8.2 points. Thus ***Obs. ❸ capability patches find cheaper substitutes where fixed-pool routers find none.***

4.3 Ablation Studies

Table 3 adds the components of `RouteSage` one at a time. Routing keeps the Regions, audits, and selector but only raw actions, so it admits direct patches alone. It saves 46.5%, 26.6%, and 14.6% on MiniWoB++, T1, and CodeFlowBench, but on T1 and CodeFlowBench quality slips ↓2.4 and ↓0.9 points below Raw Request. Without new actions, saving means leaning on cheaper models that almost suffice, so ***Obs. ❹ within the original pool, the trade returns there.*** Evolution adds guided patches, raising savings by 9.3, 12.5, and 10.0 points and quality by ↑2.1, ↑5.0, and ↑3.2 points, showing that ***Obs. ❺ guided patches raise savings and quality together.*** These gains come from lifting cheaper models: a guided patch is admitted only if it beats its executor alone (Eq. (8)).

Table 3: Progressive ablation, with Q and Save averaged over the four held-out tiers of Table 1 and arrows giving the change from the row above. Routing among existing models saves cost but falls below Raw Request on two benchmarks, while evolution lifts quality and savings together.

Configuration	MiniWoB++		T1		CodeFlowBench	
	Q (↑)	Save (↑)	Q (↑)	Save (↑)	Q (↑)	Save (↑)
Raw Request	42.2	0.0	65.5	0.0	41.8	0.0
+ Routing (direct patches)	42.9 ↑0.7	46.5 ↑46.5	63.1 ↓2.4	26.6 ↑26.6	40.9 ↓0.9	14.6 ↑14.6
+ Evolution (`RouteSage`)	**45.0** ↑2.1	**55.8** ↑9.3	**68.1** ↑5.0	**39.1** ↑12.5	**44.1** ↑3.2	**24.6** ↑10.0



(a) Online-to-held-out trajectory

(b) Held-out action composition

Figure 4: **Longitudinal analysis on MiniWoB++.** Snapshots of the online state every 500 tasks serve the same 4,000 held-out requests, 1,000 per tier. (a) Quality and relative cost (Raw Request = 100). (b) Share of requests per action. As guided patches take over, quality rises and cost falls.

4.4 Mechanism Analysis

Does Online Evidence Pay Off after Freezing? In Figure 4(a), held-out cost falls from Raw Request's level to 44.2% of it after 2,000 online tasks, while quality rises from 42.2% to 45.0% and never drops below the requested model. The two move on different clocks: cost falls by 39.4 points in the first 1,000 tasks, while 1.8 of the 2.8 points of quality arrive in the second 1,000. We observe that ***Obs. ❻ more online evidence yields a better frozen service***, so what the service learns online keeps paying after it is frozen.

Where Do the Later Savings Come From? Figure 4(b) splits held-out requests by serving action. Direct patches come first, since they only need validation, and reach 28% of held-out requests by 1,000 tasks, then level off near 30%. Guided patches start slowly, needing recurring failures to learn from, but rise from 17% to 44% after 1,000 tasks while cost falls from 60.6% to 44.2%. Hence ***Obs. ❼ direct patches saturate early, while guided patches keep lifting cheaper models.***

5 Conclusion

We show that the savings of model routing are bounded by what its cheaper models can do, and that an agent service can lift this boundary while it runs. `RouteSage` distills the procedure a cheaper model is missing into an Experience, admits the guided action once it holds up against the requested model on fresh tasks, and routes traffic through it without retraining any model. Across three agent domains, this lowers total online cost above requested quality with every patching cost counted, by 29.6% on MiniWoB++, and guided patches add savings that direct patches alone do not reach. Because patches live in a model's context rather than its weights, the same loop applies to any pool of models served through an API. Overall, our results suggest that a router's action space is something to grow, not something given.

AI USE STATEMENT

Generative AI tools were used solely to assist with language editing and polishing of the manuscript. All scientific content, analyses, and conclusions were developed and verified by the authors.

REPRODUCIBILITY STATEMENT

We detail the method and experimental setup, including the full service loop, all decision rules, models, prices, cost accounting, and baseline settings in the appendix. All benchmarks are public.

REFERENCES

Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, et al. AutoMix: Automatically mixing language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. In *Findings of the Association for Computational Linguistics: ACL*, 2026.

Amartya Chakraborty, Paresh Dashore, Nadia Bathaee, Anmol Jain, Anirban Das, Shi-Xiong Zhang, Sambit Sahu, Milind Naphade, and Genta Indra Winata. T1: A tool-oriented conversational dataset for multi-turn agentic planning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

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 (TMLR)*, 2024a.

Shuhao Chen, Weisen Jiang, Baijiong Lin, James T. Kwok, and Yu Zhang. RouterDC: Query-based router by dual contrastive learning for assembling large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024b.

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

Tao Feng, Yanzhen Shen, and Jiaxuan You. GraphRouter: A graph-based router for LLM selections. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

GLM-5 Team. GLM-5: From vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. DS-Agent: Automated data science by empowering large language models with case-based reasoning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! Outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL*, 2023.

Xiaoyan Hu, Ho fung Leung, and Farzan Farnia. An online learning approach to prompt-based selection of generative models and LLMs. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2025.

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 *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Janet L. Kolodner. An introduction to case-based reasoning. *Artificial Intelligence Review*, 6(1): 3–34, 1992.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

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 from preference data. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

OpenAI. GPT-5.6 system card. https://deploymentsafety.openai.com/gpt-5-6, July 2026.

Zhihong Pan, Kai Zhang, Yuze Zhao, and Yupeng Han. Adaptive model and strategy routing for cost-efficient LLM services. In *Proceedings of the ACM Web Conference (WWW)*, 2026.

Pranoy Panda, Raghav Magazine, Chaitanya Devaguptapu, Sho Takemori, and Vishal Sharma. Adaptive LLM routing under budget constraints. In *Findings of the Association for Computational Linguistics: EMNLP*, 2025.

Manhin Poon, XiangXiang Dai, Xutong Liu, Fang Kong, John C. S. Lui, and Jinhang Zuo. Online multi-LLM selection via contextual bandits under unstructured context evolution. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2026.

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

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2017.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, 2026.

Philip S. Thomas, Georgios Theocharous, and Mohammad Ghavamzadeh. High confidence policy improvement. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research (TMLR)*, 2024.

Jiayu Wang, Yifei Ming, Zixuan Ke, Shafiq Joty, Aws Albarghouthi, and Frederic Sala. SkillOrchestra: Learning to route agents via skill transfer. *arXiv preprint arXiv:2602.19672*, 2026a.

Sizhe Wang, Zhengren Wang, Dongsheng Ma, Yongan Yu, Rui Ling, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. CodeFlowBench: A multi-turn, iterative benchmark for complex code generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2026b.

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 the Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)*, 2025a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2025b.

Herbert Woisetschläger, Ryan Zhang, Shiqiang Wang, and Hans-Arno Jacobsen. MESS+: Dynamically learned inference-time LLM routing in model zoos with service level guarantees. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Herbert Woisetschläger, Arastun Mammadli, Ryan Zhang, and Shiqiang Wang. Cost-optimal LLM routing with limited user feedback under user satisfaction guarantees. *arXiv preprint arXiv:2606.19376*, 2026.

Yifan Wu, Roshan Shariff, Tor Lattimore, and Csaba Szepesvári. Conservative bandits. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. Large language model cascades with mixture of thought representations for cost-efficient reasoning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Guibin Zhang, Haiyang Yu, Kaiming Yang, Bingli Wu, Fei Huang, Yongbin Li, and Shuicheng Yan. EvoRoute: Experience-driven self-routing LLM agent systems. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2026.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. Qwen3 Embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2024.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

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.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. WebArena: A realistic web environment for building autonomous agents. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

APPENDIX CONTENTS

A The Full Service Loop **14**

B Method Details **14**

B.1 Region Grouping and Activation 14

B.2 Selector Updates 15

B.3 Patch Validation, Versioning, and Budget 15

C Properties of the Selector and the Admission Rule **16**

C.1 The Selector Solves Its Allocation Problem Exactly 16

C.2 Admission Decisions Are Consistent and Tighten toward Parity 17

C.3 The Routing Table Stays Upper Triangular 18

D Additional Experimental Details **18**

D.1 Benchmark Details 18

D.2 Models and Cost 18

D.3 Baseline Implementations 18

E Additional Related Work **19**

E.1 LLM Routing and Cascades 19

E.2 Online Decision-Making and Safe Improvement 19

E.3 Experience, Memory, and Skills for LLM Agents 20

E.4 Case-Based Reasoning and Strong-to-Weak Transfer 20

E.5 Benchmarks for Agent Services 20

F Case Studies **20**

F.1 Direct Patch 20

F.2 Guided Patch 21

A THE FULL SERVICE LOOP

Algorithm 1 puts the three mechanisms of Section 3 into one online loop. Each request is first matched to a Region (lines 3–4, Section 3.2), and unmatched requests run on the requested model and feed provisional memory and its activation test (line 6). A matched request is served by an action sampled from its row of the routing table (line 8, Section 3.3). When the sample is the requested model and the row has candidates, the request doubles as an audit task, on which a candidate patch is compared with the requested model and then admitted, rejected, or kept under audit (lines 10–12). Otherwise the sampled action runs with fallback on a detected failure, and admitted patches stay under monitoring (line 15). Every execution updates Region memory and the selector statistics (line 16). Whenever a Region's trace count doubles, `RouteSage` rewrites its profile and builds new candidate patches where the projected benefit covers their cost (lines 17–20, Section 3.4). Frozen held-out service keeps only lines 3–6 and 8: it matches Regions, serves unmatched requests with the requested model, and executes the cheapest retained action of the matched row whose estimated quality is at least the requested model's, with no audits, fallback, updates, or patch construction.

Algorithm 1 Online service loop of `RouteSage`

Require: executor pool $\mathcal{M}$ with base prices c^0, retrieval width K
Ensure: a response for every request
1: Start with raw actions only ($\mathcal{A}_0^{\text{exp}} = \varnothing$) and an empty Region memory
2: **for** $t = 1, 2, \ldots$ **do**
Region assignment
3: Receive (x_t, r_t) and compute $e_t = \text{Embed}(x_t)$
4: Retrieve $\mathcal{K}_t$ by Eq. (3) and let the SLM select a compatible active Region k
5: **if** no compatible active Region exists **then**
6: Execute a^{r_t} and update provisional memory (Appendix B.1)
7: **else**
Serving and auditing
8: Set $\lambda^\star$ by Eq. (12) and sample a_t from $\mathcal{Q}_{k,r_t}$ by Eq. (6)
9: **if** $a_t = a^{r_t}$ and $\mathcal{U}_{k,r_t} \neq \varnothing$ **then**
10: Pick $a \in \mathcal{U}_{k,r_t}$ by Eq. (6) and run a, a^{r_t}, and, if guided, a^{raw} from one state
11: Return the result of a^{r_t}
12: Admit a if Eq. (8) holds, reject it by Eq. (15), or keep auditing
13: **else**
14: Execute a_t and fall back to a^{r_t} on a detected terminal failure
15: Suspend the patch of a_t if Eq. (17) holds
Memory and statistics update
16: Update U_k by Eq. (4), $\mathcal{H}_k$, and the action statistics
Capability expansion
17: **if** the trace count of Region k has doubled **then**
18: Rewrite P_k and propose direct or guided patches (Section 3.4)
19: **for** each candidate a with $\widehat{V}_k(a) > 0$ by Eq. (18) **do**
20: Build $E^{(v)}$ by Eq. (7) if guided, and add a to $\mathcal{U}_{k,r}$ of each eligible row r

B METHOD DETAILS

This section states the rules behind Sections 3.2–3.4.

B.1 REGION GROUPING AND ACTIVATION

A request with no compatible active Region is proposed to its nearest provisional group, and the SLM decides whether the two share a procedure. If not, the request seeds a new group, whose first trace comes from the requested model. Prototypes follow Eq. (4), with one update per distinct task, and the profile P_k is rewritten whenever the trace count reaches a power of two, $|\mathcal{T}_k| \in \{2^j\}_{j \geq 0}$.

A provisional Region is tested prospectively: its profile is frozen, and each later request that retrieval assigns to it counts as accepted (A_k) or rejected (J_k) by that profile. The Region becomes active

when the lower posterior bound on its acceptance rate exceeds one half,

$$F^{-1}_{\mathrm{Beta}(A_k+\frac{1}{2},\, J_k+\frac{1}{2})}(\alpha) > \tfrac{1}{2}, \tag{9}$$

and its traffic stays on the requested model until then. A request that brings incompatible conditions to an active Region enters provisional memory instead of inheriting its actions. Frozen held-out service uses active Regions only and makes no memory updates.

B.2 Selector Updates

Fix a Region and a requested-model row. With n_a resolved direct outcomes and summed outcome s_a, action a is estimated by the Jeffreys posterior mean

$$\widehat{q}_a = \frac{s_a + \frac{1}{2}}{n_a + 1}. \tag{10}$$

Unresolved outcomes enter only the cost record, and fallback executions are left out of $\widehat{q}_r$, since they cover only tasks another action has already failed. Relaxing the constraint of Eq. (5) gives

$$\mathcal{L}(w, \lambda) = \sum_a w_a \frac{\widehat{c}_a}{\widehat{c}_r} + \lambda \left(\widehat{q}_r - \sum_a w_a \widehat{q}_a \right) - \tau_t H(w). \tag{11}$$

For fixed λ, its minimizer over the simplex is the softmax of Eq. (6), and the weighted quality $\bar{q}(\lambda) = \sum_a w_a(\lambda)\, \widehat{q}_a$ is nondecreasing in λ. The selector uses

$$\lambda^\star = \min\{\lambda \geq 0 :\ \bar{q}(\lambda) \geq \widehat{q}_r\}, \tag{12}$$

found by bisection whenever the estimates change, since $\bar{q}$ is continuous and nondecreasing, $d\bar{q}/d\lambda = \mathrm{Var}_{w(\lambda)}(\widehat{q})/\tau_t \geq 0$. When no finite λ meets the target, as when a^r is the unique best-estimated action, the selector takes $\lambda^\star = \infty$, the limit of Eq. (6), which spreads traffic only over the highest-estimated actions and favors the cheaper among them. Aggregating weights by executor gives the routing table of Region k,

$$W^{(k)}_{ij} = \sum_{a:\mathrm{exec}(a)=m_j} w_{k,m_i,a}, \qquad W^{(k)}_{ij} = 0 \quad \text{if } j < i, \tag{13}$$

which starts as the identity and stays upper triangular (Lemma C.3). Proposition C.1 shows that $\mathbf{w}(\lambda^\star)$ solves Eq. (5) exactly.

When a cheaper action ends in a terminal failure, the requested model reruns the task from a restored state. On side-effecting tasks, cheaper execution requires rollback or idempotent replay. The first attempt updates the selected action's statistics, and a successful recovery is never credited to it. Frozen service executes the cheapest retained action with $\widehat{q}_a \geq \widehat{q}_r$, without recovery or updates.

B.3 Patch Validation, Versioning, and Budget

Audits. A request becomes an audit task when the selector sends it to the requested model while its row has candidates. The choice precedes execution, so audits are independent of outcomes and need no sampling rate. On an audit task, the candidate, the requested model, and, for a guided patch, the unguided executor start from the same state in isolated copies of the environment. The user receives the requested model's result, and only the additional calls are charged to evolution.

Paired Posterior and Decisions. For actions a and b run on the same audits, let n_{ij} count the tasks on which a has outcome i and b has outcome j. Under the Jeffreys prior,

$$(p_{11}, p_{10}, p_{01}, p_{00}) \sim \mathrm{Dir}\big(n_{11} + \tfrac{1}{2},\, n_{10} + \tfrac{1}{2},\, n_{01} + \tfrac{1}{2},\, n_{00} + \tfrac{1}{2}\big), \qquad q_a - q_b = p_{10} - p_{01}, \tag{14}$$

and $\ell_\alpha(a, b)$ and $u_\alpha(a, b)$ are the α and $1 - \alpha$ posterior quantiles of $q_a - q_b$. For graded outcomes $y_a, y_b \in [0, 1]$ on the same audit, the pair adds $y_a y_b$ to n_{11}, $y_a(1 - y_b)$ to n_{10}, $(1 - y_a)y_b$ to n_{01}, and $(1 - y_a)(1 - y_b)$ to n_{00}. This treats each score as the success probability of an independent Bernoulli draw, so the counts are expected cell counts that preserve the observed difference in mean quality, and binary outcomes recover ordinary counting. We use the resulting Dirichlet as a working

posterior for graded scores. For admission, unresolved outcomes count as 0 for the candidate and as 1 for its comparator. After each audit, the candidate is

$$\begin{cases} \text{admitted} & \text{if Eq. (8) holds,} \\ \text{rejected} & \text{if } u_\alpha(a, a^r) < -h_r, \text{ or } E \neq \varnothing \text{ and } u_\alpha(a, a^{\text{raw}}) \leq 0, \\ \text{audited further} & \text{otherwise.} \end{cases} \tag{15}$$

The resolution of the requested model is half the gap between the same two bounds of its Jeffreys posterior in the row (Eq. (10)),

$$h_r = \tfrac{1}{2}\Big[F^{-1}_{\text{Beta}(s_r+\frac{1}{2},\, n_r-s_r+\frac{1}{2})}(1-\alpha) - F^{-1}_{\text{Beta}(s_r+\frac{1}{2},\, n_r-s_r+\frac{1}{2})}(\alpha)\Big], \tag{16}$$

and the admission and rejection regions of Eq. (15) never overlap (Proposition C.2). Since the bounds are checked after every audit, 95% is a nominal level, and monitoring removes any admitted patch that later falls short. A patch holds only for the tested Region and requested model.

Monitoring and Versions. After admission, the patch keeps updating from regular traffic and $\widehat{q}_r$ from the requested model's traffic and audits. With both qualities under their Jeffreys posteriors, the patch is suspended when

$$\Pr\big(q_a < q_r\big) > 1 - \alpha, \tag{17}$$

and its action returns to $\mathcal{U}$. Version v may use all records up to its cutoff t_v. Its Experience, profile, and retrieval exemplars are frozen before auditing, and audit feedback stays out of later refreshes until the decision is final. A patch serves only requests that also match the profile frozen at its audit. A revised version replaces a suspended one once admitted, or an active one when paired audits show no lower quality and no higher cost, with a strict gain in one of the two.

Scheduling Construction. For a candidate patch with action a in Region k, the projected net benefit is

$$\widehat{V}_k(a) = \sum_{r:\, c^0_{\text{exec}(a)} < c^0_r} \widehat{N}_{k,r}\left[\widehat{c}^{\,\text{current}}_{k,r} - \widehat{c}^{\,\text{projected}}_{k,r,a}\right]_+ - \widehat{B}(a), \tag{18}$$

where $\widehat{N}_{k,r}$, the demand for r seen so far in the Region, estimates the demand still to come, a heuristic in the spirit of the doubling argument for an unknown horizon. The current cost is that of the retained policy, the projected cost of a, including Experience context and fallback, comes from its executor's raw traces, and $\widehat{B}(a)$ is the expected cost of building and auditing a, priced from past construction and audit records. A candidate is built and audited only if $\widehat{V}_k(a) > 0$, and auditing stops once the updated value turns negative. Every incurred cost, including the audits of rejected candidates, stays in the total cost of Eq. (2).

C Properties of the Selector and the Admission Rule

This section proves the properties used in Section 3. Throughout, we fix a Region and a requested-model row, write $\mathcal{F}$ for its retained actions with $a^r \in \mathcal{F}$, and take $\widehat{q}_a \in (0,1)$, $\widehat{c}_a > 0$, and $\tau = \tau_t > 0$ as given. Let $J(\mathbf{w}) = \sum_a w_a \widehat{c}_a/\widehat{c}_r - \tau H(\mathbf{w})$ be the objective of Eq. (5), $q^\star = \max_{a\in\mathcal{F}} \widehat{q}_a$, and $\mathcal{F}^\star = \{a \in \mathcal{F} : \widehat{q}_a = q^\star\}$.

C.1 The Selector Solves Its Allocation Problem Exactly

Proposition C.1 (Exact allocation). *The empirical allocation problem of Eq. (5) is feasible and has a unique solution $\mathbf{w}^\star$. For every $\lambda \geq 0$, the Lagrangian of Eq. (11) has the unique minimizer $\mathbf{w}(\lambda)$ of Eq. (6), and $\bar{q}(\lambda) = \sum_a w_a(\lambda)\widehat{q}_a$ is continuous and nondecreasing with*

$$\frac{d\bar{q}}{d\lambda} = \frac{\text{Var}_{\mathbf{w}(\lambda)}(\widehat{q})}{\tau} \geq 0. \tag{19}$$

If $\lambda^\star$ of Eq. (12) is finite, then $\mathbf{w}(\lambda^\star) = \mathbf{w}^\star$. Otherwise $\widehat{q}_r = q^\star$, and $\mathbf{w}^\star$ is the limit of $\mathbf{w}(\lambda)$ as $\lambda \to \infty$, which is supported on $\mathcal{F}^\star$ with $w^\star_a \propto \exp\big(-\widehat{c}_a/(\widehat{c}_r\tau)\big)$.

Proof. The point mass on a^r is feasible, so the feasible set is a nonempty compact convex subset of the simplex. As $-H$ is strictly convex and continuous, J attains a unique minimum on it.

For fixed λ, the Lagrangian equals $\sum_a w_a g_a + \lambda \widehat{q}_r - \tau H(\mathbf{w})$ with $g_a = \widehat{c}_a/\widehat{c}_r - \lambda \widehat{q}_a = -u_a$. By the Gibbs variational principle, $\sum_a w_a g_a - \tau H(\mathbf{w})$ has the unique minimizer $w_a \propto \exp(-g_a/\tau) = \exp(u_a/\tau)$ over the simplex, which is Eq. (6). Differentiating the softmax gives $dw_a/d\lambda = w_a(\widehat{q}_a - \bar{q})/\tau$, hence $d\bar{q}/d\lambda = \sum_a w_a \widehat{q}_a(\widehat{q}_a - \bar{q})/\tau = \mathrm{Var}_{\mathbf{w}(\lambda)}(\widehat{q})/\tau$.

Suppose $\lambda^\star < \infty$. The set $\{\lambda \geq 0 : \bar{q}(\lambda) \geq \widehat{q}_r\}$ is nonempty, closed by continuity, and an up-set by monotonicity, so its minimum $\lambda^\star$ is attained. If $\lambda^\star > 0$, continuity and minimality give $\bar{q}(\lambda^\star) = \widehat{q}_r$. In both cases $\lambda^\star\big(\widehat{q}_r - \bar{q}(\lambda^\star)\big) = 0$, and for any feasible $\mathbf{w}$,

$$J(\mathbf{w}) \;\geq\; \mathcal{L}(\mathbf{w}, \lambda^\star) \;\geq\; \mathcal{L}\big(\mathbf{w}(\lambda^\star), \lambda^\star\big) \;=\; J\big(\mathbf{w}(\lambda^\star)\big), \tag{20}$$

where the first inequality uses $\lambda^\star \geq 0$ and feasibility of $\mathbf{w}$. Since $\mathbf{w}(\lambda^\star)$ is itself feasible, it is optimal, and uniqueness gives $\mathbf{w}(\lambda^\star) = \mathbf{w}^\star$.

Suppose instead $\bar{q}(\lambda) < \widehat{q}_r$ for all finite λ. Writing $u_a/\tau = \big(\lambda(\widehat{q}_a - q^\star) - \widehat{c}_a/\widehat{c}_r\big)/\tau + \lambda q^\star/\tau$ shows that, as $\lambda \to \infty$, the weights of actions outside $\mathcal{F}^\star$ vanish and $\mathbf{w}(\lambda)$ converges to the distribution $w_a^\infty \propto \exp\big(-\widehat{c}_a/(\widehat{c}_r\tau)\big)$ on $\mathcal{F}^\star$. Hence $\bar{q}(\lambda) \to q^\star$, so $q^\star \leq \widehat{q}_r$, and $\widehat{q}_r \leq q^\star$ because $a^r \in \mathcal{F}$. Thus $\widehat{q}_r = q^\star$, every feasible $\mathbf{w}$ is supported on $\mathcal{F}^\star$, and on $\Delta(\mathcal{F}^\star)$ the problem reduces to minimizing $\sum_a w_a \widehat{c}_a/\widehat{c}_r - \tau H(\mathbf{w})$, whose unique minimizer is $\mathbf{w}^\infty$ by the same variational principle. □

In words, the price of quality $\lambda^\star$ is the multiplier that makes the softmax weights the exact solution of the quality-constrained problem, and the weighted quality of every row meets its target $\widehat{q}_r$.

C.2 Admission Decisions Are Consistent and Tighten toward Parity

Proposition C.2 (Admission rule)**.** *Let $\alpha < \frac{1}{2}$, and let ℓ_α and u_α be the α and $1-\alpha$ quantiles of $q_a - q_b$ under the paired posterior of Eq. (14).*

(i) After every audit, at most one of admission and rejection in Eq. (15) holds.

(ii) If a patch is admitted, then under the posterior $\Pr(q_a - q_r \geq -h_r) \geq 1-\alpha$, and, for a guided patch, $\Pr(q_a > q_{\mathrm{raw}}) \geq 1-\alpha$.

(iii) The resolution satisfies $h_r \leq \frac{1}{2}\sqrt{(1-\alpha)/\big(\alpha(n_r+2)\big)}$, so $h_r = O(n_r^{-1/2})$ and admission tightens toward $\ell_\alpha(a, a^r) \geq 0$.

Proof. (i) All Dirichlet parameters in Eq. (14) are positive, so $q_a - q_b = p_{10} - p_{01}$ has a continuous distribution and $\ell_\alpha(a,b) \leq u_\alpha(a,b)$ for $\alpha < \frac{1}{2}$. Rejection through $u_\alpha(a, a^r) < -h_r$ implies $\ell_\alpha(a, a^r) < -h_r$, which violates the first condition of Eq. (8). Rejection through $u_\alpha(a, a^{\mathrm{raw}}) \leq 0$ implies $\ell_\alpha(a, a^{\mathrm{raw}}) \leq 0$, which violates the last one.

(ii) By continuity, $\Pr\big(q_a - q_r \geq \ell_\alpha(a, a^r)\big) = 1-\alpha$, and admission requires $\ell_\alpha(a, a^r) \geq -h_r$. The same argument with $\ell_\alpha(a, a^{\mathrm{raw}}) > 0$ gives the second claim.

(iii) Let $X \sim \mathrm{Beta}(s_r + \frac{1}{2}, n_r - s_r + \frac{1}{2})$ with mean μ and variance σ^2. For $\mathrm{Beta}(\beta_1, \beta_2)$ with $\beta_1 + \beta_2 = n_r + 1$, $\sigma^2 = \beta_1\beta_2/\big((\beta_1+\beta_2)^2(\beta_1+\beta_2+1)\big) \leq 1/\big(4(n_r+2)\big)$. Cantelli's inequality gives $\Pr(X - \mu \geq t) \leq \sigma^2/(\sigma^2 + t^2)$ and the same bound for $\Pr(X - \mu \leq -t)$. Setting $t = \sigma\sqrt{(1-\alpha)/\alpha}$ makes both bounds equal to α, so both quantiles in Eq. (16) lie within t of μ and $h_r \leq t \leq \frac{1}{2}\sqrt{(1-\alpha)/\big(\alpha(n_r+2)\big)}$. □

The rule therefore never contradicts itself, admits a patch only when the evidence rules out a shortfall larger than the resolution of the bar, and lets that resolution shrink to exact parity as the requested model is observed more often.

C.3 The Routing Table Stays Upper Triangular

Lemma C.3 (Upper-triangular routing table)**.** *For every Region k and time t, the routing table of Eq. (13) is row-stochastic with $W_{ij}^{(k)} = 0$ for $j < i$, and it equals the identity before any patch is admitted.*

Proof. Every action in row m_i lies in $\mathcal{D}_{m_i,t}$ (Eq. (1)), since the row starts at $\{a^{m_i}\}$ and a patch is admitted only for an executor with $c_m^0 < c_{m_i}^0$. For $j < i$, the ordering gives $c_{m_j}^0 \geq c_{m_i}^0$ and $m_j \neq m_i$, so no action of row m_i executes on m_j and $W_{ij}^{(k)} = 0$. Rows sum to one because the weights of Eq. (6) lie on the simplex. Before any admission, $\mathcal{Q}_{k,m_i} = \{a^{m_i}\}$, so $W_{ii}^{(k)} = 1$. □

D Additional Experimental Details

D.1 Benchmark Details

Table 4 summarizes the three benchmarks. Each task returns automatic feedback, and `RouteSage` learns from its quality score. The online stream interleaves requests naming each of the five backbones, and held-out tasks are drawn for every requested tier with a cheaper executor. Held-out tasks are disjoint from the online stream of the same benchmark.

Table 4: Benchmarks, quality metrics, and evaluation scale.

Benchmark	Domain	Quality	Online	Held-out
MiniWoB++	Web interaction	Success	2,000	$4 \times 1{,}000$
T1	Multi-turn tool use	Tool-calling F1	2,000	$4 \times 1{,}000$
CodeFlowBench	Iterative code generation	Pass@1	2,000	$4 \times 1{,}000$

D.2 Models and Cost

Embeddings come from Qwen3-Embedding-4B (Zhang et al., 2025). Qwen3.5-4B serves as the SLM and reviews $K = 3$ candidate Regions per request, and Qwen3.5-35B-A3B serves as the builder (Eq. (7)), with its calls billed to evolution cost. Costs are computed from the input and output tokens of every call at the prices of Table 5, and a task's cost sums all of its calls. Serving cost covers Region matching, execution, and fallback, and evolution cost covers profile consolidation, Experience construction, and audit executions, including those of rejected candidates.

Table 5: Model prices used in all cost accounting.

Model	Input (¥/1M tokens)	Output (¥/1M tokens)
Qwen3.5-4B	0.15	1.20
Qwen3.5-35B-A3B	0.40	3.20
Qwen3.5-122B-A10B	0.80	6.40
GLM-5.2	4.00	12.00
GPT-5.6-sol	8.75	70.00
Qwen3-Embedding-4B	0.14	–

D.3 Baseline Implementations

All baselines follow their original papers. For fairness, every method uses the same task cohorts, prices, feedback, and cost accounting, and chooses only among the requested model and cheaper ones (Eq. (1)). Methods that learn online carry their state into held-out evaluation, where it is frozen. RouteLLM routes between one strong and one weak model. We use the requested model and Qwen3.5-4B, and train its router on the full online split. Table 6 summarizes the implementations.

Table 6: Compared methods and how each chooses an action.

Method	Category	Routing signal
Raw Request	Reference	Always the requested model
FrugalGPT (Chen et al., 2024a)	Cascade routing	Learned scorer over a price-ordered cascade
RouteLLM (Ong et al., 2025)	Learned routing	Preference-trained strong–weak router
PAK-UCB (Hu et al., 2025)	Online routing	Prompt-aware contextual bandit
MESS+ (Woisetschläger et al., 2025)	Online routing	Cost minimization under a quality constraint
`RouteSage`	Self-evolving routing	Validated patches over a growing action set

E Additional Related Work

E.1 LLM Routing and Cascades

Learned Routing and Cascades. Routing between models of different cost began with cascades and query-level predictors. FrugalGPT queries models in increasing price order and stops once a learned scorer accepts the answer (Chen et al., 2024a), Hybrid LLM sends a query to a small or a large model according to its predicted difficulty (Ding et al., 2024), and RouteLLM learns such strong–weak routers from human preference data (Ong et al., 2025). Later work learns query and model embeddings with dual contrastive objectives (Chen et al., 2024b), casts selection as edge prediction on a graph of tasks, queries, and models (Feng et al., 2025), and combines quality and cost prediction for dynamic routing (Wang et al., 2025a). Cascades refine when to escalate, through self-verification of the small model's answer (Aggarwal et al., 2024) or answer consistency across reasoning representations (Yue et al., 2024). All of these choose among the models they are given, so the savings they reach at a fixed quality are bounded by what those models can already do. `RouteSage` is complementary: it adds actions the pool does not have, and any of these routers could serve over the enlarged table.

Online, Constrained, and Agentic Routing. Deployed routers must learn from feedback as requests arrive. PAK-UCB selects models with a prompt-aware contextual bandit (Hu et al., 2025), contextual bandits have also been adapted to evolving and unstructured contexts (Poon et al., 2026), and PILOT routes under budget constraints (Panda et al., 2025). MESS+ minimizes cost under a service-level quality guarantee (Woisetschläger et al., 2025), and related work seeks cost-optimal routing with limited feedback under user-satisfaction guarantees (Woisetschläger et al., 2026). Routing has also moved into agent systems. EvoRoute routes within agent systems using accumulated experience (Zhang et al., 2026), SkillOrchestra routes agents through transferred skills (Wang et al., 2026a), RTR chooses both a model and a strategy (Pan et al., 2026), and Agent-as-a-Router lets an agent perform the routing for coding tasks (Zhou et al., 2026). These methods learn a better policy over a fixed menu of models, strategies, or agents. `RouteSage` learns the menu itself, and its constraint is anchored to the model each request names rather than to a global target.

E.2 Online Decision-Making and Safe Improvement

Contextual bandits balance exploration and exploitation as evidence accumulates (Hu et al., 2025), and the selector of `RouteSage` shares this structure, with an entropy-regularized allocation whose temperature decays with evidence and a quality constraint priced by a multiplier (Proposition C.1). Conservative bandits keep cumulative reward close to that of a known baseline policy while exploring (Wu et al., 2016), and safe policy improvement deploys a new policy only when it is better than the behavior policy with high confidence (Thomas et al., 2015). The admission rule of `RouteSage` applies the same principle to routing: the requested model is the baseline, a patch replaces it only under high-confidence evidence from paired audits, and monitoring suspends any patch that later falls short. The difference is that the arms are not given. Candidate actions are built by the service from its own failures, so `RouteSage` validates them on audit tasks chosen before their outcomes are known rather than on the traces that motivated them.

E.3 Experience, Memory, and Skills for LLM Agents

Agents that Act. LLM agents interleave reasoning with actions (Yao et al., 2023) and learn to call tools and APIs (Schick et al., 2023). Their multi-step loops are what make agent services expensive to serve, since every step pays the price of the executing model.

Learning from Experience. A large body of work improves agents from their own executions without updating weights. Reflexion revises behavior with verbal feedback (Shinn et al., 2023), and ExpeL accumulates insights across tasks (Zhao et al., 2024). Procedural memory reuses how tasks were solved: Synapse retrieves trajectories as exemplars (Zheng et al., 2024), Voyager grows a skill library (Wang et al., 2024), and Agent Workflow Memory induces reusable workflows (Wang et al., 2025b). Recent methods adapt memory at test time (Suzgun et al., 2026), distill trajectory-level lessons into transferable skills (Ni et al., 2026), refine procedural memory dynamically (Cao et al., 2026), and optimize skills through validation feedback (Yang et al., 2026). These methods aim to make one agent succeed more often. `RouteSage` uses experience for a different purpose, to let a cheaper model stand in for a more expensive one, which requires binding each Experience to its executor and Region, validating it against the requested model, and pricing it against its savings.

E.4 Case-Based Reasoning and Strong-to-Weak Transfer

Case-Based Reasoning. Case-based reasoning solves new problems by retrieving, reusing, revising, and retaining solutions to similar past ones (Kolodner, 1992; Aamodt & Plaza, 1994). The cycle has been revisited for LLMs, where it drives agents for data science (Guo et al., 2024). Capability Regions play the role of case indexing in `RouteSage`, deciding where a retained procedure may be reused, and the audits play the role of revision, deciding whether it may be retained.

Transfer from Strong to Weak Models. Distillation transfers the behavior of a strong model into a smaller one through training, for instance with the rationales of the stronger model as supervision (Hsieh et al., 2023). These methods change the weights of the smaller model and require access to them. `RouteSage` transfers what the strong model does differently through context, one Region and one executor at a time, so it applies equally to models served only through an API.

E.5 Benchmarks for Agent Services

Agent benchmarks span web interaction, from MiniWoB (Shi et al., 2017; Liu et al., 2018) to realistic websites (Zhou et al., 2024), multi-turn tool use and planning (Chakraborty et al., 2025), and code, from repository issues to iterative development (Jimenez et al., 2024; Wang et al., 2026b). These benchmarks score the success of an agent. We use three of them as streams of recurring procedures with automatic feedback and measure what a service pays per unit of quality it delivers.

F Case Studies

We show two tasks from the online stream on which a fixed-pool router would pay for the requested model. Each figure places the requested model's execution, which is what Raw Request does, next to the action that `RouteSage` selects for the same task, with costs per task. Figure 5 covers a direct patch: the cheaper model needs no help, and the router only has to learn, from validated evidence, that it can be trusted with this family. Figure 6 covers a guided patch: the cheaper model fails this family on its own, and its Experience closes the gap.

F.1 Direct Patch

Figure 5 shows a MiniWoB++ login task requested with Qwen3.5-35B-A3B and served through a direct patch by Qwen3.5-4B. Both complete the task. The requested model answers its first step with an explanation and three actions, against the benchmark's one-action format, and takes 9 calls. Qwen3.5-4B returns one valid action per call and finishes in 3, at 10.9% of the requested model's cost. A router that has not validated the cheaper model cannot know this and sends every such request to the requested model. The patch records the evidence once, for this Region and this requested model, and every later request in the family inherits it.

Raw Request: Qwen3.5-35B-A3B

Task. Enter the username "nieves" and the password "tF0y" into the text fields and press login. The agent must return exactly one BrowserGym action per call.

Output of the first call.

```
To proceed with the task, I need to fill
the username field with the value "nieves"
and the password field with the value
"tF0y", then click the Login button.
The correct action is:
fill('16', 'nieves')
fill('19', 'tF0y')
click('20')
```

Outcome. Success
Cost. ¥0.00567 **Calls.** 9 **Tokens.** 12,914

Direct patch: Qwen3.5-4B

Task. Same as left.

Output of the first call.

```
fill('16', 'nieves')
```

Outcome. Success
Cost. ¥0.00062 **Calls.** 3 **Tokens.** 3,940

Figure 5: **Case study of a direct patch** (MiniWoB++, online stream). Left: the requested model, which is what a request costs without routing. Right: the cheaper model selected through the direct patch, which returns one valid action per call. Both succeed, and the cheaper model needs 3 calls instead of 9 at 10.9% of the cost. Only the first call's output is shown.

F.2 Guided Patch

Figure 6 shows a T1 hotel-planning turn requested with GPT-5.6-sol. The user adds a fitness center and room service to an earlier search in San Francisco whose results are already cached. Without guidance, Qwen3.5-35B-A3B filters a variable it never loads, skipping the retrieval call, and its tool-calling F1 drops to 0.5. The Experience of this Region, built from contrasting traces of such failures, states the missing step: retrieve cached results explicitly, then filter. With it, the cheaper model retrieves, filters, and saves the results as the requested model does and reaches the same F1 at 5.4% of its cost, while the Experience itself adds 6.2% to the cheaper model's own cost. The card names no hotel, city, or cache key, so the same text serves every task in the Region.

Raw Request: GPT-5.6-sol

Task. T1 hotel planning. The user wants a hotel in San Francisco with laundry facilities from November 22 to 29, 2025, 2 rooms at about $475 per night. The last turn adds "a fitness center and room service." Earlier results are cached under `hotels` and `filtered_hotels`.

Output.

```
prior_results = get_results_from_cache(
  key="filtered_hotels")
filtered_hotels = filter_hotels(
  prior_results=prior_results,
  gym_present=True, has_room_service=True)
save_to_cache(key="filtered_hotels",
  value=filtered_hotels)
print(filtered_hotels)
```

Outcome. Tool-calling F1 1.0
Cost. ¥0.1045 **Tokens.** 11,216

Guided patch: Qwen3.5-35B-A3B

Experience (excerpt). Verify all mandatory parameters (dates, city) exist before tool calls.
Retrieve cached results via explicit get function,
then apply filters.
If required data is missing, request it. Do not assume cached data exists without retrieval.

Without Experience (F1 0.5).

```
filter_hotels(prior_results=hotels,
  gym_present=True, has_room_service=True)
```

With Experience.

```
cached_hotels = get_results_from_cache(
  key="hotels")
filtered_hotels = filter_hotels(
  prior_results=cached_hotels,
  gym_present=True, has_room_service=True,
  has_laundry_service=True, num_rooms=2,
  budget=475, ...)
save_to_cache(key="filtered_hotels",
  value=filtered_hotels)
```

Outcome. Tool-calling F1 1.0
Cost. ¥0.0056 **Tokens.** 12,701

Figure 6: **Case study of a guided patch** (T1, online stream). Left: the requested model. Right: the cheaper model without and with its Experience. The highlighted rule supplies the retrieval step that the unguided model skips. Reasoning blocks and the date arguments are omitted. The guided model matches the requested model's F1 at 5.4% of its cost.