

Scaling Small Agents Through Strategy Auctions

Lisa Alazraki^{†,2}, William F. Shen^{†,3}, Yoram Bachrach¹, Akhil Mathur¹

¹Meta Superintelligence Labs, ²Imperial College London, ³University of Cambridge

[†]Work done at Meta

Small language models are increasingly viewed as a promising, cost-effective approach to agentic AI, with proponents claiming they are sufficiently capable for agentic workflows. However, while smaller agents can closely match larger ones on simple tasks, it remains unclear how their performance scales with task complexity, when large models become necessary, and how to better leverage small agents for long-horizon workloads. In this work, we empirically show that small agents’ performance fails to scale with task complexity on deep search and coding tasks, and we introduce *Strategy Auctions for Workload Efficiency (SALE)*, an agent framework inspired by freelancer marketplaces. In SALE, agents bid with short strategic plans, which are scored by a systematic cost–value mechanism and refined via a shared auction memory, enabling per-task routing and continual self-improvement without training a separate router or running all models to completion. Across deep search and coding tasks of varying complexity, SALE reduces reliance on the largest agent by 53%, lowers overall cost by 35%, and consistently improves upon the largest agent’s pass@1 with only a negligible overhead beyond executing the final trace. In contrast, established routers that rely on task descriptions either underperform the largest agent or fail to reduce cost—often both—underscoring their poor fit for agentic workflows. These results suggest that while small agents may be insufficient for complex workloads, they can be effectively “scaled up” through coordinated task allocation and test-time self-improvement. More broadly, they motivate a systems-level view of agentic AI in which performance gains come less from ever-larger individual models and more from market-inspired coordination mechanisms that organize heterogeneous agents into efficient, adaptive ecosystems.

Date: February 4, 2026

Correspondence: lisa.alazraki20@imperial.ac.uk, akhilm@meta.com

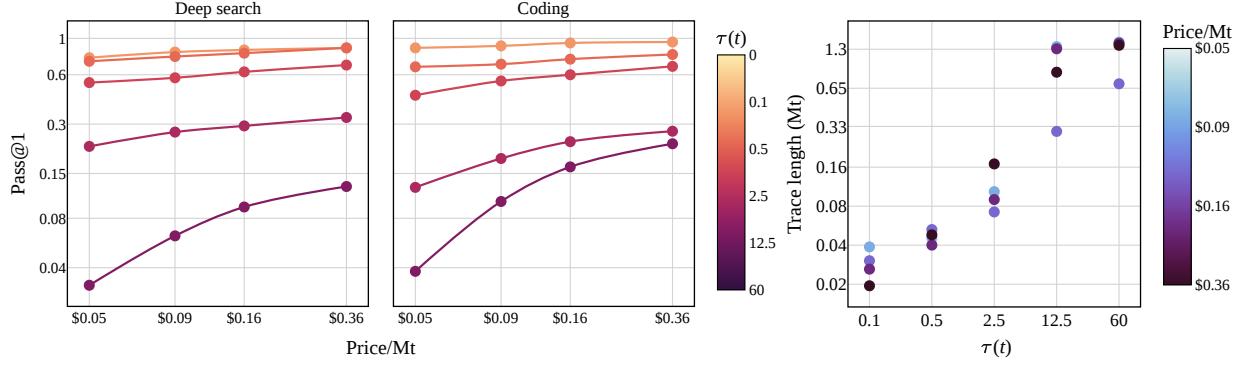


1 Introduction

Recent work on tool-augmented AI agents has led to growing optimism that small language models may be sufficient for many real-world applications. By offloading computation and knowledge to external tools and environments, small agents are argued to need less parametric capacity while still supporting complex, multi-step behavior (Houliston et al., 2025). Combined with advances that narrow the performance gap between small and large language models (Hooker, 2025), this has led to claims that small, inexpensive agents can replace large ones as the foundation of agentic AI (Belcak et al., 2025).

Yet much of the current optimism around small agents is framed in terms of model size and agentic capabilities, with comparatively little attention to how these interact with the structure and complexity of the tasks they are meant to solve. In practice, agentic workloads span a wide spectrum: from short, well-specified tasks with simple evaluation criteria to open-ended, long-horizon problems that require extended reasoning, integrating different types of information, and maintaining coherence over many steps (Wang et al., 2025d). It is not obvious that the same small agent that performs well on the former regime will also succeed on the latter, especially as demands on reasoning, planning, and context management grow with task complexity.

This perspective raises two central questions for the design of agentic AI systems. First, *how does task complexity mediate the relative effectiveness of small and large agents?* Second, given an increasingly heterogeneous landscape of models with different capabilities and costs, *how should we route tasks across agents to balance accuracy and cost—maximizing the workload handled by small, cheap agents without degrading performance on complex tasks?* Existing routing approaches provide only a partial answer. Non-predictive



(a) Pass@1 as a function of $\tau(t)$ and agent price.

(b) Trace length by $\tau(t)$ for agents of different prices.

Figure 1 Pass@1 accuracy on deep search and coding tasks (a) and average trace length in million tokens (b). We show the effective price per million tokens $\pi(a_d)$ for Qwen3 agents from smallest to largest ($d = \{4B, 8B, 14B, 32B\}$).

strategies that generate full outputs from all candidate models are tractable for single-shot QA but become infeasible for agents, whose trajectories can span tens of thousands to millions of tokens. Predictive routers, in turn, require training separate routing models that are costly to fit, do not generalize well to new models, and have been shown to degrade as task difficulty increases (Dhrif, 2025). It remains unclear how to design routing mechanisms for agentic systems that incur minimal additional inference cost, apply directly to off-the-shelf agents, remain effective on complex, long-horizon tasks, and ideally also help smaller agents shoulder more of the workload over time, effectively “scaling them up” without sacrificing accuracy.

To study how task complexity shapes the relative usefulness of small and large agents, we empirically evaluate deep search and coding tasks across multiple horizons. We choose these domains as they typify agentic workflows: deep search requires extended reasoning and information integration (Zhang et al., 2025), while coding demands multi-step planning and precise execution (Wang et al., 2025a). Both domains span short, well-specified tasks as well as open-ended, long-horizon problems, making them ideal for probing how agent capabilities scale with complexity. Following Kwa et al. (2025), we operationalize **task complexity** via human solution time: the average time expert annotators need to complete each task, from a few seconds to one hour. We apply this annotation protocol primarily to tasks from existing public benchmarks, with a small number of ad hoc tasks. Using off-the-shelf models from the Qwen3 series ranging from 4B to 32B parameters, and the Agent Research Environment (ARE) (Froger et al., 2025), we find that on the simplest tasks the smallest agent attains $\sim 87\%$ of the pass@1 performance of the largest agent, but on the most complex tasks this relative performance drops to only $\sim 21\%$ (ref. Figure 1 and Section 4). Thus, while small agents can closely match larger ones on simple tasks, their performance fails to scale with task complexity. This suggests that small agents alone are unlikely to be sufficient for many high-value applications and that model size should be treated as a *per-task* decision rather than a global choice about whether small agents can *replace* large ones.

In response, we develop a routing mechanism that is compute-efficient, applicable to off-the-shelf agents, and preserves performance on complex, long-horizon tasks. Inspired by human freelance marketplaces and virtual agent economies (Tomasev et al., 2025), we introduce **Strategy Auctions for Workload Efficiency (SALE)**, a test-time auction framework that leverages a well-established correlation between plan quality and execution quality (Sun et al., 2024). For each task, candidate agents propose strategic solution plans that are scored by predicted value and cost via peer assessment and heuristic predictors. The winning agent is selected based on this cost–value trade-off and its plan is executed, yielding an adaptive allocation of work across the agent pool. Crucially, this process is not static: plan refinement using the outcomes of past auctions can overturn the initial ranking before any strategy is executed—a self-improvement process analogous to how freelancers upskill over time to secure more work. In this way, SALE functions not only as a router but also as a mechanism that systematically increases the share of work handled by smaller, cheaper agents where possible, effectively “scaling up” small models via market-like coordination.

We find that SALE not only matches but even exceeds the largest agent’s pass@1 (+3.5% on deep search and

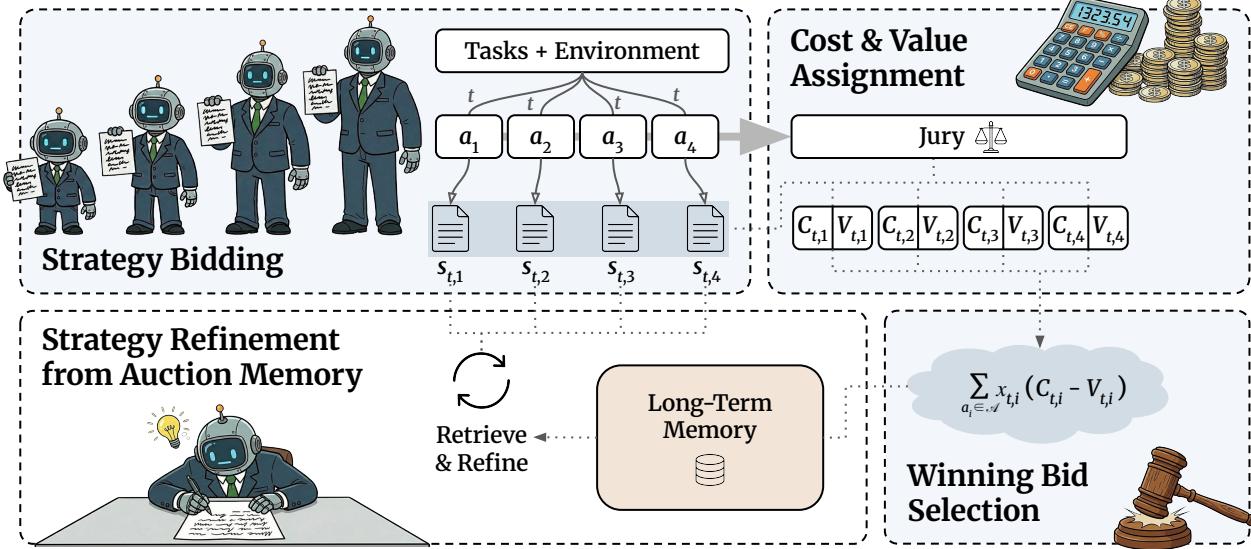


Figure 2 An illustration of the SALE pipeline. Given a task t , each agent a_i proposes a strategic plan $s_{t,i}$ as its bid. Bids are evaluated by cost $C_{t,i}$ and value $V_{t,i}$, and a provisional winner is selected by minimizing cost-minus-value. Agents cheaper than the provisional winner may then refine their strategies using similar past successes and failures retrieved from the auction memory, after which a final winner is selected and its strategy is executed.

+2.7% on coding) while offloading much of its workload (−65% and −40%, respectively) and reducing total spend (−42% on deep search and −25% on coding). These gains come with only a negligible increase in inference tokens. In contrast, established routers that rely on task descriptions either underperform the largest agent or fail to meaningfully reduce spend—often both. This underscores their poor fit for agentic workflows, where complex trajectories decouple task inputs from downstream success and strategic planning proves a more reliable routing signal. We also observe that, as the auction memory grows, the smallest agents are selected increasingly often, suggesting that they progressively capitalize on accumulated experience. Overall, SALE extends the performance–cost Pareto frontier beyond any single agent: it reduces reliance on large agents and total inference cost while improving accuracy across task complexities.

In summary, our contributions are:

1. We empirically study how task complexity affects the performance gap between small and large agents on deep search and coding tasks, finding that small agents nearly match large ones on simple tasks but diverge sharply as complexity increases. To the best of our knowledge, this is the first such investigation on realistic workloads; prior literature has examined agentic scaling behavior only on synthetic tasks.
2. We develop HST-BENCH, a benchmark that pairs agentic tasks with human solution times as a proxy for task complexity, enabling fine-grained evaluation of agent scaling behavior.
3. We introduce SALE, a marketplace-inspired framework in which heterogeneous AI agents bid with solution plans, are selected based on predicted value and cost, and use auction feedback to refine subsequent bids, yielding a unified mechanism that couples per-task model routing with test-time self-improvement.
4. We show that SALE achieves a better performance–cost Pareto frontier than any individual agent in the pool or existing routers on deep search and coding tasks. This demonstrates that strategy-based routing with continual agent self-improvement outperforms single-model and conventional routing baselines.
5. More broadly, by providing SALE as a marketplace-inspired framework, we illustrate how auction-based coordination can structure competition and collaboration among heterogeneous agents at test time, contributing to emerging discussions about how labor-like market dynamics and adaptive orchestration may shape future ecosystems of interacting AI agents.

2 Related Work

Agent Performance Under Task Complexity. Scaling AI agents to handle increasingly long and difficult tasks has become a central focus of recent work (Chan et al., 2025; Chen et al., 2025a; Froger et al., 2025; Wang et al., 2025d). Kwa et al. (2025) address this by tying capability to task duration, defining a 50%-success time horizon in terms of human solution time and studying how it scales on research and software-engineering tasks. Sinha et al. (2025) instead examine how performance degrades as tasks are extended, using synthetic, controlled multi-step tasks with explicit plans to argue that small per-step accuracy gains can yield much longer executable sequences and that many long-horizon failures reflect compounding execution errors as models condition on their own past outputs. We build on both perspectives by analyzing these scaling phenomena on real-world deep search and coding workloads and by shifting from isolated model behavior to system-level performance in a marketplace that allocates tasks across heterogeneous agents.

Multi-Agent Routing. Routing has emerged as a key strategy for harnessing the diversity of heterogeneous AI systems. There are two main approaches to routing: non-predictive routing, which selects outputs after running multiple models, and predictive routing, which chooses a model in advance based on input features or learned decision policies (Hu et al., 2024). Non-predictive methods (Chen et al., 2024) can be prohibitively expensive in agentic settings, where trajectories involve extended tool use and long interaction histories (Tsiourvas et al., 2025). Predictive approaches (Hu et al., 2024; Stripelis et al., 2024; Somerstep et al., 2025) mitigate this cost by learning separate routing models, but these are themselves costly to fit, tightly coupled to specific model sets, and have been shown to degrade as task difficulty increases (Dhrif, 2025). Moreover, existing routers are typically static: once trained, their routing policies do not incorporate test-time feedback, and thus do not improve with experience. In contrast, our framework, SALE, implements a lightweight, strategy-based, partially predictive routing mechanism in which agents bid with short plans rather than full solutions, leveraging empirical evidence that plan quality correlates with downstream task success (Sun et al., 2024; Kang et al., 2025; Xiong et al., 2025b). Auction feedback and shared memory refine future bids, progressively shifting more work onto smaller agents. Thus, SALE couples routing with continual adaptation, turning agent selection from a purely passive assignment into a mechanism that actively improves small agents’ effective capabilities under compute constraints.

Memory-Driven Adaptation. Memory systems help agents improve by reusing past behavior. Existing work typically uses memory to improve an agent’s reasoning, either by extracting reusable routines from successful trajectories to guide future actions (Cao et al., 2025; Wang et al., 2025e), or by maintaining structured records of past interactions that provide richer context and user-specific knowledge (Salama et al., 2025; Wang and Chen, 2025; Xu et al., 2025). In contrast, SALE differs both in what is stored and in how that information is used: rather than logging answers, execution traces, or user histories, we treat bidding strategies and their auction outcomes (wins and losses) as the primary memory signal. This makes memory an explicit mechanism for reallocating work and upgrading the effective capabilities of smaller agents, as feedback from past auctions is used to refine future bids and adjust the division of labor in the marketplace.

Agent Systems as Virtual Economies. Prior work has argued that as autonomous agents become economically significant, they should be coordinated through explicit market mechanisms, including auction-based interaction (Duetting et al., 2024; Zhu et al., 2024; Jiang et al., 2025; Yang et al., 2025b), virtual sandbox economies with controlled links to human markets (Tomasev et al., 2025), and settings in which assistant and service agents transact directly on behalf of users and firms (Rothschild et al., 2025). Building on this perspective, where AI agents increasingly resemble digital workers, we instantiate a labor-focused framework: SALE treats agents as freelancers in a job marketplace, where auctions over strategic plans allocate work and learning opportunities, illustrating how labor-like dynamics can shape future agent ecosystems.

3 Experimental Setup

To evaluate agentic performance across task complexities and model scales, we run all experiments within the Agent Research Environment (ARE) framework (Froger et al., 2025). ARE provides a standardized platform for benchmarking agent behavior, enabling consistent measurement and comparison across domains.

3.1 Data

We evaluate agentic performance on two domains: deep search and coding, as they broadly represent agentic workflows requiring extended reasoning and multi-step planning. For deep search, we sample from SimpleQA (Wei et al., 2024), PopQA (Mallen et al., 2023), HotpotQA (Yang et al., 2018), GAIA (Mialon et al., 2024), and an expert-validated portion of Humanity’s Last Exam (Phan et al., 2025; White, 2025). Coding tasks are drawn from MBPP (Austin et al., 2021) and LeetCode (Xia et al., 2025), supplemented with custom multiple-choice questions to cover lower-complexity cases. We select these datasets because they span a wide range of task horizons and require genuinely agentic capabilities—deep search demands dynamic tool use, iterative retrieval, and cross-source synthesis, while coding involves iterative debugging and test-driven refinement. These benchmarks have been widely adopted for evaluating agentic AI systems, ensuring both breadth and comparability with prior work (Coignion et al., 2024; Labruna et al., 2024; Liu et al., 2024b; Amini et al., 2025; Gan et al., 2025; Huang, 2025; Xie et al., 2025).

Kwa et al. (2025) validate human solution time as the primary metric for agentic task complexity, showing that it naturally integrates reasoning, planning and execution into a single scale. We adopt this measure and define the *task complexity* of $t \in \mathcal{D}$ as $\tau(t)$, the average time (in minutes) required by expert annotators to solve t . Human solution times are annotated by three expert annotators, yielding reliably reproducible estimates (Krippendorff’s $\alpha = 0.86$; details in Appendix A.2). To enable fine-grained analysis of complexity effects, we group tasks into five non-overlapping bins according to $\tau(t)$, corresponding to average human solution times of up to 6 seconds ($0 < \tau(t) \leq 0.1$), 30 seconds ($0.1 < \tau(t) \leq 0.5$), 2.5 minutes ($0.5 < \tau(t) \leq 2.5$), 12.5 minutes ($2.5 < \tau(t) \leq 12.5$), and 60 minutes ($12.5 < \tau(t) \leq 60$). Bin boundaries follow a geometric progression ($5\times$ between adjacent bins), yielding equal spacing on a log scale. This is appropriate given that human solution times span nearly three orders of magnitude, and produces approximately balanced sample sizes across bins. A breakdown of the data composition for each time bin is provided in Appendix A. In total, the resulting human-timed dataset HST-BENCH, contains 753 tasks.

3.2 Models

For all experiments, we utilize the Qwen3 family of language models (Yang et al., 2025a), chosen for their open-weight availability and broad range of sizes. Qwen3 provides checkpoints at 4B, 8B, 14B, and 32B parameters, which prior work has treated as a matched set for studying scaling behavior (Sinha et al., 2025). To support cost-aware evaluation, we define an effective price per million tokens $\pi(a_d)$ for each agent of size d , based on published API rates (see Appendix B) and an observed average input-to-output token ratio of 4:1. Under this convention, we obtain $\pi(a_{4B}) = \$0.05$, $\pi(a_{8B}) = \$0.09$, $\pi(a_{14B}) = \$0.16$, and $\pi(a_{32B}) = \$0.36$. Because model size and π are monotonically aligned, we use *smaller/cheaper* and *larger/more expensive* interchangeably when discussing these particular agents. However, in figures we plot ‘price per million tokens’ on the x -axis to emphasize the cost dimension explicitly. Note that we run all models with greedy decoding.

4 Agent Performance vs. Task Complexity

We systematically evaluate Qwen3 agents of different sizes and costs on deep search and coding tasks, conditioning performance on task complexity as measured by $\tau(t)$ (Section 3.1). This setup enables direct comparison of agent performance as task demands increase. We measure performance via pass@1, scored via LLM-as-a-judge evaluation against ground-truth answers (see Appendix C).

Across both domains, agents perform very similarly on the simplest tasks, as shown in Figure 1a. For deep search, the cheapest agent achieves about 87% of the most expensive agent’s pass@1 on tasks with $\tau(t) \leq 0.1$; for coding, this relative performance is about 92%. In this regime, the scaling curves are nearly flat: moving from cheaper/smaller to more expensive/larger agents yields only modest gains. As task complexity increases, the scaling curves gradually become steeper, and by the most complex tasks ($\tau(t) \leq 60$), the separation between agents is sharp. For deep search, the cheapest agent attains only 25% of the most expensive agent’s pass@1 on these tasks; for coding, it reaches just 17%. In this long-horizon regime, performance is strongly stratified by model size and cost.

One might hope that, although larger agents are more expensive per token, they implicitly “pay for themselves”

by solving tasks with shorter trajectories—for instance, by requiring fewer reasoning steps, tool calls, or revisions. In practice, as shown in Figure 1b, we observe this pattern only for low-complexity tasks. As $\tau(t)$ increases, total token usage grows across all models, and larger agents do not consistently achieve shorter traces than smaller ones. Indeed, on many long-horizon instances, they incur comparable or greater token counts. Thus, increased parametric capacity does not generally yield more token-efficient solutions on complex workloads, and higher per-token costs for larger agents are not naturally offset by reduced test-time compute.

In sum, cheaper agents are effective for tasks with low $\tau(t)$, but their limitations become starkly apparent as task demands intensify. More expensive agents appear indispensable for complex problems—yet deploying them universally squanders resources on tasks that do not require their power. The challenge, therefore, is to build systems that can dynamically allocate tasks to the right agent, achieving a better balance between resource efficiency and capability.

5 Strategy Auctions

Agentic pipelines commonly include a planning phase in which agents outline their intended approach before acting. These strategic plans encode task-relevant information such as decomposition strategies, tool selection, anticipated challenges, yet they are rarely leveraged beyond the agent that produced them. Our framework, SALE (Figure 2), exploits this observation by casting strategic plans as bids in an auction framework. Specifically, given an environment E , a task t , and a heterogeneous group of agents $\mathcal{A} = \{a_i\}_{i=1}^{|\mathcal{A}|}$, each agent a_i generates a strategy $s_{t,i}$ conditioned on t and E (we omit E from the notation for brevity). We interpret $s_{t,i}$ as the “bid” of agent a_i for task t , which is then used to compute both the cost and value of a_i with respect to t , enabling model selection based on strategic intent rather than task description alone. Please see Appendix C for the prompts used to obtain $s_{t,i}$.

Cost and Value Assignment. Let $C_{t,i}$ and $V_{t,i}$ denote the cost and value, respectively, of deploying agent a_i on task t . We estimate the *cost* as

$$C_{t,i} = w_c \cdot \pi(a_i) \cdot |s_{t,i}|,$$

where $\pi(a_i)$ is the price per million tokens for agent a_i , $|s_{t,i}|$ is the length of $s_{t,i}$ in tokens, and w_c is a tuned weight. We use strategy length as a cost signal motivated by two prior works. First, Goebel and Zips (2025) show that plan (or strategy) length is correlated with final trace length, hence serving as a proxy for total inference cost. Second, execution reliability degrades with plan length: prior work finds that success rates decline as plans grow longer (Xiong et al., 2025a). Because failed executions nonetheless consume compute, longer plans entail higher expected cost—both through greater token usage and increased risk of wasted computation. We also show thorough ablations in Appendix I to validate this design choice.

We estimate the *value* of agent a_i for task t as

$$V_{t,i} = w_h \cdot H(s_{t,i}) + \sum_{a_j \in \mathcal{A}} w_j \cdot \gamma_j(s_{t,i}),$$

where $H(s_{t,i})$ is the normalized entropy of $s_{t,i}$, each $\gamma_j(s_{t,i})$ is a judgment score assigned by agent a_j in \mathcal{A} to $s_{t,i}$, and the weights w_h and w_j are tunable weights. Value thus combines two signals: intrinsic quality, captured by entropy, and extrinsic quality, captured by self-and-peer assessment.

The choice of entropy as a proxy for strategy value is motivated by extensive prior literature linking higher-entropy intermediate reasoning to greater informational content and reduced redundancy (Chen et al., 2025b; Cheng et al., 2025; Li et al., 2025; Wang et al., 2025c), and by work suggesting that prioritizing higher-entropy trajectories can be beneficial for planning (Liu et al., 2024a) (validated by our ablations in Appendix I).

The second term aggregates peer assessments on the strategy by a jury of agents. Each strategy is scored by the full set of bidding agents \mathcal{A} , including the agent that proposed it. This mixed self-and-peer design is supported by literature on LLM juries (Badshah and Sajjad, 2025; Verga et al., 2024) and by evidence that combining self- and peer-evaluation yields more reliable judgments than peer-only approaches (Mousavi et al., 2023). Ablations (Appendix I) confirm that excluding self-evaluation or reducing jury size degrades performance.

The judge prompt is provided in Appendix C; further details of the cost–value estimation appear in Appendix D.

Winning Bid Selection. Given the cost and value assignments described above, SALE aims to select the agent whose strategy achieves the optimal trade-off between resource efficiency and expected performance. Our goal is thus to learn scoring weights that minimize the worst-case cost-minus-value over a training set of tasks. The cost-minus-value serves as a unified measure of an agent’s desirability for a given task: lower costs improve resource efficiency, while higher values reflect stronger expected performance. By minimizing $C - V$, we favor agents that deliver high value at low cost.

Formally, we pose a min–max optimization over both the assignment variables x and the scoring weights $w = (w_c, w_h, \{w_j\}_{a_j \in \mathcal{A}})$. Let Q denote the maximum cost-minus-value over all tasks. The objective is

$$\min_{w, x, Q} Q \quad \text{s.t.} \quad z_t \leq Q \quad \forall t, \quad \sum_{a_i \in \mathcal{A}} x_{t,i} = 1 \quad \forall t, \quad w \in \mathbb{R}^{2+|\mathcal{A}|},$$

where z_t is the cost-minus-value of the chosen strategy for task t , and additional big- M constraints are imposed (see Appendix D for details of the tuning process). The min–max formulation ensures robustness across the training distribution: by optimizing against the worst-case task, we guard against any single task receiving a disproportionately poor assignment.

At inference time, given the learned weights w , SALE then applies the resulting scoring rule to route new tasks. For each task t , we introduce binary assignment variables $x_{t,i} \in \{0, 1\}$ indicating whether a_i is selected for task t , and define

$$z_t = \sum_{a_i \in \mathcal{A}} x_{t,i} (C_{t,i} - V_{t,i}).$$

Since exactly one agent is assigned per task, this reduces to $z_t = C_{t,\hat{i}(t)} - V_{t,\hat{i}(t)}$, where $\hat{i}(t) = \arg \max_i x_{t,i}$, thus assigning each task to the agent with the lowest cost-minus-value.

Strategy Refinement from Auction Memory. After each auction, we store all proposed strategies—both winning and losing bids—in a long-term memory bank \mathcal{M} . This enables a self-improvement mechanism in which cost-efficient agents that are not selected in the initial auction round can learn from \mathcal{M} , refine their initial strategies, and submit improved bids. Importantly, this refinement is *opportunistic*: we do not use memory for all agents by default. Doing so would require every agent to produce both an initial and a memory-informed bid, increasing latency and token usage. Instead, we first collect baseline strategies without memory. If a cheap agent already wins the auction, no refinement is needed; otherwise, only agents cheaper than the provisional winner output a refined, memory-informed bid, preserving the cost-efficiency goals of SALE.

Concretely, for each past task t' , we store a record $\mathcal{M}(t') = (t', \{s_{t',i}\}_{a_i \in \mathcal{A}}, y_{t'})$ where $\{s_{t',i}\}_{a_i \in \mathcal{A}}$ are the strategies proposed by all agents for t' and $y_{t'}$ encodes the auction outcome, indicating which strategy won and which ones failed. Let $\mathcal{T}_\mathcal{M}$ denote the set of tasks for which we have stored memory records, i.e. $\mathcal{T}_\mathcal{M} = \{t' : \mathcal{M}(t') \in \mathcal{M}\}$. This memory accumulates a diverse set of strategic plans and outcomes, providing a rich resource for agents to learn from past experience. Given a new task t , the refinement procedure operates as follows (see Appendix E for a full algorithmic description):

1. *Initial bids.* Each agent $a_i \in \mathcal{A}$ submits an initial strategy $s_{t,i}$, and a provisional winner $\hat{i}(t) = \arg \min_i (C_{t,i} - V_{t,i})$ is selected.
2. *Shared memory retrieval.* For each agent a_i cheaper than the provisional winner (i.e., $\pi(a_i) < \pi(a_{\hat{i}(t)})$), we retrieve a subset $\mathcal{M}_{t,i}$ of relevant past strategy pairs:

$$\mathcal{M}_{t,i} = \left\{ (s_{t'}^{\text{lose}}, s_{t'}^{\text{win}})_i \mid t' \in \underset{t' \in \mathcal{T}_\mathcal{M}}{\text{top-}\tilde{k}} \text{ sim}(t, t') \right\}, \quad \tilde{k} = \min(k, |\mathcal{T}_\mathcal{M}|),$$

where sim denotes cosine similarity over text embeddings (see Appendix C.3 for details) and each pair $(s_{t'}^{\text{lose}}, s_{t'}^{\text{win}})_i$ contains a losing and winning strategy for t' , with at least one proposed by a_i .

3. *Contrastive prompting.* Retrieved pairs are formatted using a contrastive prompt template (Appendix C.4) that encourages agents to learn from past auction outcomes.
4. *Reassignment.* Each eligible agent produces a refined bid $s_{t,i}^r$, which is scored to obtain updated cost $C_{t,i}^r$ and value $V_{t,i}^r$. If any refined bid improves upon the provisional winner’s cost-value trade-off, the

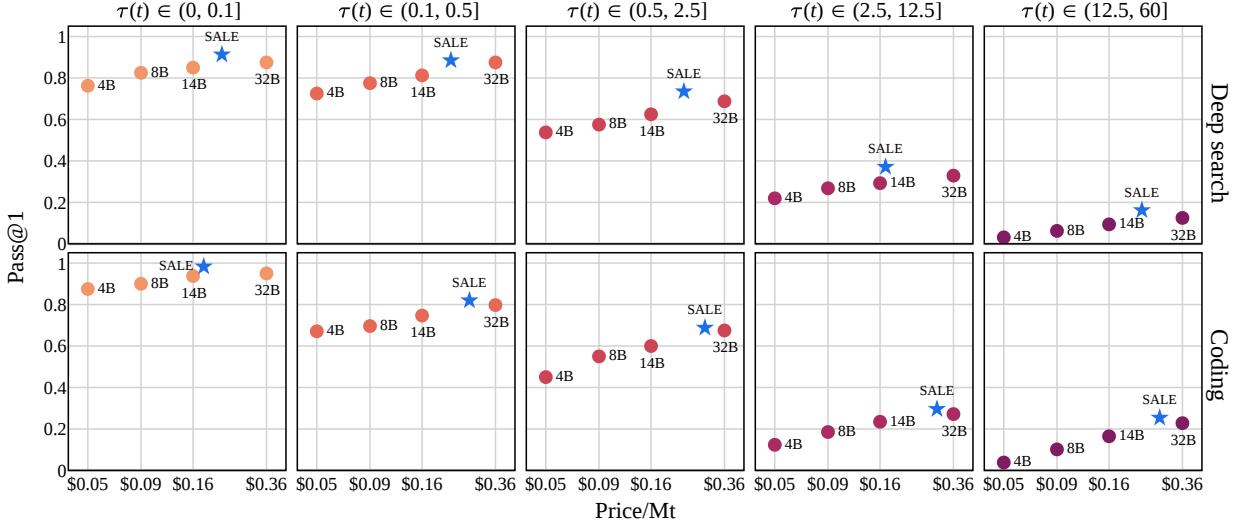


Figure 3 Performance–cost trade-offs for deep search (top row) and coding (bottom row) across task-complexity bins. At a given price per million tokens π , the SALE auction ensemble consistently attains substantially higher pass@1 than would be predicted by the approximate linear scaling trend observed for individual Qwen3 agents, showing that it systematically exceeds the expected performance–cost curve.

best such bid wins; otherwise, the provisional winner is retained:

$$i^*(t) = \begin{cases} \arg \min_{\substack{i: \pi(a_i) < \pi(a_{i(t)}) \\ \hat{i}(t)}} (C_{t,i}^r - V_{t,i}^r) & \text{if any refined bid satisfies } C_{t,i}^r - V_{t,i}^r < C_{t,\hat{i}(t)} - V_{t,\hat{i}(t)}, \\ \hat{i}(t) & \text{otherwise.} \end{cases}$$

5. *Execution.* After selecting $i^*(t)$, we execute agent $a_{i^*(t)}$ conditioned on t and its winning strategy.

It is worth noting that both jury scoring and strategy refinement incur only a small additional inference cost, on the order of a few hundred tokens¹, whereas executing the final agentic trace typically consumes tens of thousands to millions of tokens (see Figure 1b), depending on task complexity. *Thus, the overhead introduced by the auction mechanism is negligible relative to the overall test-time compute.*

6 Results

We run SALE on the full HST-BENCH test set, containing tasks from all complexity levels interleaved in random order, and only partition results into complexity bins for analysis. We use greedy decoding in all runs. For the single-model baselines we report results from a single run. For SALE, however, task order matters because its auction memory for strategy refinement evolves online. Following established practice for order-sensitive evaluation (Dash et al., 2022; Wang et al., 2025b), we thus report all SALE metrics as averages over five independent random permutations of the full test set.

Figure 3 summarizes the performance–cost trade-offs for deep search (top row) and coding (bottom row) across all five task-complexity bins, plotting pass@1 against price per million tokens for individual Qwen3 agents and for the SALE ensemble. Detailed numerical results for each bin, along with additional baselines described below, appear in Table 1. For deep search, SALE exceeds the best single agent’s pass@1 on the lowest-complexity tasks while operating at a lower effective price per million tokens (39% cost reduction, 3.8 pass@1 gain). On medium-complexity tasks, it improves pass@1 by between 1 and 4.7 percentage points over the best single agent, while reducing cost by 36–53%. On the most complex tasks, it still outperforms the best agent by 3.8 points while lowering cost by 36%. For coding, SALE likewise beats the best single agent on the simplest tasks (50% cost reduction, 3.3 pass@1 gain), and on medium-complexity tasks it achieves 1.5–3.2

¹On average, SALE requires generating 669 additional *total* tokens per task for deep search, and 1042 for coding. Here, “total tokens” denotes the sum of the initial strategy-generation, strategy-refinement and jury-vote tokens across all agents in the pool.

Task type	$\tau(t)$	Best single agent		WTP		CARROT		TO-Router		FrugalGPT		SALE w/o memory		SALE	
		Pass@1(\uparrow)	\$/Mt(\downarrow)	Pass@1(\uparrow)	\$/Mt(\downarrow)										
Deep search	≤ 0.1	87.5	0.36	83.8	0.32	85.0	0.27	86.3	0.28	86.3	0.47	91.3	0.24	91.3 _{0.0}	0.22 _{0.01}
	≤ 0.5	87.5	0.36	86.3	0.33	86.3	0.28	86.3	0.32	81.3	0.48	87.5	0.24	88.5 _{0.5}	0.22 _{0.01}
	≤ 2.5	68.8	0.36	67.5	0.31	66.3	0.29	67.5	0.34	66.3	0.53	72.5	0.25	73.5 _{1.2}	0.23 _{0.01}
	≤ 12.5	32.9	0.36	34.2	0.32	29.3	0.29	32.9	0.36	30.5	0.50	35.4	0.19	37.1 _{1.8}	0.17 _{0.01}
	≤ 60	12.5	0.36	9.4	0.31	9.4	0.32	12.5	0.36	12.5	0.60	15.6	0.26	16.3 _{1.3}	0.23 _{0.02}
	All	63.8	0.36	62.4	0.32	61.3	0.28	63.0	0.33	61.0	0.51	66.4	0.24	67.3 _{0.5}	0.21 _{0.00}
Coding	≤ 0.1	95.0	0.36	93.8	0.16	95.0	0.36	95.0	0.36	97.5	0.39	97.5	0.22	98.3 _{1.0}	0.18 _{0.00}
	≤ 0.5	79.7	0.36	76.0	0.15	82.3	0.25	79.7	0.36	69.6	0.61	82.3	0.28	82.0 _{0.5}	0.27 _{0.01}
	≤ 2.5	67.5	0.36	60.0	0.15	60.0	0.26	67.5	0.36	56.3	0.61	68.8	0.31	69.0 _{0.5}	0.29 _{0.00}
	≤ 12.5	27.2	0.36	14.8	0.05	27.2	0.36	27.2	0.36	18.5	0.61	27.2	0.32	30.4 _{2.2}	0.30 _{0.02}
	≤ 60	22.8	0.36	6.3	0.05	21.5	0.35	22.8	0.36	10.1	0.61	24.1	0.31	26.1 _{1.4}	0.29 _{0.01}
	All	58.4	0.36	50.1	0.11	57.1	0.31	58.4	0.36	50.4	0.57	59.9	0.27	61.1 _{0.6}	0.27 _{0.00}

Table 1 Deep search and coding performance (pass@1) and price per million tokens (\$/Mt) across task-complexity bins. We compare SALE with the best single agent, the Willingness-to-Pay router (WTP), the TensorOpera Router (TO-Router), FrugalGPT, and an ablated variant of SALE without memory-based self-refinement (SALE w/o memory). For SALE, we report five runs with distinct randomized test-set orders, with standard deviations shown as subscripts.

pass@1 improvement over the best single agent while achieving cost reductions of 17–25%. On the most complex coding tasks, it improves pass@1 by 3.3 points at 19% lower cost than the best single agent. Across both domains and all complexity levels, the auction ensemble dominates the single-agent Pareto frontier—no fixed model attains higher pass@1 at equal or lower price per million tokens—indicating that strategy-based routing with self-improvement yields strictly better performance–cost trade-offs than any single model size. All reported improvements are statistically significant even accounting for run-to-run variance (see Appendix F), confirming that these gains are robust across random orderings.

Comparison with Existing Routers. SALE is deliberately lightweight: it leverages agents’ existing strategic-planning abilities through a simple, low-dimensional scoring function with few global weights, rather than learning a separate high-capacity routing model. This design makes SALE directly applicable to off-the-shelf agents with minimal tuning, while explicitly accounting for agent cost rather than optimizing for performance alone. We compare against four baseline routers. Willingness-to-Pay (WTP) (Hu et al., 2024) uses nearest-neighbor retrieval over pretrained sentence embeddings to predict the model with the best cost-performance tradeoff given a task description. CARROT (Somerstep et al., 2025) fine-tunes an encoder to jointly estimate per-query cost and accuracy, routing to the model that minimizes a weighted combination of both. TensorOpera Router (TO-Router) (Stripelis et al., 2024) trains a learned task classifier—a language model encoder fine-tuned on soft performance labels—to predict the best-performing model from the task description alone, without explicitly modeling cost. All three are *predictive* routers that select an agent before execution. FrugalGPT (Chen et al., 2024), by contrast, is a *non-predictive* cascade that executes agent trajectories sequentially until a fine-tuned encoder model accepts a response, potentially running multiple traces per task. For fair comparison, we train all baselines on the same training split used to set SALE’s scoring weights (hyperparameters in Appendix G). Unlike the baselines, SALE requires no learned predictor—only a fixed scoring form with a small number of tunable weights.

As shown in Table 1, WTP yields modest cost reductions on deep-search tasks (11% on average) but slightly underperforms the best single agent at most complexity levels. On coding, WTP achieves large savings but increasingly sacrifices pass@1 as complexity rises, dropping to 6.3% at the highest complexity versus 22.8% for the best single agent. CARROT strikes a better balance, reducing cost by 22% on deep search and 14% on coding while incurring only small accuracy drops overall, though it still underperforms SALE on both metrics. TO-Router tends to default to the strongest agent, so both performance and cost remain close to the single-agent baseline. FrugalGPT matches or slightly exceeds the best single agent on low-complexity tasks (e.g., 97.5% vs. 95.0% on simple coding), but as complexity grows, its pass@1 declines sharply while its average spend increases—rising to 0.61\$/Mt on coding versus the 0.36\$/Mt of the best agent. This exposes a limitation of non-predictive routing in agentic settings: not only does the cascade potentially incur the cost of multiple full traces, but the scoring model also struggles to assess answer reliability when the mapping

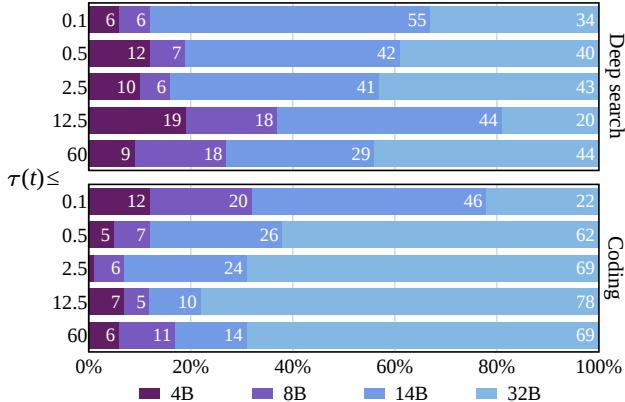


Figure 4 SALE’s average workload allocation across the 4B, 8B, 14B, and 32B agents for deep search (top) and coding (bottom) tasks, stratified by task complexity $\tau(t)$. Bar labels indicate the share of all tasks assigned to each agent.

Task type	$\tau(t)$	SALE w/o memory				SALE			
		4B	8B	14B	32B	4B	8B	14B	32B
Deep search	≤ 0.1	22.0	23.9	24.1	30.0	25.6	24.2	26.3	23.9
	≤ 0.5	21.7	23.6	24.5	30.2	24.1	25.4	25.2	24.8
	≤ 2.5	19.8	21.6	24.7	33.9	23.4	26.7	25.7	24.2
	≤ 12.5	10.9	23.6	29.3	36.2	20.0	22.6	27.3	30.0
	≤ 60	0.0	13.9	38.9	47.2	7.1	24.2	35.6	33.1
Coding	≤ 0.1	23.2	23.8	24.3	28.7	26.2	24.5	23.6	25.8
	≤ 0.5	21.4	23.2	24.0	31.4	19.1	24.8	26.4	29.7
	≤ 2.5	16.2	20.5	28.0	35.3	25.0	25.0	25.0	25.0
	≤ 12.5	3.8	17.4	22.0	56.8	13.2	27.3	35.8	23.7
	≤ 60	0.0	9.6	24.6	65.8	9.3	26.5	33.8	30.4

Table 2 Average Shapley values of each agent’s marginal contribution to the overall system, with and without memory-based self-refinement, across task types and complexity bins. Values are normalized to sum to 100 and reported as each agent’s percentage share of the total contribution.

from task to solution is indirect and mediated by long, complex trajectories. In contrast, SALE maintains or improves pass@1 relative to the best single agent across all complexity levels while reducing cost by 33–53% on deep search and 17–50% on coding, thereby advancing the performance–cost Pareto frontier more consistently than any alternative router (Figure 6). Comparisons to an oracle router are provided in Appendix H.

Ablation. To isolate the contribution of self-refinement, we ablate the memory-based stage and evaluate a variant of SALE that performs only a single auction: agents bid with strategic plans, and tasks are assigned by minimizing cost-minus-value over these bids. Even without memory, this router either matches or improves the average pass@1 of the best single agent while always reducing effective cost across all task-complexity bins, indicating that strategy-based routing provides a clear benefit. Comparing this ablated variant to the full SALE system (Table 1) shows that the memory mechanism consistently improves the trade-off: incorporating past auction outcomes either reduces cost at similar accuracy or jointly improves pass@1 and cost, thereby pushing the Pareto frontier further outward. Further ablations (Appendix I) study the effect of removing each term from the cost–value function, finding that all contribute meaningfully to performance, and show that the jury’s diversity provides a regularizing effect that no single judge or smaller jury subset can replicate.

7 Analysis

7.1 Agent Allocation

Figure 4 shows how SALE allocates workload across agents of different sizes and task-complexity bins for both deep search and coding tasks. For deep search, across all bins the 32B agent’s share ranges from 20% to 44%, with the remaining workload routed to smaller agents. The 14B agent handles 29–55% of tasks, while the 4B and 8B agents together account for approximately 12–37%. Even in the highest-complexity bin, smaller agents (4B and 8B) still process nearly 30% of tasks, indicating that SALE substantially reduces reliance on the largest model while matching or exceeding its accuracy. For coding tasks, the 32B agent is still used for a substantial proportion of the workload in all but the easiest complexity bin, where its share is only 22%. The smaller 4B and 8B agents together account for between 7% and 32% of coding queries across bins, again demonstrating that a substantial fraction of work can be offloaded from the largest model.

7.2 Agent Contributions via Shapley Values

Given the cooperative nature of SALE, where agents propose solution strategies and influence one another through jury votes and shared memory, it is important to quantify each agent’s overall contribution to the system, both when selected for final inference and through indirect effects on other agents. Formally, we define a cooperative game (\mathcal{A}, ν) in which the players are agents and the value of a coalition $\mathcal{A}' \subseteq \mathcal{A}$ is the

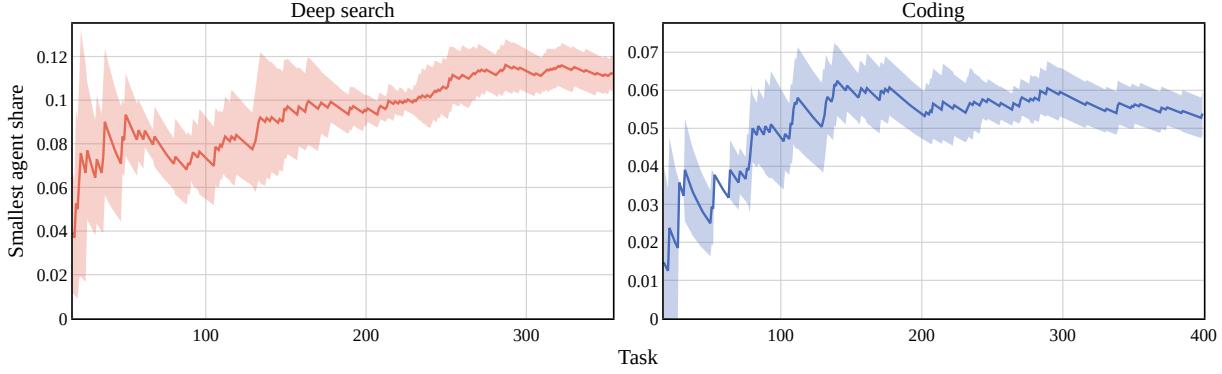


Figure 5 Cumulative share of tasks routed to the smallest agent over time. Solid lines show the mean across 5 runs with randomized task orderings; shading denote ± 1 standard deviation. An upward trend indicates that the local selection rate exceeds the historical average, reflecting increased delegation to the smallest agent as auction history accumulates.

performance achieved by running SALE with participation restricted to \mathcal{A}' . We then quantify each agent’s average marginal contribution using Shapley values (Lundberg and Lee, 2017), where ϕ_i denotes agent i ’s average marginal contribution to ensemble accuracy across all possible coalitions:

$$\phi_i = \sum_{\mathcal{A}' \subseteq \mathcal{A} \setminus \{i\}} \frac{|\mathcal{A}'|!(|\mathcal{A}| - |\mathcal{A}'| - 1)!}{|\mathcal{A}'|!} [\nu(\mathcal{A}' \cup \{i\}) - \nu(\mathcal{A}')].$$

Here, \mathcal{A} is the set of agents, $\nu(\mathcal{A}')$ is the expected utility induced by our cost–value selection mechanism when only agents in \mathcal{A}' participate in all roles (bidding, jury scoring, and memory-based refinement). The weighting is the probability that coalition \mathcal{A}' precedes agent i in a random ordering. Table 2 reports these values with and without memory-based self-refinement. We observe that in the without-memory setting, the largest agent has the highest Shapley values across all complexity bins and task domains, even though it is not the most-selected agent in all settings (Figure 4). This indicates that SALE benefits from the largest agent’s contribution in jury scoring, and yet saves inference costs by choosing smaller agents for solving the task. Further, we observe that introducing memory consistently lowers the 32B agent’s Shapley value across task domains and complexity bins, while the smaller 4B and 8B agents generally gain marginal contribution, especially on higher-complexity coding tasks.

It is worth noting that, when computing Shapley values, we remove the target agent from all roles in SALE, including the candidate pool, the jury, and the memory bank. Thus, an agent’s Shapley value captures its total contribution to the system—its ability to contribute effective strategies directly as well as its indirect impact. This explains, for example, why the 4B model can attain a relatively high Shapley value on coding tasks despite being selected infrequently for final inference: it still improves the ensemble through judgement and memory contributions. Hence the distributions in Figure 4 and Table 2 need not correlate.

7.3 Smallest Agent Selection Over Time

Beyond static routing, SALE enables smaller agents to progressively “scale up” by learning from auction feedback, effectively expanding their competitiveness over time. We test this by tracking how often the smallest (4B) agent is selected as the final executor over time. Figure 5 plots the cumulative selection rate of the 4B agent: the running fraction of all tasks processed so far that were ultimately routed to this agent. For deep search, this cumulative share increases from 3.7% in the early portion of the evaluation to 11.1% by the final tasks, approximately a threefold increase. For coding, the cumulative share grows from 1.4% to 5.3%, nearly a fourfold increase, with a marked rise over the first ~ 150 tasks followed by a plateau and a slight decline near the end. Overall, the proportion of workload handled by the smallest agent increases over time as a growing memory bank yields increasingly relevant auction feedback that progressively scales up the practical contribution of small agents. This temporal dynamic distinguishes SALE from conventional routers, which treat model selection as a stationary mapping from task features to agents. Similar plots for all agents are shown in Appendix J.

7.4 Qualitative Analysis of Refined Strategies

Consistent with recent observations on reusable behaviors in LLM reasoning (Didolkar et al., 2025), we find that memory-guided refinement systematically grounds strategies in auction feedback by reusing recurrent structural elements from past winning bids on similar tasks. For search tasks, refined strategies more frequently (i) explicitly mention tools and their arguments, (ii) impose tighter search-space constraints to specified reputable sources, and (iii) add intermediate cross-reference checks for ambiguity or inconsistencies. For coding tasks, refinement more often (i) specifies function and helper-function names and arguments precisely, (ii) maintains stronger alignment with the final objective (e.g., returning code rather than its runtime output), and (iii) performs systematic testing, including both the use of provided tests and the construction of additional test cases and edge-case checks. Across both domains, refined strategies also adopt a clearer layout and step organization. Table 9 reports the proportion of selected refined strategies in which each pattern appears, and Appendix E.4 shows representative examples illustrating all behaviors.

7.5 Complementary Failure Modes

For SALE to outperform any single agent (as shown in Table 1), there must exist tasks where smaller agents succeed and the largest agent fails, including cases where smaller agents have improved through prior auction feedback. If the largest agent’s errors were a strict subset of smaller agents’ errors, routing would offer no accuracy benefit and only cost savings. In Appendix K, we investigate such complementary failure modes, providing qualitative grounding for the quantitative improvements observed in Section 6.

Across task types and complexity levels, we find that failures of the largest agent often stem less from a lack of underlying capability and more from how that capability is exercised. In particular, the largest agent is more prone to overconfident behavior: it sometimes bypasses available tools in favor of parametric recall, over-engineers straightforward problems, or skips basic verification steps. Smaller agents, by contrast, more often adhere to simpler strategies, lean more heavily on tools, and perform explicit checks. Crucially, these tendencies are visible in the initial strategic plans agents submit before any trajectory is executed, implying that the auction has access to a reliable signal for predicting failure-mode divergence at bid time, without needing to run trajectories to completion. That said, these patterns do not mean that smaller agents are generally more accurate—the largest agent remains superior on aggregate, especially at higher complexities (Section 4). They do, however, suggest a consistent complementarity in failure modes: some tasks are handled better by simpler, tool-centric strategies than by the largest agent’s approach. SALE is designed to exploit this by using these early differences in strategy as the main signal for how work should be divided.

8 Conclusion

We investigated how task complexity affects the relative performance of small and large language-model agents, and how to allocate work across them efficiently. On deep search and coding tasks spanning multiple horizons, smaller agents perform comparably to larger ones on simple instances but fall substantially behind on more complex ones. This suggests that small agents alone are insufficient for complex workloads, whereas always defaulting to the largest model ignores substantial opportunities for efficiency.

To address this, we proposed SALE, a strategy-auction framework where heterogeneous agents bid with short strategic plans, are scored by a cost-value objective, and refine bids using shared auction memory. SALE runs entirely at test time on off-the-shelf models, without training a separate router, and introduces only a negligible additional inference overhead beyond executing the final trajectory. Empirically, across both deep search and coding domains, it improves the pass@1 of the strongest single agent while reducing cost and shifting a substantial fraction of workload away from the largest model, adaptively improving smaller agents over time so they can shoulder more of the work and maintaining these gains even on the most complex tasks. Our findings indicate that scaling individual models is only one axis of progress, and that *how* we structure and coordinate agents can be an equally powerful lever. Rather than treating capability as a property of a single, ever-larger model, SALE treats it as an emergent property of a system that allocates work, prices compute, and lets agents learn from each other and adapt their strategies over time. This points toward a view of agentic AI where advances are driven not just by stronger models, but by the coordination mechanisms and division of labor that bind them into effective and adaptive systems.

Limitations

We study two domains—deep search and coding—that are canonical benchmarks in the agentic AI literature and exercise complementary capabilities: search emphasizes retrieval and multi-step exploration, while coding emphasizes generation and logical reasoning. Together they cover diverse agentic patterns with distinct tool-use profiles and offer objective, automatable evaluation metrics. That said, they do not span all applications of agentic systems; future work can apply SALE to additional task families (e.g., data analysis or long-form report writing) to test how broadly our findings generalize.

On the modeling side, we work with Qwen3 models from 4B to 32B parameters. We focus on a single model family for our complexity-scaling analysis because cross-family comparisons would confound scale effects with architecture, tokenizer, and training recipe differences, making it impossible to isolate how model size mediates performance as task complexity grows. Qwen3 is the only contemporary open-weight suite offering a dense, consistently-trained ladder ($4B \rightarrow 8B \rightarrow 14B \rightarrow 32B$) suitable for this methodology; in contrast, other open-weight families offer narrower size ranges, larger gaps between sizes, or mix architectures across scales, and closed-source models do not disclose parameter counts. Importantly, SALE is model-agnostic: the auction mechanism and cost–value objectives do not depend on model-specific properties, so our findings about when to route to larger versus smaller models should transfer qualitatively to other families. The 4B–32B range already yields a clear task–complexity-dependent performance gap and a roughly $8\times$ cost spread, but it still sits below the largest frontier models. That said, much of the empirical literature on scaling behavior draws inferences from trends observed across multiple smaller, systematically spaced model sizes; our controlled size ladder is designed to support that style of analysis by isolating scale while holding other factors as constant as possible. Evaluating SALE with substantially larger models (e.g., 70B+) would be a useful extension to assess how the cost–value trade-offs behave when agents are even more capable and more expensive.

Additionally, our auction memory bank grows linearly with the number of tasks. In our experiments, memory size remained tractable and did not affect performance; however, extended deployment over thousands of tasks may benefit from memory management strategies such as sliding windows, importance-weighted sampling, or summarization. These are standard techniques in memory-augmented systems and can be integrated without modifying the core auction mechanism.

Finally, our cost accounting focuses on language-model tokens and does not explicitly price tool calls. This reflects standard practice in agentic benchmarks and is appropriate for our experimental setup, where token costs dominate overall spend. The SALE cost function is modular: extending it to incorporate tool pricing (e.g., commercial APIs, simulators, or human-in-the-loop services) requires only adding corresponding cost terms without modifying the auction mechanism. We leave this extension to future work targeting deployments where tool costs are non-negligible.

Broader Impacts

This work contributes to the understanding of how task complexity mediates the effectiveness of language-model agents and proposes a coordination mechanism for efficiently allocating work across heterogeneous models. We discuss several dimensions of potential impact.

On the marketplace metaphor. We employ auction-based coordination and freelancer-marketplace terminology as conceptual tools for organizing AI agents, not as prescriptions for labor markets. The analogy is strictly methodological: it motivates a mechanism design perspective on multi-agent systems in which bids, competition, and learning from feedback govern allocation among software components. Our framework neither models human workers nor recommends substituting them with AI. We emphasize that the “agents” in our system are language models executing computational tasks within sandboxed research environments, and the “marketplace” is a metaphor for principled resource allocation—distinct from, and not intended to inform, policies regarding human employment.

Efficiency and environmental considerations. By reducing reliance on the largest agent by approximately 53% and lowering overall inference cost by 35%, SALE promotes more efficient use of computational resources. Given the substantial energy footprint of large-scale language-model inference, mechanisms that route simpler

tasks to smaller models—without degrading performance—can contribute to more sustainable AI deployment. We view this as a modest but positive externality of our work.

Democratizing access to capable AI. Our findings suggest that carefully coordinated ensembles of smaller, less expensive models can match or exceed the performance of larger models on heterogeneous workloads. If such coordination mechanisms become practical, they may lower the cost barrier to deploying capable agentic systems, potentially broadening access beyond well-resourced institutions. However, we acknowledge that the benefits of efficiency gains are not automatically equitably distributed, and deployment contexts will shape who ultimately benefits.

Dual-use considerations. As with most advances in AI capabilities and efficiency, our contributions are dual-use. More efficient agentic systems could be applied to beneficial domains (e.g., scientific research, accessibility tools) or to applications with negative societal consequences. We do not foresee unique risks introduced by SALE beyond those already present in the underlying language models and agentic frameworks; our contribution is to coordination, not to novel capabilities. Nonetheless, we encourage practitioners to consider deployment contexts carefully.

Acknowledgments

We would like to thank Enrique Alfonseca, Misha Bilenko, Cheng Zhang, Yue Zhang, Igor Tufanov, Virginie Do, Emilien Garreau, Amine Benhalloum, Mathieu Rita, Romain Froger, Lovish Madaan, Anirudh Goyal, Iva Simon-Bubalo, Cindy Lee, Derek George Chan, Jordan Ward, and Joshua Lim for their valuable technical guidance and support in the development of this work. We are also grateful to Parag Jain, Amar Budhiraja, Graeme Nail, Thomas Scialom, Grégoire Mialon, Marin Vlastelica, Jenny Zhang, Md Rifat Arefin, Ulyana Piterbarg, Shashwat Goel, Philipp Mondorf, and Dulhan Jayalath for insightful discussions that helped shape and refine this research.

References

Lisa Alazraki, Maximilian Mozes, Jon Ander Campos, Tan Yi-Chern, Marek Rei, and Max Bartolo. No need for explanations: LLMs can implicitly learn from mistakes in-context. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33179–33203, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1686. <https://aclanthology.org/2025.emnlp-main.1686/>.

Danial Amin. Bayesian orchestration of multi-LLM agents for cost-aware sequential decision-making, 2026. <https://arxiv.org/abs/2601.01522>.

Soufiane Amini, Yassine Benajiba, Cesare Bernardis, Paul Cayet, Hassan Chafi, Abderrahim Fathan, Louis Faucon, Damien Hilloul, Sungpack Hong, Ingo Kossyk, Tran Minh Son Le, Rhicheek Patra, Sujith Ravi, Jonas Schweizer, Jyotika Singh, Shailender Singh, Weiyi Sun, Kartik Talamadupula, and Jerry Xu. Open agent specification (agent spec): A unified representation for AI agents, 2025. <https://arxiv.org/abs/2510.04173>.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. <https://arxiv.org/abs/2108.07732>.

Sher Badshah and Hassan Sajjad. Reference-guided verdict: LLMs-as-judges in automatic evaluation of free-form QA. In Chen Zhang, Emily Allaway, Hua Shen, Lesly Miculicich, Yinqiao Li, Meryem M’hamdi, Peerat Limkonchotiwat, Richard He Bai, Santosh T.y.s.s., Sophia Simeng Han, Surendrabikram Thapa, and Wiem Ben Rim, editors, *Proceedings of the 9th Widening NLP Workshop*, pages 251–267, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-351-7. doi: 10.18653/v1/2025.winlp-main.37. <https://aclanthology.org/2025.winlp-main.37/>.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic AI, 2025. <https://arxiv.org/abs/2506.02153>.

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, 2025. <https://arxiv.org/abs/2512.10696>.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, 2025. <https://openreview.net/forum?id=6s5uXNWG1h>.

Hui Chen, Miao Xiong, Yujie Lu, Wei Han, Ailin Deng, Yufei He, Jiaying Wu, Yibo Li, Yue Liu, and Bryan Hooi. MLR-bench: Evaluating AI agents on open-ended machine learning research. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025a. <https://openreview.net/forum?id=JX9DE6c0lf>.

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. ISSN 2835-8856. <https://openreview.net/forum?id=cSimKw5p6R>. Featured Certification.

Shuang Chen, Yue Guo, Yimeng Ye, Shijue Huang, Wenbo Hu, Haoxi Li, Manyuan Zhang, Jiayu Chen, Song Guo, and Nanyun Peng. ARES: Multimodal adaptive reasoning via difficulty-aware token-level entropy shaping, 2025b. <https://arxiv.org/abs/2510.08457>.

Xinghao Chen, Zhijing Sun, Guo Wenjin, Miaoran Zhang, Yanjun Chen, Yirong Sun, Hui Su, Yijie Pan, Dietrich Klakow, Wenjie Li, and Xiaoyu Shen. Unveiling the key factors for distilling chain-of-thought reasoning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 15094–15119, Vienna, Austria, July 2025c. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.782. <https://aclanthology.org/2025.findings-acl.782/>.

Daixuan Cheng, Shaohan Huang, Xuekai Zhu, Bo Dai, Wayne Xin Zhao, Zhenliang Zhang, and Furu Wei. Reasoning with exploration: An entropy perspective, 2025. <https://arxiv.org/abs/2506.14758>.

Tristan Coignion, Clément Quinton, and Romain Rouvoy. A performance study of LLM-generated code on leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE 2024, page 79–89. ACM, June 2024. doi: 10.1145/3661167.3661221. <http://dx.doi.org/10.1145/3661167.3661221>.

Zhuyun Dai, Vincent Y Zhao, Ji Ma, Yi Luan, Jianmo Ni, Jing Lu, Anton Bakalov, Kelvin Guu, Keith Hall, and Ming-Wei Chang. Promptagator: Few-shot dense retrieval from 8 examples. In *The Eleventh International Conference on Learning Representations*, 2023. <https://openreview.net/forum?id=gml46YMpu2J>.

Sarthak Dash, Sugato Bagchi, Nandana Mihindukulasooriya, and Alfio Gliozzo. Permutation invariant strategy using transformer encoders for table understanding. In Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz, editors, *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 788–800, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-naacl.59. <https://aclanthology.org/2022.findings-naacl.59/>.

Hassen Dhrif. Reasoning-aware prompt orchestration: A foundation model for multi-agent language model coordination, 2025. <https://arxiv.org/abs/2510.00326>.

Aniket Didolkar, Nicolas Ballas, Sanjeev Arora, and Anirudh Goyal. Metacognitive reuse: Turning recurring LLM reasoning into concise behaviors, 2025. <https://arxiv.org/abs/2509.13237>.

Paul Duetting, Vahab Mirrokni, Renato Paes Leme, Haifeng Xu, and Song Zuo. Mechanism design for large language models. In *The Web Conference 2024*, 2024. <https://openreview.net/forum?id=9Ob8Kmia9E>.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. ARE: Scaling up agent environments and evaluations, 2025. <https://arxiv.org/abs/2509.17158>.

Bingzheng Gan, Yufan Zhao, Tianyi Zhang, Jing Huang, Li Yusu, Shu Xian Teo, Changwang Zhang, and Wei Shi. MASTER: A multi-agent system with LLM specialized MCTS. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 9409–9426, Albuquerque, New Mexico,

April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.nacl-long.476. <https://aclanthology.org/2025.nacl-long.476/>.

Kai Goebel and Patrik Zips. Can LLM-reasoning models replace classical planning? a benchmark study, 2025. <https://arxiv.org/abs/2507.23589>.

Sara Hooker. On the slow death of scaling, 2025. <http://dx.doi.org/10.2139/ssrn.5877662>.

Sam Houliston, Ambroise Odonnat, Charles Arnal, and Vivien Cabannes. Provable benefits of in-tool learning for large language models, 2025. <https://arxiv.org/abs/2508.20755>.

Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. RouterBench: A benchmark for multi-LLM routing system. In *Agentic Markets Workshop at ICML 2024*, 2024. <https://openreview.net/forum?id=IVXmV8Uxwh>.

Xuedong Huang. Zoom AI sets new state-of-the-art benchmark on humanity's last exam, 2025. <https://www.zoom.com/en/blog/humanitys-last-exam-zoom-ai-breakthrough/>.

Kenan Jiang, Li Xiong, and Fei Liu. HARBOR: Exploring persona dynamics in multi-agent competition, 2025. <https://arxiv.org/abs/2502.12149>.

Minki Kang, Jongwon Jeong, Seanie Lee, Jaewoong Cho, and Sung Ju Hwang. Distilling LLM agent into small models with retrieval and code tools, 2025. <https://arxiv.org/abs/2505.17612>.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Roa Lin, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring AI ability to complete long software tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. <https://openreview.net/forum?id=CGNJL6CeV0>.

Tiziano Labruna, Jon Ander Campos, and Gorka Azkune. When to retrieve: Teaching LLMs to utilize information retrieval effectively. In *Proceedings of Recent Advances in Natural Language Processing*, page 623–632, 2024. <https://doi.org/10.26615/978-954-452-098-4-073>.

Byeongchan Lee, Jonghoon Lee, Dongyoung Kim, Jaehyung Kim, Kyungjoon Park, Dongjun Lee, and Jinwoo Shin. Efficient LLM collaboration via planning, 2025. <https://arxiv.org/abs/2506.11578>.

Zeju Li, Jianyuan Zhong, Ziyang Zheng, Xiangyu Wen, Zhijian Xu, Yingying Cheng, Fan Zhang, and Qiang Xu. Compressing chain-of-thought in LLMs via step entropy, 2025. <https://arxiv.org/abs/2508.03346>.

Xuefeng Liu, Chih chan Tien, Peng Ding, Songhao Jiang, and Rick L. Stevens. Entropy-reinforced planning with large language models for drug discovery. In *Forty-first International Conference on Machine Learning*, 2024a. <https://openreview.net/forum?id=F3Ds71Xg01>.

Zhiwei Liu, Weiran Yao, Jianguo Zhang, Le Xue, Shelby Heinecke, Rithesh R N, Yihao Feng, Zeyuan Chen, Juan Carlos Niebles, Devansh Arpit, Ran Xu, Phil L Mui, Huan Wang, Caiming Xiong, and Silvio Savarese. BOLAA: Benchmarking and orchestrating llm autonomous agents. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024b. <https://openreview.net/forum?id=BUa5ekiHlQ>.

Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4768–4777, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9802–9822, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.546. <https://aclanthology.org/2023.acl-long.546/>.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*, 2024. <https://openreview.net/forum?id=fibxvahvs3>.

Stuart Mitchell, Michael OSullivan, and Iain Dunning. Pulp: a linear programming toolkit for python. *The University of Auckland, Auckland, New Zealand*, 65:25, 2011.

Sajad Mousavi, Ricardo Luna Gutierrez, Desik Rengarajan, Vineet Gundecha, Ashwin Ramesh Babu, Avisek Naug, Antonio Guillen, and Soumyendu Sarkar. N-critics: Self-refinement of large language models with ensemble of critics. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023. <https://openreview.net/forum?id=L7vC3OYb2p>.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, Michael Choi, Anish Agrawal, Arnav Chopra, Ryan Kim, Adam Khoja, Richard Ren, Jason Hausenloy, Oliver Zhang, Mantas Mazeika, Summer Yue, Alexandr Wang, and Dan Hendrycks. Humanity’s last exam, 2025. <https://arxiv.org/abs/2501.14249>.

Farrukh Bin Rashid and Saqib Hakak. Fathom: A fast and modular RAG pipeline for fact-checking. In Mubashara Akhtar, Rami Aly, Christos Christodoulopoulos, Oana Cocarascu, Zhijiang Guo, Arpit Mittal, Michael Schlichtkrull, James Thorne, and Andreas Vlachos, editors, *Proceedings of the Eighth Fact Extraction and VERification Workshop (FEVER)*, pages 258–265, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 978-1-959429-53-1. doi: 10.18653/v1/2025.fever-1.20. <https://aclanthology.org/2025.fever-1.20/>.

David M. Rothschild, Markus Mobius, Jake M. Hofman, Eleanor W. Dillon, Daniel G. Goldstein, Nicole Immorlica, Sonia Jaffe, Brendan Lucier, Aleksandrs Slivkins, and Matthew Vogel. The agentic economy, 2025. <https://arxiv.org/abs/2505.15799>.

Rana Salama, Jason Cai, Michelle Yuan, Anna Currey, Monica Sunkara, Yi Zhang, and Yassine Benajiba. MemInsight: Autonomous memory augmentation for LLM agents, 2025. <https://arxiv.org/abs/2503.21760>.

Aaron Scher. Observations about LLM inference pricing. In *Machine Intelligence Research Institute (MIRI) Technical Report*, 2025. <https://techgov.intelligence.org/blog/observations-about-lm-inference-pricing>.

Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. The illusion of diminishing returns: Measuring long horizon execution in LLMs, 2025. <https://arxiv.org/abs/2509.09677>.

Ilja Siroš, Dave Singelée, and Bart Preneel. GitHub copilot: The perfect code compLeeter?, 2024. <https://arxiv.org/abs/2406.11326>.

Seamus Somerstep, Felipe Maia Polo, Allysson Flavio Melo de Oliveira, Prattyush Mangal, Mírian Silva, Onkar Bhardwaj, Mikhail Yurochkin, and Subha Maity. CARROT: A cost aware rate optimal router, 2025. <https://arxiv.org/abs/2502.03261>.

Dimitris Stripelis, Zhaozhuo Xu, Zijian Hu, Alay Dilipbhai Shah, Han Jin, Yuhang Yao, Jipeng Zhang, Tong Zhang, Salman Avestimehr, and Chaoyang He. TensorOpera router: A multi-model router for efficient LLM inference. In Franck Dernoncourt, Daniel Preoțiu-Pietro, and Anastasia Shimorina, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 452–462, Miami, Florida, US, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-industry.34. <https://aclanthology.org/2024.emnlp-industry.34/>.

Zhihong Sun, Chen Lyu, Bolun Li, Yao Wan, Hongyu Zhang, Ge Li, and Zhi Jin. Enhancing code generation performance of smaller models by distilling the reasoning ability of LLMs. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue, editors, *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 5878–5895, Torino, Italia, May 2024. ELRA and ICCL. <https://aclanthology.org/2024.lrec-main.521/>.

Nenad Tomasev, Matija Franklin, Joel Z. Leibo, Julian Jacobs, William A. Cunningham, Iason Gabriel, and Simon Osindero. Virtual agent economies, 2025. <https://arxiv.org/abs/2509.10147>.

Asterios Tsiourvas, Wei Sun, and Georgia Perakis. Causal LLM routing: End-to-end regret minimization from observational data. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. <https://openreview.net/forum?id=iZC5xoQQkX>.

Pat Verga, Sebastian Hofstatter, Sophia Althammer, Yixuan Su, Aleksandra Piktus, Arkady Arkhangorodsky, Minjie Xu, Naomi White, and Patrick Lewis. Replacing judges with juries: Evaluating LLM generations with a panel of diverse models, 2024. <https://arxiv.org/abs/2404.18796>.

Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. AI agentic programming: A survey of techniques, challenges, and opportunities, 2025a. <https://arxiv.org/abs/2508.11126>.

Liang Wang, Nan Yang, and Furu Wei. Learning to retrieve in-context examples for large language models. In Yvette Graham and Matthew Purver, editors, *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1752–1767, St. Julian’s, Malta, March

2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.eacl-long.105. <https://aclanthology.org/2024.eacl-long.105/>.

Peilong Wang, Jason Holmes, Zhengliang Liu, Dequan Chen, Tianming Liu, Jiajian Shen, and Wei Liu. A recent evaluation on the performance of LLMs on radiation oncology physics using questions of randomly shuffled options. *Front. Oncol.*, 15, 2025b. doi: 10.3389/fonc.2025.1557064.

Shenzhi Wang, Le Yu, Chang Gao, Chujie Zheng, Shixuan Liu, Rui Lu, Kai Dang, Xiong-Hui Chen, Jianxin Yang, Zhenru Zhang, Yuqiong Liu, An Yang, Andrew Zhao, Yang Yue, Shiji Song, Bowen Yu, Gao Huang, and Junyang Lin. Beyond the 80/20 rule: High-entropy minority tokens drive effective reinforcement learning for LLM reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025c. <https://openreview.net/forum?id=yfcpdY4gMP>.

Weixuan Wang, Dongge Han, Daniel Madrigal Diaz, Jin Xu, Victor Rühle, and Saravan Rajmohan. OdysseyBench: Evaluating LLM agents on long-horizon complex office application workflows, 2025d. <https://arxiv.org/abs/2508.09124>.

Yu Wang and Xi Chen. MIRIX: Multi-agent memory system for LLM-based agents, 2025. <https://arxiv.org/abs/2507.07957>.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025e. <https://openreview.net/forum?id=NTAhi2JEEE>.

Jason Wei, Nguyen Karina, Hyung Won Chung, Yunxin Joy Jiao, Spencer Papay, Amelia Glaese, John Schulman, and William Fedus. Measuring short-form factuality in large language models, 2024. <https://arxiv.org/abs/2411.04368>.

Andrew White. About 30% of humanity's last exam chemistry/biology answers are likely wrong, 2025. <https://www.futurehouse.org/research-announcements/hle-exam>.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Hufeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code LLMs, 2025. <https://arxiv.org/abs/2504.14655>.

Zhitian Xie, Qintong Wu, Chengyue Yu, Chenyi Zhuang, and Jinjie Gu. Profile-aware maneuvering: A dynamic multi-agent system for robust GAIA problem solving by AWorld, 2025. <https://arxiv.org/abs/2508.09889>.

Siheng Xiong, Zhangding Liu, Jieyu Zhou, and Yusen Su. Deliberate planning in language models with symbolic representation. In *Twelfth Annual Conference on Advances in Cognitive Systems*, 2025a. <https://openreview.net/forum?id=uJHpaZlIvT>.

Weimin Xiong, Yifan Song, Qingxiu Dong, Bingchan Zhao, Feifan Song, XWang, and Sujian Li. MPO: Boosting LLM agents with meta plan optimization. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 3914–3935, Suzhou, China, November 2025b. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.210. <https://aclanthology.org/2025.findings-emnlp.210/>.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. <https://openreview.net/forum?id=FiM0M8gcct>.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengan Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yingqi Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. <https://arxiv.org/abs/2505.09388>.

Yingxuan Yang, Ying Wen, Jun Wang, and Weinan Zhang. Agent exchange: Shaping the future of AI agent economics, 2025b. <https://arxiv.org/abs/2507.03904>.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural*

Language Processing, pages 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. <https://aclanthology.org/D18-1259/>.

Weizhi Zhang, Yangning Li, Yuanchen Bei, Junyu Luo, Guancheng Wan, Liangwei Yang, Chenxuan Xie, Yuyao Yang, Wei-Chieh Huang, Chunyu Miao, Henry Peng Zou, Xiao Luo, Yusheng Zhao, Yankai Chen, Chunkit Chan, Peilin Zhou, Xinyang Zhang, Chenwei Zhang, Jingbo Shang, Ming Zhang, Yangqiu Song, Irwin King, and Philip S. Yu. From web search towards agentic deep research: Incentivizing search with reasoning agents, 2025. <https://arxiv.org/abs/2506.18959>.

Kehang Zhu, John Joseph Horton, Yanchen Jiang, David C. Parkes, and Anand V. Shah. Evidence from the synthetic laboratory: Language models as auction participants. In *NeurIPS 2024 Workshop on Behavioral Machine Learning*, 2024. <https://openreview.net/forum?id=FB9mTtJpJI>.

Appendix

A Dataset

We describe here how we construct HST-BENCH, its composition across source datasets and complexity bins, and the details of the human solution-time annotation protocol.

A.1 Data Composition

HST-BENCH is built from existing open-source benchmarks spanning deep search and coding. Concretely, we draw from SimpleQA (Wei et al., 2024), PopQA (Mallen et al., 2023), HotpotQA (Yang et al., 2018), GAIA (Mialon et al., 2024), Humanity’s Last Exam (HLE) (Phan et al., 2025), MBPP (Austin et al., 2021), and LeetCode (Xia et al., 2025). In addition, we construct a small corpus of multiple-choice coding questions, which we refer to as Coding-MCQ (see example questions in Appendix A.4, to better populate the lowest-complexity bin for coding tasks. We randomly sample instances from the official test splits of each benchmark. In order to ensure label quality, we validate all samples, discarding and replacing those for which it is not possible to derive the provided ground-truth answer from the question. For HLE, we restrict to chemistry and biology questions that have been validated by domain experts (White, 2025). For GAIA, we sample from the validation split, which includes human solution times collected under comparable experimental conditions (timed, independent problem-solving by proficient users) and verified by the original authors; we directly reuse these annotations. After sampling, we annotate and aggregate human solution times for each instance and assign it to one of the five non-overlapping complexity bins defined in Section 3.1, based on its average human solution time. Table 3 reports, for each complexity bin, how many HST-BENCH instances originate from each source dataset. This reveals a shift from short-form factual QA and CODING-MCQ in the lower-complexity bins toward tasks demanding extended agentic workflows: multi-source information retrieval, cross-referencing, and synthesis for reasoning benchmarks (e.g., HotpotQA, GAIA, HLE), and iterative implementation with intermediate testing and debugging for coding problems (e.g., LeetCode ‘Hard’) in the higher-complexity bins.

Domain	Complexity bin	# tasks	Source	
			Dataset	Percentage
<i>Deep search</i>	$0 < \tau(t) \leq 0.1$	80	SimpleQA PopQA HotpotQA	38% 50% 13%
	$0.1 < \tau(t) \leq 0.5$	80	SimpleQA PopQA HotpotQA	8% 5% 88%
	$0.5 < \tau(t) \leq 2.5$	80	HotpotQA HLE	98% 3%
	$2.5 < \tau(t) \leq 12.5$	82	HotpotQA GAIA	2% 98%
	$12.5 < \tau(t) \leq 60$	32	HLE	100%
	$0 < \tau(t) \leq 0.1$	80	Coding-MCQ	100%
<i>Coding</i>	$0.1 < \tau(t) \leq 0.5$	79	MBPP	100%
	$0.5 < \tau(t) \leq 2.5$	80	MBPP LeetCode (Medium)	99% 1%
	$2.5 < \tau(t) \leq 12.5$	81	MBPP LeetCode (Medium)	2% 98%
	$12.5 < \tau(t) \leq 60$	79	LeetCode (Hard)	100%

Table 3 Composition of HST-BENCH by complexity bin (grouped by average human solution time $\tau(t)$, in minutes). We report the percentage of instances contributed from each source dataset to each bin, rounded to the nearest integer.

The distribution of source datasets across complexity bins reflects the design intent of existing benchmarks, many of which target specific difficulty ranges. This naturally results in a greater proportion of certain datasets within specific bins.

In addition to the test split, we construct separate development sets for both domains. For deep search, the development set contains 68 instances sampled from SimpleQA; for coding, it comprises 88 instances drawn from 40 Coding-MCQ questions and 48 LeetCode ‘Easy’ problems. These development sets are disjoint from the test data and reflect the need for instances on which models exhibit a balanced mix of successes and failures to enable effective validation and tuning.

A.2 Data Annotation

To obtain human solution times for HST-BENCH, we recruited a pool of paid annotators who are graduates in computer science or closely related fields, with demonstrated expertise in programming and familiarity with the types of deep search and coding problems we study. This helps ensure that the reported solution times reflect the behavior of reasonably proficient users rather than novices. For each task, we collect solution-time annotations from at least three distinct annotators, who work independently and use only tools permitted by the task guidelines (e.g., a web browser and search engine for search tasks, or a local editor/IDE for coding), while refraining from language models or other assistants that could directly solve the task. Annotators are given written task-specific guidelines (Section A.3), read each task once in full, then start a stopwatch, solve the task as quickly as possible while maintaining accuracy, and finally submit both their measured solution time and final answer (or code). For LeetCode ‘Hard’ tasks, due to annotation cost constraints, we do not collect new human timings and instead rely on published human time estimates reported by Siroš et al. (2024). To verify consistency, we independently annotate a random subset of 8 tasks ($\sim 10\%$) and confirm that all measured times fall within the published ranges.

All collected annotations undergo a subsequent quality-control pass. First, submitted solutions are checked for correctness. Once a minimum of three correct solution times has been collected for a given task, we lightly filter for outliers to reduce the influence of anomalous timings (e.g., due to interruptions or misunderstandings), and collect further annotations if necessary. Concretely, solution times associated with incorrect answers or that deviate by more than two standard deviations from the task-wise mean are removed from the dataset. Once quality control and any necessary data re-collection have concluded, the times for each task are averaged together. We find good inter-annotator agreement across HST-BENCH (CCI = 0.83, 95% CI [0.81, 0.85]; Krippendorff’s α = 0.86, 95% CI [0.84, 0.87], $p < 0.001$), indicating that human solution times are reliably reproducible.

A.3 Annotators’ Guidelines

Below we reproduce the instructions provided to annotators for both deep search and coding tasks. These guidelines specify the allowed tools, what constitutes a correct solution in each domain, and how annotators should measure and report their solution times, ensuring consistency across annotators and task types.

Deep Search Guidelines. *Read these instructions very carefully. Only after you have understood them well, navigate to the tasks in the next tab.*

Goal

The goal of this annotation exercise is to label how much time it will take a human (not an LLM) to solve a given question.

You will be provided with questions and you will need to solve each with web searches, using Google or Bing.

You will need to use a stopwatch to measure your task completion time.

Task completion time must be reported in the format <HH hours MM minutes SS seconds>, for example, 25 seconds would be written as <00 hours 00 minutes 25 seconds>. DO NOT REPORT MILLISECONDS, EVEN IF YOUR STOPWATCH SHOWS THEM.

BE FAST: We are trying to measure a human's *BEST* completion time, so please complete the task (correctly) as quickly as you can. While for most questions you will likely need web search, it is fine not to use it if you already know the answer. For some tasks, it is likely that you will need multiple, in-depth web searches.

It is assumed that the search engine is already open in a tab. To avoid wasting time unnecessarily, please arrange the windows on your screen so that you can see both the question and the search engine side by side at the same time.

Solve the task by following these steps:

- Step 1: Read the question first, slowly and carefully.
- Step 2: Start the stopwatch.
- Step 3: Text can be copy-pasted to the search engine directly from the question. Indeed, for most questions this is advisable as it can save time. As soon as the answer to the question is found, stop the stopwatch (i.e., do not wait to type the answer) and record the completion time.
- Step 4: Provide the answer and the task completion time (as per the stopwatch).

Note: You are allowed to read the question directly from the AI-generated summary at the top of the search engine page, if this is given. However, you are not allowed to copy-paste the question into an LLM chat interface. Use Google or Bing search.

Examples:

Question:

What is Miley Cyrus' occupation?

Completion Time: 00 hours 00 minutes 04 seconds

Your Answer: Singer, songwriter, actress

—

Question:

Which came out first, Titanic or Clueless?

Completion Time: 00 hours 00 minutes 17 seconds

Your Answer: Clueless

Coding Guidelines. Read these instructions very carefully. Only after you have understood them well, navigate to the tasks in the next tab.

Goal

The goal of this annotation exercise is to label how much time it will take a human (not an LLM) to solve a given coding question.

You will be provided with coding questions and you will need to solve each by writing code.

You will need to use a stopwatch to measure your task completion time.

Task completion time must be reported in the format <HH hours MM minutes SS seconds>, for example, 25 seconds would be written as <00 hours 00 minutes 25 seconds>. DO NOT REPORT MILLISECONDS, EVEN IF YOUR STOPWATCH SHOWS THEM.

BE FAST: We are trying to measure a human's *BEST* completion time, so please complete the task (correctly) as quickly as you can. You are allowed to use web search to look up syntax, however please do not overuse web search

unnecessarily, as it tends to increase the completion time.

If the question requires writing code, you MUST use a Python shell which allows running code at the click of a button. For example, use Google Colab or <https://pythonhow.com/python-shell>. For code-writing questions, you will be provided with one single test to check your code. We will run your code on more tests later to validate its correctness.

It is assumed that the Python shell and the search engine are already open in a window. To avoid wasting time unnecessarily, please arrange the windows on your screen so that you can see both the question text, the coding editor and the search engine side by side at the same time.

Solve the task by following these steps:

- Step 1: Read the question first, slowly and carefully.
- Step 1a: If the question requires writing a Python function, copy the function header and, at the bottom, the given test into your Python shell *BEFORE* you start the stop watch. The required function name and arguments will be clear from the test.

For example you may have: `def my_function(my_arg):`

```
assert my_function(3)==True
```

So that when the stopwatch starts you will only need to write the function body.

- Step 2: Start the stopwatch.
- Step 3a: If the question is multiple choice, stop the stopwatch as soon as the correct answer has been identified (no need to type it anywhere) and record the completion time.
- Step 3b: If the answer requires writing code, stop the stopwatch as soon as you have completed and run the code, and record the completion time.
- Step 4: Provide the answer and the task completion time (as per the stopwatch).

Note: You are allowed to use Google, but not allowed to use AI Assistants.

Examples:

Question:

Which of the following lines of code is the correct way to raise a to the power of b in python? Give only the number corresponding to the answer, and nothing else.

- 1: `a^b`
2: `a**b`

Completion Time: 00 hours 00 minutes 02 seconds

Your Answer: 2

—

Question:

Write a python function to find the first even number in a given list of numbers.

Your function should satisfy the following test:

```
assert first_even ([1, 3, 5, 7, 4, 1, 6, 8]) == 4
```

Completion Time: 00 hours 00 minutes 39 seconds

Your Answer:

```
def first_even(nums):  
    first_even = next((el for el in nums if el%2==0), -1)  
    return first_even
```

A.4 Coding-MCQ Examples

Below are representative multiple-choice questions from the Coding-MCQ dataset, designed to assess performance on short, low-complexity coding tasks that target core programming concepts.

Which of the following lines of code prints the word ‘hello’? Give only the number corresponding to the answer, and nothing else.

- 1: `print('hello') if 1%2==0 else print('goodbye')`
- 2: `print('goodbye') if 1%2==0 else print('helloworld')[5:]`

Which of the following lines of code will not throw an error in Python? Give only the number corresponding to the answer, and nothing else.

- 1: `100 & 100`
- 2: `100.0 & 100.0`

Which of the following files is a configuration file? Give only the number corresponding to the answer, and nothing else.

- 1: `run_agent.yaml`
- 2: `README.md`
- 3: `run_agent.py`

Which of the following lines of code returns an empty list in python? Give only the number corresponding to the answer, and nothing else.

- 1: `[elem for elem in [2,3,4,5] if elem // 2 == 0]`
- 2: `[elem for elem in [2,3,4,5] if elem % 2 == 0]`

Which of the following lines of code correctly replaces a character in a string in Python? Give only the number corresponding to the answer, and nothing else.

- 1: `"a,b,d".replace("d", "c")`
- 2: `[char if char in "a,b" else "c" for char in "a,b,d"]`

B Estimated Cost of Running Models

Inference prices for Qwen3 models vary substantially across providers and deployment settings, reflecting differences in supported context length, geographical region, and commercial factors such as traffic volume and competition.² To obtain a simple, reproducible cost model for our experiments, we adopt an empirically calibrated pricing schedule. Our approach is grounded in recent empirical analyses of inference markets demonstrating that, for dense models, per-token prices scale approximately linearly with the number of parameters (Scher, 2025).

Following this established relationship, we model cost as proportional to the number of parameters and anchor our schedule using publicly advertised prices from established inference providers. Specifically, at the time of writing Groq reports separate prices for input and output tokens for Qwen3 32B,³ listing

$$\$0.29/\text{Mt} \text{ for input tokens} + \$0.59/\text{Mt} \text{ for output tokens},$$

where Mt denotes one million tokens. We use these figures as a reference anchor for a high-capacity Qwen3 model and scale costs for other sizes in proportion to their parameter counts.

In our agentic runs we consistently observe an average input-to-output token ratio of about 4:1 across task domains and horizons. Under this assumption, we take the expected cost per million *total* tokens for an agent instantiated with Qwen3 32B to be

$$\pi(a_{32B}) = \frac{4 \cdot 0.29 + 1 \cdot 0.59}{5} \approx \$0.36/\text{Mt}.$$

Applying the same linear scaling in parameter count yields the effective prices per million total tokens used in our experiments:

$$\pi(a_{4B}) = \$0.05, \quad \pi(a_{8B}) = \$0.09, \quad \pi(a_{14B}) = \$0.16,$$

where $\pi(a_{4B}) \approx 0.045$ is rounded to \$0.05.

Empirical Validation. To verify that the linear scaling assumption in practice for the other Qwen3 sizes in our experiments, we compare our derived prices against independently advertised rates from major inference providers at the time of writing.⁴⁵⁶ Since providers list separate prices for input and output tokens, we compute comparable per-million-total-token rates by applying the same 4:1 input-to-output weighting used in our estimates. Table 4 reports this comparison. The mean absolute deviation between our estimates and observed provider averages is within 6%, confirming that the linear approximation is well-supported for this model family.

Model	Estimated (\$/Mt)	Provider Avg. (\$/Mt)	Deviation
Qwen3-4B	0.05	0.05	0%
Qwen3-8B	0.09	0.09	0%
Qwen3-14B	0.16	0.17	6%

Table 4 Comparison of estimated prices (derived via linear scaling from Qwen3 32B) against average advertised prices across providers (Nebius, Novita, Alibaba Cloud). Deviations are within 6%, validating the linear cost model.

Scope of the Cost Model. Our cost model assumes access via third-party inference APIs, where infrastructure overhead—including hardware provisioning, energy consumption, and maintenance—is fully absorbed into the provider’s per-token pricing. Under usage-based API billing, the user incurs costs only for tokens consumed, making \$/Mt the appropriate metric for our analysis. We note that latency and throughput vary substantially across providers, regions, and time of day, making them difficult to model consistently (though smaller models are typically also faster); we therefore leave them outside the scope of our study. Per-token pricing, by contrast, is publicly advertised and stable, providing a reproducible basis for cost comparison.

²<https://huggingface.co/datasets/reach-vb/inference-provider-pricing>

³<https://groq.com/pricing>

⁴<https://nebius.com/token-factory/prices>

⁵<https://novita.ai/pricing>

⁶<https://www.alibabacloud.com/help/en/model-studio/models>

C Environment, Prompts and Hyperparameters

All experiments are conducted within the open-source Agent Research Environment (ARE), which provides a standardized, tool-augmented interface for evaluating heterogeneous language-model agents on real-world tasks. Unless otherwise noted, agents, tools, and evaluation protocols follow the default ARE configuration. In the subsections below, we describe the exact model hyperparameters, environment prompts, and other implementation details needed to fully reproduce our setup.

C.1 Model Hyperparameters

All experiments were run on NVIDIA A100 and H100 GPU clusters with 40–80 GB of HBM per accelerator. We use the same decoding configuration across all agents. The full set of model- and decoding-related hyperparameters used in our experiments is summarized in Table 5.

	Max length	Temperature	Top-p	Top-k	Batch size
Values	40,960	0.0	1.0	0	10

Table 5 Decoding and batching hyperparameters used for all Qwen3 agents.

C.2 Environment Hyperparameters

We run all experiments under the default configuration of ARE. Each episode is terminated as soon as either the time or iteration budget is exhausted, and agents must return a single final solution (i.e., we report pass@1 under the environment’s LLM-as-a-judge evaluation). Notably, the LLM-as-a-judge evaluation is straightforward in our setup: search outputs are directly matched against ground truth, while coding outputs—though potentially differing lexically—can be reliably compared for functional equivalence by models like GPT-4o. We refer readers to the ARE default configuration for LLM-as-a-judge prompts and other standard hyperparameters not explicitly mentioned here. We report environment hyperparameters in Table 6.

Hyperparameter	Value	Description
task_timeout_seconds	3600	Maximum wall-clock time per task
max_iterations	100	Maximum agent steps per task
llm_judge	GPT-4o	Base model for LLM-as-a-judge evaluation

Table 6 Environment-level limits used for all tasks; an episode terminates when either limit is reached.

Tool	Description
ask_search_agent	Delegates a natural-language query to a web search agent and returns its response.
inspect_file_as_text	Reads a file from the workspace as markdown text and returns its contents for subsequent inspection and reasoning
final_answer	Submits the agent’s final solution and terminates the episode.
Python environment	Executes Python code for calculations, data manipulation, and lightweight scripting; it is preconfigured with the standard library and commonly used packages sufficient to solve the benchmark tasks.

Table 7 Tools and execution environment available to agents. Deep search tasks use all tools; coding tasks use ask_search_agent, final_answer, and the Python environment.

For deep search tasks, each episode begins with a *fact extraction* pre-step, followed by a *strategy planning*

step. For coding tasks, the agent performs only a strategic planning step without explicit fact extraction. Table 7 summarizes the tools and Python execution environment available to the agent in each domain and provides brief descriptions of their functionality. For all remaining environment details (e.g., the exact tool interfaces, the format of observations returned to the agent, and error handling), we refer the reader to the original ARE paper (Froger et al., 2025).

C.3 Retrieval Hyperparameters

We use embedding-based retrieval over the shared auction memory: at each episode, the agent retrieves strategies that both won and lost past auctions for the k most similar tasks to the current one. Following established practice, we set $k = 8$, which prior studies commonly find to be a strong practical trade-off between coverage/diversity and context/latency overhead (Dai et al., 2023; Wang et al., 2024; Rashid and Hakak, 2025). Table 8 summarizes the retrieval hyperparameters used in all experiments.

	Embedding model	Distance metric	Top-k
Values	all-MiniLM-L6-v2	cosine	8

Table 8 Strategy retrieval hyperparameters used across all experiments.

C.4 Prompts

We detail here the *judge* prompt for scoring candidate strategies, the *strategy* prompts used to generate $s_{t,i}$ for a task t , and the *refinement* prompts to produce $s_{t,i}^r$. For coding tasks, we omit fact extraction before planning as the task specification is self-contained. Strategy refinement uses a template similar to that introduced by Alazraki et al. (2025) to facilitate contrastive learning from prior outcomes. For all remaining prompts used to interact with the environment, tools, and task wrappers, refer to the standard ARE configuration.

Judge Prompt

Provide an integer reward score between 0 and 5 (inclusive) for the quality of the provided plan steps, using strict evaluation standards. Ensure the reward reflects how effectively the plan contributes to progressing toward the correct solution.

Problem Statement:

```
***begin problem statement***
{task}
***end problem statement***
```

Plan:

```
{plan}
```

Be harsh in your evaluation. Only plans that you are extremely confident will succeed should be assigned the maximum score.

Score: [Strictly provide an integer reward score between 0 and 5]

Strategy Prompt (Deep Search)

You are a world expert at making efficient plans to solve any task using a set of carefully crafted tools.

Now for the given task, develop a step-by-step high-level plan taking into account the following inputs and list of facts.

This plan should involve individual tasks based on the available tools, that if executed correctly will yield the correct answer.

Do not skip steps, do not add any superfluous steps. Only write the high-level plan, DO NOT DETAIL INDIVIDUAL TOOL CALLS.

After writing the final step of the plan, write the '`<end_plan>`' tag and stop there.

Always search for the exact task at the beginning. If you are given an external file, always inspect it first to explore its content.

Do a very concise plan that only focus on the given task.

Do not attempt to answer the question without calling tools, even if you know the answer. You must always use at least one tool to find the answer.

Here is your task:

Task:

```

{task}

```

Your plan can leverage any of these tools:

{tool_descriptions}

List of facts that you know:

```

{answer\_facts}

```

Now begin! Write your plan below.

Strategy Prompt (Coding)

You are a world expert at making efficient plans to solve any task using a set of carefully crafted tools.

Now for the given task, develop a step-by-step high-level plan taking into account the following inputs.

This plan should involve individual tasks based on the available tools, that if executed correctly will yield the correct answer.

Do not skip steps, do not add any superfluous steps. Only write the high-level plan, DO NOT DETAIL INDIVIDUAL TOOL CALLS.

After writing the final step of the plan, write the '<end_plan>' tag and stop there.

Do a very concise plan that only focus on the given task.

Do not attempt to answer the question without calling tools, even if you know the answer. You must always use at least one tool to find the answer.

Here is your task:

Task:

```

{task}

```

Your plan can leverage any of these tools:

{tool_descriptions}

Now begin! Write your plan below.

Strategy Refinement Prompt (Deep Search)

You are a world expert at making efficient plans to solve any task using a set of carefully crafted tools.

Now for the given task, develop a step-by-step high-level plan taking into account the following inputs and list of facts.

This plan should involve individual tasks based on the available tools, that if executed correctly will yield the correct answer.

Do not skip steps, do not add any superfluous steps. Only write the high-level plan, DO NOT DETAIL INDIVIDUAL TOOL CALLS.

After writing the final step of the plan, write the '<end_plan>' tag and stop there.

Always search for the exact task at the beginning. If you are given an external file, always inspect it first to explore its content.

Do a very concise plan that only focus on the given task.

Do not attempt to answer the question without calling tools, even if you know the answer. You must always use at least one tool to find the answer.

Here is your task:

Task:

```

{task}

```

Your plan can leverage any of these tools:

{tool_descriptions}

List of facts that you know:

```

{answer\_facts}

```

Below you will find some example tasks followed by two corresponding plans - one plan that lost in a previous plan competition and one that won. Use these examples to understand what makes a plan lose or win.

{retrieved_tasks_and_plans}

Now apply what you have learned and given the task and a corresponding losing plan, write a winning plan.

{previous_losing_plan}

Winning plan:

Strategy Refinement Prompt (Coding)

You are a world expert at making efficient plans to solve any task using a set of carefully crafted tools.

Now for the given task, develop a step-by-step high-level plan taking into account the following inputs.

This plan should involve individual tasks based on the available tools, that if executed correctly will yield the correct answer.

Do not skip steps, do not add any superfluous steps. Only write the high-level plan, DO NOT DETAIL INDIVIDUAL TOOL CALLS.

After writing the final step of the plan, write the '<end_plan>' tag and stop there.

Do a very concise plan that only focus on the given task.

Do not attempt to answer the question without calling tools, even if you know the answer. You must always use at least one tool to find the answer.

Here is your task:

Task:

```  
{task}  
```

Your plan can leverage any of these tools:

{tool_descriptions}

Below you will find some example tasks followed by two corresponding plans - one plan that lost in a previous plan competition and one that won. Use these examples to understand what makes a plan lose or win.

{retrieved_tasks_and_plans}

Now apply what you have learned and given the task and a corresponding losing plan, write a winning plan.

{previous_losing_plan}

Winning plan:

D Cost-Value Optimization

D.1 Cost and Value Function Design

Strategy Length in Cost Function. In our setting, strategy length correlates with realized trajectory length (Spearman’s $\rho = 0.39$, Pearson’s $r = 0.36$, both $p < 0.001$), consistent with Goebel and Zips (2025). The close alignment between rank-based and linear correlation measures indicates a stable monotonic relationship rather than an artifact driven by outliers, reinforcing that longer strategies generally yield longer trajectories. In agentic planning, such a stable and consistent proxy signal provides meaningful guidance for cost estimation. Although the correlation is moderate in absolute magnitude, this level of predictive strength is sufficient in practice: prior work on agentic orchestration demonstrates that proxies of relatively moderate strength can meaningfully inform decision-making, even in the presence of systematic over- or underestimation (Amin, 2026). Crucially, including $|s_{t,i}|$ in $C_{t,i}$ improves development set accuracy, and removing it degrades performance in ablations (Appendix I).

Entropy and Jury in Value Function. We use normalized entropy $H(s_{t,i})$, computed as the mean per-token entropy from generation log-probabilities, as a lightweight signal of informational content/non-redundancy. In addition to being motivated by prior work linking higher-entropy planning to better outcomes (see Section 5); this term is also validated by development set performance and confirmed by ablations. Finally, we score each strategy with a full jury (including self-judgment): this configuration performs best on validation, learned weights calibrate each judge’s influence, and ablations show that removing any judge reduces accuracy (Appendix I).

D.2 Min-Max Formulation

In Mathematical Program 1, we provide the full mixed-integer linear program (MILP) for learning the scoring weights $w = (w_c, w_h, \{w_j\}_{a_j \in \mathcal{A}})$. Let \mathcal{T} denote the training set of tasks.

Mathematical Program 1 Min-Max Weight Optimization

$$\begin{aligned}
 & \min_{w, x, z, Q} && Q \\
 \text{s.t.} \quad & \sum_{a_i \in \mathcal{A}} x_{t,i} = 1 && \forall t \in \mathcal{T} \\
 & x_{t,i} \in \{0, 1\} && \forall t, a_i \\
 & z_t \geq C_{t,i} - V_{t,i} - M(1 - x_{t,i}) && \forall t, a_i \\
 & z_t \leq C_{t,i} - V_{t,i} + M(1 - x_{t,i}) && \forall t, a_i \\
 & z_t \leq Q && \forall t \in \mathcal{T} \\
 & w \in \mathbb{R}^{2+|\mathcal{A}|}
 \end{aligned}$$

The big- M constraints ensure that $z_t = C_{t,i} - V_{t,i}$ only for the selected agent (i.e., when $x_{t,i} = 1$), while remaining inactive otherwise. We set $M = 10^4$ to exceed the observed range of $C_{t,i} - V_{t,i}$ scores across all task-agent pairs in the training set. Specifically, M exceeds $\max_{t,i} |C_{t,i} - V_{t,i}|$ by at least two orders of magnitude, ensuring numerical stability without introducing solver issues.

D.3 Implementation Details

We solve the MILP using PuLP (Mitchell et al., 2011) with the CBC solver. Weights are unconstrained. The optimization is performed separately on the development sets for deep search and coding tasks (see Appendix A for details), with $|\mathcal{A}| = 4$ agents.

The learned weights exhibit consistent patterns across both domains. Entropy receives the highest weight in both cases, suggesting that information density in the generated strategy is a strong indicator of expected performance. This aligns with prior findings on the informativeness of high-entropy reasoning (Li et al., 2025) and is further supported by our ablation study in Appendix I. The relative importance of individual judge models varies by domain: for deep search tasks, smaller model judgments receive higher weight, whereas for coding tasks, the largest agent (32B) contributes most to the value estimate.

D.4 Alternative Formulations

Rationale for the Optimization Objective. The difference $C_{t,i} - V_{t,i}$ can be interpreted as a *negative net utility*: lower cost and higher value both reduce this quantity, so minimizing it naturally favors agents that are both efficient and effective. This formulation mirrors classical economic frameworks where profit is defined as revenue minus cost. Crucially, because the weights $(w_c, w_h, \{w_j\})$ are learned jointly and left unconstrained, the optimization implicitly calibrates the relative scales of cost and value components without requiring manual normalization or threshold selection.

Normalized Features. An alternative approach is to normalize C and V before combining them, for instance by Z-score, standardizing all input features prior to learning the weights. While this ensures that cost and value terms are on comparable scales a priori, it discards meaningful structural information. In our setting, cost and value features operate on different scales—entropy in particular is orders of magnitude smaller than cost, while judge scores fall in between. These differences are not arbitrary; they reflect the heterogeneous nature of the signals: token-based costs, normalized entropy, and Likert-scale judgments each carry information at their natural scale. Our learned weights implicitly calibrate these differences, ensuring that smaller-scale features (particularly entropy, which receives the largest weight magnitude) contribute meaningfully to routing decisions.

To validate this intuition empirically, we evaluated a variant in which all input features are Z-score normalized using training-set statistics before weight optimization. This variant achieves lower accuracy while disproportionately selecting cheaper models. Inspection of the learned weights reveals the cause: under normalization, the optimizer discovers a degenerate solution in which several value-related weights become negative (e.g., $w_h < 0$, $w_{32B} < 0$), effectively *penalizing* quality signals rather than rewarding them. This occurs because, once all features are rescaled to comparable magnitudes, the optimizer can minimize $C - V$ simply by selecting cheap models—there is no structural pressure to preserve value contributions. In contrast, our unnormalized formulation forces the learned weights to jointly encode both relative importance and scale calibration, preventing such degenerate solutions.

Constrained Optimization. Another alternative is constrained optimization, for example minimizing cost subject to $V > \eta$ for some threshold η . However, this formulation requires choosing η a priori, which is difficult in practice: the appropriate threshold may vary across task distributions, agent pools, and deployment settings. Setting η too high excludes cost-efficient agents that would have succeeded on easier tasks, while setting it too low allows poor-quality assignments to slip through on harder ones. Our unconstrained formulation sidesteps this problem altogether. The learned weights end up implicitly encoding the right cost–value trade-off for the training distribution, and because the min–max objective optimizes against worst-case tasks, we do not need to hand-pick thresholds for robustness.

In conclusion, normalization and constrained optimization both introduce additional design choices (normalization statistics, threshold values) that need to be tuned and often do not transfer well across settings. By minimizing $C - V$ with unconstrained, jointly learned weights, we avoid this overhead, while still obtaining strong empirical results. The natural scale differences between cost and value features are informative on their own, and the learned weights can exploit this, as confirmed by ablation experiments showing that adding explicit normalization hurts routing quality.

E Strategy Refinement

E.1 Algorithm

In Algorithm 2, we provide a complete description of the strategy refinement procedure outlined in Section 5. The algorithm details how initial bids are collected, how the provisional winner is selected, and how cheaper agents refine their strategies using retrieved contrastive examples from the auction memory \mathcal{M} .

Algorithm 2 Strategy Refinement from Auction Memory

Require: Task t , agents \mathcal{A} , memory \mathcal{M} , learned weights w , retrieval size k
Ensure: Final agent assignment $i^*(t)$

```

for each agent  $a_i \in \mathcal{A}$  do
    Generate initial strategy  $s_{t,i}$ 
    Compute cost  $C_{t,i} \leftarrow w_c \cdot \pi(a_i) \cdot |s_{t,i}|$ 
    Compute value  $V_{t,i} \leftarrow w_h \cdot H(s_{t,i}) + \sum_{a_j \in \mathcal{A}} w_j \cdot \gamma_j(s_{t,i})$ 
end for

 $\hat{i}(t) \leftarrow \arg \min_i (C_{t,i} - V_{t,i})$ 
 $i^*(t) \leftarrow \hat{i}(t)$ 

for each agent  $a_i \in \mathcal{A}$  where  $\pi(a_i) < \pi(a_{\hat{i}(t)})$  do
     $\tilde{k} \leftarrow \min(k, |\mathcal{T}_M|)$ 
    Retrieve  $\mathcal{T}' \leftarrow \text{top-}\tilde{k}_{t' \in \mathcal{T}_M} \text{sim}(t, t')$ 
     $\mathcal{M}_{t,i} \leftarrow \{(s_{t'}^{\text{lose}}, s_{t'}^{\text{win}})_i \mid t' \in \mathcal{T}'\}$ 
    Format  $\mathcal{M}_{t,i}$  as contrastive examples
    Generate refined strategy  $s_{t,i}^r$  conditioned on  $s_{t,i}$  and  $\mathcal{M}_{t,i}$ 
    Compute  $C_{t,i}^r \leftarrow w_c \cdot \pi(a_i) \cdot |s_{t,i}^r|$ 
    Compute  $V_{t,i}^r \leftarrow w_h \cdot H(s_{t,i}^r) + \sum_{a_j \in \mathcal{A}} w_j \cdot \gamma_j(s_{t,i}^r)$ 
    if  $C_{t,i}^r - V_{t,i}^r < C_{t,i^*(t)} - V_{t,i^*(t)}$  then
         $i^*(t) \leftarrow i$ 
    end if
end for

return  $i^*(t)$ 

```

E.2 Contrastive Example Selection

The refinement step exposes agents to contrastive pairs of winning and losing strategies from past auctions. Since the auction objective minimizes cost-minus-value, winning strategies tend to originate from cost-efficient agents—typically smaller models achieving competitive value. This aligns naturally with our refinement setting, where only cheaper agents refine: the positive examples shown to a refining agent come from models of comparable capability. Prior work has shown that smaller models can benefit from exposure to high-quality plans (Sun et al., 2024; Kang et al., 2025; Xiong et al., 2025b), yet often struggle to execute strategies designed by much larger models when the required reasoning exceeds their capacity (Chen et al., 2025c; Lee et al., 2025). By using auction winners—which inherently balance quality with executability—as positive examples, our approach provides smaller models with effective yet feasible strategies to learn from.

An alternative design would label contrastive examples by downstream task success rather than auction selection. However, this assumes access to a reliable automated evaluator—an assumption that often fails in practice, where ground-truth feedback may be sparse, noisy, or available only through human assessment or ground truth labels. The auction selection signal, by contrast, emerges naturally from the mechanism itself and requires no external oracle, making our refinement procedure broadly applicable.

E.3 Refinement Patterns

Table 9 reports, for both search and coding tasks, how often each refinement pattern appears in selected winning strategies. These statistics show that the observed structural elements recur consistently across tasks.

Domain	Refinement pattern	Prevalence
<i>Deep search</i>	Improved tool and argument specification	68%
	Provides (additional) trusted sources for search	15%
	Intermediate cross-reference/consistency checks	32%
	Clearer/more structured layout	49%
<i>Coding</i>	Precise function and helper signatures	33%
	Explicit focus on returning the correct artifact	30%
	Systematic testing and edge-case coverage	41%
	Clearer/more structured layout	26%

Table 9 Refinement patterns and their prevalence in chosen refined plans for search and coding tasks. Note that patterns are not mutually exclusive; a single refined plan may exhibit multiple patterns simultaneously.

E.4 Representative Examples

Below we show side-by-side representative examples of initial and refined strategies for individual tasks, together with the refinement patterns (see Table 9) identified in each. These examples illustrate how refinement turns loosely specified or linear strategies into more constrained, check-pointed procedures that score higher in the auction. In all examples, the unrefined strategy is rejected and the refined one is selected, yielding a successful downstream answer.

E.4.1 Deep Search Examples

Example 1
Task
Do The Importance of Being Icelandic and The Five Obstructions belong to different film genres?
Unrefined plan
<ol style="list-style-type: none"> 1. Use the <code>ask_search_agent</code> to determine the genre of “The Importance of Being Icelandic” by querying a reliable film database or source. Provide the query: “What is the primary film genre of ‘The Importance of Being Icelandic’? Provide the genre name.” 2. Use the <code>ask_search_agent</code> again to determine the genre of “The Five Obstructions” with the query: “What is the primary film genre of ‘The Five Obstructions’? Provide the genre name.” 3. Compare the two genre names obtained from the searches. If they are distinct, conclude that the films belong to different genres; otherwise, state they belong to the same genre.
Refined plan
<ol style="list-style-type: none"> 1. Use <code>ask_search_agent</code> to determine the primary film genre of “The Importance of Being Icelandic” by querying a reliable source like IMDb or Wikipedia. Provide <code>output_content</code> as the genre name. 2. Use <code>ask_search_agent</code> to determine the primary film genre of “The Five Obstructions” using the same method. Provide <code>output_content</code> as the genre name. 3. Compare the two genre names obtained from the searches. 4. Use <code>final_answer</code> to state whether the films belong to different genres based on the comparison.
Refinement patterns
Improved tool and argument specification, provides (additional) trusted sources for search

Example 2**Task**

According to Google Finance, when was the first year the Apple stock went above \$50 (without adjusting for stock split)?

Unrefined plan

1. Use the `ask_search_agent` to query Google Finance for the first date when Apple's stock (AAPL) exceeded \$50 without adjusting for stock splits. Specify the query as: "According to Google Finance, what was the first date Apple's stock (AAPL) went above \$50 without adjusting for stock splits? Provide the date."
2. Extract the year from the identified date.
3. Verify the result by cross-checking Apple's stock split history to ensure the price threshold is unadjusted.
4. Use the `final_answer` tool to confirm the first year.

Refined plan

1. Use `ask_search_agent` to query Google Finance for the earliest date Apple's stock (AAPL) exceeded \$50 **without adjusting for stock splits**. Query: "According to Google Finance, what was the first date Apple's stock (AAPL) closed above \$50 without adjusting for stock splits? Provide the date."
2. Extract the year from the retrieved date.
3. Use `ask_search_agent` to confirm Apple's stock split history to ensure the identified date is **pre-split** (e.g., check split dates and ratios). Query: "When did Apple (AAPL) undergo stock splits? Provide split dates and ratios."
4. Cross-reference the split dates with the retrieved date to verify it is unadjusted.
5. Use `final_answer` to submit the confirmed year.

Refinement patterns

Improved tool and argument specification, intermediate cross-reference/consistency checks

Example 3**Task**

How many pages of the 2023 IPCC report (85 pages version) mention nuclear energy?

Unrefined plan

1. Use `ask_search_agent` to search for the 2023 IPCC report's 85-page version text, specifying the need for the full content to analyze mentions of nuclear energy. Provide the query: "What is the full text of the 2023 IPCC report (85 pages version)? Provide the complete document content."
2. Use `inspect_file_as_text` to analyze the retrieved report text (assuming the search provides a file path or URL) and identify all instances of terms related to nuclear energy (e.g., "nuclear energy," "nuclear power," "nuclear fission").
3. Count the unique pages containing these terms from the inspected text.
4. Use `final_answer` to report the total number of pages mentioning nuclear energy.

Refined plan

1. Use `ask_search_agent` to locate the official 2023 IPCC report (85 pages version) and retrieve its full text or a downloadable PDF link from the IPCC website (<https://www.ipcc.ch/>). Provide the query: "Where can I access the full text of the 2023 IPCC report (85 pages version)? Provide a direct download link."
2. Use `inspect_file_as_text` to analyze the retrieved report text (via the provided link) and identify all instances of terms related to nuclear energy (e.g., "nuclear energy," "nuclear power," "nuclear fission").
3. Count the number of unique pages containing these terms, ensuring no duplicates.
4. Use `final_answer` to submit the total count of pages mentioning nuclear energy.

Refinement patterns

Provides (additional) trusted sources for search

Example 4**Task**

Who was the director of Chinese Coffee?

Unrefined plan

Use `ask_search_agent` to search for “Who is the director of the movie Chinese Coffee? Provide the name.” on IMDb (<https://www.imdb.com>).

Extract the director’s name from the search result.

Use `final_answer` to provide the identified director’s name.

Refined plan

1. Use `ask_search_agent` to search for “Who directed the movie Chinese Coffee?” and request the output to provide the director’s name.
2. Analyze the detailed search results to confirm the director’s identity from reliable sources (e.g., IMDb, Wikipedia).
3. Cross-check for title variations or potential ambiguities (e.g., “Chinese Coffee” vs. similar titles) in the search results.
4. Use `final_answer` to provide the confirmed director’s name.

Refinement patterns

Provides (additional) trusted sources for search, intermediate cross-reference/consistency checks, clearer/more structured layout

E.4.2 Coding Examples

Example 5**Task**

Write a function to verify validity of a string of parentheses. Your code should pass the following test: `assert is_valid_parenthese("()[]") == True`

Unrefined plan

1. Implement a function using a stack to track opening brackets.
2. Create a mapping of closing brackets to their corresponding opening brackets.
3. Iterate through each character in the input string.
4. Push opening brackets onto the stack.
5. For closing brackets, check if the stack is empty or the top element does not match the expected opening bracket.
6. Return True if the stack is empty after processing all characters, otherwise return False.

Refined plan

1. Analyze the task: Create a function to validate balanced parentheses using a stack-based approach.
2. Define a mapping of closing brackets to their corresponding opening brackets.
3. Iterate through each character in the input string, pushing opening brackets onto the stack and checking for matches when encountering closing brackets.
4. Return True if the stack is empty after processing all characters (indicating balanced parentheses), otherwise False.
5. Validate the function with the test case `assert is_valid_parenthese("()[]") == True`.
6. Use the `final_answer` tool to submit the implemented code.

Refinement patterns

Explicit focus on returning the correct artifact, systematic testing and edge-case coverage

Example 6**Task**

Write a function to calculate magic square. Your code should pass the following test: `assert magic_square_test([[7, 12, 1, 14], [2, 13, 8, 11], [16, 3, 10, 5], [9, 6, 15, 4]])==True`

Unrefined plan

1. Verify the input matrix is a square (all rows have the same length as the number of rows).
2. Check all elements are unique and within the range 1 to n^2 .
3. Calculate the magic constant as the sum of the first row.
4. Validate all rows, columns, and both diagonals sum to the magic constant.
5. Return True if all checks pass, otherwise False.

Refined plan

1. Define the function `magic_square_test` that takes a matrix as input.
2. Verify the matrix is a square (all rows have the same length as the number of rows).
3. Check all elements are unique and within the range 1 to n^2 (where n is the size of the matrix).
4. Calculate the magic constant using the formula `n * (n**2 + 1) // 2`.
5. Validate all rows, columns, and both diagonals sum to the magic constant.
6. Return True if all checks pass, otherwise False.
7. Use the `final_answer` tool to submit the implemented function as the solution.

Refinement patterns

Precise function and helper signatures, explicit focus on returning the correct artifact

Example 7**Task**

Write a function to sort a list of elements using comb sort. Your code should pass the following tests: `assert comb_sort([5, 15, 37, 25, 79]) == [5, 15, 25, 37, 79]`

Unrefined plan

1. Create a copy of the input list to avoid modifying the original.
2. Initialize the gap as the length of the list divided by 1.3, then cast to an integer.
3. While the gap is greater than 1, iterate through the list with the current gap, comparing elements i and $i + \text{gap}$. If they are out of order, swap them.
4. After each pass, update the gap by dividing it by 1.3 and casting to an integer.
5. Once the gap is 1, perform a final bubble sort pass to ensure the list is fully sorted.
6. Return the sorted list.

Refined plan

1. Implement the `comb_sort` function to sort a list using the comb sort algorithm.
2. Initialize the gap as the length of the list divided by 1.3, then cast to an integer.
3. While the gap is greater than 1, iterate through the list and compare elements i and $i + \text{gap}$. Swap them if they are out of order.
4. After each pass, reduce the gap by dividing it by 1.3 and cast to an integer, ensuring it does not fall below 1.
5. Once the gap is 1, perform a final pass to ensure the list is fully sorted (similar to bubble sort).
6. Return the sorted list.
7. Use the `final_answer` tool to submit the implemented function as the solution.

Refinement patterns

Precise function and helper signatures, explicit focus on returning the correct artifact

Example 8**Task**

Write a python function to find the sum of common divisors of two given numbers. Your code should pass the following test: `assert sum(10,15) == 6`

Unrefined plan

1. Compute the greatest common divisor (GCD) of the two input numbers.
2. Find all divisors of the GCD by iterating from 1 to the GCD and checking for divisibility.
3. Sum all the divisors found in step 2.
4. Return the computed sum as the result of the function.

Refined plan

1. **Understand the problem**: Create a Python function `sum(a, b)` that returns the sum of all common divisors of `a` and `b`.
2. **Compute GCD**: Use Euclidean algorithm to find the greatest common divisor (GCD) of the two numbers, as all common divisors of `a` and `b` are divisors of their GCD.
3. **Find divisors of GCD**: Efficiently find all divisors of the GCD by iterating up to its square root. For each divisor `i`, add both `i` and `gcd // i` to the sum (if they are distinct).
4. **Handle edge cases**: Ensure the function works for inputs where GCD is 0 (though inputs are positive integers as implied by the test case).
5. **Implement the function** in Python using the described logic.
6. **Verify correctness** with the provided test case `sum(10, 15) == 6`.
7. **Provide the final answer** using the `final_answer` tool.

Refinement patterns

Precise function and helper signatures, systematic testing and edge-case coverage, clearer/more structured layout

F Significance Tests

Methods with memory are sensitive to test-set ordering, so we evaluate memory-based methods across five runs with different random permutations; baselines without memory use a single run. To assess significance, we employ two-tailed one-sample t -tests and bootstrap confidence intervals (CIs), which provide robustness against distributional assumptions.

We compare against five baselines: the best single agent, WTP, CARROT, TO-Router, and FrugalGPT (Tables 10 and 11). On the full test set, SALE achieves significantly higher pass@1 than all baselines across both task types, with large effect sizes (all $|t| > 5$, $p < .005$); bootstrap CIs confirm these results. Within individual bins, a few comparisons yield mixed results across the two tests, potentially reflecting reduced power in relatively smaller samples; where they diverge, at least one test typically remains significant: FrugalGPT on coding $\tau(t) \leq 0.1$ ($p = .208$ for pass@1, though $$/Mt$ remains highly significant); best single agent and TO-Router on coding $\tau(t) \leq 60$ (borderline $p \approx .057$, with $$/Mt$ strongly significant); and CARROT on coding $\tau(t) \leq 0.5$ ($p = .339$), where CARROT’s marginal advantage over SALE is not statistically significant. For cost, SALE significantly reduces $$/Mt$ versus the best single agent, CARROT, TO-Router, and FrugalGPT across all conditions. Compared to WTP, SALE dominates on deep search (both higher pass@1 and lower $$/Mt$); on coding, WTP achieves lower cost but at substantially degraded performance.

Task type	$\tau(t)$	Best single agent				WTP				CARROT				TO-Router				FrugalGPT			
		Pass@1		$$/Mt$		Pass@1		$$/Mt$		Pass@1		$$/Mt$		Pass@1		$$/Mt$		Pass@1		$$/Mt$	
		t	p	t	p	t	p	t	p	t	p	t	p	t	p	t	p	t	p	t	p
Deep search	≤ 0.1	—	—	-37.95	<.001	—	—	-27.26	<.001	—	—	-13.90	<.001	—	—	-16.57	<.001	—	—	-67.35	<.001
	≤ 0.5	4.00	.016	-36.88	<.001	8.80	<.001	-28.86	<.001	8.80	<.001	-15.50	<.001	8.80	<.001	-26.19	<.001	28.80	<.001	-68.95	<.001
	≤ 2.5	7.76	.002	-35.28	<.001	9.80	<.001	-21.92	<.001	11.76	<.001	-16.57	<.001	9.80	<.001	-29.93	<.001	11.76	<.001	-80.71	<.001
	≤ 12.5	4.54	.011	-51.31	<.001	3.15	.035	-40.62	<.001	8.52	.001	-40.62	<.001	4.57	.010	-51.31	<.001	7.20	.002	-88.73	<.001
	≤ 60	6.00	.004	-11.09	<.001	10.96	<.001	-6.95	.002	10.96	<.001	-4.47	.011	6.00	.004	-11.09	<.001	6.00	.004	-30.95	<.001
	All	14.36	<.001	-47.43	<.001	20.12	<.001	-34.79	<.001	24.64	<.001	-22.14	<.001	17.65	<.001	-37.95	<.001	25.88	<.001	-94.87	<.001
Coding	≤ 0.1	6.50	.003	-91.00	<.001	8.90	<.001	9.00	<.001	6.50	.003	-91.00	<.001	6.50	.003	-91.00	<.001	1.50	.208	-106.00	<.001
	≤ 0.5	9.00	<.001	-23.52	<.001	23.80	<.001	32.61	<.001	-1.09	.339	5.88	.004	9.19	<.001	-23.52	<.001	49.08	<.001	-90.33	<.001
	≤ 2.5	6.00	.004	-26.94	<.001	36.00	<.001	58.79	<.001	36.00	<.001	13.88	<.001	6.00	.004	-26.94	<.001	50.80	<.001	-129.01	<.001
	≤ 12.5	2.98	.041	-6.42	.003	14.47	<.001	29.14	<.001	2.95	.042	-6.42	.003	2.95	.042	-6.42	.003	11.03	<.001	-35.10	<.001
	≤ 60	2.65	.057	-14.70	<.001	15.95	<.001	48.58	<.001	3.69	.021	-12.66	<.001	2.64	.058	-14.70	<.001	12.88	<.001	-65.73	<.001
	All	5.90	.004	-38.38	<.001	24.02	<.001	63.69	<.001	8.74	<.001	-17.96	<.001	5.90	.004	-38.38	<.001	23.37	<.001	-124.11	<.001

Table 10 One-sample t -test results comparing SALE against each baseline. We report t -statistics and p -values for pass@1 and $$/Mt$ across task types and complexity bins. Dashes (—) indicate undefined tests due to zero variance.

Task type	$\tau(t)$	Best single agent				WTP				CARROT				TO-Router				FrugalGPT			
		Pass@1	$$/Mt$	Pass@1	$$/Mt$	Pass@1	$$/Mt$	Pass@1	$$/Mt$	Pass@1	$$/Mt$	Pass@1	$$/Mt$	Pass@1	$$/Mt$	Pass@1	$$/Mt$	Pass@1	$$/Mt$	Pass@1	$$/Mt$
Deep search	≤ 0.1	[3.75, 3.75]	[-0.15, -0.14]	[7.45, 7.45]	[-0.11, -0.10]	[6.25, 6.25]	[-0.06, -0.05]	[4.95, 4.95]	[-0.07, -0.06]	[4.95, 4.95]	[-0.26, -0.25]	—	—	—	—	—	—	—	—	—	—
	≤ 0.5	[0.50, 1.25]	[-0.14, -0.13]	[1.70, 2.45]	[-0.11, -0.10]	[1.70, 2.45]	[-0.06, -0.05]	[1.70, 2.45]	[-0.10, -0.09]	[6.70, 7.45]	[-0.26, -0.25]	—	—	—	—	—	—	—	—	—	—
	≤ 2.5	[3.75, 5.75]	[-0.14, -0.13]	[5.00, 7.00]	[-0.09, -0.08]	[6.20, 8.20]	[-0.07, -0.06]	[5.00, 7.00]	[-0.12, -0.11]	[6.20, 8.20]	[-0.31, -0.30]	—	—	—	—	—	—	—	—	—	—
	≤ 12.5	[2.68, 5.61]	[-0.20, -0.19]	[1.41, 4.34]	[-0.16, -0.15]	[6.31, 9.24]	[-0.16, -0.15]	[2.71, 5.64]	[-0.20, -0.19]	[5.11, 8.04]	[-0.34, -0.33]	—	—	—	—	—	—	—	—	—	—
	≤ 60	[3.13, 5.00]	[-0.16, -0.12]	[6.23, 8.10]	[-0.11, -0.07]	[6.23, 8.10]	[-0.08, -0.04]	[3.13, 5.00]	[-0.16, -0.12]	[3.13, 5.00]	[-0.40, -0.36]	—	—	—	—	—	—	—	—	—	—
	All	[3.04, 3.83]	[-0.16, -0.14]	[4.44, 5.23]	[-0.12, -0.10]	[5.54, 6.33]	[-0.08, -0.06]	[3.84, 4.63]	[-0.13, -0.11]	[5.84, 6.63]	[-0.31, -0.29]	—	—	—	—	—	—	—	—	—	—
Coding	≤ 0.1	[2.25, 3.75]	[-0.19, -0.18]	[3.45, 4.95]	[0.01, 0.02]	[2.25, 3.75]	[-0.19, -0.18]	[2.25, 3.75]	[-0.19, -0.18]	[2.25, 3.75]	[-0.25, 1.25] [†]	[-0.22, -0.21]	—	—	—	—	—	—	—	—	—
	≤ 0.5	[1.77, 2.53]	[-0.09, -0.08]	[5.52, 6.28]	[0.12, 0.13]	[-0.78, -0.02]	[0.02, 0.03]	[1.82, 2.58]	[-0.09, -0.08]	[11.92, 12.68]	[-0.34, -0.33]	—	—	—	—	—	—	—	—	—	—
	≤ 2.5	[1.25, 2.00]	[-0.07, -0.06]	[8.75, 9.50]	[0.14, 0.15]	[8.75, 9.50]	[0.03, 0.04]	[1.25, 2.00]	[-0.07, -0.06]	[12.45, 13.20]	[-0.32, -0.31]	—	—	—	—	—	—	—	—	—	—
	≤ 12.5	[1.48, 4.94]	[-0.07, -0.04]	[13.84, 17.30]	[0.24, 0.27]	[1.44, 4.90]	[-0.07, -0.04]	[1.44, 4.90]	[-0.07, -0.04]	[10.14, 13.60]	[-0.32, -0.29]	—	—	—	—	—	—	—	—	—	—
	≤ 60	[1.27, 5.32]	[-0.08, -0.06]	[17.75, 21.80]	[0.23, 0.25]	[2.55, 6.60]	[-0.07, -0.05]	[1.25, 5.30]	[-0.08, -0.06]	[13.95, 18.00]	[-0.33, -0.31]	—	—	—	—	—	—	—	—	—	—
	All	[1.95, 3.55]	[-0.10, -0.09]	[10.25, 11.85]	[0.15, 0.16]	[3.25, 4.85]	[-0.05, -0.04]	[1.95, 3.55]	[-0.10, -0.09]	[9.95, 11.55]	[-0.31, -0.30]	—	—	—	—	—	—	—	—	—	—

Table 11 Bootstrap 95% confidence intervals (10,000 resamples) for the difference between SALE and each baseline. Positive pass@1 and negative $$/Mt$ indicate SALE outperforms the baseline. CIs are constructed by resampling SALE’s five runs against each baseline’s single-run reference value. [†]Not significant (CI includes 0).

G Baselines

G.1 Baseline Implementation Details

We provide implementation details and hyperparameter configurations for the baseline routing methods evaluated in Section 6.

WTP Router. The WTP router predicts model performance using a K -Nearest Neighbors classifier, then selects models based on a cost-performance trade-off controlled by the willingness-to-pay parameter. Following Hu et al. (2024), we use $k = 50$ neighbors with cosine distance over sentence embeddings. We tune the willingness-to-pay parameter on the deep search and coding training sets. Table 12 presents the hyperparameter configuration.

Hyperparameter	Value
k (neighbors)	50
Embedding model	all-MiniLM-L6-v2
Distance metric	Cosine
Willingness-to-pay (deep search)	5.0
Willingness-to-pay (coding)	5.0

Table 12 WTP router hyperparameters.

CARROT Router. CARROT (Somerstep et al., 2025) fine-tunes a RoBERTa encoder as a multi-label classifier to predict the probability of success for each candidate model given an input query. At inference time, routing decisions are made by selecting the model that maximizes a cost-performance tradeoff score: $(1 - \mu) \cdot p_i - \mu \cdot c_i$, where p_i is the predicted success probability and c_i is the normalized cost of model i . The hyperparameter $\mu \in [0, 1]$ controls the tradeoff between performance ($\mu = 0$) and cost ($\mu = 1$). Due to the limited size of our training set, we freeze the RoBERTa backbone and only train the classification head, which we found to yield more stable predictions. We set $\mu = 0.0$ based on development set performance; higher values of μ caused the router to route almost exclusively to the smallest agent, degrading task performance without meaningful cost savings. Table 13 presents the hyperparameter configuration.

Hyperparameter	Value
<i>Performance Predictor</i>	
Base model	roberta-base
Backbone	Frozen
Optimizer	AdamW
Learning rate	1×10^{-3}
Weight decay	1×10^{-2}
Batch size	16
Epochs	10
<i>Routing</i>	
Cost-performance tradeoff (μ)	0.0

Table 13 CARROT hyperparameters.

TO-Router. The TO-Router baseline fine-tunes a BERT encoder to predict model performance scores, using soft labels and inverse class frequency weighting to handle imbalanced performance distributions. We follow the implementation of [Stripelis et al. \(2024\)](#). Table 14 presents the hyperparameter configuration.

Hyperparameter	Value
Base model	bert-base-uncased
Optimizer	AdamW
Learning rate	5×10^{-5}
Weight decay	1×10^{-4}
Batch size	8
Epochs	5
Soft label temperature	10
Sample weighting	Inverse class frequency
Loss function	Cross-entropy

Table 14 TO-Router hyperparameters.

FrugalGPT. For the FrugalGPT baseline, we follow the implementation of [Chen et al. \(2024\)](#), which consists of two components: (i) a scoring function that predicts answer correctness, and (ii) a cascade optimizer that determines the optimal model ordering and acceptance thresholds. Table 15 presents the hyperparameter configuration.

Hyperparameter	Value
<i>Scoring Function</i>	
Base model	distilbert-base-uncased
Optimizer	AdamW
Learning rate	2×10^{-5}
Weight decay	1×10^{-2}
Warmup steps	500
Batch size	8
Epochs	8
Gradient clipping	1.0
<i>Cascade Optimization</i>	
Max cascade length	3
Quantile grid steps	20
Optimizer	scipy.brute + Nelder-Mead

Table 15 FrugalGPT hyperparameters.

G.2 Pareto Frontier vs. Baseline Routers

Figure 6 visualizes the accuracy–cost trade-offs reported in Section 6 (Table 1). On deep search tasks, SALE shifts the Pareto frontier outward for every value of τ . On coding, SALE improves the frontier in all bins except $\tau \leq 0.5$, where CARROT attains a comparable trade-off. We also note that WTP can achieve lower cost on coding tasks, but only with substantial accuracy degradation.

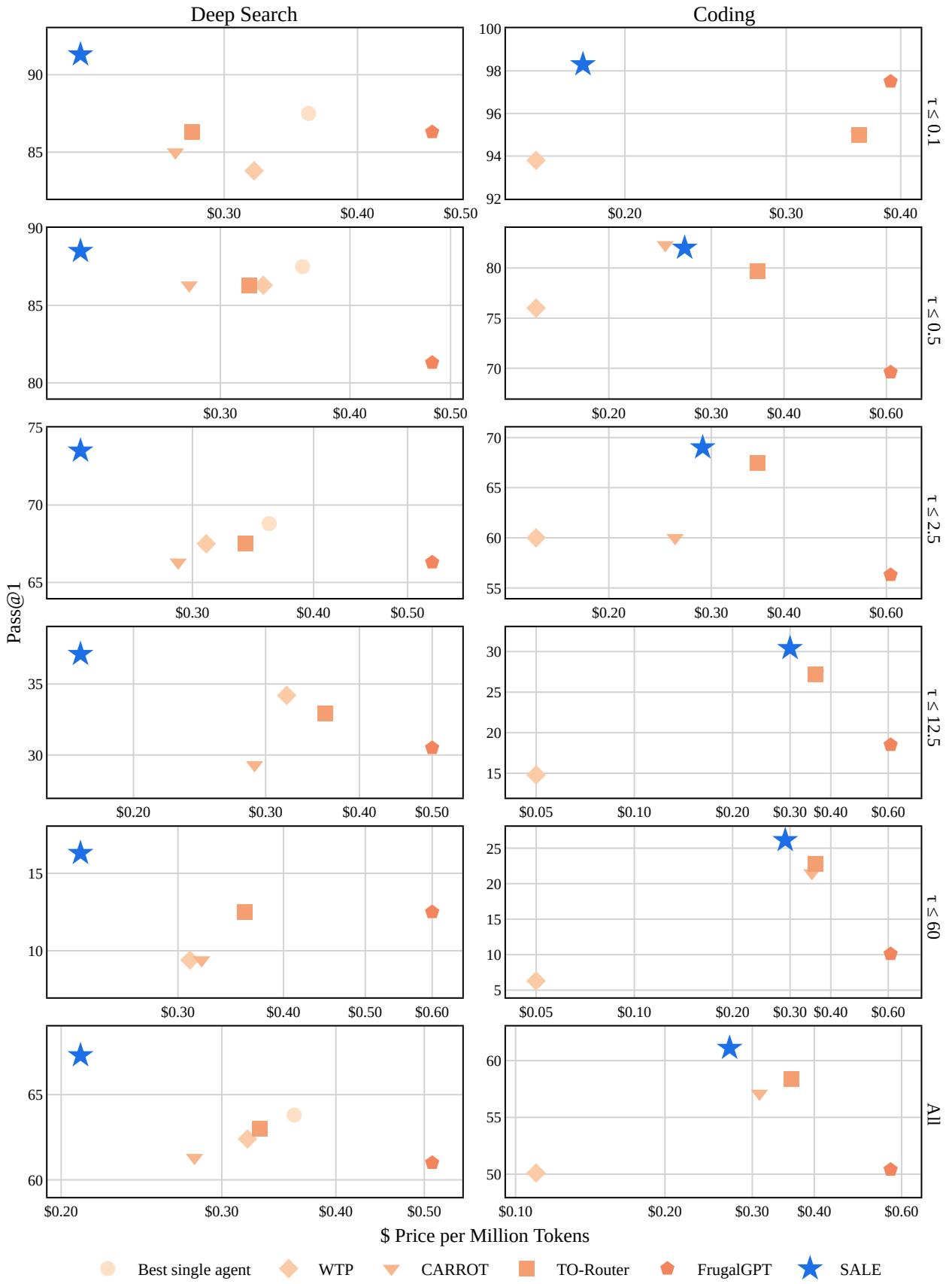


Figure 6 SALE vs. baselines on the accuracy–cost trade-off. The upper-left envelope is Pareto-optimal.

H Oracle Results and Agent Selection Analysis

H.1 Oracle Router

Table 16 reports the oracle router results on HST-BENCH, as an upper bound on what is achievable from per-task model selection alone. For each task t we run all candidate agents and select the smallest agent that produces a correct final answer. If all answers are incorrect, we select the smallest agent in the pool. This yields the minimum possible spend subject to achieving correctness whenever any agent in the pool can solve the task. We note, however, that in complexity bins where no agent solves a large fraction of instances, the oracle’s effective cost can be artificially low: on those unsolved tasks the choice of agent does not affect accuracy, and our definition defaults to the cheapest model, thereby minimizing expenditure in a regime where additional compute would not change outcomes. As a result, oracle cost in low-accuracy bins should be interpreted cautiously, primarily as a lower bound on spend rather than a representative operating point.

Relative to this hindsight upper bound, SALE captures a substantial fraction of the available routing gains on easier tasks, but the gap to oracle performance grows with task complexity. For deep search, at $\tau(t) \leq 0.1$ the oracle attains 97.5 pass@1 versus 91.3 for SALE (Table 1), while for coding the gap is only 0.5 points (98.8 vs. 98.3). At higher complexity, the oracle–SALE gap widens (deep search $\tau(t) \leq 60$: 25.0 vs. 16.3; coding $\tau(t) \leq 60$: 34.2 vs. 26.1). We also observe a growing gap between the oracle and the best single agent as task complexity increases, peaking in the medium–high complexity bins. This implies increased opportunity for routing in these instances.

While a gap between SALE and the oracle remains, SALE consistently exhibits the smallest accuracy gap to the oracle among all methods. On deep search, SALE’s gap to the oracle is 10.6 points (67.3 vs. 77.9), compared to 14.1–16.9 points for the baselines (WTP: 15.5; CARROT: 16.6; TO-Router: 14.9; FrugalGPT: 16.9; best single agent: 14.1). On coding, SALE’s accuracy gap is 7.3 points (61.1 vs. 68.4), whereas baselines range from 10.0 to 18.3 points behind the oracle (WTP: 18.3; FrugalGPT: 18.0; CARROT: 11.3; TO-Router and best single agent: 10.0). In cost, SALE also closes more of the gap on deep search (0.21 vs. oracle 0.07) than any baseline (0.28–0.51); on coding, WTP achieves lower cost (\$0.11) but at the expense of a substantially larger accuracy gap (18.3 points vs. SALE’s 7.3 points). In short, the oracle–SALE gap should be interpreted relative to the considerably wider oracle–baseline gaps: SALE captures more of the available routing gains than any existing method, and the residual headroom reflects the inherent difficulty of task-complexity prediction rather than a limitation unique to our approach.

Task type	$\tau(t)$	Oracle router	
		Pass@1(\uparrow)	\$/Mt(\downarrow)
Deep search	≤ 0.1	97.5	0.06
	≤ 0.5	93.8	0.06
	≤ 2.5	86.3	0.08
	≤ 12.5	56.1	0.08
	≤ 60	25.0	0.09
	All	77.9	0.07
Coding	≤ 0.1	98.8	0.05
	≤ 0.5	89.9	0.08
	≤ 2.5	76.3	0.08
	≤ 12.5	43.2	0.08
	≤ 60	34.2	0.09
	All	68.4	0.08

Table 16 Oracle-router performance (pass@1) and price per million tokens (\$/Mt) across task complexity bins for deep search and coding. The oracle selects, for each task, the smallest agent that returns a correct final answer; if no agent is correct, it defaults to the smallest agent.

H.2 Routing Diagnostics

To better understand SALE’s routing behavior and identify opportunities for improvement, we conduct a systematic error analysis comparing SALE’s agent selections over a single run against the oracle router described in Appendix H.1. We categorize each routing decision into one of four outcomes:

1. *Correct*: SALE selects the same agent as the oracle.
2. *Over-escalation*: SALE selects a larger (more expensive) agent than necessary.
3. *Under-escalation*: SALE selects a smaller agent that fails when a larger one would succeed.
4. *Unavoidable*: No agent in the pool produces a correct answer.

Deep Search Diagnostics. Figure 7 presents row-normalized confusion matrices comparing SALE’s agent selections against the oracle router described in Appendix H.1, both overall and stratified by complexity bin. Each cell shows the percentage of tasks where SALE selected a given agent (row) and the oracle selected another (column), annotated as: correct (✓, diagonal), over-escalation (↑, SALE selects larger than necessary), under-escalation (↓, SALE selects an agent that fails when a larger one succeeds), or unavoidable (—, no agent succeeds).

In the aggregate matrix ($n = 354$), the dominant off-diagonal mass lies in the leftmost column, indicating frequent over-escalation: when SALE selects the 14B or 32B agents, 65.1% and 61.0% of those cases could have been handled by the 4B agent. Under-escalation (↓) appears infrequently across all rows, confirming that SALE rarely sacrifices accuracy by selecting an insufficiently capable agent. Notably, when SALE does route to the 4B agent, it achieves the highest diagonal accuracy—54.0% overall, and even higher within individual bins (100% at $\tau \leq 0.1$, 75.0% at $\tau \leq 2.5$)—suggesting the system correctly identifies 4B-suitable tasks but triggers such routing too conservatively.

Across complexity bins, over-escalation decreases systematically: at $\tau \leq 0.1$, over-escalation from 32B selections to the 4B oracle accounts for 86.2%, dropping to 16.7% at $\tau \leq 12.5$ and near-zero at $\tau \leq 60$. This shift reflects a structural change in the error landscape: as complexity grows, the “none” column increasingly dominates (reaching 73–100% at $\tau \leq 60$), indicating that most failures become unavoidable regardless of routing.

Coding Diagnostics. Figure 8 shows row-normalized confusion matrices for coding-agent selections. Mirroring the deep-search pattern, over-escalation dominates the aggregate matrix ($n = 399$): when SALE selects 14B or 32B, the oracle often prefers smaller agents—55.2% of 14B selections could have been handled by 4B, and 31.5% of 32B selections map to 4B (with an additional 25.0% mapping to 8B). Under-escalation remains similarly limited: SALE’s 4B selections are predominantly correct (69.6% diagonal), with most remaining errors unavoidable (21.7%). This asymmetry—SALE prioritizing accuracy via conservative routing at the cost of compute—closely matches the deep-search behavior.

Stratifying by complexity reveals a parallel transition. At $\tau \leq 0.1$, small-agent routing is perfect (4B achieves 100%), but over-escalation is extreme: all 14B and 32B selections could have been routed to 4B (100% in both rows). As complexity increases ($\tau \leq 0.5$, $\tau \leq 2.5$), over-escalation persists while the “none” column grows. However, the hardest bin ($\tau \leq 60$) diverges from deep search: rather than collapsing to near-total unavoidable outcomes, coding shows more heterogeneous behavior—the “none” column reaches 50–78% across rows, but under-escalation also emerges (e.g., 4B shows 25% under-escalation to 8B). This indicates that high-complexity coding tasks retain meaningful variation in required agent capability, whereas deep-search tasks become predominantly unsolvable.

Overall, while SALE already achieves the smallest oracle gap among all evaluated routers (Section 6), the error analysis reveals that further improvements require further reducing over-escalation on low-complexity tasks where the oracle favors 4B yet SALE triggers larger agents. Importantly, this failure mode is conservative: SALE errs toward preserving accuracy rather than sacrificing correctness, and under-escalation remains rare across both domains. Moreover, when SALE does route to 4B, diagonal accuracy is high (54.0% for deep search, 69.6% for coding), suggesting the system can correctly identify easy tasks but currently triggers such routing too infrequently.

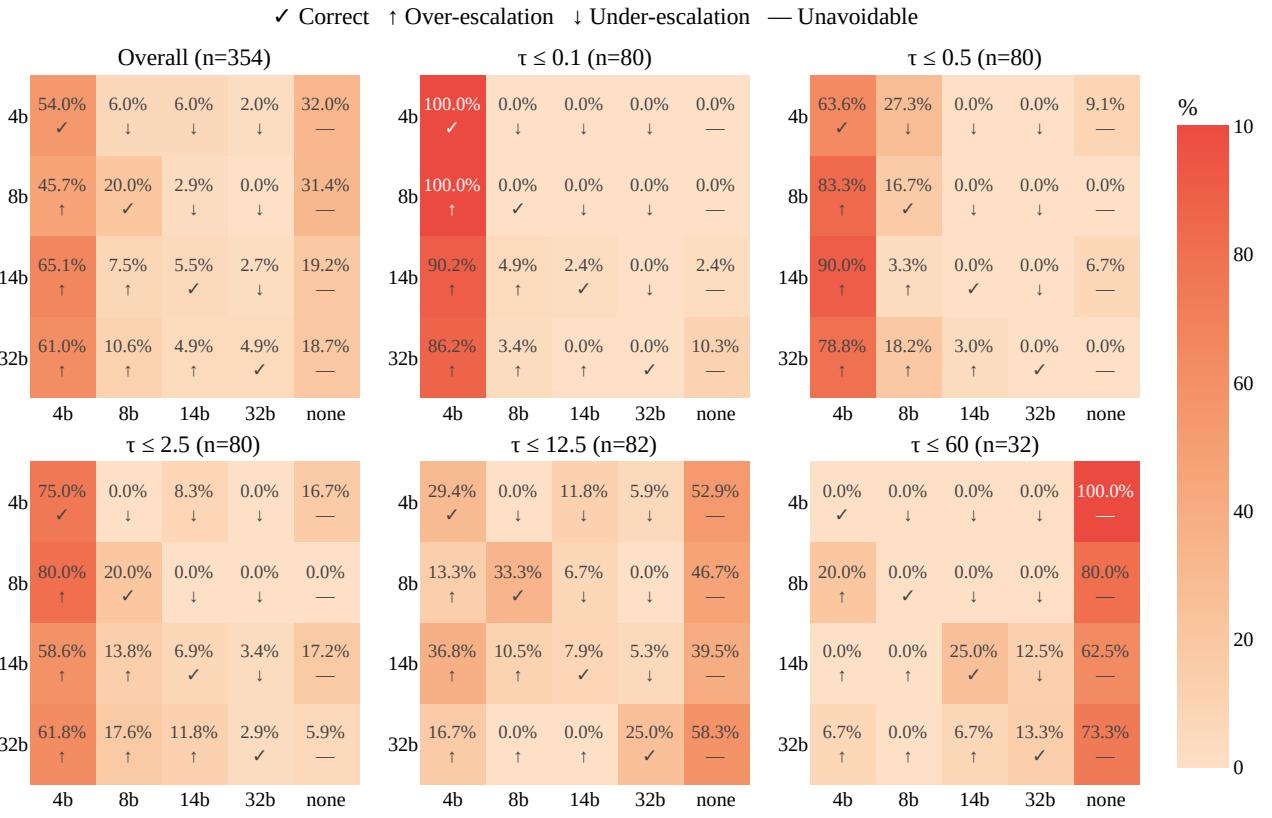


Figure 7 Deep search routing confusion matrices (SALE vs. oracle). Cells show row-normalized percentages.

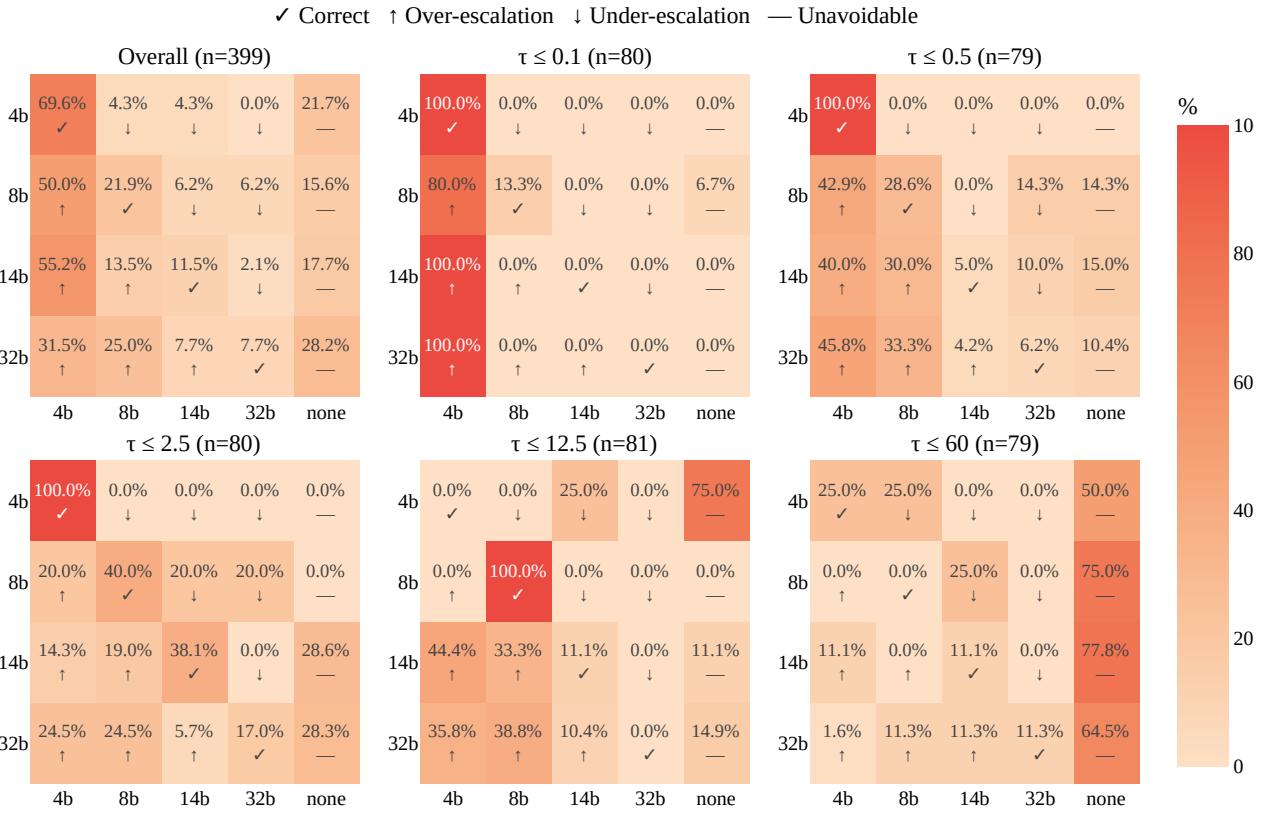


Figure 8 Coding routing confusion matrices (SALE vs. oracle). Cells show row-normalized percentages.

I Further Ablations

I.1 Cost–Value Function Ablations

Table 17 reports the results of ablating individual terms in the cost–value function. We separately remove the price per million tokens $\pi(a_i)$, the strategy length $|s_{t,i}|$, the normalized entropy $H(s_{t,i})$, and the entire jury term $\sum w_j \cdot \gamma(s_{t,i})$. For each ablated configuration, we re-optimize the remaining weights to best fit that specific setup.

Across both deep search and coding tasks, ablating any of the four terms leads to lower pass@1 scores, indicating that each component contributes meaningfully to SALE’s overall performance. For deep search, all ablations also result in higher average cost per million tokens, suggesting that the full cost–value function more effectively balances accuracy and efficiency. In contrast, for coding tasks, ablating $\pi(a_i)$ and $H(s_{t,i})$ yields slightly lower costs, reflecting an increased selection of smaller models, though this comes at the expense of accuracy. This pattern is consistent with the findings in Section 7.1, where we show that SALE already favors larger models for coding tasks even in its full form, whereas model selection is more varied for deep search.

The impact of each ablation also differs by task type and complexity. Removing the jury assessment causes the largest performance drop for deep search (-3.6 pass@1 on average), with the gap widening substantially on more complex tasks (-7.6 at $\tau \leq 12.5$; -6.3 at $\tau \leq 60$), underscoring the importance of jury-based scoring in challenging settings. For coding, SALE is more robust to jury removal; instead, ablating any of the cost terms (price and strategy length) results in the steepest decline, particularly at mid-complexity levels (-4.2 and -5.5 , respectively, at $\tau \leq 2.5$), reflecting their role in routing beyond cost alone. Entropy ablation presents a nuanced trade-off: at $\tau \leq 0.1$ for deep search, it achieves the lowest cost but sacrifices accuracy, suggesting entropy helps prevent under-spending on deceptively simple tasks. We note that a few ablated configurations marginally outperform SALE in isolated bins, though these gains are offset by larger losses in other bins.

Task type	$\tau(t)$	w/o price		w/o length		w/o entropy		w/o jury	
		Pass@1(\uparrow)	\$/Mt(\downarrow)						
Deep search	≤ 0.1	89.8 _{0.5}	0.22 _{0.01}	91.0 _{0.5}	0.28 _{0.00}	89.8 _{1.5}	0.19 _{0.00}	86.3 _{0.0}	0.28 _{0.00}
	≤ 0.5	88.5 _{0.9}	0.26 _{0.01}	88.8 _{1.0}	0.31 _{0.01}	85.5 _{1.7}	0.25 _{0.00}	89.0 _{1.7}	0.23 _{0.01}
	≤ 2.5	69.7 _{1.5}	0.25 _{0.01}	71.8 _{1.7}	0.30 _{0.01}	73.0 _{1.9}	0.26 _{0.01}	72.2 _{0.9}	0.21 _{0.00}
	≤ 12.5	33.7 _{2.3}	0.19 _{0.01}	33.4 _{1.8}	0.26 _{0.01}	35.1 _{0.9}	0.22 _{0.01}	29.5 _{0.5}	0.19 _{0.00}
	≤ 60	16.9 _{1.5}	0.25 _{0.01}	15.6 _{2.0}	0.29 _{0.01}	14.4 _{1.5}	0.25 _{0.01}	10.0 _{2.3}	0.20 _{0.01}
	All	65.4 _{0.7}	0.23 _{0.00}	66.0 _{0.6}	0.29 _{0.00}	65.5 _{0.7}	0.23 _{0.00}	63.7 _{0.5}	0.23 _{0.00}
Coding	≤ 0.1	98.3 _{1.0}	0.18 _{0.00}	98.8 _{0.0}	0.28 _{0.00}	97.2 _{0.5}	0.17 _{0.00}	96.8 _{0.6}	0.17 _{0.00}
	≤ 0.5	79.2 _{1.5}	0.28 _{0.00}	78.7 _{2.0}	0.29 _{0.00}	78.0 _{2.2}	0.28 _{0.01}	79.2 _{0.6}	0.30 _{0.00}
	≤ 2.5	64.8 _{2.2}	0.28 _{0.01}	63.5 _{1.2}	0.28 _{0.00}	64.8 _{2.5}	0.25 _{0.00}	67.5 _{1.9}	0.28 _{0.00}
	≤ 12.5	30.1 _{2.4}	0.27 _{0.00}	28.1 _{2.4}	0.29 _{0.01}	29.9 _{2.0}	0.29 _{0.00}	29.1 _{1.0}	0.29 _{0.00}
	≤ 60	23.8 _{0.5}	0.26 _{0.01}	24.8 _{1.6}	0.28 _{0.01}	25.6 _{3.0}	0.28 _{0.00}	27.3 _{0.6}	0.29 _{0.00}
	All	59.4 _{0.8}	0.26 _{0.00}	58.7 _{0.7}	0.28 _{0.00}	59.1 _{1.0}	0.26 _{0.00}	59.9 _{0.5}	0.27 _{0.00}

Table 17 Deep search and coding pass@1 and price per million tokens (\$/Mt) across task-complexity bins for SALE in its full form and under four ablation conditions. We remove the price term $\pi(a_i)$ (w/o price), the strategy length $|s_{t,i}|$ (w/o length), the entropy term $H(s_{t,i})$ (w/o entropy), and the jury scores $\sum w_j \cdot \gamma(s_{t,i})$ (w/o jury).

I.2 Jury Ablations

Table 18 reports results when the full jury (comprising the 4B, 8B, 14B, and 32B agents) is replaced by a single judge. Across both task types, all single-judge configurations underperform the full jury: for deep search, overall Pass@1 drops from 67.3 to 65.0–66.4, while for coding it falls from 61.1 to 59.0–60.5. Notably, no single judge is consistently superior—the 8B achieves the highest deep search accuracy (66.4), while the 14B leads on coding (60.5)—and larger judges do not reliably outperform smaller ones. Single judges also incur substantially higher costs for deep search (0.25–0.32 vs. 0.21 \$/Mt), suggesting the jury enables more efficient model selection. These results indicate that the jury’s strength lies not in any individual member but in the diversity of perspectives: combining judges of varying capacity produces a regularizing effect that yields more robust and cost-effective decisions than any single judge alone. Crucially, this ensemble incurs negligible overhead—each judge produces only a single token (a 1–5 discrete score) and we reuse the same agents already loaded in memory as part of the pool \mathcal{A} —making the jury’s benefits effectively free.

Given that individual judges underperform the full jury, we further ask whether each member is necessary by removing one judge at a time (Table 19). Indeed, every ablation leads to a decline in pass@1, confirming that each judge contributes unique signal. For coding, removing the 4B judge causes the largest drop (−2.9 pass@1), despite the 4B not being the strongest individual judge, highlighting its complementary role within the ensemble. For deep search, removing the 14B has the smallest impact (66.9 vs. 67.3), though performance still decreases. Interestingly, removing the 4B also substantially increases deep search cost (0.28 vs. 0.21 \$/Mt), suggesting the smaller judge helps steer the system toward more economical selections. Overall, these results reinforce that jury diversity is not redundant: even the smallest judge provides information that improves both accuracy and efficiency.

Finally, we ablate agent self-judgment (retaining only peer-judgment) and peer-judgment (retaining only self-judgment) from SALE (Table 20). Without self-judgment, the system approaches and occasionally slightly exceeds the best single agent baseline (64.1 vs. 63.8 for deep search; 58.8 vs. 58.4 for coding), though it remains below the full SALE (67.3 and 61.1, respectively). In contrast, removing peer-judgment causes a larger drop, especially for coding, where overall pass@1 falls to 48.7. Notably, this configuration yields markedly lower costs (0.07 \$/Mt for deep search vs. 0.21), suggesting that smaller agents tend to be more confident self-scorers, leading the system to favor them when peer signals are unavailable. At low task complexity, relying solely on self-judgment still matches or even slightly exceeds the best single agent (e.g., 97.5 vs. 95.0 at $\tau \leq 0.1$ for coding); yet as complexity increases, performance degrades sharply (5.6 vs. 12.5 at $\tau \leq 60$ for deep search), indicating that self-judgment alone does not scale to harder tasks. These results underscore the complementary nature of both feedback types: peer-judgment provides the external calibration necessary for difficult problems, while self-judgment contributes efficient, low-cost signal on simpler ones.

Task type	$\tau(t)$	4B judge only		8B judge only		14B judge only		32B judge only	
		Pass@1(↑)	\$/Mt(↓)	Pass@1(↑)	\$/Mt(↓)	Pass@1(↑)	\$/Mt(↓)	Pass@1(↑)	\$/Mt(↓)
Deep search	≤ 0.1	91.3 _{0.0}	0.25 _{0.01}	90.0 _{0.0}	0.30 _{0.00}	89.0 _{0.5}	0.29 _{0.00}	90.0 _{0.0}	0.28 _{0.00}
	≤ 0.5	88.0 _{0.6}	0.28 _{0.01}	88.8 _{0.0}	0.30 _{0.01}	86.3 _{0.6}	0.33 _{0.01}	88.0 _{0.6}	0.32 _{0.01}
	≤ 2.5	71.0 _{1.5}	0.26 _{0.01}	72.5 _{0.8}	0.34 _{0.01}	69.2 _{0.6}	0.35 _{0.01}	69.2 _{1.0}	0.32 _{0.01}
	≤ 12.5	33.4 _{0.6}	0.22 _{0.01}	35.4 _{0.0}	0.33 _{0.00}	36.1 _{1.0}	0.31 _{0.01}	34.1 _{2.2}	0.31 _{0.01}
	≤ 60	15.6 _{0.0}	0.25 _{0.01}	15.6 _{0.0}	0.31 _{0.01}	15.6 _{0.0}	0.31 _{0.01}	14.4 _{1.5}	0.29 _{0.01}
	All	65.7 _{0.4}	0.25 _{0.00}	66.4 _{0.2}	0.32 _{0.00}	65.0 _{0.3}	0.32 _{0.00}	65.1 _{0.6}	0.31 _{0.00}
Coding	≤ 0.1	96.8 _{0.6}	0.18 _{0.00}	97.5 _{0.0}	0.18 _{0.00}	98.5 _{0.5}	0.20 _{0.01}	97.8 _{0.5}	0.19 _{0.00}
	≤ 0.5	80.3 _{0.6}	0.29 _{0.00}	80.3 _{1.0}	0.28 _{0.00}	80.5 _{0.6}	0.30 _{0.00}	81.5 _{0.6}	0.29 _{0.00}
	≤ 2.5	66.8 _{1.0}	0.29 _{0.00}	67.0 _{2.8}	0.28 _{0.00}	68.2 _{1.0}	0.29 _{0.00}	66.5 _{1.5}	0.29 _{0.00}
	≤ 12.5	27.2 _{1.4}	0.27 _{0.00}	29.1 _{1.7}	0.30 _{0.01}	29.4 _{1.2}	0.31 _{0.00}	28.4 _{1.6}	0.30 _{0.02}
	≤ 60	24.3 _{0.9}	0.28 _{0.00}	25.3 _{1.6}	0.28 _{0.01}	26.1 _{1.7}	0.29 _{0.01}	25.8 _{2.2}	0.29 _{0.00}
	All	59.0 _{0.4}	0.26 _{0.00}	59.8 _{0.8}	0.26 _{0.00}	60.5 _{0.5}	0.28 _{0.00}	60.0 _{0.6}	0.27 _{0.00}

Table 18 Deep search and coding pass@1 and price per million tokens (\$/Mt) across task-complexity bins for SALE with the agent jury replaced by one single agent judge.

Task type	$\tau(t)$	w/o 4B judge		w/o 8B judge		w/o 14B judge		w/o 32B judge	
		Pass@1(\uparrow)	\$/Mt(\downarrow)						
Deep search	≤ 0.1	89.2 _{0.6}	0.20 _{0.00}	92.3 _{0.5}	0.23 _{0.00}	91.8 _{1.0}	0.22 _{0.01}	90.0 _{0.0}	0.22 _{0.01}
	≤ 0.5	86.0 _{0.9}	0.28 _{0.00}	86.0 _{0.9}	0.25 _{0.01}	88.8 _{1.1}	0.22 _{0.01}	87.0 _{1.0}	0.24 _{0.01}
	≤ 2.5	71.2 _{0.8}	0.35 _{0.00}	68.2 _{1.5}	0.24 _{0.01}	73.0 _{2.0}	0.23 _{0.01}	72.5 _{1.1}	0.23 _{0.01}
	≤ 12.5	36.1 _{1.0}	0.31 _{0.01}	34.9 _{1.8}	0.18 _{0.01}	36.3 _{1.8}	0.18 _{0.00}	33.9 _{2.4}	0.17 _{0.00}
	≤ 60	14.4 _{1.5}	0.28 _{0.01}	16.9 _{1.5}	0.23 _{0.01}	13.8 _{1.7}	0.23 _{0.02}	16.3 _{1.3}	0.23 _{0.01}
	All	65.3 _{0.4}	0.28 _{0.00}	65.3 _{0.6}	0.23 _{0.00}	66.9 _{0.7}	0.21 _{0.00}	65.7 _{0.7}	0.22 _{0.00}
Coding	≤ 0.1	97.0 _{0.6}	0.16 _{0.00}	97.8 _{1.2}	0.18 _{0.00}	97.0 _{0.6}	0.18 _{0.00}	97.8 _{1.2}	0.18 _{0.00}
	≤ 0.5	75.7 _{2.6}	0.25 _{0.01}	79.0 _{1.3}	0.27 _{0.00}	79.5 _{1.5}	0.27 _{0.01}	77.7 _{1.5}	0.27 _{0.01}
	≤ 2.5	62.8 _{1.8}	0.22 _{0.01}	65.8 _{1.3}	0.28 _{0.00}	65.0 _{2.2}	0.26 _{0.00}	66.0 _{2.2}	0.27 _{0.00}
	≤ 12.5	29.9 _{0.9}	0.30 _{0.00}	29.4 _{1.4}	0.31 _{0.01}	29.6 _{1.6}	0.30 _{0.01}	28.9 _{1.7}	0.30 _{0.00}
	≤ 60	25.8 _{1.5}	0.28 _{0.00}	25.6 _{1.2}	0.28 _{0.00}	25.1 _{1.2}	0.28 _{0.00}	24.6 _{1.0}	0.27 _{0.00}
	All	58.2 _{0.7}	0.24 _{0.00}	59.5 _{0.6}	0.26 _{0.00}	59.2 _{0.7}	0.26 _{0.00}	59.0 _{0.7}	0.26 _{0.00}

Table 19 Deep search and coding pass@1 and price per million tokens (\$/Mt) across task-complexity bins for SALE with one judge removed from the jury in turn.

Task type	$\tau(t)$	w/o self-feedback		w/o peer-feedback	
		Pass@1(\uparrow)	\$/Mt(\downarrow)	Pass@1(\uparrow)	\$/Mt(\downarrow)
Deep search	≤ 0.1	92.3 _{0.5}	0.26 _{0.01}	91.0 _{0.9}	0.06 _{0.01}
	≤ 0.5	86.0 _{0.9}	0.21 _{0.01}	87.3 _{1.8}	0.07 _{0.01}
	≤ 2.5	68.2 _{0.6}	0.22 _{0.01}	65.8 _{2.0}	0.07 _{0.00}
	≤ 12.5	31.5 _{2.9}	0.17 _{0.01}	32.7 _{1.6}	0.09 _{0.01}
	≤ 60	12.5 _{1.9}	0.21 _{0.02}	5.6 _{1.3}	0.06 _{0.01}
	All	64.1 _{0.7}	0.21 _{0.00}	63.2 _{0.7}	0.07 _{0.00}
Coding	≤ 0.1	97.5 _{0.8}	0.17 _{0.00}	97.5 _{0.0}	0.05 _{0.01}
	≤ 0.5	81.0 _{1.1}	0.27 _{0.01}	71.9 _{1.9}	0.05 _{0.00}
	≤ 2.5	65.3 _{2.9}	0.27 _{0.01}	49.0 _{1.7}	0.06 _{0.00}
	≤ 12.5	26.8 _{0.6}	0.29 _{0.01}	15.1 _{1.5}	0.08 _{0.00}
	≤ 60	23.8 _{0.5}	0.29 _{0.01}	10.4 _{1.8}	0.12 _{0.01}
	All	58.8 _{0.7}	0.26 _{0.00}	48.7 _{0.7}	0.07 _{0.00}

Table 20 Deep search and coding pass@1 and price per million tokens (\$/Mt) across task-complexity bins for SALE with ablated self-feedback and peer-feedback.

J Agent Selection Over Time

In Figure 9, we show the cumulative selection rate over time, defined as the running fraction of tasks ultimately delegated to each agent, for all agents that can refine their strategies during SALE’s auction mechanism (4B, 8B, and 32B). Recall that an agent only enters the memory-based self-refinement stage when a more expensive agent is the provisional winner, so cheaper agents have more opportunities to refine their strategies than more expensive ones. As a result, the 4B agent generates a refined strategy on 92% of samples for deep search and 93% for coding, the 8B agent on 91% for deep search and 92% for coding, and the 14B agent on 62% for deep search and 77% for coding, leading to different patterns in how their cumulative selection rates evolve as auction memory accumulates.

Across both domains, the 4B agent’s cumulative selection share (also reported in Figure 5) shows a clear upward trend, indicating that it is increasingly chosen as auction memory grows. The 8B agent’s share is also broadly increasing for deep search, albeit with higher variance, but remains relatively flat for coding. In contrast, the 14B agent’s cumulative share exhibits a mild downward drift in both domains, consistent with workload being gradually reallocated toward the cheaper agents. Because each curve at time step t reflects the average selection rate computed over all tasks up to t , the trajectories are naturally more volatile at the beginning, when this average is based on few samples, and become smoother as more tasks accumulate.

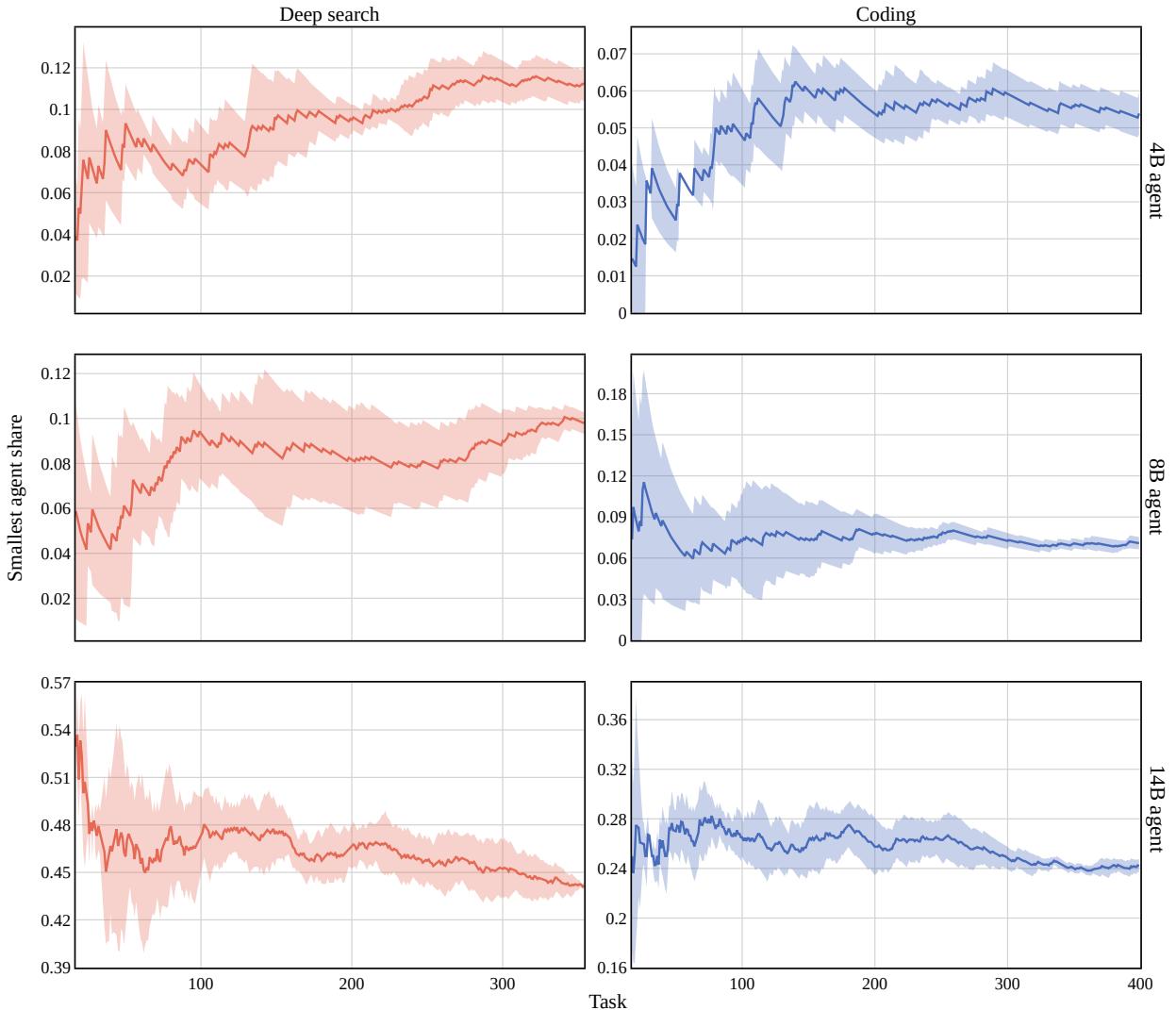


Figure 9 Cumulative selection rates over time for the 4B, 8B, and 14B agents on deep search and coding tasks, showing increasing delegation to cheaper agents while the 14B share gradually declines as auction memory accumulates.

K Qualitative Performance Comparison

This section presents examples, spanning task complexity levels, where the largest agent (32B) fails and a smaller agent succeeds. As these examples originate from the test set, we omit the agents' final answers. Given that full traces span tens of thousands to millions of tokens, we present only the agent's solution strategy and indicate whether the final answer is correct. In many cases, the strategy alone exhibits telltale signs of failure: misunderstanding of the task, excessive complexity, or hallucinated details that ultimately lead to an incorrect answer.

K.1 Performance Comparison on Deep Search Tasks

For deep search tasks, we identify three recurring failure modes in the 32B agent. First, the agent occasionally bypasses tool use entirely and provides an answer from parametric memory, often incorrect. This tendency is sometimes visible in the initial strategy itself, where the eventual answer is already present despite a search plan having been outlined (Example 1). Second, the agent may over-engineer its approach to straightforward questions, pursuing tangential information—such as a musician's date of death (Example 2)—or issuing excessive sequential queries that lead it further from the answer (Example 5). Third, the agent retrieves partial or related information from parametric knowledge that interferes with the search process: in Example 3, a legal case is conflated with an unrelated one, while in Example 4, parametric recall of an intermediate fact introduces ambiguity between a television series and its sequel.

Deep Search Example 1

Task

A patient suffers a stroke just in the tip of the paracentral artery on the left side where the artery rounds over to the lateral surface. This would likely result in

Answer Choices:

- A. More sensory loss in the right arm than the foot
- B. More sensory loss in the right foot than the arm
- C. More sensory loss in the left arm than the foot
- D. More sensory loss in the left foot than the arm
- E. More weakness of the right foot than the arm

Answer with the letter of the single correct option.

Incorrect 32B response

Plan:

ask_search_agent: What is the anatomical territory of the paracentral artery and its role in sensory/motor functions? Provide a summary of the brain regions it supplies and their functional roles.

ask_search_agent: How does a stroke in the left paracentral artery affect sensory/motor functions contralaterally? Provide a summary of the expected deficits on the right side of the body.

ask_search_agent: In the sensory homunculus, is the foot or arm more represented in the medial portion of the paracentral lobule? Provide a concise answer based on neuroanatomy resources.

final_answer: B

Final answer: 

Correct 14B response

Plan:

ask_search_agent: What is the somatotopic representation of the arm and foot in the paracentral lobule of the somatosensory cortex? Provide a concise summary of the anatomical distribution. ask_search_agent: Does a stroke in the left paracentral lobule cause contralateral sensory loss in the right arm or foot? Provide a concise summary of the clinical implications.

Final answer: 

Deep Search Example 2

Task

How many studio albums were published by Mercedes Sosa between 2000 and 2009 (included)? You can use the latest version of english wikipedia.

Incorrect 32B response

Plan:

1. Use `ask_search_agent` to retrieve Mercedes Sosa's discography (studio albums with release years) from the English Wikipedia page. Specify the query: "List all studio albums by Mercedes Sosa with their release years. Provide a table of album titles and years." Website: https://en.wikipedia.org/wiki/Mercedes_Sosa.
2. Use `ask_search_agent` to confirm Mercedes Sosa's death date and check for posthumous album releases in 2009. Query: "When did Mercedes Sosa die? Were any studio albums released posthumously in 2009? Provide dates and album names."
3. Filter the discography data to count studio albums released between 2000–2009 (inclusive), excluding posthumous 2009 releases.
4. Use `final_answer` to submit the verified count.

Final answer:

Correct 8B response

Plan:

1. Use `ask_search_agent` to retrieve the list of studio albums by Mercedes Sosa along with their release years from the English Wikipedia page (https://en.wikipedia.org/wiki/Mercedes_Sosa). Provide `output_content` as a structured list of albums with release years.
2. Extract the release years of all studio albums from the search result.
3. Filter the albums to include only those released between 2000 and 2009 (inclusive).
4. Count the number of filtered albums.
5. Use `final_answer` to provide the derived count.

Final answer:

Deep Search Example 3

Task

What amount was the settlement that the character from the Son of al Qaeda got in 2017?

Incorrect 32B response

Plan:

1. Use `ask_search_agent` to verify if "Son of al Qaeda" is a real entity or a misstatement (e.g., "Son of Sam"). Query: "Is 'Son of al Qaeda' a real person or legal case? If not, what is the correct reference (e.g., 'Son of Sam')? Provide clarification."
2. If the search confirms a misstatement (e.g., "Son of Sam"), use `ask_search_agent` to check for 2017 settlements: "What legal settlement amount was associated with the 'Son of Sam' case in 2017? Provide the numerical value."
3. If no 2017 data is found, cross-reference historical records for the original "Son of Sam" case (1979) and determine if updates or subsequent settlements occurred in 2017.
4. Use `final_answer` to report the verified settlement amount.

Final answer:

Correct 8B response

Plan:

1. Use `ask_search_agent` to identify what "Son of al Qaeda" refers to (e.g., a real person, fictional character, or legal case). Query: "What is 'Son of al Qaeda'? Provide `output_content`".
2. Use `ask_search_agent` to determine the specific settlement amount related to the identified entity in 2017. Query: "What was the 2017 settlement amount related to [entity from Step 1]? Provide `output_content`".
3. Verify the settlement amount via credible sources (e.g., court records, news articles) using `ask_search_agent` if needed.
4. Use `final_answer` to state the confirmed settlement amount.

Final answer:

Deep Search Example 4

Task

What season was the character introduced that becomes the main antagonist in the following season, from the animated television series created by Bryan Konietzko and Michael Dante DiMartino as a sequel to "...", which aired from 2005 to 2008?

Incorrect 32B response

Plan:

1. Use `ask_search_agent` to confirm the name of the sequel series to "Avatar: The Last Airbender" created by Bryan Konietzko and Michael Dante DiMartino. Provide the name of the sequel.
2. Use `ask_search_agent` to identify characters introduced in each season of the sequel series who later become main antagonists in subsequent seasons. Provide a list of characters with their introduction season and antagonist season.
3. Use `ask_search_agent` to verify the specific season number where the character was introduced that becomes the main antagonist in the following season. Provide the season number.

Final answer: 

Correct 8B response

Plan:

1. Use `ask_search_agent` to identify the original series (2005-2008) created by Bryan Konietzko and Michael Dante DiMartino, and confirm the sequel series name. Query: "What is the name of the animated series created by Bryan Konietzko and Michael Dante DiMartino that aired from 2005 to 2008? Provide the name."
2. Use `ask_search_agent` to determine the main antagonists for each season of the sequel series. Query: "What are the main antagonists for each season of [sequel series name]? Provide a list of seasons and corresponding antagonists."
3. Use `ask_search_agent` to identify which antagonist was introduced in a season and became the main antagonist in the following season. Query: "In which season was a character introduced in [sequel series name] that became the main antagonist in the next season? Provide the season number."
4. Use `final_answer` to compile the result from step 3.

Final answer: 

Deep Search Example 5

Task

D1NZ is a series based on what oversteering technique?

Incorrect 32B response

Plan:

- `ask_search_agent`: "What is the D1NZ series and which oversteering technique is it based on? Provide a concise explanation of the series and the specific oversteering technique it employs."
- `ask_search_agent`: "What oversteering techniques are commonly used in drifting competitions like D1NZ? Provide a list of techniques and their association with specific drifting series or events."
- `ask_search_agent`: "What vehicle specifications or rules define the D1NZ drifting series? Provide details on car configurations that indicate the required oversteering technique."
- `final_answer`: <answer derived from cross-referencing D1NZ's vehicle specs, competition style, and associated oversteering techniques>

Final answer: 

Correct 4B response

Plan:

1. Use `ask_search_agent` to identify the oversteering technique associated with the series D1NZ. Query: "What oversteering technique is the series D1NZ based on? Provide the name of the technique."
2. Use `final_answer` to submit the identified oversteering technique as the answer.

Final answer: 

K.2 Performance Comparison on Coding Tasks

For coding tasks, we observe analogous failure modes in the 32B agent. First, the agent over-engineers straightforward problems: in Example 6, it plans a recursive solution for what is explicitly fixed two-level nesting, whereas the 4B correctly identifies the specific structure. Such unnecessary complexity increases the surface area for errors. Second, the agent draws on parametric knowledge at the expense of task constraints: Example 7 shows the 32B planning to import a library function not in the available import list, while the 14B opts for a simpler, more robust approach. This overconfidence extends to skipping the planning phase entirely: in Examples 8 and 9, the 32B produces code directly in place of a high-level strategy, without using the Python shell for verification and pursuing over-optimized solutions that introduce subtle errors—ignoring intermediate computation steps or applying mathematically incorrect shortcuts—where simpler approaches would have fewer failure points. Third, the agent neglects edge cases: in Example 10, the 32B fails to handle the "no duplicate found" case, resulting in an incorrect return type that breaks hidden tests.

Coding Example 6

Task

You are an expert Python programmer, and here is your task: Write a function to convert a tuple of string values to a tuple of integer values. Your code should pass the following test: `assert tuple_int_str((('333', '33'), ('1416', '55'))) == ((333, 33), (1416, 55))`

Incorrect 32B response

Plan:

1. Analyze the task: Create a function that converts nested tuples of strings to integers.
2. Implement a solution using nested tuple comprehensions to recursively convert each string element to an integer.
3. Validate the solution against the provided test case to ensure correctness.

Final answer: 

Correct 4B response

Plan:

The plan is to create a function that converts each string in the input tuple of tuples to an integer, preserving the nested structure. This is achieved by using a generator expression within a `tuple()` to iterate over each inner tuple, convert each string to an integer, and collect the results into a new tuple of tuples.

Final answer: 

Coding Example 7

Task

You are an expert Python programmer, and here is your task: Write a function to merge multiple sorted inputs into a single sorted iterator using heap queue algorithm. Your code should pass the following tests: `assert merge_sorted_list([25, 24, 15, 4, 5, 29, 110],[19, 20, 11, 56, 25, 233, 154],[24, 26, 54, 48])==[4, 5, 11, 15, 19, 20, 24, 24, 25, 25, 26, 29, 48, 54, 56, 110, 154, 233]`

Incorrect 32B response

Plan:

1. Import the `heapq` module to utilize the `merge` function for efficiently combining sorted iterables.
2. Define a function `merge_sorted_list` that accepts multiple lists as input.
3. For each input list, sort it individually to ensure they are in ascending order.
4. Use `heapq.merge` to merge the sorted lists into a single sorted iterator.
5. Convert the merged iterator to a list and return it as the final result.

Final answer:

Correct 14B response

Plan:

1. Define the function `merge_sorted_list` to accept multiple lists as input.
2. Sort each input list individually to ensure they are in ascending order.
3. Initialize a min-heap to track the smallest elements from each sorted list, storing tuples of (value, list index, element index).
4. While the heap is not empty, extract the smallest element, append it to the result, and push the next element from the same list into the heap if available.
5. Return the merged sorted list as the final result.

Final answer:

Coding Example 8

Task

You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program that matches the specification and passes all tests.

Question:

You are given an array of integers `nums` of size `n` and a positive integer `threshold`.

There is a graph consisting of `n` nodes with the `i`th node having a value of `nums[i]`. Two nodes `i` and `j` in the graph are connected via an undirected edge if `lcm(nums[i], nums[j]) <= threshold`.

Return the number of connected components in this graph.

A connected component is a subgraph of a graph in which there exists a path between any two vertices, and no vertex of the subgraph shares an edge with a vertex outside of the subgraph.

The term `lcm(a, b)` denotes the least common multiple of `a` and `b`.

Example 1:

Input: `nums = [2,4,8,3,9]`, `threshold = 5`

Output: 4

Explanation:

The four connected components are (2, 4), (3), (8), (9).

Example 2:

Input: `nums = [2,4,8,3,9,12]`, `threshold = 10`

Output: 2

Explanation:

The two connected components are (2, 3, 4, 8, 9), and (12).

Constraints:

```
1 <= nums.length <= 105
1 <= nums[i] <= 109
```

All elements of nums are unique.

$1 \leq \text{threshold} \leq 2 * 10^5$

Format: You will use the following starter code to write the solution to the problem and enclose your code within delimiters.

```
```python final_answer"""
class Solution:
 def countComponents(self, nums: List[int], threshold: int) -> int:
"""```

```

### Answer: (use the provided format with backticks)

### Incorrect 32B response

#### Plan:

To solve this problem, we need to determine the number of connected components in a graph where nodes are connected if the least common multiple (LCM) of their corresponding values is less than or equal to a given threshold. The solution involves efficiently using Union-Find (Disjoint Set Union, DSU) to group connected nodes.

### Approach

1. \*\*Filter and Sort\*\*: First, filter out elements from the input array that are greater than the threshold and sort the remaining elements.
2. \*\*Union-Find Data Structure\*\*: Use a Union-Find (DSU) structure to manage the connected components.
3. \*\*Process Multiples\*\*: For each number in the filtered list, connect it with all its multiples that are also in the list and within the threshold.
4. \*\*Process Pairs with LCM Check\*\*: For each number, check pairs where the LCM is within the threshold and connect them if they meet the condition.
5. \*\*Count Components\*\*: After processing all connections, count the number of unique connected components using the DSU structure and add components from elements that were filtered out due to exceeding the threshold.

### Solution Code

```
```python
import bisect
from math import gcd
from typing import List

class Solution:
    def countComponents(self, nums: List[int], threshold: int) -> int:
        # Filter and sort the numbers that are <= threshold
        filtered = [x for x in nums if x <= threshold]
        filtered.sort()
        n = len(filtered)
        if n == 0:
            return len(nums)

        # Map each value to its index in the filtered list
        value_to_index = {v: i for i, v in enumerate(filtered)}

        # Define DSU (Disjoint Set Union) structure
        class DSU:
            def __init__(self, size):
                self.parent = list(range(size))
                self.rank = [0] * size

            def find(self, x):
                if self.parent[x] != x:
                    self.parent[x] = self.find(self.parent[x])
                return self.parent[x]

            def union(self, x, y):
                x_root = self.find(x)
                y_root = self.find(y)
                if x_root == y_root:

```

```

        return
    if self.rank[x_root] < self.rank[y_root]:
        self.parent[x_root] = y_root
    else:
        self.parent[y_root] = x_root
        if self.rank[x_root] == self.rank[y_root]:
            self.rank[x_root] += 1

dsu = DSU(n)

# Process each element in the filtered list
for i in range(n):
    x = filtered[i]
    # Connect with multiples of x
    m = 2 * x
    while m <= threshold:
        if m in value_to_index:
            idx = value_to_index[m]
            dsu.union(i, idx)
        m += x
    # Connect with other numbers where LCM(x, y) <= threshold
    upper = threshold // x
    if upper >= x:
        left = i + 1
        right_idx = bisect.bisect_right(filtered, upper) - 1
        if left <= right_idx:
            for j in range(left, right_idx + 1):
                y = filtered[j]
                if (x * y) // gcd(x, y) <= threshold:
                    dsu.union(i, j)

    # Count the number of unique roots in DSU
    roots = set()
    for i in range(n):
        roots.add(dsu.find(i))

    # Total components is the number of roots in DSU plus elements > threshold
    return len(roots) + (len(nums) - n)
```

```

### Explanation

- \*\*Filtering and Sorting\*\*: The input array is filtered to include only values that are less than or equal to the threshold. This helps in reducing unnecessary computations.
- \*\*Union-Find Structure\*\*: The DSU structure is used to efficiently manage and merge connected components.
- \*\*Multiples Processing\*\*: For each number, we connect it with all its multiples within the threshold. This ensures that all direct connections based on multiples are considered.
- \*\*LCM Check for Pairs\*\*: For each number, we check pairs where the LCM is within the threshold. This ensures that even non-multiples that meet the LCM condition are connected.
- \*\*Component Counting\*\*: After processing all connections, the number of unique roots in the DSU structure gives the number of connected components for the filtered values. Components from values exceeding the threshold are added separately.

This approach efficiently handles the constraints and ensures that all valid connections are considered, leading to the correct count of connected components.

**Final answer:** ✘

#### Correct 14B response

##### Plan:

- \*\*Sort the input array NUMS\*\* to facilitate efficient processing of multiples.
- \*\*Create a mapping from values to their indices\*\* to quickly check if a number exists in the array.
- \*\*Use a Disjoint Set Union (DSU) structure\*\* to manage connected components.
- \*\*For each number in NUMS\*\*, iterate through its multiples up to the threshold and connect the number with its multiples if they exist in the array.
- \*\*Count the number of unique parents\*\* in the DSU to determine the number of connected components.
- \*\*Return the count of connected components\*\* as the result.

**Final answer:** ✓

---

---

## Coding Example 9

---

### Task

You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program that matches the specification and passes all tests.

### Question:

You are given a string `initialCurrency`, and you start with 1.0 of `initialCurrency`.

You are also given four arrays with currency pairs (strings) and rates (real numbers):

`pairs1[i] = [startCurrencyi, targetCurrencyi]` denotes that you can convert from `startCurrencyi` to `targetCurrencyi` at a rate of `rates1[i]` on day 1.

`pairs2[i] = [startCurrencyi, targetCurrencyi]` denotes that you can convert from `startCurrencyi` to `targetCurrencyi` at a rate of `rates2[i]` on day 2.

Also, each `targetCurrency` can be converted back to its corresponding `startCurrency` at a rate of `1 / rate`.

You can perform any number of conversions, including zero, using `rates1` on day 1, followed by any number of additional conversions, including zero, using `rates2` on day 2.

Return the maximum amount of `initialCurrency` you can have after performing any number of conversions on both days in order.

Note: Conversion rates are valid, and there will be no contradictions in the rates for either day. The rates for the days are independent of each other.

Example 1:

Input: `initialCurrency = "EUR", pairs1 = [["EUR","USD"],["USD","JPY"]], rates1 = [2.0,3.0], pairs2 = [["JPY","USD"],["USD","CHF"],["CHF","EUR"]], rates2 = [4.0,5.0,6.0]`

Output: 720.00000

Explanation:

To get the maximum amount of EUR, starting with 1.0 EUR:

On Day 1:

Convert EUR to USD to get 2.0 USD.

Convert USD to JPY to get 6.0 JPY.

On Day 2:

Convert JPY to USD to get 24.0 USD.

Convert USD to CHF to get 120.0 CHF.

Finally, convert CHF to EUR to get 720.0 EUR.

Example 2:

Input: `initialCurrency = "NGN", pairs1 = [["NGN","EUR"]], rates1 = [9.0], pairs2 = [["NGN","EUR"]], rates2 = [6.0]`

Output: 1.50000

Explanation:

Converting NGN to EUR on day 1 and EUR to NGN using the inverse rate on day 2 gives the maximum amount.

Example 3:

Input: `initialCurrency = "USD", pairs1 = [["USD","EUR"]], rates1 = [1.0], pairs2 = [["EUR","JPY"]], rates2 = [10.0]`

Output: 1.00000

Explanation:

In this example, there is no need to make any conversions on either day.

Constraints:

`1 <= initialCurrency.length <= 3`

`initialCurrency` consists only of uppercase English letters.

`1 <= n == pairs1.length <= 10`

`1 <= m == pairs2.length <= 10`

`pairs1[i] == [startCurrencyi, targetCurrencyi]`

```

pairs2[i] == [startCurrencyi, targetCurrencyi]
1 <= startCurrencyi.length, targetCurrencyi.length <= 3
startCurrencyi and targetCurrencyi consist only of uppercase English letters.
rates1.length == n
rates2.length == m
1.0 <= rates1[i], rates2[i] <= 10.0
The input is generated such that there are no contradictions or cycles in the conversion graphs for either day.
The input is generated such that the output is at most 5 * 1010.

```

```

Format: You will use the following starter code to write the solution to the problem and enclose your code
within delimiters.
```python final_answer"""
class Solution:
    def maxAmount(self, initialCurrency: str, pairs1: List[List[str]], rates1: List[float], pairs2: List[List[str]], rates2:
List[float]) -> float:
"""
```

```

#### Answer: (use the provided format with backticks)

### Incorrect 32B response

#### Plan:

```

```python
class Solution:
    def maxAmount(self, initialCurrency: str, pairs1: List[List[str]], rates1: List[float], pairs2: List[List[str]], rates2:
List[float]) -> float:
        def bellman_ford(start, edges, all_currencies):
            max_amounts = currency: -float('inf') for currency in all_currencies
            max_amounts[start] = 1.0
            V = len(all_currencies)
            for _ in range(V - 1):
                updated = False
                for (u, v, rate) in edges:
                    if max_amounts[u] != -float('inf'):
                        if max_amounts[u] * rate > max_amounts[v]:
                            max_amounts[v] = max_amounts[u] * rate
                            updated = True
                if not updated:
                    break
            return max_amounts

        # Process day1
        edges1 = []
        for i in range(len(pairs1)):
            a, b = pairs1[i]
            rate = rates1[i]
            edges1.append((a, b, rate))
            edges1.append((b, a, 1.0 / rate))
        currencies_day1 = set()
        for a, b in pairs1:
            currencies_day1.add(a)
            currencies_day1.add(b)
        currencies_day1.add(initialCurrency)
        day1_max = bellman_ford(initialCurrency, edges1, currencies_day1)

        # Process day2
        edges2 = []
        for i in range(len(pairs2)):
            a, b = pairs2[i]
            rate = rates2[i]
            edges2.append((a, b, rate))
            edges2.append((b, a, 1.0 / rate))
        currencies_day2 = set()
        for a, b in pairs2:
            currencies_day2.add(a)
            currencies_day2.add(b)

```

```

max_result = 1.0 # initial amount

for c in day1_max:
    amount_c = day1_max[c]
    if amount_c == -float('inf'):
        continue
    all_currencies_day2_run = currencies_day2.copy()
    all_currencies_day2_run.add(c)
    day2_max = bellman_ford(c, edges2, all_currencies_day2_run)
    current_value = day2_max.get(initialCurrency, -float('inf'))
    if current_value > max_result:
        max_result = current_value

return max_result```

```

Final answer:

Correct 8B response

Plan:

To solve the problem of maximizing the amount of the initial currency after performing conversions on two days, we need to model the conversion process as a graph traversal problem. The key steps involve:

1. **Building the Conversion Graphs**: For each day, construct a graph where nodes represent currencies and edges represent conversion rates (both forward and reverse).
 2. **Topological Sorting**: Since the input guarantees no cycles, we can perform a topological sort on the graph to process nodes in an order that ensures all dependencies are resolved.
 3. **Maximizing Currency Amounts**: Using dynamic programming on the topologically sorted nodes, compute the maximum amount of each currency achievable after each day's conversions.
 4. **Combining Results**: After processing both days, compute the maximum amount of the initial currency by considering all possible paths from the currencies obtained after day 2 back to the initial currency.
-

Step-by-Step Plan

1. **Build the Graph for Day 1**:
- Construct a graph where each currency pair from `pairs1` and `rates1` is represented as bidirectional edges (forward and reverse conversions).
 2. **Compute Day 1 Amounts**:
- Starting with 1.0 of the initial currency, use topological sorting to compute the maximum amount of each currency achievable after day 1 conversions.
 3. **Build the Graph for Day 2**:
- Construct a similar graph using `pairs2` and `rates2`.
 4. **Compute Day 2 Amounts**:
- Use the results from day 1 as the starting point for day 2 conversions. Again, use topological sorting to compute the maximum amount of each currency after day 2.
 5. **Convert Back to Initial Currency**:
- For each currency obtained after day 2, compute the maximum amount of the initial currency that can be obtained by converting through day 2's graph.
 6. **Return the Maximum Result**:
- The final answer is the maximum value among all possible conversions back to the initial currency.
-

Final Plan

1. **Build the day1 graph** using `pairs1` and `rates1`, including both forward and reverse conversions.
2. **Compute day1_amounts** by traversing the graph in topological order, starting from the initial currency.
3. **Build the day2 graph** using `pairs2` and `rates2`, including both forward and reverse conversions.
4. **Compute day2_amounts** by traversing the day2 graph in topological order, starting from the day1_amounts.
5. **For each currency in day2_amounts**, compute the maximum amount of the initial currency by traversing the day2 graph again, starting from that currency.
6. **Return the maximum value** among all computed results.

Final answer:

Coding Example 10

Task

You are an expert Python programmer, and here is your task: Write a function to find the first duplicate element in a given array of integers. Your code should pass the following test: `assert find_first_duplicate([1, 2, 3, 4, 4, 5]) == 4`

Incorrect 32B response

Plan:

1. Create a function `find_first_duplicate` that takes an array as input.
2. Initialize an empty set to track seen elements.
3. Iterate through each element in the array:
 - a. If the element is already in the set, return it immediately.
 - b. Otherwise, add the element to the set.
4. Use the `final_answer` tool to return the implemented function code.

Final answer: 

Correct 14B response

Plan:

The task is to write a Python function that identifies the first duplicate element in an array of integers. The approach involves iterating through the array while tracking seen elements using a set. When a duplicate is encountered, it is immediately returned as the result.

1. Initialize an empty set to keep track of elements that have been encountered.
2. Iterate through each element in the input array.
3. For each element, check if it is already in the set:
 - If it is, return the element as it is the first duplicate.
 - If it is not, add the element to the set.
4. If no duplicates are found during the iteration, return a default value (e.g., -1) to indicate the absence of duplicates.

This method ensures that the first duplicate is identified efficiently with a time complexity of $O(n)$ and a space complexity of $O(n)$, where n is the number of elements in the array.

Final answer: 
