

Verified Anchor Selection and Adaptive Curriculum for Dynamic Cheatsheet Memory

Stanford CS224N Custom Project

Mengqian Chen

Department of Computer Science
Stanford University
fionamqc@stanford.edu

1 Key Information to include

- Mentor: **Mirac Suzgun**.
- External Collaborators or Sharing project : None.
- Late Days :3 days from Mengqian Chen

Abstract

Dynamic Cheatsheet (DC) improves reasoning by writing reusable notes at test time, but its gains depend on two fragile decisions: *which* problems first seed memory and *whether* each new write is trustworthy enough to keep. We study these issues through a two-stage extension of **DC-RS** across eight dataset splits spanning AIME, Game of 24, GPQA, MMLU-Pro, and MathEquationBalancer. Stage I performs *verified anchor selection*: problems are clustered with the same normalized OpenAI text embeddings used in the original DC pipeline, then anchor candidates are searched using one centrality seed (the cluster medoid) and three universal simplicity seeds (shortest input, fewest symbolic entities, and minimum embedding L1 norm), and finally scored by verified local transfer to nearby neighbors. Stage II runs an anchor-first curriculum with `replicate_orig`, `random_global`, `static`, and `ucb` traversal policies, optionally gating memory writes with a verifier. On the harder AIME benchmarks, static ordering and confidence gating consistently improve over the original DC-RS order; on simpler tasks, the gains are smaller and more heterogeneous. A token analysis further shows that verification is cheap relative to generation, motivating a hierarchical deployment strategy that routes easier tasks to a cheaper generator-verifier pair and reserves the stronger pair for hard reasoning.

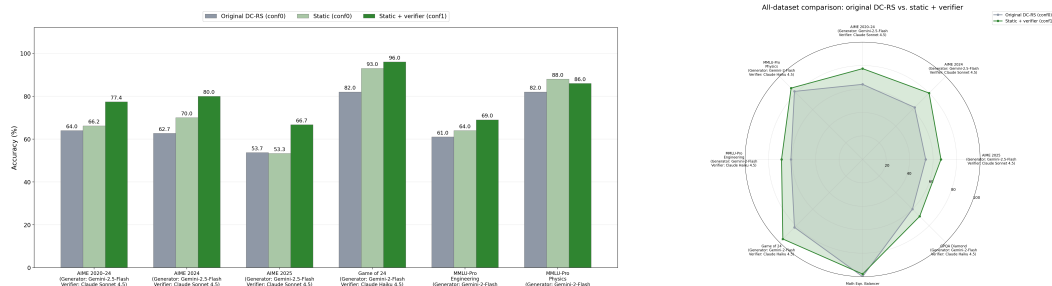


Figure 1: Visual summary of the DC-RS comparisons. **Left:** original DC-RS (`replicate_orig`, `conf0`), static ordering without verification, and static ordering with verification on the six datasets where static ordering is beneficial or competitive. **Right:** all-dataset radar comparison between original DC-RS (`conf0`) and static + verifier (`conf1`).

2 Introduction

Dynamic Cheatsheet (DC) shows that a black-box language model can improve at test time by storing compact, reusable notes instead of solving every benchmark instance from scratch [1]. This idea is especially compelling on hard reasoning benchmarks such as AIME, where many questions share latent tactics but the model must first discover them. In practice, the original DC results suggest both strong upside and strong path dependence: once the system finds a useful pattern, later questions can benefit substantially, but early mistakes or weak writes can also steer the memory in the wrong direction.

More broadly, this work fits into the emerging theme of *test-time compute scaling* [2]: rather than training ever-larger models, we ask how smaller, cost-efficient models can achieve better accuracy at inference time through intelligent allocation of extra compute—here in the form of memory management, curriculum ordering, and lightweight verification. Chain-of-thought prompting [3] and self-consistency decoding [4] have shown that spending more tokens at inference can substitute for larger models; our work extends this principle from single-query strategies to a *multi-query* memory pipeline where the extra compute is amortized across problems. The guiding intuition is that a well-orchestrated pipeline of cheaper models can match or exceed a single expensive model on structured reasoning tasks, provided the extra compute is spent wisely.

This project studies that *memory bootstrap problem* for **DC-RS**, with AIME as the main target setting. Rather than asking whether memory helps in general, we ask a narrower and more practical question: *how should a DC-RS run start, and how should it progress, if we want the evolving memory to become useful quickly and stay reliable?* The central challenge is that the early phase of a run matters disproportionately. A poor first cluster or a low-quality early write can poison later retrieval, while a good initial cluster can create a stable trajectory of reusable insights.

We therefore focus on two interventions. The first is **verified anchor selection**: before Stage II begins, we cluster the benchmark, search for promising seed problems inside each cluster, and choose anchors by verified local transfer rather than by geometry alone. The second is **anchor-aware curriculum scheduling**: after anchors are selected, we compare multiple traversal policies and optionally gate memory writes with a verifier so that low-confidence writes are less likely to enter the cheatsheet.

The report is organized around three questions that match the final experiments and conclusions:

1. Can verified local-transfer scoring identify better cluster anchors than a purely geometric choice?
2. On the harder AIME benchmarks, do anchor-aware static ordering and confidence-gated writes improve over the original DC-RS order and random-order baselines?
3. When we compare AIME with simpler reasoning tasks such as GPQA, MMLU-Pro, GameOf24, and MathEquationBalancer, do the resulting accuracy and token profiles suggest a cost-efficient hierarchy that reserves the stronger model pair for the hardest problems?

This framing makes AIME the primary lens for evaluation, while the other datasets play two supporting roles: they test whether the same mechanisms transfer beyond olympiad-style math, and they help reveal where a cheaper generator-verifier pair may already be sufficient. Figure 2 summarizes the final two-stage DC-RS pipeline studied in the report.

3 Related Work

This project builds directly on **Dynamic Cheatsheet** [1], which demonstrated that a black-box LM maintaining an external memory of reusable strategies can substantially improve test-time reasoning without gradient updates. DC itself draws on a rich line of test-time reasoning methods, including chain-of-thought prompting [3], self-consistency [4], program-aided reasoning [5], and meta-prompting [6]. Its retrieval-then-generate architecture also connects to retrieval-augmented generation [7].

Our curriculum scheduling draws on ideas from curriculum learning [8, 9], which orders training examples to accelerate convergence, and the UCB policy adapts the classic multi-armed bandit exploration-exploitation framework [10, 11]. The confidence-gating mechanism is motivated by process-based and outcome-based verification [12, 13].

Our work does not propose a new memory framework; instead, it studies one open issue from the original paper—the sensitivity of DC-RS to how memory is initialized and updated—and treats anchor selection, traversal order, and verifier-gated writes as a control layer for stabilizing memory formation.

4 Approach

4.1 Problem setting

We study a continual-evaluation setting with problems $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, focusing on **DC-RS** [1]. For each input x_i , DC-RS retrieves relevant prior solved examples, synthesizes them into a fresh cheatsheet, and generates a solution conditioned on that memory. The pipeline has two stages: **Stage I** selects one verified anchor per cluster and builds a global initialization cheatsheet; **Stage II** runs a curriculum over the remaining problems, optionally gating memory writes with a verifier.

4.2 Stage I: verified anchor selection

Embeddings and clustering. Unlike the earlier milestone plan, the final pipeline uses a single embedding family throughout: the OpenAI text embeddings [14] used in the original DC setup. Let $e_i \in \mathbb{R}^d$ denote the normalized embedding for problem x_i . The anchor-selection script compares two clustering algorithms on these embeddings: k-means [15] and k-medoids [16]. For each algorithm, it sweeps over candidate values of k and selects the best valid clustering by silhouette score [17]. The output of this step is a partition $\{C_1, \dots, C_k\}$ together with one medoid-like representative per cluster.

Cluster-local search order. A purely central candidate search is often brittle when the cluster solve rate is low: the most central problems may still be too hard to verify, so geometry alone may fail to discover any usable seed. Our final search therefore combines *centrality* [18] with *simplicity*. For each cluster C_c , we build a candidate seed set consisting of: (i) the cluster medoid, which serves as the centrality seed, (ii) the shortest-input member, measured by input token length, (iii) the member with the fewest symbolic entities, where entities are counted as numbers, math symbols, and $\text{\LaTeX}$ commands appearing in the input, and (iv) the member with minimum embedding L1 norm. The last heuristic replaces the raw embedding-norm heuristic used in earlier drafts: because OpenAI text embeddings are L2-normalized, raw magnitude is unavailable, so minimum L1 norm acts as an embedding-concentration proxy for simplicity. Each seed induces a cosine-similarity ranking of cluster members, and we aggregate those rankings to obtain the cluster-local scan order.

Functional anchor scoring. A good anchor should not merely sit near the center of a cluster; it should produce a cheatsheet that helps solve neighboring problems. For each cluster, we therefore scan members in the order described above and evaluate verified candidates by local transfer on nearest neighbors [19]. Suppose $a \in C_c$ is a candidate problem. The anchor-selection script performs three steps:

1. Solve a without any cheatsheet and verify the answer.
2. If a verifies, rerun it with cumulative cheatsheet extraction to obtain an anchor cheatsheet $M(a)$.
3. Freeze $M(a)$ and evaluate it on the d nearest unvisited neighbors of a inside the same cluster.

The local budget is

$$d_c = \max\left(\left\lfloor \sqrt{|C_c|/5} \right\rfloor, 1\right),$$

which grows mildly with cluster size. The transfer score of anchor a is

$$s(a) = \mathbb{I}[a \text{ verifies}] \cdot \frac{1}{d_c} \sum_{j \in \mathcal{N}_c(a)} \mathbb{I}[x_j \text{ verifies using } M(a)].$$

The cluster winner is the verified candidate with the largest $s(a)$, with ties broken in favor of the earlier candidate in the scan order. To save tokens, we do not exhaustively score every verified member in the cluster: search stops as soon as a candidate achieves perfect local transfer or once two verified candidates have been collected, and only those verified candidates are compared.

Stage I: Verified Anchor Selection

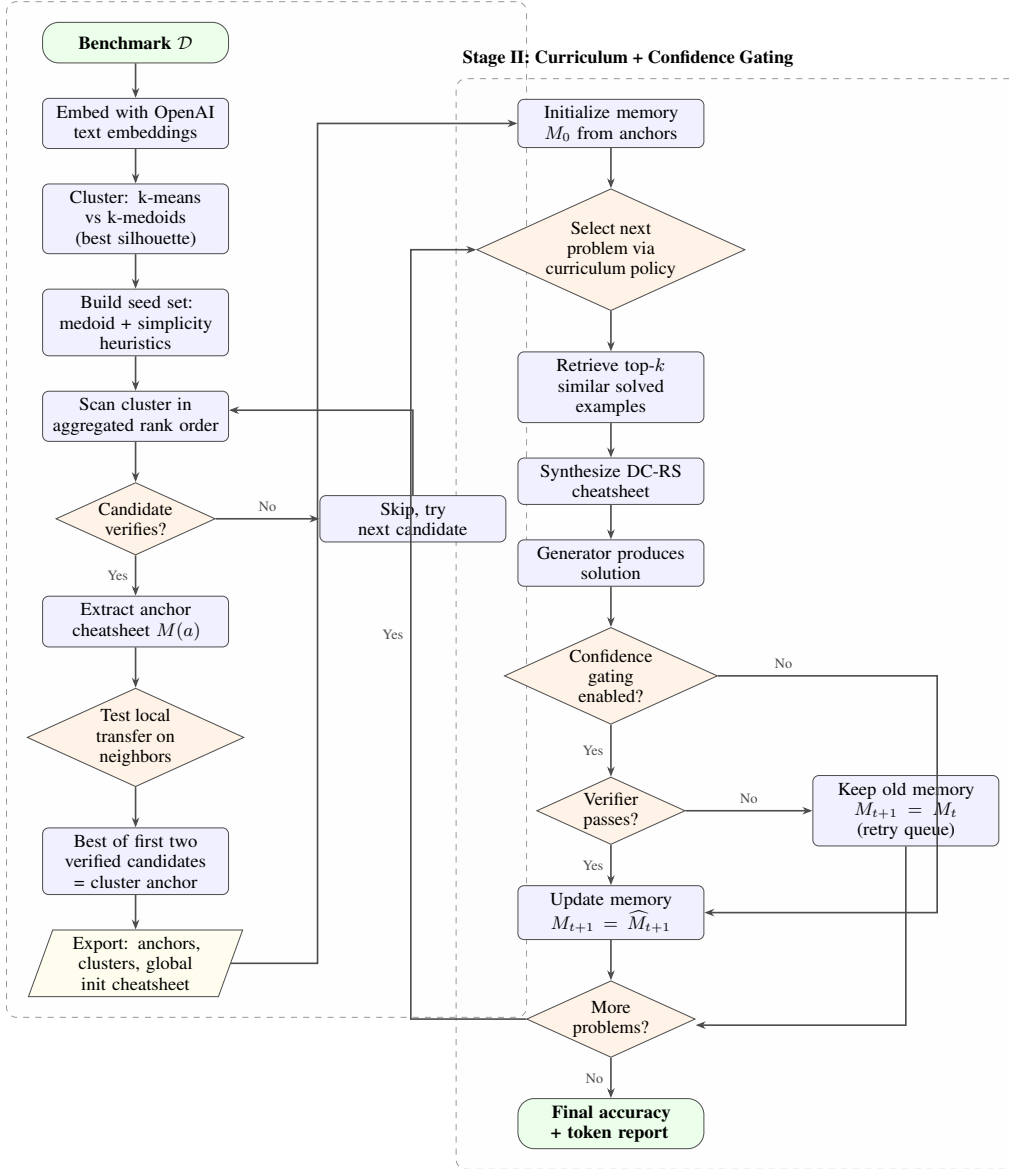


Figure 2: Directed flowchart of the two-stage pipeline. **Stage I** (left) clusters problems, searches for verified anchors via centrality and simplicity seeds, selects the best among the first two verified candidates per cluster, and exports anchors with a global initialization cheatsheet. **Stage II** (right) processes remaining problems under a curriculum policy; the verifier acts as a conditional write gate—if confidence gating is disabled, every solution updates memory directly.

Choosing the upstream clustering. After scoring all clusters under both k-means and k-medoids, we compare the two upstream pipelines using the mean neighbor-transfer score across clusters. Ties are broken by total token usage, favoring the cheaper clustering. The winning configuration is saved as `openai_kmeans` or `openai_kmedoids` and exported as a curriculum-compatible artifact containing: (i) the selected anchor per cluster, (ii) the set of verified neighbors, and (iii) a deduplicated global cheatsheet formed by merging the anchor cheatsheets.

4.3 Stage II: curriculum over the remaining problems

The downstream stage consumes the saved cluster assignments and selected anchors. Every cluster queue is ordered *anchor-first*: the chosen anchor is placed first, and the remaining members are sorted

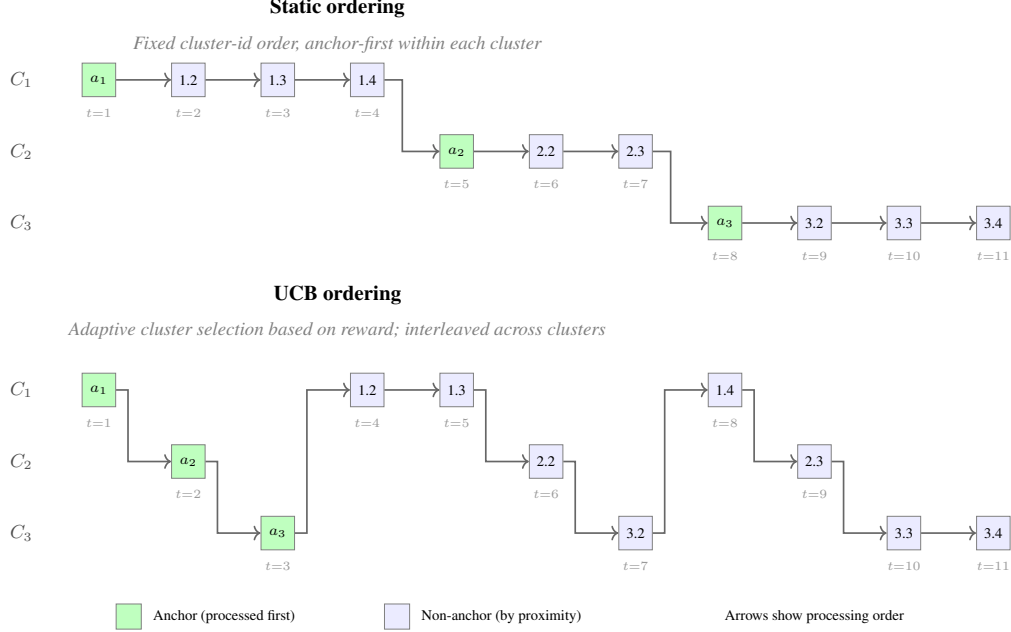


Figure 3: Comparison of static and UCB traversal policies for three clusters (C_1 – C_3). **Static** exhausts each cluster sequentially in fixed cluster-id order, always starting with the anchor. **UCB** interleaves across clusters adaptively: it first visits all anchors (since unvisited clusters have infinite UCB score), then allocates subsequent visits to whichever cluster has the highest upper-confidence-bound reward estimate. Both policies place anchors first within each cluster.

either by proximity to the anchor (the default `closest` policy) or randomly. The main curriculum policies are:

- **Static**: process clusters in deterministic cluster-id order.
- **UCB**: choose the next cluster adaptively using online reward statistics.
- **Random-global**: ignore clusters and process all eligible problems in a globally shuffled order.
- **Replicate-orig**: preserve the original benchmark traversal order used by the earlier DC evaluation.

For UCB scheduling [10], if cluster c has been selected n_c times and has produced r_c operationally successful steps, the score at time t is

$$\text{UCB}_c(t) = \begin{cases} \infty, & n_c = 0, \\ \frac{r_c}{n_c} + \lambda \sqrt{\frac{\log t}{n_c}}, & n_c > 0, \end{cases}$$

where λ corresponds to the code parameter `ucb_c`. This favors clusters that have both high empirical reward and high uncertainty.

4.4 Confidence-gated memory updates

The anchor-selection and curriculum code uses two notions of correctness. *Ground-truth correctness* is computed by the task-specific evaluator (for Game of 24, exact execution-based checking). *Operational correctness* is the signal the system uses online to decide whether to update memory. When confidence gating is disabled, every step is treated as operationally correct and memory always updates. When confidence gating is enabled, a separate verifier model examines the generator output and returns a pass/fail verdict [12, 13].

For cheatsheet-based methods, the update rule is

$$M_{t+1} = \begin{cases} \widehat{M}_{t+1}, & \text{if the verifier passes,} \\ M_t, & \text{otherwise,} \end{cases}$$

where $\widehat{M}_{t+1}$ is the cheatsheet extracted by the generator/curator pair. Intuitively, the verifier acts as a write gate that trades recall for memory precision. Incorrect or unverified steps can be retried

later through a global retry queue, but they are not allowed to contaminate the persistent cheatsheet immediately.

Verifier model selection. The choice of verifier model reflects a deliberate cost-efficiency trade-off. The verifier is intended to serve as a *peer reviewer*—slightly stronger than the generator, but not dramatically so. A much stronger verifier would be wasteful: if a frontier model is available and affordable, it should be used as the generator itself, producing higher-quality solutions directly rather than reviewing weaker ones. Conversely, a verifier weaker than the generator adds no value, since it cannot reliably catch the generator’s mistakes. The sweet spot is a model that is strong enough to independently verify or reject the generator’s reasoning at a fraction of the generation cost. In our experiments, this translates to pairing Gemini-2.5 Flash with Claude Sonnet for AIME and Gemini-2.0 Flash with Claude Haiku for the simpler tasks.

5 Experiments

5.1 Data

The final evaluation covers five benchmark families and eight concrete dataset splits: **AIME_2020_2024**, **AIME_2024**, **AIME_2025**, **GameOf24** [20], **GPQA_Diamond** [21], **MMLU_Pro_Engineering**, **MMLU_Pro_Physics** [22], and **MathEquationBalancer**. This mix spans exact symbolic arithmetic, olympiad-style math reasoning, science question answering, domain-focused academic QA, and structured equation manipulation.

In the Stage-II analysis we emphasize two regimes. The first is *hard reasoning*, represented by the three AIME splits. The second is a *simpler reasoning* group consisting of GameOf24, GPQA, and MMLU, which is useful for testing whether the curriculum still helps when the base generator is already reasonably strong and the tasks expose less headroom for memory bootstrapping. MathEquationBalancer remains part of the full experimental sweep, but it is not central to the ordering analysis reported in the main paper due to space limits.

5.2 Evaluation method

We report the following metrics for every run.

- **Benchmark accuracy:** final task accuracy under the same evaluation logic used by the benchmark harness.
- **Verifier accuracy:** when confidence gating is enabled, the fraction of attempts that the verifier marks as operationally correct.
- **Total tokens:** generator and verifier tokens summed across the entire run.
- **Estimated cost:** token-based cost using the pricing configuration referenced by the code.
- **Runtime:** wall-clock execution time, including retry sleep.
- **Accuracy per token:** benchmark accuracy normalized by token usage, reported both per token and per 1k tokens.

The main headline metric for the final report is the trade-off between benchmark accuracy and total cost/tokens, because the whole purpose of curriculum and verification is not only to raise accuracy but to do so efficiently.

5.3 Experimental details

All experiments use the implementation provided in `anchor_selection_v3.py`, `run_curriculum_ablation.py`, and the accompanying `language_model.py` wrapper. The final Stage-II runs use two model regimes. For the hard AIME benchmarks, the generator is `vertex_ai/gemini-2.5-flash` and the verifier is `anthropic/claude-sonnet-4-6`. For the simpler-task analysis, the generator is `vertex_ai/gemini-2.0-flash`; the lightweight verifier analysis focuses on the `anthropic/claude-haiku-4-5` runs on GPQA and MMLU, while GameOf24 is retained mainly for the ordering comparison. Generation is run at temperature 0 with code execution enabled. AIME uses a larger 16k-token budget for both generation and verification, while the simpler tasks use smaller generation budgets and verifier budgets (4k tokens).

Task	Rand. conf0 $\mu \pm \sigma$	Rep. conf0	Static conf0	UCB conf0	Rand. conf1 $\mu \pm \sigma$	Rep. conf1	Static conf1	UCB conf1
AIME_2020_2024	n/a	0.6400	0.6617	0.7293	n/a	0.7444	0.7744	0.7293
AIME_2024	0.6750 ± 0.0739	0.6267	0.7000	0.6667	0.7167 ± 0.0577	0.7000	0.8000	0.7333
AIME_2025	0.5167 ± 0.0430	0.5367	0.5333	0.5667	0.5333 ± 0.0272	0.6333	0.6667	0.4667

Table 1: Stage-II AIME results. The seeded `random_global` rows provide the cleanest order-agnostic baseline on AIME_2024 and AIME_2025. Static ordering improves over the seeded random mean under both confidence settings on both splits, while UCB is mixed.

Upstream anchor selection compares `openai_kmeans` and `openai_kmedoids`, then saves the better configuration by verified neighbor transfer. Downstream curriculum runs use **DC-RS only**, with a shared persistent memory, at most two attempts per problem, and a single global retry queue for operational failures after the primary pass. Retrieval uses top- k similarity lookup [23] with $k = 3$ in the same OpenAI embedding space, and the generator synthesizes a fresh cheatsheet from those retrieved examples before answering. Unless otherwise specified, within-cluster order is `closest` and the UCB coefficient is left at its default value.

A practical detail that matters for interpreting the curriculum tables is the distinction between *ordering* and *memory initialization*. The `replicate_orig`, `static`, and `random_global` settings in the reported tables start from an empty initialization cheatsheet, while the reported `ucb` runs use a global initialization cheatsheet. UCB therefore changes both the traversal policy and the initial memory state, so it should be interpreted as a stronger but less controlled comparison than static-versus-random order alone.

5.4 Results

Hard reasoning: AIME is strongly order-sensitive. For AIME_2024 and AIME_2025 we have four `random_global` seeds under both confidence settings, so we use the seeded random runs as the order-agnostic baseline. Turning confidence on improves the random-global mean on both splits, from 0.6750 ± 0.0739 to 0.7167 ± 0.0577 on AIME_2024 and from 0.5167 ± 0.0430 to 0.5333 ± 0.0272 on AIME_2025. This indicates that verification improves not only average accuracy but also run-to-run stability.

Static ordering consistently beats the seeded random mean under both confidence settings on both AIME splits. UCB is mixed: it underperforms the random baseline on AIME_2024 with confidence off, exceeds it with confidence on, and then reverses on AIME_2025. AIME_2020_2024 does not have a seeded `random_global` baseline in the current table, but its paired results show the same pattern that motivated the method: verification helps `replicate_orig` ($0.6400 \rightarrow 0.7444$) and `static` ($0.6617 \rightarrow 0.7744$), whereas `ucb` remains flat at 0.7293.

Simpler reasoning: ordering helps less consistently. The simpler tasks tell a different story. Without the verifier, static ordering improves over the original DC-RS order on GameOf24, MMLU-Pro Engineering, and MMLU-Pro Physics, but it hurts GPQA. Averaged across GameOf24, GPQA, and the two MMLU subsets, `static conf0` improves over the original order by $+0.0275$ accuracy, while `ucb conf0` is slightly worse on average. Figure 1 makes this contrast visible: the AIME splits show the clearest and most consistent benefit, whereas the simpler tasks exhibit smaller task-specific shifts.

The verifier also has heterogeneous effects on the simpler tasks. It helps static ordering on GameOf24, GPQA, and MMLU-Pro Engineering, but slightly hurts MMLU-Pro Physics. Under UCB it produces a large jump on GameOf24, a moderate gain on MMLU-Pro Engineering, and small degradations on GPQA and MMLU-Pro Physics. This suggests that verification is most useful when the task and policy create enough ambiguity in memory quality for an extra confidence filter to matter; when the base generator is already strong and memory quality is less ambiguous, the benefit is smaller and sometimes disappears.

An important caveat is that the simpler datasets lack seeded `random_global` baselines, so the comparison is always against the original benchmark order, which may itself be favorable (e.g., if it already groups related problems or progresses from easier to harder). Ordering demonstrably matters on AIME; the evidence on simpler tasks is suggestive but less controlled.

Model pair	Gen tok / prob.	Ver tok / prob.	Total tok / prob.	Cost / prob.
Gemini-2.5 + Sonnet (AIME)	48.9k	4.9k	108.3k	\$0.200
Gemini-2.0 + Haiku (GPQA/MMLU)	9.1k	5.6k	42.4k	\$0.016

Table 2: Per-problem token and cost comparison for the hard and simple model pairs. The main cost increase comes from the stronger generator rather than from the verifier.

6 Analysis

Verification is usually cheap relative to generation. The AIME results suggest that verification is a cost-effective way to improve reliability. On the seeded `random_global` AIME runs, confidence increases average accuracy by +4.17 points on AIME_2024 and by +1.67 points on AIME_2025 while also reducing standard deviation. Yet the verifier budget remains a relatively small fraction of generation: across the AIME confidence-on runs in Table 1, verifier tokens are roughly one-tenth of generator tokens on average. This is important because it means the gain does not require replacing the generator with a second equally expensive generator pass; instead, a cheaper reviewer can filter bad writes and stabilize memory formation.

The same point becomes clearer when we compare the hard and simple model pairs per problem. For the hard regime (Gemini-2.5 Flash generator with Sonnet verifier on the AIME splits), the run average is about 48.9k generator tokens and 4.9k verifier tokens per problem, for 108.3k total tokens and \$0.200 per problem. For the lightweight regime (Gemini-2.0 Flash generator with Haiku verifier on GPQA and MMLU), the run average is about 9.1k generator tokens and 5.6k verifier tokens per problem, for 42.4k total tokens and \$0.016 per problem. The main cost increase therefore comes from the stronger generator rather than from the verifier, which stays in roughly the same token range across the two regimes.

This pattern motivates a hierarchical deployment strategy: route easier problems to a cheaper generator-verifier pair and escalate only hard cases to the stronger generator, preserving verification’s benefit without paying the larger generation cost on every problem.

Verifier failure as a source of `conf1` performance drops. On some datasets, enabling confidence gating (`conf1`) slightly *hurts* accuracy relative to the ungated baseline (`conf0`). We attribute this primarily to verifier failure: when the verifier incorrectly rejects a correct solution, the cheatsheet misses a useful write, and downstream problems lose the benefit of that memory entry. This effect is most visible on tasks where the verifier’s domain competence is marginal—for instance, MMLU-Pro Physics under UCB scheduling. The implication is that further improvement to the verification prompt, or the use of a stronger verifier LLM that can more reliably distinguish correct from incorrect solutions, would narrow or eliminate these `conf1` regressions. As verification costs remain a small fraction of total tokens, upgrading the verifier is a high-leverage investment compared with upgrading the generator.

7 Conclusion

Ordering matters most on the harder AIME benchmarks, where static traversal and verification both improve over the original DC-RS order. On simpler tasks the gains are smaller and more heterogeneous, suggesting that memory bootstrapping has the most leverage when the underlying reasoning problem is difficult enough for reusable strategies and write precision to matter. Token accounting shows that verification is cheap relative to generation, motivating a hierarchical deployment strategy that reserves the stronger generator-verifier pair for hard problems.

One promising direction for future work is the UCB policy, which produced mixed results. Its current binary reward signal is a coarse proxy for memory quality; richer reward functions—incorporating verifier confidence, cheatsheet change magnitude, or downstream transfer—could make adaptive cluster selection more effective.

References

- [1] Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. arXiv preprint arXiv:2504.07952, 2025.
- [2] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. arXiv preprint arXiv:2408.03314, 2024.
- [3] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
- [4] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171, 2022.
- [5] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.
- [6] Mirac Suzgun and Adam Tauman Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding. arXiv preprint arXiv:2401.12954, 2024.
- [7] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [8] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, pages 41–48, 2009.
- [9] Alex Graves, Marc G. Bellemare, Jacob Menick, Rémi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 1311–1320, 2017.
- [10] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3):235–256, 2002.
- [11] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 661–670, 2010.
- [12] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, et al. Let’s verify step by step. arXiv preprint arXiv:2305.20050, 2023.
- [13] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168, 2021.
- [14] OpenAI. Embeddings guide. <https://platform.openai.com/docs/guides/embeddings>, 2026. Accessed 2026-02-27.
- [15] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.
- [16] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, 1990.
- [17] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

- [18] Linton C. Freeman. Centrality in social networks: Conceptual clarification. *Social Networks*, 1(3):215–239, 1978.
- [19] Thomas M. Cover and Peter E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.
- [20] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [21] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.
- [22] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyang Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhui Chen. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. *arXiv preprint arXiv:2406.01574*, 2024.
- [23] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, 2019.

A Appendix: Verification Prompt

The following is the full verification prompt used by the confidence-gating module to decide whether a generator’s output should be committed to the persistent cheatsheet memory. Placeholders `[[QUESTION]]`, `[[PARSED_FINAL_ANSWER]]`, and `[[GENERATOR_OUTPUT]]` are filled at runtime.

SOLUTION VERIFIER

Purpose

You are a verification module. Given a question, the generator’s output, and its parsed final answer, decide whether the final answer is correct.

Inputs

- **QUESTION**: The original problem statement (including any rules or constraints).
- **GENERATOR OUTPUT**: The generator’s full reasoning trace.
- **PARSED FINAL ANSWER**: The candidate answer extracted from the generator output.

Verification Procedure

1. Read the QUESTION carefully. Identify every constraint and requirement (e.g., “use each number exactly once”, target value, answer format, etc.).
2. **Verify the FINAL ANSWER directly against the question’s requirements.** Do NOT just follow the generator’s reasoning – independently check the answer yourself.
3. For simple arithmetic answers (e.g., “What is 12+13?”):
 - Evaluate the expression yourself step-by-step.
 - Check the result matches the target.
 - Check ALL constraints (e.g., each input number used exactly once, no extra numbers, correct operators, etc.).
- 3b. For competition math / multi-step derivation problems:
 - **Step C1 – Sanity check the answer FIRST** (before reading reasoning):
 - * Does the answer fall within the expected range?
 - * Is the answer type correct? (integer when integer is expected, etc.)
 - **Step C2 – Pick ONE critical computation** from the solution and redo it yourself from scratch. Do NOT just read the generator’s arithmetic and nod along – actually recompute it independently. If your result disagrees → FAIL.
 - **Step C3 – Back-check if possible**: Can you substitute the final answer back into the problem to verify it satisfies the original conditions? If yes, do so. If the answer fails the back-check → FAIL.
 - **Step C4 – Approach plausibility**: Does the overall method make sense for this type of problem? Are there logical gaps or unjustified leaps?
 - Only PASS if steps C1–C4 all check out. If you cannot independently verify at least one key computation, FAIL.
4. For constrained expressions (e.g., “use specific inputs to reach a target”):
 - **Number audit**: Extract all values used in the expression and compare against the required inputs. They must match exactly (no extras, no missing).
 - **Evaluate**: Compute the expression step-by-step and verify the result matches the target.

- FAIL if the inputs don't match OR the result is wrong.
5. For multiple-choice questions:
- **Step M1 – Independent solve:** BEFORE looking at the generator's reasoning, read the question and all options yourself. Determine the correct answer based **ONLY** on your own knowledge. Write down your independent answer first.
 - **Step M2 – Confidence check:** Can you independently and confidently determine the correct answer? Ask yourself: "Does this question require specialized graduate-level knowledge, complex spatial reasoning, or domain expertise I might lack?" If YES, or if you have **ANY** uncertainty → **FAIL immediately**. Do **NOT** assume the generator is correct just because its reasoning sounds plausible.
 - **Step M3 – Eliminate alternatives:** You **MUST** explain why **EACH** of the other options is definitively wrong. If you cannot confidently rule out even **ONE** alternative option → **FAIL immediately**.
 - **Step M4 – Compare:** If you **CAN** independently solve it **AND** have eliminated all alternatives, compare your answer with the **PARSED FINAL ANSWER**. If they disagree → **FAIL immediately**. Do not let the generator's reasoning change your mind.
 - **Step M5 – Reasoning check:** Even if answers agree, check for: logical errors, contradictions, unsupported leaps, missing considerations. If **ANY** flaw is found → **FAIL**.
 - **PASS only if ALL of the following hold:** (a) you can independently verify with certainty, (b) you have ruled out every other option with clear justification, (c) your independent answer matches, (d) reasoning is completely sound, **AND** (e) you are highly confident. If any one condition fails → **FAIL**.
6. For free-form answers: check factual correctness and completeness.

PASS / FAIL Rules

Your default stance is FAIL. Only upgrade to PASS when you have strong, verifiable evidence the answer is correct.

- PASS requires HIGH confidence that the PARSED FINAL ANSWER is correct. Mere plausibility is NOT enough.
- FAIL if you cannot independently verify the answer. NEVER assume the generator is correct.
- FAIL if the answer violates ANY requirement, if your independent assessment disagrees, or if you have ANY doubt.
- Keep your analysis concise. If you find yourself writing extensively without reaching a clear conclusion, that signals uncertainty → lean toward FAIL.
- Do NOT penalize equivalent expressions (e.g., $a + b$ vs $b + a$, extra parentheses).
- For pure arithmetic / equation problems ONLY: do not penalize reasoning-path imperfections when the final numerical answer is verifiably correct through direct computation. This exception does NOT apply to multiple-choice or knowledge-based questions.
- For multiple-choice / knowledge questions: ANY flaw in reasoning IS grounds for FAIL, even if the final letter matches. You cannot “verify” an MC answer through arithmetic alone.
- **When in doubt, FAIL.** It is much better to wrongly reject a correct answer than to wrongly accept an incorrect one.

Output Format (IMPORTANT: Reasoning MUST come before verdict)

You MUST follow this exact format. Write the ANALYSIS first, then decide PASS or FAIL.

ANALYSIS:

(Keep this concise (under 400 words). Focus on: (1) sanity-checking the answer format/range, (2) independently recomputing one key step, (3) back-checking if possible, (4) any concrete errors found.)

VERIFICATION RESULT: PASS or FAIL

CHECKED ANSWER:

⟨answer⟩

...

⟨/answer⟩

Constraints for CHECKED ANSWER

- If PASS, CHECKED ANSWER must be exactly `[PARSED_FINAL_ANSWER]`.
- If FAIL and you can confidently provide a corrected answer, put it inside `<answer>`.
- If FAIL and you cannot confidently fix it, leave the `<answer>` block empty.

[[QUESTION]]

PARSED FINAL ANSWER: [[PARSED_FINAL_ANSWER]]

GENERATOR OUTPUT: [[GENERATOR_OUTPUT]]

Figure 4: Verification prompt used by the confidence-gating module. The verifier receives a question, the generator’s full output, and the parsed final answer, then independently decides PASS or FAIL. Placeholders (shown in teal) are filled at runtime.