

Recursive Self-Improvement for Continual Adaptation in Code Generation

Custom Project

Aaditya Nalawade
Department of Computer Science
Stanford University
aaditya1@stanford.edu

Ethan Goodhart
Department of Computer Science
Stanford University
goodhart@stanford.edu

Chandra Suda
Department of Computer Science
Stanford University
csuda@stanford.edu

Project Mentor: Minsik Oh
External Collaborators: No
Late Days: No

Abstract

We present a verifier-only recursive self-improvement framework for code generation that combines Self-Distilled Policy Optimization (SDPO) with Retrieval-Augmented Generation (RAG) under a prompt-based routing policy. Given only test-case feedback (no golden solutions), the agent generates solutions, recursively refines failed attempts, distills self-discovered successful behaviors into model parameters, and optionally stores reusable lessons in an external vector memory. On LiveCodeBench, the best 7B configuration (4 recursive attempts per rollout) improves Qwen2.5-7B-Instruct from 32% to 38% solve rate (+6 pp) and from 51.3% to 55.9% test-case accuracy. Ablations across attempt budgets (4/16/32), model scales (1.5B vs. 7B), and adaptation choices show three key findings: (1) recursive self-improvement requires a minimum base capability (the 1.5B model degrades), (2) SDPO-only outperforms SDPO+RAG in our current regime, and (3) larger attempt budgets increase training-time discovery of perfect rollouts but reduce evaluation-time generalization, indicating over-specialized self-corrections. We argue that the RAG gap here is primarily a memory scale and coverage issue: with only tens of stored lessons, retrieval lacks sufficient semantic density to consistently match new problems.

1 Introduction

Deployed language models must continuously acquire new skills without forgetting prior ones, a challenge known as *continual learning* [1]. Self-Distilled Policy Optimization (SDPO) [2] addresses this with a demonstration-conditioned teacher copy that generates on-policy training signals, but it applies parametric updates indiscriminately and relies on externally provided demonstrations.

We propose **recursive self-improvement**: rather than requiring golden solutions, our agent bootstraps its own improvement using only a test-case verifier that provides rich structured feedback

(error traces, failing inputs, execution diagnostics). A model capable of occasionally solving a problem across several attempts can *distill that success back into itself*, progressively expanding the frontier of problems it can reliably solve. To our knowledge, this is the first system where a model improves itself through self-distillation in a fully autonomous loop: it generates and recursively refines solutions, externalizes reusable lessons into a retrievable knowledge store, and directs its own learning through prompt-based tool calls that select adaptation actions. No golden solutions are ever supplied.

Our contributions are:

- A recursive self-improvement framework requiring only verifier feedback, combining on-policy self-distillation with externalized knowledge under a prompt-based routing policy the model controls itself.
- An empirical evaluation on LiveCodeBench with Qwen2.5-7B-Instruct [3], achieving +6 pp solve rate through pure self-improvement.
- Ablation studies across recursive attempt counts, model scales, and adaptation strategies that reveal critical design constraints for self-improving agents, including a paradoxical finding that more training-time attempts improve solution discovery but degrade evaluation performance.

2 Related Work

Our approach sits at the intersection of self-improvement, distillation, and external memory. In the distillation line, classic knowledge distillation [4] transfers behavior from teacher to student, while SDPO [2] brings this into an on-policy setting by conditioning the teacher on demonstrations from current rollouts. SPIN [5] similarly highlights that self-generated training signals can drive meaningful gains. We build on this family of ideas, but remove reliance on externally provided demonstrations.

In parallel, reasoning-bootstrapping methods such as STaR [6] improve reasoning quality by training on successful traces selected from model outputs. Our method is closest in spirit to this self-bootstrapping view, but differs in mechanism: rather than rejection-style filtering over traces, we use recursive verifier feedback and top- k distributional distillation to transfer self-discovered corrections back into the base policy.

For non-parametric adaptation, MemoryBank [7] and RAG [8] demonstrate that retrieval can extend model behavior with external memory. We use this memory channel not as a standalone replacement for weight updates, but as *lesson consolidation*: storing transferable failure-derived insights that can be retrieved on future problems while SDPO performs parametric updates in parallel.

3 Approach

3.1 System Architecture

Our agent consists of four components: a **student model** (Qwen2.5-7B-Instruct), an **EMA teacher** (exponential moving average copy), a **RAG knowledge store** (FAISS-indexed vector database [9]), and a **prompt-based router** (the student model itself, prompted to select adapta-

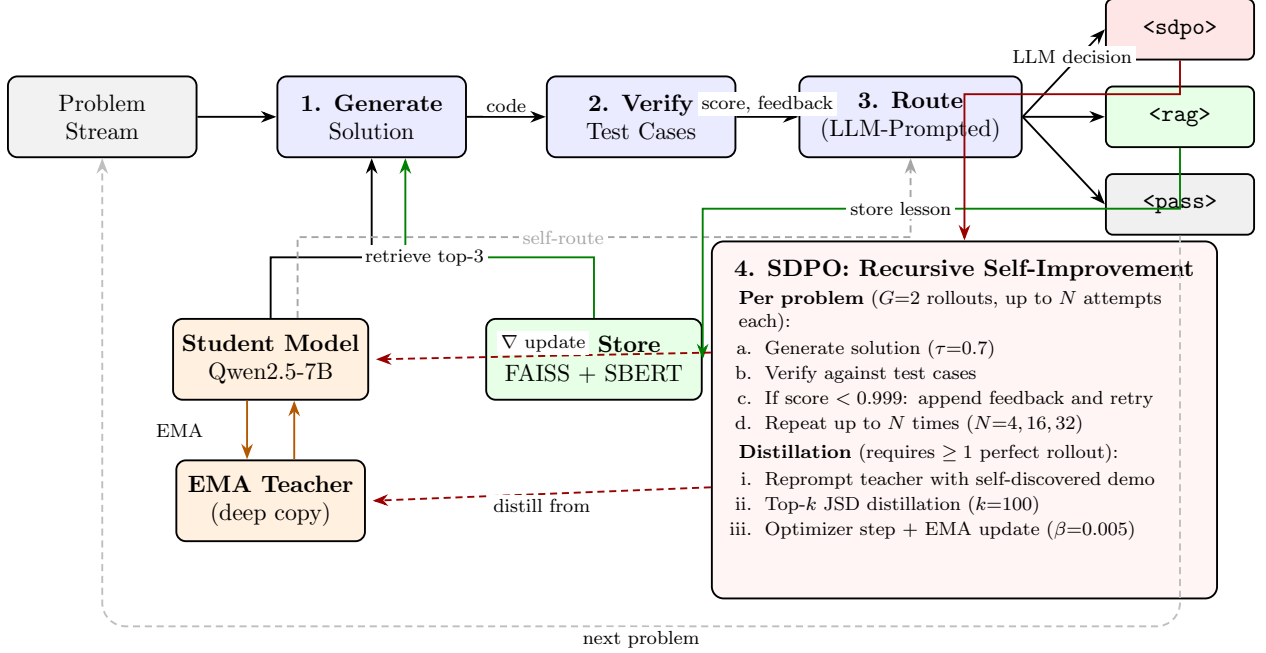


Figure 1: Architecture of the recursive self-improvement system. For each problem: **(1)** the student generates a solution with optional RAG context, **(2)** the solution is verified against test cases, **(3)** the student-as-router selects an adaptation action via prompt-based reasoning (with rule-based fallback), and **(4)** the chosen action executes. The SDPO path triggers recursive self-improvement: $G=2$ rollouts each perform up to N sequential attempts with cumulative feedback. Only rollouts achieving $\geq 99.9\%$ accuracy are used for distillation. No golden solutions are provided at any stage.

tion actions). Figure 1 illustrates the full system.

For each problem in the training stream, the agent executes a four-phase loop:

1. **Generate:** Build a code prompt from the problem description and test specification. Query the RAG store for relevant knowledge chunks (top-3 by cosine similarity using Sentence-BERT embeddings [10]). Generate a solution greedily (temperature = 0).
 2. **Verify:** Execute the solution against test cases in a sandboxed environment. Obtain a score (fraction of tests passed), accuracy, and structured feedback describing which tests failed and why, including error traces and execution diagnostics.
 3. **Route:** The student model, prompted with the problem summary, score, and feedback, outputs one of three action tags via prompt-based reasoning:
 - **<pass>**: Solution is correct or near-correct; no adaptation needed.
 - **<rag>**: Partial success with specific, transferable errors. Store a concise lesson.
 - **<sdpo>**: Fundamental failure requiring parametric improvement. Queue for self-distillation.
- A rule-based fallback triggers only if the model’s output cannot be parsed ($\text{score} \geq 0.9 \rightarrow \text{pass}$, $0 < \text{score} < 0.9 \rightarrow \text{rag}$, $\text{score} = 0 \rightarrow \text{sdpo}$).
4. **Adapt:** Execute the routed action. SDPO problems are batched before executing a gradient step.

3.2 Self-Distilled Policy Optimization (SDPO)

When a batch of SDPO-routed problems is collected, we execute the recursive self-improvement and distillation process. This is the core mechanism through which the model improves itself using only verifier feedback.

Phase 1: Recursive rollout generation. For each problem, we generate $G = 2$ independent rollouts. Each rollout proceeds as a recursive self-improvement loop for up to N sequential attempts (where $N \in \{4, 16, 32\}$ across our experimental configurations):

1. **Attempt 1:** Generate a solution from the original prompt (temperature $\tau = 0.7$).
2. **Verify:** Execute against the test suite. If score ≥ 0.999 (near-perfect), stop; this rollout has succeeded.
3. **Feedback accumulation:** Append the structured failure feedback to a cumulative *feedback window*, a growing context of all prior failed attempts and their error descriptions.
4. **Re-attempt:** Re-prompt the model with the original problem plus the full feedback window, explicitly instructing it to analyze prior failures and generate a fundamentally different solution.
5. Repeat steps 2–4 until success or N attempts exhausted.

The feedback window template explicitly instructs: “*Carefully analyze each error above, understand WHY the previous approach failed, then choose a DIFFERENT algorithm or fix the root cause. Do NOT repeat the same logic that already failed.*”

Only rollouts achieving $\geq 99.9\%$ test-case accuracy proceed to distillation. This ensures the teacher is always conditioned on solutions the model *proved correct through its own recursive refinement*.

Phase 2: Reprompted teacher distillation. For each problem with at least one successful rollout, we construct an enriched teacher prompt containing (a) the original problem, (b) the self-discovered correct solution, and (c) feedback from the initial failed attempt. The EMA teacher performs a forward pass on this reprompted input with no gradients, extracting top- k log-probabilities ($k = 100$) at each response position. The student performs a forward pass on the *original* prompt (without demonstrations) and gathers its log-probabilities at the teacher’s top- k indices. We compute Jensen-Shannon Divergence ($\alpha = 0.5$):

$$\mathcal{L}_{\text{JSD}} = (1 - \alpha) \text{KL}(p_s \| m) + \alpha \text{KL}(p_t \| m), \quad m = (1 - \alpha) p_s + \alpha p_t \quad (1)$$

where p_t and p_s are the teacher and student top- k distributions. A tail probability bucket ($\log(1 - \sum_i p_i)$) accounts for the remaining vocabulary mass. No separate reference model KL penalty is used ($\beta_{\text{ref}} = 0$), eliminating the need for a frozen reference copy and saving ~ 14 GB VRAM.

Phase 3: Optimizer step and EMA update. Gradients are scaled by $1/N_{\text{rollouts}}$, clipped ($\|\nabla\|_{\text{max}} = 1.0$), and applied via 8-bit AdamW [11]. The teacher is then updated: $\theta_t \leftarrow (1 - \beta)\theta_t + \beta\theta_s$ with $\beta = 0.005$.

The teacher has an *informational advantage*: it sees a correct solution the model discovered through recursive self-improvement, so its predictions encode knowledge the student currently lacks. Distilling from this enriched teacher teaches the student to reach correct solutions more directly, without recursive refinement at inference time.

3.3 RAG Knowledge Store

When the router selects `<rag>`, the model generates a concise lesson (2–3 sentences, focusing on the *type* of mistake rather than problem-specific code) along with a truncated problem summary, and stores the combined chunk in a FAISS `IndexFlatIP` index with Sentence-BERT (`all-MiniLM-L6-v2`) embeddings [10]. Each stored chunk contains both the problem context and the extracted lesson, enabling retrieval by problem similarity. During generation, the top-3 most similar chunks are retrieved and prepended as system context, giving the model a non-parametric self-improvement channel.

3.4 Data

We use LiveCodeBench [12], a contamination-free competitive programming benchmark with 924 problems sourced from Codeforces, LeetCode, and AtCoder after model training cutoff dates. Problems span difficulty levels from easy to extremely hard. We split the dataset 70/30: approximately 647 problems for training and 277 for testing. At evaluation time, we subsample 50 test problems for tractable compute.

LiveCodeBench is substantially harder than standard benchmarks: GPT-4 achieves $\sim 55\%$ pass rate versus $>85\%$ on HumanEval. Our Qwen2.5-7B baseline achieves 32% solve rate, consistent with 7B-class models on competitive programming.

3.5 Evaluation Method

We evaluate five configurations:

1. **Baseline 7B**: Qwen2.5-7B-Instruct, no training, no RAG.
2. **Model + SDPO**: SDPO-trained 7B model without RAG at inference.
3. **Model + RAG + SDPO**: Full system with both SDPO training and RAG retrieval.
4. **Baseline 1.5B**: Qwen2.5-1.5B-Instruct, no training.
5. **1.5B + SDPO/RAG**: Same pipeline applied to the 1.5B model.

We report **test-case accuracy** (fraction of individual test cases passed, averaged over problems) and **solve rate** (fraction of problems where all test cases pass). Evaluation uses greedy decoding (temperature = 0). Checkpoints are saved every 50 training problems.

3.6 Experimental Details

All experiments run on NVIDIA H100 GPUs via Modal cloud compute. The model uses BFloat16 precision with gradient checkpointing for memory efficiency. Across the three 7B runs, observed wall-clock training times were approximately 6h12m (v1, 500 problems), 11h11m (v2, 250 problems), and 11h10m (v3, 200 problems), with SDPO updates as the dominant runtime cost.

We conduct four training runs: three on the 7B model with 4, 16, and 32 sequential attempts per rollout, and one on the 1.5B model with 4 attempts. The three 7B Modal runs currently process 500 (v1), 250 (v2), and 200 (v3) training problems, with checkpoints every 50 steps; our primary checkpoint-to-checkpoint comparisons use the overlapping 50–250 range.

Hyperparameter	Value	Hyperparameter	Value
Base model	Qwen2.5-7B-Instruct	Sampling temperature	0.7
Learning rate	2×10^{-6}	Distillation top- k	100
Optimizer	8-bit AdamW	JSD mixture (α)	0.5
EMA rate (β)	0.005	Success threshold (θ)	0.999
Rollouts per problem (G)	2	Reference KL (β_{ref})	0.0 (disabled)
Sequential attempts per rollout (N)	4 / 16 / 32	Max gradient norm	1.0
RAG top- k retrieval	3	RAG embedding model	all-MiniLM-L6-v2

Table 1: Hyperparameter configuration.

3.7 Results

Table 2 gives the main takeaway: SDPO-only with 4 recursive attempts (v1) is the best configuration, improving the 7B baseline by +6 pp solve rate at its best checkpoint. This is achieved without golden solutions or external demonstrations. The same table also shows an important boundary condition: the 1.5B model degrades under the same pipeline, indicating a clear capability threshold for successful self-improvement.

Configuration	Test-Case Acc	Solve Rate	Correct
Baseline (Qwen 7B)	0.5130	0.3200	16/50
<i>7B, v1 (4 attempts per rollout)</i>			
Model + SDPO	0.5595	0.3800	19/50
Model + RAG + SDPO	0.5200	0.3400	17/50
<i>7B, v2 (16 attempts per rollout)</i>			
Model + SDPO	0.5545	0.3000	15/50
Model + RAG + SDPO	0.4760	0.3200	16/50
<i>7B, v3 (32 attempts per rollout)</i>			
Model + SDPO	0.5305	0.3000	15/50
Model + RAG + SDPO	0.4715	0.2800	14/50
Baseline (Qwen 1.5B)	0.1400	0.1400	7/50
1.5B + SDPO	0.0800	0.0800	4/50
1.5B + RAG + SDPO	0.1000	0.1000	5/50
1.5B + RAG only	0.1200	0.1200	6/50

Table 2: Performance on 50 test problems (best checkpoint per configuration). The 7B SDPO-only with 4 attempts achieves the best result. The 1.5B model degrades after training, establishing a capability threshold.

Table 3 shows a simple training pattern for v1: steady gains, a peak around step 200, and a mild late regression consistent with overfitting. Across checkpoints, SDPO-only is consistently stronger than RAG+SDPO on solve rate, so most of the improvement in our setting comes from the parametric SDPO update path rather than retrieval.

Comparing v1/v2/v3 in Table 2 and Table 4 reveals our key paradox: increasing recursive attempts improves training-time recovery (more perfect rollouts found after retries) but hurts evaluation-time

Step	SDPO Acc	SDPO Solve	RAG+SDPO Acc	RAG+SDPO Solve
50	54.4%	32%	50.8%	32%
100	56.5%	36%	53.6%	34%
150	54.9%	36%	53.9%	34%
200	55.9%	38%	51.8%	34%
250	52.9%	36%	54.9%	32%

Table 3: Training checkpoint progression for v1 (4 recursive attempts). Baseline: 51.3% accuracy, 32% solve rate.

generalization. Intuitively, deeper retry loops appear to encourage locally patched solutions that pass in training but transfer less reliably at test time.

Config	Perfects	Att. 1	Att. >1	% from Att. >1	Eval Solve Rate
v1 ($N=4$)	106	40	66	62%	38%
v2 ($N=16$)	54	11	43	80%	30%
v3 ($N=32$)	67	11	56	84%	30%

Table 4: The attempt-count paradox: more recursive attempts improve training-time solution discovery but degrade evaluation performance. “Perfects” counts rollouts achieving $\geq 99.9\%$ accuracy. Despite 84% of v3 perfects coming from recursive refinement, v3 evaluation solve rate (30%) is worse than v1 (38%).

The same evidence also explains why SDPO-only outperforms SDPO+RAG in our runs: with only ~ 20 – 30 stored lessons by step 200, retrieval coverage is still sparse, so RAG often adds weakly relevant context. We interpret this as a scale limitation of the memory store, not a fundamental failure of retrieval. Together with the 1.5B ablation in Table 2, the results point to two practical requirements for this framework: sufficient base model capability and sufficient memory coverage.

4 Analysis

Verifier-only self-improvement. A critical property of our system is that no golden solutions are ever provided. The entire pipeline operates with only a test-case verifier providing rich structured feedback (error traces, failing inputs, execution diagnostics). The model uses this signal to recursively refine its approach. When it discovers a correct solution, it distills that self-discovered knowledge into its weights. This is genuine recursive self-improvement: the model is both the learner and the source of training signal, with the verifier serving only as an objective function.

Training dynamics: routing breakdown. Table 5 shows the routing decisions across training runs. The v1 configuration (500 problems processed) routes 52% of problems to SDPO, 14% to RAG, and 34% to PASS. The high PASS rate (34%) reflects problems the model can already solve correctly on first attempt, consistent with its 32% baseline solve rate.

Config	Problems	SDPO	RAG	PASS	Perfects	From Att. > 1
v1 ($N=4$)	500	261	69	170	106	66 (62%)
v2 ($N=16$)	250	81	56	113	54	43 (80%)
v3 ($N=32$)	200	63	39	98	67	56 (84%)

Table 5: Routing breakdown and solution discovery across training configurations.

RAG knowledge quality. To understand why RAG did not help quantitatively, we examine what the model chose to store. Figure 2 shows a representative chunk.

```

Problem summary: You are given a string s consisting of only
lowercase English letters. In one operation, you can do the
following: Select any non-empty substring of s, possibly the
entire string, then replace each one of its characters with
the previous character of the En...

Lesson: The solution likely attempted to process the entire
string in a single pass without properly managing the time
complexity, leading to a Time Limit Exceeded error. In similar
problems, be cautious of operations that require processing
substrings repeatedly or in a nested manner, as they can quickly
become inefficient, especially with large inputs.

```

Figure 2: Example RAG chunk from the externalized knowledge store. Each chunk contains a truncated problem summary and a model-generated lesson focusing on the *type* of mistake. The model autonomously decides when to store such lessons via the prompt-based routing policy.

The stored chunks capture genuine algorithmic insights (e.g., “carefully consider overlapping intervals,” “ensure the algorithm correctly identifies the longest valid substring”). However, they are too sparse (~ 20 – 30 chunks by step 200) to reliably match dissimilar test problems, introducing noise into the generation context rather than helping. We therefore interpret the current RAG underperformance as a *scale limitation*, not a conceptual failure of external memory: once the memory reaches a threshold where it covers broader algorithm families and repeated motifs, retrieval should become more likely to surface transferable lessons and improve generation.

Failure analysis. With 16 and 32 recursive attempts, we observe performance degradation despite improved solution discovery. Inspection of training logs reveals that with more attempts, the model increasingly generates solutions that pass test cases by hardcoding expected outputs for specific inputs rather than fixing the underlying algorithmic logic. The recursive feedback loop amplifies this pattern. Addressing this will require richer feedback mechanisms, potentially incorporating chain-of-thought reasoning about *why* a solution is wrong rather than simply reporting which tests fail.

5 Conclusion

We presented a recursive self-improvement framework for code generation that combines self-distilled policy optimization with externalized knowledge storage under a prompt-based routing policy. The entire self-improvement process requires only a verifier providing rich structured feed-

back; no golden solutions, human demonstrations, or external teachers are needed. The model generates solutions, recursively refines them, discovers correct approaches through trial-and-error, and distills this self-discovered knowledge into its own parameters.

Our experiments on LiveCodeBench demonstrate +6 pp solve rate improvement through pure self-improvement, but reveal important constraints: (a) more recursive attempts paradoxically improve training-time solution discovery while degrading evaluation performance, (b) externalized memory via RAG introduces noise at small scale, and (c) a minimum base capability is required for the self-improvement loop to function (the 1.5B model actively degrades). These findings highlight both the promise and the challenges of recursive self-improvement: the model can genuinely teach itself, but the quality of self-generated training signal is as important as its quantity.

Team Contributions

Aaditya Nalawade: Led the SDPO implementation including the self-distillation update, top- k KL divergence loss computation, reprompted teacher design, and EMA teacher management. Ran training experiments on Modal H100 infrastructure.

Ethan Goodhart: Built the main training loop, model architecture, and generation infrastructure. Coordinated experimental design across attempt-count and model-scale ablations. Led analysis and paper writing.

Chandra Suda: Implemented the RAG knowledge store, prompt-based routing policy, and verification pipeline. Designed and ran the evaluation framework with ablation configurations across checkpoints.

References

- [1] James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dhharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *CoRR*, abs/1612.00796, 2016.
- [2] Idan Shenfeld, Mehul Damani, Jonas Hübötter, and Pulkit Agrawal. Self-distillation enables continual learning. *arXiv preprint arXiv:2601.19897*, 2026.
- [3] Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 Technical Report. *arXiv e-prints*, January 2025. arXiv:2412.15115 [cs].
- [4] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.
- [5] Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning

- converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024.
- [6] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping Reasoning With Reasoning. *arXiv e-prints*, page arXiv:2203.14465, March 2022.
 - [7] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing Large Language Models with Long-Term Memory. *arXiv e-prints*, page arXiv:2305.10250, May 2023.
 - [8] 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. *arXiv e-prints*, page arXiv:2005.11401, May 2020.
 - [9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *CoRR*, abs/1702.08734, 2017.
 - [10] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. page arXiv:1908.10084, August 2019.
 - [11] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *arXiv e-prints*, page arXiv:1711.05101, November 2017.
 - [12] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv e-prints*, page arXiv:2403.07974, March 2024.