Scaling Code-Assisted Chain-of-Thoughts and Instructions for Model Reasoning

Honglin Lin[1,2*], **Qizhi Pei**[2,4*], **Zhuoshi Pan**[2,3], **Yu Li**[2], **Xin Gao**[1,2], **Juntao Li**[5], **Conghui He**[2], **Lijun Wu**[2†]

[1]Shanghai Jiao Tong University [2]Shanghai Artificial Intelligence Laboratory
[3]Tsinghua University [4]Renmin University of China [5]Soochow University
{linhonglin,wulijun}@pjlab.org.cn

https://github.com/LHL3341/Caco

https://huggingface.co/datasets/LHL3341/Caco-1.3M

Abstract

Reasoning capability is pivotal for Large Language Models (LLMs) to solve complex tasks, yet achieving reliable and scalable reasoning remains challenging. While Chain-of-Thought (CoT) prompting has become a mainstream approach, existing methods often suffer from uncontrolled generation, insufficient quality, and limited diversity in reasoning paths. Recent efforts leverage code to enhance CoT by grounding reasoning in executable steps, but such methods are typically constrained to predefined mathematical problems, hindering scalability and generalizability. In this work, we propose Caco (Code-Assisted Chain-of-ThOught), a novel framework that automates the synthesis of high-quality, verifiable, and diverse instruction-CoT reasoning data through code-driven augmentation. Unlike prior work, Caco first fine-tunes a code-based CoT generator on existing math and programming solutions in a unified code format, then scales the data generation to a large amount of diverse reasoning traces. Crucially, we introduce automated validation via code execution and rule-based filtering to ensure logical correctness and structural diversity, followed by reverse-engineering filtered outputs into natural language instructions and language CoTs to enrich task adaptability. This closed-loop process enables fully automated, scalable synthesis of reasoning data with guaranteed executability. Experiments on our created Caco-1.3M dataset demonstrate that Caco-trained models achieve strong competitive performance on mathematical reasoning benchmarks, outperforming existing strong baselines. Further analysis reveals that Caco's code-anchored verification and instruction diversity contribute to superior generalization across unseen tasks. Our work establishes a paradigm for building self-sustaining, trustworthy reasoning systems without human intervention.

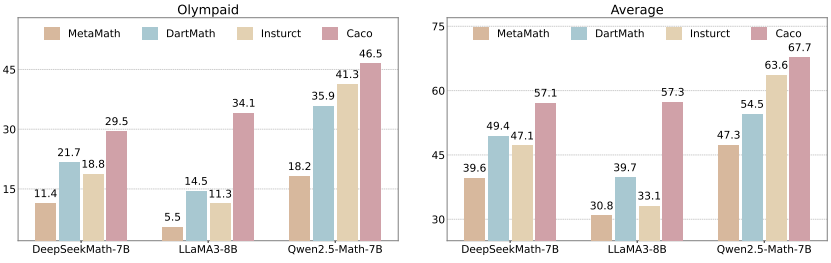

Figure 1: Overview of Caco results. Caco shows superior performance on Olympiad Bench and on average than baseline methods.

[*] Equal contribution. [†] Corresponding author.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

1 Introduction

The advent of Large Language Models (LLMs) [44, 32, 35] has revolutionized domains requiring complex reasoning, such as mathematics, code, and algorithmic problem-solving [42, 15, 1]. Recent LLMs demonstrate remarkable capabilities in generating step-by-step solutions through Chain-of-Thought (CoT) [42, 25, 45] prompting, where intermediate reasoning steps are explicitly articulated before final answers. This paradigm has become instrumental in tasks like mathematical problem solving and program synthesis, where systematic logic decomposition is critical. A prevalent strategy involves generating long CoT sequences [29, 12, 36, 37] to mimic human-like deliberation.

However, these CoT approaches predominantly rely on natural language reasoning traces, which suffer from several limitations. (1) *Unverifiability*, since natural language reasoning is not executable, errors in intermediate steps may propagate and lead to incorrect conclusions; (2) *Scalability constraints*, high-quality CoT data typically requires manual annotation, making it difficult to scale to diverse problem domains.

To address these issues, recent works have explored code-assisted reasoning [11, 41, 23, 10], where reasoning steps are grounded in executable code snippets (e.g., Python codes or algorithm sketches). By translating natural language logic into formal code, these methods enable automatic verification through code execution. Preliminary studies [11] demonstrate that code-verified CoT can reduce hallucination and improve answer accuracy. However, existing implementations struggle to generalize beyond predefined mathematical problems, limiting their adaptability and scalability [39, 41, 10].

In this work, we introduce `Caco`, a scalable code-assisted CoT and instruction generation framework designed to automate the production of high-quality reasoning training data through code-anchored refinement. A core innovation of `Caco` lies in its fine-tuning of a base LLM on a compact set of structured code CoT demonstrations, enabling the model to learn systematic code reasoning solutions. Leveraging this fine-tuned LLM, we generate large-scale candidate code-based CoT solutions, which are subsequently refined via an automated verification engine. This engine executes code snippets, verifies logical consistency, and enforces diversity in reasoning patterns. Finally, the validated code solutions are translated back into natural language instructions and the corresponding language CoTs, yielding instruction-aligned data pairs that establish bidirectional alignment between code and textual reasoning paths. The `Caco` generated natural language CoT offers several advantages. (1) *Scalability*: Through these model-generated synthetic code CoTs, we eliminate reliance on manual annotation of the aligned language CoTs, enabling the creation of millions of high-quality reasoning traces (e.g., our `Caco`-1.3M dataset); (2) *Verifiability*: Not only are the answers guaranteed to be correct for the augmented instructions, but the executable and automatic validation of intermediate steps of Code CoTs also ensures the aligned language CoTs to be correct solutions. (3) *Diversity*: By harnessing the fine-tuned LLM's generative capacity and sampling mechanism, `Caco` produces varied reasoning paths as well as the instructions, enhancing generalization across different problem types.

We evaluate `Caco` through extensive experiments on standard mathematical reasoning benchmarks. Models fine-tuned using our `Caco`-1.3M dataset achieve strong competitive performance; for example, attaining 92.6% accuracy on GSM8K and 82.4% on MATH, significantly outperforming prior approaches. `Caco` also exhibits strong generalization, the trained model maintains 67.7% accuracy on average over multiple benchmarks, surpassing comparable methods by a margin exceeding 7.9%. Further analysis confirms that `Caco`-generated CoT data preserves high diversity and scalability. Beyond advancing superior performance in mathematical reasoning, our work establishes a generalizable framework for developing self-improving and verifiable LLMs across algorithmic domains.

2 Related Work

2.1 Data Augmentation for Mathematical Reasoning

A wide range of recent efforts have explored different strategies for constructing instruction-tuning datasets tailored to mathematical reasoning [41, 20, 46]. For example, WizardMath [25], MetaMath [45], Orca-Math [28], MMIQC [21], and MathFusion [30] enhance answers and rationales for seed problems through prompt engineering and reinforcement learning techniques. KPMath [16], MathScale [34], and ScaleQuest [8] generate new problems from scratch by extracting mathematical concepts and topical structures. MAmooTH2 [47] and Numina-Math [18] construct instruction-tuning datasets by collecting and curating large-scale data from the web. DART-Math [38] applies rejection

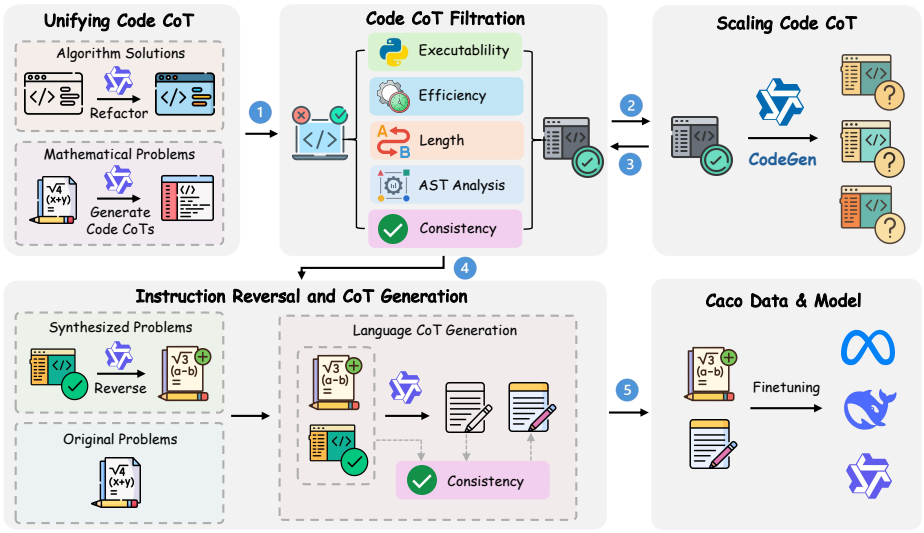

Figure 2: An overview framework of Caco data generation, including unifying Code CoT, scaling Code CoT with CodeGen, and instruction reversal and language CoT generation.

sampling based on problem difficulty to ensure the quality of generated solutions. Caco also falls within this scope and uses code as a scalable medium to generate diverse mathematical problems.

2.2 Code Integration for Enhanced Reasoning

LLMs often make calculation errors in complex mathematical reasoning (e.g., computing eigenvalues) when using CoT prompting [3, 9]. To address this, methods such as Program of Thoughts (PoT) [3], Program-Aided Language models (PAL) [9], and Code-based Self-Verification (CSV) [50] are proposed to prompt LLMs to generate executable code, leveraging external code interpreters for accurate computation. As open-source models improve, code-integrated data for post-training has gained attention. OpenMathInstruct-1 [39], TORA [10], MathCoder [41, 24], MegaMath [51], and DotaMath [17] embed code within natural language, enabling more robust reasoning. MAmmoTH [46] introduces MathInstruct, a hybrid of CoT and PoT datasets, allowing for different reasoning strategies for different problems. rStar-Math [11] generates paired natural language rationales and Python code, keeping only verified executable steps. CodeI/O [19] distills diverse reasoning patterns embedded in code by transforming it into a code input-output prediction format. MathGenie [23] synthesizes math problems and code-integrated solutions through solution augmentation, question back-translation, and verification-based filtering. Unlike previous works, our Caco leverages a code generation model to ensure both *scalability* and *verifiability*, while additionally introducing algorithmic problem types to promote greater *diversity* in problem coverage.

3 Method

Overview. Figure 2 presents the overall framework of Caco. We begin by abstracting each problem's solution into an executable code template. Based on this, we fine-tune a problem generation model (CodeGen) to learn diverse reasoning strategies by extending these templates. Sampling from the trained model yields a large number of new programs, each representing a unique solution pattern for a particular family of problems. Each code pattern is back-translated into concrete mathematical problems and corresponding step-by-step solutions. Only the instances where the natural language answer matches the code output are retained. Details regarding the prompt formulations, model specifications, and the criteria for filtering Code CoTs are provided in Appendix A.

3.1 Unifying Code CoT

To improve the quality, consistency, and verifiability of CoT reasoning for math problems, we explore a unified Code CoT representation. Motivated by prior findings that code data can enhance mathematical reasoning in language models [4, 46], we collect and standardize Code CoTs from both

mathematical and algorithmic domains. Specially, we use a general LLM $G_{p\to c} : \mathcal{P} \to \mathcal{C}$ to map each problem p to code c and an executor F that returns the correct answer a^* upon running c. We retain only verified traces; namely, the seed set is $\mathcal{C}_{\text{seed}} = \{\, G_{p\to c}(p) \mid F(c) = a^* \,\}$. This unified representation not only improves interpretability and execution fidelity but also lays the groundwork for scalable data generation and model training.

Mathematical Problems. We collected a broad set of mathematical problems from multiple sources to ensure diversity, such as the MATH dataset [14] (7.5K), DeepScaleR [26] (40K), and BigMath [2] (251K). These problems vary in complexity and format; some are accompanied by natural language CoT explanations, while others are not. To unify their representation, we convert each solution into a structured Python program following a generic template (See Prompt 1). This template encodes problem inputs as dictionaries and defines problem-solving logic through explicit function calls. It supports a wide range of reasoning types—arithmetic, algebraic, geometric, probabilistic—while enabling direct execution for correctness verification.

For example, consider the problem:

> *George has an unfair six-sided die. The probability that it rolls a 6 is $\frac{1}{2}$, and the probability that it rolls any other number is $\frac{1}{10}$. What is the expected value of the number shown when this die is rolled?*

We transform its solution into the following code representation:

```
def expected_value(probabilities, values):
    return sum(p * v for p, v in zip(probabilities, values))

probabilities = [1/10, 1/10, 1/10, 1/10, 1/10, 1/2] # Probabilities for 1, 2, 3, 4,
    5, 6
values = [1, 2, 3, 4, 5, 6] # Values on the die

input = {"probabilities": probabilities, "values": values}
output = expected_value(**input)
print(output)
```

This standardized representation ensures structural consistency across different problem types and facilitates easier interpretation by both models and humans.

Algorithmic Problems. In parallel, we incorporate algorithmic problems as an additional source of structured reasoning. We sample 40K problems from the Kodcode [43] dataset, covering key algorithmic domains such as sorting, searching, and dynamic programming. These problems typically come with code-level solutions and brief natural language comments, providing a native form of Code CoT. To ensure consistency across data sources, we normalize all algorithmic solutions into the same Python-based template used for mathematical problems. This standardization enables joint training and evaluation under a single format. The conversion prompts are described in Prompt 2.

Unified Seed Code CoTs. After Code CoT generation, we perform rigorous post-processing to ensure quality. Following the procedure described in Section A.4, we validate each code sample through execution: only programs that run successfully, produce correct outputs, and conform to the standardized format are retained. This filtering yields a curated seed corpus of 146K high-quality Code CoT instances (122K Math + 24K Code). Among these, 109K problems originally had solutions, which we refer to as Seed109K in the experiments. The resulting dataset provides a robust foundation for training models to enable effective generation of verifiable and scalable CoT reasoning in executable form in Section 3.2.

3.2 Scaling Code CoT with CodeGen

To scale the generation of high-quality code-based reasoning chains, we leverage the seed Code CoT dataset introduced previously to train a dedicated Code CoT generation model, `CodeGen`, so as to enable automated synthesis of executable, diverse, and logically coherent Code CoTs at scale. By training a model to internalize the structure and logic of our unified format, we facilitate the creation of new reasoning traces without relying on costly human annotations or handcrafted solutions.

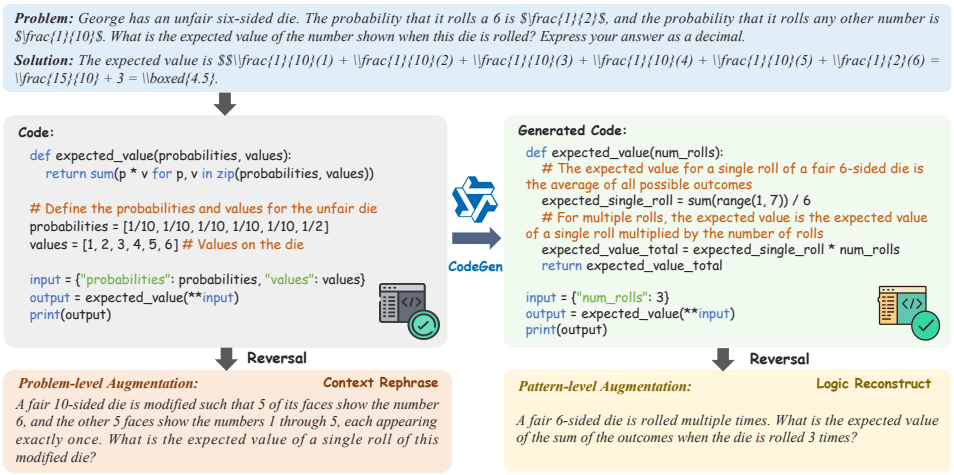

Figure 3: A case of one problem with its Code CoT. We demonstrate two augmentations, where problem-level augmentation refers to the original Code CoT can be back-translated into multiple question variants, and pattern-level augmentation means our `CodeGen` is capable of generating novel Code CoTs that generalize beyond the original seed patterns.

Training CodeGen on Unified Code CoTs. We fine-tune a unconditional `CodeGen` U_θ on $\mathcal{C}_{\text{seed}}$ to model the distribution of valid reasoning programs.

$$\min_{\theta} \mathcal{L}(\theta) = \sum_{c \in \mathcal{C}_{\text{seed}}} \sum_{t=1}^{|c|} \log p_\theta\big(c_t \mid c_{<t}\big). \tag{1}$$

The resulting model, `CodeGen`, is designed to generalize the reasoning patterns embedded in our dataset and produce structurally consistent code-based CoTs for both mathematical and algorithmic problems. Fine-tuning is conducted using a pretty simple prompt described in Appendix Table 3. Notably, training uses only Code CoTs, without problem contexts or requirements, focusing on internalizing the reasoning trace space rather than specific problem-solution pairs. In this way, we aim to largely explore the diverse Code CoTs in the generation phase.

Large-Scale CoT Generation via Sampling. After fine-tuning, we employ `CodeGen` to generate a large number of new Code CoT samples $\mathcal{C}_{\text{samp}} = \{c' \sim U_\theta\}$. Using temperature sampling, we generate multiple candidate programs use prompt in Appendix Table 4 (same as training `CodeGen`). This sampling-based approach introduces stochasticity into the decoding process, allowing the model to explore a diverse set of reasoning paths and solution strategies. The result is a scalable and flexible pipeline for synthesizing varied Code CoTs.

As illustrated in Figure 3, even for problem types the model has seen during training—such as calculating the expected value of a biased die—the model is capable of ***restructuring the logic***, e.g., by decomposing the problem into multiple rolls and aggregating expected outcomes. This demonstrates that `CodeGen` supports two complementary modes of augmentation:

Problem-level Augmentation arises when natural language problems are synthesized from code by varying the situational context or rephrasing the same underlying logic in different stylistic forms. This introduces diversity in surface formulations (implemented in Section 3.3).

Pattern-level Augmentation arises when the CodeGen explores novel reasoning structures—such as problem decompositions or alternative solution strategies—thereby enriching the pool of underlying logic templates.

Together, these modes yield both surface-level diversity and deeper structural variability in the synthesized dataset. Additional representative samples along with training settings and sampling configurations are provided in Appendix A.

Code CoT Filtration. To ensure the quality of generated Code CoTs, we apply the execution-based filtering criteria similar to the unified seed Code CoTs. The only difference is that at this stage, we do not enforce output matching with known answers, as consistency verification is deferred to the later back-translation stage. In total, we synthesize approximately 5.3M Code CoT samples. After filtering, we retain a high-quality subset of around 4.6M executable and structurally valid programs.

This large-scale dataset forms the basis for the subsequent stage of problem synthesis, enabling us to bootstrap new question–solution pairs and further expand the reach of code-based reasoning.

3.3 Instruction Reversal and Language CoT Generation

Following the generation of a substantial set of executable code templates, we distill their underlying logic to synthesize natural language problems alongside their corresponding solutions, derived from the combined set $\mathcal{C}_{\text{seed}} \cup \mathcal{C}_{\text{samp}}$. As shown in Figure 3, this process significantly expands our dataset by producing diverse and high-quality problem-answer pairs.

Two-Stage QA Generation. For quality control, we adopt a two-stage method to generate problem and language CoT instead of one-step generation for both. In our preliminary experiments, we find that jointly generating the instruction and language CoT together based on the Code CoT is easy to lead to low quality or incorrect language solutions, perhaps due to the 'lazy' mode [39] by LLMs since it sees the correct Code CoT as 'guidance'. Therefore, each code snippet is paired with representative input-output examples (code-instruction pair) and provided as input to the LLM (see Prompt 5), which generates a natural language problem at the first stage. Secondly, we prompt the generated problem to the LLM (see Prompt 6) for natural language CoT synthesis, which largely forces the LLM to think and generate correct language CoTs.

Dual Verification. Two filtration and verification ways are processed to ensure the correctness of the instruction and natural language CoT.

(1) *Answer Consistency*: We execute the code and compare its output to the answer inferred from the LLM's CoT reasoning. Any mismatches are discarded to maintain high precision.

(2) *CoT Consistency*: We remove samples where the language CoT and Code CoT for the same problem are not aligned, based on the consistency judgment in Prompt 7. This process ensures the correctness of the reasoning steps in the language CoT.

Only tuples (p', s', c') that simultaneously satisfy both conditions are retained. This filtering process can be formally expressed as:

$$\mathcal{D}_{\text{final}} = \{(p', s', c') \mid p' = G_{c\to p}(c'),\ s' = G_{p\to s}(p'),\ (\text{Ans}(s') = \text{Exec}(c')) \wedge \text{Con}(s', c')\}, \quad (2)$$

where G denotes a general-purpose LLM used for instruction reversal and answer generation; $\text{Ans}(s')$ extracts the final answer from the solution s' and compares it with the execution result of the code $\text{Exec}(c')$; and $\text{Con}(s', c')$ represents the CoT consistency check between the natural language solution and the code. After this pipeline, we obtain approximately 1.3M validated instruction-answer pairs in $\mathcal{D}_{\text{final}}$, which significantly enhance the diversity and reliability of the training data and serve as a valuable resource for downstream reasoning tasks.

4 Experiment

4.1 Experimental Setup

Baselines. We compare our `Caco`-generated dataset against several mainstream synthesized instruction-tuning datasets for math reasoning, including data-centric methods such as MetaMath [45], MMIQC [21], NuminaMath [18], MathFusion [34], RFT [38], and DART-Math [38], which all demonstrate strong reasoning enhancement. Besides, we also include well-known open-source instruction-tuned or reinforcement learning (RL)-based models as baselines: LLaMA3-7B-Instruct [49], Qwen2.5-Math-Instruct [44], and DeepSeekMath-7B-RL [32].

Training Configuration. To evaluate the generalizable effectiveness of our `Caco` produced dataset, our experiments are conducted on two math-specialized LLMs—DeepSeekMath-7B [32] and Qwen2.5-Math-7B [44], as well as one general-purpose model, LLaMA3-8B [49]. Unless otherwise specified, all models are fine-tuned for 3 epoch using a learning rate of 5×10^{-6}, a batch size of 128, and a cosine decay schedule with a warm-up ratio of 0.03. Additional implementation details are provided in Appendix B.

Evaluation Setup. Following the evaluation protocol of DartMath [38], we evaluate on multiple popular benchmarks to show the advantages, including MATH [14], GSM8K [6], CollegeMath [34], DeepMind-Mathematics [31], OlympiadBench-Math [13], and TheoremQA [5]. Solutions are

generated using greedy decoding with a maximum sequence length of 2048 tokens, and we report Pass@1 accuracy in the zero-shot setting without tool integration. Further evaluation details and benchmark statistics can be found in Appendix B.

4.2 Main Results

Model	# Samples	MATH	GSM8K	College	DM	Olympiad	Theorem	AVG
DeepSeekMath-7B (Math-Specialized Base Model)								
DeepSeekMath-7B-RL	-	51.1	**88.8**	34.5	58.2	18.8	30.9	47.1
DeepSeekMath-7B-MetaMath	400K	40.2	80.5	35.7	48.1	11.4	21.8	39.6
DeepSeekMath-7B-MMIQC†	2.3M	45.3	79.0	35.3	52.9	13.0	23.4	41.5
DeepSeekMath-7B-NuminaMath	860K	47.7	78.5	38.0	56.2	18.2	22.1	43.5
DeepSeekMath-7B-RFT†	590K	53.0	88.2	41.9	60.2	19.1	27.2	48.3
DeepSeekMath-7B-DartMath†	590K	53.6	86.8	40.7	61.6	21.7	32.2	49.4
DeepSeekMath-7B-MathFusion†	60K	53.4	77.9	39.8	65.8	23.3	24.6	47.5
Caco-Seed109K-DeepSeekMath-7B	109K	58.7	82.4	42.9	71.3	22.4	28.9	51.1
Caco-596K-DeepSeekMath-7B	596K	63.5	85.2	44.4	78.0	25.8	30.2	54.5
Caco-1.3M-DeepSeekMath-7B	1.3M	**68.2**	85.1	**46.0**	**80.2**	**29.5**	**33.8**	**57.1**
Qwen2.5-Math-7B (Math-Specialized Base Model)								
Qwen2.5-Math-7B-Instruct	-	82.1	**94.1**	50.4	72.9	41.3	40.8	63.6
Qwen2.5-Math-7B-MetaMath	400K	51.7	84.7	40.0	62.6	18.2	26.5	47.3
Qwen2.5-Math-7B-NuminaMath	860K	70.6	90.8	46.1	75.1	35.9	37.4	59.3
Qwen2.5-Math-7B-DartMath	590K	61.4	89.7	42.5	72.0	25.8	35.5	54.5
Qwen2.5-Math-7B-MathFusion	60K	75.2	83.5	43.0	76.0	39.5	41.5	59.8
Caco-Seed109K-Qwen2.5-Math-7B	109K	80.6	92.3	47.1	83.0	41.6	45.9	65.1
Caco-596K-Qwen2.5-Math-7B	596K	81.1	92.4	50.3	86.7	43.3	45.5	66.6
Caco-1.3M-Qwen2.5-Math-7B	1.3M	**82.4**	92.6	**51.4**	**87.1**	**46.5**	**46.0**	**67.7**
LLaMA3-8B (General Base Model)								
LLaMA3-8B-Instruct	-	44.3	53.4	29.8	42.0	11.3	17.7	33.1
LLaMA3-8B-MetaMath†	400K	32.5	77.3	20.6	35.0	5.5	13.8	30.8
LLaMA3-8B-MMIQC†	2.3M	39.5	77.6	29.5	41.0	9.6	16.2	35.6
LLaMA3-8B-NuminaMath	860K	43.6	79.7	24.7	43.1	16.4	19.9	37.9
LLaMA3-8B-RFT†	590K	39.7	81.7	23.9	41.7	9.3	14.9	35.2
LLaMA3-8B-DartMath†	590K	46.6	81.1	28.8	48.0	14.5	19.4	39.7
LLaMA3-8B-MathFusion†	60K	46.5	79.2	27.9	43.4	17.2	20.0	39.0
Caco-Seed109K-Llama3-8B	109K	55.3	86.0	42.2	52.0	19.1	25.6	46.7
Caco-596K-LLaMA3-8B	596K	64.3	88.6	44.8	66.7	24.7	27.6	52.8
Caco-1.3M-LLaMA3-8B	1.3M	**70.6**	**89.1**	**46.2**	**72.5**	**34.1**	**31.0**	**57.3**

Table 1: Performance comparison on mathematical benchmarks including MATH, GSM8K, CollegeMATH (College), DeepMind-Mathematics (DM), OlympiadBench-Math (Olympiad), and TheoremQA (Theorem). The best results are highlighted in **bold**. Baseline results labeled with † are derived from MathFusion [30].

Table 1 presents a comprehensive comparison of our Caco against a series of strong baselines across the three different base models (DeepSeekMath-7B, Qwen2.5-Math-7B, and LLaMA3-8B). We report results for two synthesized data sizes: Caco-596K and Caco-1.3M samples. From the results, we can summarize the following findings:

Consistent improvements across base models. Caco consistently outperforms existing methods across all three base models. For instance, on LLaMA3-8B, Caco-1.3M achieves an average score of 57.3, surpassing the previous best of 39.7 from DartMath [38] by a relative improvement of 44.3%.

Improvement over scaled synthetic data. Performance improves obviously when increasing the Caco-generated data from 596K to 1.3M. On Qwen2.5-Math-7B, Caco-1.3M achieves 67.7, outperforming Caco-596K by 1.1 and demonstrating the scalability and effectiveness of our approach under larger supervision.

Strong performance on challenging subsets. Notably, Caco shows superior performance on harder benchmarks such as OlympiadBench and TheoremQA, where other baselines struggle. For instance, on LLaMA-8B, Caco-596K improves OlympiadBench from 17.2 to 34.1 and TheoremQA from 20.0 to 31.0 compared to MathFusion, which shows the great potential of our approach.

Competitive with strong instruction-tuned and RL-based models. Remarkably, Caco matches or exceeds the performance of strong instruction-tuned or RL-finetuned models. For example, on Qwen2.5-Math-7B, Caco-1.3M achieves 67.7, which is comparable to Qwen2.5-Math-7B-Instruct (63.6). On DeepSeekMath and LLaMA series, Caco-1.3M trained models significantly surpass

DeepSeekMath-7B-RL (47.1) and LLaMA-8B-Instruct (33.1). This greatly demonstrates the superiority of our method.

Effectiveness of Caco Data. Compared to the seed data we used to train CodeGen (Caco-Seed-109K), Caco-596K and Caco-1.3M consistently deliver substantial improvements. For instance, on LLaMA3-8B, Caco-1.3M achieves 57.3, a significant increase from Caco-Seed-109K's score of 46.7. This validates our data scaling strategy, showing that our method yields performance gains by ensuring the training data comprehensively represents diverse and challenging problems.

5 Analysis

To further understand the strengths of our proposed approach, we analyze three key aspects that contribute to Caco's effectiveness: the *diversity*, the *scalability*, and the *verification* mechanism in the Caco data construction pipeline. Together, these components form the foundation of Caco's training methodology and help explain its strong performance across models and benchmarks. In the following sections, we provide a detailed analysis of each component and its contribution. More experiments and discussion of cost are in Appendix C and D.

5.1 Analysis on Data Diversity

We conduct a comprehensive investigation into the diversity of the dataset to assess the range and variability of the Caco-generated problems. This analysis is crucial for understanding how well the model can generate problems across various domains and ensure broad coverage of mathematical topics. By examining both the distribution of problems and the variety of problem types, we aim to demonstrate that the dataset not only spans a wide range of topics but also captures diverse problem-solving scenarios that are representative of real-world mathematical challenges.

Problem Diversity. We analyze the distribution of problems in the synthesized Caco dataset to assess its coverage and diversity. Specifically, we randomly sample 5K problems from Caco and compare them with samples from the original seed datasets (MATH, DeepScaleR and BigMath). We encode all problems using the all-MiniLM-L6-v2 sentence embedding model[2], and visualize their distributions via t-SNE [40], as shown in Figure 4a. The resulting plot demonstrates that Caco's synthesized data broadly and evenly spans the embedding space, effectively covering the original seed distributions. Notably, we observe a distinct region on the left side of the plot where Caco samples diverge from the seed data clusters, suggesting that our generation pipeline introduces novel and diverse problem types beyond the original datasets. This supports the claim that Caco enhances distributional generalization through its diverse synthetic augmentation.

Topic Diversity. To further assess the topical diversity of the Caco dataset, we apply clustering analysis to the problem embeddings. Using the same embedding method as before, we encode all problems and then apply the KMeans algorithm [27] to partition them into 12 distinct clusters. The clustering results are visualized in Figure 4b. The clusters reveal a wide range of mathematical and algorithmic topics, including algebra, geometry, applied mathematics, data structures, algorithms, and more. This confirms that Caco spans a broad spectrum of problem types, rather than concentrating on narrow domains. Representative samples from each identified topic cluster are provided in Appendix G for qualitative reference.

5.2 Analysis on Data Scalability

We evaluate the scalability of Caco by analyzing its impact on model performance under varying amounts of training data. Figure 5c presents the results on the MATH benchmark and the overall average across all benchmarks for DeepSeekMath-7B and LLaMA3-8B. For both models, we observe a clear upward trend as the training data size increases from 109K to 1.3M. This demonstrates the strong scalability of our approach. Notably, the performance gains are more pronounced for the general-purpose LLaMA3-8B, especially in the early stages (e.g., from Seed109K to 596K), highlighting Caco's ability to significantly improve less specialized models. On Qwen2.5-Math model, the performance also improves with increasing data size, but the improvement is less pronounced due to the already strong capabilities of the base model.

[2] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

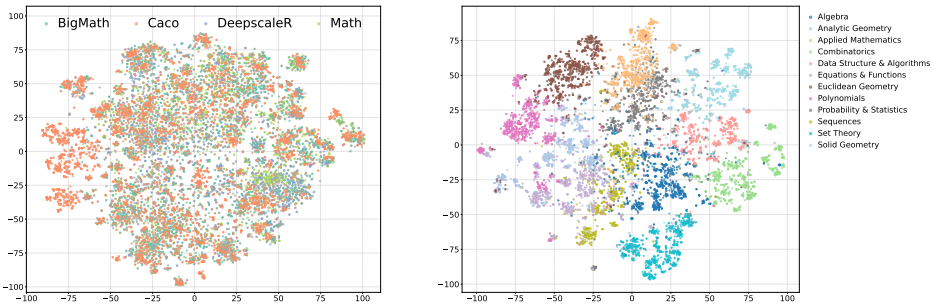

Figure 4: **Left**: Problem distribution of our Caco dataset and the original data sources. **Right**: KMeans clustering result of the problem types.

5.3 Ablation on Verification

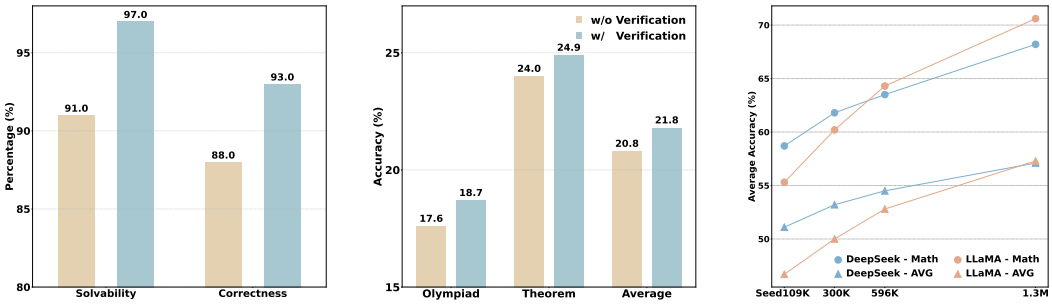

Figure 5: **Left**: Comparison of solvability and correctness between generated samples with and without verification. **Middle**: Accuracy comparison between models trained on verified and non-verified data. **Right**: Performance improvements of the Caco model as data size increases.

Verification is a crucial process in our Caco data generation. To further investigate the impact of verification on data quality and reliability, we compare the data with and without applying the verification filtering. We randomly sample 100K data points for each version and use Qwen3-32B to evaluate both the solvability (i.e., whether the problem can be solved) and the correctness (i.e., whether the final answer is accurate) of the generated samples. We further evaluate downstream performance by fine-tuning the LLaMA model on each dataset.

Impact on Data Quality. As shown in Figure 5a, the verification mechanism substantially improves the quality of the training data. With verification, the ratio of solvable problems increases from 91K to 97K, and the number of correct answers rises from 88K to 93K. These improvements suggest that the verification process—based on answer validation and consistency checks over reasoning chains—effectively filters out low-quality or incorrect samples, resulting in more reliable supervision.

Impact on Model Performance. In addition to improving data quality, verification also yields tangible benefits in downstream performance (Figure 5b). The model trained with verified data achieves an average accuracy of 21.8, compared to 20.8 without verification, reflecting a consistent improvement across benchmarks. The performance gain is especially notable on more challenging tasks: for instance, on Olympiad, the verified model scores 18.7, outperforming its non-verified counterpart by 1.1 points. This demonstrates that the enhanced data reliability introduced by verification translates into better generalization and reasoning robustness in trained models.

5.4 Generality Beyond Mathematics

We first evaluate the generalization of Caco models. Using OpenCompass [7], we assess Caco-1.3M models across a broad set of reasoning tasks, including mathematics (AIME24), code generation (HumanEval+), scientific QA (ARC-c), logic puzzles (BBH, KorBench), and general knowledge/science (AGIEval). Caco models demonstrate substantial improvements beyond math, with notable gains in logic puzzles, general reasoning, science reasoning, and code tasks. These results indicate that the models trained on Caco data generalize effectively across diverse benchmarks.

We next discuss the generality of the Caco methodology itself beyond mathematics. Although our primary experiments focused on mathematical reasoning, Caco is fundamentally a general-purpose

Model	AGIEval	AIME24	HumanEval+	ARC-c	BBH	KorBench	Average
Qwen2.5-Math-7B-base	42.5	20.0	12.8	72.2	19.9	39.7	34.5
Caco-1.3M-Qwen2.5-Math-7B	**53.3**	**23.3**	**53.1**	**81.4**	**65.1**	**47.1**	**53.9**
LLaMA3-8B-base	28.5	0.0	32.3	79.0	19.8	23.8	30.6
Caco-1.3M-LLaMA3-8B	**46.5**	**10.8**	**34.2**	**83.1**	**33.8**	**44.1**	**42.1**

Table 2: Performance comparison of base models and Caco-augmented models across diverse out-of-domain benchmarks.

Model	AGIEval	ARC-c	MMLU-STEM	Average
LLaMA-MegaScience-Seed5.2K	42.8	78.6	55.4	59.0
LLaMA-MegaScience-Caco37K	**45.0**	**84.8**	**60.5**	**63.4**

Table 3: Evaluation of LLaMA models trained on MegaScience seed data (5.2K) vs. Caco-augmented expansion (37K).

framework for structured, code-based reasoning, and is applicable to domains exhibiting *logical, symbolic, or programmatic* structure, such as logic puzzles, scientific reasoning, and procedural tasks. In logic puzzles, for instance, many problems share a reusable underlying reasoning template (e.g., arithmetic expression puzzles, countdown problems), which can be parameterized in code to generate diverse instances. This aligns with Caco's central principle: *code abstracts problem logic more compactly than natural language*, enabling systematic sampling and verification.

To test cross-domain applicability, we applied Caco to 5.2K science reasoning seeds from MegaScience. The pipeline generated 37K valid QA samples, and fine-tuning LLaMA on these yielded an average score improvement from 59.0 to 63.4 across AGIEval, ARC-c, and MMLU-STEM (Table 3). These results confirm that Caco's code-driven design enables effective extension to new domains where logic can be programmatically represented.

6 Conclusion

In this work, we present Caco, a code-assisted framework for generating high-quality, verifiable, and diverse chain-of-thought reasoning data. By leveraging code execution and automated filtering, Caco enables scalable synthesis of logically grounded instruction data without human supervision. Models trained with Caco outperform strong baselines on both mathematical reasoning benchmarks and out-of-domain benchmarks. Our findings highlight the effectiveness of code-driven verification and instruction diversity in improving reasoning generalization.

Acknowledgements

This work is supported by Shanghai Artificial Intelligence Laboratory.

References

[1] Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. Large language models for mathematical reasoning: Progresses and challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pp. 225–237, 2024.

[2] Alon Albalak, Duy Phung, Nathan Lile, Rafael Rafailov, Kanishk Gandhi, Louis Castricato, Anikait Singh, Chase Blagden, Violet Xiang, Dakota Mahan, et al. Big-math: A large-scale, high-quality math dataset for reinforcement learning in language models. *arXiv preprint arXiv:2502.17387*, 2025.

[3] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

[4] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

[5] Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. Theoremqa: A theorem-driven question answering dataset. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7889–7901, 2023.

[6] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[7] OpenCompass Contributors. Opencompass: A universal evaluation platform for foundation models. `https://github.com/open-compass/opencompass`, 2023.

[8] Yuyang Ding, Xinyu Shi, Xiaobo Liang, Juntao Li, Qiaoming Zhu, and Min Zhang. Unleashing reasoning capability of llms via scalable question synthesis from scratch. *arXiv preprint arXiv:2410.18693*, 2024.

[9] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023.

[10] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Minlie Huang, Nan Duan, Weizhu Chen, et al. Tora: A tool-integrated reasoning agent for mathematical problem solving. In *The Twelfth International Conference on Learning Representations*.

[11] Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*, 2025.

[12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[13] Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. *arXiv preprint arXiv:2402.14008*, 2024.

[14] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *Sort*, 2(4):0–6.

[15] Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 1049–1065, 2023.

[16] Yiming Huang, Xiao Liu, Yeyun Gong, Zhibin Gou, Yelong Shen, Nan Duan, and Weizhu Chen. Key-point-driven data synthesis with its enhancement on mathematical reasoning. *arXiv preprint arXiv:2403.02333*, 2024.

[17] Chengpeng Li, Guanting Dong, Mingfeng Xue, Ru Peng, Xiang Wang, and Dayiheng Liu. Dotamath: Decomposition of thought with code assistance and self-correction for mathematical reasoning. *arXiv preprint arXiv:2407.04078*, 2024.

[18] Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Huang, Kashif Rasul, Longhui Yu, Albert Q Jiang, Ziju Shen, et al. Numinamath: The largest public dataset in ai4maths with 860k pairs of competition math problems and solutions. 2024.

[19] Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. Codei/o: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316*, 2025.

[20] Honglin Lin, Zhuoshi Pan, Yu Li, Qizhi Pei, Xin Gao, Mengzhang Cai, Conghui He, and Lijun Wu. Metaladder: Ascending mathematical solution quality via analogical-problem reasoning transfer. *arXiv preprint arXiv:2503.14891*, 2025.

[21] Haoxiong Liu, Yifan Zhang, Yifan Luo, and Andrew C Yao. Augmenting math word problems via iterative question composing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 24605–24613, 2025.

[22] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[23] Zimu Lu, Aojun Zhou, Houxing Ren, Ke Wang, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. Mathgenie: Generating synthetic data with question back-translation for enhancing mathematical reasoning of llms. *arXiv preprint arXiv:2402.16352*, 2024.

[24] Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. Mathcoder2: Better math reasoning from continued pretraining on model-translated mathematical code. *arXiv preprint arXiv:2410.08196*, 2024.

[25] Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jian-Guang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, Yansong Tang, and Dongmei Zhang. Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct. In *The Thirteenth International Conference on Learning Representations*.

[26] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Tianjun Zhang, Li Erran Li, et al. Deepscaler: Surpassing o1-preview with a 1.5 b model by scaling rl. *Notion Blog*, 2025.

[27] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, volume 5, pp. 281–298. University of California press, 1967.

[28] Arindam Mitra, Hamed Khanpour, Corby Rosset, and Ahmed Awadallah. Orca-math: Unlocking the potential of slms in grade school math. *arXiv preprint arXiv:2402.14830*, 2024.

[29] OpenAI. Learning to reason with llms, 2024. URL `https://openai.com/index/learning-to-reason-with-llms`.

[30] Qizhi Pei, Lijun Wu, Zhuoshi Pan, Yu Li, Honglin Lin, Chenlin Ming, Xin Gao, Conghui He, and Rui Yan. Mathfusion: Enhancing mathematic problem-solving of llm through instruction fusion. *arXiv preprint arXiv:2503.16212*, 2025.

[31] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations*.

[32] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[33] Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, et al. Abstract syntax tree for programming language understanding and representation: How far are we? *arXiv preprint arXiv:2312.00413*, 2023.

[34] Zhengyang Tang, Xingxing Zhang, Benyou Wang, and Furu Wei. Mathscale: Scaling instruction tuning for mathematical reasoning. In *International Conference on Machine Learning*, pp. 47885–47900. PMLR, 2024.

[35] Mistral AI team. Learning to reason with llms, 2024. URL `https://mistral.ai/en/news/mathstral`.

[36] Qwen Team. Qwq: Reflect deeply on the boundaries of the unknown, November 2024. URL `https://qwenlm.github.io/blog/qwq-32b-preview/`.

[37] Qwen Team. Qwen3: Think deeper, act faster, April 2025. URL `https://qwenlm.github.io/blog/qwen3/`.

[38] Yuxuan Tong, Xiwen Zhang, Rui Wang, Ruidong Wu, and Junxian He. Dart-math: Difficulty-aware rejection tuning for mathematical problem-solving. *Advances in Neural Information Processing Systems*, 37:7821–7846, 2024.

[39] Shubham Toshniwal, Ivan Moshkov, Sean Narenthiran, Daria Gitman, Fei Jia, and Igor Gitman. Openmathinstruct-1: A 1.8 million math instruction tuning dataset. *Advances in Neural Information Processing Systems*, 37:34737–34774, 2024.

[40] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[41] Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. In *The Twelfth International Conference on Learning Representations*.

[42] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[43] Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. *arXiv preprint arXiv:2503.02951*, 2025.

[44] An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*, 2024.

[45] Longhui Yu, Weisen Jiang, Han Shi, YU Jincheng, Zhengying Liu, Yu Zhang, James Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. In *The Twelfth International Conference on Learning Representations*.

[46] Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. Mammoth: Building math generalist models through hybrid instruction tuning. In *The Twelfth International Conference on Learning Representations*.

[47] Xiang Yue, Tuney Zheng, Ge Zhang, and Wenhu Chen. Mammoth2: Scaling instructions from the web. *arXiv preprint arXiv:2405.03548*, 2024.

[48] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

[49] Di Zhang, Jiatong Li, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*, 2024.

[50] Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. In *The Twelfth International Conference on Learning Representations*.

[51] Fan Zhou, Zengzhi Wang, Nikhil Ranjan, Zhoujun Cheng, Liping Tang, Guowei He, Zhengzhong Liu, and Eric P Xing. Megamath: Pushing the limits of open math corpora. *arXiv preprint arXiv:2504.02807*, 2025.

A Data Generation

A.1 Prompts

We show the prompts used for Code CoT unifying in Prompt 1 and Prompt 2, and CodeGen training and sampling in Prompt 3 and 4. We also provide the back-translation prompt for question generation in Prompt 5 and answer generation prompt in Prompt 6. prompts for problem solvability, answer correctness, the consistency between the answer and the code's chain-of-thought (CoT) evaluations are displayed in Prompt 8, 9, and 7.

A.2 Model Usage

We detail the models employed at each stage of our pipeline:

- **Unifying Code CoTs:** We used Qwen2.5-72B-Instruct to generate unified Code CoTs.
- **CodeGen:** The unconditional CodeGen was fine-tuned from Qwen2.5-Coder-7B.
- **Problem Reversal & Solution Generation:** Both question back-translation and answer synthesis were performed using Qwen3-8B.
- **Evaluation:** All assessments (problem solvability, answer correctness, and CoT consistency) were conducted with Qwen3-32B.

A.3 Implementation Details

Focusing on Challenging Code CoT. To increase the difficulty of the dataset, we applied additional filtering to the largest subset, bigmath, of the Code CoT dataset. Based on the solve rate annotations provided with the dataset, we retained only those Code CoTs with a solve rate of less than 0.3.

Hyperparameters. During the Unifying Code CoT stage, we deployed Qwen2.5-72B-Instruct on 4 A100 GPUs to generate code from the raw datasets. For each sample, we performed a single pass of sampling with a temperature of 0.6.

For training the CodeGen model, we used the LlamaFactory framework and adopted the same training configuration as in the main experiments. During inference, we sampled with a temperature of 0.9 and a maximum sequence length of 1024 tokens.

For problem and solution generation, we followed the Qwen3-8B best practice [37]. Specifically, we used: `Temperature = 0.7, TopP = 0.8, TopK = 20, MinP = 0`, and `enable_thinking = False`.

We use Qwen3-32B for evaluating problem solvability, answer correctness, and the CoT consistency between the natural language solution and code CoT.

A.4 Filtering Mechanism for Code CoTs

As discussed in method section, many stages of our pipeline require rigorous filtering to ensure the quality, correctness, and executability of the generated Code CoTs. Here, we formally describe the filtering criteria used throughout our work.

- **Executability.** The code must be syntactically valid and executable without raising runtime errors. This ensures basic correctness and structural integrity.
- **Execution Efficiency.** To prevent degenerate or non-terminating programs, we discard any samples that exceed a 10-second execution time limit under a controlled runtime environment.
- **Minimum Code Length.** To avoid trivial or underdeveloped solutions, we require that each code snippet contain at least six non-comment lines of code. This encourages a minimal degree of reasoning complexity and explanatory depth.
- **AST-Based Semantic Validation.** Using abstract syntax tree (AST) analysis [33], we ensure that all variables declared in the input dictionary are functionally utilized in the program's logic. This discourages redundant or templated outputs and promotes semantically meaningful solutions.
- **Output Consistency.** When ground-truth answers are available, we verify that the program output exactly matches the expected solution. This check is applied in cases where reference answers are known and consistency can be reliably evaluated.

B Train and Evaluation

B.1 Training Setup

Model training was conducted using the LLaMA Factory [3] framework on 8 NVIDIA A100 GPUs. All models were trained for 3 epoch with a batch size of 128. We used the AdamW optimizer [22] with a learning rate of 5×10^{-6}, cosine learning rate decay, and a warm-up ratio of 0.03. The maximum sequence length (cutoff) was set to 4096, and the weight decay was 0.1. The prompt used for training is shown in Prompt 10.

B.2 Evaluation Setup

All models were evaluated using a unified framework [4] under the zero-shot setting. We used greedy decoding with a maximum generation length of 2048 tokens. The prompt used for evaluation is shown in Prompt 11.

B.3 Evaluation Benchmarks

The following datasets are used for evaluation:

- **MATH** [14]: A benchmark of 12,500 high school math competition problems, with 7,500 for training and 5,000 for testing. Problems are categorized into 7 topics (Prealgebra, Intermediate Algebra, Algebra, Precalculus, Geometry, Counting & Probability, and Number Theory) and 5 difficulty levels.
- **GSM8K** [6]: This dataset contains 8,792 high-quality grade school math word problems, with 7,473 for training and 1,319 for testing. Each problem typically requires 2 to 8 reasoning steps to solve.
- **CollegeMath** [34]: A test set containing 2,818 college-level math problems collected from 9 college textbooks, covering 7 core subjects: Algebra, Precalculus, Calculus, Vector Calculus, Probability, Linear Algebra, and Differential Equations.
- **DeepMind-Mathematics** [31]: This test set consists of 1,000 problems covering a wide range of mathematical reasoning tasks including algebra, arithmetic, calculus, and probability. It is designed to assess the mathematical reasoning abilities of models.
- **OlympiadBench-Math** [13]: A benchmark of 675 Olympiad-level math problems. We evaluate only on the English text-only subset of OlympiadBench.
- **TheoremQA** [5]: A theorem-driven question-answering benchmark containing 800 problems grounded in 350 domain-specific theorems. It evaluates a model's ability to apply mathematical and scientific theorems across disciplines such as mathematics, physics, electrical engineering, computer science, and finance.

C Additional Experiments

C.1 Distinguishing Caco from Teacher Knowledge Transfer and STaR-style Self-Improvement

A natural concern is whether Caco's performance gains stem primarily from knowledge transfer from the large teacher model (Qwen-2.5-72B-Instruct) used to generate the seed dataset, rather than from the Caco procedure itself. To isolate this factor, we conducted a control experiment in which the same teacher model was used to directly produce natural language Chain-of-Thought (CoT) answers for the same seed questions, resulting in a 300K QA dataset (QWEN72B-SEED-DISTILLED). We compared models fine-tuned on this dataset to those trained on a 300K subset of Caco (Caco-300K) and the full CACO-1.3M. Even at equal data size, Caco outperformed the distilled baseline (e.g., 66.2 vs. 65.5 AVG for Qwen-7B), and scaling Caco to 1.3M samples yielded further improvements (up to 67.7 AVG). This suggests that prompt and reasoning diversity, enabled by Caco's code-based augmentation, provides benefits beyond direct teacher distillation, and that Caco scales more effectively.

[3] https://github.com/hiyouga/LLaMA-Factory

[4] https://github.com/ZubinGou/math-evaluation-harness/tree/main

Another question is whether the gains could also be achieved by simpler self-improvement methods such as STaR [48]. Conceptually, Caco differs from these in that it does not focus on iteratively refining answers to a fixed set of questions; instead, it trains a dedicated code generator to produce executable CoTs from scratch, enabling scalable, verifiable creation of new problems. Nevertheless, to provide a direct comparison, we implemented a single iteration of STaR-style self-improvement on the seed dataset, generating multiple CoTs per seed question, filtering for correctness, and sampling 300K verified solutions (SEED-SELF-IMPROVE). Across both DeepSeek-Math-7B and Qwen-7B backbones, CACO-300K consistently outperformed SEED-SELF-IMPROVE by substantial margins (e.g., 53.2 vs. 48.9 AVG for DS-7B, and 66.2 vs. 52.6 for Qwen-7B). These results reinforce that Caco's improvements derive from its code-driven, diversity-oriented generation process, rather than simply inheriting knowledge from a stronger teacher or applying standard self-improvement on seed data.

Model	#Samples	MATH	GSM8K	College	DM	Olympiad	TheoremQA	AVG
Qwen-7B-Seed-self-improve	300K	70.7	83.0	47.1	47.6	39.0	28.2	52.6
Qwen-7B-Qwen72B-Seed-distilled	300K	79.0	91.2	52.1	84.4	41.3	45.0	65.5
Qwen-7B-Caco-300K	300K	81.6	92.4	51.2	84.8	42.5	44.9	66.2
Qwen-7B-Caco-1.3M	**1.3M**	**82.4**	**92.6**	**51.4**	**87.1**	**46.5**	**46.0**	**67.7**
DS-7B-Seed-self-improve	300K	53.1	86.7	41.6	62.5	19.3	30.2	48.9
DS-7B-Qwen72B-Seed-distilled	300K	57.4	83.0	42.4	69.0	23.4	31.4	51.1
DS-7B-Caco-300K	300K	61.8	83.2	43.3	76.0	23.9	31.1	53.2
DS-7B-Caco-1.3M	**1.3M**	**68.2**	**85.1**	**46.0**	**80.2**	**29.5**	**33.8**	**57.1**

Table 4: Control experiment comparing teacher-distilled natural language CoTs vs. Caco-generated data at matched size (300K) and at scale (1.3M). Bold indicates the best within each student block.

D Computational Cost and Efficiency

Stage	#Samples	Time (Hours)
Unifying Code CoT	339K	2h
Scaling Code CoT	5.3M	8h
Question Reversal	4.6M	5h
Answer Generation	4.6M	40h
Total (for 1.3M valid data)	–	**55h**

Table 5: Computation time for each stage in generating the CACO-1.3M dataset on a single 8×A100 machine.

To quantify the efficiency of our method, we report the full computational cost of generating the Caco-1.3M dataset (Table 5). All experiments were conducted on a single machine equipped with 8× NVIDIA A100 GPUs. The pipeline consists of four main stages: *unifying Code CoTs* (339K samples, 2h), *scaling Code CoTs* (5.3M samples, 8h), *question reversal* (4.6M samples, 6h), and *answer generation* (4.6M samples, 38.5h), totaling approximately 55 hours to produce 1.3M verified samples. Importantly, the entire process relies solely on *open-source* models, avoiding the substantial cost of proprietary API usage.

From a cost breakdown perspective, the majority of the runtime is consumed by the answer generation stage, which is *unavoidable* in any instruction tuning or self-improvement setup. For example, prior works such as DartMath [38] also incur comparable or higher costs in solution generation, particularly when sampling multiple candidate answers per prompt. The additional steps specific to Caco—Code CoT generation and question reversal—are lightweight (combined ∼16h), as natural language solutions are substantially longer than questions or Code CoTs.

Overall, these results demonstrate that Caco can generate over one million verified, diverse reasoning samples in under three days on a single 8-GPU node, highlighting its strong *scalability* and *accessibility*. While we acknowledge that data-efficient methods have their merits, Caco is designed with a complementary focus: producing *large-scale, diverse, and verifiable* reasoning data to support cross-domain generalization.

E Limitations

Although Caco demonstrates strong capabilities in generating diverse reasoning paths and instructions, its performance is still limited by the predefined problem types used during training. The system may struggle when faced with highly innovative or unconventional problems, particularly those that do not align with the templates or problem categories used during training. As a result, generating high-quality code-based CoTs for more complex or uncommon problem types remains a challenge, potentially leading to biases in the distribution of generated data.

Additionally, while the code can be executed accurately, converting it back into human-readable natural language instructions may result in the loss of some details or require simplification, causing the final output to be less rich or specific than the original reasoning steps.

Furthermore, the generated code is primarily used for filtering data and not for final training purposes. It helps ensure the correctness and consistency of the reasoning process, but does not directly contribute to the final training dataset. In future work, it will be essential to explore how the generated code can be used to further improve the quality of the data and enhance the training process.

Currently, Caco's application scope is focused mainly on mathematical and algorithmic reasoning tasks. Future work will need to explore extending it to broader domains, such as logical puzzles or STEM problem solving, which will require further effort.

F Future Works

We outline three complementary directions: increasing *difficulty*, expanding *diversity*, and leveraging Caco in *reinforcement learning (RL)*.

Raising Difficulty. Since the completion of this work, several math corpora with higher difficulty and quality than DEEPSCALER and BIGMATH have emerged, such as AM-THINKING-V1-DISTILLED and the DAPO dataset. Starting from harder, cleaner seed sets is likely to further amplify the benefits of code-based augmentation. Concretely, we plan to (i) replace/augment the seed pool with high-difficulty problems (e.g., Olympiad-style, exam-grade items) and (ii) adopt hardness-aware sampling and adversarial program mutations during Code CoT generation.

Expanding Diversity. As demonstrated in Section 5.4, our method generalizes beyond mathematics and applies naturally to domains with logical, symbolic, or procedural structure. While we discussed science and logic reasoning, a broader coverage (e.g., data reasoning, procedural planning, code debugging, diagram/physics problems, proofs) should allow CodeGen to learn richer templates and compose more diverse problems. Furthermore, extending the framework beyond Python to support formal languages (e.g., Lean, Coq, or Wolfram Language) could enhance rigor and verifiability. We will (i) train multi-domain CodeGen with domain tags, (ii) design compositional templates that factor shared subroutines across domains.

Applications: Reinforcement Learning with Verifiable Rewards (RLVR). Recent RL-based training has shown strong gains for reasoning models but often depends critically on the correctness of reference answers. Caco's executable traces provide a natural, low-noise reward signal. We will integrate Caco with RL by (i) deriving rewards from execution correctness (ii) employing a curriculum over program length and control-flow complexity. This combination targets scalable, verifiable RL training without heavy reliance on noisy external references.

G More Cases

This section presents samples from the Caco dataset, including the subsequence counting problem (Case 1), geometric sequences problem (Case 2), permutation and combination problem (Case 3), mathematical expression calculation problem (Case 4), and analytical geometry problem (Case 5).

Code Generation Prompt

Given the following math problem in natural language, provide the complete code solution that solves the problem.

Requirements:

- The final output of the program **must be the correct numerical or symbolic answer** to the problem.
- You must actually **compute** the result using Python code (e.g., using arithmetic or libraries like ‘sympy‘), **not just explain in text or comments**.
- The code must define an ‘input‘ dictionary, call a function using that input, assign the result to a variable ‘output‘, and finally ‘print(output)‘.
- Please provide a complete, standalone executable script.

Example Math Problem:

A snail is at the bottom of a 20-foot well. Each day, it climbs up 3 feet, but at night, it slips back 2 feet. How many days will it take for the snail to reach the top of the well?

Example Code Solution:

```
def days_to_reach_top(well_height, climb_distance, slip_distance):
    days = 0
    current_height = 0

    while current_height < well_height:
        current_height += climb_distance
        if current_height >= well_height:
            break
        current_height -= slip_distance
        days += 1

    return days + 1

# Represent the input as a dictionary named 'input'
input = {{"well_height": 20, "climb_distance": 3, "slip_distance": 2}}
# Call the function with the input dictionary, assign the result to 'output'
output = days_to_reach_top(**input)
# Print the output
print(output)
```

Now, please provide the code solution for the following math problem directly. Make sure your code solution defines the input as a dictionary named input, calls the solution function using this dictionary, stores the result in a variable named output, and prints output.

Math Problem:
{problem}

Solution (Optional):
{solution}

Code Solution:

Prompt 1: Code Generation Prompt for solving a math problem using Python code.

Code Unifying Prompt

Given the following code and test function, please refactor the solution into the required format:
Example Output Format:

```
def add(a, b):
    return a + b

# Represent the input as a dictionary named 'input'
input = {{"a": 3, "b": 5}}
# Call the function with the input dictionary, assign the result to 'output'
output = add(**input)
# Print the output
print(output)
```

<answer>8</answer>
Code:
{code}
Test Function:
{test_code}
Please refactor the code to follow the required format.

- The code must define an 'input' dictionary, call a function using that input, assign the result to a variable 'output', and finally 'print(output)'.
- If there are multiple test cases in test function, just select one of them.
- Please provide a complete, standalone executable script.

Output:

Prompt 2: Prompt for refactoring code into the required input-output format.

CodeGen Training Prompt

<|im_start|>system
You are a helpful assistant.<|im_end|>
<|im_start|>user
{code}<|im_end|>

Prompt 3: Prompt for training the CodeGen model.

CodeGen Sampling Prompt

<|im_start|>system
You are a helpful assistant.<|im_end|>
<|im_start|>user

Prompt 4: Prompt for sampling from the trained CodeGen model.

Question Back-translation Prompt

The code represents a solution to a math problem, and your task is to generate the original math problem that corresponds to the code.

Example Code:

```
def change_ref(amt, coins):
    if amt <= 0: return 0
    if amt != 0 and not coins: return float("inf")
    elif coins[0] > amt:
        return change_ref(amt, coins[1:])
    else:
        use_it = 1 + change_ref(amt - coins[0], coins)
        lose_it = change_ref(amt, coins[1:])
        return min(use_it, lose_it)

# Represent the input as a dictionary named 'input'
input = {"amt": 13, "coins": [1, 3, 5, 7]}
# Call the function with the input dictionary, assign the result to 'output'
output = change_ref(**input)
# Print the output
print(output)
```

Example Math Problem:

What is the minimum number of coins needed to make a total of 13 units using the available coin denominations of 1, 3, 5, and 7 units, each in unlimited supply? ### End Problem

Please generate **Math Problem** based on the following code. Ensure the generated problem is fully self-contained, solvable, and doesn't miss any necessary conditions or context.

You may add a concrete scenario or express the problem in different styles for diversity.

Code:

`{code}`

Math Problem:

Prompt 5: Question Back-translation Prompt. The prompt for generating a math problem based on a given code solution, where the generated problem should fully capture the conditions and context of the code.

Answer Generation Prompt

Instruction:
{problem}. Please reason step by step, and put your final answer within \boxed{}.
Response:

Prompt 6: Instructions for generating step-by-step reasoning and the final answer enclosed in a boxed format.

Consistency Checking Prompt

Solution:
{solution}
Code:
{code}
Please determine if the logic of the code and the chain-of-thought in the solution are consistent.
Answer with a single word: "Yes" or "No".
Answer:

Prompt 7: Prompt for checking the consistency between the logic of the code and the chain-of-thought in the solution, where the answer is expected to be either "Yes" or "No".

Solvability Checking Prompt

```
Problem:
{problem}
Please determine if the problem is solvable.
Answer with a single word: "Yes" or "No".
Answer:
```

Prompt 8: Prompt for determining the solvability of a given problem, where the answer is expected to be either "Yes" or "No".

Correctness Checking Prompt

```
Problem:
{problem}
Solution:
{solution}
Please determine if the chain-of-thought of solution is correct.
Answer with a single word: "Yes" or "No".
Answer:
```

Prompt 9: Prompt for verifying the correctness of the chain-of-thought in a given solution, where the answer is expected to be either "Yes" or "No".

Training Prompt

```
### Instruction:
{question}

### Response:
{solution}
```

Prompt 10: Training prompt of Caco model and baseline.

Evaluation Prompt

```
### Instruction:
{question}

### Response: Let's think step by step.
```

Prompt 11: Evaluation prompt of Caco model and baseline with step-by-step reasoning.

Subsequence Counting Problem

Problem:
How many unique subsequences can be formed from the list of numbers [1, 2, 3], where a subsequence is defined as any sequence that can be derived by deleting some or no elements from the list without changing the order of the remaining elements?

Code:

```
def days_to_reach_top(well_height, climb_distance, slip_distance):
    days = 0
    current_height = 0

    while current_height < well_height:
        current_height += climb_distance
        if current_height >= well_height:
            break
        current_height -= slip_distance
        days += 1

    return days + 1

# Represent the input as a dictionary named 'input'
input = {{"well_height": 20, "climb_distance": 3, "slip_distance": 2}}
# Call the function with the input dictionary, assign the result to 'output'
output = days_to_reach_top(**input)
# Print the output
print(output)
```

Solution:
We are asked to find the number of **unique subsequences** that can be formed from the list of numbers:

$$[1, 2, 3]$$

Step 1: Understand what a subsequence is
A **subsequence** is any sequence that can be derived by **deleting some or no elements** from the original list, **without changing the order** of the remaining elements.
For example, from [1, 2, 3], valid subsequences include: [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]
Also the empty subsequence (which is often considered, depending on the problem's context)
Step 2: Count all possible subsequences
Each element in the list can either be **included** or **excluded** in a subsequence. So for a list of length n, there are 2^n total possible subsequences (including the empty subsequence).
Here, the list has 3 elements, so the total number of subsequences is:

$$2^3 = 8$$

Step 3: List all the subsequences to verify
Let's list all 8 possible subsequences of [1, 2, 3]:
[] (empty subsequence), [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]
These are all the possible subsequences. All of them are **unique** because the elements are all distinct, and the order is preserved.
Final Answer:

$$\boxed{8}$$

There are **8 unique subsequences** that can be formed from the list [1, 2, 3].

Case 1: Subsequence Counting Problem: Counting the number of unique subsequences formed from a list of numbers.

Geometric Sequence Problem

Problem:
If the real numbers -1, a, b, c, -9 form a geometric sequence, then $b =$ _____.
Code:

```
from sympy import symbols, solve

def find_b(a, c):
    b = symbols('b')
    # Since a, b, c form a geometric sequence, b^2 = ac
    equation = b**2 - a*c
    solutions = solve(equation, b)
    # We need to determine the correct sign of b
    if solutions[0] * solutions[1] < 0:
        # If the product is negative, one of the solutions is negative
        # and the other is positive. We choose the one that is consistent
        # with the sequence -1, a, b, c, -9 being decreasing.
        if solutions[0] < 0:
            return solutions[0]
        else:
            return solutions[1]
    else:
        return solutions[0]

input = {"a": -1, "c": -9}
output = find_b(**input)
print(output)
```

Solution:
We are given that the real numbers -1, a, b, c, -9 form a **geometric sequence**. We are to find the value of b.
—
Step 1: Understand the structure of a geometric sequence
In a geometric sequence, each term is obtained by multiplying the previous term by a constant ratio r.
So, if the sequence is:

$$-1,\ a,\ b,\ c,\ -9$$

Then:
- $a = -1 \cdot r$
- $b = a \cdot r = (-1 \cdot r) \cdot r = -r^2$
- $c = b \cdot r = -r^2 \cdot r = -r^3$
- $-9 = c \cdot r = -r^3 \cdot r = -r^4$

—
Step 2: Use the last term to solve for r
We are told the last term is -9, so:

$$-r^4 = -9$$

Divide both sides by -1:

$$r^4 = 9$$

Now solve for r:

$$r = \pm \sqrt[4]{9}$$

Note that $\sqrt[4]{9} = \sqrt{\sqrt{9}} = \sqrt{3}$, so:

$$r = \pm\sqrt{3}$$

—
Step 3: Find $b = -r^2$
Recall that:

$$b = -r^2$$

Now compute r^2 for both values of r:
- If $r = \sqrt{3}$, then $r^2 = (\sqrt{3})^2 = 3$
- If $r = -\sqrt{3}$, then $r^2 = (-\sqrt{3})^2 = 3$

So in both cases:

$$b = -r^2 = -3$$

—
Final Answer:

$$\boxed{-3}$$

Case 2: Geometric Sequence Problem: Solving for b in a geometric sequence with given terms.

Counting and Combinatorics

Problem:
Using the digits 1 and 3 to form a four-digit number, and both digits 1 and 3 must appear at least once, there are a total of _____ such four-digit numbers.
Code:

```
from itertools import product

def count_four_digit_numbers():
    count = 0
    # Generate all 4-digit combinations of 1 and 3
    for combo in product([1, 3], repeat=4):
        # Ensure both 1 and 3 appear at least once
        if 1 in combo and 3 in combo:
            count += 1
    return count

# Represent the input as a dictionary named 'input'
input = {}
# Call the function with the input dictionary, assign the result to 'output'
output = count_four_digit_numbers(**input)
# Print the output
print(output)
```

Solution:
We are asked to find how many **four-digit numbers** can be formed using only the digits **1** and **3**, such that **both digits 1 and 3 appear at least once**.
—
Step 1: Total number of four-digit numbers using digits 1 and 3
Each of the 4 positions in the number can be filled with either **1** or **3**, so:

$$\text{Total combinations} = 2^4 = 16$$

So, there are **16** total four-digit numbers that can be formed using only digits 1 and 3.
—
Step 2: Subtract the cases where only one digit is used
We need to exclude the numbers that **do not contain both digits**. That is, the cases where:
- All digits are **1** $\rightarrow$ number: 1111 - All digits are **3** $\rightarrow$ number: 3333
So, there are **2** such numbers that we need to exclude.
—
Step 3: Compute the valid numbers

$$\begin{aligned}\text{Valid numbers} &= \text{Total combinations} - \text{Invalid combinations} \\ &= 16 - 2 = \boxed{14}\end{aligned}$$

—
Final Answer:

$$\boxed{14}$$

There are **14** four-digit numbers that can be formed using the digits 1 and 3, with both digits appearing at least once.

Case 3: Counting and Combinatorics problem: Counting the number of valid four-digit numbers formed by the digits 1 and 3 where both digits appear at least once. The table includes the problem description, step-by-step solution, and corresponding Python code to calculate the number of valid combinations.

Mathematical Expression Calculation

Problem:
Calculate:

$$\sqrt{12} + |1 - \sqrt{3}| + (\pi - 2023)^0$$

Code:

```
import math

def calculate_expression():
    sqrt_12 = math.sqrt(12)
    abs_value = abs(1 - math.sqrt(3))
    power_0 = (math.pi - 2023) ** 0
    result = sqrt_12 + abs_value + power_0
    return result

# Represent the input as a dictionary named 'input'
input = {}
# Call the function with the input dictionary, assign the result to 'output'
output = calculate_expression(**input)
# Print the output
print(output)
```

Solution:
We are asked to calculate the expression:

$$\sqrt{12} + |1 - \sqrt{3}| + (\pi - 2023)^0$$

Let's break it down step by step.

—

Step 1: Simplify $\sqrt{12}$
We can simplify the square root:

$$\sqrt{12} = \sqrt{4 \cdot 3} = \sqrt{4} \cdot \sqrt{3} = 2\sqrt{3}$$

—

Step 2: Simplify $|1 - \sqrt{3}|$
We know that $\sqrt{3} \approx 1.732$, so:

$$1 - \sqrt{3} \approx 1 - 1.732 = -0.732$$

Since the result is negative, the absolute value is:

$$|1 - \sqrt{3}| = \sqrt{3} - 1$$

—

Step 3: Simplify $(\pi - 2023)^0$
Any non-zero number raised to the power of 0 is 1. Since $\pi \approx 3.1416$, and $\pi - 2023 \neq 0$, we have:

$$(\pi - 2023)^0 = 1$$

—

Step 4: Add all the simplified parts
Now, add the three parts together:

$$\sqrt{12} + |1 - \sqrt{3}| + (\pi - 2023)^0 = 2\sqrt{3} + (\sqrt{3} - 1) + 1$$

Simplify:

$$2\sqrt{3} + \sqrt{3} - 1 + 1 = 3\sqrt{3}$$

—

Final Answer:

$$\boxed{3\sqrt{3}}$$

Case 4: Mathematical Expression Calculation: Solving the expression involving square roots, absolute values, and exponentiation. The table provides a detailed breakdown of the problem-solving steps and the corresponding Python code implementation.

Tangency Condition of Line and Parabola

Problem:
Given the line $x - y - 1 = 0$ and the parabola $y = ax^2$ are tangent to each other, find the value of a.
Code:

```
import sympy as sp

def find_tangent_coefficient():
    x, y, a = sp.symbols('x y a')
    line_eq = x - y - 1
    parabola_eq = y - a*x**2
    # Substitute y from the line equation into the parabola equation
    substituted_eq = parabola_eq.subs(y, x - 1)
    # Simplify the equation
    simplified_eq = sp.simplify(substituted_eq)
    # For the line and parabola to be tangent, the quadratic equation must have
        exactly one solution.
    discriminant = sp.discriminant(simplified_eq, x)
    # Solve for a
    a_value = sp.solve(discriminant, a)
    return a_value[0]

# Represent the input as a dictionary named 'input'
input = {}
# Call the function with the input dictionary, assign the result to 'output'
output = find_tangent_coefficient(**input)
# Print the output
print(output)
```

Solution:
We are given:
- A **line**: $x - y - 1 = 0$
- A **parabola**: $y = ax^2$
We are told that the **line and the parabola are tangent to each other**, which means they intersect at **exactly one point** (i.e., the system has exactly one solution).
—
Step 1: Solve the system of equations
We are given:
1. $x - y - 1 = 0 \Rightarrow y = x - 1$
2. $y = ax^2$
Substitute the expression for y from the line into the parabola:

$$x - 1 = ax^2$$

Rewriting:

$$ax^2 - x + 1 = 0$$

This is a **quadratic equation** in x:

$$ax^2 - x + 1 = 0$$

—
Step 2: Use the condition for tangency
For the line and the parabola to be **tangent**, the quadratic equation must have **exactly one solution**. That means the **discriminant** of the quadratic must be zero.
Recall that for a quadratic $ax^2 + bx + c = 0$, the discriminant is:

$$\Delta = b^2 - 4ac$$

In our case, the quadratic is:

$$ax^2 - x + 1 = 0$$

So:
$a = a, b = -1, c = 1$
Now compute the discriminant:

$$\Delta = (-1)^2 - 4(a)(1) = 1 - 4a$$

Set the discriminant equal to zero (for tangency):

$$1 - 4a = 0$$

Solve for a:

$$4a = 1 \Rightarrow a = \frac{1}{4}$$

Final Answer:

$$\boxed{\frac{1}{4}}$$

Case 5: Tangency Condition of Line and Parabola: Finding the value of a for the line and parabola to be tangent.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: In the abstract and Section 1, we claimed our contribution and scope.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: In Appendix E, we discuss the limitations of our method.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.

- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: Our work is not strictly a purely theoretical work.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We have provided detailed descriptions of the experimental model, data and implementation in Section 4.1 and Appendix B.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the

results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.

- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Most of the data we used are available in the public. we provide an anonymized url to share our code.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: All details are provided in the main paper and Appendix.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

Justification: In both the main paper and the appendix, we conducted extensive experiments. However, due to resource limitations, we do not report error bars.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: In Appendix B, we specify the computational resources used for the experiments.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We followed the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: The purpose of this paper is to explore data synthesis for LLM reasoning. Therefore, it does not pose any societal impacts.

    Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The creators or original owners of assets (e.g., code, data, models), used in the paper, have been properly credited and the license and terms of use have been explicitly mentioned and properly respected.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The paper introduces new dataset and models, both of which are well documented.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

Justification: The core method development in this research does not involve LLMs as any important, original, or non-standard components.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.