PuzzleClone: A DSL-Powered Framework for Synthesizing Verifiable Data

Anonymous ACL submission

Abstract

High-quality mathematical and logical datasets with verifiable answers are essential for strengthening the reasoning capabilities of large language models (LLMs). While recent data augmentation techniques have facilitated the creation of large-scale benchmarks, existing LLM-generated datasets often suffer from limited reliability, diversity, and scalability. To address these challenges, we introduce PuzzleClone, a formal framework for synthesizing verifiable data at scale using a novel DSL-driven approach. Our approach features three key innovations: (1) encoding seed puzzles into structured logical specifications, (2) generating scalable variants through systematic variable and constraint randomization, and (3) ensuring validity via a reproduction mechanism. Applying PuzzleClone, we construct PC-83K, a benchmark comprising over 83K diverse and programmatically validated puzzles. The generated puzzles span a wide spectrum of difficulty and formats, posing significant challenges to current state-of-the-art models. Experimental results show that post training (SFT and RL) on PC-83K yields substantial improvements not only on the testset but also on various logic and mathematical benchmarks. Post training raises average performance on PC-83K from 14.5 to 66.0 and delivers consistent improvements across 7 logic and mathematical benchmarks up to 18.4 absolute percentage points (SATBench from 51.6 to 70.0). Our code and data are available at https://github.com/puzzleclone.

1 Introduction

Large-language models (LLMs) have recently demonstrated impressive zero-shot and few-shot performance on a wide spectrum of reasoning tasks, yet consistently achieving robust logical reasoning remains an open challenge. To push the frontier, researchers have released a variety of mathematical-puzzle and formal-logic benchmarks that expose models to carefully crafted, high-difficulty problems. Unfortunately, the manual effort required to compose and validate such items has kept existing corpora relatively small and homogeneous, impeding further progress.

One promising approach to address this bottleneck is *data augmentation*, which involves generating new problems by systematically modifying existing "seed" instances (Feng et al., 2021; Lu et al., 2024). This strategy has the potential to vastly expand the size and diversity of reasoning datasets while reducing the manual effort involved in their creation. However, current augmentation pipelines largely rely on LLMs to annotate new problems, generate solutions, and verify the answers (Lu et al., 2024; Tan et al., 2024; Shah et al., 2024), which introduces several critical limitations. First, without a robust, end-to-end verification approach, it is hard to ensure the data reliability throughout the synthesis pipeline, leading to flawed, inaccurate, or biased data (Wang et al., 2024a). Second, existing pipelines lack formalization and disproportionately depend on the generative capabilities of the underlying LLMs, which threatens data diversity. For example, an individual model typically explores only a narrow range of variations in assumptions, conditions, parameters, and queries. This limited coverage reduces the dataset's ability to challenge and generalize LLM reasoning capabilities effectively. Finally, the substantial computational costs associated with involving LLMs in every step of the synthesis pipeline also severely constrain scalability (Wang et al., 2024a). Motivated by the need for a scalable path to unbounded yet trustworthy data creation, our work seeks a principled procedure that can, in principle, generate reliable reasoning data indefinitely.

In this work, we introduce PuzzleClone, a new data curation framework featuring a novel Domain Specific Language (DSL) capable of modeling complex puzzle intricacies, alongside a rig-

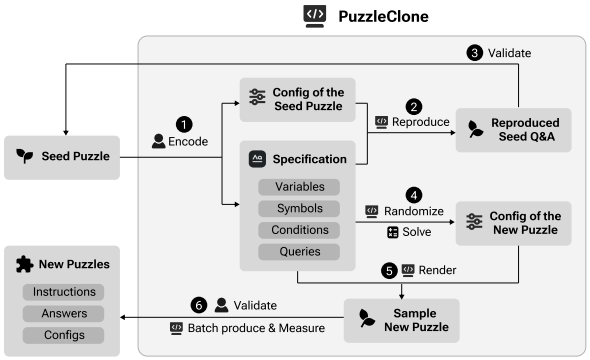

Figure 1: Overview of the PuzzleClone framework.

orous pipeline to ensure programmatic validity of solutions. Utilizing this framework, we construct PC-83K, a novel benchmark of 83,657 challenging, diverse, and fully verifiable puzzles, covering diverse puzzle types such as Satisfiability Modulo Theories (SMT), Linear Programming (LP), and classical logic puzzles like Sudoku. Figure 1 shows an overview of PuzzleClone. Each seed puzzle is first manually encoded into a structured problem specification that specifies the data synthesis pipeline, with a config file that contains the values of the parameters specific to the seed puzzle (❶). Based on them, an instance generator can automatically batch produce new questions with diverse mathematical configurations by systematically varying parameters and combinations of constraints with randomization (❹, ❺). Meanwhile, it can programmatically derive ground-truth answers for each generated instance with the assistance of symbolic or custom solvers (❹). To ensure fidelity, our pipeline includes a reproduction step that verifies the original seed questions can be regenerated from the DSL specifications (❷, ❸), thereby guaranteeing the accuracy of both problem formulations and associated answers.

Our experiments demonstrate that PC-83K poses substantial challenges for state-of-the-art large language models (LLMs), including ChatGPT-4o and DeepSeek-R1. We further use it to distill the reasoning capabilities of large models into small ones. After post-training, Qwen2.5-7B-Instruct improves from 14.5 to 66.0 on PC-83K, and achieves gains of up to 18.4 absolute percentage points across 7 logic and mathematical benchmarks.

2 PuzzleClone

The data synthesis process can be formalized as follows: For each input seed puzzle P^0, our goal is to generate a set of new puzzles $P^* = (I^*, A^*)$, where I^* is the set of puzzle instructions and A^* is the set of answers. The process comprises three stages: puzzle encoding, puzzle generation, and config-based validation.

2.1 Puzzle Encoding

Most puzzles can be conceptualized as a combination of a universal puzzle template, which encapsulates the core logic of the puzzle, and a set of parameters that define its specific details (Pan et al., 2023). For example, in the seed puzzle illustrated in Figure 2A, the underscored texts represent the puzzle-specific parameters while the rest describe the template. Thus, data augmentation can be viewed as systematically varying these parameters and embedding them into the universal template to generate new puzzles.

During puzzle encoding, each seed puzzle Q^0 is manually encoded into a structured problem specification Q_s and a configuration file Q_c. Contrary to prior works that rely on logic programming languages such as Prolog and SMT-LIB (Clocksin and Mellish, 2003; Pan et al., 2023; Barrett et al., 2010), we design a new domain-specific language (DSL) to describe Q_s, as shown in Figure 2B. This DSL not only captures the core puzzle logic but also encodes constraints that are implicit and only applicable to data augmentation, such as the parameters' value domains and the number of constraints. Furthermore, the DSL represents the puzzle in a more human-readable format while still machine-parsable for downstream processing. The DSL specification $Q_s = (V, S, C, Q, D)$ has the following components[1]:

Variables V: Parameters that can be varied to generate new puzzles. For instance, variables in the seed puzzle in Figure 2A include the number of students `s_num` and food types `f_num`, as well as their names `names` and `food`. Each variable v is characterized by its type (`type`) and value domain (`domain`), or a formula (`formula`) deriving it from other variables. For instance, `get_faker` is an internal function which leverages Faker[2] to generate random natural-language-based instance names. Finally, each variable is assigned a *difficulty factor* (`diff_factor`) that specifies how the variable's value contributes to the puzzle difficulty ($+1$, -1, 0 for positive, negative, and neutral effect.).

Symbols S: Quantities to be solved in the puzzle. Symbols of various types (`type`) can be created

[1] See Appendix C and D for DSL schema and puzzle specs.
[2] https://github.com/joke2k/faker

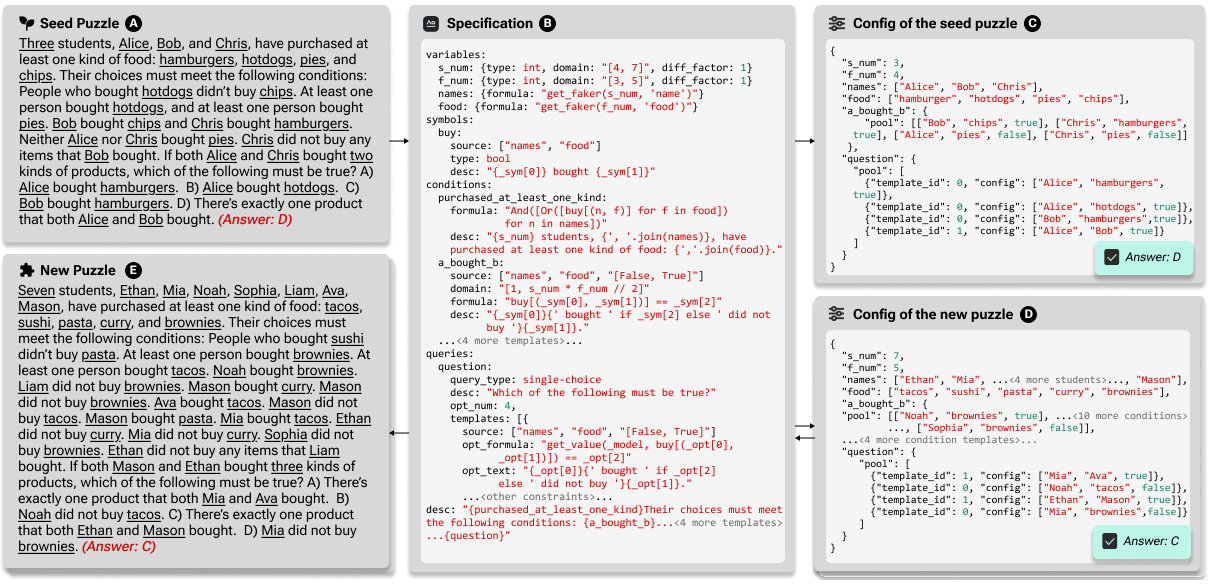

Figure 2: The data synthesis pipeline of PuzzleClone. (A) An example seed puzzle. (B, C) The DSL specification and the config encoded from the seed puzzle. (D) A randomly generated config produced by the puzzle generator for this DSL specification. (E) A new puzzle instance synthesized by the puzzle renderer using the DSL specification and the generated config.

by mapping existing variables (`source`), and each symbol can be binded with a natural language (NL) template (`desc`) that outlines its expression in the puzzle instructions. For instance, a set of boolean symbols buy are generated for each combination of students' and food names representing the purchase information, as depicted in Figure 2B.

Conditions C: A pool of all potential constraints that can be applied to the puzzle, including *static conditions*, which remain the same among all puzzles, and *dynamic conditions*, which are randomly generated from a template and can appear a variable number of times. For example, the statement "*[students] have purchased at least one kind of food*" in the seed puzzle in Figure 2A can be viewed as a static condition, while the subsequent text under "*their choices must meet the following conditions*" outlines dynamic conditions with reusable structures and mutable parameters for replication. Our DSL provides structural declarations for randomly generating such conditions. First, the dynamic parameters within each condition can be randomly determined according to the specified pool (`source`) and the allowable range for the number of instances of each condition type (`domain`). Additionally, every condition is associated with a formula template (`formula`) and a NL template (`desc`). The randomly chosen parameters are integrated into the templates to generate the complete condition.

Queries Q: The questions in the puzzle. For example, the `question` in Figure 2B presents a single-choice question (`query_type`) with four options (`opt_num`). The structure of these options is defined by two distinct templates (`templates`), each with its own parameter pool (`source`), formula and instruction template (`opt_formula`, `opt_text`).

Description D: An NL template assembling previous components for the eventual puzzle (`desc`).

In parallel with the specification, the puzzle-specific values of variables and the parameters within constraints and queries are extracted as a configuration file Q_c, as shown in Figure 2C. Thus, a puzzle can be uniquely determined by combining a specification and a configuration.

2.2 Puzzle Generation

After encoding the seed puzzle, a puzzle generator will translate the specification into new puzzles. Specifically, it first generates a Python script which automatically decides all variables, conditions, and queries by random while leveraging either built-in symbolic solvers (e.g., Z3 (De Moura and Bjørner, 2008) for SMT problems) or custom solvers to solve the puzzle. Thus, new puzzles can be generated by executing the script.

2.3 Config-based Answer Validation

One of the key goals of PuzzleClone is to keep each synthesis step verifiable for the reliability of the synthesized data. Since the puzzle encoding pro-

cess relies on careful manual effort, it is necessary to make sure the process is accurate, leading to our design of a config-based answer validator. Specifically, a validation script is generated in parallel with the puzzle generator, which is capable of reproducing the seed puzzle by acquiring the puzzle-specific values directly from the config instead of randomization. Thus, a deterministic reference answer can be computed and compared against the ground-truth solution of the seed puzzle, thereby verifying the correctness of the encoding process.

3 Dataset Construction

We constructed our dataset via a multi-stage process to ensure quality, diversity, and verifiability.

3.1 Seed Puzzle Collection and Curation

We started by collecting raw puzzles from diverse sources, including mathematics competitions for primary and secondary schools, Olympiad-style problems, logical reasoning books/blogs, and established puzzle benchmarks, e.g., Sudoku and combinatorial optimization problems (Mittal et al., 2024; Chen et al., 2025b; Li et al., 2025; Liu et al., 2025a). We then employed the Qwen2.5-72B-Instruct model with specially crafted prompts (see Appendix A.2) to select potential seed puzzles based on two key criteria:

- The problem contains variable values or logical constraints that can be replicated by random without altering its semantic validity.
- The puzzle is solvable using symbolic or custom solvers within tractable time and memory bounds, and a correct Python solution can be generated by the model.

3.2 Human Verification and Enhancement

The AI-filtered puzzles underwent rigorous manual review, during which we (1) conducted brainstorming sessions to enhance problem complexity through additional variables and novel constraints (e.g., `A1-ant` vs. `A2-athlete` in our dataset), (2) introduced diversified questioning approaches to increase variety, and (3) eliminated problems with conceptually similar solutions (e.g., those relying on the same underlying mathematical identity or solving trick) to ensure mathematical diversity.

This process yielded 86 high-quality seed puzzles, including 72 SMT-solvable and 14 pure formula/code-based (labeled as `A`-series).

3.3 Puzzle Generation and Deduplication

We utilized PuzzleClone to generate 1K variants for each seed puzzle, resulting in an initial set of 86K puzzle instances. For each generated variant, we assigned a unique identifier, its source puzzle, question type (`qtype`), and evaluation type (`eval_type`). The `qtype` indicates the format of the question, such as multiple-choice, fill-in-the-blank, or short answer. The `eval_type` specifies the answer structure and corresponding evaluation strategy, including formats like `numeral`, `nominal`, `option`, `ordered_array`, and `unordered_array`. Based on the assigned `qtype` and `eval_type`, we designed custom prompt wrappers (Appendix A.2) and implemented appropriate evaluation operators.

Since instances were generated independently and randomly, we deduplicated puzzles by comparing their logical `configs` (Figure 2D) rather than surface text (Figure 2E). Specifically, we ignored nominal fields (e.g., *names*) and treated value pools as unordered sets, allowing detection of logically equivalent puzzles despite textual variations (e.g., differences in wording or ordering).

After deduplication, we retained 83,657 unique puzzle instances. Detailed statistics of the resulting dataset—including the distributions of question types, answer types, and duplication patterns—are provided in Appendix B.

3.4 Difficulty Assessment and Analysis

To quantify puzzle complexity, we designed a heuristic difficulty score based on four features: the number of logical symbols (`sym_num`), the number of logical constraints (`cond_num`), the length of problem description (`desc_len`), and a variable-scale metric (`var_scale`) capturing the difficulty impact of variable domains. Details of the difficulty score computation and its empirical validation us-

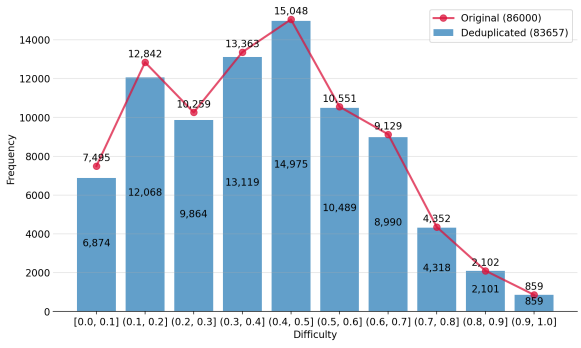

Figure 3: Puzzle difficulty distribution before and after deduplication.

ing model accuracy are given in Appendix A.1.

The difficulty distribution of the puzzles is shown in Figure 3. Hard puzzles are relatively rare due to increased likelihood of being unsolvable when the complexity and diversity (i.e., the variable values and constraint number) increase. Furthermore, duplicates were disproportionately found among easier puzzles due to their narrower randomization space.

3.5 Dataset Partitioning Strategy

To facilitate a comprehensive evaluation of popular LLMs while supporting both supervised fine-tuning (SFT) and reinforcement learning (RL) training, we implemented a stratified dataset partitioning strategy based on puzzle difficulty and seed origin. Specifically, puzzles were classified into *normal* (difficulty ≤ 0.5) or *hard* (difficulty > 0.5) categories. For each seed puzzle, its variants in each category were randomly split into 90% train and 10% test. From the training set, 25 *normal* and 25 *hard* samples were selected per seed puzzle (falling back to normal in case of insufficient hard samples) to form the SFT set ($86 \times 25 \times 2 = 4300$). A smaller subset (5 *normal* + 5 *hard* per seed) was selected for RL validation (860 in total), with the remaining training samples assigned to RL training.

	SFT	RL-Train	RL-Val	Test
Normal	2161	50738	430	5730
Hard	2139	23616	430	2713
Sum	4300	74354	860	8443

Table 1: Dataset partitioning statistics.

The final partitioning (Table 1) ensures comprehensive coverage, with each set containing puzzles derived from all 86 seed puzzles while maintaining difficulty stratification.

4 Experiments

4.1 Experimental Setup

All model training and inference were conducted on NVIDIA H100 GPUs. We trained our models using an 8×NVIDIA H100 GPU cluster, with Qwen2.5-7B-Instruct as the foundation model for all training experiments (detailed training paramter setting can be found in Appendix E).

4.2 Baseline Results

We evaluate a wide range of models on the PC-83K test set, including both open-source and proprietary models. Similar to the training test, the test set is also divided into *Normal* and *Hard* subsets to reflect different levels of reasoning complexity. As shown in Table 2, current LLMs remain limited in handling complex logical reasoning.

Model	Normal	Hard	Avg.
Proprietary Models			
ChatGPT-4o (OpenAI, 2024)	31.7	24.6	28.2
ChatGPT-o3 (OpenAI, 2025b)	87.1	83.4	85.3
ChatGPT-5 (OpenAI, 2025a)	**91.1**	**86.3**	**88.7**
Gemini-2.0-flash (Gemini Team, 2025a)	42.0	31.6	36.8
Gemini-2.5-pro (Gemini Team, 2025b)	75.8	67.2	71.5
Gemini-3-pro (Gemini Team, 2025c)	86.5	83.0	84.8
Claude-3.5-sonnet (Anthropic, 2024)	37.6	27.4	32.5
Claude-4-sonnet (Anthropic, 2025)	62.7	47.8	55.3
Seed1.6 (ByteDance, 2025)	87.8	82.4	85.1
GLM Series (THUDM, 2025)			
GLM-Z1-9B-0414	63.6	53.5	58.6
GLM-Z1-32B-0414	71.1	60.9	66.0
Qwen2.5 Series (Yang et al., 2024)			
Qwen2.5-7B-Instruct	16.8	12.1	14.5
Qwen2.5-14B-Instruct	24.3	17.9	21.1
Qwen2.5-32B-Instruct	31.4	23.5	27.4
Qwen2.5-72B-Instruct	32.8	25.3	29.0
Qwen3 Series (Yang et al., 2025)			
Qwen3-8B	71.6	59.4	65.5
Qwen3-14B	78.6	67.0	72.8
Qwen3-32B	77.0	68.1	72.5
Qwen3-235B-A22B	82.9	73.8	78.3
DeepSeek Series (Guo et al., 2025)			
DeepSeek-R1-Distill-Qwen-14B	47.9	38.4	43.1
DeepSeek-R1-Distill-Qwen-32B	53.3	43.2	48.3
DeepSeek-R1-0528-Qwen3-8B	76.0	66.8	71.4
DeepSeek-R1-0528	88.7	82.6	85.6

Table 2: Baseline performance on PC-83K

Among the open-source models, the Qwen and GLM-Z1 series demonstrate strong performance trends as model size increases. Within the Qwen2.5 family, scores steadily improve from the 7B (14.5 average) to the 72B (29.0 average). Qwen3-8B achieves an average of 65.5, already outperforming many larger models from the previous generation. The Qwen3-14B and Qwen3-32B models achieve average scores of 72.8 and 72.5, respectively, while Qwen3-235B-A22B reaches 78.3.

The GLM-Z1 models also exhibit strong results. GLM-Z1-9B achieves 58.6 on average, while GLM-Z1-32B outperforms it with a score of 66.0. These results suggest the GLM-Z1 series is competitive with Qwen3 in the same parameter range.

The DeepSeek-R1-Distill models based on Qwen show moderate performance (43.1 and 48.3), while the full DeepSeek-R1-0528-Qwen3-8B model attains a strong 71.4. The most powerful open-source model DeepSeek-R1-0528 achieves an impressive 85.6 average score, surpassing all other

Model	PC-83K†		PC-SL	Logic Benchmarks		Mathematical Benchmarks				
	Normal	Hard	35K†	SATBench	BBEH-mini	AIME24	AIME25	AMC2023	MATH500	OlympiadBench
Qwen2.5-7B-Instruct	16.8	12.1	9.6	51.6	11.3	13.3	6.7	52.5	75.2	41.0
SFT	61.9	48.0	14.7	**70.0**	9.8	20.0	13.3	**67.5**	**80.8**	43.4
RL (PC-83K)	**71.0**	**61.0**	15.2	62.0	**17.0**	16.7	13.3	65.0	80.0	44.4
SynLogic-7B‡	-	-	-	-	8.0	10.0	-	55.0	71.8	-
RL (PC-SL-35K)	22.0	14.3	**55.3**	58.4	16.5	**23.3**	10.0	62.5	79.8	42.4
RL (PC-83K+PC-SL-35K)	64.8	54.1	54.2	57.2	**17.0**	16.7	**16.7**	60.0	80.4	**52.2**

Table 3: Performance of models trained on PC-83K and/or PC-SL-35K. † indicates the corresponding held-out test sets. ‡ denotes results reported in the SynLogic (Liu et al., 2025b) paper due to incompatible data formats.

open models and even some proprietary models.

Among the proprietary models, ChatGPT-5 leads (Avg. 88.7), ChatGPT-o3 (Avg. 85.3), and Seed1.6 (Avg. 85.1) are the top three, while Gemini 3 Pro (Avg. 84.8) shows the smallest *Normal* to *Hard* drop (-3.5), evidencing strong, stable reasoning. Gemini-2.5-pro ranks next (Avg. 71.5) with moderate robustness, while Claude-4-sonnet is mid-pack (Avg. 55.3) but degrades sharply on *Hard*. Non-flagship models perform notably worse, with Gemini-2.0-flash at 36.8, ChatGPT-4o at 28.2, and Claude-3.5-sonnet at 32.5.

4.3 Post Training

To evaluate the quality of PC-83K, we perform post-training (SFT and RL) on Qwen2.5-7B-Instruct. During RL (GRPO (Shao et al., 2024)) training, we use Qwen2.5-7B-Instruct as the base model to directly compare the effectiveness of SFT and RL training stages. As shown in Table 1, we use 74,354 training samples for the RL stage. For SFT, we first use DeepSeek-R1-0528 to generate thinking traces for 4,300 training samples to populate the reasoning context. We then apply rule-based filters to remove thinking traces with duplicated generations and multilingual reasoning content. After filtering, 3,574 samples remain for SFT.

To demonstrate its effectiveness in enhancing the logical reasoning capabilities of LLMs, we evaluate the model on standard logic benchmarks, including SATBench (Wei et al., 2025a) and BBEH-mini (Kazemi et al., 2025), as well as mathematical benchmarks such as AIME24 (MAA and users, 2024), AIME25 (MAA and users, 2025), AMC2023 (MAA and users, 2023), MATH500 (Hendrycks et al., 2021a), and OlympiadBench (He et al., 2024). These evaluations demonstrate the capability of our training data to enhance both logical reasoning and mathematical problem-solving.

The main results are presented in Table 3, demonstrating that model performance is significantly improved by training on the PC-83K dataset at both the SFT and RL stages.

Training on PC-83K brings large gains over Qwen2.5-7B-Instruct. The average scores rise from 14.5 to 55.0 on PC-83K test set after SFT, along with consistent improvement on the *Normal* (16.8 to 61.9) and *Hard* (12.1 to 48.0) sets respectively. RL further improves the baseline to 66.0 on average with 71.0 on the *Normal* set and 61.0 on the *Hard* set, showing better handling of both easy difficult items than the baseline and SFT.

SFT substantially improves SATBench, from 51.6 to 70.0, though it slightly lowers BBEH-mini to 9.8. After RL training stage, SATBench raises to 62.0, while BBEH-mini climbs to 17.0. Interestingly, after the SFT stage, we observe a higher rate of duplicated generations, which likely explains the drop on BBEH-mini (from 11.3 to 9.8). This suggests a distributional skew introduced by SFT data. Adding more diverse SFT samples should rebalance the data and reduce the drop. SFT yields consistent gains across AIME24, AIME25, AMC2023, MATH500, and OlympiadBench. RL partially regresses these improvements, with three lower scores and one higher score than SFT on these sets.

4.4 Case Study: Generalizing SynLogic

To demonstrate the practicality and generalizability of PuzzleClone, we conduct a case study by reproducing and extending SynLogic (Liu et al., 2025b), a representative dataset of over 33K puzzles spanning 35 logical reasoning tasks. Specifically, we treat SynLogic puzzles as seeds and curate DSL specs to synthesize a new dataset of 35,829 puzzles with PuzzleClone, termed PC-SL-35K, which is split into training and test sets using a 90/10 ratio. We perform two RL experiments, trained on PC-SL-35K and on PC-83K+PC-SL-35K, respectively, with results summarized in Table 3.

Our analysis yields four key observations. First, Qwen2.5-7B-Instruct achieves comparable accu-

racy on the original SynLogic benchmark (9.0) and the PC-SL-35K test set (9.6), indicating similar difficulty levels across the two datasets. Second, RL (PC-SL-35K) consistently outperforms the SynLogic-7B baseline on external benchmarks, demonstrating that PuzzleClone-synthesized data is at least as effective as SynLogic data in enhancing model reasoning performance. Third, while RL (PC-83K+PC-SL-35K) generally outperforms RL (PC-SL-35K), it slightly underperforms RL (PC-83K) on some benchmarks, which we attribute to differences in training progress (600 vs. 900 steps). Fourth, both SFT and RL (PC-83K) models achieve higher accuracy on the PC-SL-35K test set than Qwen2.5-7B-Instruct, indicating improved generalization to unseen seed puzzles and further validating the effectiveness of PuzzleClone.

5 Related Work

5.1 Data generation

Research on data generation mainly focus on *data synthesis* and *data augmentation* (Wang et al., 2024a).

Data synthesis approaches broadly start by extracting instructions from existing data sources or synthesizing them with rule-based or model-based synthesizers (Maiya et al., 2025; Ding et al., 2023; Xu et al., 2024b). Next, responses will be generated for each instruction instance, either manually or automatically. Machine learning has been widely adopted to accelerate the annotation process of various types of data (Zhang et al., 2021; Lu et al., 2023), despite their limitations in efficiency, data consistency, and domain knowledge (Tan et al., 2024). These limitations are addressed by modern LLMs, whose reasoning abilities make them powerful universal annotators (Zhang et al., 2021; Lu et al., 2023; Csanády et al., 2024; Zhang et al., 2023). Tan et al. provide a detailed survey of these methods (Tan et al., 2024). However, expert knowledge remains essential for accurate labels. Consequently, semi-automatic approaches have emerged, utilizing human-AI collaborative tools and co-labeling workflows to enhance efficiency and accuracy (Li et al., 2023; Wang et al., 2024b; Zhu et al., 2024).

By contrast, data augmentation focuses on adapting existing data items, called seeds, to new instances with similar structures. In natural language processing, traditional approaches mainly focus on paraphrasing the instructions, leveraging approaches like rule-based transformation and language models (Feng et al., 2021; Xu et al., 2024a; Sugiyama and Yoshinaga, 2019; Bayer et al., 2022). With the advent of LLMs, modern approaches have explored involving LLMs throughout the pipeline (Wang et al., 2024a), allowing LLMs to automatically select promising seeds, generate data and self-improve their reasoning, fine-tune the instructions, and perform evaluation-based filtering to enhance data quality (Lu et al., 2024; Shah et al., 2024; Huang et al., 2024). Some works further involve human validators or LLM-generated scripts in symbolic languages for better accuracy (Leang et al., 2025; Shah et al., 2024). However, there remains a need for robust and scalable strategies that enable end-to-end verification and fine-grained controllability throughout the pipeline.

5.2 Benchmarks for logical reasoning

Numerous public datasets for mathematical and logical reasoning have been collected through academic challenges and crowdsourcing (Hendrycks et al., 2021b; Math-AI, 2025). Curated datasets, however, mostly focus on specific classes of puzzles. For instance, numerous studies generate equation systems or deductive, inductive, and abductive puzzles by combining unit equations or basic logical clauses while crafting natural language instructions using templates or language models (Chen et al., 2025c; Parmar et al., 2024; Luo et al., 2023; Wei et al., 2025b; Mirzadeh et al., 2024). However, it becomes increasingly difficult to map the abstract logical model to a real-world scenario when it becomes difficult, which in turn jeopardizes the articulation of the instructions. By contrast, another line of research start from classical complicated and challenging mathematical puzzles such as sudoku, hitori, and crosswords, and curate a number of similar puzzles simply through rule-based methods (Gui et al., 2024; Mittal et al., 2024; Chen et al., 2025b,a; Li et al., 2025; Liu et al., 2025a,b). However, a universal framework capable of generating various challenging puzzles is still absent, leading us to design the PuzzleClone framework.

6 Discussion

In the process of designing and implementing the PuzzleClone framework, as well as synthesizing variants from seed puzzles, we encountered several key insights and reflections worth discussing.

Diverse Question and Evaluation Types: Un-

like some existing datasets that typically focus on a single task format, our dataset features a wide range of puzzle types and answer formats (Appendix B). Notably, a single puzzle in our dataset may contain multiple sub-questions, each with its own `qtype` and `eval_type`, which further enhances the overall richness and versatility of the dataset. Such diversity enables more comprehensive benchmarking, supports evaluation of specific reasoning capabilities, and promotes model robustness across heterogeneous formats—better reflecting real-world problem-solving scenarios.

Specification Strategies: We observed two distinct approaches to puzzle synthesis based on the specification: forward and backward generation. In the forward strategy (e.g., specs for Figure 2, `15-tabletennis`, and `23-product`), we first define random domains for variables and use symbolic constraints to allow the solver to search for feasible solutions. In contrast, the backward strategy (e.g., specs for `7-age` and `11-wine`) involves randomly generating a feasible solution first, then building symbolic constraints around it to verify its correctness and uniqueness using the solver. The backward approach significantly improves generation efficiency by avoiding unsolvable instances due to random variable combinations. Therefore, we recommend using the backward strategy for most specification designs.

Hyperparameter Sensitivity: While writing specifications, careful tuning of hyperparameters such as variable domains, constraint templates, and their count ranges is crucial to maintain the semantic soundness of generated puzzles. For example, in `2-graduation`, where m out of n students are selected under constraints, it is necessary to enforce $0 < m < n$. However, making m too small or too large ($m \to 0$ or $m \to n$) reduces puzzle difficulty. Thus, domain ranges should be both logically valid and aligned with the intended cognitive challenge.

Upper Bound of Synthesizable Puzzles: While PuzzleClone can synthesize puzzles at scale, each specification inherently has a finite upper bound on the number of valid (i.e., non-duplicate) puzzles it can generate. This bound is determined by the size of the puzzle space defined in the specification. As shown in Figure 4, for `9-vase`, we synthesized variants using three specs with decreasing `p_num` domains: [3,10], [3,7], and [3,5]. After generating 100K puzzles for each setting and performing de-duplication, we observed that [3,5] quickly saturated at around 3.3K unique puzzles, while [3,10] continued to grow steadily. The [3,7] setting fell in between and showed signs of approaching saturation. This demonstrates that a larger solution space allows the synthesis of more diverse valid puzzles.

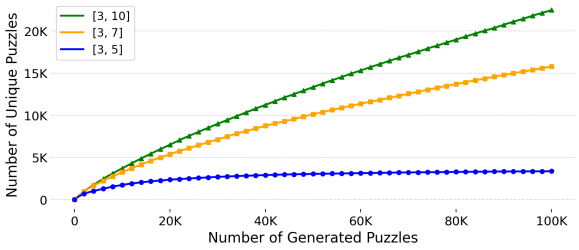

Figure 4: Number of valid puzzles (after de-duplication) synthesized from the seed puzzle `9-vase` using three different `p_num` domain settings.

Preventing Redundancy in Puzzle Statements: It is critical to ensure that the question and its constraints do not inadvertently leak the answer. For instance, in `23-product`, if a condition states that a specific product is in position i and the question also asks about the position of that product, the answer becomes trivially inferable without reasoning. Such overlap undermines the puzzle's intent. Specifications should explicitly avoid this redundancy to preserve reasoning difficulty.

Advanced Features for Practical Synthesis: To enhance expressiveness and practical utility, PuzzleClone supports several advanced features:

- *Coupled Logic:* Supports joint control of variables and constraints (e.g., $A + B = 10$), enabling fine-grained coordination beyond independent random sampling.
- *Custom Operators:* Users can define new functions or import scripts, compensating for limitations in built-in operators.
- *Dynamic Rephrasing:* PuzzleClone supports modifying the textual surface of puzzles (see Appendix), enabling translation into multiple languages or thematic adaptation, thereby increasing data diversity.

7 Conclusion

We present PuzzleClone, a DSL-driven data synthesis framework and a diverse and challenging dataset containing over 83K puzzles. Experiments show that existing state-of-the-art models face significant challenges on this benchmark. Furthermore, SFT and RL applied to Qwen2.5-7B-Instruct show substantial gains in mathematical reasoning, validating the effectiveness of our approach.

Limitations

Challenges in measuring puzzle difficulty accurately: Although the experiment in Appendix A.1 demonstrates the validity of the difficult metric, it has two main limitations. First, the `var_scale` metric relies on a simple `diff_factor` with only three discrete values (+1, 0, –1) to indicate correlation with puzzle difficulty, which cannot capture more nuanced relationships. Second, not all difficulty-variable relationships are monotonic; for example, increasing a variable's value may initially make a puzzle harder but later easier. Future work could explore more expressive difficulty metrics that better capture non-linear and variable-specific effects.

Biased puzzle difficulty distribution: When variable domains are large and constraints are complex, the probability of randomly generating solvable instances diminishes. This leads to a distributional bias in synthesized puzzles (see Figure 3): easier puzzles (associated with simpler parameter combinations) are overrepresented, while harder ones are under-sampled. Future research could design targeted generation procedures to ensure adequate representation of challenging puzzles.

Human effort in specification authoring: Authoring DSL specifications from seed puzzles currently requires substantial manual effort, including decomposing and rephrasing puzzles, brainstorming new variants and constraints to enhance puzzle diversity and complexity, and encoding them into DSL specifications. This labor-intensive process limits scalability. A promising future direction is to leverage LLMs to automate these steps with minimal human intervention.

References

Anthropic. 2024. Introducing claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet. Announcement; see also model card addendum.

Anthropic. 2025. Claude opus 4 & claude sonnet 4: System card. https://www.anthropic.com/claude-4-system-card. Model referenced in paper as Claude-4-sonnet.

Clark Barrett, Aaron Stump, Cesare Tinelli, and 1 others. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14.

Markus Bayer, Marc-André Kaufhold, and Christian Reuter. 2022. A survey on data augmentation for text classification. *ACM Computing Surveys*, 55(7):1–39.

ByteDance. 2025. Seed1.6 tech introduction. https://seed.bytedance.com/en/seed1_6. Model referenced in paper as Seed1.6.

Guizhen Chen, Weiwen Xu, Hao Zhang, Hou Pong Chan, Chaoqun Liu, Lidong Bing, Deli Zhao, Anh Tuan Luu, and Yu Rong. 2025a. FINEREASON: Evaluating and Improving LLMs' Deliberate Reasoning through Reflective Puzzle Solving. *arXiv preprint*. ArXiv:2502.20238 [cs].

Jianghao Chen, Zhenlin Wei, Zhenjiang Ren, Ziyong Li, and Jiajun Zhang. 2025b. LR^{2}Bench: Evaluating Long-chain Reflective Reasoning Capabilities of Large Language Models via Constraint Satisfaction Problems. *arXiv preprint*. ArXiv:2502.17848 [cs].

Michael K Chen, Xikun Zhang, and Dacheng Tao. 2025c. Justlogic: A comprehensive benchmark for evaluating deductive reasoning in large language models. *arXiv preprint arXiv:2501.14851*.

William F Clocksin and Christopher S Mellish. 2003. *Programming in PROLOG*. Springer Science & Business Media.

Bálint Csanády, Lajos Muzsai, Péter Vedres, Zoltán Nádasdy, and András Lukács. 2024. Llambert: Large-scale low-cost data annotation in nlp. *arXiv preprint arXiv:2403.15938*.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg. Springer-Verlag.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. *arXiv preprint arXiv:2305.14233*.

Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A survey of data augmentation approaches for NLP. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 968–988, Online. Association for Computational Linguistics.

Gemini Team. 2025a. Gemini 2.0 flash. https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash. Model referenced in paper as Gemini-2.0-flash.

Gemini Team. 2025b. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. Technical report, Google DeepMind. Covers Gemini 2.5 Pro (and 2.5 Flash); model referenced in paper as Gemini-2.5-pro.

Gemini Team. 2025c. Gemini 3 pro: the frontier of vision ai. https://blog.google/technology/developers/gemini-3-pro-vision/. Model referenced in paper as Gemini-3-pro.

Jiayi Gui, Yiming Liu, Jiale Cheng, Xiaotao Gu, Xiao Liu, Hongning Wang, Yuxiao Dong, Jie Tang, and Minlie Huang. 2024. Logicgame: Benchmarking rule-based reasoning abilities of large language models. *arXiv preprint arXiv:2408.15778*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Zongfang Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, DeepSeek-AI Team, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*. Covers DeepSeek-R1/-R1-Zero and distilled variants.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, and 1 others. 2024. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. *arXiv preprint arXiv:2402.14008*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021a. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021b. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Yue Huang, Siyuan Wu, Chujie Gao, Dongping Chen, Qihui Zhang, Yao Wan, Tianyi Zhou, Chaowei Xiao, Jianfeng Gao, Lichao Sun, and 1 others. 2024. Datagen: Unified synthetic dataset generation via large language models. In *The Thirteenth International Conference on Learning Representations*.

Mehran Kazemi, Bahare Fatemi, Hritik Bansal, John Palowitch, Chrysovalantis Anastasiou, Sanket Mehta, Lalit D. Jain, Virginia Aglietti, Disha Jindal, Peter Chen, Nishanth Dikkala, Gladys Tyen, Xin Liu, Uri S. Shalit, Yi Tay, Vinh Q. Tran, Quoc V. Le, and Orhan Firat. 2025. Big-bench extra hard. *arXiv preprint arXiv:2502.19187*.

Joshua Ong Jun Leang, Giwon Hong, Wenda Li, and Shay B. Cohen. 2025. Theorem Prover as a Judge for Synthetic Data Generation.

Minzhi Li, Taiwei Shi, Caleb Ziems, Min-Yen Kan, Nancy F Chen, Zhengyuan Liu, and Diyi Yang. 2023. Coannotating: Uncertainty-guided work allocation between human and large language models for data annotation. *arXiv preprint arXiv:2310.15638*.

Naiqi Li, Peiyuan Liu, Zheng Liu, Tao Dai, Yong Jiang, and Shu-Tao Xia. 2025. Logic-of-thought: Empowering large language models with logic programs for solving puzzles in natural language. *arXiv preprint arXiv:2505.16114*.

Junqi Liu, Xiaohan Lin, Jonas Bayer, Yael Dillies, Weijie Jiang, Xiaodan Liang, Roman Soletskyi, Haiming Wang, Yunzhou Xie, Beibei Xiong, and 1 others. 2025a. Combibench: Benchmarking llm capability for combinatorial mathematics. *arXiv preprint arXiv:2505.03171*.

Junteng Liu, Yuanxiang Fan, Zhuo Jiang, Han Ding, Yongyi Hu, Chi Zhang, Yiqi Shi, Shitong Weng, Aili Chen, Shiqi Chen, Yunan Huang, Mozhi Zhang, Pengyu Zhao, Junjie Yan, and Junxian He. 2025b. Synlogic: Synthesizing verifiable reasoning data at scale for learning logical reasoning and beyond. *Preprint*, arXiv:2505.19641.

Yingzhou Lu, Minjie Shen, Huazheng Wang, Xiao Wang, Capucine van Rechem, Tianfan Fu, and Wenqi Wei. 2023. Machine learning for synthetic data generation: a review. *arXiv preprint arXiv:2302.04062*.

Zimu Lu, Aojun Zhou, Houxing Ren, Ke Wang, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024. MathGenie: Generating Synthetic Data with Question Back-translation for Enhancing Mathematical Reasoning of LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2732–2747, Stroudsburg, PA, USA. Association for Computational Linguistics.

Man Luo, Shrinidhi Kumbhar, Mihir Parmar, Neeraj Varshney, Pratyay Banerjee, Somak Aditya, Chitta Baral, and 1 others. 2023. Towards logiglue: A brief survey and a benchmark for analyzing logical reasoning capabilities of language models. *arXiv preprint arXiv:2310.00836*.

MAA and Benchmark users. 2023. Amc 2023 benchmark (american mathematics competition 2023). Used as evaluation benchmark in LLM reasoning research.

MAA and Benchmark users. 2024. Aime 2024 benchmark (american invitational mathematics examination 2024). Used as evaluation benchmark in LLM reasoning research.

MAA and Benchmark users. 2025. Aime 2025 benchmark (american invitational mathematics examination 2025). Used in recent benchmarks like SAT-Bench analysis, etc.

Anirudh Maiya, Razan Alghamdi, Maria Leonor Pacheco, Ashutosh Trivedi, and Fabio Somenzi. 2025. Explaining puzzle solutions in natural language: An exploratory study on 6x6 sudoku. *arXiv preprint arXiv:2505.15993*.

Math-AI. 2025. Aime25 dataset. Accessed: 2025-07-01.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229*.

Chinmay Mittal, Krishna Kartik, Mausam, and Parag Singla. 2024. PuzzleBench: Can LLMs Solve Challenging First-Order Combinatorial Reasoning Problems? *arXiv preprint*. ArXiv:2402.02611 [cs] version: 2.

OpenAI. 2024. Gpt-4o system card. https://arxiv.org/abs/2410.21276. Model referenced in paper as ChatGPT-4o.

OpenAI. 2025a. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/. Model referenced in paper as ChatGPT-5.

OpenAI. 2025b. Openai o3 and o4-mini system card. https://openai.com/index/o3-o4-mini-system-card/. Model referenced in paper as ChatGPT-o3.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824, Singapore. Association for Computational Linguistics.

Mihir Parmar, Nisarg Patel, Neeraj Varshney, Mutsumi Nakamura, Man Luo, Santosh Mashetty, Arindam Mitra, and Chitta Baral. 2024. Logicbench: Towards systematic evaluation of logical reasoning ability of large language models. *arXiv preprint arXiv:2404.15522*.

Vedant Shah, Dingli Yu, Kaifeng Lyu, Simon Park, Nan Rosemary Ke, Michael Mozer, Yoshua Bengio, Sanjeev Arora, and Anirudh Goyal. 2024. AI-Assisted Generation of Difficult Math Questions.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*. Introduces Group Relative Policy Optimization (GRPO).

Amane Sugiyama and Naoki Yoshinaga. 2019. Data augmentation using back-translation for context-aware neural machine translation. In *Proceedings of the fourth workshop on discourse in machine translation (DiscoMT 2019)*, pages 35–44.

Zhen Tan, Dawei Li, Song Wang, Alimohammad Beigi, Bohan Jiang, Amrita Bhattacharjee, Mansooreh Karami, Jundong Li, Lu Cheng, and Huan Liu. 2024. Large language models for data annotation and synthesis: A survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 930–957, Miami, Florida, USA. Association for Computational Linguistics.

THUDM. 2025. Glm-z1-32b-0414. https://huggingface.co/THUDM/GLM-Z1-32B-0414. GLM-Z1 reasoning model series (incl. 9B/32B); model card.

Ke Wang, Jiahui Zhu, Minjie Ren, Zeming Liu, Shiwei Li, Zongye Zhang, Chenkai Zhang, Xiaoyu Wu, Qiqi Zhan, Qingjie Liu, and Yunhong Wang. 2024a. *A Survey on Data Synthesis and Augmentation for Large Language Models*, volume 1. Association for Computing Machinery.

Yifan Wang, David Stevens, Pranay Shah, Wenwen Jiang, Miao Liu, Xu Chen, Robert Kuo, Na Li, Boying Gong, Daniel Lee, and 1 others. 2024b. Model-in-the-loop (milo): Accelerating multimodal ai data annotation with llms. *arXiv preprint arXiv:2409.10702*.

Anjiang Wei, Yuheng Wu, Yingjia Wan, Tarun Suresh, Huanmi Tan, Zhanke Zhou, Sanmi Koyejo, Ke Wang, and Alex Aiken. 2025a. Satbench: Benchmarking llms' logical reasoning via automated puzzle generation from sat formulas. *arXiv preprint arXiv:2505.14615*.

Anjiang Wei, Yuheng Wu, Yingjia Wan, Tarun Suresh, Huanmi Tan, Zhanke Zhou, Sanmi Koyejo, Ke Wang, and Alex Aiken. 2025b. Satbench: Benchmarking llms' logical reasoning via automated puzzle generation from sat formulas. *arXiv preprint arXiv:2505.14615*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024a. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2024b. Magpie: Alignment data synthesis from scratch by prompting aligned llms with nothing. *arXiv preprint arXiv:2406.08464*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 42 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 24 others. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Ruoyu Zhang, Yanzeng Li, Yongliang Ma, Ming Zhou, and Lei Zou. 2023. Llmaaa: Making large language models as active annotators. *arXiv preprint arXiv:2310.19596*.

Shikun Zhang, Omid Jafari, and Parth Nagarkar. 2021. A survey on machine learning techniques for auto labeling of video, audio, and text data. *arXiv preprint arXiv:2109.03784*.

Zhen Zhu, Yibo Wang, Shouqing Yang, Lin Long, Runze Wu, Xiu Tang, Junbo Zhao, and Haobo Wang. 2024. Coral: Collaborative automatic labeling system based on large language models. *Proc. VLDB Endow.*, 17(12):4401–4404.

A Supplement to Dataset Construction

This supplement provides additional details on the heuristic puzzle difficulty metric and its validation, as well as the prompt templates used in this work.

A.1 Puzzle Difficulty Metric

As described in Section 3.4, the difficulty metric is computed from four features: `sym_num`, `cond_num`, `desc_len`, and `vars_scale`. Each variable v is associated with a `diff_factor` (Figure 2B), which specifies the direction of correlation between the variable value and puzzle difficulty: a larger value makes the puzzle harder ($+1$), easier (-1), or has no effect (0, by default). Algorithm 1 details the procedure for computing the final difficulty score.

To evaluate the alignment between our proposed metric and empirical difficulty, we conducted a study using four models from the Qwen2.5 family (7B, 14B, 32B, and 72B Instruct). We tested these models on puzzles synthesized from five randomly selected seeds in PuzzleClone, each seed leading to between 947 and 1,000 unique puzzles after deduplication. To mitigate the impact of randomness and ensure objective results, we conducted 10 independent trials for each model on every puzzle instance. Puzzles were then categorized into bins with a 0.1 difficulty score increment, and the mean accuracy was calculated for each group. As illustrated in Figure 5, our metric demonstrates a strong inverse relationship with model performance. Specifically, the Pearson correlation coefficients range from -0.816 to -0.980, confirming that the metric accurately reflects task difficulty.

A.2 Prompt Templates

Figure 6 and Figure 7 show the prompt templates used for seed puzzle selection and downstream experiments, respectively. Each template is presented in both its original Chinese form (used in practice) and its English translation (for readability).

Algorithm 1: Compute Puzzle Difficulty Score

Require: Variables $\mathcal{V}$ from the puzzle `config`, features sym_num, $cond\_num$, $desc\_len$, together with their minimum and maximum values computed over all puzzle instances

Ensure: Difficulty score $D \in [0, 1]$

Define $\textsc{MinMaxNorm}(x) = \dfrac{x - x_{\min}}{x_{\max} - x_{\min}}$
$\mathcal{A} \leftarrow \emptyset$ {Adjusted variable values}
for each variable $v \in \mathcal{V}$ **do**
  if $diff\_factor(v) = 0$ **then**
    continue
  end if
  $\hat{v} \leftarrow \textsc{MinMaxNorm}(v)$
  if $diff\_factor(v) > 0$ **then**
    $v_{adj} \leftarrow \hat{v}$
  else
    $v_{adj} \leftarrow 1 - \hat{v}$
  end if
  add v_{adj} to $\mathcal{A}$
end for
$vars\_scale \leftarrow \text{mean}(\mathcal{A})$
$\hat{s} \leftarrow \textsc{MinMaxNorm}(sym\_num)$
$\hat{c} \leftarrow \textsc{MinMaxNorm}(cond\_num)$
$\hat{d} \leftarrow \textsc{MinMaxNorm}(desc\_len)$
$\hat{v} \leftarrow \textsc{MinMaxNorm}(vars\_scale)$
$D \leftarrow \text{mean}(\hat{s}, \hat{c}, \hat{d}, \hat{v})$
return D

B Benchmark Statistics

B.1 Question Type Distribution

Our benchmark encompasses three primary question formats: short-answer, multiple-choice, and fill-in-the-blank questions. Table 4 presents the distribution of these question types across different data splits. When a puzzle contains multiple sub-questions, each sub-question is counted independently according to its type.

The benchmark demonstrates a predominance of short-answer questions (75,486 instances), which require models to perform complex reasoning and generate precise responses. Multiple-choice questions constitute the second-largest category (18,929 instances), providing evaluation through option selection. Fill-in-the-blank questions, while less common (2,000 instances), test the model's ability to complete partial puzzle descriptions. This distribution reflects the natural complexity spectrum of logic puzzles, where open-ended reasoning challenges are most prevalent.

Data Split	Short Answer	Multiple-Choice	Fill-in-the-Blank	Total
`normal/RL_validate.jsonl`	390	95	10	495
`normal/Test.jsonl`	5,108	1,262	124	6,494
`normal/SFT.jsonl`*	1,966	475	50	2,491
`normal/RL_train.jsonl`	45,195	11,202	1,099	57,496
`hard/RL_validate.jsonl`	390	95	10	495
`hard/Test.jsonl`	2,514	644	78	3,236
`hard/SFT.jsonl`*	1,934	475	50	2,459
`hard/RL_train.jsonl`	21,889	5,631	679	28,199
Total†	**75,486**	**18,929**	**2,000**	**96,415**

Table 4: Distribution of question types across data splits in PuzzleClone. *SFT samples are subsets selected from other training splits. †Total excludes SFT splits to avoid double-counting.

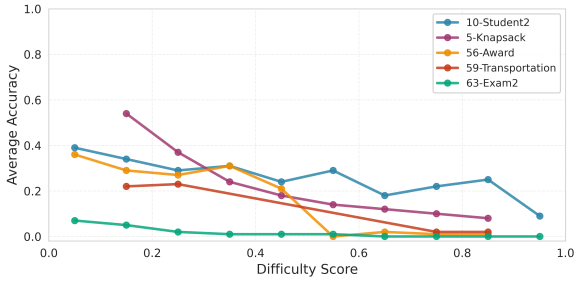

Figure 5: Average performance of four models on puzzles synthesized from 5 sampled seeds. The mid point is used as the difficulty score for each bin. Missing points indicate that no puzzles exist in the correponding bin.

B.2 Answer Type Distribution

Beyond question format diversity, our benchmark exhibits rich structural variation in answer types. Table 5 categorizes answers into eight distinct types, reflecting the diverse reasoning outputs required by logic puzzles:

- **numeral**: Numeric values or lists thereof (e.g., *42*, *[3, 7, 12]*).
- **option**: Single-letter option identifiers (e.g., *A*, *D*).
- **ordered array**: Sequences where element order is semantically meaningful (e.g., rankings, positions).
- **nominal**: Categorical labels or entity names (e.g., person names, object types).
- **unordered array**: Sets where element order is irrelevant (e.g., collections of items).
- **ooa_numeral**: Two-dimensional numeric arrays with order preservation in both dimensions.
- **ooa_nominal**: Two-dimensional nominal arrays with order preservation in both dimensions.
- **oua_nominal**: Two-dimensional nominal arrays where only outer-dimension order matters.

This taxonomy enables fine-grained evaluation of model capabilities across different reasoning output structures. The distribution shows that simpler answer types (*numeral* and *option*) dominate, while complex two-dimensional structures (*ooa_*, *oua_**) represent challenging edge cases with 1,000 instances each, deliberately included to test advanced compositional reasoning.

B.3 Duplicate Instance Distribution

The distribution of duplicate puzzle instances grouped by seed puzzles during the deduplication step (Section 3.3) is shown in Figure 8. Among the 86 seed puzzles, 32 had at least one duplicated variant. We observe that puzzles with limited randomizable space tend to exhibit higher duplication rates. For example, `28-exam` exhibited the highest duplication count, with 678 repeated instances.

C DSL Schema

This appendix provides a detailed specification of the Domain-Specific Language (DSL) used for puzzle encoding and generation. The DSL is structured around a main `PuzzleTemplate` object, which integrates all necessary components for defining a

Data Split	num.	opt.	ord.	nom.	unord.	oon.	oonl.	oun.	Total
normal/RL_validate	180	95	80	70	55	5	5	5	495
normal/Test	2,089	1,262	1,232	961	727	55	77	91	6,494
normal/SFT*	910	475	404	352	275	25	25	25	2,491
normal/RL_train	18,444	11,202	10,936	8,503	6,442	483	681	805	57,496
hard/RL_validate	180	95	80	70	55	5	5	5	495
hard/Test	1,501	644	351	433	227	46	24	10	3,236
hard/SFT*	890	475	396	348	275	25	25	25	2,459
hard/RL_train	13,175	5,631	3,011	3,756	1,928	406	208	84	28,199
Total†	**35,569**	**18,929**	**15,690**	**13,793**	**9,434**	**1,000**	**1,000**	**1,000**	**96,415**

Table 5: Distribution of answer types across data splits. Column headers are abbreviated: **num.**=numeral, **opt.**=option, **ord.**=ordered array, **nom.**=nominal, **unord.**=unordered array, **oon.**=ooa_numeral, **oonl.**=ooa_nominal, **oun.**=oua_nominal. *SFT samples are subsets from other splits. †Total excludes SFT splits.

puzzle's logic, parameters, and text. The following tables detail each of the core components.

C.1 PuzzleTemplate

The root object, `PuzzleTemplate`, is the top-level container that orchestrates all other elements of the puzzle definition. Its fields are detailed in Table 6.

C.2 Variables

The `Variable` object defines the parameters of a puzzle. A variable can be defined either by its type and domain or by a formula, as shown in Table 7.

C.3 Symbols

Symbols represent the quantities to be solved. They can be directly defined (`DefinedSymbol`) or derived from existing variables (`DerivedSymbol`).

C.4 Conditions

Conditions define the logical constraints of the puzzle, as either static (`StaticCondition`) or dynamically generated (`DynamicCondition`) rules.

C.5 Queries

Queries define the questions posed to the user, supporting open-ended and multiple-choice formats.

C.6 Auxiliary Definitions

The DSL includes objects for post-processing and optimization tasks.

In addition, PuzzleClone supports several internal operators and reserved words for expressiveness. For instance, `_opt` and `_sym` refer to the parameter randomization result of each option and symbol, while `get_faker` is an internal operator for fake entity generation.

D Puzzle Specification Examples

This section provides the complete DSL specification files for three representative puzzles from our benchmark. All descriptive texts, originally in Chinese, have been translated to English.

- `1-hamburger` (Figure 2): See Listings 1 and 2.
- `2-graduation`: See Listing 3.
- `9-vase`: See Listing 4.
- `11-wine`: See Listing 5.
- `23-product`: See Listing 6.
- `28-exam` See Listing 7.

E Training Parameters

Table 16 outlines SFT setup: 32K context, cosine LR schedule with warmup, bf16/tf32, gradient checkpointing, ZeRO-3 Offload via DeepSpeed, six epochs, small per-device batch with accumulation, and step-based checkpointing.

Table 17 summarizes key GRPO hyperparameters: KL regularization, vLLM rollouts, fsdp2 distributed training, Qwen2.5-7B-Instruct, five epochs, sixteen H100 across two nodes, dynamic batching and token-limits.

F Supplement to Experimental Results

Figure 9 presents the average accuracy of all evaluated models on the PuzzleClone test set, grouped by their originating seed puzzles. For most seed puzzles, the *hard* variants yield lower average accuracy compared to the *normal* ones, demonstrating the effectiveness of our difficulty stratification. However, for certain seed puzzles, the *hard* subset shows unexpectedly higher accuracy. This counterintuitive trend reveals limitations in our current difficulty scoring method, which may not fully capture the underlying reasoning complexity. For a

Listing 1: DSL Specification for the "Hamburger" Puzzle (Part 1/2).

```
variables:
  s_num: {type: int, domain: "[4, 7]", diff_factor:
       1}
  f_num: {type: int, domain: "[3, 5]", diff_factor:
       1}
  names: {formula: get_faker(s_num, 'name')}
  food: {formula: get_faker(f_num, 'food')}
symbols:
  buy: {source: [names, food], type: bool}
conditions:
  purchased_at_least_one_kind:
    formula: And([Or([buy[(p, f)] for f in food])
         for p in names])
    desc: "{s_num} students, {', '.join(names)},
         have purchased at least one kind of food:
         {','.join(food)}."
  if_a_then_not_b:
    source: [food, "[False, True]"]
    amount: ['2', '2']
    formula: >
      And([Implies(buy[(p, _sym[0][0])] if _sym
           [1][0] else Not(buy[(p, _sym[0][0])]),
           buy[(p, _sym[0][1])] if _sym[1][1] else
           Not(buy[(p, _sym[0][1])])) for p in names
           ])
    desc: >
      People who {'did not buy' if _sym[1][0] else '
           bought'} {_sym[0][0]} {'did not buy' if
           _sym[1][1] else 'bought'} {_sym[0][1]}.
  at_least_one_person_bought:
    source: [food]
    domain: "[1, 2]"
    formula: Or([buy[(p, _sym[0])] for p in names])
    desc: At least one person bought {_sym[0]}.
  a_bought_b:
    source: [names, food, "[False, True]"]
    domain: "[s_num*f_num//3, s_num*f_num//2]"
    formula: buy[(_sym[0], _sym[1])] == _sym[2]
    desc: "{_sym[0]} {'bought' if _sym[2] else 'did
         not buy'} {_sym[1]}."
  a_b_exclusive:
    source: [names]
    amount: ['2']
    domain: "[1, 2]"
    formula: >
      And([Implies(buy[(_sym[0][0], f)], Not(buy[(
           _sym[0][1], f)])) for f in food])
    desc: "{_sym[0][1]} did not buy any items that {
         _sym[0][0]} bought."
  assumption:
    source: [names, "range(2, f_num)"]
    amount: ['2', '1']
    formula: And([Sum([If(buy[(p, f)], 1, 0) for f
         in food]) == _sym[1][0] for p in
      _sym[0]])
    desc: "If both {_sym[0][0]} and {_sym[0][1]}
         bought {_sym[1][0]} kinds of products,"
```

Listing 2: DSL Specification for the "Hamburger" Puzzle (Part 2/2).

```
# --- (Continuation) Spec of "1-hamburger" ---
queries:
  question:
    desc: "which of the following must be true?"
    opt_num: 4
    templates:
    - source: [names]
      amount: ['2']
      cond: all
      opt_formula: >
        sum([1 for f in food if get_value(_model,
             And(buy[(_opt[0][0], f)], buy[(_opt
             [0][1], f)]))]) == 1
      opt_text: >
        There is exactly one food item that both {
             _opt[0][0]} and {_opt[0][1]} bought.
    - source: [names, food, "[False, True]"]
      amount: ['1', '1', '1']
      cond: all
      opt_formula: get_value(_model, buy[(_opt
           [0][0], _opt[1][0])]) == _opt[2][0]
      opt_text: "{_opt[0][0]} {'bought' if _opt
           [2][0] else 'did not buy'} {_opt[1][0]}."
desc: >
  {purchased_at_least_one_kind}Their choices satisfy
       these conditions: {if_a_then_not_b}{
      at_least_one_person_bought}{a_bought_b}{
      a_b_exclusive}{assumption}{question}
```

more detailed analysis, please refer to the Revisiting Difficulty Estimation part in Section 6.

G Details of Dynamic Rephrasing

This section introduces *dynamic rephrasing*, a key functionality of PuzzleClone that enables transforming the original puzzle into new languages or scenarios in a fully accurate and verifiable way.

Given an original puzzle P generated by a pair of specification and configuration files (Q_s and Q_c), PuzzleClones enabling adapting P to a new Puzzle P' based on a new specification file Q'_s, where Q'_s and Q_s differ only in the descriptive texts. During generation, the values of all variables and parameters of symbols, conditions, and queries will be directly extracted from Q_c instead of through randomization. This new puzzle can be validated in a similar manner to the reproduced seed puzzle, ensuring its validity.

It is important to note that in a few cases, the values of some variables or parameters still need to be randomized, such as the names of people mentioned in the puzzle randomized by Faker. To accommodate such cases, PuzzleClone enables *partial config-based generation*, enabling users to specify variables for randomization through command-line options.

Listing 3: DSL specification for the “Graduation” puzzle.

```
variables:
  p_num: {type: int, domain: "[6, 12]", diff_factor:
       1}
  select_num: {type: int, domain: "[p_num // 2 - 1,
      p_num // 2 + 1]"}
  names: {formula: generate_letters(p_num)}
  name_desc: {formula: "', '.join(names)"}
symbols:
  events: # Represents whether a person is selected
    source: [names]
    type: bool
    desc: "{_names} was selected for the ceremony"
conditions:
  base: # Total number of selected people
    formula: gen_event_count_condition(events, '
        equal', select_num)
    desc: "Select {select_num} people for the
        graduation ceremony."
  cond1: # XOR condition
    source: [events]
    domain: "[1, 3]"
    amount: ['2']
    formula: gen_event_count_condition(_sym[0], '
        equal', 1)
    desc: "Either {get_p(_sym[0][0], 'names')} or {
        get_p(_sym[0][1], 'names')} must be
        selected, but not both."
  cond2: # Implication condition
    source: [events]
    domain: "[1, 3]"
    amount: ['2']
    formula: Implies(_sym[0][1], _sym[0][0])
    desc: "Unless {get_p(_sym[0][0], 'names')} is
        selected, {get_p(_sym[0][1], 'names')}
        cannot be."
queries:
  question:
    source: [events]
    desc: "Which of the following could be a valid
        selection?"
    opt_num: 5
    amount: [select_num]
    cond: any
    opt_formula: sum([get_value(_model, _opt[0][i])
        for i in range(select_num)]) == select_num
    opt_text: "{', '.join(get_p(_opt[0], 'names'))}"
  q2:
    source: [events]
    desc: "The selected group must include:"
    opt_num: 4
    amount: ['2']
    cond: all
    opt_formula: sum([get_value(_model, _opt[0][i])
        for i in range(2)]) >= 1
    opt_text: "{get_p(_sym[0][0], 'names')} or {
        get_p(_sym[0][1], 'names')}."
  q3:
    source: [events]
    desc: "Which two people cannot be selected at
        the same time?"
    opt_num: 5
    amount: ['2']
    cond: all
    opt_formula: sum([get_value(_model, _opt[0][i])
        for i in range(2)]) <= 1
    opt_text: "{get_p(_sym[0][0], 'names')} and {
        get_p(_sym[0][1], 'names')}"
desc: "From {p_num} graduates ({name_desc}), {base}.
     The selection must satisfy these conditions: {
    cond1} {cond2} {queries}"
```

Listing 4: DSL specification for the “Vase” puzzle.

```
variables:
  p_num: {type: int, domain: "[3, 10]", diff_factor:
       1}
  broken_vase_num: {type: int, domain: "[1, round(
      p_num / 3)]"}
  names: {formula: get_faker(p_num, 'name')}
  name_desc: {formula: "', '.join(names)"}
symbols:
  names_s: # Who broke the vase
    source: [names]
    type: bool
    desc: "{_names} broke the vase"
  speeches_s: # What each person said
    source: [names_s, "[True, False]", "['eq']"]
    domain: p_num
    domain_cond: false # Allow duplicate speeches
    formula: make_expr(_sym[2], _sym[0], _sym[1])
    desc: >
      {names[_index]} says: "{get_p(_sym[0], 'names
          ')}
      {'broke' if _sym[1] else 'did not break'} the
          vase"
conditions:
  cond1: # Total number of culprits
    formula: gen_event_count_condition(names_s, '
        equal', broken_vase_num)
    desc: >
      The mother knows {broken_vase_num} of {p_num}
          children broke the vase.
  cond2: # Culprits are liars
    formula: "[Implies(names_s[names[i]], Not(
        speeches_s[i])) for i in range(p_num)]"
    desc: "The children who broke the vase are
        definitely lying."
queries:
  question:
    desc: "Who broke the vase?"
    ans_formula: >
      get_p(get_TF_events_for_each_solution(names_s,
          _solutions, True), 'names')
    ans_text: "','.join(_ans[0])"
    ans_assertion: len(_ans) == 1
desc: >
  There are {p_num} children: {name_desc}. {
      broken_vase_num} broke a vase.
  Their statements: {', '.join(get_desc(speeches_s))
      }. {cond1}, and {cond2}. {question}
```

Listing 5: DSL specification for the “Wine” puzzle.

```
variables:
  wine_num: {type: int, domain: "[6, 12]",
       diff_factor: 1}
  beer_num: {type: int, domain: "[1, 2]"}
  bought_wine_of_first_customer: {type: int, domain:
        "[1, wine_num/2]"}
  vol_times: {type: int, domain: "[2, 5]"}
  wines: {formula: generate_letters(wine_num)}
symbols:
  wine_s: # Each barrel has a volume and belonging
       (0:beer, 1:cust1, 2:cust2)
    source: [wines]
    attr: [volume, belonging]
    type: [int, int]
conditions:
  wine_belonging: {formula: "And([Or(x==0, x==1, x
       ==2) for x in wine_s.get('belonging')])"}
  wine_0:
    formula: "Sum([If(x==0, 1, 0) for x in wine_s.
         get('belonging')]) == beer_num"
    desc: "{beer_num} barrels contain beer."
  wine_1:
    formula: "Sum([If(x==1, 1, 0) for x in wine_s.
         get('belonging')]) ==
         bought_wine_of_first_customer"
    desc: "The first customer bought {
         bought_wine_of_first_customer} barrels of
         wine."
  wine_times:
    formula: >
      Sum([If(x1 == 2,x2,0) for x1,x2 in zip(wine_s.
           get('belonging'), wine_s.get('volume'))])
      == vol_times * Sum([If(x1==1,x2,0) for x1,x2
           in zip(wine_s.get('belonging'), wine_s.
           get('volume'))])
    desc: "The second customer bought {vol_times}
         times the volume of the first."
  wine_volume_domain: {formula: "And([And(x > 0, x
       <= 50) for x in wine_s.get('volume')])"}
  wine_volume_distinct: {formula: "
       gen_event_count_condition(wine_s.get('volume
       '), 'distinct')"}
post_generation: # Solve for volumes first, then fix
      them
  post_gen_vars:
    vol: "get_value(_sol, wine_s.get('volume'))"
  post_gen_conditions:
    vol_cond:
      formula: "And([wine_s[w].get('volume') == vol[
           i] for i, w in enumerate(wines)])"
queries:
  question:
    source: [range(0, wine_num)]
    desc: "Which barrel(s) contain beer? Please
         provide the letters for the correct options
         ."
    opt_num: 4
    amount: [beer_num]
    cond: all
    opt_formula: "sum([get_value(_model, wine_s[
         wines[_opt[0][i]]].get('belonging')) == 0
         for i in range(beer_num)]) == beer_num"
    opt_text: "{','.join([str(vol[_opt[0][i]]) for i
          in range(beer_num)])}"
desc: >
  A merchant has {wine_num} barrels of wine and beer
       with volumes: {','.join([str(v) + ' gallons'
       for v in vol])}.
  There are {wine_num - beer_num} barrels of wine. {
       wine_0}, {wine_1}, and {wine_times}.
  No wine is left. {question}
```

Listing 6: DSL specification for the “Product” puzzle.

```
variables:
  p_num: {type: int, domain: "[6, 17]", diff_factor:
       1}
  products: {formula: get_faker(p_num, 'product')}
  names: {formula: get_faker(1, 'name')}
symbols:
  pos: # The position of each product on a conveyor
       belt
    source: [products]
    type: int
conditions:
  pos_domain: {formula: "And([And(pos[x] >= 1, pos[x
       ] <= p_num) for x in products])"}
  pos_distinct: {formula: "gen_event_count_condition
       (pos, 'distinct')"}
  cond1: # Relative distance between products
    source: [products, "range(1, p_num - 2)"]
    amount: ['2', '1']
    domain: "[p_num // 2, p_num]"
    formula: "Or(pos[_sym[0][0]] - pos[_sym[0][1]]
         == _sym[1][0], pos[_sym[0][1]] - pos[_sym
         [0][0]] == _sym[1][0])"
    desc: "{_sym[0][0]} and {_sym[0][1]} have {'no'
         if _sym[1][0] == 1 else str(_sym[1][0] - 1)
         } items between them."
  cond2: # Adjacent products
    source: [products]
    amount: ['2']
    domain: "[1, p_num // 2]"
    formula: "pos[_sym[0][1]] - pos[_sym[0][0]] ==
         1"
    desc: "{_sym[0][1]} is placed immediately after
         {_sym[0][0]}."
  cond3: # Absolute position constraint
    source: [products, "range(1, p_num+1)"]
    amount: ['1', '1']
    domain: "[0, p_num // 3]"
    formula: "pos[_sym[0][0]] != _sym[1][0]"
    desc: "{_sym[0][0]} is not in position {_sym
         [1][0]}."
max_solution: 600
queries:
  question:
    source: [products, "range(1, p_num+1)"]
    desc: "If the statements are true, which of the
         following must be true?"
    opt_num: 6
    amount: ['1', '1']
    cond: all
    opt_formula: "get_value(_model, pos[_opt[0][0]])
         == _opt[1][0]"
    opt_text: "{_opt[0][0]} is in position {_opt
         [1][0]}."
desc: >
  After shopping, {names[0]} placed {p_num} items on
       a conveyor belt,
  ordered from front to back (pos 1 to {p_num}). {
       conditions} {question}
```

Listing 7: DSL specification for the “Exam” puzzle.

```
variables:
  p_num: {type: int, domain: "[4, 10]", diff_factor:
       1}
  exam_num: {type: int, domain: "[2, p_num]",
      diff_factor: 1}
  num_performed_well: {type: int, domain: "[1, p_num
       - 1]"}
  names: {formula: get_faker(p_num, 'name')}
  exams: {formula: get_faker(exam_num, 'major')}
symbols:
  performed_well: {source: [names], type: bool}
  passed: {source: [names, exams], type: bool}
conditions:
  num_performed_well_cond:
    formula: gen_event_count_condition(
        performed_well, 'equal', num_performed_well
        )
    desc: "Only {num_performed_well} of {p_num}
        people performed well."
  cond1:
    source: [exams]
    amount: ['2']
    domain: "[p_num, p_num]"
    domain_cond: false
    formula: >
      And(Implies(performed_well[names[_index]],
          passed[(names[_index], _sym[0][0])]),
          Implies(Not(performed_well[names[_index]])
              , Not(passed[(names[_index], _sym
              [0][1])])))
    desc: '{names[_index]} says: "If I perform well,
         I will pass {_sym[0][0]}. If not, I will
        fail {_sym[0][1]}."'
  only_one_who_passed_some_exam:
    source: ["['False', 'True']"]
    amount: ['2']
    domain: "[2, 2]"
    formula: >
      Or([Sum([If(And(passed[(p, e)] == _sym[0][1],
          performed_well[p] != _sym[0][0]),
          1, 0) for p in names]) == 0 for e in exams
              ])
    desc: >
      For one subject, only those who performed {'
          well' if _sym[0][0] else 'poorly'}
      {'' if _sym[0][1] else 'did not '}pass.
max_solution: 500
queries:
  question:
    desc: "Who performed well?"
    ans_formula: to_unique([[p for p in names if
        get_value(_model, performed_well[p])] for
        _model in _solutions])
    ans_text: "','.join(_ans[0])"
    ans_assertion: len(_ans) <= 1
desc: >
  {p_num} people ({', '.join(names)}) took {exam_num
      } exams ({', '.join(exams)}).
  {num_performed_well_cond} Statements: {cond1} Also
      : {only_one_who_passed_some_exam} {question}
```

The prompt for seed puzzle selection

CHN

我是一名数据科学研究员，想通过一道种子题目合成大量与之类似的衍生题。而你是一个数学逻辑问题分析专家，你需要帮助我判断一道题是否合适作为种子题。

判断依据

1. 是否存在变量值或逻辑条件可以随机替换，并且不会改变题目的语义？
2. 是否能将该题目转换为 SMT（可满足性模理论）问题，并通过 SMT 求解器（如 z3）在可接受的时间和内存范围内求解？

如果以上两个条件均成立，请返回 `is_seed_puzzle` 为 `True`，同时返回 `code` 为你的 python 求解代码；否则返回 `is_seed_puzzle` 为 `False`，同时返回 `code` 为 `null`。
`reason` 为你的判断依据，当`is_seed_puzzle` 为 `False`时，需要在 `reason` 中说明不符合哪项依据。
例如：给定一个问题和答案：

```json
{
  "question": "有一个正整数x加上13是一个平方数，x加上5是一个立方数，那么x的值是多少？",
  "answer": "3"
}
```

期望的回答为：

```json
{
  "is_seed_puzzle": true,
  "reason": "该题目中存在可以随机替换的变量值（如13和5）和逻辑条件（平方数和立方数），且可以通过SMT求解器在可接受的时间和内存范围内求解。",
  "code": "from z3 import *\n\nx = Int('x')\nsolver = Solver()\n\n# 定义条件\nsolver.add(x > 0)\nsolver.add(Exists([y], y * y == x + 13))\nsolver.add(Exists([z], z * z * z == x + 5))\n\n# 求解\nif solver.check() == sat:\n    m = solver.model()\n    print(m[x])\nelse:\n    print(\"No solution found\")"
}
```

待分析的题目

```json
{
  "question": {question},
  "answer": {answer},
}
```

回复格式

请直接返回一个 JSON 对象，格式如下：

```json
{
  "is_seed_puzzle": <true or false>,
  "reason": <reason>,
  "code": <python code or null>
}
```

注：请确保输出仅为 JSON 数据，不要返回其他文本或解释。

ENG

I am a data science researcher aiming to generate a large number of derivative puzzles from a single seed puzzle. You are an expert in mathematical logic problem analysis, and your task is to determine whether a given puzzle is suitable to be used as a seed puzzle.

Evaluation Criteria

1. Are there variable values or logical conditions in the puzzle that can be randomly replaced without altering the semantics of the question?
2. Can the puzzle be translated into an SMT (Satisfiability Modulo Theories) problem and solved using an SMT solver (e.g., Z3) within tractable time and memory bounds?

If both criteria are satisfied, return `is_seed_puzzle` as `true`, and provide the corresponding Python code in `code`.
Otherwise, return `is_seed_puzzle` as `false`, and set `code` to `null`. The `reason` field should contain your justification, and if `is_seed_puzzle` is `false`, please specify which criterion is not satisfied.

For example, given the following question and answer:

```json
{
  "question": "A positive integer x plus 13 is a perfect square, and x plus 5 is a perfect cube. What is the value of x?",
  "answer": "3"
}
```

The expected response would be:

```json
{
  "is_seed_puzzle": true,
  "reason": "The puzzle contains variable values (e.g., 13 and 5) and logical conditions (perfect square and cube) that can be randomly replaced without changing the semantics. It can also be solved within reasonable time and memory limits using an SMT solver.",
  "code": "from z3 import *\n\nx = Int('x')\nsolver = Solver()\n\n# Define the constraints\nsolver.add(x > 0)\nsolver.add(Exists([y], y * y == x + 13))\nsolver.add(Exists([z], z * z * z == x + 5))\n\n# Solve\nif solver.check() == sat:\n    m = solver.model()\n    print(m[x])\nelse:\n    print(\"No solution found\")"
}
```

Puzzle to Analyze

```json
{
  "question": {question},
  "answer": {answer}
}
```

Response Format

Please respond with a JSON object in the following format:

```json
{
  "is_seed_puzzle": <true or false>,
  "reason": <reason>,
  "code": <python code or null>
}
```

Note: Ensure the output is strictly a JSON object, without any additional text or explanation.

Figure 6: Seed Puzzle Selection Prompt for filtering seed puzzles via the Qwen2.5-72B-Instruct model.

Field	Type	Description
custom_operator	dict (opt)	A dictionary of custom operators. The key is the operator name (e.g., "double"), and the value is a string containing either a Python lambda function (e.g., "lambda x: x*2") or the path to a Python file defining the operator.
variables	dict	A dictionary defining the puzzle's parameters. The key is the variable name and the value is a Variable object. See Table 7 for details.
symbols	dict (opt)	A dictionary defining the symbols to be solved. Keys are symbol group names. Values can be DefinedSymbol, DerivedSymbol, or DerivedSymbols objects. See Tables 8 and 9.
conditions	dict (opt)	A dictionary of constraints. Keys are condition names. Values can be StaticCondition or DynamicCondition objects. See Tables 10 and 11.
calc_solution	bool	If true (default), the solver will compute the set of valid solutions for the generated puzzle instance.
max_solution	int	The maximum number of valid symbol configurations to find. If the solver exceeds this limit (default 6000), it halts. This refers to satisfying all constraints, not the final answer to a query.
post_generation	PostGen (opt)	Defines operations to be performed after the initial solution is found, such as deriving new variables from the solution. See Table 14.
optimize	Optimize (opt)	Defines an optimization objective for problems that require minimizing or maximizing a certain value. See Table 15.
queries	dict (opt)	A dictionary of questions for the puzzle. The key is the query name and the value is a Query or a selection-based query object. See Tables 12 and 13.
desc	str	A natural language template for the puzzle's introductory text, which can include placeholders for variables.

Table 6: Schema for the PuzzleTemplate Root Object.

Field	Type	Description
type	str (opt)	The data type of the variable, e.g., "int", "bool". Must be defined if formula is not defined.
domain	str (opt)	A string representing the variable's value space. Examples: "[1, 10]" for integers, "['red', 'blue']" for string options. Must be defined if formula is not defined.
formula	str (opt)	A Python expression string to compute the variable's value. Example: "randint(1,6) + randint(1,6)". If defined, type and domain must be omitted.

Table 7: Schema for the Variable Object.

Field	Type	Description
`source`	`list[str]`	A list of string expressions that serve as primary keys. For example, if `source` is `["children"]` and the variable `children` is `["Alice", "Bob"]`, two symbols are created.
`attr`	`list[str]` (opt)	A list of attribute names for the symbol (e.g., `["color", "size"]`). If not provided, the symbol is a simple value. If provided, the symbol is a dictionary-like object with these attributes.
`type`	`str` or `list[str]`	The Z3 type(s) for the symbol (e.g., `"Int"`, `"Bool"`). Must be a single string if `attr` is not defined. Must be a list of strings matching the length of `attr` if it is defined.
`desc`	`str` or `list[str]` (opt)	A natural language description template. Follows the same format constraints as `type` based on the presence of `attr`.

Table 8: Schema for the `DefinedSymbol` Object.

Field	Type	Description
source	list[str]	Data sources for selection. Each string is an expression evaluating to a list (e.g., a variable name or a literal list like "[1, 2, 3]").
amount	list[str] (opt)	The number of items to select from each corresponding source. If omitted, one item is selected from each source. Length must match source.
order	list[bool] (opt)	If true for a source, selection order matters (permutation). If false, it does not (combination). Defaults to all true. Length must match source.
duplicate	list[bool] (opt)	If true for a source, items can be selected more than once. Defaults to all false. Length must match source.
domain	str (opt)	The total number of symbols (selections) to generate. Can be an integer literal or a variable name.
domain_cond	bool	If true (default), identical symbol combinations are disallowed across the entire selection process.
dim	int	The number of dimensions for the symbol (default 1). Useful for creating matrices of related symbols.
dim_cond	list (opt)	A list of lists specifying inter-dimensional constraints. Each inner list contains source indices whose selected values cannot be identical.
custom_cond	list (opt)	A list of custom constraint dictionaries. Each dictionary specifies a scope ("domain" or "dim"), a list of source fields, and a Python lambda constraint string.
formula	str (opt)	A Python expression string that defines the Z3 constraint for the generated symbol.
desc	str	A natural language description template for the set of generated symbols.

Table 9: Schema for the DerivedSymbol Object.

Field	Type	Description
formula	str	The constraint logic as a Python expression string (e.g., "x + y < 10").
desc	str (opt)	The corresponding natural language description for the puzzle text.

Table 10: Schema for the StaticCondition Object.

Field	Type	Description
source	list[str]	Data sources for parameter selection, same as in DerivedSymbol.
amount	list[str] (opt)	Number of items to select from each source, same as in DerivedSymbol.
domain	str (opt)	The number of conditions to generate, specified as a range string (e.g., "[1, 5]"). If omitted, one condition is generated.
domain_cond	bool	If true (default), identical parameter combinations are disallowed.
custom_cond	list (opt)	Custom constraints on parameter selection, same as in DerivedSymbol.

Table 11: Additional fields for the DynamicCondition Object.

Field	Type	Description
desc	str	The natural language text of the question.
ans_formula	str	A Python expression to compute the correct answer from the solution.
ans_text	str	A template for formatting the display of the answer.
ans_assertion	str (opt)	A Python expression that must evaluate to true for the answer to be considered valid (e.g., to ensure a unique solution exists).

Table 12: Schema for the Query (Open-Ended) Object.

Field	Type	Description
Base fields for all selection queries include desc, query_type, select_type, *and* opt_num.		
source	list[str]	Data sources for generating option parameters.
cond	str	Condition scope for an option to be correct/incorrect. "any": satisfies at least one solution; "all": satisfies all solutions.
opt_formula	str	A Python expression that evaluates the correctness of a generated option.
opt_text	str (opt)	A template for the display text of the option.
custom_cond	list (opt)	Custom constraints on option parameter selection.

Table 13: Schema for the QuerySelectionTemplate Object (for multiple-choice options).

Field	Type	Description
post_gen_vars	dict (opt)	A dictionary to define new variables. Keys are new variable names, values are Python expressions to extract values from the solution.
post_gen_conditions	dict (opt)	A dictionary to add new constraints. Keys are new constraint names, values are StaticCondition objects.

Table 14: Schema for the PostGen Object.

Field	Type	Description
`type`	`str`	The optimization type: `"minimize"` or `"maximize"`.
`formula`	`str`	The formula representing the value to be optimized.

Table 15: Schema for the `Optimize` Object.

Category	Setting	Value
Objective	Task	Supervised fine-tuning (prompt: `chat_template`)
Model & Optimization	Base model	`Qwen2.5-7B-Instruct`
	Model max length	32,768
	Per-device batch size	1
	Grad accumulation steps	4
	Learning rate	1×10^{-5}
	Weight decay	0.05
	Warmup ratio	0.03
	LR scheduler	cosine
Regularization & Checkpointing	Gradient checkpointing	True
	Filter by length	True
	Save only model	True
Precision	Numeric formats	bf16=True; tf32=True
Distributed / System	DeepSpeed	ZeRO-3 Offload
	Dataloader workers	1
Training & Logging	Epochs	6
	Evaluation	`eval_strategy=no`
	Saving	`save_strategy=steps`; `save_steps=615`; `save_total_limit=999`
	Logging	`logging_steps=1`; `report_to=none`
Compute	GPUs	16 (8 H100 per node $\times$ 2)

Table 16: Key hyperparameters for SFT training.

Category	Setting	Value
Algorithm	RL type	PPO with GRPO advantage estimator
	KL regularization	use_kl_loss=True; kl_loss_coef=0.001; kl_loss_type=low_var_kl; use_kl_in_reward=False; entropy_coeff=0
	Critic warmup	0
Model & Optimization	Base model	`Qwen2.5-7B-Instruct`
	Learning rate	1×10^{-6}
	Gradient checkpointing	True
	PPO mini-batch size	128
	Dynamic batch size	True
	Max token len / GPU (PPO)	24,000
Rollout / Inference	Engine	vLLM; use_remove_padding=True
	Generations per prompt	$n = 5$
	Max batched tokens (rollout)	5,120
	GPU memory utilization	0.6
	Tensor model parallel size	2
Distributed / FSDP	Strategy	fsdp2 (actor / ref / critic / reward model)
	Offload	actor: param=False, optimizer=False; ref: param=True
Training & Logging	Epochs	5
	Validation	val_before_train=True
	Save / Test freq (steps)	save=290; test=80
	Logger	console, tensorboard
Compute	GPUs	16 (8 H100 per node $\times$ 2)

Table 17: Key hyperparameters for GRPO training.

The prompt for wrapping puzzles

A Single Choice

CHN 本题是单选题（只有一个正确选项），请阅读题目后用中文一步一步推理，并将您认为的正确选项填写在\boxed{}中。

ENG This is a multiple-choice question (only one correct option). Please read the question carefully and reason step by step in Chinese, then put your final selected option inside \boxed{}.

B Short Answer

CHN 本题是简答题，请阅读题目后用中文一步一步推理，并将您的最终精简答案放在\boxed{}中。

ENG This is a short-answer question. Please read the question carefully and reason step by step in Chinese, then place your concise final answer inside \boxed{}.

C Fill In The Blanks

CHN 本题是填空题，请阅读题目后用中文一步一步推理，填补题目中空留的部分，并将这些填补内容全部放在\boxed{}中。

ENG This is a fill-in-the-blank question. Please read the question carefully and reason step by step in Chinese to fill in the missing parts, then put all filled contents inside \boxed{}.

D Multiple Queries

CHN 以下共有**{qtype_len}**道小题，分别是**{qtype_list}**，请用中文一步一步推理，并将这些小题的最终答案按顺序统一放在回答最末尾，并用\boxed{}将最终答案包裹。如果是单选题，请将正确选项作为小题的最终答案；如果是简答题，请将精简后的答案作为小题的最终答案。

ENG There are **{qtype_len}** sub-questions below, which are **{qtype_list}**. Please reason step by step in Chinese, and summarize all final answers in order at the end of your response, wrapped inside \boxed{}. For single choice questions, provide the correct option; for short-answer questions, give the concise final answer.

Figure 7: Question-Type Prompt Wrappers for LLM reasoning/training.

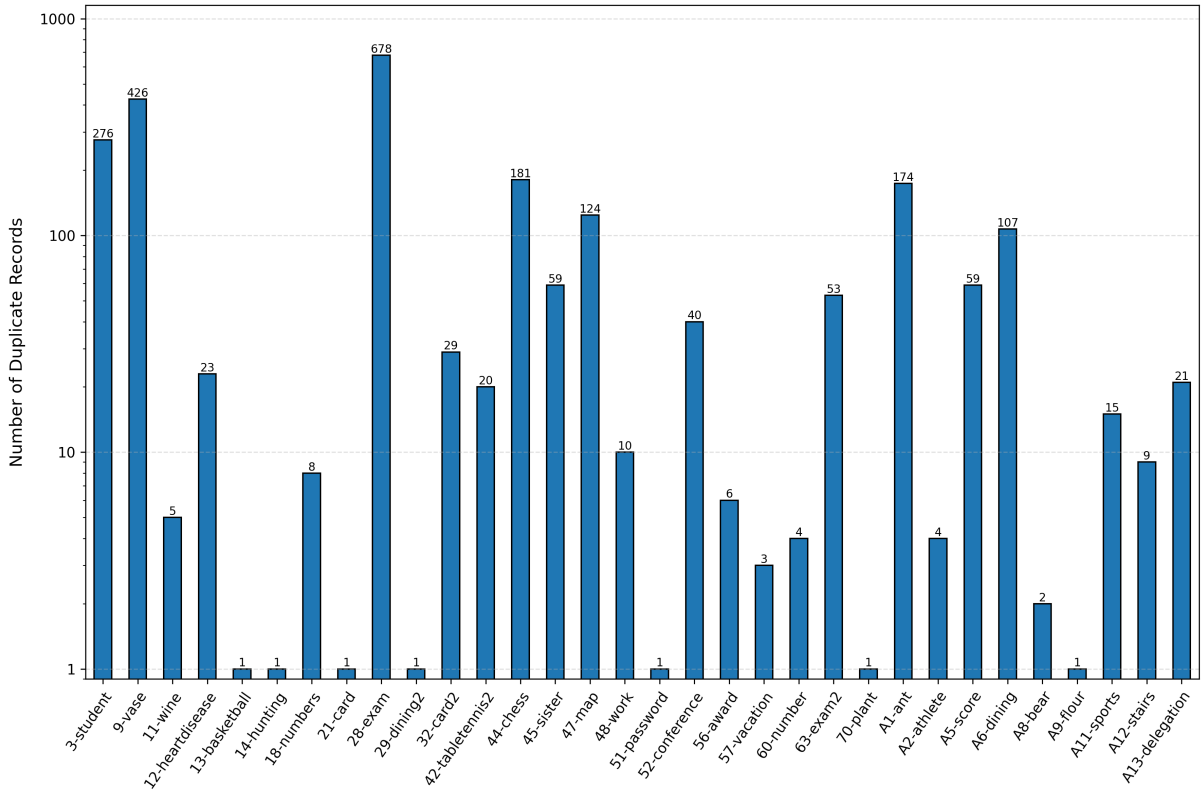

Figure 8: Distribution of duplicated instances across 32 seed puzzles. Only seed puzzles with at least one duplicate are shown.

Figure 9: The average accuracy of all models on the PuzzleClone test set grouped by the seed puzzles.