TPLP **26** (1-2): 253–266, 2026. © The Author(s), 2026. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068426100489 First published online 8 July 2026

From Reasoning to Code: GRPO Optimization for Underrepresented Languages

FEDERICO PENNINO, BIANCA RAIMONDI, MASSIMO RONDELLI, ANDREA GURIOLI and MAURIZIO GABBRIELLI
DISI, Università di Bologna, Italy
(*e-mail:* federico.pennino2@unibo.it)

submitted 23 May 2026 UTC; revised 23 May 2026 UTC; accepted 28 May 2026 UTC

Abstract

Generating accurate and executable code using Large Language Models (LLMs) remains a significant challenge for underrepresented programming languages, such as Prolog and Lisp, due to the scarcity of public training data compared to high-resource languages like Python. This paper introduces a generalizable Reinforcement Learning (RL) approach that combines small-scale versions of the `Qwen2.5-Coder` model with Group Relative Policy Optimization (GRPO) to enable effective code generation through reasoning. To address the limitations of sparse datasets, we integrate execution-driven feedback directly into the RL loop, utilizing a reward system that exploits both logical correctness and structural formatting. Experimental results on GSM8K dataset demonstrate significant improvements in reasoning quality and code accuracy across underrepresented languages. These findings underscore the potential of our approach to benefit a wide range of programming languages lacking extensive training resources by leveraging symbolic reasoning and interpreter-based feedback.

Keywords: GRPO, logic programming, functional programming

1 Introduction

The resurgence of connectionist methods over the past decade has produced Large Language Models (LLMs) of remarkable capability, yet the field of Artificial Intelligence has long drawn on a second tradition: symbolic reasoning, grounded in logic, formal verification, and languages such as Prolog, a tradition exemplified by the Fifth Generation Computer Systems project discussed by Moto-Oka and Stone (1984). These two paradigms have historically developed in isolation, and bridging them remains a central challenge for the field.

From the programming languages perspective this bridge could mean an important (last?) step toward the "Holy Grail" of programming introduced by Freuder (1997): "The user states the problem and the computer solves it." This dream has always been a fil rouge of most research in logic programming. We are now very close to making this dream come true, and one promising path is deceptively simple: teach neural models to *write* the symbolic programs themselves. If an LLM can reliably generate executable

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

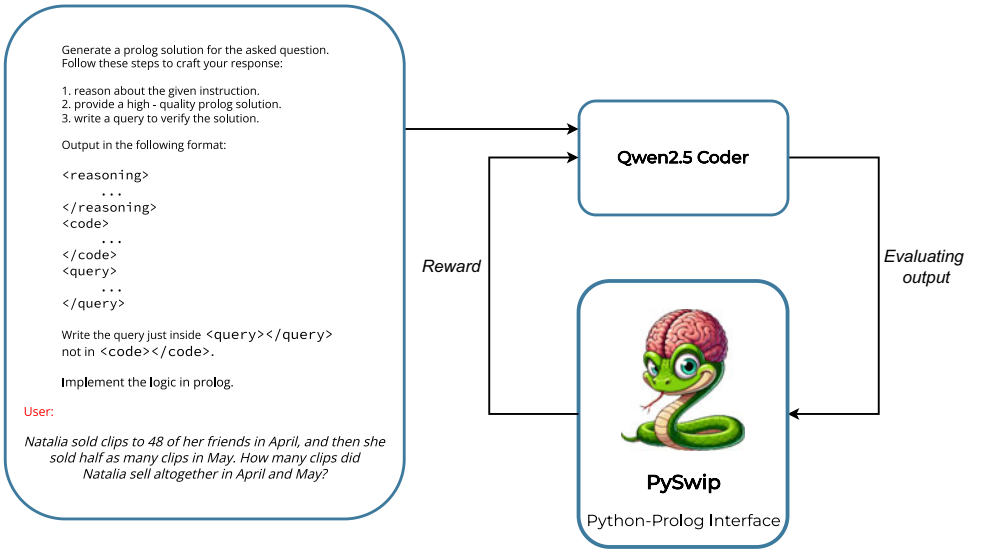


Fig. 1. Schematic representation of the GRPO training pipeline. The framework utilizes `Qwen2.5-Coder` to generate Prolog solutions, which are then evaluated via `PySwip` to provide execution-based reward signals.

Prolog code from natural-language specifications, it becomes both a natural-language interface to formal reasoning and a system whose outputs are inspectable, verifiable, and grounded in logic, which are properties that purely neural generation cannot guarantee.

In practice, however, this vision runs into an immediate obstacle. LLMs have achieved remarkable proficiency at generating code in mainstream programming languages such as Python and JavaScript, yet this proficiency does not transfer to logic programming. When models such as `OpenAI Codex`, `Code-Llama` by Rozière *et al.* (2023), `StarCoder` by Li *et al.* (2023), and `Qwen2.5-Coder` by Hui *et al.* (2024) are asked to produce Prolog programs, they consistently fail: they hallucinate non-existent built-in predicates, impose imperative control flow on a declarative paradigm, and mishandle the interplay of unification and backtracking that defines logic programming. The root cause is not architectural but distributional: these languages are scarce in modern pre-training corpora, and no amount of scale will compensate for data that does not exist.

This paper shows that the obstacle can be addressed by training small, open-weight models with reinforcement learning (RL), using a Prolog interpreter as the source of the signal rather than scarce human-written examples, following the code-evaluation framing of Chen *et al.* (2021). More precisely, we explore a methodology that, rather than fine-tuning a model to imitate scarce examples, shifts the learning objective to *maximizing execution correctness* as judged by a SWI-Prolog interpreter embedded in the training loop, as shown in Figure 1. This is an instance of RL with Verifiable Rewards (RLVR), discussed by Wen *et al.* (2025), in which the symbol system serves not as a tool for the neural network but as a teacher that supervises it.

To implement this idea, we adopt Group Relative Policy Optimization (GRPO), introduced by Shao *et al.* (2024), a policy-gradient algorithm that eliminates the separate

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

value-function network required by Proximal Policy Optimization. GRPO estimates the advantage of each generated output relative to a group of peer outputs sampled from the same prompt, reducing computational overhead and stabilizing learning in the sparse-reward regime characteristic of code-generation tasks.

We apply GRPO to the Qwen2.5-Coder-Instruct family described by Hui *et al.* (2024), scaling from 0.5B to 7B parameters, and train on the GSM8K dataset introduced by Cobbe *et al.* (2021), whose emphasis on structured, logical problem-solving makes it a valuable proxy for the reasoning skills.

First, we identify a failure mode we term *reasoning contraction*: under standard GRPO training with a Kullback–Leibler (KL) divergence penalty, the 7B model collapses to short, memorized output patterns, and its accuracy on GSM8K degrades over training steps (from 0.814 to 0.672 in one-shot). We attribute this to the KL penalty anchoring the policy too closely to the supervised baseline, thereby impeding the distributional shift required to internalize a fundamentally different programming paradigm. Second, we show that combining the removal of the KL constraint with a length-aware reward mechanism resolves this contraction: the model's completions expand, its reasoning chains deepen, and its one-shot pass@4 on GSM8K rises to 0.893, thus outperforming both `PROPER`, introduced by Yang *et al.* (2024), a variant of `Mistral-7B`, presented by Jiang et al. (2023), with specialized supervised fine-tuning (0.72), and substantially larger models including `Llama 3.3 70B` (0.839), from the model family described by Llama Team (2024), and `Qwen2.5-Coder-32B` (0.873). This work makes four contributions:

1. It demonstrates that RLVR with execution-based feedback can compensate for extreme data scarcity, establishing a new state of the art for Prolog code generation on GSM8K with a 7B-parameter model.
2. It provides a systematic ablation of GRPO training dynamics for logic programming, identifying reasoning contraction as a failure mode of KL-regularized RL and characterizing the conditions under which relaxing this constraint is beneficial.
3. It shows that the resulting gains generalize beyond arithmetic reasoning: improvements transfer to 20 diverse Prolog tasks from Rosetta Code and the GSM-Symbolic benchmark, where zero-shot p2 accuracy more than doubles (from 0.298 to 0.673).
4. It validates the generality of the approach by applying the same framework to Lisp, where zero-shot pass@4 rises from 0.572 to 0.876, suggesting that interpreter-driven RL is effective across distinct families of underrepresented languages.

Taken together, these results demonstrate that for formal languages with a deterministic semantics, the interpreter itself can serve as the primary training signal. This replaces the scarce human data on which current approaches depend, providing a rigorous, automated supervisor that teaches the model to respect the syntax and logic of the target language.

2 Related work

Program synthesis, the task of automatically generating executable code from high-level specifications, has traditionally been the domain of symbolic AI. Inductive Logic

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

Programming (ILP), a subfield prominent in the 1990s, used formal logic to infer hypotheses (programs) from positive and negative examples. Systems such as Progol and Aleph can learn recursive logic programs given a strict background knowledge base. However, ILP systems faced severe scalability limitations; they struggled with noise, required extensive domain engineering, and could not process natural language ambiguity.

The deep learning revolution introduced Neural Program Synthesis, replacing the discrete search of ILP with the continuous optimization of neural networks. Early sequence-to-sequence models (e.g. LSTMs) treated code generation as a translation task (Natural Language $\rightarrow$ Code), but struggled with long-range dependencies and the hierarchical structure of abstract syntax trees. The Transformer architecture introduced by Vaswani *et al.* (2017) and the pre-training paradigm (BERT, GPT) fundamentally altered this landscape.

Modern Code LLMs are decoder-only Transformers trained on vast corpora of source code (e.g. The Stack, GitHub). Despite these advances, the performance of these models is highly uneven. While they excel at Python or Java, their ability to generate Prolog or Lisp is often superficial and merely mimics the syntax (indentation, parentheses) without understanding the semantics (unification, recursion, macro expansion). This is a direct result of the distribution of training data: Prolog code constitutes a vanishingly small percentage of GitHub repositories compared with imperative languages. To address the reliability issues of LLMs in reasoning tasks, researchers have developed frameworks that combine neural intuition with symbolic rigor.

Chain-of-Thought (CoT) prompting encourages LLMs to emit intermediate reasoning steps before the final answer. While this significantly improves performance on math benchmarks such as GSM8K, it suffers from the "validity hallucination" problem: the model can generate a plausible-sounding reasoning chain that contains logical errors, leading to an incorrect answer. Furthermore, CoT relies entirely on the internal representations of the LLM, which are probabilistic and prone to arithmetic drift.

This led to Program-of-Thought (PoT) prompting, introduced by Chen *et al.* (2021), which advances CoT by decoupling reasoning from computation. In PoT, the LLM generates a program (usually Python) to solve the problem, which is then executed by an external interpreter. This offloads calculation to a deterministic engine, significantly boosting accuracy on arithmetic tasks.

Recent work has identified Prolog as a superior target for PoT in logical reasoning tasks. GSM8K-Prolog, presented by Yang *et al.* (2024), demonstrated that mapping math word problems to Prolog predicates yields higher accuracy than Python PoT for problems requiring constraint satisfaction or symbolic manipulation. Unlike Python, which is imperative (specifying how to calculate), Prolog is declarative (specifying what is true). This aligns more closely with the semantic parsing of natural language logic. The same work also introduced ProPro (Predicate Permutation), a data augmentation method that leverages the commutative property of Prolog rules (i.e. A, B is logically equivalent to B, A in pure Prolog) to improve model robustness.

While SFT effectively teaches syntax, it does not directly optimize for correctness. RL allows models to optimize non-differentiable metrics, such as "does this code compile?" or "does this code pass test cases?" This paradigm leverages the fact that, in domains such as mathematics and programming, the validity of an output can be objectively

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

verified. Le *et al.* (2022) introduced CodeRL, which employed an actor-critic architecture in which unit test results served as the reward signal. RLEF (Reinforcement Learning with Execution Feedback), presented by Gehring *et al.* (2025), further demonstrated that models can learn to debug their own code by receiving execution traces as feedback, thereby creating an iterative improvement loop. Based on this, we target a setting in which both the training data and the test suites are scarce. By embedding a Prolog interpreter directly into the reward loop, we obtain a dense, ground-truth correctness signal without needing hand-written tests, thereby turning the interpreter from a development tool into a training supervisor.

3 Methodology

3.1 Models and datasets

This study mainly examined smaller-scale LLMs for Prolog code generation, employing the 0.5B, 1.5B, 3B, and 7B versions of `Qwen2.5-Coder-Instruct`. These models were chosen for their effectiveness and low resource demands.

A key challenge was the limited availability of publicly accessible Prolog code for supervised training, significantly restricting the models' initial exposure to Prolog's syntax. To address this, we adopted a RL strategy using the GSM8K dataset, which, although primarily designed for arithmetic reasoning, its emphasis on structured, logical problem-solving makes it a valuable proxy for evaluating and improving the reasoning skills necessary for Prolog code generation. Training used only the official GSM8K set, and all reported metrics are computed on the official test split to prevent train-test leakage.

Figure 1 shows an example of the execution pipeline: the LLM learns to map natural language prompts to executable Prolog programs via feedback from execution results.

To evaluate the generalization capabilities of our post-training models, we built a 20-task Prolog dataset sourced from Rosetta Code (2007), including common algorithmic tasks: Fibonacci sequence, Sieve of Eratosthenes, Quicksort, Binary search, Greatest common divisor, Factorial, Towers of Hanoi, Palindrome detection, Prime decomposition, Dijkstra's Algorithm, Levenshtein distance, N-queens problem, Ackermann function, Balanced brackets, Knight's tour, Merge sort, Roman numerals decode, Longest common subsequence, Huffman coding, 24 game. These 20 tasks were selected to represent core logic programming constructs, including recursion, list manipulation, and symbolic backtracking.

We also evaluated models on GSM-Symbolic, a benchmark focused on symbolic reasoning over abstract mathematical expressions. This dataset was created by Mirzadeh *et al.* (2025) solely for testing, with the aim of assessing the generalization capabilities of LLMs trained on GSM8K. We included its variants: GSM-Symbolic-Plus-1 (P1), in which each question includes one additional reasoning clause, and GSM-Symbolic-Plus-2 (P2), with two additional clauses. These variants increase the complexity of reasoning in a controlled manner: for instance, P2 tasks may involve multi-step comparisons across multiple entities, whereas the base version focuses on single-step calculations. The GSM-Symbolic setup enables fine-grained analysis of symbolic-reasoning depth, extending beyond GSM8K's arithmetic focus. By systematically modifying numerical values

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

Table 1. *Reward function breakdown*

Reward function	Reward range
`xmlcount_reward_func`	$[-0.5, 0.625]$
`strict_format_reward_func`	$\{0, 0.5\}$
`soft_format_reward_func`	$\{0, 0.5\}$
`correctness_reward_func`	$\{-1, -0.5, 1\}$

and contextual details while preserving the underlying logical structure, we can assess whether the models have learned to extract and encode the core reasoning patterns into Prolog predicates and rules, or whether their translations are overfitted to specific problem templates.

3.2 Experimental setup

To standardize the training process and ensure reproducible results, we conducted experiments with common customizations. We used TRL, introduced by von Werra *et al.* (2020), for the training phase, alongside Low-Rank Adaptation, introduced by Hu *et al.* (2022), with a rank of 32, to enable efficient adaptation and fine-tuning across these configurations. The evaluation process exploited varying hyperparameters:

- Model size: Different configurations of the model size were tested, corresponding to 7B, 3B, 1.5B, and 0.5B.
- Checkpoints: The model was evaluated at three different training checkpoints 500, 1000, 1500.
- Zero-shot *versus* One-shot: The model was trained in both one-shot and zero-shot learning to measure its ability to generalize based on the context provided.

The resulting models were appropriately prompted based on the learning hyperparameter to be consistent with the prompt setting. The testing script executed inference using the vLLM framework, introduced by Kwon *et al.* (2023), on a multi-GPU server equipped with 8xNVIDIA A100-SXM4 GPUs (80 GB VRAM each), running CUDA 12.2 to ensure efficient execution of large-scale language models. Training and inference were performed on a single GPU unless otherwise specified.

Evaluation metrics and experiment scripts were standardized to facilitate reproducibility and future benchmarking.

3.3 Reward system

The model is trained using GRPO, refining code generation through a multi-objective reward function. The training objective enforces two primary constraints: adherence to the system-prompted template (Figure 1) and the functional correctness of the generated Prolog code. Total rewards are calculated by aggregating the individual signals summarized in Table 1.

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

3.3.1 Format reward

To ensure the Prolog code is extractable and interpretable, we enforce a structured format using < `reasoning` >, < `code` >, and < `query` > tags. This mechanism serves as an early-stage signal.

- Tag Presence (`xmlcount`): Grants + 0.125 for each required tag. A penalty of −0.5 is applied if the query is incorrectly nested inside the < `code` > block to prevent execution errors;
- Strict Evaluation: A binary + 0.5 reward if the output matches the template exactly;
- Soft Evaluation: A binary + 0.5 reward if the code is extractable via regex, even if minor formatting errors exist outside the tags.

This structured formatting strategy increases sample efficiency by penalizing structural deviations, allowing the model to quickly converge on a consistent and executable output format.

3.3.2 Correctness reward

We evaluate semantic correctness by executing the generated code via PySwip, a Python interface for SWI-Prolog. The environment is sandboxed with a timeout to prevent infinite loops. The reward is assigned based on the interpreter's output compared to the ground truth:

- Success (+1.0): The execution result matches the expected answer exactly;
- Logical Error (−1.0): The code executes but returns an incorrect result;
- Syntax Error (−0.5): The code is malformed and fails to compile/execute;
- Timeout/No Output (−0.1): The interpreter fails to return a result within the limit of 5 s.

By penalizing syntax errors more heavily than timeouts, we prioritize generating "runnable" code during the initial phases of training.

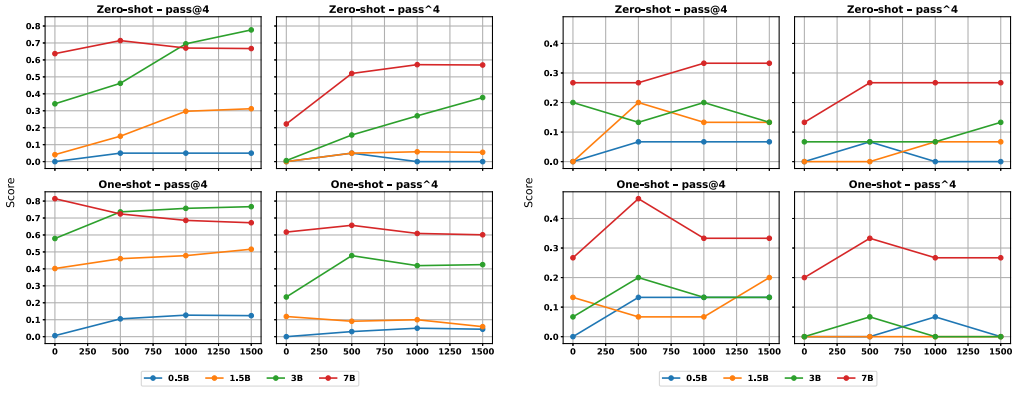
3.4 Evaluation metrics

We assess the model's capability to generate functionally correct Prolog code through evaluation. For each problem instance in our dataset, the model generates $k = 4$ candidate solutions. We employ two primary metrics based on this execution framework to quantify performance: Pass@k and Pass^k.

Pass@k This metric measures the success rate considering the k generated candidates per problem. A problem is deemed successfully solved if *at least one* of the k generated solutions produces the correct output when executed. It is calculated over the entire dataset of N problems as shown in equation (1). In this formula, N is the total number of problems, c_i^j is the j-th code for the i-th problem, $P(c_i^j)$ is its execution result via interpreter P, r_i is the ground truth, and the disjunction ($\bigvee$) captures the "at least one" success condition.

$$pass@k = \frac{1}{N} \sum_{i=1}^{N} \left(\bigvee_{j=1}^{k} P\left(c_i^j\right) = r_i \right) \tag{1}$$

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press



(a) GSM8K (b) Rosetta Code

Fig. 2. pass@4 and pass^4 results for `Qwen2.5-Coder` models in zero-shot and one-shot settings.

Pass^k Providing a stricter assessment of performance, this metric requires *all* k generated solutions for a given problem to execute correctly. It measures the model's consistency and reliability in generating valid code. The formal definition is shown in equation (2).

$$pass^{\wedge}k = \frac{1}{N}\sum_{i=1}^{N}\left(\bigwedge_{j=1}^{k} P\left(c_i^j\right) = r_i\right) \tag{2}$$

The notation follows equation (1), but the conjunction ($\bigwedge$) enforces the requirement that all k generated candidates must match the ground truth r_i for the problem to be counted as correct. This stricter metric evaluates the model's reliability across multiple generations of attempts.

4 Results and discussion

4.1 GSM8K dataset

The performance of `Qwen2.5-Coder` models on the GSM8K benchmark reveals distinct behavior in logic programming with respect to model size. As illustrated in Figure 2a, the smallest variant, `Qwen2.5-Coder-0.5B`, is initially not able to learn in zero-shot settings with a 0.00 accuracy, highlighting a fundamental lack of Prolog-specific structural priors. However, introducing a single demonstration (one-shot) serves as a *syntax anchor*, enabling the model to reach 0.13 accuracy after 1000 steps.

As model size scales to 3B and 7B, we observe the emergence of zero-shot generalization capabilities. Notably, while the 7B model achieves a high one-shot baseline of 0.81, its performance is worsening during training, settling at 0.67 after 1500 steps. Instead, the 3B model significantly improves its performance during training, from 0.35 to 0.78. The 7B "performance drift" suggests that without specific GRPO fine-tuning, RL may lead the model to overlook strict prolog syntax, a phenomenon further evidenced by the decline in the strict *pass*^4 metric over training steps.

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

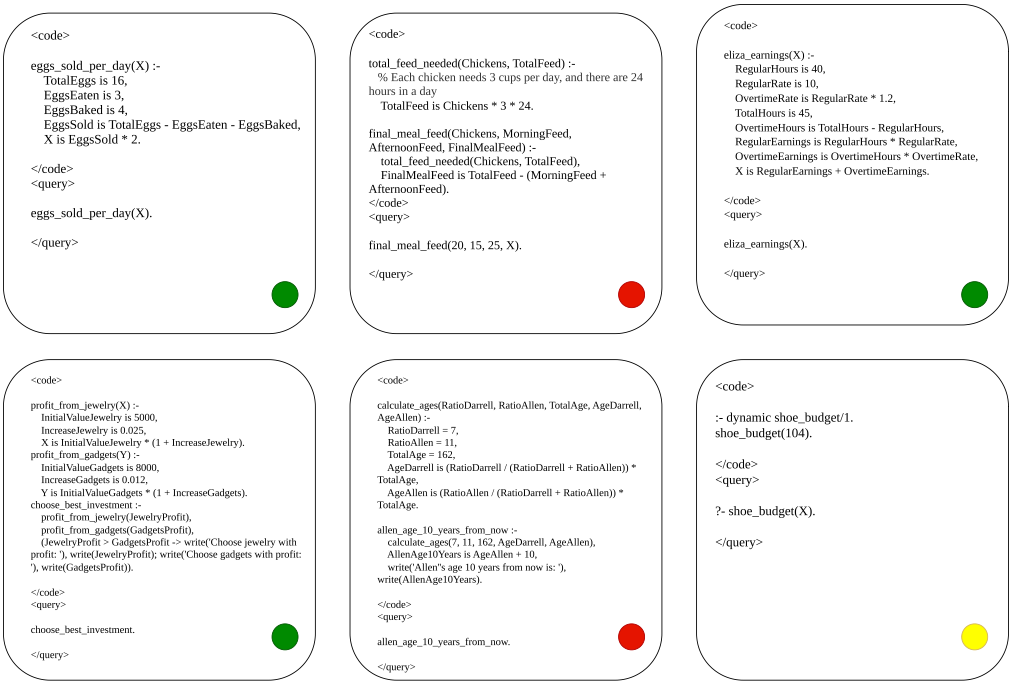


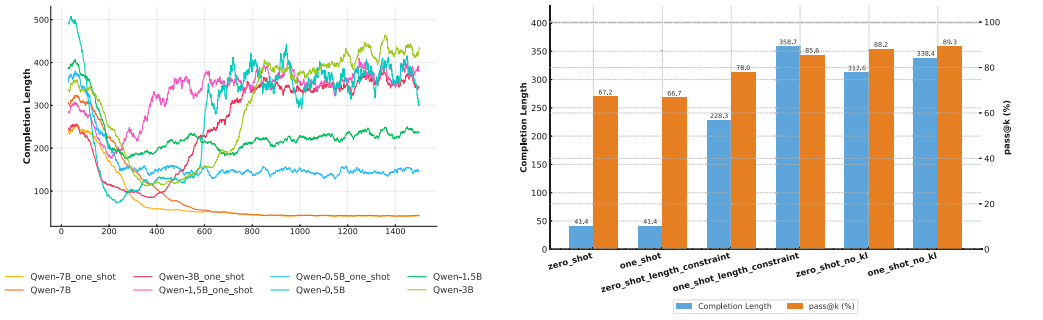
Fig. 3. Examples of Prolog code generated by the trained models on GSM8K problems. Green dots indicate solutions that executed correctly and returned the expected answer. Red dots denote solutions that failed due to logical or syntax errors. Yellow dots highlight generations that, while syntactically plausible, were just hardcoding the final value.

Figure 3 illustrates representative Prolog solutions produced across different training configurations. Correctly executing solutions (green) demonstrate the model's ability to translate arithmetic word problems into well-structured predicates with appropriate variable bindings and queries. Failed generations (red) typically stem from mishandled predicate arity or incorrect arithmetic decomposition. The yellow dot marks a generation that, while syntactically valid, hardcodes intermediate values directly into the predicate body rather than encoding general-purpose logic – a pattern frequently observed among smaller model variants.

4.2 Rosetta code

To evaluate if reasoning gains transfer to general programming, we tested the models on Rosetta Code (Figure 2b). The results confirm that GRPO adaptation is efficient for low-resource languages: the 1.5B model, which was non-functional at base, reached a *pass*@4 of 0.20 within just 500 steps; The 7B model peaked early at 500 steps (0.46 one-shot), nearly doubling its zero-shot baseline. The persistent gap between zero-shot and one-shot performance suggests that even after reasoning-based fine-tuning, example-based prompting remains essential for maintaining correct predicate arity and recursion patterns in traditionally underrepresented languages.

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press



(a) GSM8K (b) Rosetta Code

Fig. 4. (a) Completion length for six Qwen model variants (0.5B, 1.5B, 3B, 7B with and without one-shot prompting) over training steps; (b) Comparison of different strategies applied to `Qwen2.5-Coder-7B`: completion length (left y-axis) and pass@k (right y-axis). Models are grouped by level of supervision (zero-shot vs. one-shot) and whether length constraints or KL-regularization are applied.

5 Ablation study

5.1 Prolog code generation

As visualized in Figure 4a, there is a correlation between the expansion of completion length and the final code accuracy; specifically, the transition from short reasoning chains to longer length sequences is the primary driver of the observed performance leaps. During initial training, we observed a "reasoning contraction" in which the 7B model shortened its output, at the cost of expressivity.

To counteract this effect, we introduce a length-constrained reward on the reasoning segment. Given a completion (c), let ($r_{\text{length}}(c)$) be the token-length of the reasoning (scaled by (10^{-4})). We define the length constraint reward as:

$$r(c) = \begin{cases} 1 & \text{if } 0.009 < r_{\text{length}}(c) < 0.013; \\ 0 & \text{otherwise} \end{cases}$$

This formulation encourages reasoning traces within a target length range.

By implementing a length-aware reward mechanism and removing the KL-divergence penalty, we effectively expanded the model's planning process. As shown in Table 2, these modifications shifted the zero-shot *pass*@4 from 0.637 to a peak of 0.882 at 1500 steps. The impact of these modifications is summarized in Figure 4b, which illustrates the concurrent increase in both completion length and pass@k performance across zero-shot and one-shot settings.

According to the results in Table 2, our optimized `Qwen2.5-Coder-7B` configuration – utilizing length constraints and the removal of KL divergence – achieves a peak one-shot score of 0.893, remarkably outperforming `PROPER` (0.72), a Mistral 7B variant that underwent specialized supervised fine-tuning for Prolog code generation in GSM8K. Furthermore, our model significantly outperforms larger general-purpose architectures, such as `Llama 3.3 70B` (0.839) and `Qwen2.5-Coder-32B` (0.873), on the same benchmark. Baselines for `Llama 3.3 70B` and `Qwen2.5-Coder 32B` were evaluated using the same system prompts and XML-based output templates to ensure a fair comparison.

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

Table 2. *pass@4 scores for* `Qwen2.5-Coder` *trained with new reward function, reporting results for both standard and length-optimized reward functions*

Configuration	Model	Base	500	1000	1500
Yang *et al.* (2024)	`Mistral 7B`	**0.72**	–	–	–
Zero-shot	`Llama 3.3 70B`	**0.839**	–	–	–
Zero-shot	`Qwen2.5-Coder-32B`	**0.873**	–	–	–
Zero-shot	`Qwen2.5-Coder-7B`	0.637	**0.714**	0.670	0.667
	+ length constraint		0.764	**0.782**	0.780
	+ no KL divergence		0.847	0.880	**0.882**
One-shot	`Qwen2.5-Coder-7B`	**0.814**	0.724	0.686	0.672
	+ length constraint		0.852	**0.862**	0.856
	+ no KL divergence		0.879	0.883	**0.893**

Table 3. *pass@4 scores for GSM-symbolic p1 and p2 datasets*

Dataset	Configuration	Model	Base	500	1000	1500
p1	Zero-shot	`Qwen2.5-Coder-7B`	0.469	**0.535**	0.418	0.427
		+ length constraint		0.575	**0.638**	0.634
		+ no KL divergence		0.679	**0.763**	0.752
	One-shot	`Qwen2.5-Coder-7B`	**0.754**	0.578	0.479	0.470
		+ length constraint		0.747	0.782	**0.787**
		+ no KL divergence		0.770	0.792	**0.803**
p2	Zero-shot	`Qwen2.5-Coder-7B`	0.298	**0.336**	0.197	0.198
		+ length constraint		0.388	**0.451**	0.433
		+ no KL divergence		0.534	0.644	**0.673**
	One-shot	`Qwen2.5-Coder-7B`	**0.588**	0.398	0.205	0.201
		+ length constraint		0.645	0.665	**0.679**
		+ no KL divergence		0.673	**0.721**	0.717

5.2 GSM-symbolic

To ensure the model's reasoning capabilities are truly robust and not overfitted to the standard GSM8K templates, we evaluated our configurations on GSM-Symbolic (p1 and p2), as detailed in Table 3. A critical observation from this evaluation is the significant performance degradation caused by standard GRPO fine-tuning on reasoning-heavy tasks; specifically, in the one-shot setting on the p2 dataset, accuracy dropped from a baseline of 0.588 to 0.201 after 1500 steps.

Our length-aware reward mechanism effectively mitigated this decline, while removing the KL-divergence penalty further improved performance on both benchmarks. On the p1 dataset, the one-shot accuracy rose from 0.754 to 0.803. More notably, on the more demanding p2 dataset – which requires deeper symbolic manipulation – the zero-shot score more than doubled, increasing from 0.298 to 0.673. These results demonstrate that our methodology enhances genuine symbolic reasoning and cross-task robustness, ensuring that the model can handle structural variations in logic problems rather than relying on simple memorization of patterns.

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

Table 4. *pass@4 scores for GSM-8K dataset for lisp*

Configuration	Model	Base	500	1000	1500
Zero-shot	`Qwen2.5-Coder-7B`	0.5716	0.830	0.8653	**0.8702**
	+ length constraint		0.837	**0.8712**	0.8702
	+ no KL divergence		0.840	**0.8757**	0.8711
One-shot	`Qwen2.5-Coder-7B`	0.8112	0.851	**0.868**	0.863
	+ length constraint		0.851	**0.871**	0.864
	+ no KL divergence		0.861	**0.873**	0.868

5.3 Lisp code generation

To validate the versatility of our approach, we also evaluated Lisp (Table 4). The base `Qwen2.5-Coder-7B` model exhibited higher initial proficiency in Lisp (0.5716) than in Prolog, likely due to a stronger representation in its pretraining data. Our adaptation significantly improved performance, reaching a peak zero-shot score of 0.8757.

However, the impact of removing the KL-divergence was notably more modest in Lisp (0.8712 to 0.8757) than the transformative gains observed in Prolog. This suggests that "un-regularized" training is most critical for highly specialized, low-resource languages such as Prolog, where the model must deviate substantially from its base policy to achieve formal correctness. For Lisp, once reasoning length is optimized, policy regularization is less of a limiting factor.

6 Conclusion

This work validates GRPO as an effective method for improving Prolog code generation in `Qwen2.5-Coder`. We observe clear gains across sizes, here reported as pass@4 scores and absolute deltas: the 0.5B model improves by + 0.118 in one-shot, the 1.5B model reaches 0.312 zero-shot, the 3B model gains + 0.436 zero-shot, and the 7B model without KL reaches 0.882 (zero-shot) and 0.893 (one-shot), establishing a new state of the art on Prolog GSM8K under our evaluation protocol. Improvements generalize to broader programming tasks (Rosetta Code: 7B one-shot + 0.200) and transfer to Lisp (zero-shot + 0.299, from 0.5716 to 0.8702; peaking at 0.8757 without KL; one-shot 0.873 without KL). KL removal helps most in Prolog, and length-aware rewards provide consistent additional gains. These results indicate a robust, adaptable path for enhancing code generation in underrepresented languages that often lack extensive training datasets and could benefit significantly from our methodology.

Future work could investigate hybrid neural-symbolic architectures where LLMs delegate complex logical inference or constraint satisfaction to a Prolog or Lisp engine, moving beyond simple code generation. Extending PoT approaches, LLMs could learn to generate intermediate Prolog queries to structure their reasoning. Furthermore, Prolog's capabilities can be leveraged for more sophisticated verification, checking the logical consistency of LLM reasoning steps or formal properties of generated code, potentially using execution traces for enhanced explainability. Moreover, given the success demonstrated with Prolog and Lisp, we plan to extend our study to other underrepresented languages.

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

Finally, we are considering the construction of a new training dataset specifically designed to better capture non-numerical reasoning patterns, potentially addressing limitations of GSM8K in this regard. Such a dataset could enable more robust evaluation and training of models on symbolic and logical reasoning tasks beyond arithmetic problem solving.

Data availability

The source code for preprocessing and evaluation is at: https://github.com/biancaraimondi/LLM_Format

References

Chen, M., et al. 2021. Evaluating large language models trained on code. arXiv preprint arXiv: 2107.03374.

Chen, W., Ma, X., Wang, X. and Cohen, W. W. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research* 2023.

Cobbe, K., et al. 2021. Training verifiers to solve math word problems. arXiv preprint arXiv: 2110.14168.

Freuder, E. C. 1997. In pursuit of the Holy Grail. *Constraints* 2, 1, 57–61.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L. and Chen, W. 2022. LoRA: Low-rank adaptation of large language models. In *ICLR 2022*.

Hui, B., et al. 2024. Qwen2.5-coder technical report. arXiv preprint arXiv: 2409.12186.

Jiang, A. Q., et al. 2023. Mistral 7B. arXiv preprint arXiv: 2310.06825.

Gehring, J., Zheng, K., Copet, J., Mella, V., Cohen, T. and Synnaeve, G. 2025. RLEF: Grounding code LLMs in execution feedback with reinforcement Learning. In *ICML 2025*. PMLR (Proceedings of Machine Learning Research), Vol. 267.

Kwon, W., et al. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Le, H., et al. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems* 35, 21314–21328.

Li, R., et al. 2023. StarCoder: May the source be with you! *Transactions on Machine Learning Research* 2023.

Llama Team. 2024. The Llama 3 Herd of models. arXiv preprint arXiv: 2407.21783.

Mirzadeh, I., et al. 2025. GSM-Symbolic: Understanding the limitations of mathematical reasoning in large language models. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*.

Moto-Oka, T. and Stone, H. S. 1984. Fifth-generation computer systems: A Japanese project. *Computer* 17, 3, 6–13.

Norvig, P. n.d. An ((Even Better) Lisp) Interpreter (in Python). URL: https://norvig.com/lispy2.html [Accessed on October 14, 2025].

Rosetta Code contributors. n.d. Rosetta Code. URL: https://rosettacode.org/wiki/Rosetta_Code.

Rozière, B., et al. 2023. Code Llama: open foundation models for code. arXiv preprint arXiv: 2308.12950.

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press

Shao, Z., et al. 2024. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models, arXiv preprint arXiv: 2402.03300.

Vaswani, A., et al. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, 30. Curran Associates, Inc.

von Werra, L., et al. 2020. TRL: Transformer reinforcement learning.. GitHub repository. URL: https://github.com/huggingface/trl.

Wen, X., et al. 2025. Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base LLMs. CoRR abs/2506.14245.

Yüce, T. and PySwip contributors. PySwip v0.3.0. https://pyswip.org.

Yang, X., Chen, B. and Tam, Y.-C. 2024. Arithmetic reasoning with LLM: Prolog generation & permutation. In *Findings of the Association for Computational Linguistics: NAACL 2024*, 699–710. Association for Computational Linguistics.

https://doi.org/10.1017/S1471068426100489 Published online by Cambridge University Press