The Interpreter as Teacher: GRPO with Execution Feedback for Prolog Code Generation*

Federico Pennino[1,*], Bianca Raimondi[1], Massimo Rondelli[1], Andrea Gurioli[1] and Maurizio Gabbrielli[1]

[1]*Università di Bologna, Italy*

Abstract

Large Language Models (LLMs) generate fluent code in mainstream languages but fail systematically when asked to produce Prolog: they hallucinate built-ins, impose imperative control flow on a declarative paradigm, and mishandle the interplay of unification and backtracking. The cause is distributional – logic-programming code is scarce in modern pre-training corpora – and no amount of scale will fix data that does not exist. We show that this gap can be closed by training small open-weight models with reinforcement learning where the reward signal comes from a SWI-Prolog interpreter embedded in the training loop. Using Group Relative Policy Optimization (GRPO) on `Qwen2.5-Coder-Instruct` (0.5B–7B), we identify a failure mode – *reasoning contraction* – in which the KL penalty anchors the policy too tightly to the supervised baseline; the model collapses to short memorised patterns and accuracy degrades over training. Removing the KL term and adding a length-aware reward expands reasoning chains and lifts one-shot pass@4 on GSM8K-Prolog from 0.814 to 0.893, outperforming both PROPER (0.72) and substantially larger general-purpose models such as Llama 3.3 70B (0.839) and Qwen2.5-Coder-32B (0.873). Gains transfer to 20 Rosetta Code Prolog tasks and to GSM-Symbolic (zero-shot p2 more than doubles, from 0.298 to 0.673), and the same recipe applied to Lisp lifts zero-shot pass@4 from 0.572 to 0.876. Taken together, the results indicate that, for formal languages with a deterministic semantics, the interpreter itself is sufficient to act as the primary training supervisor.

Keywords

Logic programming, Prolog code generation, Large Language Models, Reinforcement Learning with Verifiable Rewards, GRPO, Neuro-symbolic AI

1. Introduction

The resurgence of connectionist methods over the past decade has produced Large Language Models of remarkable capability, yet Artificial Intelligence has long drawn on a second tradition: symbolic reasoning, grounded in logic, formal verification, and languages such as Prolog. Bridging the two paradigms remains a central open problem.

From a programming-languages perspective, this bridge is also a step toward the "Holy Grail" articulated by [1]: *the user states the problem and the computer solves it.* A promising path is deceptively simple: teach neural models to *write* the symbolic programs themselves. If an LLM can reliably generate executable Prolog 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 – properties that purely neural generation cannot guarantee.

In practice, this vision runs into an immediate obstacle. LLMs have achieved remarkable proficiency at generating code in mainstream languages such as Python and JavaScript, yet this proficiency does not transfer to logic programming. When models such as `OpenAI Codex`, `Code-Llama` ([2]), `StarCoder` ([3]), and `Qwen2.5-Coder` ([4]) are asked to produce Prolog programs, they consistently fail: hallucinated built-in predicates, imperative control flow superimposed on a declarative paradigm,

---

*An extended version of this work has been accepted at ICLP 2026. This paper revisits the same methodology and results for the CILC audience, with a sharpened discussion of the logic-programming aspects.

*Corresponding author.

federico.p98@gmail.com (F. Pennino); bianca.raimondi@unibo.it (B. Raimondi); massimo.rondelli@unibo.it (M. Rondelli); andrea.gurioli2@unibo.it (A. Gurioli); maurizio.gabbrielli@unibo.it (M. Gabbrielli)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

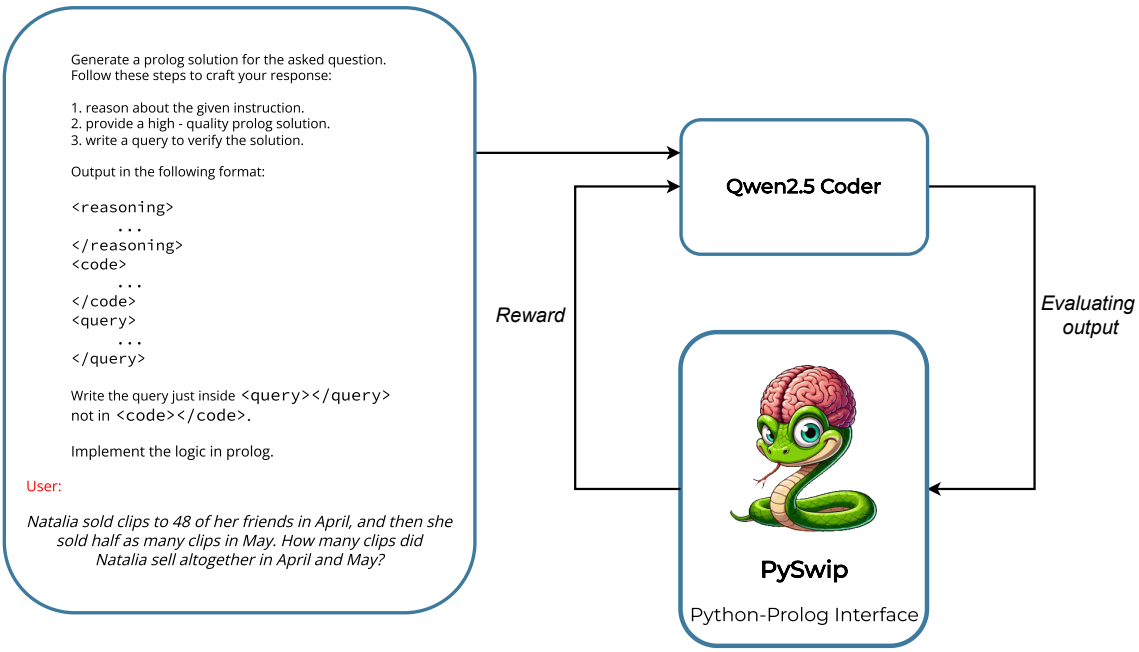


Figure 1: Schematic of the GRPO training pipeline. `Qwen2.5-Coder` generates Prolog solutions, which are evaluated via `PySwip` to produce execution-based reward signals.

and broken handling of unification and backtracking. The root cause is not architectural but distributional: these languages are scarce in modern pre-training corpora.

This paper shows that the obstacle can be addressed by training small, open-weight models with reinforcement learning, using a Prolog interpreter as the source of the supervisory signal rather than scarce human-written examples ([5]).

More precisely, rather than fine-tuning the model to imitate scarce examples, we shift the learning objective to *maximising execution correctness* as judged by a SWI-Prolog interpreter embedded in the training loop (Fig. 1). This is an instance of Reinforcement Learning with Verifiable Rewards (RLVR) as discussed by [6]: the symbolic 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) [7], a policy-gradient algorithm that removes the separate value-function network required by Proximal Policy Optimization (PPO). GRPO estimates each output's advantage relative to a group of peer outputs sampled from the same prompt, reducing computational overhead and stabilising learning in the sparse-reward regime characteristic of code generation.

We apply GRPO to the `Qwen2.5-Coder-Instruct` family (0.5B–7B) [4] and train on the GSM8K dataset [8], whose emphasis on structured logical problem-solving makes it a useful proxy for reasoning skills.

We identify a failure mode we term *reasoning contraction*: under standard GRPO with a Kullback–Leibler (KL) penalty, the 7B model collapses to short, memorised patterns, and one-shot accuracy on GSM8K degrades from 0.814 to 0.672. We attribute this to the KL penalty anchoring the policy too tightly to the supervised baseline, impeding the distributional shift required to internalise a fundamentally different programming paradigm. Removing the KL constraint and adding a length-aware reward resolves the contraction: completions expand, reasoning chains deepen, and one-shot pass@4 on GSM8K rises to 0.893, outperforming PROPER (0.72) [9], a Mistral-7B variant with specialised supervised fine-tuning, and substantially larger models including `Llama 3.3 70B` (0.839) and `Qwen2.5-Coder-32B` (0.873). This paper makes the following 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 model.
2. It provides a systematic ablation of GRPO training dynamics for logic programming, identifies *reasoning contraction* as a failure mode of KL-regularised RL, and characterises when relaxing the constraint is beneficial.
3. It shows that the gains generalise: improvements transfer to 20 diverse Prolog tasks from Rosetta Code and to GSM-Symbolic, where zero-shot p2 accuracy more than doubles ($0.298 \rightarrow 0.673$).
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.

For formal languages with deterministic semantics, the interpreter itself can serve as the primary training signal, replacing the scarce human data on which current approaches depend.

2. Related Work

Program synthesis – generating executable code from high-level specifications – has historically been the domain of symbolic AI. Inductive Logic Programming (ILP), prominent in the 1990s, used formal logic to infer hypotheses from positive and negative examples; systems such as Progol and Aleph can learn recursive logic programs given a strict background knowledge base. ILP, however, struggled with noise, required extensive domain engineering, and could not handle natural-language ambiguity.

Neural Program Synthesis replaced this discrete search with continuous optimisation. Early sequence-to-sequence models treated code generation as translation, but struggled with long-range dependencies and AST hierarchy. The Transformer [10] and the pre-training paradigm fundamentally changed the landscape, leading to modern decoder-only Code LLMs trained on vast corpora of source code. Performance, however, is highly uneven: while these models excel at Python or Java, their ability to generate Prolog or Lisp is often superficial, mimicking syntax (indentation, parentheses) without respecting semantics (unification, recursion, macro expansion). This is a direct consequence of the distribution of training data: Prolog code is a vanishingly small fraction of public repositories.

Chain-of-Thought (CoT) prompting encourages LLMs to emit intermediate reasoning steps before the final answer. While effective on math benchmarks, CoT suffers from the "validity hallucination" problem: a plausible-sounding chain may contain logical errors. Program-of-Thought (PoT) [11] addresses this by decoupling reasoning from computation: the LLM generates a program (typically Python) which is executed by an external interpreter, offloading calculation to a deterministic engine.

Recent work identifies Prolog as a superior PoT target for logical reasoning. GSM8K-Prolog [9] showed 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 (*how* to compute), Prolog is declarative (*what* is true), aligning more closely with the semantic parsing of natural-language logic. [9] also introduced ProPro (Predicate Permutation), a data augmentation method exploiting the commutativity of pure-Prolog conjunctions ($A, B \equiv B, A$).

Supervised Fine-Tuning teaches syntax but does not directly optimise for correctness. Reinforcement Learning permits optimising non-differentiable metrics such as "does this code compile?" or "does it pass the tests?". CodeRL [12] used unit-test results as the reward signal in an actor-critic setup. RLEF (Reinforcement Learning with Execution Feedback) [13] demonstrated that models can learn to debug their own code from execution traces. Building on these, we target a setting in which both training data *and* test suites are scarce: by embedding a Prolog interpreter directly into the reward loop, we obtain a dense, ground-truth correctness signal without hand-written tests, turning the interpreter from a development tool into a training supervisor.

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`	$\{-1, -0.5, 1\}$

3. Methodology

3.1. Models and datasets

We focus on small-scale LLMs for Prolog code generation, using the 0.5B, 1.5B, 3B, and 7B versions of `Qwen2.5-Coder-Instruct`, chosen for their effectiveness and modest resource requirements.

A core challenge is the limited availability of public Prolog code, which restricts the models' initial exposure to Prolog syntax. To address this, we adopt an RL strategy on the GSM8K dataset which, although designed for arithmetic reasoning, emphasises structured, logical problem-solving and is therefore a useful proxy for the reasoning skills required to write Prolog. Training uses only the official GSM8K training split, and all reported metrics are computed on the official test split, preventing leakage.

Figure 1 sketches the execution pipeline: the LLM learns to map natural-language prompts to executable Prolog programs via feedback from execution.

To assess generalisation, we built a 20-task Prolog benchmark sourced from [14]: 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, Ackermann, Balanced brackets, Knight's tour, Merge sort, Roman numerals decoding, Longest common subsequence, Huffman coding, and the 24 game. These tasks were selected to cover core logic-programming constructs: recursion, list manipulation, and symbolic backtracking.

We also evaluate on GSM-Symbolic [15], a benchmark for symbolic reasoning over abstract mathematical expressions, designed solely for testing the generalisation of models trained on GSM8K. We use its variants GSM-Symbolic-Plus-1 (P1) and Plus-2 (P2), which add one or two additional reasoning clauses per question. By systematically modifying numerical values and contextual details while preserving logical structure, GSM-Symbolic reveals whether the model has learned to extract the underlying reasoning patterns or has merely overfitted to GSM8K templates.

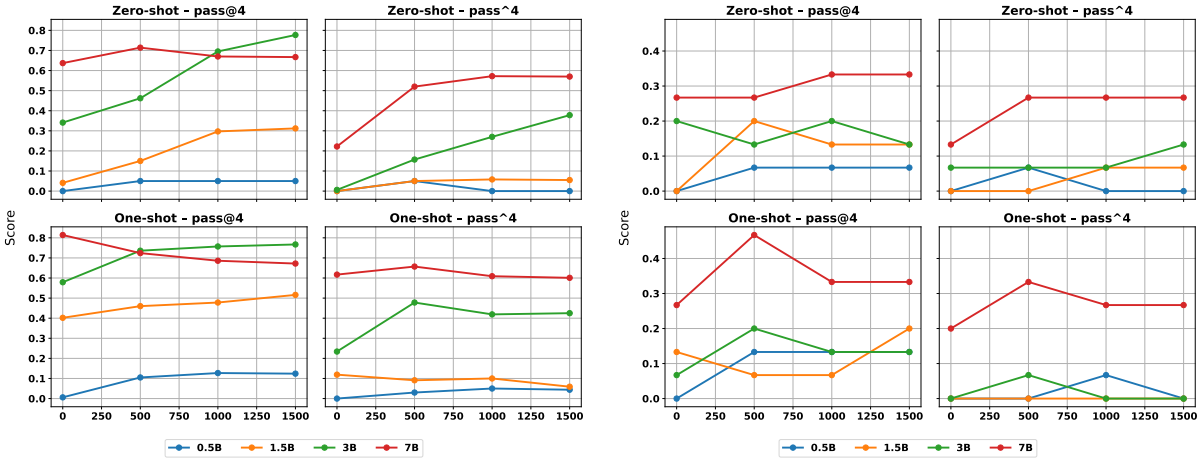
3.2. Experimental setup

We use TRL [16] for training, together with Low-Rank Adaptation (LoRA) [17] of rank 32 to enable efficient fine-tuning across all model sizes. Evaluation varies along three axes: model size (0.5B, 1.5B, 3B, 7B), training checkpoint ($\{500, 1000, 1500\}$), and prompting regime (zero-shot vs. one-shot). Models are prompted consistently with the training setting. Inference is performed with vLLM [18] on a multi-GPU server with 8×NVIDIA A100-SXM4 GPUs (80 GB VRAM, CUDA 12.2); training and inference run on a single GPU unless otherwise noted.

3.3. Reward system

Training uses GRPO with a multi-objective reward enforcing two constraints: adherence to the system-prompted template (Fig. 1) and functional correctness of the generated Prolog. Total rewards are the sum of the components in Table 1.

Format reward. To make the generated Prolog extractable and interpretable, we enforce a structured output using `<reasoning>`, `<code>`, and `<query>` tags. This serves as an early-stage signal:



(a) GSM8K

(b) Rosetta Code

Figure 2: Pass@4 and pass^4 for `Qwen2.5-Coder` models in zero-shot and one-shot settings.

- *Tag presence* (`xmlcount`): $+0.125$ per required tag, with a -0.5 penalty if the query is incorrectly nested inside the `<code>` block (which would break execution).
- *Strict evaluation*: binary $+0.5$ if the output matches the template exactly.
- *Soft evaluation*: binary $+0.5$ if the code is extractable by regex even with minor formatting deviations outside the tags.

This formatting strategy improves sample efficiency by penalising structural deviations, allowing the model to converge quickly on a consistent, executable output format.

Correctness reward. We evaluate semantic correctness by executing the generated code via PySwip, a Python interface to SWI-Prolog. The environment is sandboxed with a 5 s timeout to prevent infinite loops:

- 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 does not return a result within 5 s.

Penalising syntax errors more heavily than timeouts prioritises generating "runnable" code in the early phases of training.

3.4. Evaluation metrics

For each problem the model generates $k = 4$ candidate solutions; we report two execution-based metrics. **Pass@**k counts a problem as solved if *at least one* of k candidates produces the correct output:

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

where N is the number of problems, c_i^j the j-th candidate for problem i, $P(\cdot)$ its execution result, and r_i the ground truth. **Pass**$^\wedge k$ is the stricter variant requiring *all* k candidates to be correct:

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

and measures the model's reliability across multiple generations.

```
<code>

eggs_sold_per_day(X) :-
    TotalEggs is 16,
    EggsEaten is 3,
    EggsBaked is 4,
    EggsSold is TotalEggs - EggsEaten - EggsBaked,
    X is EggsSold * 2.

</code>
<query>

eggs_sold_per_day(X).

</query>
```

```
<code>

total_feed_needed(Chickens, TotalFeed) :-
  % Each chicken needs 3 cups per day, and there are 24
hours in a day
    TotalFeed is Chickens * 3 * 24.

final_meal_feed(Chickens, MorningFeed,
AfternoonFeed, FinalMealFeed) :-
    total_feed_needed(Chickens, TotalFeed),
    FinalMealFeed is TotalFeed - (MorningFeed +
AfternoonFeed).
</code>
<query>

final_meal_feed(20, 15, 25, X).

</query>
```

```
<code>

eliza_earnings(X) :-
    RegularHours is 40,
    RegularRate is 10,
    OvertimeRate is RegularRate * 1.2,
    TotalHours is 45,
    OvertimeHours is TotalHours - RegularHours,
    RegularEarnings is RegularHours * RegularRate,
    OvertimeEarnings is OvertimeHours * OvertimeRate,
    X is RegularEarnings + OvertimeEarnings.

</code>
<query>

eliza_earnings(X).

</query>
```

```
<code>

profit_from_jewelry(X) :-
    InitialValueJewelry is 5000,
    IncreaseJewelry is 0.025,
    X is InitialValueJewelry * (1 + IncreaseJewelry).
profit_from_gadgets(Y) :-
    InitialValueGadgets is 8000,
    IncreaseGadgets is 0.012,
    Y is InitialValueGadgets * (1 + IncreaseGadgets).
choose_best_investment :-
    profit_from_jewelry(JewelryProfit),
    profit_from_gadgets(GadgetsProfit),
    (JewelryProfit > GadgetsProfit -> write('Choose jewelry with
profit: '), write(JewelryProfit); write('Choose gadgets with profit:
'), write(GadgetsProfit)).

</code>
<query>

choose_best_investment.

</query>
```

```
<code>

calculate_ages(RatioDarrell, RatioAllen, TotalAge, AgeDarrell,
AgeAllen) :-
    RatioDarrell = 7,
    RatioAllen = 11,
    TotalAge = 162,
    AgeDarrell is (RatioDarrell / (RatioDarrell + RatioAllen)) *
TotalAge,
    AgeAllen is (RatioAllen / (RatioDarrell + RatioAllen)) *
TotalAge.

allen_age_10_years_from_now :-
    calculate_ages(7, 11, 162, AgeDarrell, AgeAllen),
    AllenAge10Years is AgeAllen + 10,
    write('Allen"s age 10 years from now is: '),
write(AllenAge10Years).

</code>
<query>

allen_age_10_years_from_now.

</query>
```

```
<code>

:- dynamic shoe_budget/1.
shoe_budget(104).

</code>
<query>

?- shoe_budget(X).

</query>
```

Figure 3: Examples of Prolog code generated by trained models on GSM8K. Green: correct execution. Red: failures due to logical or syntactic errors. Yellow: syntactically plausible solutions that hardcode the final answer.

4. Results and Discussion

4.1. GSM8K

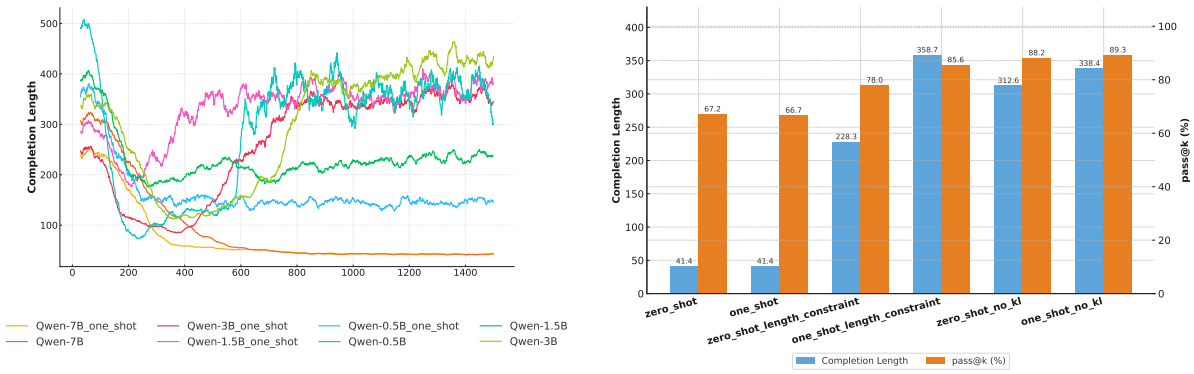
The performance of `Qwen2.5-Coder` on GSM8K reveals distinct behaviour across model sizes. As shown in Fig. 2a, the smallest variant (`Qwen2.5-Coder-0.5B`) is unable to learn in zero-shot (0.00 accuracy), highlighting a fundamental lack of Prolog-specific structural priors; a single demonstration (one-shot) acts as a *syntax anchor*, enabling 0.13 accuracy after 1000 steps.

As size scales to 3B and 7B, zero-shot generalisation emerges. Notably, while the 7B model achieves a strong one-shot baseline of 0.81, its performance *worsens* during training, settling at 0.67 after 1500 steps. The 3B model improves substantially during training, from 0.35 to 0.78. The 7B drift suggests that without targeted GRPO tuning, RL leads the model to disregard strict Prolog syntax – a phenomenon further evidenced by the decline of the strict pass^4 metric over training steps.

Figure 3 shows representative Prolog solutions across configurations. Correctly executing solutions (green) demonstrate the 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 rather than encoding general-purpose logic – a pattern frequently observed among smaller variants.

4.2. Rosetta Code

To test whether reasoning gains transfer to general programming, we evaluated the models on Rosetta Code (Fig. 2b). GRPO adaptation is efficient on low-resource languages: the 1.5B model, non-functional at the base, reaches pass@4 of 0.20 within 500 steps. The 7B model peaks early at 500 steps (0.46 one-shot), nearly doubling its zero-shot baseline. The persistent gap between zero-shot and one-shot regimes



(a) Completion length over training steps

(b) Strategy comparison on `Qwen2.5-Coder-7B`

Figure 4: (a) Completion length for six Qwen variants (0.5B, 1.5B, 3B, 7B with and without one-shot prompting) over training steps. (b) Strategy comparison on `Qwen2.5-Coder-7B`: completion length (left axis) and pass@k (right axis), grouped by supervision regime (zero/one-shot) and whether length constraints or KL regularisation are applied.

indicates that, even after fine-tuning, example-based prompting remains essential for maintaining correct predicate arity and recursion patterns in underrepresented languages.

5. Ablation Study

5.1. Prolog code generation

As shown in Fig. 4a, completion length and final code accuracy are correlated: the transition from short reasoning chains to longer sequences is the primary driver of the observed performance leaps. During initial training we observe *reasoning contraction*, in which the 7B model shortens its outputs at the cost of expressivity.

To counteract this, we introduce a length-constrained reward on the reasoning segment. Let $r_{\text{length}}(c)$ denote the token-length of the reasoning of completion c, scaled by 10^{-4}. The length reward is

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

This encourages reasoning traces in a target length range (roughly 90–130 reasoning tokens at our scale), without rewarding arbitrary verbosity.

Table 2
Pass@4 for `Qwen2.5-Coder` on GSM8K-Prolog with and without length-aware reward and KL removal.

Configuration	Model	Base	500	1000	1500
[9]	`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 on GSM-Symbolic P1 and P2.

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

Combined with the removal of the KL-divergence penalty, this length-aware reward effectively expands the model's planning process. Table 2 shows that these modifications shift the zero-shot pass@4 from 0.637 to a peak of **0.882** at 1500 steps. Figure 4b summarises the joint increase in completion length and pass@k across zero- and one-shot settings.

The optimised `Qwen2.5-Coder-7B` configuration – length constraint plus KL removal – reaches a peak one-shot score of 0.893, outperforming PROPER (0.72), a Mistral-7B variant with specialised supervised fine-tuning for Prolog GSM8K. It also surpasses much larger general-purpose architectures, `Llama 3.3 70B` (0.839) and `Qwen2.5-Coder-32B` (0.873). Baselines for these larger models were evaluated under the same system prompts and XML-based output templates to ensure a fair comparison.

5.2. GSM-Symbolic

To check whether the model's reasoning capabilities are robust and not overfitted to GSM8K templates, we evaluated our configurations on GSM-Symbolic P1/P2 (Table 3). A critical observation is the substantial degradation caused by standard GRPO fine-tuning on reasoning-heavy tasks: in one-shot on P2, accuracy drops from 0.588 to 0.201 after 1500 steps.

The length-aware reward mitigates the decline; removing the KL penalty further improves performance on both benchmarks. On P1, one-shot accuracy rises from 0.754 to 0.803. More notably, on the harder P2 – which requires deeper symbolic manipulation – zero-shot accuracy more than doubles from 0.298 to **0.673**. These results indicate that our methodology enhances genuine symbolic reasoning and cross-task robustness, rather than simple memorisation of templates.

Table 4
Pass@4 on GSM8K 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 exhibits higher initial proficiency in Lisp (0.5716) than in Prolog, plausibly reflecting a stronger representation in its pre-training corpus. Our adaptation yields a peak zero-shot score of **0.8757**.

The impact of removing the KL penalty, however, is much more modest in Lisp ($0.8712 \rightarrow 0.8757$) than the transformative gains observed in Prolog. This suggests that "un-regularised" training is most critical for highly specialised, 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 optimised, policy regularisation is a less limiting factor.

6. Conclusions

This work validates GRPO as an effective method for improving Prolog code generation in `Qwen2.5-Coder`. We observe consistent gains across sizes, 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 generalise 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 lack extensive training datasets.

Future work could investigate hybrid neuro-symbolic architectures in which the LLM delegates complex logical inference or constraint satisfaction to a Prolog or Lisp engine, moving beyond simple code generation. Extending program-of-thought approaches, the LLM could learn to generate intermediate Prolog queries to structure its reasoning. Prolog's capabilities could also support more sophisticated verification, checking the logical consistency of LLM reasoning steps or formal properties of the generated code, potentially using execution traces for enhanced explainability. Given the success demonstrated on Prolog and Lisp, we plan to extend this study to other underrepresented languages. Finally, we are considering the construction of a new training dataset specifically designed to capture non-numerical reasoning patterns, addressing the limitations of GSM8K and enabling more robust evaluation and training of models on symbolic and logical reasoning tasks beyond arithmetic.

Acknowledgments

An earlier, extended version of this work has been accepted for publication at ICLP 2026. We thank the ICLP and CILC reviewers for their constructive feedback.

Declaration on Generative AI

The author(s) used generative AI tools (LLM-based assistants) for grammar and spelling check during the preparation of this manuscript. After using these tools the author(s) reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] E. C. Freuder, In pursuit of the holy grail, Constraints 2 (1997) 57–61.
[2] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton Ferrer, A. Grattafiori, W. Xiong,

A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, G. Synnaeve, Code Llama: Open Foundation Models for Code, arXiv e-prints (2023) arXiv:2308.12950. doi:10.48550/arXiv.2308.12950. arXiv:2308.12950.

[3] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. M. V, J. T. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, H. de Vries, Starcoder: may the source be with you!, Trans. Mach. Learn. Res. 2023 (2023). URL: https://openreview.net/forum?id=KoFOg41haE.

[4] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, et al., Qwen2.5-coder technical report, arXiv preprint arXiv:2409.12186 (2024).

[5] Y. Li, et al., Competition-level code generation with AlphaCode, Science 378 (2022) 1092–1097.

[6] e. a. Wang, Reinforcement learning with verifiable rewards: A survey, arXiv preprint arXiv:2506.14245 (2025).

[7] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu, D. Guo, DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models, arXiv e-prints (2024) arXiv:2402.03300. doi:10.48550/arXiv.2402.03300. arXiv:2402.03300.

[8] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, J. Schulman, Training verifiers to solve math word problems, CoRR abs/2110.14168 (2021). URL: https://arxiv.org/abs/2110.14168. arXiv:2110.14168.

[9] X. Yang, B. Chen, Y.-C. Tam, Arithmetic reasoning with LLM: Prolog generation & permutation, in: K. Duh, H. Gomez, S. Bethard (Eds.), Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers), Association for Computational Linguistics, Mexico City, Mexico, 2024, pp. 699–710. URL: https://aclanthology.org/2024.naacl-short.61/. doi:10.18653/v1/2024.naacl-short.61.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in Neural Information Processing Systems, 2017.

[11] W. Chen, X. Ma, X. Wang, W. W. Cohen, Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, Transactions on Machine Learning Research (TMLR) (2023).

[12] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, S. C. H. Hoi, CodeRL: Mastering code generation through pretrained models and deep reinforcement learning, Advances in Neural Information Processing Systems (2022).

[13] J. Gehring, et al., RLEF: Grounding code LLMs in execution feedback with reinforcement learning, in: International Conference on Machine Learning (ICML), 2025.

[14] Rosetta Code contributors, Rosetta code, https://rosettacode.org/wiki/Rosetta_Code, 2007. URL: https://rosettacode.org/wiki/Rosetta_Code.

[15] I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, M. Farajtabar, Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models, in: The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025, OpenReview.net, 2025. URL: https://openreview.net/forum?id=AjXkRZIvjB.

[16] L. von Werra, Y. Belkada, L. Tunstall, E. Beeching, T. Thrush, N. Lambert, S. Huang, K. Rasul, Q. Gallouédec, Trl: Transformer reinforcement learning, https://github.com/huggingface/trl, 2020.

[17] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, LoRA: Low-Rank Adaptation of Large Language Models, arXiv e-prints (2021) arXiv:2106.09685. doi:10.48550/arXiv.2106.09685. arXiv:2106.09685.

[18] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, I. Stoica, Efficient memory management for large language model serving with pagedattention, in: Proceedings of

the ACM SIGOPS 29th Symposium on Operating Systems Principles, 2023.