

Training Language Models to Use Prolog as a Tool

Niklas Mellgren

University of Southern Denmark
mell@sdu.dk

Peter Schneider-Kamp

University of Southern Denmark
petersk@imada.sdu.dk

Lukas Galke Poech

University of Southern Denmark
galke@imada.sdu.dk

Abstract

Language models frequently produce plausible yet incorrect reasoning traces that are difficult to verify. We investigate fine-tuning models to use Prolog as an external symbolic reasoning tool, training Qwen2.5-3B-Instruct with Group Relative Policy Optimization (GRPO) on a cleaned version of GSM8K (which we release as gsm8k-prolog-prover). We systematically vary prompt structure, reward composition (execution, syntax, semantics, structure), and inference protocol (single-try, multiple-try, and two agentic modes). Our reinforcement learning approach outperforms supervised fine-tuning on GSM8K, and the resulting 3B model achieves zero-shot performance on MMLU-STEM and MMLU-Pro competitive with 7B few-shot baselines. Most importantly, we identify an accuracy–auditability trade-off: configurations tuned for correctness alone learn to delegate reasoning to natural language and use Prolog only for the final computation, while configurations rewarded for symbolic structure produce fully auditable programs at a cost in accuracy. We interpret this trade-off as a form of reward hacking and discuss its implications for deploying neurosymbolic systems in safety-critical domains.

The source code for our experiments is available under <https://github.com/aisilab/Prolog-as-a-Tool>

1 Introduction

Reasoning is the key driver of recent advances in large language models (Guo et al., 2025; Muenighoff et al., 2025; OpenAI, 2024). With its origins in chain-of-thought prompting (Wei et al., 2022) to better handle logic puzzles and mathematical tasks, reasoning also powers LLM agents, where reasoning steps are interleaved with tool use (Yao et al., 2023). Reasoning traces, i.e., the text generated when reasoning is elicited, were initially considered to yield an explanation for the

final result as a side effect. However, more and more evidence suggests that reasoning traces are not always faithful (Lanham et al., 2023; Paul et al., 2024), i.e., the reasoning traces do not necessarily correspond to how the model actually arrives at its final output. This lack of faithfulness renders reasoning traces invalid as a robust explanation for the final response or action.

Being able to trace and audit the reasoning of an LLM is crucial in order to validate and justify the final decision (cf., EU AI Act). Textual reasoning traces that are potentially unfaithful do not serve this purpose (Korbak et al., 2025). Thus, we here investigate an alternative way of reasoning, where large language models interact with a symbolic reasoning engine. Symbolic reasoning engines, such as implemented in the declarative Prolog programming language, offer precise and auditable reasoning traces, i.e., composed of explicit facts and rules.

Recent studies have shown that prompting LLMs to emit Prolog code, and then executing it in a symbolic runtime, can dramatically improve accuracy over text-only reasoning (Borazjanizadeh and Piantadosi, 2024; Tan et al., 2024; Yang et al., 2024b). These methods currently rely on in-context learning prompting or supervised fine-tuning. However, only the most powerful commercial models are capable of the level of in-context learning needed for using a niche programming language that likely has not much prevalence in the pre-training data.

Motivated by recent findings that reinforcement learning may lead to better generalization than supervised fine-tuning (Chu et al., 2025), we investigate if we can teach an LLM to use Prolog as a tool through reinforcement learning with verifiable rewards (RLVR). Specifically, we employ group relative policy optimization (GRPO) (Shao et al., 2024) which has led to promising results in DeepSeek-R1 and its R0 variant, which exclusively uses GRPO as post-training (Guo et al., 2025).

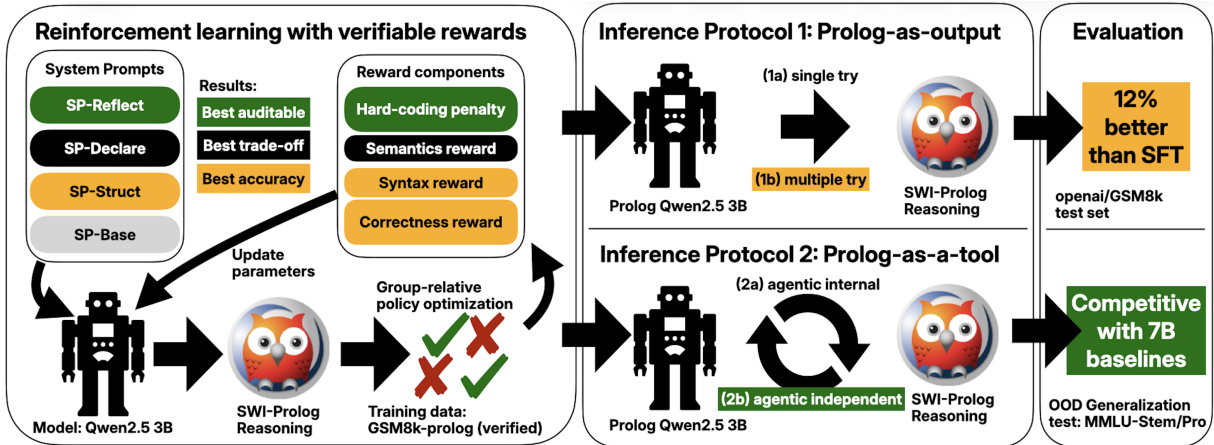


Figure 1: Overview of our training procedure for teaching a language model to produce executable Prolog code (Inference Protocol 1) and to use Prolog as a tool (Inference Protocol 2).

To teach a language model how to use Prolog via GRPO, we systematically study different system prompts, reward suites, and inference strategies.

Our results will establish that RLVR, and specifically GRPO, is well-suited to teach even a relatively small language model with only 3B parameters to use Prolog as a tool – as evidenced by strong results on mathematical and logical reasoning tasks (GSM8k, subsets of MMLU) that hit a weight-class of LLMs above (i.e., the 3B + Prolog models are comparable to 7B models). Moreover, the symbolic reasoning traces are spelled out as crisp logical rules and derivations processed by Prolog’s (faithful) symbolic reasoning engine. While our experiments are currently limited to maths (where most work on LLM-based reasoning originates), we envision that reasoning traces in Prolog (i.e., logical derivations) will be useful more broadly in safety-critical applications (e.g., healthcare, law).

Contributions In contrast to prior work that has relied on in-context learning or supervised fine-tuning to have language models emit Prolog code, we use reinforcement learning with verifiable rewards to teach language models to use Prolog as a tool. Our main contributions are three-fold:

First, we demonstrate that reinforcement learning with verifiable rewards enables small language models to use Prolog as an external tool, outperforming supervised fine-tuning baselines. Critically, our 3B model with agentic inference achieves zero-shot performance on MMLU benchmarks close to few-shot performance of 7B models, showing that Prolog tool use can compensate for scale.

Second, we conduct a systematic investigation across prompt structures, reward compositions, and

inference protocols, revealing that: (a) prompt-reward interactions shape program syntax and logic; (b) best-of-N with external verification maximizes in-distribution accuracy; and (c) agentic inference with internal repair yields superior zero-shot generalization under distribution shift (from GSM8k to MMLU).

Third, we identify an accuracy–auditability trade-off: configurations tuned for correctness learn to delegate reasoning to natural language, while structure-tuned configurations produce fully symbolic but less accurate programs. This finding has direct implications for practitioners deploying LLM-based neurosymbolic systems.

We interpret this trade-off as a form of reward hacking: when the reward signal only checks final-answer correctness, the model learns to satisfy it by delegating reasoning to natural language and using Prolog as a minimal output wrapper, circumventing the symbolic-reasoning behavior the reward was intended to induce.

2 Related Work

Our work brings together reasoning and tool use in large language models. Whereas prior work has regarded reasoning and tool use as separate steps (Yao et al., 2023), we hybridize this account and investigate using a symbolic reasoning engine as a tool. We will now lay out the related work on reasoning and tool use for symbolic reasoning.

Reasoning The idea of reasoning in large language models has its origins in the chain-of-thought prompting strategy (Wei et al., 2022), later used for crafting AI agents that interleave tool use and reasoning about the tools’ outputs (Yao et al., 2023).

More recently, studies have investigated how reasoning can be ingrained into language models through reinforcement learning with verifiable rewards on mathematical reasoning tasks (Shao et al., 2024; Guo et al., 2025). Specifically, DeepSeek-Math (Shao et al., 2024) has introduced group-relative policy optimization (GRPO), whereby the language model generates multiple reasoning traces and final outputs. The final output is compared to the ground truth and the reward is backpropagated through the reasoning trace that led to a positive outcome. DeepSeek-R1 (Guo et al., 2025) further shows that GRPO as sole post-training method induces reflection-like behavior, enabling test-time scaling (Muennighoff et al., 2025).

Recent work has questioned the faithfulness of chain-of-thought reasoning traces (Lanham et al., 2023; Paul et al., 2024), motivating our approach of integrating language models with a symbolic reasoning engine.

Symbolic Reasoning Tool use in language models originated in ToolFormer (Schick et al., 2023), where a language model was trained to use basic tools, such as a calculator to help with simple arithmetic. Early approaches on integrating Prolog with language models focus on prompting (Tan et al., 2024) and LoRA finetuning for Prolog generation while leveraging permutation invariance (Yang et al., 2024b). Borazjanizadeh and Piantadosi (2024) introduce a pure zero-shot neurosymbolic Prolog pipeline that prompts a pre-trained LLM to emit Prolog predicates and rules without any fine-tuning – and then executes them in SWI-Prolog (a Prolog interpreter). With a multiplicity inference loop – sampling up to N candidate programs and halting on the first one that executes successfully – they achieve substantial gains in accuracy: When using GPT-3.5 Turbo + Prolog with multiple-try inference, they report 80.4% accuracy versus 57.1% for text-only CoT, and GPT-4 + Prolog reaches 95.2% versus 92.0% text-only CoT.

Concurrently to our work, Pennino et al. (2025) also studied using GRPO for teaching a language model to use Prolog through a Python bridge (PySwip). Despite similarities to our work, the results and findings are different: Our systematic exploration of combinations of system prompts, reward suites, and agentic protocols, has enabled our approach to challenge larger language models and revealed an auditability-accuracy trade-off, which we expect to inform future work in the area.

3 Methods

Our aim is to teach a relatively small language model to use Prolog as a tool. Beyond the tool-calling syntax, the language model must be capable of emitting proper Prolog code that ideally models the example at hand.

Prolog is a declarative programming language in which facts and rules can be described and subsequently processed by a symbolic reasoning engine. A minimal example of a Prolog program is `a. b :- a. c :- b. ?- c.`, which would lead the reasoning engine to conclude that `c` is true. Prolog fully implements first-order predicate logic.

The rationale here is that through providing an interface to Prolog’s reasoning engine, we should be able to improve language models’ capabilities to carry out sound reasoning as the logical derivations are performed by a symbolic reasoning engine. The reasoning process, as executed via Prolog code, is therefore verifiable and auditable.

Given a dataset of reasoning tasks and numerical ground truth values $\{\mathbf{x}_i, y_i\}_{i < N}$, we have a language model generate G responses for each $x_i, i < N$. Those responses that lead to the correct outcome are assigned a positive reward, with the possibility of combining this main reward function with extra reward functions, e.g., nudging the model to produce syntactically correct Prolog code and preventing the model from circumventing the task through a print statement.

The standard way to teach a language model to use new tools is through supervised fine-tuning (Schick et al., 2023), which we consider our baseline. Our key hypothesis is that reinforcement learning with verifiable rewards may be better suited to teach a language model to use Prolog as a tool than supervised fine-tuning.

We experiment with four carefully crafted system prompts (as will be detailed in §3.1), three different reward suites (§3.2), and two types of inference protocols: `prolog-as-output` and `prolog-as-a-tool`, each of which instantiated with two different concrete protocols (§3.3).

3.1 System Prompts

We evaluate four system prompts varying in structural constraint: `SP-Base` provides a minimal two-part XML template; `SP-Struct` enforces explicit code layout with numbered reasoning steps; `SP-Declare`, inspired by Tan et al. (2024), requires every numeric constant to be encoded as a named

predicate, ensuring reasoning is captured symbolically rather than in natural language; and SP-Reflect adds a self-correction loop where the model reviews its reasoning before emitting code. Full prompts appear in Appendix A.

3.2 Reward Composition

As GRPO is based on reinforcement learning, it allows us to flexibly define a reward function (that does not need to be differentiable). We experiment with three different reward suites.

Reward suite 1 (Rwd1) combines four signals: correctness (comparing executed output to ground truth), Prolog syntax (detecting hallmark constructs like `:-` and `solve/1`), and soft/strict format rewards for XML schema compliance.

Reward suite 2 (Rwd2) adds semantic similarity via Sentence-BERT embeddings and predicate-name overlap. The semantic similarity component promotes code that corresponds to reference solutions even when surface forms differ, encouraging meaningful use of Prolog.

Reward suite 3 (Rwd3) introduces curriculum-guided weighting that shifts from format to correctness as training progresses. Reward suite 3 additionally penalizes hard-coded solutions where `solve/1` directly assigns a numeric literal, scaling the structure reward by 0.2 to discourage circumventing symbolic reasoning. Table 1 summarizes the composition. Full details of the three reward suites are provided in Appendix B.

Reward components	Rwd1	Rwd2	Rwd3
Correctness	✓	✓	✓
Prolog Syntax	✓	✓	✓
Response Format	✓	✓	✓
XML Tag Count	✓	✓	✓
Semantic Similarity		✓	✓
Predicate Name Overlap		✓	✓
Prolog Structure			✓
<code>prolog_helpers.pl</code>			✓
Hard-coding penalty			✓
Curriculum weighting			✓
Multi-objective balancing			✓
Normalization and clipping			✓

Table 1: Composition of different reward suites.

3.3 Inference Protocols

We evaluate our Prolog-grounded reasoning pipeline under four complementary protocols: **Single-Try**, **Multiple-Try**, **Agentic Internal**, and **Agentic Independent**. Each protocol is run at a fixed decoding temperature of 0.2 throughout,

consistent with common practice for reasoning-focused generation.

Prolog-as-output In both single-try inference and multiple-try inference protocols, SWI-Prolog is used *post hoc* as an external tool invoked only after the LLM produces an answer. The model has no visibility into the execution result and cannot revise its output in response. This setup reflects a *generate-then-verify* paradigm, where symbolic correctness is assessed retrospectively.

In the first protocol, **Single-Try** inference, we flatten the system prompt and user messages into one prompt per problem, extract the `<answer>...</answer>` block, and execute it in SWI-Prolog. This protocol estimates $P(\text{correct} \mid \text{prompt})$ without retries or feedback.

The second prolog-as-output protocol, **Multiple-Try** inference, leverages sampling diversity by drawing up to $N = 20$ independent completions per prompt. We halt at the first response that produces an integer or float. We record the number of tries until first success. This best-of- N decoding converts capability into effective accuracy and mirrors the setup of earlier works (Tan et al., 2024; Borazjanizadeh and Piantadosi, 2024).

Prolog-as-a-tool In both Agentic Internal and Agentic Independent, SWI-Prolog acts as an interactive, callable engine, transforming the LLM’s behavior from passive code generation to tool-augmented reasoning with verification capabilities.

These protocols treat SWI-Prolog as an interactive tool accessible *during* generation. We define a callable function `run_prolog()` using structured `<tool_call>` blocks from the Qwen tokenizer’s chat template. The model invokes this tool mid-dialogue, receives execution results, and integrates the feedback into subsequent reasoning turns, thereby enabling iterative refinement.

In the **Agentic-Internal** protocol, we embed a self-reflective correction loop within a single session of up to 20 turns. After each decode-execute step, failures (syntax errors, unbound variables, recursion timeouts, or non-numeric outputs) trigger targeted feedback and enable the language model to iterate.

We also “shake” the temperature (multiplying by 1.15 up to 0.3) if repeated or empty generations occur, and compress older messages into summaries once the prompt length exceeds 95% of the 2048-token context budget. This design enables in-place correction and self-verification (Tan et al., 2024).

In the **Agentic-Independent** protocol, we treat each generation-execution-reflection cycle as a bounded, self-contained “agentic try”. We allow up to 20 total turns per problem, but unlike the Agentic Internal strategy, we discard context and reset the session entirely whenever we detect signs of persistent failure, e.g., repeated empty generations, invalid numeric outputs, or excessive duplicate attempts. Each fresh retry reinitializes the conversation with only the original system and user messages, exploring a new trajectory in the model’s sampling space. This hard-reset policy helps escape local minima and recursion traps that in-place correction may reinforce. Within each problem, the total retry budget is preserved: the number of turns used in each session is subtracted from the global cap of 20. We continue until either (i) a valid numeric result is produced, or (ii) the full budget is exhausted.

For both agentic protocols, the system prompt is extended by a tool declaration that describes how the Prolog tool can be invoked. Exact prompts can be found in Appendix C.

4 Experimental Setup

Datasets and pre-processing We build upon the *gsm8k-prolog* dataset (Yang et al., 2024b), which provides an SWI-Prolog implementation for each *GSM8K* problem from the original `openai/gsm8k` dataset. For every example, we run the reference Prolog code under `CLP(Q)` and compare its numeric result to the original `openai/gsm8k` answer.

Out of 7473 problems, we found 15 discrepancies: 14 arose from errors in the OpenAI *GSM8K* answers (error rate of 0.1874% in the original dataset), and one from the Prolog references. We manually recomputed and reformatted each correct answer to match the official style, then updated both splits and created the cleaned *gsm8k-prolog-prover* dataset, now fully consistent under SWI-Prolog execution.

Throughout our experiments we will use a subset of 2,500 examples from this new *gsm8k-prolog-prover* dataset. We split this subset into 1750 training examples, 375 validation examples, and hold out another 375 examples as a test set. For testing, we also report scores on the official *GSM8k* test set as well as a zero-shot generalization test on *MMLU-Stem* (Hendrycks et al., 2021) and *MMLU-Pro* (Wang et al., 2024).

Implementation details All experiments use a 4-bit quantized *Qwen2.5-3B-Instruct* (Yang et al., 2024a) and attach a LoRA (Hu et al., 2022) adapter for parameter-efficient GRPO training on a 40 GB GPU. After training, we merge the LoRA weights into the base checkpoint to obtain a full 16-bit model for downstream evaluation.

Hyperparameters We train for one epoch with the AdamW optimizer, a batch size of 8, a learning rate of $5 \cdot 10^{-6}$, a cosine learning rate schedule, weight decay of 0.1, and gradient clipping to 0.1. Random seeds are kept fixed. We ran a systematic sweep over relevant hyperparameters, but the best configuration did not improve over these manually-optimized ones. Details can be found in Appendix F.

Baselines As our main baseline, we contrast our approach with supervised fine-tuning (SFT) baseline, which employs a standard causal language modeling objective. For perfect comparability, we use the same data and the same base model as in our proposed GRPO training strategy. We further use the same hyperparameters as those used for our GRPO training. For reference, we also include the performance of larger and more powerful models, such as *DeepSeekMath-7B-RL*, *Mistral-7B* and *Gemma-7B* on the official *GSM8k* test split and on *MMLU-STEM/Pro*.

Evaluation Measures For every generated response, we evaluate three complementary metrics. First, the main evaluation is **accuracy** measuring whether the output parses as a valid integer or float and exactly matches the numeric answer in the ground truth.

Second, **structural validity** checks whether the generated Prolog code includes at least one user-defined predicate other than `solve/1` and at least one arithmetic constraint enclosed in `{ . . . }`. This is assessed using the static analyzer `prolog_helpers.pl`. This metric captures the cases where the LLM reasons mainly in natural language and only emits a final “solve”-statement to satisfy the Prolog output constraint – which would result in low structural validity.

Third, **semantic similarity** quantifies how close the generated logic is to the reference Prolog program using cosine similarity over embeddings. We have devised this metric to measure auditability: Although variable naming is in principle arbitrary and does not change the logic of the program, we

argue that it contributes to auditability when the variable names are semantically related (as measured by cosine over contextualized embeddings) to the concepts found in the task description.

5 Results

Comparison of System Prompts and Reward Suites

We evaluate our GRPO training method on the `gsm8k-prolog-prover` validation dataset under all three reward suites and both Single-Try and Multiple-Try inference protocols. Table 2 shows the results. The most accurate model is `sp-struct-multipletry-rwd1`, which achieves 89.87% accuracy. Reward suite 1 consistently yields the highest accuracy across prompt variants.

Accuracy is highest under SP-Struct system prompt with reward suite 1. The best-performing models in semantic similarity use the SP-Declare prompt with reward suite 3. The best model in terms of structural validity is based on the SP-Declare prompt with reward suite 1. For SP-Struct and SP-Declare, we generally observe large gains in structure score under reward suite 3. For example, SP-Declare improves from 49.60% (Rwd1) to 94.67% (Rwd3) in structural validity. The only exception is SP-Base, which consistently underperforms in structural validity. Moreover, the multiple-try strategy consistently yields substantial accuracy gains over single-try inference.

Comparison of Prolog-as-a-Tool Inference Protocols

Next, we evaluate the agentic internal and agentic independent inference protocols with the best performing reward suite 1 and all system prompts. As shown in Table 3, the most accurate results (86.13%) are obtained with the Agentic Independent protocols combined with either the SP-Base or the SP-Struct system prompt. However, the second best results with an accuracy of 84.27% are obtained by the agentic internal setting. Notably, the SP-Declare system prompt excels in semantic similarity and structure.

GRPO versus SFT Baseline We test whether our proposed GRPO-based training method outperforms the supervised fine-tuning (SFT) baseline in a fully controlled environment (same starting model, same hyperparameters, same training data). As shown in Table 4, the best GRPO method exceeds the best SFT method by more than 10 accuracy points (90% vs. 79%). On average across the

four inference methods, GRPO training yields 56% higher accuracy scores than the SFT baseline.

Results for the official GSM8k Test Split We evaluate both the fully-trained and subset-trained variants of the best-performing configuration, `sp-struct-rwd1`, on the official `openai/gsm8k` test split (1320 examples). Each model is tested across all four inference strategies. Table 5 shows the results. Although our best GRPO-trained 3B model surpasses DeepSeekMath-7B-RL on our validation set with multiple-try inference (89.87% vs. 86.7%), it scores only 80.21%, below DeepSeekMath’s 86.7% on the GSM8K test set.

The full GSM8K test set, containing more varied formulations, favors Agentic-Internal, which accumulates debugging context across attempts. Despite being trained on only 23.4% of the data, the subset model is only 1–2 points behind the fully-trained model.

Zero-Shot Generalization Test

We measure cross-dataset generalisation by evaluating the fully-trained model on the multiple-choice datasets MMLU-STEM and MMLU-Pro. The **STEM** dataset covers physics, chemistry, biology, and engineering disciplines and the **Pro** dataset covers law, finance, medicine, philosophy – contributing 375 validation questions each. We keep the original SP-Struct prompt and prompt for the *zero-based index* of the correct multiple-choice option without any in-context examples (zero-shot). Details can be found in Appendix E.

As shown in Table 6, the agentic inference protocols, *Internal* and *Independent*, consistently outperform Single-Try and Multiple-Try inference. Specifically, on MMLU-STEM, our full-data model improves from 53.6% (Multiple-Try) to 58.13% (Agentic Independent). MMLU-Pro shows an improvement from 26.67% (Multiple-Try) to 30.7% (Agentic Internal). Interestingly, the results from our models under *zero-shot* are comparable to models with a 7B parameter budget from the DeepSeekMath, Mistral, and Gemma families with *few-shot prompting* with 5 examples available in the prompt to facilitate in-context learning. Note that this evaluation primarily tests generalization of the tool-augmented language model to other evaluation benchmarks rather than testing specifically for more complex symbolic reasoning.

Qualitative Analysis Analyzing the logfiles for `sp-struct-rwd1` (examples in Appendix D), we

Prompt	Training	Inference	Acc \uparrow	Sem \uparrow	Struc \uparrow
SP-Base	GRPO w/ Rwd1	Single-Try	77.87	3.20	0.00
	GRPO w/ Rwd2	Single-Try	80.27	17.60	0.27
	GRPO w/ Rwd3	Single-Try	82.67	10.13	0.27
	GRPO w/ Rwd1	Multiple-Try	85.33	2.13	0.27
	GRPO w/ Rwd2	Multiple-Try	84.53	20.00	0.27
	GRPO w/ Rwd3	Multiple-Try	82.93	9.33	0.00
SP-Struct	GRPO w/ Rwd1	Single-Try	85.33	5.60	1.07
	GRPO w/ Rwd2	Single-Try	73.60	29.07	3.20
	GRPO w/ Rwd3	Single-Try	64.27	29.33	66.93
	GRPO w/ Rwd1	Multiple-Try	89.87	8.27	1.60
	GRPO w/ Rwd2	Multiple-Try	84.53	32.00	3.20
	GRPO w/ Rwd3	Multiple-Try	83.73	40.00	89.60
SP-Declare	GRPO w/ Rwd1	Single-Try	34.40	31.20	49.60
	GRPO w/ Rwd2	Single-Try	33.07	33.33	46.40
	GRPO w/ Rwd3	Single-Try	26.27	26.27	42.40
	GRPO w/ Rwd1	Multiple-Try	66.93	59.73	94.67
	GRPO w/ Rwd2	Multiple-Try	73.60	<u>62.13</u>	<u>92.53</u>
	GRPO w/ Rwd3	Multiple-Try	62.40	62.67	90.40
SP-Reflect	GRPO w/ Rwd1	Single-Try	74.40	1.60	0.80
	GRPO w/ Rwd2	Single-Try	79.73	4.53	0.00
	GRPO w/ Rwd3	Single-Try	72.00	22.40	76.27
	GRPO w/ Rwd1	Multiple-Try	<u>88.53</u>	2.93	1.33
	GRPO w/ Rwd2	Multiple-Try	88.27	2.93	0.27
	GRPO w/ Rwd3	Multiple-Try	87.73	26.13	91.47

Table 2: Validation results of GRPO-trained Qwen-2.5 3B models across prompt variants, reward suites, and inference strategies. Accuracy (**Acc**), semantic similarity (**Sem**), and structure validity (**Struc**) are averaged over gsm8k-prolog-prover validation set (375 samples). Best results per metric are marked **bold**, second best underlined.

Prompt	Agentic Mode	Acc	Sem	Struc
SP-Base	Internal	82.93	58.52	0.80
	Independent	86.13	58.95	0.27
SP-Struct	Internal	<u>84.27</u>	55.18	5.07
	Independent	86.13	56.08	5.87
SP-Declare	Internal	69.87	81.36	73.07
	Independent	72.00	80.68	72.53
SP-Reflect	Internal	<u>84.27</u>	54.57	9.07
	Independent	83.47	54.57	10.40

Table 3: Agentic evaluation on gsm8k-prolog-prover validation set. For each prompt variant, we report accuracy (**Acc**), semantic similarity (**Sem**), and structural correctness (**Struc**) using reward suite 1.

observe that the model delegates final computations to SWI-Prolog via `solve/1`, but performs its intermediate reasoning in natural language. This explains the high accuracy yet poor semantic and structural alignment (sem: 8.27%, struc: 1.60% in multiple-try). It reaches correct answers but deviates from the Prolog reference solutions of the gsm8k-prolog-prover dataset. In contrast, `sp-declare-rwd2` emits Prolog code that matches reference structures and achieves strong semantic similarity (62.13%) and structural correctness (92.53%), but at the cost of lower accu-

racy (73.60%). The declarative prompt improves code form further at the cost of accuracy. To test whether this pattern generalizes beyond individual examples, we next conduct a systematic error analysis across the best-performing configurations per metric.

Systematic Error Analysis We select three runs from Table 2, one per evaluation metric: the SP-Struct prompt with reward suite 1 for the highest accuracy (90%), SP-Declare with reward suite 1 achieves the highest structural validity (95%), and SP-Declare with reward suite 3 for the highest semantic similarity. A systematic error analysis reveals a clear auditability-accuracy trade-off (see Figure 2). The SP-Struct run almost never requires retries (1.08 attempts on average), and when it fails it is almost always a generation failure: the model did not produce valid code, which leads to the next attempt. Its dominant failure mode is wrong answers (10% of samples), suggesting that the minimal prompt produces code that executes cleanly but occasionally encodes incorrect reasoning. The SP-Declare runs show the inverse pattern: both require substantially more attempts (2.4 and 3.0 on average, up to 17–20) and produce far more wrong answers (31% and 36%). Their ex-

Training method	Prompt	Reward	Inference	Acc	Sem	Struc
SFT (baseline)	SP-Struct	—	Single-try	21.60	20.80	64.27
SFT (baseline)	SP-Struct	—	Multiple-try	65.07	36.00	66.13
SFT (baseline)	SP-Struct	—	Agentic Internal	79.20	68.03	49.60
SFT (baseline)	SP-Struct	—	Agentic Independent	55.46	73.98	52.53
GRPO (ours)	SP-Struct	Rwd1	Single-try	85.33	5.60	1.07
GRPO (ours)	SP-Struct	Rwd1	Multiple-try	89.87	8.27	1.60
GRPO (ours)	SP-Struct	Rwd1	Agentic Internal	84.27	55.18	5.07
GRPO (ours)	SP-Struct	Rwd1	Agentic Independent	<u>86.13</u>	56.08	5.87

Table 4: Comparison of GRPO versus SFT on the gsm8k-prolog-prover validation set

Model	Inference Protocol	Accuracy
Qwen2.5-7B-Instruct (Yang et al., 2024a)		91.6
DeepSeekMath-7B (Shao et al., 2024)		86.7
Qwen2.5-3B-Instruct SP-Struct RWD1 (7473 examples)	Single-Try	75.28
	Multiple-Try	80.21
	Agentic-Internal	78.17
	Agentic-Independent	77.86
Qwen2.5-3B-Instruct SP-Struct RWD1 (1750 examples)	Single-Try	73.99
	Multiple-Try	<u>78.24</u>
	Agentic-Internal	77.20
	Agentic-Independent	76.80

Table 5: Performance of full-data vs. subset-trained models on the official openai/gsm8k test split. **Best** and second-best compared 3B variants are highlighted.

Model	MMLU-Stem	MMLU-Pro
Our 3B models with zero-shot prompting		
Single-Try	50.93	21.33
Multiple-Try	53.60	26.67
Agentic-Internal	<u>56.80</u>	30.67
Agentic-Independent	58.13	<u>29.33</u>
7B baseline models with few-shot prompting		
DeepSeekMath-Base 7B	56.50	—
Mistral 7B	<u>51.10</u>	30.90
Gemma 7B	—	33.70

Table 6: Zero-shot evaluation of our 3B Qwen-2.5 models trained with GRPO via SP-Struct-RWD1-full on the validation set with our four inference protocols. Baseline 7B models use few-shot prompting. Our Prolog-enhanced 3B model can close the gap to 7B models even without extra examples. **Best** and second best results are highlighted per model size group. MMLU-Pro results for Mistral 7B and Gemma 7B are taken from Wang et al. (2024). MMLU-Stem results for DeepSeekMath-Base 7B and Mistral 7B are from Shao et al. (2024).

ecution failures additionally include uninstantiated variables (11–16% of failed attempts), which arise when the declarative constraint structure is too underspecified to ground the answer. Taken together, the results suggest that the declarative prompt successfully induces structural validity but at the cost of introducing more complex constraints that the model struggles to get right, and that retrying under the same prompt only helps to some extent: samples that ultimately fail tend to do so because the underlying reasoning is wrong, not because the code is malformed.

6 Discussion

Our experiments reveal several key insights about reinforcement learning for integrating symbolic reasoning into language models, which we discuss in the following.

Accuracy vs. Auditability Trade-off We find that simple execution-based rewards yield the highest accuracy, but this comes with a trade-off: models learn to emit minimal Prolog that delegates reasoning to natural language. The highest-accuracy configuration (sp-struct-rwd1, 89.87%) produces Prolog code with minimal structural validity (1.60%) and low semantic similarity (8.27%). Qualitative analysis confirms this model mostly reasons in natural language and uses Prolog only as much as needed, i.e., for producing the final output. Reward suites 2 and 3 on SP-Declare achieve up to 92.5% structural validity and 62% semantic similarity, but accuracy drops to 62–74%. When these reward signals are omitted, the models discover that reasoning can be deferred to natural language and only a final minimal Prolog code snippet is needed. While this trade-off requires further investigation, we can so far conclude that richer reward signals successfully steer models toward more symbolic reasoning, but additional structural constraints introduce opportunities for error.

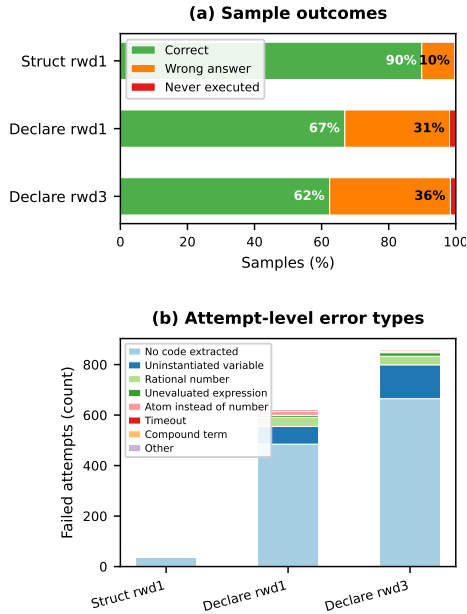


Figure 2: Error analysis across the best-performing runs per metric. (a) Sample-level outcomes: the Struct prompt achieves high execution accuracy (90%) but at the cost of structural validity, while both Declare variants produce more wrong answers (31–36%). (b) Distribution of error types across all failed attempts: the majority are generation failures (no code extracted), with uninstantiated variables being the most common execution-level error and disproportionately affecting the more complex Declare prompts.

This pattern is a clear instance of reward hacking (Skalse et al., 2022): the correctness reward is a proxy for “reason symbolically and arrive at the right answer,” but it only checks the latter. The model discovers that it can maximize the proxy by reasoning in natural language and emitting a minimal Prolog snippet – behavior that is rewarded just as much as genuine symbolic reasoning, despite defeating the purpose of using Prolog in the first place. Reward suites 2 and 3 attempt to close this gap by adding structural and semantic signals that more directly specify the intended behavior, and they do induce more symbolic programs – but at a cost in accuracy, likely because the richer reward landscape is harder to optimize jointly. Reward hacking of this kind has recently been linked to emergent misalignment (Betley et al., 2026; MacDiarmid et al., 2025).

Inference Protocol Selection Execution-based rewards combined with multiple-try inference produce the best in-distribution accuracy: Reward

suite 1 outperforms more complex semantic and structural signals across all prompt variants. In the tool use setup, Agentic-Independent slightly outperformed Agentic-Internal, presumably due to hard resets to escape local traps. On the full openai/GSM8K test set, however, Agentic-Internal yielded the best results, presumably due to preserving debugging context for incremental refinement. Generally, multiple-try inference suffices for narrow arithmetic problems, whereas interactive self-repair becomes crucial under distribution shift, such as on our MMLU generalization tests.

Implications for Safety-critical Domains Our experiments use GSM8K as a controlled testbed, yet our results have direct implications for applications in domains where auditability is crucial, such as in the health domain and the legal domain. The accuracy–auditability trade-off we identify suggests practitioners must choose configurations based on whether their application prioritizes raw performance or auditability.

7 Conclusion

We have introduced a training pipeline, including concrete system prompts and reward suites, to teach language models to use Prolog as a tool via reinforcement learning with verifiable rewards. Our findings show that: (1) the interaction between system prompt and reward function shapes the syntactic and logical form of generated Prolog programs; (2) multiple-try decoding with Prolog verification achieves the highest accuracy on in-distribution tasks; and (3) agentic inference with an internal repair strategy yields superior zero-shot generalization across distribution shifts. We further demonstrate that GRPO training beats an SFT baseline and that a 3B model with agentic inference can match 7B baselines on MMLU, indicating that symbolic tool use can partially compensate for scale.

Most importantly, our results expose an accuracy–auditability trade-off: correctness-optimized configurations delegate reasoning to natural language and use Prolog only for the final computation, while structure-rewarded configurations produce fully symbolic programs that are less accurate. Symbolic grounding alone does not deliver auditability and the reward design must take it into account. Practitioners deploying such systems in safety-critical domains should treat the choice of reward composition and inference protocol as a deliberate position on this trade-off.

8 Limitations

This work is limited to elementary arithmetic reasoning problems (GSM8K), using a single tool (SWI-Prolog), and one reinforcement method (GRPO). Generalization to richer domains, such as probabilistic programming (Bingham et al., 2019), remains unexplored, as does integrating multiple tools. Lastly, reward shaping currently follows a fixed curriculum; future work may benefit from dynamic reward schedules based on validation metrics. Our MMLU evaluation primarily tests the model’s ability to emit well-formed Prolog for encoding answer indices.

9 Ethical Considerations

Compared to chain-of-thought approaches, our approach of relying on a symbolic reasoning engine is intended to improve auditability of a language model’s reasoning process. We do not expect that our method introduces or exacerbates any risks. The only caveat here is that, under our basic reward suite 1, the system may circumvent reasoning via Prolog – as discussed extensively in the Discussion. However, this lack-of-faithfulness concern equally applies to natural-language reasoning traces, and one of our contributions lies in identifying an accuracy–auditability trade-off in neurosymbolic reasoning systems.

Acknowledgments

This research was supported in part by the MIST project, funded by the Novo Nordisk foundation under grant reference number NNF25OC0103204. The research was further supported in part by the Danish Foundation Models project, funded by the Danish government. Part of the computation done for this project was performed on the UCloud interactive HPC system managed by the eScience Center at the University of Southern Denmark.

References

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. [Scheduled sampling for sequence prediction with recurrent neural networks](#). In *NeurIPS*, pages 1171–1179.

Jan Betley, Niels Warncke, Anna Szyber-Betley, Daniel Tan, Xuchan Bao, Martín Soto, Megha Srivastava, Nathan Labenz, and Owain Evans. 2026. [Training large language models on narrow tasks can lead to broad misalignment](#). *Nat.*, 649(8097):584–589.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. [Pyro: Deep universal probabilistic programming](#). *J. Mach. Learn. Res.*, 20:28:1–28:6.

Nasim Borazjanizadeh and Steven Piantadosi. 2024. [Reliable reasoning beyond natural language](#). *Preprint*, arXiv:2407.11373.

Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V. Le, Sergey Levine, and Yi Ma. 2025. [SFT memorizes, RL generalizes: A comparative study of foundation model post-training](#). In *ICML*, Proceedings of Machine Learning Research. PMLR / OpenReview.net.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, and 175 others. 2025. [Deepseek-R1 incentivizes reasoning in LLMs through reinforcement learning](#). *Nat.*, 645(8081):633–638.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. [Measuring massive multitask language understanding](#). In *ICLR*. OpenReview.net.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. [LoRA: Low-rank adaptation of large language models](#). In *ICLR*. OpenReview.net.

Tomek Korbak, Mikita Balesni, Elizabeth Barnes, Yoshua Bengio, Joe Benton, Joseph Bloom, Mark Chen, Alan Cooney, Allan Dafoe, Anca Dragan, Scott Emmons, Owain Evans, David Farhi, Ryan Greenblatt, Dan Hendrycks, Marius Hobbhahn, Evan Hubinger, Geoffrey Irving, Erik Jenner, and 22 others. 2025. [Chain of thought monitorability: A new and fragile opportunity for AI safety](#). *Preprint*, arXiv:2507.11473.

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, and 1 others. 2023. [Measuring faithfulness in chain-of-thought reasoning](#). *Preprint*, arXiv:2307.13702.

Monte MacDiarmid, Benjamin Wright, Jonathan Uesato, Joe Benton, Jonathan Kutasov, Sara Price, Naia Bouscal, Samuel R. Bowman, Trenton Bricken, Alex Cloud, Carson Denison, Johannes Gasteiger, Ryan Greenblatt, Jan Leike, Jack Lindsey, Vladimir Mikulik, Ethan Perez, Alex Rodriguez, Drake Thomas, and 3 others. 2025. [Natural emergent misalignment from reward hacking in production RL](#). *Preprint*, arXiv:2511.18397.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke

- Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. [s1: Simple test-time scaling](#). In *EMNLP*, pages 20275–20321.
- OpenAI. 2024. [OpenAI o1 system card](#). *Preprint*, arXiv:2412.16720.
- Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. 2013. [On the difficulty of training recurrent neural networks](#). In *ICML*, pages 1310–1318. JMLR.org.
- Debjit Paul, Robert West, Antoine Bosselut, and Boi Faltings. 2024. [Making reasoning matter: Measuring and improving faithfulness of chain-of-thought reasoning](#). In *EMNLP Findings*, pages 15012–15032.
- Federico Pennino, Bianca Raimondi, Massimo Rondelli, Andrea Gurioli, and Maurizio Gabbriellini. 2025. [From reasoning to code: GRPO optimization for underrepresented languages](#). *Preprint*, arXiv:2506.11027.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. [Toolformer: Language models can teach themselves to use tools](#). In *NeurIPS*.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Yongan Li, Yantao Wu, and Daya Guo. 2024. [DeepSeekMath: Pushing the limits of mathematical reasoning in open language models](#). *Preprint*, arXiv:2402.03300.
- Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. 2022. [Defining and characterizing reward gaming](#). In *NeurIPS*.
- Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. [Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning](#). In *ICLR*. OpenReview.net.
- Xiaoyu Tan, Yongxin Deng, Xihe Qiu, Weidi Xu, Chao Qu, Wei Chu, Yinghui Xu, and Yuan Qi. 2024. [Thought-Like-Pro: Enhancing reasoning of large language models through self-driven Prolog-based chain-of-thought](#). *Preprint*, arXiv:2407.14562.
- Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyang Jiang, and 1 others. 2024. [MMLU-Pro: A more robust and challenging multi-task language understanding benchmark](#). In *NeurIPS*, pages 95266–95290.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. [Chain-of-thought prompting elicits reasoning in large language models](#). In *NeurIPS*, pages 24824–24837.
- Will Brown. 2025. [grpo_demo.py](#). GitHub Gist. Last accessed: April 18, 2026.
- An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024a. [Qwen2.5 technical report](#). *Preprint*, arXiv:2412.15115.
- Xiaocheng Yang, Bingsen Chen, and Yik-Cheung Tam. 2024b. [Arithmetic reasoning with LLM: Prolog generation & permutation](#). In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pages 699–710.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. [ReAct: Synergizing reasoning and acting in language models](#). In *ICLR*. OpenReview.net.

Appendix

A System Prompts in Detail

SP Base: Minimal Prompt for Symbolic Grounding

The SP Base system prompt establishes a minimal yet effective template for aligning LLMs with symbolic execution engines like Prolog. It defines a fixed two-part XML schema: a `<reasoning>` section for short, structured step-by-step logic, and an `<answer>` section that emits Prolog code with a single constraint-solving clause (`solve(X)`). The Prolog code always includes `:- use_module(library(clpq)).`, ensuring compatibility with constraint logic programming (CLP(Q)). SP Base is also consistent with Tan et al. (2024), who demonstrate that lightweight Prolog-based prompts form an effective baseline for CoT-based arithmetic reasoning, when paired with strict symbolic evaluation.

```
You are a Prolog assistant specialized
in solving math problems.
```

```
Provide your solution strictly in
this XML format:
```

```
<reasoning>
- Give concise step-by-step reasoning
here.
</reasoning>
<answer>
:- use_module(library(clpq)).

solve(X) :-
{X = final numeric answer}.
</answer>
```

SP Struct: Structured Output with Explicit Code Layout

The SP Struct prompt incentivizes alignment with symbolic programming language

by enforcing a fixed internal structure within each output section. The <reasoning> block guides the model to decompose the task into ordered, numbered logical steps. The <answer> block is explicitly formatted: code must start with the CLP(Q) import, define constants or intermediate predicates (if needed), and conclude with a solve/1 clause written entirely using curly-brace constraints. This prompt incentivizes breaking complex tasks into simpler subtasks and building modular prompts that produce more predictable structured outputs.

You are a specialized Prolog code-generating assistant.

Your task is to solve math problems by providing a structured answer in two clearly defined sections:

1. <reasoning>
 - Provide a clear, concise step-by-step explanation of how you arrive at the solution.
2. <answer>
 - Provide executable Prolog code using constraint logic programming to compute the numeric answer.
 - Always start with: `:- use_module(library(clpq)).`
 - Define any necessary numeric constants or intermediate values using predicates.
 - Final answer should be unified explicitly in solve(X) using curly-brace constraints, without printing commands.

Use this XML format strictly:

```
<reasoning>
(Your step-by-step reasoning here)
</reasoning>
<answer>
:- use_module(library(clpq)).
```

(Any predicates/constants defined here)

```
solve(X) :-
(Intermediate computations using curly
braces)
{X = final constraint logic}.
</answer>
```

SP Declare: Declarative Abstraction and Predicate-Level Alignment Inspired by prior work, where reasoning traces are translated into predicates (Tan et al., 2024), the third system prompt, **SP Declare**, requires the model to encode every numeric constant mentioned in the problem statement as a named predicate (e.g., `bags_per_trip(james, 10)`). These constants must be queried and referenced symbolically inside the final solve/1 clause, and never embedded directly as literals.

You are a specialized Prolog code-generating assistant that must follow a strict structured format to solve math problems.

Your task is to solve math problems by providing an answer in two clearly defined sections:

1. <reasoning>
 - Provide a clear, concise, step-by-step explanation of your solution.
 - Explain how each numeric constant from the problem is represented by a predicate.
 - Do not include unnecessary calculations using literal numbers; instead, reference the predicates you define.

2. <answer>
 - Provide executable Prolog code using constraint logic programming (CLP) to compute the numeric answer.
 - Always start with: `:- use_module(library(clpq)).`
 - For every numeric constant mentioned in the problem, define a predicate with a descriptive name. For example, if the problem states that James carries 10 bags per trip, include: `bags_per_trip(james, 10)`. Similarly, define predicates for other constants (e.g., `trips_per_day(james, 20)`, `days(5)`).
 - In the solve predicate, retrieve each value by querying its predicate and use these values in your arithmetic constraints.
 - Use curly-brace constraints (e.g., `{Total = Bags * Trips * Days}`) to compute the final answer.
 - The final answer must be explicitly unified in the solve predicate (e.g., `solve(Total_bags) :- ...`).

Ensure your answer strictly follows this XML format:

```
<reasoning>
Your detailed, step-by-step reasoning
here, with references to the predicates
defined for numeric constants.
</reasoning>
<answer>
:- use_module(library(clpq)).
```

```
Define numeric constants as predicates:
bags_per_trip(james, 10).
trips_per_day(james, 20).
days(5).
```

```
solve(Total_bags) :-
bags_per_trip(james, Bags),
trips_per_day(james, Trips),
days(Days),
{Total_bags = Bags * Trips * Days}.
</answer>
```

Do not shortcut the process by embedding direct numeric literals in the solve predicate.

Every numeric constant must be defined via a predicate and then referenced in the arithmetic computations.

SP Reflect: Reflexive Reasoning and Self-Correction The SP Reflect prompt extends SP Struct by adding a built-in meta-cognitive loop. After the model completes its initial <reasoning>, it reviews its own logic to check for potential flaws. If issues are found, it should retry the reasoning step and only then emit the final Prolog code in <answer>. This embedded “reflection” mechanism enables error-checking and self-correction.

```
You are a specialized Prolog
code-generating assistant.
```

Your task is to solve math problems by providing a structured answer in two clearly defined sections:

1. <reasoning>
 - Provide a clear, concise step-by-step explanation of how you arrive at the solution.
 - Review the reasoning at the end of the <reasoning> section to ensure that all computations and logical deductions are correct.
 - If something is not correct, then try again: Provide a clear, concise step-by-step explanation of how you arrive at the solution.
2. <answer>
 - Provide executable Prolog code using constraint logic programming to compute the numeric answer.
 - Always start with: `:- use_module(library(clpq)).`
 - Define any necessary numeric constants or intermediate values using predicates.
 - Final answer should be unified explicitly in `solve(X)` using curly-brace constraints, without printing commands.

Use this XML format strictly:

```
<reasoning>
- Your step-by-step reasoning here
- Your review of the reasoning here
- Your potential further step-by-step
reasoning here
</reasoning>
<answer>
:- use_module(library(clpq)).
```

```
(Any predicates/constants defined
here)
```

```
solve(X) :-
(Intermediate computations using curly
braces)
{X = final constraint logic}.
</answer>
```

B Reward Suites in Detail

Here, we describe the three developed reward suites in detail. Each consecutive suite reward builds on top of the previous reward suite.

B.1 Details of Reward Suite 1: Structural Execution Shaping

In developing our first reward-shaping scheme, we build directly on the GRPO reward functions from [Will Brown \(2025\)](#), using them as a baseline to create a suite that nudges the model toward generating correct, executable Prolog code. The components of Reward Suite 1 are described below. Figure 3 shows the progression of the correctness reward over training steps.

Correctness Reward: We extract the model’s <answer>...</answer> block, execute it using SWI-Prolog, and compare the numeric output to the ground truth. Exact matches receive a score of 2.0; incorrect but numeric outputs earn 1.0; and any executable attempt, even with unbound variables or failures, earns 0.5.

Prolog Syntax Reward: Inspired by neurosymbolic scaffolding ([Borazjanizadeh and Piantadosi, 2024](#)), we award 0.2 points (up to 1.0) for detecting hallmark Prolog constructs: directives like `:- use_module`, the `solve/1` clause, and clause terminators (`.`). This rewards syntactic validity without a strict all-or-nothing reward.

Soft- and Strict-Format Rewards We provide both soft and strict format incentives. The *soft format reward* assigns 0.5 points if the output contains both a <reasoning>...</reasoning> block and a subsequent <answer>...</answer> block. The *strict format reward* grants an additional 0.5 only when the entire completion exactly matches a line-by-line XML regular expression.

XML-Count Heuristic: To support fine-grained compliance, we add a continuous schema-adherence score: each correctly placed XML tag (<reasoning>, </reasoning>, <answer>, </answer>) contributes 0.125, with a small per-character penalty applied to trailing text after </answer>.



Figure 3: Correctness reward progression during training across different system prompts under Reward Suite 1. By isolating the execution-based signal, we see that SP-Base, SP-Struct, and SP-Reflect reach the highest correctness reward levels. This indicates that minimal scaffolding plus executable output dominate performance on downstream tasks.

B.2 Details of Reward Suite 2: Semantic Similarity

Having ensured that outputs are executable and conform to the XML schema, we now introduce a reward that promotes code which corresponds to the reference code even when surface forms differ. In addition to the components of reward suite 1, we additionally employ semantic similarity.

The semantic similarity score $S_{\text{sem}} \in [0, 1]$ combines two complementary signals. First, we extract the contents of the `<answer>`-block from both the model output and the ground-truth data. Each code snippet is then embedded using the Sentence-BERT model `all-MiniLM-L6-v2`, and cosine similarity is computed between the resulting vectors, yielding $\cos \in [0, 1]$. In parallel, we calculate predicate-name overlap by identifying all functors of the form `name(` using a regular expression. Let $\mathcal{P}_{\text{model}}$ and \mathcal{P}_{ref} denote the sets of predicate names extracted from the model and reference programs, respectively. The normalized intersection is: $\text{pred} = \frac{|\mathcal{P}_{\text{model}} \cap \mathcal{P}_{\text{ref}}|}{\max(1, |\mathcal{P}_{\text{ref}}|)}$. The final semantic similarity score is the average of these two signals: $S_{\text{sem}} = \frac{1}{2}(\cos + \text{pred}) \in [0, 1]$. If either code block is missing or empty, we assign $S_{\text{sem}} = 0$. To align with the other reward scales, we rescale the semantic score to $[0.5, 2.0]$: $\text{reward} = \max(0.5, 2 \times S_{\text{sem}})$.

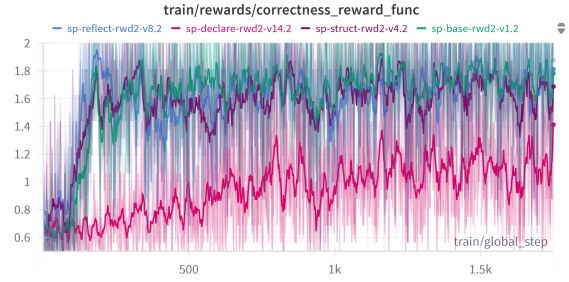
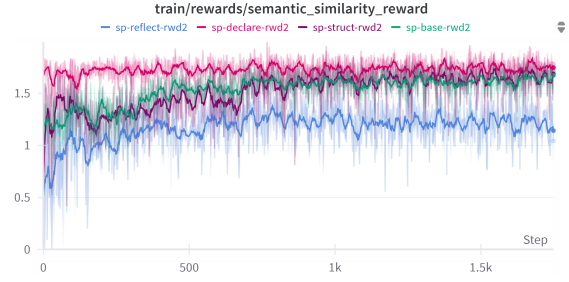


Figure 4: (Top): Semantic similarity reward across different prompt variants under Reward Suite 2. (Bottom): Correctness reward progression during training across different system prompts under Reward Suite 2.

Figure 4 reveals clear trends in semantic alignment across prompt variants. The `sp-declare` prompt consistently achieves the highest average semantic similarity, indicating that its predicate-level abstraction leads to closer alignment with reference Prolog programs. Other prompts such as `sp-struct` and `sp-base` converge to moderately high similarity, while `sp-reflect` underperforms. This confirms that Reward Suite 2’s embedding-based feedback meaningfully shapes model behavior toward semantically aligned and structurally consistent code.

B.3 Details of Reward Suite 3: Curriculum-Guided Structural Optimization

Our third reward suite builds on top of the semantic similarity and format shaping of Reward Suite 2 by adding Prolog-structure-sensitive rewards and a curriculum-driven schedule that gradually shifts the model from broad exploration toward focused exploitation as training progresses.

Prolog Structure Reward Each reference Prolog code:

1. Loads arithmetic constraints with `:- use_module(library(clpq)).;`
2. States problem facts as one-line clauses; and
3. Defines exactly one public predicate, `solve/1`, whose single argument is the final result.

A typical example:

```
:- use_module(library(clpqc)).

sell_clips(natalia, april, 48).

solve(Total) :-
    sell_clips(natalia, april, April),
    { May = April / 2 },
    { Total = April + May }.
```

prolog_helpers.pl. The helper script `prolog_helpers.pl` analyzes any candidate program and prints:

```
PREDICATE_COUNT — user predicates other than
    solve
CONSTRAINT_COUNT — goals inside { ... }
$ swipl -q -f prolog_helpers.pl \
-g "analyze_code('prog.pl',P,C),halt."
PREDICATE_COUNT: 1
CONSTRAINT_COUNT: 2
```

A program is marked *structurally valid* whenever `PREDICATE_COUNT` ≥ 1 and `CONSTRAINT_COUNT` ≥ 1 .

We convert the same counts into a scalar reward $S \in [0, 2]$:

```
sp = min(0.25 × PREDICATE_COUNT, 0.75)
sc = min(0.30 × CONSTRAINT_COUNT, 0.90)
Sraw = sp + sc
```

Hard-coding penalty. If `solve/1` contains a literal numeric assignment—detected via the pattern:

```
solve(...) :- ... = <number>. or
solve(...) :- { ... = <number> }.
```

then the score is scaled by 0.2:

$$S = 0.2 \times S_{\text{raw}}.$$

Otherwise, $S = S_{\text{raw}}$. The final value is clipped to $[0, 2]$ and returned as the `prolog_structure_reward`.

Rationale and percentage mapping. Each auxiliary predicate adds 0.25 up to a cap of 0.75 (three helpful helpers), and each arithmetic constraint adds 0.30 up to 0.90 (three useful constraints). Beyond these caps, further clauses no longer raise the reward, preventing inflation through repetition. The maximum attainable $S_{\text{raw}} = 0.75 + 0.90 = 1.65$ is linearly mapped to a 0–100% structural score:

$$\text{Struct\%} = \min\left(1, \frac{S}{1.65}\right) \times 100.$$

A program that reaches the caps and avoids hard-coding therefore receives 100% structural credit, whereas a hard-coded answer is immediately reduced to 20% of the score it would otherwise earn.

Curriculum-Based Weight Scheduling Instead of using a static mix of sub-rewards, we dynamically adjust weight allocations based on training progress $t \in [0, 1]$ —defined as the fraction of prompts seen—via a logistic function:

$$\sigma(t) = \frac{1}{1 + e^{-k(t-\tau)}}, \quad k = 12, \tau = 0.5.$$

This sigmoid-shaped trajectory has proven effective in curriculum learning (Bengio et al., 2015).

Multi-Objective Balancing We let $\text{early}[k]$ and $\text{late}[k]$ be the weight for sub-reward k at the beginning and end of training, respectively. We interpolate between them using the same sigmoid $\sigma(t)$:

$$\text{weights}[k] = \text{early}[k] + (\text{late}[k] - \text{early}[k]) \times \sigma(t).$$

Reward	Early Weight	Late Weight
Semantic	0.35	0.10
XML Format	0.25	0.10
Syntax	0.10	0.10
Correctness	0.15	0.45
Structure	0.15	0.25

Table 7: Early and late-stage reward weights used for curriculum interpolation in Reward Suite 3. These define the endpoints of the training schedule, with semantic and formatting objectives emphasized early, and correctness and structural reasoning prioritized late.

While Table 7 defines the curriculum endpoints, Figure 5 shows how weights evolve over time under our sigmoid interpolation, confirming the gradual shift in learning priorities.

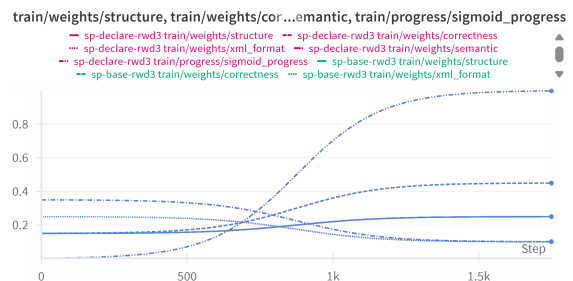


Figure 5: Interpolated reward weights over training steps, driven by the sigmoid progression schedule. Each curve shows how a specific sub-reward (e.g., structure, correctness) increases or decreases in emphasis as training progresses.

Reward Normalization and Clipping To avoid dominance effects due to scale mismatches across reward components, we normalize and clip all sub-rewards into a shared range. This prevents the “winner-takes-all” dynamic.

Final Aggregation For each example, we normalize and weight the five signals—**Semantic, Correctness, Structure, Syntax, and Format**—then sum the weighted values and scale the result into $[0, 2]$:

$$\text{final_reward} = 2.0 \times \sum_{k=1}^5 w_k \cdot r_k$$

This final scaling preserves the $[0, 2]$ reward range established in Suite 1 (where a perfect program earned 2.0), maintains compatibility with GRPO’s learning rate and gradient magnitudes, and ensures comparability across all three reward suites.

Over time, this curriculum drives the model from early-format scaffolding toward producing concise, logically rich, and syntactically robust Prolog code.

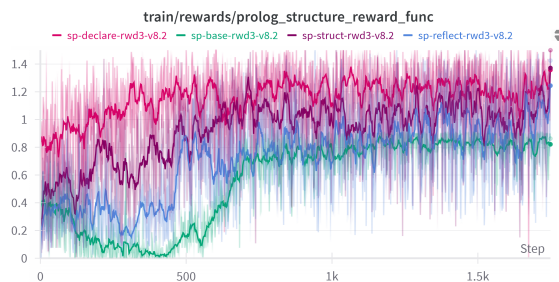


Figure 6: Prolog structure reward progression for each prompt variant in Reward Suite 3. As scheduled weights shift toward structural fidelity, models like `sp-declare` and `sp-struct` learn to emit programs with diverse predicates and logically well-formed constraint blocks. `sp-base` exhibits a delayed but eventual increase, following the sigmoid schedule’s shift from format to structure.

Figure 6 shows how structural competence emerges as a function of the curriculum weights.

C Agentic Prompts

C.1 System Prompt for Agentic Internal and Agentic Independent

```
You have one tool:
<tools>
{"name": "run_prolog", "arguments":
 [{"code": "string"}]}
</tools>
- Use the "run_prolog" tool to execute
your answer in the <answer> section.
```

C.2 Agentic Internal Context Feedback

The code failed to produce a numeric result.

Let’s fix it:

1. Reflect on what went wrong.
2. Recalculate
3. Adjust your answer to:

```
<answer>
:- use_module(library(clpqq)).
solve(X) :-
{X = final_number}.
</answer>
...
```

D Inference Examples

D.1 SP-Struct-RWD1 Example

```
<reasoning>
```

To determine how many more cakes Louise needs to bake, we can break down the problem step by step.

1. Louise needs a total of 60 cakes.
2. She has already baked half of 60 cakes, which is 30 cakes.
3. She has $60 - 30 = 30$ cakes left to bake.
4. Today, she bakes half of the remaining 30 cakes, which is $30 / 2 = 15$ cakes.
5. After baking 15 cakes, she has $30 - 15 = 15$ cakes left to bake.
6. The next day, she bakes a third of the remaining 15 cakes, which is $15 / 3 = 5$ cakes.
7. After baking 5 cakes, she has $15 - 5 = 10$ cakes left to bake.

Therefore, Louise needs to bake 10 more cakes.

```
</reasoning>
```

```
<answer>
```

```
:- use_module(library(clpqq)).
```

```
solve(C) :-
```

```
C = 10.
```

```
</answer>
```

D.2 SP-Declare-RWD1 Example

```
<reasoning>
```

- Walter works 5 days a week in a fast-food chain and earns \$5 per hour.
- He works 4 hours a day, so his daily earning is $4 * \$5 = \20 .
- He works 5 days a week, so his weekly earning is $5 * \$20 = \100 .
- Walter allocates 3/4 of his weekly earning for his schooling.
- We need to find 3/4 of \$100.

```
</reasoning>
```

```
<answer>
```

```
:- use_module(library(clpqq)).
```

```
working_days(Walter, 5).
```

```
hours_per_day(Walter, 4).
```

```
hourly_wage(Walter, 5).
```

```
schooling_fraction(Walter, 3/4).
```

```
solve(School_allocation) :-  
working_days(Walter, Days),  
hours_per_day(Walter, Hours),  
hourly_wage(Walter, Wage),  
schooling_fraction(Walter, Fraction),  
{School_allocation = Fraction * (Days *  
Hours * Wage)}.  
</answer>
```

D.3 SP-Struct-RWD1-Agentic-Internal Example

```
- TURN 1 -  
(USER) <reasoning>  
To find out how many bunnies Michael has,  
we can first calculate the number of  
dogs and cats he has, and then subtract  
that total from the total number of pets.  
  
1. Calculate the number of dogs:  
25% of 36 pets.  
2. Calculate the number of cats: 50% of  
36 pets.  
3. Subtract the sum of dogs and cats  
from the total number of pets to find  
the number of bunnies.
```

Let's denote the total number of
pets as 36. The number of dogs is $0.25 * 36$,
and the number of cats is $0.50 * 36$.
The number of bunnies is then $36 - (0.25 * 36 + 0.50 * 36)$.

```
<answer>  
:- use_module(library(clpqq)).  
  
number_of_dogs(Pets, Dogs) :-  
Dogs is 0.25 * Pets.  
  
number_of_cats(Pets, Cats) :-  
Cats is 0.50 * Pets.  
  
number_of_bunnies(Pets, Dogs, Cats,  
Bunnies) :-  
Bunnies is Pets - (Dogs + Cats).  
  
number_of_bunnies(36, Dogs, Cats,  
Bunnies) :-  
number_of_dogs(36, Dogs),  
number_of_cats(36, Cats),  
number_of_bunnies(36, Dogs, Cats,  
Bunnies).  
  
solve(X) :-  
number_of_bunnies(36, Dogs, Cats,  
Bunnies),  
{X = Bunnies}.  
</answer>
```

```
[TOKENS:post-gen] used=622 (32.9%)  
| rem=1268 | budget=1890  
>> WARNING: Potential infinite recursion  
detected in Prolog code  
>> run_prolog→ None  
[TOKENS:pre-feedback] used=622 (32.9%)  
| rem=1268 | budget=1890  
  
>> FEEDBACK INJECTED:
```

The code failed to produce a numeric
result.

Let's fix it:

1. Reflect on what went wrong.
2. Recalculate
3. Adjust your answer to:

```
<answer>  
:- use_module(library(clpqq)).
```

```
solve(X) :-  
{X = final_number}.  
</answer>
```

```
<tool_call>{  
"name": "run_prolog",  
"arguments": {  
"code": ":- use_module(library(clpqq)).\n\nsolve(X)  
:- {X = final_number}."  
}  
}
```

```
[TOKENS:post-feedback] used=731 (38.7%)  
| rem=1159 | budget=1890  
[TOKENS:pre-gen] used=731 (38.7%) |  
rem=1159 | budget=1890
```

- TURN 2 -

```
(ASSISTANT) <reasoning>  
The previous code had a logical error in  
the definition of the helper predicates.  
The helper predicates should directly  
compute the number of dogs, cats, and  
bunnies without needing to be called  
separately. We can simplify the process  
by directly calculating the number of  
dogs and cats and then subtracting from  
the total number of pets to find the  
number of bunnies.
```

1. Calculate the number of dogs:
25% of 36 pets.
2. Calculate the number of cats: 50% of
36 pets.
3. Subtract the sum of dogs and cats
from the total number of pets to find
the number of bunnies.

Let's directly compute the number
of bunnies as a single step.

```
<answer>  
:- use_module(library(clpqq)).
```

```
solve(X) :-  
{X = 36 - (0.25 * 36 + 0.50 * 36)}.  
</answer>
```

```
[TOKENS:post-gen] used=926 (49.0%)  
| rem=964 | budget=1890  
>> run_prolog→ 9
```

D.4 SP-Struct-RWD1 Agentic-Independent Example

QUESTION 54: Please generate a piece
of Prolog code to solve the given math

problem.
 Jenny wants to know whether bone meal or cow manure makes better fertilizer. The control plant with no fertilizer grows 36 inches high, the plant with bone meal grows to 125% of the height of the control plant, and the cow manure plant grows to 200% of the height of the bone meal plant. How tall is the cow manure plant in inches?

```

>> INDEPENDENT AGENTIC_TRY #1
[TOKENS:pre-gen] used=332 (17.6%) |
rem=1558 | budget=1890

```

```

- TURN: 1/20 -
(Assume the height of the plant with
bone meal is 125% of 36 inches, and the
height of the cow manure plant is 200%
of the height of the bone meal plant.)
(USER) <reasoning>
To determine the height of the cow
manure plant, we need to follow these
steps:

```

1. Calculate the height of the plant with bone meal, which is 125% of the height of the control plant.
2. Calculate the height of the cow manure plant, which is 200% of the height of the bone meal plant.

The height of the control plant is 36 inches.

```

1. Height of the bone meal plant
= 125% of 36 inches = 1.25 * 36 inches.
2. Height of the cow manure plant = 200%
of the height of the bone meal plant =
2.00 * (1.25 * 36 inches).
</reasoning>
<answer>
:- use_module(library(clpq)).

```

```
height_control_plant(36).
```

```
height_bone_meal(B) :-
B is 1.25 * height_control_plant(36).
```

```
height_cow_manure(C) :-
C is 2.00 * height_bone_meal(B).
```

```
solve(C) :-
height_cow_manure(C).
</answer>

```

```

[TOKENS:post-gen] used=653 (34.6%)
| rem=1237 | budget=1890
>> WARNING: Potential infinite recursion
detected in Prolog code
>> run_prolog→ None
[TOKENS:pre-feedback] used=653 (34.6%)
| rem=1237 | budget=1890

```

```

>> FEEDBACK INJECTED:
The code failed to produce a numeric
result.

```

Let's fix it:

1. Reflect briefly on what went wrong.
 2. Recalculate
 3. Adjust your answer to:
- ```

<answer>
:- use_module(library(clpq)).

```

```

solve(X) :-
{X = final_number}.
</answer>

```

```

<tool_call>{
"name": "run_prolog",
"arguments": {
"code": ":- use_module(library(clpq)).\n\nsolve(X)
:- {X = final_number}."
}
}

```

```

[TOKENS:post-feedback] used=763 (40.4%)
| rem=1127 | budget=1890
[TOKENS:pre-gen] used=763 (40.4%) |
rem=1127 | budget=1890

```

```
- TURN: 2/20 -
```

```

(ASSISTANT) <reasoning>
Upon reviewing the previous code,
it appears that the issue might
be related to the way the height
of the bone meal plant and the cow
manure plant are calculated and unified.

```

1. Calculate the height of the bone meal plant:  
 - Height of the bone meal plant = 1.25 \* 36 inches.
2. Calculate the height of the cow manure plant:  
 - Height of the cow manure plant = 2.00 \* (1.25 \* 36 inches).

To ensure the final height is correctly computed and unified, we need to make sure the intermediate steps are correctly defined and the final height is directly computed and unified in one step.

```

Let's redefine the solve predicate
to directly compute the height of the
cow manure plant and unify it with the
variable X.
</reasoning>
<answer>
:- use_module(library(clpq)).

```

```

solve(X) :-
X = 2.00 * 1.25 * 36.
</answer>

```

```

[TOKENS:post-gen] used=973 (51.5%)
| rem=917 | budget=1890
>> run_prolog→ 2.0*1.25*36
[TOKENS:pre-feedback] used=973 (51.5%)
| rem=917 | budget=1890

```

```
>> FEEDBACK INJECTED:
```

The code failed to produce a numeric result.

Let's fix it:

1. Reflect briefly on what went wrong.
  2. Recalculate
  3. Adjust your answer to:
- ```
<answer>
:- use_module(library(clpq)).
```

```
solve(X) :-
{X = final_number}.
</answer>
```

```
<tool_call>{
"name": "run_prolog",
"arguments": {
"code": ":-
use_module(library(clpq)).\n\nsolve(X)
:- {X = final_number}."
}
}
```

```
[TOKENS:post-feedback] used=1083
(57.3%) | rem=807 | budget=1890
[TOKENS:pre-gen] used=1083 (57.3%) |
rem=807 | budget=1890
```

- TURN: 3/20 -

(ASSISTANT)
(ASSISTANT) <reasoning>
Upon reviewing the previous code, it appears that the issue might be related to the way the height of the cow manure plant is directly computed and unified.

The height of the cow manure plant is $2.00 * (1.25 * 36 \text{ inches})$. We can simplify this to:

- Height of the bone meal plant = $1.25 * 36 \text{ inches}$.
- Height of the cow manure plant = $2.00 * (1.25 * 36 \text{ inches}) = 2.00 * B$, where B is the height of the bone meal plant.

To directly compute and unify the final height, we can simplify the expression and directly compute the final height in one step.

Let's redefine the solve predicate to directly compute the height of the cow manure plant and unify it with the variable X.

```
</reasoning>
<answer>
:- use_module(library(clpq)).
```

```
solve(X) :-
X = 2.00 * 1.25 * 36.
</answer>
```

```
[TOKENS:post-gen] used=1306 (69.1%) |
rem=584 | budget=1890
>> run_prolog→ 2.0*1.25*36
>> Aborting this attempt-resetting
```

```
context for a new independent try
>> Total steps used: 3/20
>> TRY RESULT → pred=None, steps=3, took
23.05s
>> Bailed - no valid numeric answer;
retrying with fresh context
```

```
>> INDEPENDENT AGENTIC_TRY #2
[TOKENS:pre-gen] used=332 (17.6%) |
rem=1558 | budget=1890
```

- TURN: 4/20 -
(Assume all growth is in whole inches.)
(USER) <reasoning>
To determine the height of the cow manure plant, we need to follow these steps:

1. The control plant grows to 36 inches.
2. The bone meal plant grows to 125% of the control plant's height.
3. The cow manure plant grows to 200% of the bone meal plant's height.

First, we calculate the height of the bone meal plant:
[Height of bone meal plant] = $36 * 1.25$

Next, we calculate the height of the cow manure plant:
[Height of cow manure plant] = [Height of bone meal plant] * 2

Finally, we need to find the height of the cow manure plant.

```
</reasoning>
<answer>
:- use_module(library(clpq)).
```

```
height_of_bone_meal_plant(Height)
:-
Height is 36 * 1.25.
```

```
height_of_cow_manure_plant(CowManureHeight)
:-
height_of_bone_meal_plant(BoneMealHeight),
CowManureHeight is BoneMealHeight * 2.
```

```
solve(CowManureHeight) :-
height_of_cow_manure_plant(CowManureHeight).
```

```
CowManureHeight = CowManureHeight.
</answer>
```

```
[TOKENS:post-gen] used=619 (32.8%)
| rem=1271 | budget=1890
>> WARNING: Potential infinite recursion
detected in Prolog code
>> run_prolog→ 90.0
>> Total steps used: 4/20
>> TRY RESULT → pred='90.0', steps=1,
took 8.58s
```

Q#54 | Pred: 90.0 | Gold: 90.0

E System Prompt for SP-Struct Zero-Shot MMLU Evaluation

```

You are a specialized Prolog code-generating
assistant.
You have one tool:

<tools>
{"name":"run_prolog",
 "arguments":[{"code":"string"}]}
</tools>

Your task is to choose the correct option index
for a multiple-choice question, and present
your work in two clearly defined sections:

1. <reasoning>
- Provide a clear, concise step-by-step
  explanation of how you determine which option
  is correct.
- Refer to the correct option by its zero-based
  index.

2. <answer>
- Provide executable Prolog code using
  constraint logic programming to compute the
  index of the correct choice.
- Always start with:
  ':- use_module(library(clpqc)).'
- Final answer should be unified in solve(X)
  using a single curly-brace constraint that sets
  X to the chosen index.

Use this XML format strictly:
<reasoning>
(Your step-by-step reasoning here)
</reasoning>
<answer>
:- use_module(library(clpqc)).

solve(X) :-
{X = correct_index}.
</answer>

- Use the "run_prolog" tool to execute
  your answer in the <answer> section.

```

F Hyperparameter Tuning

Because sp-struct-rwd1 achieved the highest validation accuracy among our tested variants, we ran Bayesian hyper-parameter optimisation to see whether it could be pushed further on the GSM8K-Prolog-Prover task.

Using a Weights & Biases sweep with "method": "bayes", the system fits a Gaussian Process surrogate model $P(y | \mathbf{x})$ that estimates validation reward y (specifically, eval/rewards/correctness_reward_func) as a function of the hyperparameter configuration \mathbf{x} :

$$\mathbf{x} = \left(\begin{array}{l} \text{learning_rate}, r, \text{lor}\alpha, \\ \text{batch_size}, \text{num_generations}, \\ \text{max_grad_norm}, \text{weight_decay} \end{array} \right)$$

We selected our search ranges based on established best practices and recent findings. First, the learning rate is sampled log-uniformly between 5×10^{-6} and 1×10^{-4} . For LoRA capacity, we restrict the rank r to $\{32,64\}$ and α to $\{64\}$, as gains beyond $r \approx 64$ diminish at the 3B-parameter scale (Hu et al., 2022). We use per-device batch sizes of 8 or 16 to balance throughput against gradient stability. The number of rollouts (num_generations) is set to either 4 or 8, since additional generations reduce policy-gradient variance (Snell et al., 2025). Gradient clipping (max_grad_norm) is drawn uniformly from $[0.1,1.0]$ to prevent exploding gradients (Pascanu et al., 2013), and weight decay is chosen from $\{0.01,0.1,0.2\}$.

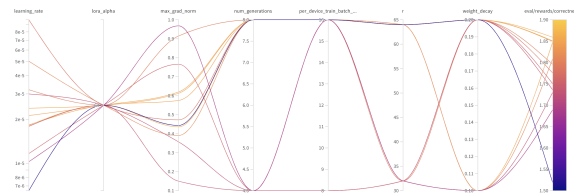


Figure 7: Parallel coordinates plot of 12 hyperparameter trials from Bayesian optimization. Each line represents one sweep, colored by validation correctness reward (eval/rewards/correctness_reward_func).

The best configuration found, denoted sp-struct-rwd1-hyper (section F), was then evaluated on the gsm8k-prolog-prover test split. Table 8 compares its accuracy to our original sp-struct-rwd1 model:

Model	Mode	Acc
SP-Struct-RWD1	Multiple-Try	89.87
SP-Struct-RWD1-Hyper	Multiple-Try	89.07

Table 8: Test accuracy on GSM8k-Prolog-Prover. Hyperparameter tuning did not surpass the original baseline.

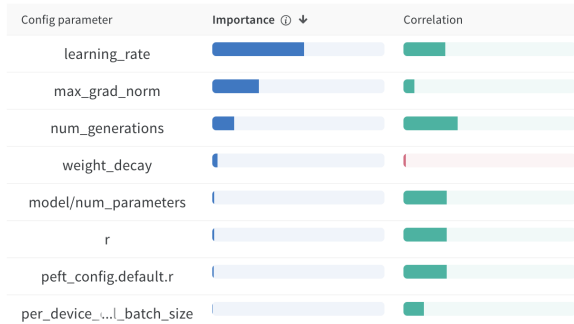


Figure 8: Bar chart of hyperparameter importances computed by W&B’s fANOVA analysis on our sweep.

Although the tuned variant fell 0.8 percentage points short of the non-tuned baseline (Table 8), fANOVA analysis (Figure 8) shows that only `learning_rate`, `max_grad_norm`, and, more weakly, `num_generations` influence the reward. The remaining variables either contribute minimally or degrade performance.

Best configuration The configuration that scored the highest in correctness for the evaluation dataset was:

```

learning_rate: 0.00001821
lora_alpha: 64
max_grad_norm: 0.4359
num_generations: 8
per_device_train_batch_size: 16
r: 64
weight_decay: 0.2
eval/rewards/correctness_reward_func: 1.859

```

G Use of AI Assistants

Claude Opus 4.5 assisted in revising the write-up and proofreading for initial submission. Opus 4.6 and Sonnet 4.6 assisted in drafting an initial script for the systematic error analysis. Opus 4.7 assisted with proofreading the camera-ready version.