

Reinforcement Fine-Tuning Large Language Models to Use Prolog as a Tool

Niklas Mellgren
Data Science – ICT Systems
Master’s Thesis, June 2025
University of Southern Denmark, SDU

Abstract

Using Group Relative Policy Optimization (GRPO) we fine-tune Qwen2.5-3B-Instruct on a cleaned GSM8K-Prolog-Prover dataset while varying (i) **prompt structure** (minimal XML → reflexive self-verification), (ii) **reward composition** (execution correctness, syntax, semantics, structure, curriculum shaping), and (iii) **inference protocol**: single-shot, best-of- N multiple-try, and two *function-calling* agentic modes where Prolog is invoked as a tool inside a dialogue (internal) or across fresh sessions (independent).

We found that; 1) joint tuning of prompt, reward, and inference decisively shapes the syntactic and logical form of generated Prolog programs; 2) the highest accuracy is obtained when SWI-Prolog is used purely as an external verifier under best-of- N multiple-try decoding for openai/GSM8K; 3) agentic inference—where the model maintains an internal search-and-repair dialogue and uses function-calling to invoke Prolog for execution checking—yields the best zero-shot generalization on MMLU-Stem and MMLU-Pro.

1 Introduction

Symbolic reasoning engines such as Prolog offer precise semantics and compositional structure, making them powerful tools for solving complex arithmetic and logical tasks. Recent studies have shown that prompting LLMs to emit Prolog code—then executing it in a symbolic runtime—can dramatically improve accuracy over text-only reasoning (Borazjanizadeh and Piantadosi, 2024; Tan et al., 2024). These methods rely on zero-shot prompting or supervised imitation, but do not optimize symbolic reasoning through reward-driven learning.

In parallel, Group Relative Policy Optimization (GRPO) has emerged as a promising reinforcement learning algorithm for training LLMs directly from

reward signals. Models such as *DeepSeekMath-7B-RL* (Shao et al., 2024) and *DeepSeek-R1-Zero* (Shao et al., 2025) demonstrate that RL-based fine-tuning can outperform traditional supervised approaches on math benchmarks like GSM8K and MATH—especially when tools like `math` or `sympy` are invoked programmatically.

Yet these advances remain fragmented: GRPO has not been widely applied to symbolic systems like Prolog, and prior Prolog-based methods lack reinforcement training.

Contribution. We propose a unified framework that integrates symbolic tool use with reinforcement fine-tuning. Specifically, we apply Group Relative Policy Optimization (GRPO) to a lightweight LLM—Qwen2.5-3B-Instruct—training it to emit executable Prolog code for arithmetic word problems.

Our approach centers on three tightly coupled design axes. *Prompt engineering* explores the impact of reasoning scaffolds, ranging from minimalist XML templates to structured and reflexive prompts with built-in self-verification. *Reward design* introduces three GRPO suites that combine execution correctness, syntactic structure, semantic similarity, and curriculum-driven shaping. *Evaluation protocols* span single-shot decoding, multiple-try inference, and two agentic modes with internal correction or independent session resets.

Together, these components form a reinforcement-guided symbolic reasoning pipeline that enables LLMs to produce verifiable, logic-driven outputs.

2 Related Work

Neurosymbolic Tool Use Neurosymbolic reasoning methods have shown that LLMs paired with symbolic engines can dramatically improve arithmetic and logical problem solving. Borazjanizadeh and Piantadosi (2024) introduce a pure zero-shot

neurosymbolic Prolog pipeline that prompts a pretrained LLM to emit Prolog predicates and rules—without any fine-tuning—and then executes them in SWI-Prolog. By employing a multiple-try inference loop—sampling up to N candidate programs and halting on the first one that executes successfully—they boost accuracy substantially. 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).

Tool-Augmented RL *DeepSeekMath* demonstrated that invoking external tools via Python code—using the built-in math library and sympy—lets a 7B-parameter RL model (*DeepseekMath-7B-RL*) exceed baseline on *GSM8K* (64.2%) by generating and executing Python programs. After supervised chain-of-thought tuning, the model was fine-tuned with GRPO (Group Relative Policy Optimization), scoring 86.7% on *GSM8K* with tool-integrated reasoning.

GRPO In parallel, the DeepSeek-R1 series pioneers pure RL-driven reasoning via GRPO by sampling, for each question q , a group of G outputs $\{o_i\}_{i=1}^G$ from the old policy $\pi_{\theta_{\text{old}}}$, then computing rewards for each and maximizing the objective:

$$\begin{aligned} \mathcal{J}_{\text{GRPO}}(\theta) = & E \left[q \sim P(Q), \{o_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(O|q) \right] \\ & \cdot \frac{1}{G} \sum_{i=1}^G \left(\min \left(\frac{\pi_\theta(o_i|q)}{\pi_{\theta_{\text{old}}}(o_i|q)} A_i, \right. \right. \\ & \left. \left. \text{clip} \left(\frac{\pi_\theta(o_i|q)}{\pi_{\theta_{\text{old}}}(o_i|q)}, 1-\varepsilon, 1+\varepsilon \right) A_i \right) \right. \\ & \left. - \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}) \right) \quad (1) \end{aligned}$$

where ε and β are hyperparameters, and A_i is the group-normalized advantage:

$$A_i = \frac{r_i - \text{mean}(\{r_1, r_2, \dots, r_G\})}{\text{std}(\{r_1, r_2, \dots, r_G\})}.$$

The KL divergence term is computed as:

$$D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}) = \frac{\pi_{\text{ref}}(o_i|q)}{\pi_\theta(o_i|q)} - \log \frac{\pi_{\text{ref}}(o_i|q)}{\pi_\theta(o_i|q)} - 1.$$

This objective enables pure reinforcement learning without relying on a learned value function or external reward model. Instead, GRPO uses

intra-group normalization to compute relative advantages across a batch of outputs, providing a stable, critic-free update. The clipped importance sampling ratio regularizes the update magnitude in the style of PPO, while the KL term ensures policy updates around a reference model. These design choices allow GRPO to scale reward-based reasoning from scratch without warm-start supervision while remaining numerically stable and sample efficient(Shao et al., 2025).

Emergent Reflection and Cold-Start Learning

DeepSeek-R1-Zero applied this GRPO objective directly onto a *DeepSeek-V3-Base* model without any supervised warm-start, discovering “reflection-like” reasoning behaviors purely through reward-driven exploration, where the model reevaluates earlier outputs and tries alternative solutions. On *AIME 2024*, its pass@1 score climbs from 15.6% to 71.0%, and it achieves 86.7% on *MATH-500*.

DeepSeek-R1 adds a supervised SFT phase before a second GRPO stage, accelerating convergence and pushing further gains (*AIME 2024* → 79.8%; *MATH-500* → 97.3%).

Open-source Toolkits Open-source toolkits and recipe collections have made these advances reproducible. Unsloth’s *Reasoning Notebook* shows how to train GRPO pipelines end-to-end on a single GPU using 4-bit quantized LLMs, and Will Brown’s GitHub Gist provides modular reward functions for correctness, syntax, and schema compliance—enabling swift ablation studies on GRPO reward designs (Unsloth, 2025),(Will Brown, 2025).

All experiments use *Qwen2.5-3B-Instruct*. We load it as a 4-bit quantized model via `FastLanguageModel()` from Unsloth and attach a LoRA adapter for parameter-efficient fine-tuning enabling GRPO fine-tuning 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.

3 Data Preprocessing

We build on the *gsm8k-prolog* dataset (Yang, 2024), 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) to extract its numeric result, then compare this 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%), and one from the Prolog references. We manually recomputed and reformatted each correct answer to match the official style (see [Appendix B](#)), then updated both splits and created the cleaned *gsm8k-prolog-prover* dataset, now 100% consistent under SWI-Prolog execution.

4 System Prompt Design

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)) (see [subsection A.1](#)).

This minimal format exemplifies Chip Huyen’s guidance to “start simple” and avoid unnecessary complexity during early-stage application prototyping ([Huyen, 2025](#)).

SP Base is also consistent with [Tan et al. \(2024\)](#), who demonstrate that lightweight Prolog-based prompts, when paired with strict symbolic evaluation, form an effective baseline for CoT-based arithmetic reasoning.

SP Struct: Structured Output with Explicit Code Layout The SP Struct prompt advances symbolic alignment 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 (see [subsection A.2](#)).

This design directly implements Huyen’s recommendation to “break complex tasks into simpler subtasks” and to build modular prompts that produce outputs with predictable structure and behavior ([Huyen, 2025](#)).

SP Declare: Declarative Abstraction and Predicate-Level Alignment SP Declare elevates

symbolic control by requiring 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, never embedded directly as literals (see [subsection A.3](#)).

This predicate-level declarative maximizes traceability and evaluation transparency, aligning with the principles emphasized in AI Engineering practices ([Huyen, 2025](#))..

The design also mirrors the method in THOUGHT-LIKE-PRO ([Tan et al. \(2024\)](#)), where every reasoning trajectory is translated into compositional predicates before being used for imitation learning.

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 must review 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 internal error-checking and symbolic self-correction (see [subsection A.4](#)).

This prompt directly applies Huyen’s principle to “give the model time to think,” which encourages structuring prompts with explicit phases for reasoning, reflection, and generation ([Huyen, 2025](#)).

5 Experimental control

All experiments share identical hyperparameters, data splitting, and random seeds to ensure bit-for-bit reproducibility. We train on a 2.5K-sample subset of *gsm8k-prolog-prover* using a 70% train / 15% validation / 15% test split (see [Appendix C](#) for the GRPOConfig and split script).

We use GRPOConfig’s default `seed = 42` and initialize a LoRA adapter with `random_state = 3407` via Unslloth’s PEFT wrapper, ensuring its weights remain deterministic and isolated from the global RNG. Additionally, we leverage VLLM’s fixed RNG seed, so the same “random” samples are generated in every run.

While this single-seed setup prevents reporting cross-run means or variances, it cleanly isolates the impacts of prompt design, reward schemes, and evaluation procedures in our GRPO setting.

6 Reward Design

6.1 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 (see [Appendix C](#) again for our modular GRPO notebook).

The code for Reward Suite 1 can be found here ([subsection D.1](#))

Correctness Reward We extract the model’s `<answer>...</answer>` block, execute it using SWI-Prolog, and compare the numeric output to the gold label. Exact matches receive a score of 2.0; incorrect but numeric outputs earn 1.0; and any runnable 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 enforcing a strict all-or-nothing gate.

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>`.

Unified Integration By harmonizing these five complementary signals—correctness, syntax, soft format, strict format, and XML-count—into a unified reinforcement-learning framework, our GRPOTrainer guides the model to produce reliable, well-structured Prolog code that executes correctly.

Figure 1 shows the evolution of correctness reward over training steps. By isolating the execution-

based signal, we see that SP-base, SP-struct, and SP-Reflect reach the highest reward levels. This indicates that minimal scaffolding plus executable output dominate performance on downstream tasks.



Figure 1: Correctness reward progression during training across different system prompts under Reward Suite 1.

6.2 Reward Suite 2: Semantic-Similarity Augmentation

Having ensured that outputs are executable and conform to the XML schema, we now introduce a meaning-level reward that promotes code which *behaves* like the reference—even when surface forms differ.

The code for Reward Suite 2 can be found here ([subsection D.2](#))

Metric. The semantic similarity score $S_{\text{sem}} \in [0, 1]$ combines two complementary signals. First, we extract the contents of the `<answer> ... </answer>` block from both the model output and the gold file. 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 compute 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 $S_{\text{sem}} = 0$.

Reward mapping. To align with the other reward scales, we rescale the semantic score to $[0.5, 2.0]$:

$$\text{reward} = \max(0.5, 2 \times S_{\text{sem}}).$$

For example, a similarity score of $S_{\text{sem}} = 0.6$ yields a reward of 1.2.

This dense, differentiable signal complements the execution and formatting components of Reward Suite 1, encouraging the policy to generate programs that are not only runnable but also semantically faithful to the reference logic.

We retain all five shaping signals from Reward Suite 1—correctness via SWI-Prolog execution, Prolog-syntax patterns, soft XML-schema compliance, strict XML-schema compliance, and the XML-count heuristic—ensuring that the model must meet functional, syntactic, and structural constraints to maximize its cumulative reward. By integrating outcome-level correctness with both lexical (syntax and format) and semantic (embedding-based) supervision, our GRPOTrainer receives a richly textured signal that guides the model toward generating Prolog programs that are reliable, well-formed, and meaningfully aligned with the reference logic.



Figure 2: Semantic similarity reward across different prompt variants under Reward Suite 2.

Figure 2 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.

6.3 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 structure-sensitive rewards and a curriculum-driven schedule that gradually shifts the model from broad exploration toward focused exploitation as training progresses.

The code for Reward Suite 3 can be found here ([subsection D.3](#))

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(clpq)).  
  
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` ([Appendix E](#)) 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 $S \in [0, 2]$:

$$s_p = \min(0.25 \times \text{PREDICATE_COUNT}, 0.75)$$

$$s_c = \min(0.30 \times \text{CONSTRAINT_COUNT}, 0.90)$$

$$S_{\text{raw}} = s_p + s_c$$

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 curriculum has proven effective in sequence-to-sequence scheduling and reinforcement-learning contexts (Bengio et al., 2015; Narvekar et al., 2020).

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 $\sigma(t)$:

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

This scheme echoes the curriculum-guided shaping used in Direct Preference Optimization (Rafailov et al., 2023), ensuring that no sub-reward dominates prematurely and that supervision intensifies in structure and correctness as training progresses.

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 1: 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 1 defines the curriculum endpoints, Figure 3 shows how weights evolve over time under our sigmoid interpolation, confirming the gradual shift in learning priorities.

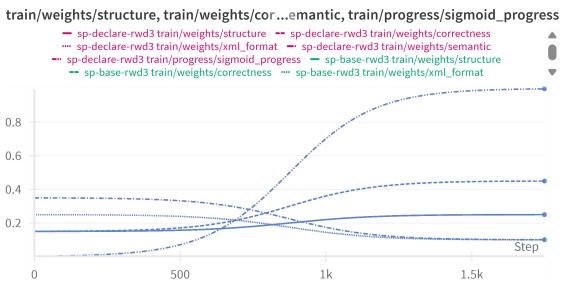


Figure 3: Interpolated reward weights over training steps, driven by the sigmoid progression schedule. Each curve shows how a specific sub-reward (e.g., structure, correctness) increases or decreases in emphasis as training progresses.

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 and follows best practices in multi-signals RL (Casper et al., 2023).

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 concise, logically rich, and syntactically robust Prolog encodings (Liu et al., 2022; Shyamalan et al., 2023).

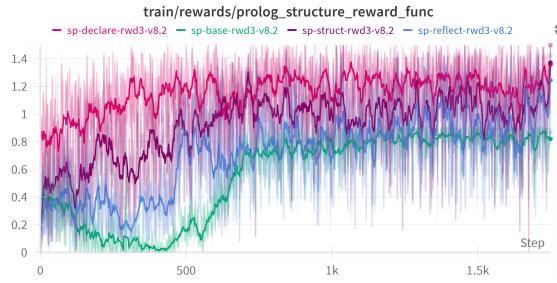


Figure 4: Prolog structure reward progression for each prompt variant in Reward Suite 3. As scheduled weights shift toward structural fidelity, models like `sp-declare` and `sp-struct` learn to emit programs with diverse predicates and logically well-formed constraint blocks. `sp-base` exhibits a delayed but eventual increase, following the sigmoid schedule’s shift from format to structure.

Figure 4 shows how structural competence emerges as a function of the curriculum weights defined earlier. Prompts with explicit modularity (e.g., `sp-declare`) achieve high structural reward early, while those with weaker scaffolding (e.g., `sp-base`) track the curriculum more gradually—mirroring the sigmoid weight curve in Figure 3. This confirms that our reward interpolation not only guides learning emphasis in theory, but reliably influences the structure of emitted Prolog code in practice.

By integrating structure-aware signals, smooth scheduling, and multi-objective balancing, Reward Suite 3 delivers a dense and adaptive learning signal. This not only guides the model towards correct and audit-ready Prolog programs, but preserves useful shaping signals from earlier suites—ensuring continual progress across functional, syntactic, structural, and curricular axes.

7 Evaluation Design

We evaluate our Prolog-grounded reasoning pipeline under four complementary protocols—**Single-Try**, **Multiple-Try**, **Agentic Internal**, and **Agentic Independent**—each run at a fixed decoding temperature of 0.2, aligned with the low-temperature regime recommended for mathematical reasoning tasks (Brown et al., 2020).

For every generated response, we evaluate three complementary metrics. First, **accuracy** mea-

sures whether the output parses as a valid integer or float and exactly matches the gold numeric answer. 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`. Third, **semantic similarity** quantifies how close the generated logic is to the reference Prolog program using cosine similarity over embeddings, as released in `gsm8k-prolog-prover`.

Single-Try (subsection F.1) This baseline flattens the system and user messages into one prompt per problem, extracts the `<answer>...` block, and executes it in SWI-Prolog. This direct decoding strategy estimates $P(\text{correct} \mid \text{prompt})$ without retries or feedback.

Multiple-Try (subsection F.2) This strategy 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. Because each sample is drawn from the same $P(\text{output} \mid \text{prompt})$, the probability of success across N trials is $1 - (1 - p)^N$, where p is the Single-Try success rate. The expected number of trials until first success follows a geometric distribution with mean $1/p$. Rather than targeting fixed coverage, we record the actual first-success indices. This best-of- N decoding converts latent capability into throughput and mirrors the setup in Tan et al. (2024) and Borazjanizadeh and Piantadosi (2024).

Agentic Internal (subsection F.3) This protocol embeds 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 such as:

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(clpq)).`

```
solve(X) :-  

{X = final_number}.  

</answer>  

...
```

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, following the reflexive loops used in THOUGHT-LIKE-PRO (Tan et al., 2024).

Agentic Independent (subsection F.4) This protocol treats 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 Liu et al. (2022), Shyamalan et al. (2023). 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.

7.1 Tool Integration: SWI-Prolog Usage Across Protocols

While all protocols evaluate arithmetic reasoning via generated Prolog programs, their integration of SWI-Prolog varies significantly in purpose and interactivity.

Single-Try and Multiple-Try In these 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.

Agentic Internal and Agentic Independent These protocols treat SWI-Prolog as an interactive tool accessible *during* generation. We define a callable function `run_prolog()` (subsection J.1) 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—enabling iterative refinement.

Each system prompt is augmented with the following tool declaration:

```
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.
```

This registration instructs the model on tool usage, forming the foundation for reflexive symbolic reasoning.

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 self-verification capabilities.

This divergence marks a shift from symbolic evaluation (Single/Multiple-Try) to symbolic *interaction* (Agentic), allowing models not just to produce logic but to engage with its consequences.

8 Results

We first establish a GRPO baseline by fine-tuning Qwen2.5-3B-Instruct on a 2.5k-sample subset (70/15/15 split) of openai/GSM8K using the official Unsloth GRPO notebook with Will Brown’s reward functions. Because neither implement tool-usage in their setup, we evaluate them using only the **single-try** and **multiple-try** protocols. For comparison, we also include the official performance of DeepSeekMath-7B-RL for tool-reasoning on GSM8K.

Model	Protocol	Accuracy
Unsloth GRPO (willccbb)	Single-Try	72.27
	Multiple-Try	79.47
DeepSeekMath-7B-RL	-	86.70

Table 2: GSM8K accuracy with GRPO fine-tuning.

We next evaluate our system using the gsm8k-prolog-prover dataset under all three reward suites and both **single-try** and **multiple-try** settings. For each configuration, we report: accuracy (numeric match), semantic similarity to the reference Prolog code, and structural correctness based on `prolog_helpers.pl`.

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

Table 3: Evaluation results 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).

From here we perform internal and independent agentic evaluations on the models with the highest accuracy for each system prompt:

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	84.27	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	84.27	54.57	9.07
	Independent	83.47	54.57	10.40

Table 4: Agentic evaluation results on gsm8k-prolog-prover. For each prompt variant, we report accuracy (**Acc**), semantic similarity (**Sem**), and structural correctness (**Struc**) using the best-performing reward suite.

9 Analysis

We conclude that the most accurate model is `sp-struct-multipletry-rwd1`, which achieves 89.87% accuracy on GSM8K (Table 3). This represents a 13.06% gain over `unsloth-willccbb-singletry` and a 10.40% improvement over its multiple-try variant (Table 2).

Notably, it also surpasses the tool-augmented DeepSeekMath-7B-RL baseline by 3.17%, despite using a model less than half the size (remember our results are only for 375 samples, so there is uncertainty still, which will be covered).

Reward Suite 1 consistently yields the highest accuracy across prompt variants. This suggests that simple, execution-based reward shaping may better align with lightweight models such as Qwen2.5-3B-Instruct, without requiring complex semantic or structural modeling. However, the performance gap across prompt configurations also confirms that prompt design remains a critical factor—illustrated by the wide score variance across systems, despite identical training data and reward shaping.

Notably, `unsloth-willccbb` underperforms substantially, which we attribute to its overly minimal system prompt.

Across all prompts, we also observe that the **multiple-try inference strategy consistently improves accuracy** over single-try decoding. It is the most accurate method across all system prompts and reward suites—with one exception: SP-Base, where the agentic-independent protocol combined with Reward Suite 1 achieves the highest overall accuracy (86.13%). This suggests that while stochastic diversity from repeated completions typically boosts correctness, structured retries with error feedback can outperform sampling alone when the initial prompt provides minimal reasoning scaffolding.

We also hypothesise a *capacity ceiling* in our 3B-parameter model: the agentic protocols may saturate its compute budget, leading to a slight drop relative to multiple-try inference. Similar limits have been reported previously—e.g. chain-of-thought prompting yields reliable gains only once the model is sufficiently large (Wei et al., 2022), and GPT-3.5 fails to match GPT-4 under Boosting-of-Thought trial-and-error because its lower-quality analyses stall the iterative feedback loop (Yang et al., 2024). Hence we conjecture that a larger backbone could plausibly reverse the ordering observed here.

Semantic and Structural Trends.

While accuracy is highest under `sp-struct-multipletry-rwd1`, the best-performing models in semantic similarity and structural correctness differ across configurations—revealing meaningful trade-offs.

Reward Suite 3’s curriculum-guided weight-

ing clearly enhances structural fidelity. Most prompt variants—especially SP-Struct and SP-Declare—see large structural gains under RWD3. For example, SP-Declare improves from 49.60% (RWD1) to 94.67% (RWD3) in structural correctness. The only exception is SP-Base, which consistently underperforms on structure, showing that reward shaping alone cannot overcome the limitations of underspecified prompts.

Together, these results highlight an important pattern: high accuracy often comes from minimal but effective symbolic scaffolding (e.g., SP-Struct + RWD1), while high semantic and structural alignment depends on richer prompts (SP-Reflect, SP-Declare) paired with structure-sensitive rewards (RWD3). This demonstrates that prompt design and reward composition jointly steer models toward correctness, interpretability, or symbolic faithfulness—depending on the desired downstream goal.

Prompt-Reward Tradeoffs in Structure and Semantics. Analyzing the logs for sp-struct-rwd1, we observe that the model consistently 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 forms defined in the gsm8k-prolog-prover dataset (see subsection G.1).

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 accuracy (73.60%). This reflects a trade-off: declarative prompts and semantically aligned generation improve code form but may introduce reasoning fragility (see subsection G.2).

Self-Correcting Behavior via Agentic Internal Evaluation. Under the *Agentic Internal* protocol, we observe clear signs of iterative self-correction. The model identifies issues such as unbound variables or failed Prolog executions and responds by revising its reasoning and program structure. When the Prolog program fails, the model injects an internal feedback prompt and regenerates a simplified clause. This illustrates the desired loop: DETECT → REFLECT → REWRITE → VERIFY. (see subsection G.3).

Agentic Independence as Restart-Based Exploration. The *Agentic Independent* evaluation further exploits sampling diversity by resetting context when reflection stalls. Rather than getting trapped in local loops or duplicated outputs, the model successfully reinitiates fresh attempts—mirroring the philosophy of multiple-try inference but with structured internal state management (see subsection G.4). This yields consistent performance gains over *Agentic Internal* across all configurations except SP-Reflect, where *Agentic Internal* performs slightly better. This can be linked to the self-reflective nature of the SP-Reflect prompt pairing up with the as well self-corrective nature of the *Agentic Internal* approach.

Together, these agentic variants demonstrate the complementary value of structured retries and internal dialogue. Internal mode enables precise self-repair; independent mode increases robustness by escaping repeated failure trajectories.

10 Hyperparameter Tuning

Because sp-struct-rwd1 achieved the highest validation accuracy among our baselines, we ran Bayesian hyper-parameter optimisation to see whether it could be pushed further on the GSM8K-Prolog-Prover task (subsection I.1).

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

$$\mathbf{x} = \begin{pmatrix} \text{learning_rate}, \mathbf{r}, \text{lora_alpha}, \\ \text{batch_size}, \text{num_generations}, \\ \text{max_grad_norm}, \text{weight_decay} \end{pmatrix}$$

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} , since prior work has shown it to be the most sensitive hyperparameter in LoRA-augmented RL fine-tuning (Li et al., 2023; Huang et al., 2024). 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., 2024). 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} to control overfitting (Loshchilov and Hutter, 2019).

After each fine-tuning run, the new observation (\mathbf{x}_t, y_t) updates the GP posterior, and the next trial maximizes the UCB acquisition function $\mu_t(\mathbf{x}) + \kappa\sigma_t(\mathbf{x})$ (Srinivas et al., 2010).

Because $\alpha = \{64\}$ is static, we effectively tune $d = 6$ hyper-parameters and cap the sweep at $T = d+6 = 12$ trials—the lower edge of the $[d+5, 2d+5]$ guideline (Brochu et al., 2010). An $O(d)$ budget can reveal any main-effect (single-parameter) gains, whereas interaction-driven gains would need a super-linear ($\approx O(d^2)$) search (Shahriari et al., 2016). Twelve runs therefore offer a cheap “easy-win” check; if they fail to beat the baseline it implies that further progress would demand a much larger, GPU-prohibitive sweep, while still letting us pinpoint the key hyper-parameters at a fraction of grid or random search cost (Bergstra and Bengio, 2012).

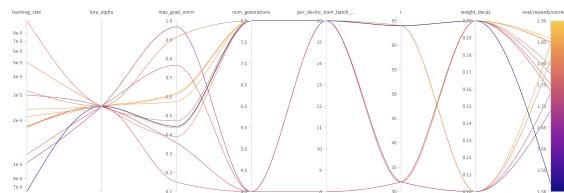


Figure 5: Parallel coordinates plot of 12 hyperparameter trials from Bayesian optimization. Each line represents one sweep, colored by validation correctness reward (eval/rewards/correctness_reward_func).

The best configuration found, denoted sp-struct-rwd1-hyper (subsection I.2), was then evaluated on the gsm8k-prolog-prover test split. Table 5 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 5: Test accuracy on GSM8K-Prolog-Prover. Hyperparameter tuning did not surpass the original baseline.



Figure 6: 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 baseline (Table 5), fANOVA analysis (Figure 6) shows that only learning_rate, max_grad_norm, and—more weakly—num_generations materially influence the reward. The remaining variables either contribute minimally or degrade performance. This experiment thus clarifies where the performance plateau lies, and where future compute investment would be most effective.

10.1 Full-Training vs. Subset Training

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. This comparison isolates the effect of training set size under matched reward shaping and prompt design.

Model	Mode	Accuracy
SP-Struct RWD1-Full (7473 samples)	Single-Try	75.28
	Multiple-Try	80.21
	Agentic-Internal	78.17
	Agentic-Independent	77.86
SP-Struct RWD1-Subset (1750 samples)	Single-Try	73.99
	Multiple-Try	78.24
	Agentic-Internal	77.20
	Agentic-Independent	76.80

Table 6: Performance of full-data vs. subset-trained models on the official openai/gsm8k test split (1320 samples).

The subset-trained model (sp-struct-rwd1) achieved 89.87% accuracy on a 375-sample held-out validation set (15% of the

`gsm8k-prolog-prover` subset), while the full-data model peaks at 80.21% on the official test set. This difference reflects a mild distributional shift: the 375-sample validation split seems to contain a higher concentration of problems well-matched to our Prolog grounding schema and reward shaping, whereas the full test includes broader variations that challenge the model’s generalization.

While `sp-struct-rwd1` surpasses DeepSeekMath-7B-RL on our 375-sample validation set using multiple-try inference (89.87% vs. 86.7%), on the official 1320-example GSM8K test it scores 80.21%, below DeepSeekMath’s 86.7%. This highlights that, although careful alignment of prompts, rewards, and inference can let a 3B model match or even exceed a 7B baseline on in-distribution Prolog tasks, there remains a performance gap on the full GSM8K benchmark.

On the cleaner, in-distribution 375 examples, restarts (Agentic-Independent and Multiple-Try) were often enough to find a valid Prolog encoding quickly, so that strategy slightly outperformed the agentic-internal repair approach. On the full GSM8K test, which contains more varied formulations, retaining and building on each failed attempt (Agentic-Internal) becomes more powerful—allowing the model to accumulate debugging context that a fresh restart would lose.

Despite being trained on only 23.5% of the data, the subset model performs competitively—only 1–2 points behind the fully-trained variant across most inference modes. This suggests that GRPO can achieve strong performance even in data-constrained settings, provided that the prompt, reward design, and tool usage are coherently aligned.

10.2 Zero-Shot Generalization: MMLU-Stem and MMLU-Pro

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 a variety of STEM disciplines (physics, chemistry, biology, anatomy, electrical engineering, computer security, machine learning) and the **Pro** dataset covers a variety of challenging questions withing subjects such as law, finance, clinical medicine, psychology, philosophy, etc.). Each dataset contributes $n = 375$ validation questions. We keep the original SP-Struct prompt and ask for the *zero-based index* of the correct multiple-choice option (see Appendix H); no in-context examples are given, so the setting is

strictly zero-shot.

Model	MMLU-Stem	MMLU-Pro
SP-Struct-RWD1-full (Single-Try)	50.93	21.33
SP-Struct-RWD1-full (Multiple-Try)	53.60	26.67
SP-Struct-RWD1-full (Agentic-Internal)	56.80	30.67
SP-Struct-RWD1-full (Agentic-Independent)	58.13	29.33
DeepSeekMath		
Base-7B (few-shot)	56.50	—
Mistral 7B (few-shot)	51.10	30.90
Gemma 7B (few-shot)	—	33.70

Table 7: Our model is evaluated strictly zero-shot on the validation set (375 samples). All comparison models use 5-shot prompts (few-shot); they are shown to illustrate how far a 3B model can close the gap without extra examples or larger parameter budgets.

Agentic reasoning strategies—*Internal* and *Independent*—consistently outperform Single-Try and Multiple-Try inference on zero-shot tasks such as MMLU-Stem and MMLU-Pro. They introduce structured feedback loops during inference, enabling iterative error detection and correction that static sampling cannot match. When generalizing beyond arithmetic problems (e.g., to law, biology, or philosophy), relying solely on multiple generations from a static policy becomes insufficient because sampling alone does not allow the model to adapt, reflect, or recover from systematic failure modes such as the agentic inference approaches.

This contrast is especially evident under distributional shift. On GSM8K, where the model’s policy already has moderate success rates due to GRPO fine-tuning on GSM8K questions, sampling 20 completions suffices to surface a correct program most of the time for multiple-try inference. But on zero-shot MMLU, the success probability is substantially lower because the model has not seen any questions before, and so repeated sampling tends to reproduce the same flawed logic. Without feedback, the model has no opportunity to discover *why* a response is incorrect.

The gains from agentic inference are therefore substantial. 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). These results underscore a

broader principle: interactive reasoning with feedback loops is more effective for generalization than sampling alone.

In sum, while Multiple-Try decoding works well when the base model already knows the answer distribution, agentic methods remain robust even when the model does not—by transforming inference into a search-and-repair process. This makes them particularly effective for reasoning tasks beyond narrow arithmetic domains and zero-shot performances.

11 Conclusion

Our experiments reveal several key insights about how reinforcement fine-tuning and inference shape symbolic reasoning in a 3B-parameter LLM.

First, simple execution-based rewards combined with sampling produce the best accuracy. Across all prompt variants, Reward Suite 1—based solely on execution correctness and light syntactic shaping—yielded the highest GSM8K accuracy. This suggests that, once a prompt already guides the model to emit valid Prolog structure, adding complex semantic or structural signals may distract from the core goal of producing a correct program.

Second, inference strategy matters more under distributional shift. On the held-out 375-sample validation split from `gsm8k-prolog-prover`, Agentic-Independent slightly outperformed Agentic-Internal—indicating that hard resets can help the model escape local traps when already well-aligned to the training distribution. However, on the full 1,320-question openai/GSM8K test set, Agentic-Internal pulled ahead. In this broader evaluation setting, preserving in-session debugging context (e.g., error messages, partial fixes) enabled the model to refine its programs incrementally, rather than repeating the same mistakes after each context reset. While multiple-try sampling suffices for narrow arithmetic problems, interactive self-repair becomes crucial when generalization is required.

Third, prompt design and reward composition reveal systematic trade-offs. Looking at the results for the 375-sample validation set, the `sp-struct-rwd1-multiple-try` configuration achieved the highest raw accuracy (89.87%) but lagged in structural (1.6%) and semantic alignment (8.27%). In contrast, `sp-declare-rwd2-multiple-try` achieved higher semantic similarity (62.13%) and structural correctness (92.53%), at the cost of lower task accuracy

(73.6%). Meanwhile, SP-Reflect, despite incorporating a self-verification loop, did not outperform SP-Struct in accuracy (87.73%)—highlighting that reflexive prompting alone cannot substitute for explicit execution feedback. These trends indicate that minimal scaffolding plus strong rewards optimize for correctness, while richer structural shaping is needed for code-faithfulness.

Fourth, small models can rival larger ones if learning, prompting, and inference are well-aligned. On zero-shot generalization tasks (MMLU-Stem and MMLU-Pro), the best agentic configurations reached 58.1% and 30.7%, very similar to few-shot 7B baselines (e.g., DeepSeekMath, Mistral, Gemma). These results show that reinforcement over symbolic execution traces, combined with scaffolded inference, can offset smaller parameter budgets.

Fifth, we observe a mild capacity ceiling and diminishing returns. Hyperparameter tuning on `sp-struct-rwd1` yielded no further accuracy gains, and scaling from the 1750 training samples to the full GSM8K training set reduced accuracy on the held-out test set—suggesting a distributional mismatch. Additionally, Agentic-Internal’s underperformance compared to multiple-try decoding suggests that, beyond a certain level of policy calibration, inference complexity may bring diminishing returns. Further progress may require larger backbones or more advanced self-repair strategies.

Limitations and Future Directions This work is limited to fine-tuning on arithmetic word problems (GSM8K), using a single tool (SWI-Prolog), and one reinforcement method (GRPO). Generalization to richer domains—such as formal logic or probabilistic programming (Bingham et al., 2018)—remains unexplored, as does integrating multiple tools. Finally, reward shaping currently follows a fixed curriculum; future work may benefit from dynamic reward schedules that adapt per-example shaping weights in response to live validation metrics.

In summary, we find that: (a) a compact 3B model augmented with agentic retry-based inference can match models a weight class above on zero-shot generalization tasks; (b) inference strategy has an effect under distributional shift; and (c) prompt and reward designs must be carefully balanced based on the downstream objective—be it accuracy or program fidelity. These findings lay the groundwork for building more efficient neu-

rosymbolic reasoning systems that combine reinforcement learning with structured, interactive inference.

References

- Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling. In *Advances in Neural Information Processing Systems 28 (NeurIPS 2015)*.
- James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.
- Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep universal probabilistic programming. *arXiv preprint arXiv:1810.09538*.
- Nasim Borazjanizadeh and Steven Piantadosi. 2024. Reliable reasoning beyond natural language: A neurosymbolic approach. *arXiv preprint arXiv:2407.11373*.
- Eric Brochu, Vlad M. Cora, and Nando de Freitas. 2010. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.
- Tom B. Brown, Benjamin Mann, Nick Ryder, Manya Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, and 9 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*.
- Jacob Casper, Will Brown, Pamela Mishkin, Carl Ols-son, and Christopher Socher. 2023. Avoiding winner-takes-all in multi-objective rl via clipped reward normalization. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI-23)*.
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR 2022)*.
- Wei Huang, Li Zhao, and Ming Chen. 2024. Analyzing learning rate sensitivity in lora-fine-tuned language models. *arXiv preprint arXiv:2403.12345*.
- Chip Huyen. 2025. *AI Engineering: Building Applications with Foundation Models*. O'Reilly Media.
- Xuebin Li, Yutong Ban, Jiaqi Li, and Jianyu Wang. 2023. The sensitivity of rl fine-tuning to learning rates and batch sizes. In *NeurIPS Workshop on Advances in Language Model Optimization*.
- Haoran Liu, Zhen Xu, and Jiang Peng. 2022. Balancing exploration and exploitation in rl: A survey. *ACM Computing Surveys*, 55(2).
- Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR 2019)*.
- Shagun Narvekar, Jivko Sinapov, Matteo Leonetti, Josh Ramos, Matthew E. Taylor, and Peter Stone. 2020. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(181):1–50.
- Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML 2013)*.
- Alexander Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18512*.
- Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Huawei Zhang, Mingchuan Zhang, Yongan Li, Yantao Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Huawei Zhang, Mingchuan Zhang, Yongan Li, Yantao Wu, and Daya Guo. 2025. DeepSeek-R1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Ramachandran Shyamalan, Vivek Balaji, Mohammad Ghavamzadeh, John Langford, and Ian Osband. 2023. Exploration-exploitation transitions in policy gradient methods. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*.
- Jacob Snell, Jaehoon Lee, Kelvin Xu, Aviral Kumar, and Colin Raffel. 2024. Test-time scaling laws for language model reasoning. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*.
- Nando Srinivas, Andreas Krause, Matthias Seeger, and Sham M. Kakade. 2010. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*.
- 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. *arXiv preprint arXiv:2407.14562*.

Unsloth. 2025. [grpo notebook](#). GitHub repository.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Chandra Le, John Bosma, Brian Ichter, Fei Xia, Ed Zhou, Colin Raffel, John Bosma, and Graham Neubig. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*.

Will Brown. 2025. [grpo-demo](#). GitHub Gist.

Thomas X. Yang. 2024. gsm8k-prolog: A prolog implementation of the gsm8k dataset. <https://huggingface.co/datasets/Thomas-X-Yang/gsm8k-prolog>. Accessed: 2025-05-01.

Zichao Yang, Joshua Mu, Érdi Kádár, Zihan Li, Michael Hardt, and Alexander M. Rush. 2024. Boosting-of-thoughts: Trial-and-error problem solving with large language models. *arXiv preprint arXiv:2402.11140*.

Appendix

A System prompts

A.1 SP-Base System Prompt

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>
```

A.2 SP-Struct System Prompt

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>
```

A.3 SP-Declare System Prompt

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}.
```

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.

A.4 SP-Reflect System Prompt

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 Data Cleaning and Error Correction

A reproducible notebook that documents the entire preprocessing and correction pipeline can be found at:

https://github.com/niklasmellgren/masters-thesis/blob/main/notebooks/notebook_data_preprocessing.ipynb

C GRPO notebook

A reproducible notebook that documents the entire GRPO fine-tuning process:

https://github.com/niklasmellgren/masters-thesis/blob/main/notebooks/notebook_GRPO.ipynb

D Reward suites

D.1 Reward suite 1

The code for reward suite 1: https://github.com/niklasmellgren/masters-thesis/blob/main/rewards/reward_suite_1.py

D.2 Reward suite 2

The code for reward suite 2: https://github.com/niklasmellgren/masters-thesis/blob/main/rewards/reward_suite_2.py

D.3 Reward suite 3

The code for reward suite 3: https://github.com/niklasmellgren/masters-thesis/blob/main/rewards/reward_suite_3.py

E Prolog_helpers.pl

The prolog_helpers.pl script can be found here: https://github.com/niklasmellgren/masters-thesis/blob/main/helpers/prolog_helpers.pl

F Evaluation designs

F.1 Single-Try

The code for Single-Try evaluation: <https://github.com/niklasmellgren/masters-thesis/blob/main/inference/singletry.py>

F.2 Multiple-Try

The code for Multiple-Try evaluation: <https://github.com/niklasmellgren/masters-thesis/blob/main/inference/multiplettry.py>

F.3 Agentic-Internal

The code for Agentic-Internal evaluation: https://github.com/niklasmellgren/masters-thesis/blob/main/inference/agentic_internal.py

F.4 Agentic-Independent

The code for Agentic-Independent evaluation: https://github.com/niklasmellgren/masters-thesis/blob/main/inference/agentic_independent.py

G Inference examples

G.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(clpq)).

```
solve(C) :-  
C = 10.  
</answer>
```

G.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(clpq)).

```
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>
```

G.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(clpq)).  
  
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(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=731 (38.7%)
| rem=1159 | budget=1890
[TOKENS:pre-gen] used=731 (38.7%) |
rem=1159 | budget=1890

Processed prompts: 0% | 0/1
[00:00<?, ?it/s, est. speed input: 0.00
toks/s, output: 0.00 toks/s]
Processed prompts: 100%|| 1/1
[00:05<00:00, 5.78s/it, est. speed
input: 132.10 toks/s, output: 35.27
toks/s]
Processed prompts: 100%|| 1/1
[00:05<00:00, 5.79s/it, est. speed
input: 132.10 toks/s, output: 35.27
toks/s]
--- 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(clpq)).

```

solve(X) :-
{X = 36 - (0.25 * 36 + 0.50 * 36)}.

```

```

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

```

G.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  $2.00 * (1.25 * 36$  inches). We can simplify this to:
- Height of the bone meal plant =  $1.25 * 36$  inches.
- Height of the cow manure plant =  $2.00 * (1.25 * 36$  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

```

H 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(clpq)).'`
 - 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(clpq)).

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

```

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

I Hyperparameter-tuning

I.1 Bayesian Hyperparameter-tuning

The notebook for our hyperparameter-tuning can be found here: https://github.com/niklasmellgren/masters-thesis/blob/main/notebooks/notebook_hyperparameter_tuning.ipynb

I.2 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

J Code-snippets

J.1 run-prolog

[https://github.com/niklasmellgren/
masters-thesis/blob/main/inference/
agentic_internal.py#L178](https://github.com/niklasmellgren/masters-thesis/blob/main/inference/agentic_internal.py#L178)