Learned Verification for Mathematical Reasoning: A Detailed Technical Review

Siddharth Magesh

October 2026

Paper 1: Training Verifiers to Solve Math Word Problems

Authors: Karl Cobbe et al.
Year: 2021

Core Problem and Proposed Solution

The core problem is that autoregressive language models struggle with **robust multi-step mathematical reasoning**: a single incorrect intermediate step can propagate through the rest of a solution, and the model has no inherent mechanism to detect or recover from that mistake. The authors want to improve this without relying solely on substantially larger models. They therefore formulate the problem as one of **candidate generation plus correctness evaluation** rather than one-shot generation.

Their solution is to first fine-tune a **generator** on GSM8K, use it to sample many diverse candidate solutions for each problem, label those candidates by whether their final answer is correct, and then train a separate **verifier** to estimate the correctness of a candidate conditioned on the question and solution. At inference time, the generator produces multiple solutions, the verifier assigns each a correctness score, and the system selects the highest-scoring solution. Formally, they turn

$$Q \to S$$

into

$$Q \to \{S_1, \ldots, S_N\} \to V(Q, S_i) \to \arg\max_i V(Q, S_i),$$

with the goal of exploiting the fact that **searching over multiple generated solutions plus learned verification can scale more effectively than simply making one generator larger**.

Dataset: GSM8K

GSM8K (Grade School Math 8K) is the dataset introduced in this paper to evaluate **multi-step mathematical reasoning in natural language**. It contains **8.5K human-written problems**: **7.5K training** and **1K test** problems, typically requiring **2–8 reasoning steps** using elementary arithmetic.

Mathematical Formulation

The method consists of two learned components: a **generator** that produces candidate reasoning traces and a **verifier** that estimates whether those traces will lead to the correct answer. The important distinction is that the generator is optimized to model the distribution of solutions, while the verifier is optimized to predict their correctness.

1. Generator Objective. For a math problem Q and a solution represented as a sequence of tokens $S = (s_1, s_2, \ldots, s_T)$, the generator is an autoregressive language model:

$$\boxed{P_\theta(S \mid Q) = \prod_{t=1}^{T} P_\theta(s_t \mid Q, s_{<t})}$$

where θ denotes the generator parameters and $s_{<t}$ represents all solution tokens preceding token t. The generator is fine-tuned using the standard language-model negative log-likelihood objective,

$$\boxed{\mathcal{L}_{\mathrm{LM}} = -\sum_{t=1}^{T} \log P_\theta(s_t \mid Q, s_{<t})}$$

so it learns a probability distribution over complete natural-language reasoning traces rather than directly predicting only a final numerical answer.

2. Candidate Solution Generation. Rather than using a single generated solution, the generator is used to sample multiple candidate reasoning traces:

$$\boxed{S_1, S_2, \ldots, S_N \sim P_\theta(S \mid Q)}$$

The paper uses $N = 100$ completions per training problem, sampled at temperature $T = 0.7$. Given token logits z_k, temperature sampling can be written as

$$P(s_t = k) = \frac{\exp(z_k/T)}{\sum_j \exp(z_j/T)}.$$

Temperature controls candidate diversity: a **lower temperature** produces more concentrated and similar outputs, while a **higher temperature** produces more diverse reasoning trajectories. The paper uses $T = 0.7$ for multi-sample generation and observes that overtraining reduces diversity and hurts test@100 performance. Thus, verification depends on both candidate coverage and verifier accuracy:

$$\boxed{\text{Verification Performance} \propto \text{Generation Coverage} \times \text{Verifier Quality}}$$

The purpose of sampling is to maintain sufficient diversity in the generator's outputs so that different candidate solutions can contain different reasoning trajectories. The verifier can then search over this candidate set rather than being restricted to a single generation.

3. Verifier Target. Each generated solution S_i is assigned a binary outcome label according to whether its *final answer* is correct:

$$\boxed{y_i = \begin{cases} 1, & \text{if } S_i \text{ reaches the correct final answer,} \\ 0, & \text{otherwise.} \end{cases}}$$

The verifier, parameterized by ϕ, therefore learns a function of the form

$$\boxed{V_\phi(Q, S) \approx P(y = 1 \mid Q, S).}$$

Importantly, the label describes the outcome of the *entire solution*; it does not indicate whether every intermediate reasoning step is logically correct. Consequently, a solution containing flawed reasoning can still receive $y = 1$ if it happens to reach the correct final answer, which the authors identify as a source of false-positive training labels.

4. Token-Level Verifier. Instead of predicting correctness only after observing the complete solution, the default verifier predicts a value after every token:

$$\boxed{V_\phi(Q, s_{\le t}) \approx P(y = 1 \mid Q, s_{\le t}), \qquad t = 1, \ldots, T.}$$

Thus, a reasoning trajectory

$$Q \to s_1 \to s_2 \to \cdots \to s_T$$

produces a sequence of verifier estimates

$$V_1, V_2, \ldots, V_T.$$

Each V_t can be interpreted as the verifier's estimate, given the reasoning generated so far, of whether the trajectory will ultimately produce the correct answer. The authors describe this as a **token-level value function**. Crucially, these are not independently supervised step-correctness labels: every token-level prediction is trained using the same final outcome y associated with the complete solution.

5. Token-Level Verifier Loss. The verifier is trained using mean squared error (MSE). For a solution containing T tokens, the token-level verification objective can be represented as

$$\boxed{\mathcal{L}_{\text{ver}} = \frac{1}{T}\sum_{t=1}^{T}\left(V_\phi(Q, s_{\le t}) - y\right)^2}$$

where T is the number of solution tokens, Q is the original math problem, and $s_{\le t}$ is the partial solution from the first token through token t. The quantity $V_\phi(Q, s_{\le t})$ is the verifier's predicted value at that point in the reasoning trajectory, while $y \in \{0, 1\}$ is the binary label assigned to the *complete* solution.

The squared-error term

$$\left(V_\phi(Q, s_{\le t}) - y\right)^2$$

penalizes disagreement between the verifier's intermediate prediction and the eventual outcome. If the complete solution is correct ($y = 1$), predictions along its trajectory are pushed toward 1; if the final answer is incorrect ($y = 0$), they are pushed toward 0. The factor $1/T$ averages this error across the solution tokens. Therefore, although supervision is obtained only from the final answer, the training signal is propagated across the entire generated trajectory. The paper reports using MSE for this objective and notes that replacing it with cross-entropy produced negligible differences.

6. Joint Verifier Objective. The verifier is also trained with a language-modeling objective. Therefore, it simultaneously learns to model solution text and estimate solution correctness. Conceptually, the combined objective is

$$\boxed{\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{LM}} + \mathcal{L}_{\text{ver}}}$$

with no additional weighting between the two losses. The language-model component learns

$$\underbrace{P_\phi(s_t \mid Q, s_{<t})}_{\text{model the solution distribution}},$$

while the verification component learns

$$\underbrace{V_\phi(Q, s_{\le t})}_{\text{estimate eventual correctness}}.$$

The losses on question tokens are masked so that the training objectives are applied to the generated solution portion. The paper's ablation experiments show that retaining the language-modeling objective alongside verification improves verifier performance.

7. Test-Time Selection. For a new problem Q, the generator first samples a collection of candidate solutions,

$$S_1, \ldots, S_N \sim P_\theta(S \mid Q).$$

After each complete solution is generated, the verifier assigns it a score

$$v_i = V_\phi(Q, S_i),$$

and the final output is selected by

$$\boxed{S^* = \underset{S_i \in \{S_1, \ldots, S_N\}}{\arg\max} V_\phi(Q, S_i).}$$

The generator therefore defines the **search space** of possible reasoning traces, while the verifier provides the **selection criterion**. The overall reasoning procedure can be summarized as

$$\boxed{Q \to \{S_1, \ldots, S_N\} \to \{V_\phi(Q, S_1), \ldots, V_\phi(Q, S_N)\} \to \arg\max_i V_\phi(Q, S_i).}$$

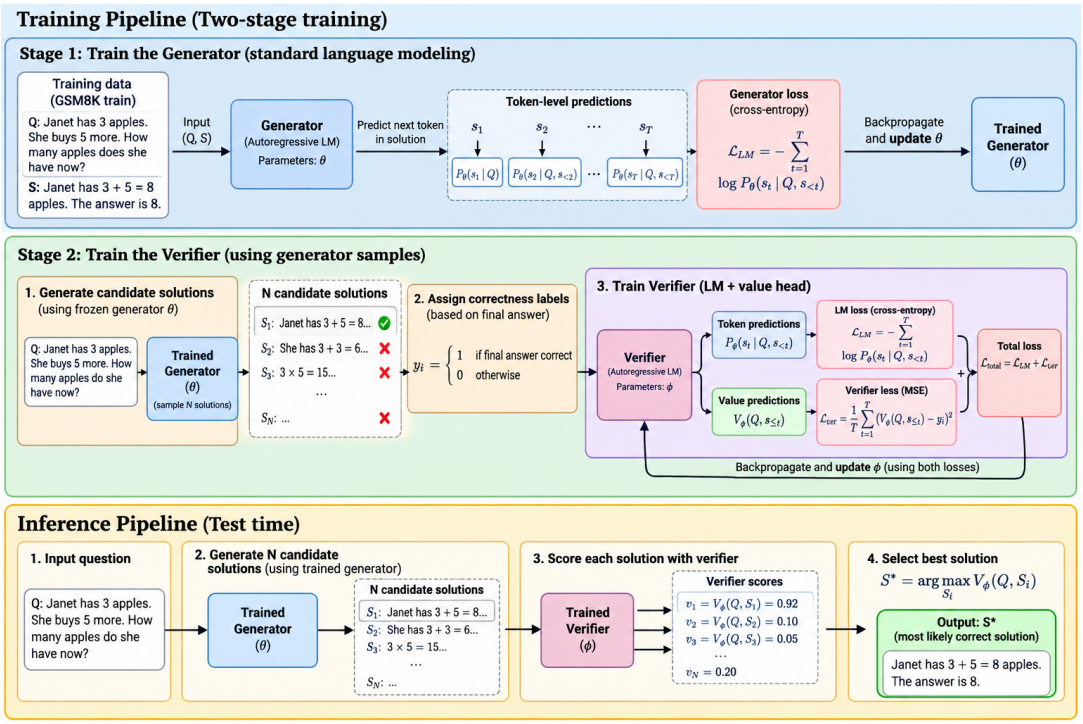


Figure 1: Generator–Verifier reasoning pipeline for candidate generation and selection.

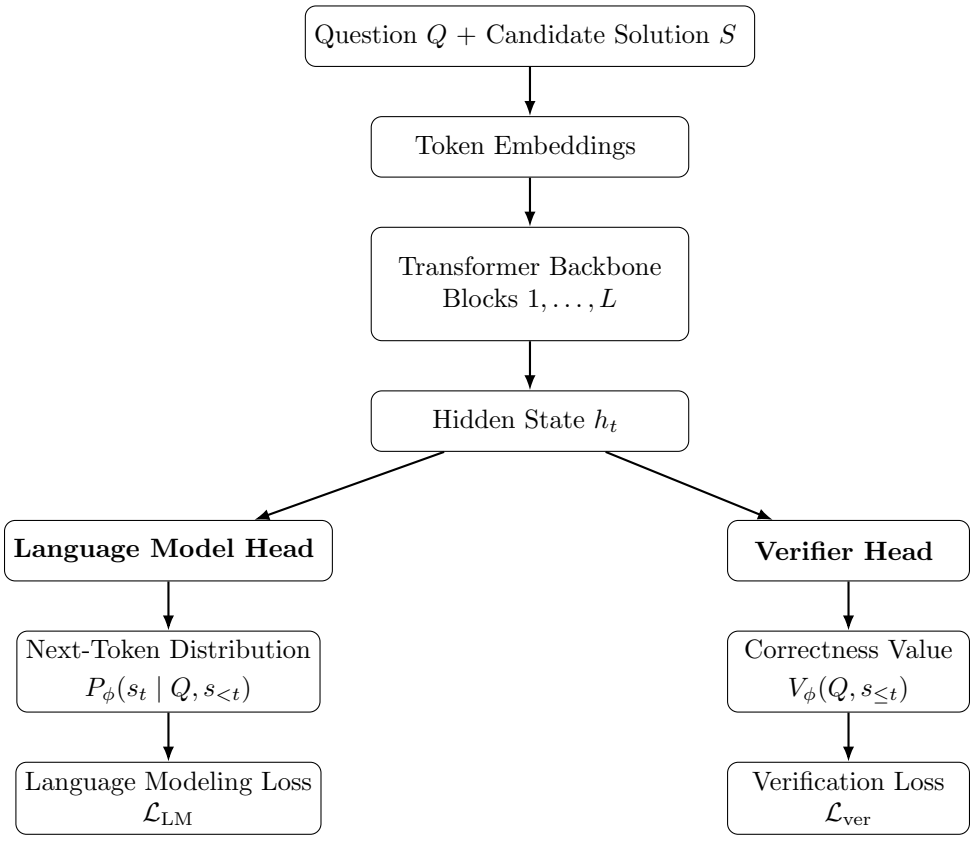


Figure 2: High-level architecture of the token-level verifier. A shared Transformer backbone produces hidden states h_t. The language-model head predicts the next token, while the verifier head predicts the correctness value $V_\phi(Q, s_{\leq t})$.

Verification Training and Inference

The verification pipeline first fine-tunes a **generator for 2 epochs** on GSM8K and samples **100 candidate solutions per training problem at** $T = 0.7$, labeling each candidate by final-answer correctness. The **verifier is initialized from its corresponding fine-tuned generator** and trained for **1 epoch** using the joint objective

$$\mathcal{L} = \mathcal{L}_{\text{LM}} + \mathcal{L}_{\text{ver}}.$$

Importantly, the two losses use different training signals. The language-modeling objective is trained on an equal mixture containing the **original GSM8K solution data**, effectively upsampled to match the verifier data, while the verification objective operates on the **generated, outcome-labeled candidate solutions**. Losses on question tokens are masked. The separate **fine-tuning baseline**, in contrast, is trained for **20 epochs**.

At inference time, 100 candidate solutions are generated and ranked by the verifier. **test@100** measures whether at least one of these candidates is correct, whereas **verifier-selected accuracy** measures whether the candidate actually chosen by the verifier is correct. Conceptually, these two requirements can be separated as

$$\boxed{P(\text{selected answer correct}) = P(\text{correct candidate exists})\, P(\text{select correct} \mid \text{correct candidate exists})}$$

highlighting that successful verification requires both **generator coverage** and **accurate verifier selection**.

Lightweight Verifier Head. The verifier does not require a separate reasoning network; it reuses the Transformer's learned representations and adds only a lightweight prediction mechanism. In the paper's implementation, a learned **gain and bias** are applied to the language-model logit of a special vocabulary token reserved for verification. Thus, most of the reasoning capacity remains in the shared Transformer backbone, while the additional verifier component is minimal.

Hence, instead of relying on a single generation, the method converts mathematical reasoning into a **sample-and-rank procedure**: generate multiple reasoning trajectories, estimate their probability of correctness, and return the trajectory receiving the highest verifier score.

Solution-Level vs. Token-Level Verification

Solution-level verification makes a single correctness prediction after reading the complete solution, whereas **token-level verification** predicts correctness throughout the reasoning trajectory:

$$\text{Solution-level: } (Q, S) \to V(Q, S)$$

$$\text{Token-level: } V(Q, s_{\leq 1}),\ V(Q, s_{\leq 2}), \ldots, V(Q, s_{\leq T})$$

Importantly, token-level verification is **not process supervision**. Every token-level prediction is trained using the same final-answer label $y \in \{0, 1\}$; there are no labels specifying whether individual reasoning steps are correct. Instead, it learns a token-level value function,

$$\boxed{V_\phi(Q, s_{\leq t}) \approx P(\text{eventual success} \mid Q, s_{\leq t})}$$

Although this objective is initially harder and noisier, it ultimately performs better and overfits less than solution-level verification. The authors hypothesize that predicting value throughout the trajectory encourages the verifier to learn properties of the evolving reasoning process rather than relying mainly on patterns associated with the completed answer.

Optionality and Inference-Time Compute

A key advantage of verification is **optionality**. Instead of committing to a single generated solution,

$$\text{Generate One Solution},$$

the system can generate multiple candidates and select among them:

$$\boxed{\text{Generate Many Solutions} \to \text{Verify} \to \text{Select Best}}$$

This allows additional **inference-time compute** to be spent on exploring more candidate solutions rather than relying only on increasing model size.

Generator Quality vs. Verifier Quality

The paper finds that a **large generator with a small verifier** performs better than a **small generator with a large verifier**. The verifier can only select from solutions that the generator produces; therefore, a stronger verifier cannot compensate for the absence of correct candidates.

$$\boxed{\text{Generator provides candidate support } \rightarrow \text{ Verifier performs selection}}$$

Solve Rate and Scaling

Solve rate is the fraction of test problems solved correctly. The results suggest that merely scaling the generator can be inefficient: on the full GSM8K dataset, verification with a **6B model slightly outperforms a fine-tuned 175B model**.

Verification also benefits strongly from **data scaling**. With small datasets, the verifier may overfit to correct answers before learning generalizable properties of correct reasoning. As the dataset grows, verification becomes increasingly effective:

$$\boxed{\text{More Training Data } \rightarrow \text{ Greater Benefit from Verification}}$$

This supports the paper's conclusion that verification scales more favorably with additional training data than standard fine-tuning.

Limits of Test-Time Search

Increasing the number of candidates N initially improves performance by increasing the chance of generating a correct solution. However, because the verifier is imperfect, searching over too many candidates also increases the chance of finding an incorrect solution that receives an erroneously high verifier score:

$$\boxed{N \uparrow \Rightarrow \begin{cases} \text{more correct candidates} \\ \text{more opportunities to fool the verifier} \end{cases}}$$

In Figure 7a, performance improves up to roughly **400 candidates** and then declines. Thus, additional test-time search is beneficial only when verifier reliability is sufficient for the size of the candidate space.

Verifier Ranking with Top-k Voting

Selecting only the highest-scoring solution can be risky because an incorrect candidate may occasionally receive an erroneously high verifier score. To reduce this dependence on a single prediction, the authors combine **verifier ranking with majority voting**. The verifier first ranks all generated solutions, after which only the top-k solutions are retained and their **final answers** are used for voting:

$$\boxed{\text{Candidates} \rightarrow \text{Verifier Ranking} \rightarrow \text{Top-}k \rightarrow \text{Majority Vote on Final Answers}}$$

The verifier therefore acts as a **filter**, removing low-quality candidates, while voting uses agreement among several highly ranked solutions rather than trusting the verifier's single favorite candidate. This can make selection more robust to occasional verifier errors.

The optimal k depends on the number of generated candidates. With 100 candidates, approximately the **top 3–5** solutions work best, whereas with 3,200 candidates, roughly the **top 30** can participate effectively. Thus, increasing the search space also allows more high-quality candidates to contribute to the final decision.

Effect of Dropout

The paper uses **20% residual dropout** ($p = 0.2$) as a regularizer and finds that it improves both fine-tuning and verification. The effect is particularly important for **solution-level verifiers**, which are more prone to overfitting. Token-level verifiers are already more robust, but still receive a smaller improvement from dropout. Overall, dropout helps reduce the recurring overfitting problem observed throughout the experiments.

Verifier Behavior and Limitations

The token-level confidence visualizations show that the verifier can sometimes assign a low score to a solution that reaches the **correct final answer through incorrect reasoning**. However, it can also miss subtle reasoning errors and assign high confidence to an incorrect solution. Thus, the verifier should be viewed as a **learned statistical judge rather than a perfect mathematical proof checker**.

Calculator-Assisted Arithmetic

Since language models frequently make arithmetic errors, the authors train the models to generate special **calculator annotations**. During inference, when such an annotation is produced, the arithmetic computation is performed externally and the exact result is inserted into the solution. Therefore, the language model primarily handles the reasoning structure while deterministic arithmetic can be delegated to a calculator:

$$\boxed{\text{Reasoning Generation} \rightarrow \text{Calculator Call} \rightarrow \text{Exact Arithmetic Result}}$$

Training and Inference Pipeline

```
# Training
generator = finetune(generator, GSM8K)

for question in GSM8K:
    solutions = generate(generator, question, N=100, T=0.7)
    for solution in solutions:
        label = check_final_answer(solution)
        verifier_data.add(question, solution, label)

verifier = train(verifier_data, LM_loss + verifier_loss)

# Inference
solutions = generate(generator, question, N=100, T=0.7)
scores = verifier.score(solutions)
answer = solutions[argmax(scores)]
```

Implementation Details and Hyperparameters

The main experimental setup trains the generator for **2 epochs** and the verifier for **1 epoch**. For verification, the generator samples **100 solutions per problem** at temperature 0.7. The verifier uses MSE together with the language-modeling objective. The experiments use Adam with $\beta_1 = 0.9$, $\beta_2 = 0.95$, a verification learning rate of 10^{-5}, a maximum sample length of 400 tokens, and residual dropout of $p = 0.2$ in the dropout experiments.

Limitations

Outcome-based supervision. The verifier is trained using **final-answer correctness**, not reasoning correctness. Therefore, a solution with flawed intermediate reasoning can still receive a positive label if it reaches the correct final answer, introducing false-positive training examples.

No true step-level supervision. Although the token-level verifier makes predictions throughout the solution, every prediction is trained from the same final outcome label. There are no ground-truth labels indicating whether individual reasoning steps are correct.

Imperfect verifier. The verifier is not a correctness oracle. Increasing the number of candidates initially improves performance, but sufficiently large search can uncover incorrect solutions that receive unusually high verifier scores:

$$\boxed{\text{More Search} \;\Rightarrow\; \text{Better Coverage} \;+\; \text{Greater Exposure to Verifier Errors}}$$

Dependence on generator quality. Verification can only select among generated candidates. If the generator fails to produce a correct solution, the verifier cannot recover one. The paper's generator/verifier-size ablation also shows that increasing generator size has a larger impact than increasing verifier size.

Inference cost and limited scope. Verification requires generating and evaluating many candidate solutions, increasing test-time computation. Moreover, the experiments primarily focus on GSM8K and elementary mathematical reasoning, so the paper does not establish that the same gains will extend to substantially more complex reasoning tasks.

Conclusion

The central contribution of the paper is the decomposition of mathematical reasoning into two distinct components:

$$\boxed{\text{Reasoning} = \text{Candidate Generation} + \text{Candidate Evaluation}}$$

Rather than requiring a language model to produce the correct solution in a single attempt, the proposed system generates multiple reasoning trajectories and trains a separate verifier to select among them. On GSM8K, this **generate-and-verify** framework improves over standard fine-tuning, benefits from additional training data and test-time sampling, and can achieve strong performance even when the verifier is substantially smaller than the generator.

More broadly, the paper demonstrates that reasoning performance can be improved not only by scaling the generator, but also by allocating computation to **search and learned verification**. However, its verifier ultimately learns from outcome-level correctness:

Did the solution reach the correct answer?

rather than explicit supervision of

Was each intermediate reasoning step correct?

This distinction leaves a central unresolved problem: how to directly supervise and verify the reasoning process itself rather than only its final outcome.

Paper 2: Let's Verify Step by Step

Authors: Hunter Lightman et al.
Year: 2023

The central problem is that language models can generate convincing **multi-step reasoning containing hidden logical errors**. Outcome supervision evaluates only the final result, providing no direct information about **where the reasoning fails** and potentially rewarding incorrect reasoning that happens to reach the correct answer.

The authors instead investigate **process supervision** by training a **Process Reward Model (PRM)** to evaluate individual reasoning steps. Human annotators label each step as *positive*, *negative*, or *neutral*, providing direct supervision over the reasoning process. At inference time, the PRM evaluates each step of a candidate solution and aggregates the step-level scores into an overall solution score for Best-of-N selection.

Outcome Reward Model vs. Process Reward Model

Outcome Reward Model (ORM). For a problem x and a generated solution $S = (s_1, s_2, \ldots, s_T)$, the ORM receives a single outcome label $y \in \{0, 1\}$ indicating whether the final answer is correct. It learns to estimate

$$P_\theta(y = 1 \mid x, S).$$

Given the predicted correctness probability p, the ORM is trained using binary cross-entropy,

$$\boxed{\mathcal{L}_{\text{ORM}} = -\left[y \log p + (1 - y) \log(1 - p)\right].}$$

During training, the same solution-level label is assigned throughout the solution, while at inference time the prediction at the **final token** is used as the overall solution score. The key limitation is **credit assignment**: the ORM knows whether the complete solution succeeded or failed, but receives no direct information about **which reasoning step caused the failure**. Thus, ORM supervision is fundamentally solution-level, whereas the PRM receives explicit step-level feedback.

Process Reward Model (PRM). The PRM instead receives explicit supervision for individual reasoning steps. Each labeled step is assigned

$$y_t \in \{\text{positive}, \text{negative}, \text{neutral}\},$$

and the model learns

$$P_\theta(y_t \mid x, s_{\le t}).$$

Training maximizes the likelihood of the correct step-level labels, which can be written as

$$\mathcal{L}_{\text{PRM}} = -\sum_t \log P_\theta(y_t \mid x, s_{\le t}).$$

At inference time, let p_t denote the PRM's probability that step t is positive. The paper converts the step-level predictions into a single solution score using the product

$$\boxed{R(S) = \prod_{t=1}^{T} p_t}$$

and ranks candidate solutions according to this score. Thus, the key distinction is that the ORM provides **outcome-level credit assignment**, whereas the PRM receives direct information about **where the reasoning succeeds or fails**.

Motivation from Prior Work

Uesato et al. (2022) previously compared **outcome supervision** and **process supervision** on grade-school mathematics. They found that process supervision was more data-efficient, but both approaches achieved similar final-answer performance at the scale studied.

Let's Verify Step by Step revisits this comparison under a substantially stronger setting: **more capable models**, **much more process supervision**, and the significantly harder **MATH benchmark**. The central question is whether the advantage of process supervision becomes more pronounced as the model, supervision scale, and reasoning difficulty increase.

Methodology: Process Reward Model

The PRM pipeline consists of four stages: **step-level data collection**, **PRM training**, **solution-level scoring**, and Best-of-N selection. For a problem x, a base language model is first fine-tuned to produce **newline-separated, step-by-step reasoning**. The authors few-shot generate solutions to MATH problems, retain solutions that reach the correct final answer, and fine-tune the generator for **one epoch**. Importantly, this stage is primarily intended to enforce a clean reasoning format rather than teach new mathematical capabilities.

The resulting generator produces

$$G(x) \to S = (s_1, s_2, \ldots, s_T),$$

where each s_t corresponds to an identifiable reasoning step. This structured format allows human annotators to provide explicit step-level supervision by assigning each step a label

$$y_t \in \{\text{positive}, \text{negative}, \text{neutral}\}.$$

A positive label indicates a correct and reasonable step that progresses toward the solution, a negative label indicates an incorrect or unreasonable step, and a neutral label represents an acceptable but potentially ambiguous or non-progressing step. For incorrect solutions, supervision is deliberately stopped after the **first incorrect step**.

Step-Level Prediction and Training. The PRM is initialized from a mathematical language-model checkpoint trained on **MathMix**, a corpus containing approximately **1.5B math-related tokens**. It is then fine-tuned to classify individual reasoning steps. The PRM does not evaluate a step in isolation; for step t, it conditions on the problem and the complete reasoning prefix,

$$\boxed{P_\theta(y_t \mid x, s_{\le t})}$$

where θ denotes the PRM parameters. The three labels are implemented as prediction tokens, allowing training within a standard language-model pipeline. The PRMs are trained for **two epochs**, and the authors report that **low learning rates** are important for stable training because moving from ordinary language modeling to step classification creates a substantial distribution shift.

For a human-provided label y_t^*, the step-level negative log-likelihood is

$$\mathcal{L}_t = -\log P_\theta(y_t^* \mid x, s_{\le t}).$$

Across a dataset containing M solutions, the complete training objective can therefore be written as

$$\boxed{\mathcal{L}_{\text{PRM}}(\theta) = -\sum_{i=1}^{M} \sum_{t=1}^{T_i'} \log P_\theta \left(y_{i,t}^* \mid x_i, s_{i,\le t}\right)}$$

where T_i' denotes the number of supervised steps for solution i. Training then optimizes

$$\boxed{\theta^* = \arg\min_\theta \mathcal{L}_{\text{PRM}}(\theta).}$$

The PRMs are trained for **two epochs**. The authors also report that low learning rates are important because fine-tuning a language model from ordinary next-token prediction to step-level classification introduces a substantial distribution shift.

From Step Scores to a Solution Score. After training, a single PRM forward pass over a candidate solution produces correctness probabilities for its individual reasoning steps. Let

$$p_t = P_{\theta^*}(\text{step } t \text{ is positive} \mid x, s_{\le t}).$$

For scoring, the authors treat **neutral labels as positive** and aggregate the step-level scores using their product:

$$\boxed{R(S) = \prod_{t=1}^{T} p_t.}$$

This aggregation makes the overall score sensitive to weak reasoning steps: even if most steps receive high scores, one suspicious step can substantially lower the score of the complete trajectory. The authors also test minimum aggregation, but find that the **product with neutral treated as positive** performs slightly best.

Best-of-N Selection. At inference time, the fixed generator produces N candidate solutions,

$$S_1, S_2, \ldots, S_N.$$

The trained PRM computes a solution-level score $R(S_i)$ for each candidate, and the final solution is selected as

$$\boxed{S^* = \arg\max_{i=1,\ldots,N} R(S_i).}$$

The complete PRM pipeline can therefore be summarized as

$$\boxed{\text{Generate Solutions} \rightarrow \text{Human Step Labels} \rightarrow \text{Train Step-Level PRM} \rightarrow \text{Score Steps} \rightarrow \prod_t p_t \rightarrow \text{Best-of-}N}$$

The central distinction is that **training supervision occurs at the reasoning-step level**, while at inference time these step-level predictions are **aggregated to rank complete solutions**.

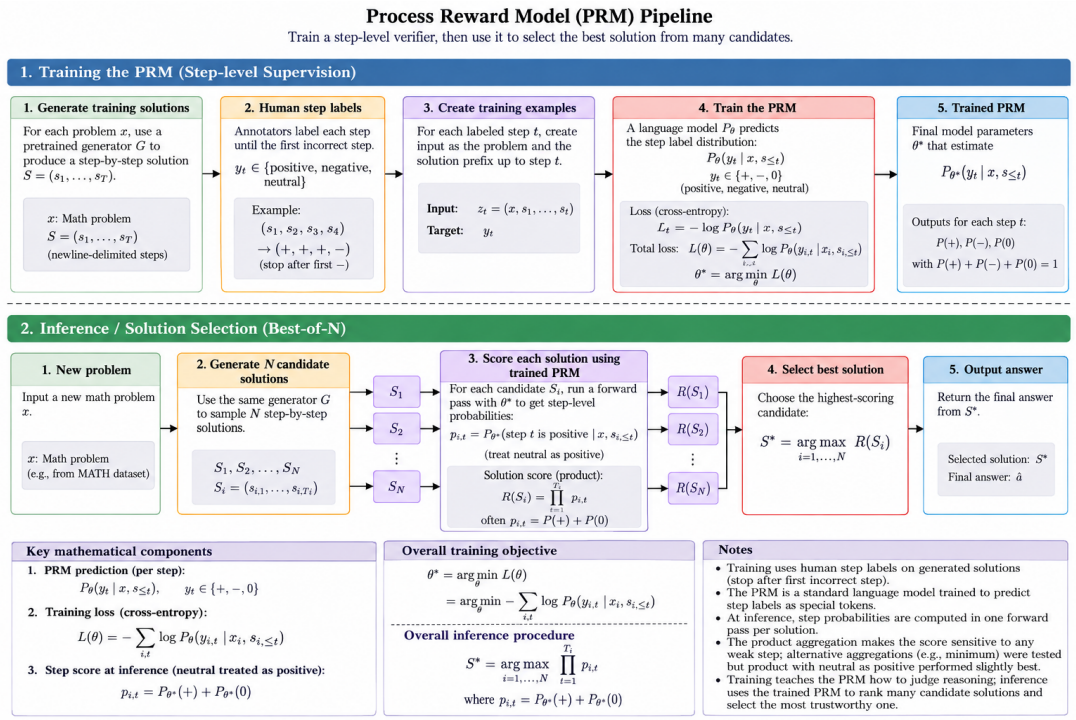


Figure 3: PRM End-to-End Pipeline

PRM Training and Inference Pipeline

```
# Training
for problem in math_train:
    solutions = generator.sample(problem, N)

    for solution in solutions:
        steps  = split_into_steps(solution)
        labels = get_human_labels(steps)

        for t, label in enumerate(labels):
            probs = prm(problem, steps[:t+1])
            update(prm, cross_entropy(probs, label))

# Inference
def solve(problem):
    solutions = generator.sample(problem, N)
```

```
scores = [
    product(prm.score_steps(problem, split_into_steps(s)))
    for s in solutions
]
return solutions[argmax(scores)]
```

Datasets and Evaluation

The PRM800K annotation procedure differs across its two collection phases. In **phase 1**, annotators interacted with multiple alternative step completions and could select or provide steps during the trajectory. In **phase 2**, complete solutions were generated in advance, and annotation stopped once the **first negative step** was identified.

The paper primarily uses the **MATH dataset**, a challenging benchmark of competition-level mathematics problems. For process supervision, the authors construct **PRM800K**, containing approximately **800,000 human step-level labels** across **75,000 model-generated solutions** from roughly **12,000 problems**. Each reasoning step is labeled as *positive*, *negative*, or *neutral*, with supervision typically ending at the first incorrect step.

The main evaluation uses a held-out set of **500 MATH problems**. To examine out-of-distribution generalization, the authors additionally evaluate on recent **AP Physics, AP Calculus, AP Chemistry, AMC10, and AMC12** problems that were released after the model's pretraining data was collected.

PRM Solution Scoring. For each reasoning step t, the PRM predicts probabilities over positive, negative, and neutral labels. Under the paper's default scoring rule, **neutral is treated as positive**, so the step-level correctness score is

$$\boxed{p_t = P_\theta(+ \mid x, s_{\le t}) + P_\theta(0 \mid x, s_{\le t})}$$

and the complete solution score is

$$\boxed{R(S) = \prod_t p_t.}$$

Thus, candidate solutions are ranked by the product of their acceptable-step probabilities, and the highest-scoring solution is selected. Among the four tested combinations of neutral handling and aggregation strategy, this default rule achieves the best reported performance.

Active Learning for Process Supervision

Active learning is used to make **human step-level annotation more efficient**. Instead of labeling randomly sampled solutions, the authors prioritize **convincing wrong-answer solutions**: candidates that reach an incorrect final answer but receive a high score from the current PRM. Since these solutions successfully fool the verifier, human step labels provide targeted supervision on the PRM's current weaknesses.

The process can be summarized as

$$\boxed{\text{Generate} \rightarrow \text{PRM Score} \rightarrow \text{Select Convincing Errors} \rightarrow \text{Human Label} \rightarrow \text{Retrain PRM}}$$

In the small-scale experiments, this strategy is estimated to be approximately **2.6× more data-efficient** than uniform labeling. However, an experiment that iteratively retrained the selector was unstable and produced no additional improvement, so the 2.6× result applies specifically to the tested active-learning strategy rather than active learning in general.

Experimental Design and Best-of-N Evaluation

The experiments use two complementary settings. In the **large-scale setting**, GPT-4-based models and human labels from PRM800K are used to train the strongest PRM and ORM. However, their training datasets are not directly comparable. To obtain a more controlled comparison, the **small-scale setting** uses the large PRM as a synthetic supervisor for smaller reward models, allowing process and outcome supervision to be compared while keeping the underlying solution data fixed.

The primary evaluation metric is **Best-of-N**. For each problem, the generator produces N candidate solutions, the reward model scores them, and the highest-scoring candidate is selected:

$$S^* = \arg\max_{i=1,\dots,N} R(S_i).$$

The problem is considered solved if the selected candidate has the correct final answer. Thus, Best-of-N directly measures the reward model's ability to **search through generated reasoning trajectories and identify a correct solution**.

The reported **78.2% Best-of-1860** result is evaluated on a **500-problem held-out subset of MATH**. During data collection, the authors added **4,500 problems from the original MATH test set** to the training pool, leaving 500 randomly selected problems for final evaluation. This modified split is separate from the paper's discussion of possible pretraining contamination.

More Candidates → More Search Opportunity → PRM Selects the Best Reasoning

Controlled Comparison of Process and Outcome Supervision

The large-scale PRM–ORM comparison is not perfectly controlled because the two models are trained on different datasets. PRM800K is constructed using active learning and is strongly biased toward convincing wrong-answer solutions, whereas the ORM uses a much larger set of uniformly sampled solutions. In addition, final-answer supervision can produce **false positives** when incorrect reasoning happens to reach the correct answer. Therefore, the large-scale results alone cannot fully isolate the effect of process supervision.

To obtain a controlled comparison, the authors use the trained $\text{PRM}_{\text{large}}$ as a **synthetic supervisor** for smaller reward models. For the same generated solutions, $\text{PRM}_{\text{large}}$ provides either step-level labels for process supervision or a single solution-level label for outcome supervision. A third model is trained using ordinary final-answer checking. This keeps the underlying candidate solutions fixed while changing the supervision signal.

The controlled experiments show that **process supervision consistently outperforms both forms of outcome supervision** across different training-data scales. Outcome supervision from $\text{PRM}_{\text{large}}$ also performs better than final-answer supervision because it can identify solutions containing incorrect reasoning despite reaching the correct final answer.

Same Solutions + Different Supervision $\Rightarrow$ Process Supervision Performs Better

This provides stronger evidence that the PRM's advantage is not solely caused by differences in dataset construction or noisy final-answer labels; the **step-level supervision itself provides a more informative learning signal**.

Generalization and Difficulty Analysis

Out-of-Distribution Generalization. The authors evaluate the reward models on **224 recent STEM questions** from AP Calculus, AP Chemistry, AP Physics, AMC10, and AMC12. These exams were released after the model's pretraining dataset was compiled, reducing concerns about memorization. The PRM achieves **72.9%** aggregate Best-of-100 performance, compared with **63.8% for the ORM** and **61.3% for majority voting**. This provides evidence that the PRM advantage survives a moderate distribution shift beyond MATH, although the evaluation remains within mathematical and STEM reasoning.

Performance Across Difficulty. When MATH problems are grouped by generator solve rate, the PRM outperforms the ORM across **all difficulty levels**. The difference is therefore not restricted to the hardest problems. For some easier problems, ORM performance even decreases as more candidates are searched, indicating that larger search can expose convincing incorrect solutions that fool the ORM. The PRM remains more robust as the candidate set grows.

Overall, the experiments provide evidence that **process supervision improves Best-of-N mathematical reasoning**, that the tested active-learning strategy improves labeling efficiency, and that the PRM advantage extends to the paper's OOD STEM benchmark. However, these experiments do not establish that the same advantage necessarily generalizes to reasoning domains outside mathematics and STEM.

Why Process Supervision Helps: Credit Assignment

The authors argue that the main advantage of process supervision is improved **credit assignment**. When an ORM receives a negative label for an entire solution, it must infer which part of the reasoning caused the failure. A PRM instead receives localized feedback identifying the first incorrect step, providing a much more precise learning signal:

Outcome Supervision: "Something went wrong" Process Supervision: "It went wrong here"

The experiments strongly support the effectiveness of process supervision, but **credit assignment is the authors' explanation for the improvement rather than an independently isolated causal result**. Other factors, including active learning, label distributions, step-level modeling, and solution-score aggregation, may also contribute. Therefore, the supported conclusion is that process supervision performs better in this setting, with improved credit assignment as a plausible major mechanism.

Limitations

Dependence on human supervision. PRM performance depends on the quality and consistency of human step-level labels. Some reasoning steps are inherently ambiguous, motivating the *neutral* label for steps that may be valid but unclear, unhelpful, or insufficiently progressive. Human labeling errors or inconsistencies can therefore propagate into the learned reward model.

Length bias in product scoring. The default solution score is

$$R(S) = \prod_{t=1}^{T} p_t.$$

Since step probabilities are generally below one, multiplying more terms naturally penalizes longer reasoning trajectories. Consequently, a longer correct solution may receive a lower score simply because it contains more steps. The authors explicitly acknowledge this slight length bias.

Potential test-set contamination. The authors cannot guarantee that MATH problems were completely absent from the models' pretraining data. Although they apply decontamination procedures, including string matching and n-gram checks, rephrased versions may remain undetected. This creates uncertainty around absolute benchmark performance, although the authors argue that relative comparisons between methods are less likely to be substantially affected.

Limited domain generalization. The paper primarily evaluates process supervision on **mathematical reasoning**, where intermediate steps have relatively objective correctness criteria. The results therefore do not establish that the same benefits will extend to domains with more subjective or difficult-to-evaluate reasoning. The authors explicitly leave broader generalization beyond mathematics as an open question.

Active-learning distribution bias. The active-learning pipeline lets the current PRM determine which solutions are selected for human labeling,

$$\text{PRM}_t \rightarrow \text{Select Hard Examples} \rightarrow \text{Human Labels} \rightarrow \text{PRM}_{t+1}.$$

This efficiently targets verifier weaknesses, but creates a **non-representative training distribution** biased toward difficult, incorrect solutions. In PRM800K, only **14.2% of solutions** ended correctly, although **73.1% of individual steps** were labeled correct.

Key Takeaways

The paper shows that **final-answer supervision provides weak credit assignment** for multi-step reasoning, while process supervision directly identifies where reasoning fails. Its central contribution is the comparison between **Outcome Reward Models (ORMs)** and **Process Reward Models (PRMs)**, supported by the release of **PRM800K**, containing approximately 800K human step-level labels.

The PRM converts step-level correctness predictions into a solution score using

$$R(S) = \prod_{t} p_t,$$

and uses this score for Best-of-N selection. Active learning further improves labeling efficiency by prioritizing convincing wrong solutions. At Best-of-1860, the main large-scale result is

$$\boxed{\text{PRM } 78.2\% > \text{ORM } 72.4\% > \text{Majority Voting } 69.6\%}$$

The authors attribute the PRM's advantage primarily to **better credit assignment**, although the evidence is mainly within mathematical reasoning and the method remains limited by human-label quality, scoring bias, possible contamination, and uncertain generalization beyond mathematics.

$$\boxed{\text{Supervising where reasoning fails is more informative than only supervising the final outcome.}}$$

Paper 3: Generative Verifiers: Reward Modeling as Next-Token Prediction

Authors: Lunjun Zhang et al.
Year: 2025

Core Problem and Proposed Solution

The core problem addressed in *Generative Verifiers: Reward Modeling as Next-Token Prediction* is that existing LLM-based verifiers are typically trained as **discriminative reward models** that map a problem–solution pair directly to a scalar correctness score. While effective for Best-of-N selection, this formulation does not fully exploit the generative capabilities of pretrained language models. In particular, discriminative verifiers cannot naturally produce verification rationales, use chain-of-thought reasoning, or improve their judgment by allocating additional inference-time computation. This is especially limiting for reasoning tasks, where candidate solutions can appear fluent and plausible while still containing subtle logical or arithmetic errors.

The proposed solution is **Generative Reward Modeling (GenRM)**, which recasts verification as standard next-token prediction. Instead of using a separate scalar classifier, GenRM asks whether a candidate answer is correct and uses the probability of the `Yes` token as the verification score. The stronger **GenRM-CoT** variant first generates a natural-language verification rationale and then predicts the final `Yes/No` judgment, allowing the verifier to reason explicitly about the candidate. Because verification is generative, the model can also sample multiple independent verification rationales and average their correctness probabilities, using additional test-time compute to improve robustness. The same next-token objective further allows solution generation and verification to be jointly trained in one model, enabling positive transfer between learning how to solve a problem and learning how to judge whether a proposed solution is correct.

Methodology

The central methodological contribution of Zhang et al. is to **recast reward modeling as ordinary autoregressive language modeling**. Conventional discriminative reward models map a problem–solution pair directly to a scalar correctness score. GenRM instead expresses correctness as text: the verifier is trained to produce either a `Yes` or `No` token, optionally after generating a natural-language verification rationale. Because verification remains inside the next-token prediction framework, the same pretrained LLM can be used for direct verification, explicit chain-of-thought verification, joint solution generation and verification, and inference-time scaling through repeated verification runs.

Throughout the method, let x denote the original problem, y a candidate solution, y^+ a correct candidate, y^- an incorrect candidate, and θ the parameters of the verifier. The instruction

$$I = \text{“Is the answer correct (Yes/No)?”}$$

is used to convert verification into a next-token prediction task.

Next-Token Training Objective. GenRM is trained using the same supervised fine-tuning objective used for ordinary language-model training. For a dataset

$$\mathcal{D} = \{(x, y)\},$$

the paper defines

$$\boxed{\mathcal{L}_{\text{SFT}}(\theta, \mathcal{D}) = -E_{(x,y)\sim\mathcal{D}}\left[\sum_{t=1}^{|y|} \log p_\theta\left(y_t \mid x, y_{<t}\right)\right]}$$

where $y = (y_1, \ldots, y_T)$ is the target output sequence, $y_{<t}$ denotes all target tokens preceding position t, and $p_\theta(y_t \mid x, y_{<t})$ is the probability assigned by the language model to the correct next token. Minimizing this negative log-likelihood increases the probability of the desired output sequence token by token.

One notation detail is worth noting: in the generic SFT equation, y denotes the **target response sequence**. In the later verification equations, the paper uses y specifically for the **candidate solution being verified**. The underlying optimization is nevertheless the same.

Direct Generative Verifier. The simplest GenRM variant performs verification without producing an intermediate critique. Given the problem x, candidate solution y, and verification instruction I, the model is trained to generate one correctness token.

The direct-verification dataset is

$$\mathcal{D}_{\text{Direct}} = \left\{ \left((x, y^+, I), \texttt{Yes} \right) \right\} \cup \left\{ \left((x, y^-, I), \texttt{No} \right) \right\}.$$

Thus, correct solutions are supervised with the target token `Yes`, whereas incorrect solutions are supervised with `No`. Importantly, this does not require a separate classification head; correctness is represented directly in the LLM vocabulary and learned through the same SFT objective used for ordinary text generation.

At inference time, GenRM does not merely take the discrete `Yes` or `No` prediction. Instead, the probability assigned to `Yes` is treated as the continuous reward:

$$\boxed{r_{\text{Direct}}(x, y) = p_\theta \left(\texttt{Yes} \mid x, y, I \right)}$$

where $r_{\text{Direct}}(x, y) \in [0, 1]$ is the direct verifier score. A value near 1 indicates strong confidence that the candidate is correct, while a value near 0 indicates the opposite. Using the probability rather than a hard binary prediction is essential for Best-of-N reranking because it allows several candidates that would all be classified as `Yes` to still be ordered according to verifier confidence.

This formulation is the first major conceptual change introduced by the paper:

$$\boxed{\text{scalar classification} \quad \longrightarrow \quad \text{probability of a generated correctness token.}}$$

Unified Solution Generation and Verification. A second component of GenRM is the observation that **solving a problem and verifying a proposed solution are closely related tasks**. A model that learns how correct solutions are constructed may acquire representations useful for identifying incorrect ones; conversely, learning to critique incorrect solutions may improve its ability to generate valid reasoning.

The authors therefore train the same model on both verification examples and correct solution-generation examples. Their unified objective is

$$\boxed{\mathcal{L}_{\text{GenRM}}(\theta, \mathcal{D}_{\text{verify}}) = \mathcal{L}_{\text{SFT}}(\theta, \mathcal{D}_{\text{verify}}) + \lambda \mathcal{L}_{\text{SFT}}(\theta, \mathcal{D}_{\text{correct}})}$$

where $\mathcal{D}_{\text{verify}}$ contains verification examples, $\mathcal{D}_{\text{correct}}$ contains problems paired with correct solution generations, and $\lambda > 0$ controls the contribution of solution-generation training.

When

$$\lambda = 0,$$

the model is trained only as a verifier. For

$$\lambda > 0,$$

generation and verification are jointly optimized. The authors interpret this as potential **positive transfer** between the two tasks. Their later ablations show that including some generation training improves verification, although too much generation data can reduce verifier performance, indicating that the mixture must remain balanced.

The importance of this objective is therefore not merely architectural. It allows one autoregressive model to learn two related conditional distributions:

$$x \longrightarrow \text{correct solution}$$

and

$$(x, y) \longrightarrow \text{correctness judgment.}$$

Because both are implemented through next-token prediction, they can share the same model parameters and training interface.

GenRM-CoT: Generative Verification with Explicit Reasoning. Direct GenRM produces a correctness token immediately. However, many reasoning errors cannot be reliably detected without first examining the intermediate logic of the candidate solution. GenRM-CoT therefore introduces an explicit **verification rationale** before the final judgment.

Let

$$I_{\text{CoT}} = \text{“Let's verify step by step.”}$$

be the instruction that initiates verification reasoning, and let

$$v_{\text{CoT}}$$

denote the generated verification rationale.

A CoT training example maps

$$(x, y, I_{\text{CoT}})$$

to a target sequence containing

$$(v_{\text{CoT}}, I, \texttt{Yes/No}).$$

For correct candidates, the final token is `Yes`; for incorrect candidates, it is `No`. The entire verification rationale and final decision are learned autoregressively through SFT.

At inference time, the rationale itself is generated from the verifier:

$$v_{\text{CoT}} \sim p_\theta\left(\cdot \mid x, y, I_{\text{CoT}}\right).$$

The symbol $\sim$ indicates that the rationale is **sampled from the model's conditional language distribution** rather than retrieved from a fixed reference.

After this rationale has been generated, the correctness reward is defined as

$$\boxed{r_{\text{CoT}}(x, y) = p_\theta\left(\texttt{Yes} \mid x, y, I_{\text{CoT}}, v_{\text{CoT}}, I\right)}$$

where the final probability is conditioned on the original problem, candidate solution, CoT instruction, generated verification rationale, and final Yes/No question.

The difference from direct GenRM is therefore

$$\text{Direct GenRM: } (x, y) \to P(\texttt{Yes})$$

whereas

$$\text{GenRM-CoT: } (x, y) \to v_{\text{CoT}} \to P(\texttt{Yes}).$$

The reward is still the probability of the `Yes` token, but that probability is now computed **after the model has explicitly reasoned about the candidate solution**.

Synthetic Verification Rationales. Training GenRM-CoT requires verification rationales. Human-written critiques would be expensive, so the authors investigate synthetic rationales generated by an LLM. For GSM8K, they improve rationale quality through **reference-guided grading**: during synthetic-data creation, the rationale-generating model receives the problem, the candidate solution, and an additional known-correct solution. This privileged reference helps the model identify subtle discrepancies in the candidate reasoning.

Crucially, the correct reference solution is used only to generate training rationales. It is not included as input when GenRM-CoT itself is fine-tuned or used at inference. Thus, the deployed verifier must eventually evaluate

$$(x, y)$$

without access to a gold solution.

This distinction is important because the reference-guided teacher improves the **training signal**, rather than making test-time verification artificially easier.

Inference-Time Scaling with Multiple Verification Rationales. A single generated verification rationale may itself contain an error. Because GenRM-CoT is generative, the authors exploit stochastic decoding to produce multiple independent verification rationales for the same candidate:

$$v_{\mathrm{CoT}}^{(1)}, v_{\mathrm{CoT}}^{(2)}, \ldots, v_{\mathrm{CoT}}^{(K)}.$$

Each rationale gives a potentially different correctness probability. Rather than trusting one reasoning trajectory, the method approximately marginalizes over these alternatives by averaging their `Yes` probabilities:

$$\boxed{r_{\mathrm{MajV@K}}(x, y) = \frac{1}{K} \sum_{i=1}^{K} p_\theta \left(\texttt{Yes} \mid x, y, I_{\mathrm{CoT}}, v_{\mathrm{CoT}}^{(i)}, I \right)}$$

with

$$v_{\mathrm{CoT}}^{(i)} \sim p_\theta \left(\cdot \mid x, y, I_{\mathrm{CoT}} \right).$$

Here, K is the number of independently sampled verification rationales, i indexes a particular verification run, and $v_{\mathrm{CoT}}^{(i)}$ is the rationale produced in run i.

The paper refers to this procedure as majority voting, although the equation technically performs an **average of soft `Yes` probabilities rather than a hard count of Yes/No decisions**. Averaging retains confidence information and reduces the influence of occasional erroneous verification trajectories. Unless otherwise stated, the experiments use

$$K = 32.$$

This gives GenRM-CoT an important property unavailable to a standard discriminative reward model: verification quality can be improved by allocating **additional inference-time computation**.

Relationship Between the Main Equations. The four central equations play different roles and should not be interpreted as independent methods.

$$\boxed{\begin{aligned} \text{Direct:} \quad & r_{\mathrm{Direct}}(x, y) = P(\texttt{Yes} \mid x, y, I) \\ \text{Unified Training:} \quad & \mathcal{L}_{\mathrm{GenRM}} = \mathcal{L}_{\mathrm{verify}} + \lambda \mathcal{L}_{\mathrm{generation}} \\ \text{CoT:} \quad & r_{\mathrm{CoT}}(x, y) = P(\texttt{Yes} \mid x, y, v_{\mathrm{CoT}}) \\ \text{MajV@K:} \quad & r_{\mathrm{MajV@K}}(x, y) = \frac{1}{K} \sum_{i=1}^{K} P(\texttt{Yes} \mid x, y, v_{\mathrm{CoT}}^{(i)}). \end{aligned}}$$

The unified objective is the **training objective**. The other equations specify progressively richer **verification scoring procedures**:

$$\boxed{\text{direct decision} \rightarrow \text{reason then decide} \rightarrow \text{reason repeatedly and aggregate.}}$$

This progression captures the paper's main methodological argument: once verification is represented as next-token prediction, the verifier can exploit capabilities normally associated with generative LLMs—reasoning, multitask training, sampling, and test-time compute—without introducing a separate discriminative reward head.

Worked Example: GSM8K Streaming-Service Error. The paper's streaming-service example provides a compact illustration of the complete GenRM-CoT mechanism. The problem states that Netflix costs \$10 per month, Hulu and Disney Plus each cost \$10, and bundling Hulu and Disney Plus gives a 20% discount. The candidate solution incorrectly computes the bundle as

$$10 \times 0.8 = 8$$

and consequently concludes that the total streaming cost is

$$10 + 8 = 18$$

and the savings relative to the \$60 cable plan are

$$60 - 18 = 42.$$

The candidate is wrong because the discount applies to the combined \$20 cost of Hulu and Disney Plus:

$$(10 + 10) \times 0.8 = 20 \times 0.8 = 16.$$

The correct total streaming cost is therefore

$$10 + 16 = 26,$$

giving savings of

$$60 - 26 = 34.$$

A discriminative reward model assigns this incorrect candidate a very high correctness score, whereas GenRM-CoT generates a verification rationale that explicitly identifies the ignored word "each" and reconstructs the correct bundle calculation.

For one verification trajectory $v_{\mathrm{CoT}}^{(i)}$, the verifier ultimately evaluates

$$p_\theta \left(\texttt{Yes} \mid x, y, I_{\mathrm{CoT}}, v_{\mathrm{CoT}}^{(i)}, I \right).$$

Repeating this process across multiple sampled rationales produces the final GenRM-CoT reward. In the paper's reported example, the aggregated score is approximately

$$\boxed{r_{\mathrm{MajV}}(x, y) = 0.0015}$$

indicating extremely low confidence that the candidate solution is correct. By contrast, the discriminative reward model assigns the same incorrect solution a score of approximately 0.999.

The significance of the example is therefore not merely that GenRM-CoT outputs the correct `No`. The generated verification rationale exposes the specific logical failure that drives that judgment, while repeated rationale sampling makes the final score less dependent on any single verification trajectory.

Core Methodological Takeaway. The method can be summarized compactly as

GenRM: Reward modeling is reformulated as next-token prediction.

Direct GenRM represents correctness through the probability of `Yes`; unified training jointly learns solution generation and verification; GenRM-CoT introduces an explicit generative reasoning stage; and repeated verification rationales convert additional test-time computation into a more robust verification score by averaging judgments across multiple independently generated reasoning paths. This sequence is the core technical contribution of the paper.

Training and Implementation

GenRM is implemented by fine-tuning pretrained **Gemma language models with the standard autoregressive next-token objective**, rather than attaching a separate reward head. Let θ denote the trainable parameters of the Gemma verifier and $\mathcal{D} = \{(x, y)\}$ an input–target dataset. For an output sequence $y = (y_1, \ldots, y_T)$, supervised fine-tuning minimizes

$$\boxed{\mathcal{L}_{\mathrm{SFT}}(\theta, \mathcal{D}) = -E_{(x,y)\sim\mathcal{D}} \left[\sum_{t=1}^{T} \log p_\theta(y_t \mid x, y_{<t}) \right]}$$

so the same language-model parameters are optimized to predict verification text and the final `Yes/No` token. GenRM additionally mixes verification supervision with correct solution generation:

$$\boxed{\mathcal{L}_{\mathrm{GenRM}} = \mathcal{L}_{\mathrm{SFT}}(\theta, \mathcal{D}_{\mathrm{verify}}) + \lambda \mathcal{L}_{\mathrm{SFT}}(\theta, \mathcal{D}_{\mathrm{correct}})}$$

where $\mathcal{D}_{\mathrm{verify}}$ contains direct verification targets or CoT verification rationales, $\mathcal{D}_{\mathrm{correct}}$ contains correct solution-generation examples, and λ controls their relative contribution. The paper uses $\lambda = \frac{1}{3}$ for the algorithmic tasks and $\lambda = \frac{1}{4}$ for GSM8K.

For each optimization step, the gradient is computed with respect to the verifier parameters,

$$g_t = \nabla_\theta \mathcal{L}_{\mathrm{GenRM}},$$

clipped so that

$$\|g_t\|_2 \leq 1.0,$$

and the trainable Gemma parameters are updated using Adam with the scheduled learning rate η_t:

$$\boxed{\theta_{t+1} = \text{Adam}\left(\theta_t, g_t, \eta_t, \text{weight decay} = 10^{-2}\right)}$$

The paper does not specify frozen transformer blocks or a partial-parameter training scheme; it describes the Gemma verifier as being fine-tuned directly. Adam is used with no dropout, a batch size of 64, 1000 warm-up steps, cosine learning-rate decay to 10% of the peak value, gradient clipping at 1.0, and training for 300,000 steps. The learning-rate sweep is

$$\{2 \times 10^{-6},\ 1 \times 10^{-6},\ 5 \times 10^{-7}\},$$

with 2×10^{-6} generally performing best. Checkpoints are selected using verifier accuracy on held-out validation examples.

Data and Model Configuration. For the **algorithmic reasoning experiments**, the verifier is **Gemma-2B**. Last Letter Concatenation is trained on input lengths $2, 3, 4$, producing roughly 50,000 verifier examples after deduplication; Word Sorting similarly uses shorter training lists and yields roughly 100,000 examples. Because these tasks are algorithmically checkable, their verification CoTs are generated programmatically, providing essentially noise-free rationale supervision.

For mathematical reasoning, the verifier models are **Gemma-2B, Gemma-7B, and Gemma-2 9B**. Candidate solutions for GSM8K are generated using **Gemini 1.0 Pro**, which is also used to construct synthetic verification rationales. GSM8K uses approximately 7.2K training problems, 128 validation problems, and 1.3K test problems. Fifty candidate solutions are generated for each training problem, after which at most 16 correct and 16 incorrect candidates are retained. Training batches are balanced to contain 50% correct and 50% incorrect examples.

Synthetic Verification Rationales. For GenRM-CoT on GSM8K, **Gemini 1.0 Pro** is prompted as a math teacher and receives the original problem, the candidate solution being graded, and a known-correct reference solution. This **reference-guided grading** is used only during synthetic-data construction; the reference solution is removed before GenRM-CoT training and is unavailable at inference. The target sequence contains the generated verification rationale followed by the final `Yes` or `No` decision.

Because synthetic critiques can themselves be incorrect, the authors generate multiple rationales and retain cases only when **more than half of the verification rationales agree with the correctness supplied by the answer checker**. The checker relies on string matching or symbolic checking with SymPy and may itself contain false positives or false negatives, so this agreement filter is intended to reduce noise rather than guarantee perfect labels.

Increasing both the number of candidate solutions per problem and the number of verification rationales per candidate improves GenRM-CoT performance. The authors suggest that multiple rationales reduce dependence on errors in any single synthetic critique, producing an effect analogous to ensembling during supervision.

Inference. During Best-of-N evaluation, the generator is kept fixed so that improvements reflect verifier quality rather than better candidate generation. For GSM8K, **Gemini 1.0 Pro** generates the candidate solutions, while the fine-tuned Gemma verifier evaluates them. For GenRM-CoT, the verifier samples K complete verification rationales for each candidate; the paper uses $K = 32$ unless otherwise specified. The probability of `Yes` after each rationale is averaged, and the candidate with the highest resulting score is selected.

Unified Implementation Sketch.

```
# Models used in the GSM8K setting
solution_generator = Gemini-1.0-Pro
rationale_teacher   = Gemini-1.0-Pro
verifier             = Gemma-{2B, 7B, Gemma-2-9B}

# ------------------------------------------------
# 1. Build verifier training data
# ------------------------------------------------

for each GSM8K training problem x:

    candidates = Gemini-1.0-Pro.generate(x, num_solutions=50)
```

```
    classify each candidate y using final-answer checking
    retain <=16 correct and <=16 incorrect solutions

    for each retained candidate y:

        reference = a solution known to reach the correct answer

        generate multiple synthetic verification rationales with
        Gemini-1.0-Pro using:
            problem x
            candidate y
            reference solution
            reference-guided grading prompt

        keep training cases only when
        >50% of rationale judgments agree with answer-checker label

        add:
            input  = [x, y, "Let's verify step by step."]
            target = [verification rationale,
                      "Is the answer correct (Yes/No)?",
                      Yes/No]

also add generation examples:
    input  = x
    target = correct solution y+

balance verifier data:
    50% correct
    50% incorrect


# ------------------------------------------------
# 2. Fine-tune Gemma verifier
# ------------------------------------------------

objective =
    SFT_loss(verification_data)
    + lambda * SFT_loss(correct_solution_data)

lambda = 1/4 for GSM8K
lambda = 1/3 for algorithmic tasks

optimizer       = Adam
learning_rates  = {2e-6, 1e-6, 5e-7}
typical best LR = 2e-6
weight_decay    = 1e-2
dropout         = 0
batch_size      = 64
warmup_steps    = 1000
gradient_clip   = 1.0
schedule        = cosine decay to 10% of peak LR
training_steps  = 300000

for each optimization step:
```

```
    compute GenRM loss
    backpropagate through Gemma parameters theta
    clip gradient norm to 1.0
    update theta with Adam

select checkpoint using held-out verification accuracy


# ------------------------------------------------
# 3. Best-of-N inference
# ------------------------------------------------

candidates = Gemini-1.0-Pro.generate(test_problem, N)

for each candidate y:

    for i = 1 ... K:       # K = 32 by default

        rationale_i =
            Gemma-verifier.generate(
                problem,
                y,
                "Let's verify step by step."
            )

        score_i =
            probability of next token being "Yes"
            after rationale_i

    verifier_score[y] = mean(score_1, ..., score_K)

return candidate with maximum verifier_score
```

The implementation is therefore notable less for a new neural architecture than for the combination of **pretrained Gemma fine-tuning, carefully constructed positive/negative verifier data, synthetic reference-guided CoT supervision, joint generation–verification training, and repeated verification sampling at test time**. The paper does not report enough information to determine some lower-level reproduction details such as the exact hardware configuration, random seeds, or default Adam β parameters; these should not be invented in a review.

Critical Research Review

What Actually Caused the Gains. The strongest improvements in GenRM do not come from a single component, but from the interaction of **generative verification, explicit CoT reasoning, unified generation–verification training, and additional inference-time sampling**. Direct GenRM is generally comparable to or slightly better than a discriminative reward model, while larger gains appear once verification CoT is introduced and multiple rationales are sampled. The ablations also show that adding correct-solution generation data improves verification, although excessive generation weight can hurt performance.

The quality of verification rationales also matters. Reference-guided synthetic rationales outperform unguided ones, and increasing the number of rationales per candidate improves both reward-model accuracy and Best-of-N performance. Thus, GenRM's advantage is not merely representing correctness using a `Yes/No` token; it comes from enabling the verifier to explicitly reason about candidate solutions and use additional test-time computation.

Strength of the Evidence. The experimental design provides reasonably strong evidence that the verifier itself drives the observed Best-of-N gains because the candidate generator is kept fixed when comparing LLM-as-a-Judge, DPO, discriminative RM, GenRM, and GenRM-CoT. This prevents improvements from being attributed simply to stronger candidate generation.

The paper reports gains across algorithmic reasoning, GSM8K, MATH500, and mathematical subsets of MMLU, including easy-to-hard transfer from grade-school mathematics to substantially harder problems. However, the evidence supports the

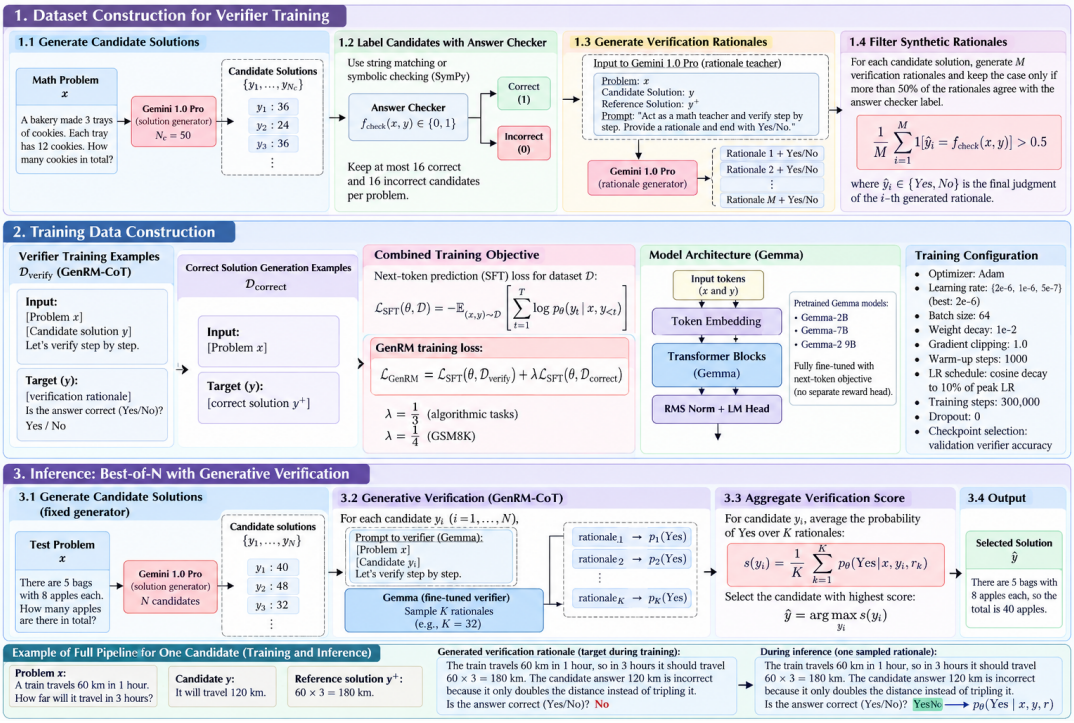


Figure 4: GenRM Generative Reward Modelling Pipeline

narrower claim that **generative verification is effective on the studied reasoning tasks**, rather than proving that it universally dominates discriminative or process-based reward models.

Limitations. **Inference cost.** GenRM-CoT generates verification text and, in its strongest setting, samples up to $K = 32$ rationales for each candidate. This creates a significant accuracy–compute trade-off compared with single-pass discriminative reward models or PRMs.

Dependence on synthetic rationale quality. For GSM8K, Gemini 1.0 Pro generates the verification rationales. The authors show that unguided rationales are noisy, and therefore use reference-guided grading and filtering. Even so, the supervision still depends on model-generated critiques and an imperfect answer checker.

Limited evaluation scope. The experiments focus primarily on mathematical and synthetic algorithmic reasoning. Broader applications such as coding, alignment, open-ended generation, and multimodal evaluation are not experimentally established in this work.

Partial reproducibility. The paper documents the main models, datasets, prompts, optimization settings, and training procedure, but does not provide every low-level detail required for exact reproduction, such as random seeds, hardware configuration, total compute, and some decoding settings.

Relationship to Earlier Verifier Work

GenRM can be understood as a further development of the verifier framework introduced in earlier reasoning work.

Cobbe et al. established the generate-and-rank paradigm: generate multiple candidate mathematical solutions, use a learned verifier to assign correctness scores, and select the highest-scoring candidate. GenRM preserves this Best-of-N structure but replaces the conventional scalar verifier with a generative next-token formulation.

Lightman et al. move from solution-level outcome supervision to **process supervision**, where individual reasoning steps receive explicit correctness labels. GenRM explores a different direction: rather than directly scoring each step, the verifier generates a **natural-language verification reasoning process** before producing its final correctness judgment.

The progression can therefore be summarized as

Outcome Verification → Process-Level Verification → Generative Reasoning for Verification

These approaches are complementary rather than mutually exclusive. The GenRM paper itself identifies combining generative verification with process-level supervision as a promising extension.

Final Assessment

The paper's central contribution is a change in how verification is represented:

$$\boxed{\text{Verification as Scalar Prediction} \quad \rightarrow \quad \text{Verification as Language Generation}}$$

This reformulation allows the verifier to reason before judging, share training with solution generation, and use additional inference-time computation through repeated verification sampling. The experimental evidence supports these advantages strongly within mathematical and algorithmic reasoning, especially under Best-of-N evaluation and easy-to-hard transfer.

The main costs are increased inference computation, dependence on synthetic rationale quality, and limited validation outside reasoning benchmarks. The overall GenRM procedure can be summarized as

$$\boxed{\text{Generate Candidate} \rightarrow \text{Reason About Candidate} \rightarrow \text{Judge via } P(\texttt{Yes}) \rightarrow \text{Repeat and Aggregate}}$$

In this sense, GenRM extends verifier research from learning only **whether** a solution is correct toward learning **how to reason about why** it is correct or incorrect before assigning a reward.