

The Climb Carves Wisdom Deeper Than the Summit: On the Noisy Rewards in Learning to Reason

Ang Lv¹ Ruobing Xie^{2*} Xingwu Sun^{2,3} Zhanhui Kang² Rui Yan^{1,4*}

¹ GSAI, Renmin University of China

² Large Language Model Department, Tencent

³ University of Macau

⁴ School of Computer Science, Wuhan University

{anglv, ruiyan}@ruc.edu.cn

{ruobingxie, sammsun, kegokang}@tencent.com

Abstract

Recent studies on post-training large language models (LLMs) for reasoning through reinforcement learning (RL) typically focus on tasks that can be accurately verified and rewarded, such as solving math problems. In contrast, our research investigates the impact of reward noise, a more practical consideration for real-world scenarios involving the post-training of LLMs using reward models. We found that LLMs demonstrate strong robustness to substantial reward noise. For example, manually flipping 40% of the reward function’s outputs in math tasks still allows a Qwen-2.5-7B model to achieve rapid convergence, improving its performance on math tasks from 5% to 72%, compared to the 75% accuracy achieved by a model trained with noiseless rewards. Surprisingly, by only rewarding the appearance of key reasoning phrases (namely reasoning pattern reward, RPR), such as “first, I need to”—without verifying the correctness of answers, the model achieved peak downstream performance (over 70% accuracy for Qwen-2.5-7B) comparable to models trained with strict correctness verification and accurate rewards. Recognizing the importance of the reasoning process over the final results, we combined RPR with noisy reward models. RPR helped calibrate the noisy reward models, mitigating potential false negatives and enhancing the LLM’s performance on open-ended tasks. These findings suggest the importance of improving models’ foundational abilities during the pre-training phase while providing insights for advancing post-training techniques. Our code and scripts are available at <https://github.com/trestad/Noisy-Rewards-in-Learning-to-Reason>.

1 Introduction

Reinforcement learning (RL) applied to post-training large language models (LLMs) has led to significant advancements in enhancing their thinking and reasoning abilities [6, 29], resulting in improved performance on many challenging downstream tasks. Most current research focuses on math tasks [13, 23, 33, 10, 3], as these can be easily verified as correct or incorrect by simple rule-based reward functions. However, in many real-world applications, such as preference alignment [22, 38] and open question-answering [14, 21], responses cannot be easily quantified with simple rule-based functions and instead require evaluation by neural reward models. These models, being imperfect, often introduce noise even resulting in opposite rewards. In this study, we studied scenarios in which

*Corresponding authors

the reward, whether derived from a neural reward model or a rule-based function, contains noise, aiming to gain a more practical understanding of noisy rewards in teaching LLMs to reason.

We made an unexpected discovery: despite the presence of substantial noise in the rewards, the model can still be effectively trained and achieve fast convergence. For example, when training on math problems and introducing noise by randomly flipping 40% of the reward function’s outputs (i.e., assigning positive rewards for incorrect answers), a Qwen-2.5-7B [32] model still improved its MATH-500 [12] score from an initial 5% to a surprising peak of 72.02%, close to the 75.85% achieved using a noiseless reward function. Training collapses when the flip rate reaches 50%, where the reward becomes entirely random. The model’s ability to tolerate substantial noisy rewards suggests that, although outputs with incorrect answers are mistakenly rewarded, they still exhibit valuable logical reasoning processes. These reasoning patterns are also valuable. If this were not the case, a high frequency of incorrect rewards would likely hinder performance or, at the very least, slow convergence. Thus, we hypothesize that the effectiveness of RL in enhancing LLMs primarily stems from the exploration of appropriate reasoning patterns during rollouts. Through this exploration, the model can more effectively leverage its pretrained knowledge by adopting strong reasoning patterns, which increases the likelihood of arriving at the correct answer.²

To support this hypothesis, we conducted another experiment in which the model was rewarded whenever key reasoning phrases, such as “first, I need to” or “let me first,” appeared in the outputs, without verifying the correctness of the final answer. We name this strategy as Reasoning Pattern Reward (RPR). Using only RPR, the model achieved peak task performance (70.21% for Qwen-2.5-7B) comparable to the performance achieved when the correctness of the solution was strictly verified. This provides strong evidence suggesting that LLMs can reason through RL because they have already learned to reason during pretraining, as no correctness supervision signals were given, meaning no new knowledge was learned.

LLMs’ robustness to noisy rewards extends to open-ended NLP tasks as well. We conducted experiments using the NVIDIA HelpSteer3 dataset [30], which consists of a broad range of challenging open-ended questions requiring AI assistance. We varied reward model accuracy by adjusting training set size and found that LLMs trained with a 75% accurate reward model performed comparably to those using our best model (85% accuracy).

These insights motivate our proposal of a simple yet effective method to improve LLM performance on open-ended NLP tasks post-trained with noisy reward models. We use RPR to calibrate the reward models by compensating for potential false negative signals, resulting in up to a 30% net win rate over LLMs post-trained with vanilla reward models. RPR-calibrated reward models also enable smaller models, such as Qwen-2.5-3B, to demonstrate strong reasoning capabilities on complex open-ended tasks, where vanilla reward models lead to training collapse.

In summary, we provide some insights into training LLMs to reason via RL with noisy rewards:

1. We demonstrate that LLMs with strong inherent reasoning abilities are surprisingly robust to reward noise.

²A premise of this hypothesis is that the model must have acquired substantial reasoning capabilities during pretraining. As we show later, Qwen significantly outperforms Llama [19] not only in downstream performance after RL but also in robustness to noisy rewards—a difference that aligns with Llama’s widely recognized weakness in reasoning [23, 33, 10].

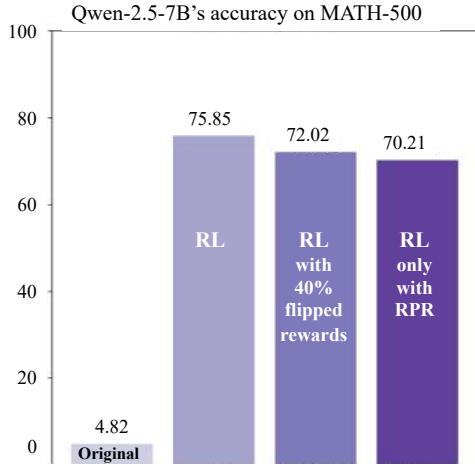


Figure 1: All of (1) standard RL, (2) RL with 40% of the rewards manually flipped to the opposite, and (3) RL with only Reasoning Pattern Rewards (RPR) (i.e., rewards are given whenever key reasoning phrases appear, without verifying the final answer)—can improve Qwen-2.5-7B’s accuracy on MATH-500 from an initial 5% to over 70%. The performance gap between these three setups is minimal compared to the overall improvements.

2. We present direct evidence showing that after RL, models’ enhanced performance on challenging tasks primarily stems from adapted output patterns, which are more likely to lead to correct answers, rather than from learning much new knowledge during RL.
3. We propose a simple yet effective method to calibrate noisy reward models by rewarding reasoning patterns, leading to improved LLM performance on open-ended NLP tasks and unlocking smaller models’ reasoning capabilities.

2 Noisy verification rewards in math tasks

Mathematics is one of the most commonly studied domains in LLM reasoning, due to its straightforward rule-based reward and evaluation. To explore the RL performance under noisy rewards, we begin by manually introducing noise to RL rewards in math tasks. While math tasks are typically assumed to be noiseless and verification is considered accurate, our noisy scenario is practical for two reasons: (1) false negative rewards are common in math tasks, where a correct answer may be flagged as incorrect due to formatting issues, and (2) many proof-based questions in math tasks are difficult to verify accurately.

2.1 Settings

Training. Most of the training setups follow the approach in [13], which provides a simplified framework designed to help LLMs learn to reason. The experiments are based on VeRL [28] framework by Volcengine.

A conversation between User and Assistant. The user asks a question, and the Assistant solves it. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. The reasoning process and answer are enclosed within `<think>` `</think>` and `<answer>` `</answer>` tags, respectively, i.e., `<think>` reasoning process here `</think>` `<answer>` answer here `</answer>`. User: You must put your answer inside `<answer>` `</answer>` tags, i.e., `<answer>` answer here `</answer>`. And your final answer will be extracted automatically by the `\boxed{}` tag. `{question}` Assistant: `<think>`

Figure 2: The prompt used in math training, where the “question” placeholder will be replaced with a specific question.

Specifically: the dataset includes 57K high-quality, source-mixture math problems spanning various difficulty levels. The prompts used in these tasks are shown in Figure 2. The outputs are extracted from the box tag, normalized for format (e.g., converting fractions to decimals and translating LaTeX answers to plain text), and compared to the ground truth. In the absence of manual reward noise, the model is given a reward of 1 if the output matches the ground truth and 0 otherwise. For training, we employ vanilla PPO [27] with GAE [26], using $\lambda = 1$ and $\gamma = 1$, with no KL-regularization. The training batch size is 128, with a maximum response length of 4096. The learning rate for the actor is 10^{-6} , and for the critic, it is 5×10^{-6} . To ensure stable training, we apply critic warmups for 20 steps, initially training the critic before training the actor model. We set the rollout number to 4. The primary focus of the main text is the Qwen-2.5-7B [32], which has demonstrated strong reasoning potential [10, 33], while experimental results on other models are provided in the appendix.

Evaluation. We use three datasets—MATH-500 [12], GPQA [25], and AIME 2024 [2]—to assess the model’s reasoning ability on challenging tasks. We report the Pass@1 accuracy dynamics throughout the training.

Random reward flip. We train the model by randomly flipping the reward with a probability p , where a reward of 1 is transformed to 0, and vice versa. This flip is applied on a question-wise basis, meaning that if a reward flip occurs for a given question, the rewards for all rollout outputs corresponding to that question will be flipped. Note that flipping rewards on an output-wise basis does not effectively introduce noise. For instance, when an LLM generates multiple correct outputs, some of which are rewarded correctly while others are not, it results in a sparse reward distribution. Such sparsity can only slow convergence and has minimal impact on the model’s final performance.

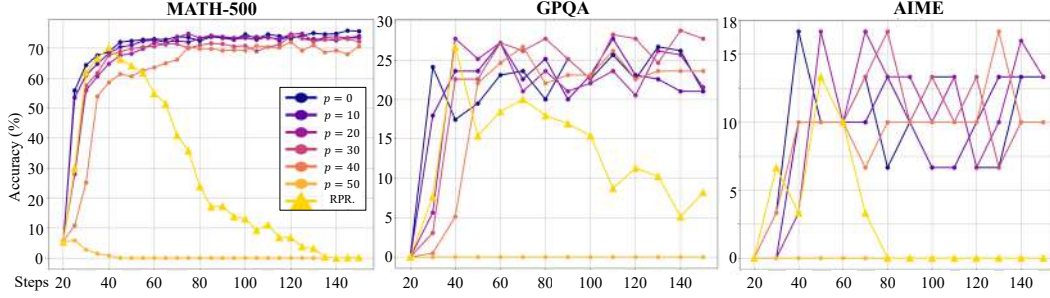


Figure 3: Accuracy on three test sets during training. Due to critic warmup, the actor model is not updated during the first 20 steps; thus, the x-axis begins at step 20.

2.2 Experiments

Experiment 1. We train the model with the probability p of noise increasing from 0% to 50%, with intervals of 10%, corresponding to increasingly random reward flips. The results are shown in Figure 3. We only display the first 150 steps, as the performance has already plateaued.

In MATH-500, it is evident that even with a high flip rate of 40%, the final performance is only slightly lower than that of the model trained with no noise (peak score of 72.02% versus 75.85%). For lower noise levels, the final performance is comparable, and the convergence also occurs at a similar rate. Only when we increase p to 50%, leading the reward entirely random, the training collapse.

In the other two tasks, similar trends are observed, though fluctuations are more pronounced. In particular, for $p=20\%$, the peak performance is even higher than when $p=0$. We attribute such fluctuations to the inherent difficulty of the datasets, which cause larger variability in model performance. In Appendix A, we also demonstrate that Qwen-2.5-3B exhibits strong robustness to noise, comparable to the 7B models. In contrast, Llama-3.1-8B shows considerably weaker reasoning capabilities and underperforms even in noiseless settings. This underperformance aligns with prior findings that highlight the limited foundational reasoning abilities and RL improvements in Llama models [23, 33, 10]. These limitations result in poor noise tolerance.

Based on the overall performance of the three models analyzed in this study, we carefully formulate Takeaway 1 with an important caveat: while large language models can exhibit robustness to reward noise, this property holds primarily for models with strong reasoning capabilities. This is both a practical and necessary condition—models lacking foundational reasoning abilities, such as Llama-3.1-8B, are unlikely to be effectively used in real-world applications, regardless of the level of reward noise, and are therefore of limited interest for our study.

Takeaway 1. For models with strong reasoning potential, even with significant opposite noise in the *verification* rewards, the model can still be effectively trained during RL.

Experiment 2. Given the surprising result from Experiment 1, the key question is why assigning a reward of 1 to outputs with genuinely incorrect answers does not have a significant detrimental effect. Since the answer is incorrect, we hypothesize that the reasoning process itself might still be valuable and worth rewarding. If this were not the case, it would be challenging to achieve performance comparable to noiseless setups, and convergence would likely be slower at the very least.

To test this hypothesis, we conducted the experiment 2: We first identified n high-frequency phrases that imply certain desired reasoning patterns, such as “*We know that*” and “*First I need to,*” in the outputs of a model trained with $p=0$.³ Next, we designed a rule-based reward function: instead of verifying the correctness of the answer, the model would receive a reward of value r each time a pre-identified reasoning phrase appeared in the output. The total reward is clipped to 1, creating a simple keyword-matching reward that ranged from 0 to 1. We name this strategy as **Reasoning Pattern Reward (RPR)**.

³These phrases also frequently appear in model outputs trained with higher levels of noise.

Figure 4 illustrates how RPR works, and an example code is provided in Figure 9. To prevent the model from hacking the reward by outputting repeated reasoning phrases (e.g., “We know that We know that...”), a repetition penalty [33] is used. Since our goal was to validate the hypothesis rather than achieving stable state-of-the-art accuracies, n was somewhat arbitrarily set to 40 without value tuning, and for simplicity, we set $r = 1/n = 0.025$.

The results are presented in Figure 3 for ease of comparison. Remarkably, even *without* verifying the correctness of reasoning during the training process, the model demonstrates strong reasoning capabilities in the early stages, achieving a peak performance of 70.21% on the MATH-500 task. The peak performance on other tasks also shows a minimal gap compared to models trained without noise. However, as training progresses, the performance eventually declines. Upon analyzing the outputs, we found that this decline was due to overthinking. Specifically, after reasoning through a few steps, the model revisits its prior thoughts, and this reasoning-then-revisiting cycle continues for too many iterations, resulting in an excessively long chain of reasoning that exceeds the context limit and is truncated before the final answer can be generated. The model produces these long reasoning steps from multiple viewpoints, effectively “escaping” the repetition penalty. An example of this is shown in Figure 10, where the model has already arrived at the correct answer but continues to reason, preventing the extraction of the final answer.

The lack of answer verification and supervision, combined with strong peak performance, suggests that the model does not require substantial new knowledge through supervision to final answers in RL post-training. Instead, most of the improvements in solving challenging tasks were already learned during pretraining and are activated by RL through rewarding effective reasoning patterns that can lead to correct answers.

This experiment provides direct evidence illustrating the role of RL in LLMs during post-training.

Takeaway 2. Training LLMs solely based on reasoning pattern rewards, without any correctness check, can develop powerful, though transient, reasoning abilities. This is a direct evidence that LLMs do not require much new knowledge during RL; instead, RL explores outputs that are likely to lead to correct answers and reinforces those reasoning patterns.

Remark. Key reasoning phrases we collected are broadly applicable across tasks. As shown in Experiment 4 (Section 3.3), RPR patterns from math tasks remain effective in diverse, open-ended domains. While there may be other effective keywords or even implicit hidden states [11] that trigger reasoning, pursuing a more refined design is not the focus of this study.

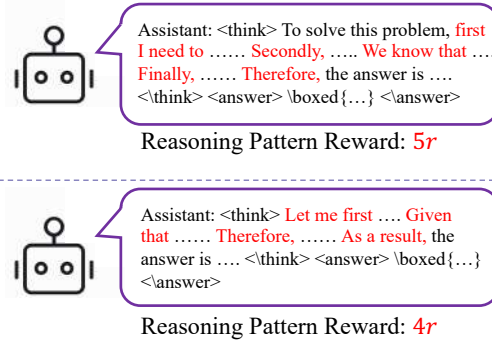


Figure 4: An illustration of how the reasoning pattern reward works through two example outputs. Suppose the red text represents high-frequency phrases that we have pre-identified as indicating key reasoning processes. In the first output, five key phrases are present, so the reward is $5r$. Similarly, the second output contains four key phrases, so the reward is $4r$. We do not verify the correctness of the answer.

3 Noisy reward models in open NLP tasks

Now we turn to the open NLP tasks requiring reward models (RMs). Different from manually flipping rewards in math tasks, the noise level in open NLP tasks can be approximately reflected in the varying evaluation accuracies of RMs. We first introduce the data, RM training details, and then introduce experiments with noisy RL rewards and corresponding findings.

{Chat history} User: I present a question, and you, the assistant, first thinks about the reasoning process in the mind and then provides the user with the answer. Enclose the reasoning within `<think>...</think>` tags and the final answer — which should also summarize the reasoning — within `<answer>...</answer>` tags. For example: `<think>Reasoning process here</think> <answer>Answer with summary of reasoning here</answer>`. Now, here is my question: **{question}** Assistant: `<think>`

Figure 6: The prompt used in the HelpSteer3 task, where the “question” and “chat history” placeholders are filled accordingly.

3.1 Preliminaries: Training reward models with varying accuracy

Dataset. We use the NVIDIA HelpSteer3 [30] dataset, which contains 40.5K multi-domain open-ended questions that require helpful assistance. Each question is paired with two responses, evaluated by multiple annotators for helpfulness, categorized into seven fine-grained levels. There is also a chat history preceding the current question, providing context for the question. The dataset is split into a training set of 38.5K samples and a validation set of 2K samples.

Training. Our reward model is built on a Qwen-2.5-7B model with an added prediction head. We simplify the original seven-level helpfulness scale into a binary classification task: the more helpful response in each pair is labeled as 1, and the less helpful one as 0. For each response, we concatenate it with the chat history as the input to the reward model. The prediction head produces a scalar output s , and we optimize the model using the MSE between s and the corresponding binary label [16, 35, 7]. The model learns to predict the absolute helpfulness, facilitating further RL, instead of using contrastive learning to compare the relative helpfulness of paired responses. The learning rate is 10^{-6} , and the RM is trained for 25,000 steps.

The evaluation accuracy dynamics of RMs are shown in Figure 5. The best-performing model achieved an evaluation accuracy of 85%. Different RM models with varying accuracies are used in subsequent experiments to simulate the scenarios with different levels of reward noises similarly in practical usages.

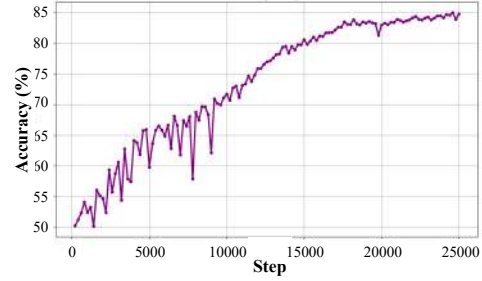


Figure 5: Reward model’s accuracy across the training. Checkpoints at specific steps are used for RL experiments.

3.2 Learning to reason using reward models of varying accuracy

Training. The hyperparameters used in this section basically follow those employed in previous math experiments. Training is conducted with Qwen-2.5-7B on the HelpSteer3 dataset, lasting a total of 200 steps. Figure 6 shows the prompt template, which instructs the model to first carefully consider how to provide useful assistance. It then asks the model to summarize its reasoning and present the final response within the `<answer>` tag. Importantly, the RM only evaluates the text within the `<answer>` tag, not the entire output. This approach ensures that the reward pipeline aligns with the one used in the mathematical experiments.

Evaluation. Evaluating open-ended tasks during training presents a much greater challenge than evaluating math problems, due to the lack of objective criteria and the absence of reliable, efficient evaluators. Because our most accurate reward model (RM) is used during training, it cannot be employed for evaluation at test time, as LLMs may learn to hack its preferences. It is also prohibitive to ask more advanced LLMs like ChatGPT or human evaluators to frequently evaluate models during training. As a result, we perform evaluation only after training, using a subset of 200 samples from the evaluation set, assessed by both GPT-4o and human evaluators. Specifically, we compare two models by having GPT-4o and human evaluators assess their responses to the same question. Only text in `<answer>` tags is used for evaluation.

The prompt used for GPT’s evaluation is shown in Figure 16. The evaluation considers factors including helpfulness, informativeness, reasoning, and coverage of user needs. To avoid bias from positional preferences [17, 5, 36] in language models, ChatGPT-4o evaluates an output pair twice for the same question, each time with a different order. A model’s response may result in a win, loss, or tie relative to the other model’s response, with the results presented in pie charts. In the main text, we report GPT evaluation scores, as they are more reproducible for the community. Details on human evaluation—guidelines, results, and inter-evaluator agreement measured by Fleiss’ Kappa [8]—are provided in Appendix B. There, we show that human evaluation aligns with GPT assessments, with evaluators demonstrating moderate to substantial agreement.

Experiment 3. We compare the performance of the Qwen-2.5-7B model trained with reward models (RMs) of varying accuracies: 85%, 75%, and 65%. The results are presented in Figure 7. The 85%-accurate RM yields only a modest 4% net win-rate advantage over the 75%-accurate RM, suggesting that their performances are similar.

Notably, the LLM trained using the 65%-accurate RM shows a significant decline in downstream performance. This decline can be attributed to multiple factors:

First, while one might expect the 85% RM to be just 1.31 times (i.e., $85/65$) better than the 65% RM based on raw accuracy, the actual difference in the number of misclassified labels is more than double (35% vs. 15%). This nonlinear increase in noise significantly impacts the quality of the training signal.

Second, beyond accuracy, the magnitude and distribution of reward scores are also critical. In contrast to domains like math problem solving—where rewards are typically binary (e.g., 0 or 1)—RMs, especially less accurate ones, tend to produce scores clustered around 0.5, even when they make correct classifications. This clustering reflects underlying model uncertainty. This effect is exhibited in the reduced variance of reward outputs from lower-accuracy RMs: on a validation set, the score variances are 0.1937, 0.1161, and 0.0672 for the 85%, 75%, and 65%-accurate RMs, respectively. A more accurate RM pushes scores further from the decision boundary, helping to avoid both over- and under-estimated rewards. These observations align with findings by [24], who emphasized that higher variance is also a key factor in RM effectiveness. In summary, both the lower accuracy and lower variance of the 65%-accurate RM likely contribute to its weak downstream performance. However, disentangling the individual effects of these factors remains a challenge, as training RMs with both targeted accuracy and targeted variance for clean ablations is difficult.

Nonetheless, our results demonstrate that the Qwen-2.5-7B model performs comparably when trained with reward models that are 75% and 85% accurate, indicating a degree of robustness to reward noise—though this robustness is less pronounced than what has been observed in mathematical tasks.

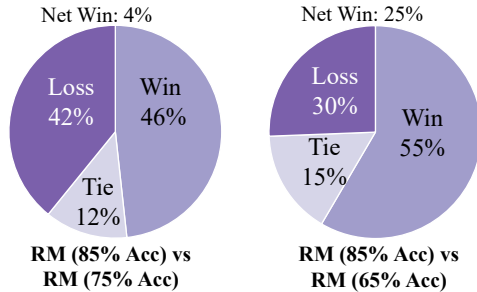


Figure 7: Qwen-2.5-7B trained with an 85%-accurate RM performs similarly to using a 75%-accurate RM, but significantly better than using a 65%-accurate RM. The “Net Win” refers to the performance advantage of the former RM over the latter.

Takeaway 3. While reward noise in neural reward models arises from multiple factors, the robustness to such noise observed in mathematical tasks persists—albeit to a different extent—in open-ended tasks.

We now understand that effective reasoning does not necessarily require using reward models (RMs) with the highest possible accuracy. This insight may offer some relief for real-world applications, where researchers often worry that their RMs are not sufficiently accurate. However, as demonstrated by our RM with 65% accuracy, some noisy RMs are indeed inadequate for practical use as the sole source of reward. Recognizing the importance of reasoning patterns, we propose a method for calibrating RMs with RPR (Section 2). This approach overcomes the performance ceiling imposed by the limitations of the reward models at hand.

3.3 Calibrating noisy RMs with reasoning pattern reward

Method. Considering that (1) it is impossible to train a perfect RM, and (2) exploring effective reasoning patterns is crucial for LLMs, we wonder whether reasoning pattern reward (RPR, Section 2) could help calibrate noisy RMs and thus obtain better performance. There are two potential ways to calibrating noisy rewards based on the value of reasoning patterns:

1. Compensatory reward for underestimated responses. When an RM produces false negatives, assigning a low score to an “objectively” good response, we give it some compensation.⁴ We assume that responses that display better reasoning patterns are likely to be closer to the “objectively” good ones. Therefore, we reuse the RPR as the compensation reward.

2. Discounting for overestimated responses. Conversely, when an RM provides false positive results, that is, it incorrectly assigns a high score to a “objectively” poor response, we can apply a discount to RM scores. However, this situation is more complex than discounting false negatives. The main challenge is determining the appropriate discount factor. For instance, if a response receives a full score but lacks key reasoning phrases, should its reward be near zero? Setting it too low could overemphasize reasoning pattern rewards, leading to overthinking and performance collapse, as discussed in Section 2. This remains an open research question: how can we effectively calibrate an RM when the noisy reward is a false positive?

Given these considerations, we only introduce the first method to calibrate the RM model: When the RM outputs a low score (as determined by a threshold τ), we calculate an RPR score only for the thought text (enclosed in `<think>` tags), while text in `<answer>` tags is not considered. This RPR score is added to the RM output, scaled by a weight α . This calibration incurs no additional time or memory costs.

Note that in our approach, RPR compensates not only for false negatives but also for true negatives. This is not problematic, as we demonstrated in Section 2 that true negative responses still contain valuable reasoning patterns and are therefore worth rewarding. Another potential concern is that using RPR as the sole reward signal might lead to performance collapse (Figure 3). However, when RPR is used as an auxiliary signal rather than the sole reward, LLMs are trained effectively without such collapse in the following experiments.

Experiment 4. We use reward models with accuracies of 65% and 85% to conduct several comparisons: (1) Calibrating the RMs using RPR, applying it to post-training Qwen models, and comparing their performance with models trained solely with the original RMs. (2) Comparing a Qwen model trained with a 65%-accurate RM calibrated by RPR to a model trained with an 85%-accurate RM. We set the threshold τ to 0.5 and α to 0.1. The choice of α is discussed in Appendix E. The results in Figure 8 demonstrate the effectiveness of RPR calibration:

1. The calibrated 65%-accurate reward model lags only 8% behind the 85%-accurate model—an improvement from an initial 25% gap, highlighting the substantial gains achieved through calibration.
2. Calibrating noisy RMs boosts downstream LLM performance, outperforming LLMs trained with original RMs. Even the 85%-accurate RM continues to improve after RPR calibration. RPR calibration addresses the limitations of RMs at hand.

Notably, the improvements observed are not due to an increase in reward score variance (0.1889 and 0.0653 for the 85% and 65%-accurate RMs post-calibration), as variance actually decreases

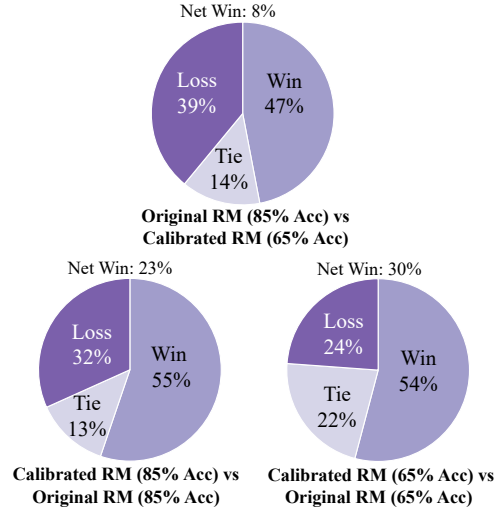


Figure 8: Reward noise calibration effectively enhances downstream performance.

⁴We acknowledge that there are no objective rules in this task; By “objectively,” we refer to whether rewarding the response eventually improves performance on the test set. If it does, the response should be considered good, at least from a deep learning perspective.

slightly. We provide output examples from models trained with both the original and calibrated RMs in Appendix C. Furthermore, in Appendix A, we show that the Qwen-2.5-3B model, which fails to emerge strong reasoning capabilities under the original RMs, demonstrates such abilities when trained with calibrated RMs. This suggests that reasoning pattern rewards not only raise the performance ceiling constrained by reward models in post-trained language models, but also lower the requirements for pre-trained models, enabling less capable models to exhibit reasoning behavior.

Takeaway 4. RPR can calibrate noisy reward models, particularly when they provide potentially false negative rewards. Its downstream success highlights the importance of strong reasoning patterns in improving reasoning ability.

Remark. While our RPR is straightforward and could potentially benefit from a more refined design, our primary goal is to demonstrate the importance of reasoning patterns and their effectiveness in real-world scenarios. For this purpose, the current simple RPR is sufficient to convey both points through Experiment 4.

4 Related works

Reward model accuracy. An accurate reward model was considered crucial for successful RL [9, 15, 18, 37]. Even in math tasks where rewards are calculated by verification functions, Yeo et al. [33] proposed that it is beneficial to refine the reward function with a fine-grained approach for accurately evaluating math answers, considering factors such as output length, correctness, and repetition. However, Chen et al. [4] found that more accurate reward models do not necessarily lead to stronger LLMs in downstream tasks. Razin et al. [24] argues that high reward variance is also important for making the reward model a good teacher. Additionally, Wen et al. [31] suggested that relying solely on accuracy does not fully capture the impact of reward models on policy optimization.

Studying the accuracy of reward models from a noise perspective offers some new insights. We argue that an accurate reward model is not always necessary in practice, though calibration to noisy rewards improves evaluation. We provide the first evidence of LLMs’ robustness to significant reward noise.

The role of RL in post-training LLMs. This paper aligns with recent studies suggesting that pre-trained models already possess the fundamental reasoning abilities needed for complex tasks. Yeo et al. [33] found that pre-training data often includes long chain-of-thought patterns, establishing a foundation for reasoning. Similarly, Yue et al. [34] noted that base models can perform similarly to RL-post-trained models after multiple attempts at difficult tasks. Gandhi et al. [10] showed that Qwen models outperform Llama [19] models in downstream tasks post-RL, with Qwen models exhibiting natural reasoning behavior. Prior works [1] demonstrated that reasoning can emerge during pre-training, with models using a reasoning trigger token like “wait” to activate chain-of-thoughts and arrive at the correct answer.

We provide strong evidence that models can achieve peak performance, comparable to those trained with strict verification, by rewarding key reasoning patterns instead of requiring correctness verification. While RL post-training has seen significant progress, our findings highlight the continued importance of pre-training in building advanced LLMs. From a post-training perspective, this also explains why a small amount of high-quality data [20] can enhance reasoning abilities, as the foundational capabilities are already present and need effective triggers.

5 Conclusions

We studied the reward noise, a practical consideration for real-world post-training of LLMs. Our findings show that LLMs are highly robust to significant reward noise. Surprisingly, when trained solely with reasoning pattern rewards (RPR), the model achieved peak downstream performance on par with models trained with strict correctness verification and accurate rewards. Recognizing the importance of reasoning processes over final answers, we use RPR to calibrate noisy reward models. RPR reduces false negative rewards and improves LLM performance on open-ended tasks. In the future, enhancing foundational ability during pre-training continues to be promising, and our findings also provide insights for improving post-training techniques.

References

- [1] Essential AI, :, Darsh J Shah, Peter Rushton, Somanshu Singla, Mohit Parmar, Kurt Smith, Yash Vanjani, Ashish Vaswani, Adarsh Chaluvvaraju, Andrew Hojel, Andrew Ma, Anil Thomas, Anthony Polloreno, Ashish Tanwer, Burhan Drak Sibai, Divya S Mansingka, Divya Shivaprasad, Ishaan Shah, Karl Stratos, Khoi Nguyen, Michael Callahan, Michael Pust, Mrinal Iyer, Philip Monk, Platon Mazarakis, Ritvik Kapila, Saurabh Srivastava, and Tim Romanski. Rethinking reflection in pre-training, 2025.
- [2] Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions, February 2025.
- [3] Xingyu Chen, Jiahao Xu, Tian Liang, Zhiwei He, Jianhui Pang, Dian Yu, Linfeng Song, Qiuzhi Liu, Mengfei Zhou, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. Do not think that much for $2+3=?$ on the overthinking of o1-like llms, 2025.
- [4] Yanjun Chen, Dawei Zhu, Yirong Sun, Xinghao Chen, Wei Zhang, and Xiaoyu Shen. The accuracy paradox in RLHF: When better reward models don’t yield better language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 2980–2989, Miami, Florida, USA, November 2024. Association for Computational Linguistics.
- [5] Yuhao Chen, Ang Lv, Ting-En Lin, Changyu Chen, Yuchuan Wu, Fei Huang, Yongbin Li, and Rui Yan. Fortify the shortest stave in attention: Enhancing context awareness of large language models for effective tool use. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11160–11174, Bangkok, Thailand, August 2024. Association for Computational Linguistics.
- [6] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.
- [7] Yanchen Deng, Chaojie Wang, Zhiyi Lyu, Jujie He, Liang Zeng, Shuicheng YAN, and Bo An. Q*: Improving multi-step reasoning for LLMs with deliberative planning, 2024.
- [8] J.L. Fleiss et al. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.
- [9] Evan Frick, Tianle Li, Connor Chen, Wei-Lin Chiang, Anastasios N. Angelopoulos, Jiantao Jiao, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. How to evaluate reward models for rlhf, 2024.
- [10] Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D. Goodman. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars, 2025.
- [11] Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space, 2024.
- [12] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.
- [13] Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model, 2025.
- [14] Natasha Jaques, Judy Hanwen Shen, Asma Ghandeharioun, Craig Ferguson, Agata Lapedriza, Noah Jones, Shixiang Gu, and Rosalind Picard. Human-centric dialog training via offline reinforcement learning. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3985–4003, Online, November 2020. Association for Computational Linguistics.

- [15] Nathan Lambert, Valentina Pyatkin, Jacob Morrison, Lester James Validad Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. RewardBench: Evaluating reward models for language modeling. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1755–1797, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics.
- [16] Chris Yuhao Liu, Liang Zeng, Jiakai Liu, Rui Yan, Jujie He, Chaojie Wang, Shuicheng Yan, Yang Liu, and Yahui Zhou. Skywork-reward: Bag of tricks for reward modeling in llms, 2024.
- [17] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.
- [18] Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. RM-bench: Benchmarking reward models of language models with subtlety and style. In *The Thirteenth International Conference on Learning Representations*, 2025.
- [19] AI @ Meta Llama Team. The llama 3 herd of models, 2024.
- [20] Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling, 2025.
- [21] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2022.
- [22] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS ’22, Red Hook, NY, USA, 2022. Curran Associates Inc.
- [23] Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. Tinyzero. <https://github.com/Jiayi-Pan/TinyZero>, 2025. Accessed: 2025-01-24.
- [24] Noam Razin, Zixuan Wang, Hubert Strauss, Stanley Wei, Jason D. Lee, and Sanjeev Arora. What makes a reward model a good teacher? an optimization perspective, 2025.
- [25] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. Gpqa: A graduate-level google-proof q&a benchmark, 2023.
- [26] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.
- [27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.
- [28] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.
- [29] Kimi Team. Kimi k1.5: Scaling reinforcement learning with llms, 2025.
- [30] Zhilin Wang, Jiaqi Zeng, Olivier Delalleau, Daniel Egert, Ellie Evans, Hoo-Chang Shin, Felipe Soares, Yi Dong, and Oleksii Kuchaiev. Dedicated feedback and edit models empower inference-time scaling for open-ended general-domain tasks, 2025.
- [31] Xueru Wen, Jie Lou, Yaojie Lu, Hongyu Lin, XingYu, Xinyu Lu, Ben He, Xianpei Han, Debing Zhang, and Le Sun. Rethinking reward model evaluation: Are we barking up the wrong tree? In *The Thirteenth International Conference on Learning Representations*, 2025.

- [32] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- [33] Edward Yeo, Yuxuan Tong, Morry Niu, Graham Neubig, and Xiang Yue. Demystifying long chain-of-thought reasoning in llms, 2025.
- [34] Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model?, 2025.
- [35] Dan Zhang, Sining Zhou, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. ReST-MCTS*: LLM self-training via process reward guided tree search. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.
- [36] Zhenyu Zhang, Runjin Chen, Shiwei Liu, Zhewei Yao, Olatunji Ruwase, Beidi Chen, Xiaoxia Wu, and Zhangyang Wang. Found in the middle: How language models use long contexts better via plug-and-play positional encoding, 2024.
- [37] Enyu Zhou, Guodong Zheng, Binghai Wang, Zhiheng Xi, Shihan Dou, Rong Bao, Wei Shen, Limao Xiong, Jessica Fan, Yurong Mou, Rui Zheng, Tao Gui, Qi Zhang, and Xuanjing Huang. Rmb: Comprehensively benchmarking reward models in llm alignment, 2025.
- [38] Banghua Zhu, Michael Jordan, and Jiantao Jiao. Principled reinforcement learning with human feedback from pairwise or k-wise comparisons. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 43037–43067. PMLR, 23–29 Jul 2023.

```

1 from collections import Counter
2 def calculate_ngram_repetition_penalty(text, n):
3     words = text.split()
4     ngrams = [tuple(words[i:i+n]) for i in range(len(words) - n + 1)]
5     ngram_counts = Counter(ngrams)
6     total_ngrams = len(ngrams)
7     repeated_ngrams = sum(1 for count in ngram_counts.values() if
8                             count > 1)
9     repetition_penalty = repeated_ngrams / total_ngrams if
10                             total_ngrams > 0 else 0
11     return repetition_penalty
12
13 def reasoning_pattern_reward(solution):
14     reason_pos = solution.find("Assistant:␣<think>")
15     solution_str = solution[think_pos + len("Assistant:␣<think>␣"):]
16     score = 0
17     solution_str = solution.lower()
18     score += float("i␣need␣to" in solution_str)
19     score += float("we␣need␣to" in solution_str)
20     score += float("wait" in solution_str)
21     score += float("alternatively" in solution_str)
22     score += float("let␣me␣check" in solution_str)
23     score += float("let␣me␣see" in solution_str)
24     score += float("let's␣focus␣on" in solution_str)
25     score += float("we␣know␣that" in solution_str)
26     score += float("we␣can␣observe␣" in solution_str)
27     score += float("we␣can␣see␣" in solution_str)
28     score += float("let␣me␣try" in solution_str)
29     score += float("let's␣try" in solution_str)
30     score += float("let␣us␣try" in solution_str)
31     score += float("first," in solution_str)
32     score += float("firstly," in solution_str)
33     score += float("next," in solution_str)
34     score += float("finally," in solution_str)
35     score += float("let␣us␣first" in solution_str)
36     score += float("let's␣first" in solution_str)
37     score += float("let␣me␣first" in solution_str)
38     score += float("try␣again" in solution_str)
39     score += float("still␣not" in solution_str)
40     score += float("not␣working" in solution_str)
41     score += float("not␣correct" in solution_str)
42     score += float("does␣not␣work" in solution_str)
43     score += float("doesn't␣work" in solution_str)
44     score += float("makes␣sence" in solution_str)
45     score += float("since␣we" in solution_str)
46     score += float("because␣we" in solution_str)
47     score += float("consequently" in solution_str)
48     score += float("as␣a␣result" in solution_str)
49     score += float("thus" in solution_str)
50     score += float("therefore" in solution_str)
51     score += float("hence" in solution_str)
52     score += float("so␣that" in solution_str)
53     score += float("thereby" in solution_str)
54     score += float("if␣we" in solution_str)
55     score += float("given␣there" in solution_str)
56     score += float("for␣instance" in solution_str)
57     score += float("for␣example" in solution_str)
58     score /= 40
59     score -= calculate_ngram_repetition_penalty(solution_str, 20)
60     score = max(0, score)
61     return score

```

Figure 9: An example code of reasoning pattern reward without checking answer correctness.

A conversation between User and Assistant. The user asks a question, and the Assistant solves it. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. The reasoning process and answer are enclosed within `<think>` and `<answer>` tags, respectively, i.e., `<think>` reasoning process here `</think>` `<answer>` answer here `</answer>`. User: You must put your answer choice (a single capital letter) inside `<answer>` and `</answer>` tags, i.e., `<answer>` answer here `</answer>`. And your final answer will be extracted automatically by the `\boxed{}` tag. Dienes are organic compounds with two adjacent double bonds in their structure, and they exhibit unique reactivity due to their conjugated pi-electron system. They play a significant role in organic chemistry and are involved in various chemical reactions and natural processes.

Among the given options which one is the possible reactant (A) for the given reaction also mention the correct sequence of the dienes according to their reactivity (most reactive to least reactive) B.

Cyclohexene + A \longrightarrow 8,8-diiodobicyclo[4.2.0]octan-7-one

(B) 1. 2,3-dimethylbuta-1,3-diene, 2. (2E,4E)-hexa-2,4-diene, 3. cyclopenta-1,3-diene, 4. (2Z,4Z)-hexa-2,4-diene

A. A = 2,2-diiodoethen-1-one, B = 3, 1, 2, 4
 B. A = 4,4-diiodocyclobut-2-en-1-one, B = 3, 1, 2, 4
 C. A = 2,2-diiodoethen-1-one, B = 4, 2, 1, 3
 D. A = 4,4-diiodocyclobut-2-en-1-one, B = 4, 2, 1, 3

Assistant: `<think>` First, I need to understand what we know about dienes and how they react with other compounds. We know that dienes have two adjacent double bonds, so let's see if we can figure out which of these options contains a diene. Additionally, we need to determine the correct sequence of dienes based on their reactivity, so let me try to recall some information about that.

Firstly, since we are looking at a reaction involving cyclohexene, it seems like we might be dealing with an addition reaction where one of the dienes would add across the double bond of cyclohexene. Next, I need to consider the structure of each option provided and identify any dienes present within them. Furthermore, since we want to find the most reactive diene first, I should look into the properties of different types of dienes and their reactivity patterns.

Next, let's try to analyze the structures of these compounds and see if we can spot any dienes among them. Also, we need to figure out how these dienes would react with cyclohexene and form the product mentioned in the question. Lastly, we need to determine the correct order of reactivity for these dienes.

Firstly, let us examine option A because it has a simple structure and contains a diene. Now, let us check if this diene could react with cyclohexene to produce the desired product. Secondly, let us compare the structures of the other options to see if they contain dienes as well. Finally, we need to establish the correct sequence of reactivity for these dienes.

Firstly, let us focus on option A, which includes 2,2-diiodoethen-1-one and 3,1,2,4 as its components. Since we know that dienes are characterized by having two adjacent double bonds, let us try to identify those in this option. Upon closer inspection, we notice that there is indeed a diene present in this option, specifically 2,2-diiodoethen-1-one. This compound contains two adjacent double bonds, making it a diene. Therefore, we can conclude that option A contains a diene. However, we still need to determine if this diene can react with cyclohexene to form the required product.

Next, let us examine option B, which consists of 4,4-diiodocyclobut-2-en-1-one and 3,1,2,4 as its components. Again, we need to check if this option contains a diene. After analyzing the structure, we find that there is no diene present in this option. Hence, we can eliminate option B as a potential reactant.

Now, let us move on to option C, which comprises 2,2-diiodoethen-1-one and 4,2,1,3 as its components. Similar to our previous analysis, we observe that there is a diene present in this option, namely 2,2-diiodoethen-1-one. Thus, we can confirm that option C also contains a diene. However, we must now determine if this diene can react with cyclohexene to yield the desired product.

Lastly, let us analyze option D, which includes 4,4-diiodocyclobut-2-en-1-one and 4,2,1,3 as its components. Once more, we need to verify if this option contains a diene. Upon examination, we realize that there is no diene present in this option. Consequently, we can rule out option D as a viable reactant.

Having analyzed all the options, we have identified that both options A and C contain a diene. Therefore, we need to decide which one is the correct reactant for the given reaction. To do this, we need to consider the reactivity of these dienes and determine which one is more likely to participate in the reaction.

Firstly, let us think about the general reactivity pattern of dienes. Generally speaking, dienes with more substituted double bonds tend to be more reactive than those with fewer substituents. Moreover, dienes with electron-donating groups attached to the double bond are typically more reactive than those without such groups. With this knowledge in mind, let us compare the structures of the dienes present in options A and C.

Upon comparing the structures, we notice that the diene in option A has two iodine atoms attached to the double bond, whereas the diene in option C does not have any substituents on the double bond. Consequently, we can infer that the diene in option A is likely to be more reactive than the one in option C due to the presence of the electron-withdrawing iodine atoms.

Therefore, we can conclude that option A is the most probable reactant for the given reaction. Now, we need to establish the correct sequence of reactivity for these dienes.

Secondly, let us think about the reactivity of dienes in general. Typically, dienes with more substituted double bonds tend to be more reactive than those with fewer substituents. Furthermore, dienes with electron-donating groups attached to the double bond are usually more reactive than those without such groups. Based on this information, we can deduce that the diene in option A, which has two iodine atoms attached to the double bond, is likely to be the most reactive diene among the ones we have considered. Next, we need to determine the reactivity of the diene in option C.

Finally, let us consider the reactivity of the diene in option C. As previously mentioned, the diene in option C lacks any substituents on the double bond. Consequently, we can infer that this diene is likely to be less reactive than the one in option A. Thus, we can conclude that the diene in option C is the least reactive diene among the ones we have analyzed.

Hence, we can establish the correct sequence of reactivity for these dienes as follows: the diene in option A is the most reactive, followed by the diene in option C, which is the least reactive.

Lastly, we need to determine the correct sequence of the dienes according to their reactivity. Since we have already established that the diene in option A is the most reactive and the diene in option C is the least reactive, we can conclude that the correct sequence of reactivity for these dienes is option A, then option C.

Thus, we need to select the correct option from the given choices. Given that we have determined that option A is the correct reactant and the correct sequence of reactivity is option A, we can now proceed to choose the appropriate answer.

Hence, we need to select the option that matches our findings. Since we have concluded that option A is the correct reactant and the correct sequence of reactivity is option A, we can deduce that the correct answer must be option A.

Therefore, we need to choose the option that corresponds to our findings. Hence, we need to select the option that matches our conclusion. Consequently, we can conclude that the correct answer is option A.

Finally, we need to express our final answer using the provided format. Thus, we need to enclose our answer within the specified tags.

Hence, I shall now provide my final answer in the required format.

Thus, we can conclude that the correct answer is `\boxed{\text{A}}`.

Therefore, we can write our final answer as follows: `\$ \therefore \text{The correct answer is } \boxed{\text{A}} \$`. Hence, we can express our final answer as `\$ \boxed{\text{A}} \$`.

Figure 10: An example of output in the later stage of RL math training, where only the reasoning pattern reward is used without correctness verification. The model has arrived at the correct answer “A,” but due to the ongoing reasoning process, the `<think>` tag remains open, causing the output length to reach the limit and preventing the correct answer from being generated in answer tags.

A Experiments on Qwen-2.5-3B and Llama-3.1-8B

In Figure 11, we demonstrate that Qwen-2.5-3B exhibits strong robustness to significant reward noise (*Experiment 1*). Specifically, it can tolerate up to 40% of rewards being flipped while still achieving final performance comparable to the noiseless setup. However, its convergence under noisy rewards is noticeably slower than that of the 7B models. Additionally, performance on the AIME tasks continues to fluctuate, consistent with the behavior observed in the 7B model.

For *Experiment 2*, we find that using RPR as the sole reward signal effectively enables the model to reach peak performance comparable to the noiseless baseline. Notably, on the most challenging AIME tasks, RPR yields the highest peak performance across all setups.

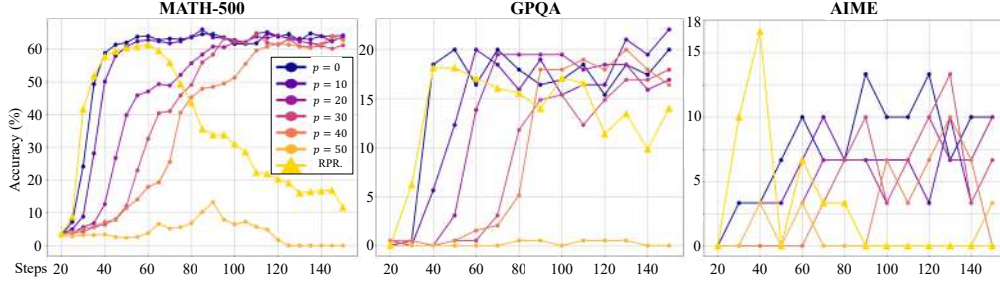


Figure 11: Qwen-2.5-3B results for Experiments 1 and 2: Accuracy on three test sets during training.

In the HelpSteer3 task, vanilla RL fails to enable Qwen-2.5-3B to perform effective reasoning. We observe that response lengths initially increase but then rapidly collapse to just a few tokens. This pattern of “first-reason-then-collapse” has also been observed in [23], where an LLM is trained to reason on tasks beyond its initial capabilities. However, the underlying mechanisms driving this length dynamics remain understudied. In contrast, when trained with RPR-calibrated RMs (accuracy $> 75\%$), Qwen-2.5-3B exhibits clear reasoning behaviors. As shown in Figure 12(a), the response lengths differ significantly between models trained with original versus calibrated RMs. These results echo the core insight of *Experiment 4*: calibrated RMs more effectively evoke reasoning abilities in large language models. Subfigure (b) shows that using an 85%-accurate RM yields only a 5-point improvement in net win rate over the 75%-accurate RM—mirroring observations in 7B models in *Experiment 3*.

Figure 13 and Figure 14 present two sample outputs from Qwen-2.5-3B trained with the 85%-accurate RM, demonstrating that we have successfully elicited the basic reasoning capabilities of this small-scale model, despite some imperfections in these outputs. In Figure 13, the Chinese query asks for the creation of a PowerPoint file to teach primary school students about statistical charts. The 3B model engages in step-by-step reasoning to generate a coherent PowerPoint structure and follows through on its plan. In Figure 14, the model processes a complex chat history and solves a physics problem, despite not being explicitly trained for mathematics or physics.

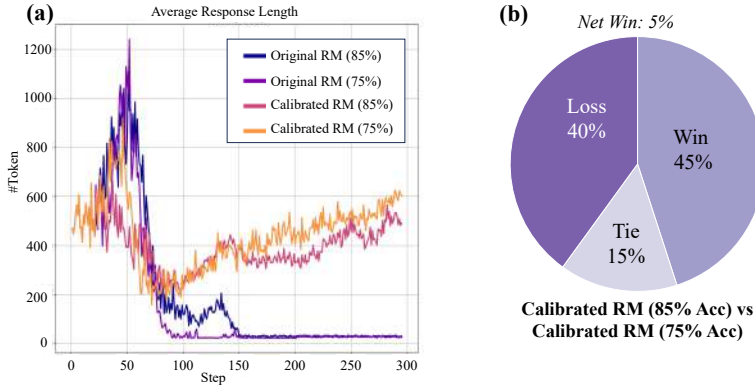


Figure 12: (a) Average response length of Qwen-2.5-3B during training with original vs calibrated RMs. The calibrated RMs successfully enable this small-scale model to perform reasoning, whereas the original RMs fail. (b) Experiment 3 using Qwen-2.5-3B models trained with calibrated RMs.

Question:

幫我設計教學用ppt，要帶小學生學習[統計圖表]，整體PPT結構要有導入活動、開展活動、總結活動，這三個活動中每個活動要有引發學生探究的問題並附上一張圖片來說明問題情境、概念講解(可以詳細點)，以及數學多重表徵實體操作活動(詳列步驟)，最後面要有形成性評量檢核點，每個檢核點，前面置放“□”符號置左，檢核點要有知識、技能、態度面向。整個ppt請用markdown標記語言格式，三級結構，分數部分請以a/b形式我比較看得懂。\\n 圖片部分，請先將 QUERY，翻譯成英文，再由這個網站全部英文輸入取得 (https://source.unsplash.com/960x640/?QUERY)。使用Markdown 語法來顯示圖片

Answer:

First, I need to plan the structure of the PPT for teaching statistics to primary school students. This includes creating a clear introduction, engaging activities, and a conclusion. Each section should include a problem to stimulate inquiry, concept explanation, hands-on activities, and assessment points. For images, I will use a combination of relevant images and custom ones generated using the provided website. Finally, I will format the PPT using Markdown with appropriate headers and lists.

Reasoning process:

1. Begin by drafting an introduction that captures the students' attention and explains the importance of learning about statistical graphs.
2. Design the "engage" section with a problem that encourages students to explore different types of graphs, such as bar charts or pie charts, and how they represent data.
3. Develop the "explain" section by providing clear definitions and examples of each type of graph, using visual aids to enhance understanding.
4. Create the "do" section with hands-on activities where students can create their own graphs using real-life data sets, reinforcing the concepts learned.
5. Prepare the "summarize" section with a review of the key points and encourage students to reflect on what they've learned.
6. Incorporate assessment points at the end of each activity, detailing what students should be able to demonstrate in terms of knowledge, skills, and attitudes.
7. Generate images for the PPT by translating search queries into English and using the website to fetch appropriate images.

Final Answer:

```markdown

# Teaching Statistics to Primary School Students

## Introduction

Welcome to our lesson on statistical graphs! Understanding how to interpret and create these visual representations is crucial for analyzing data in everyday life.

## Engage

\*\*Question:\*\* How can we visually compare the favorite fruits of our classmates?

\*\*Image:\*\* [Image of students discussing their favorite fruits]

## Explain

\*\*Concept:\*\* What are bar charts and pie charts? How do they differ?

\*\*Image:\*\* [Bar Chart Image]

[Pie Chart Image]

\*\*Definition:\*\* A bar chart uses bars to show comparisons between different categories of data. A pie chart shows the proportion of each category in relation to the whole.

## Do

\*\*Activity:\*\* Create a bar chart and a pie chart based on the following data:

- 15 students like apples
- 10 students like bananas
- 5 students like oranges

\*\*Image:\*\* [Example Bar Chart]

[Example Pie Chart]

## Summarize

\*\*Review:\*\* What did you learn about bar charts and pie charts? Can you describe their differences?

## Assessment Points

- \*\*Knowledge:\*\* Can you identify which type of graph is best suited for comparing quantities?
- \*\*Skills:\*\* Can you create a bar chart or pie chart from given data?
- \*\*Attitudes:\*\* Are you curious about how statistics can help us understand the world better?

☐ Knowledge ☐ Skills ☐ Attitudes

...

This PPT structure ensures a comprehensive learning experience for the students, combining theoretical knowledge with practical application, all while keeping the content engaging and interactive.

**Figure 13:** Our calibrated RMs successfully elicit Qwen-2.5-3B’s reasoning ability, whereas the original RM fails to do so. This figure presents 1 of 2 output cases. The Chinese question translates to: “Please design a teaching PowerPoint for teaching elementary school students about [statistical charts]. The overall structure of the PPT should include three parts: introductory activity, developmental activity, and summary activity. Each of these activities must include: A question that triggers student inquiry, along with an image illustrating the context of the problem. A concept explanation (detailed). A math hands-on activity using multiple representations, with a clearly listed step-by-step process. At the end of the PPT, include a formative assessment checklist. Each checklist item should start with the symbol "□" aligned to the left and should address three aspects: knowledge, skills, and attitudes. Please present the entire PPT in Markdown format with a three-level heading structure. For fractions, use the a/b format as it is easier for me to understand. As for the images, first translate the QUERY into English, and then use the following website with an English query to retrieve the images:https://source.unsplash.com/960x640/?QUERY Use Markdown syntax to display the images.”

Llama models are widely recognized to exhibit inherent weaknesses in reasoning capabilities and limited potential for improvement through reinforcement learning [23, 33, 10]. As shown in Figure 15, LLaMA-3.1-8B performs significantly worse than the Qwen models under noiseless conditions and suffers a marked degradation in performance as noise increases. The accuracy on MATH-500 drops to 0 at  $p = 30\%$ . Due to its limited foundational capabilities, LLaMA-3.1-8B is not suitable for effective training on the HelpSteer3 task, and thus we omit its results.

### Chat History:

User: What was the total length of the Circus Maximus? How many laps were in a traditional chariot race? How long would a chariot race take on average?

Assistant: The Circus Maximus in ancient Rome was approximately 621 meters (2,037 feet) in length. It was a massive stadium used for chariot racing and other public events. In a traditional chariot race, charioteers would have to complete 7 laps around the Circus Maximus. The starting point was at the closed end of the track, and the chariots would race towards the open end, then turn around a metae (turning post) and head back in the other direction. The average time for a chariot race in the Circus Maximus would depend on various factors such as the number of competitors, the condition of the track, and the skill of the charioteers. However, a typical race could last anywhere from 8 to 15 minutes.

### Question:

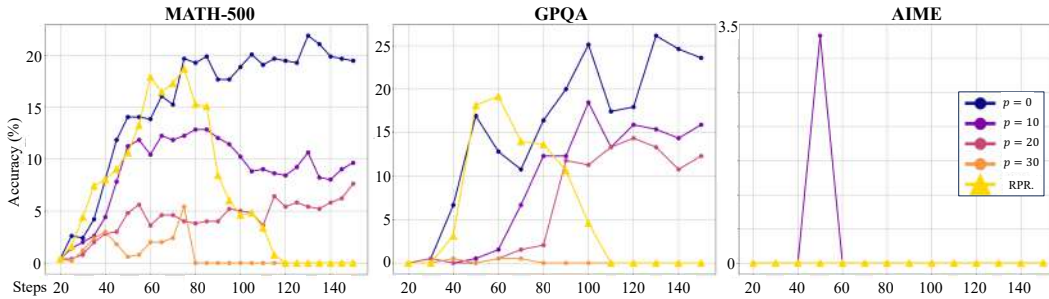
If a car is driving on a 45° bank 1 mile oval track, how fast does the car have to be to avoid falling off the track?

### Answer:

To ensure the car stays on the banked track, we need to balance the forces acting on it. The forces include gravity, the normal force, and friction. On a banked curve, the normal force acts at an angle, and we can resolve the gravitational force into components parallel and perpendicular to the surface. The component parallel to the surface provides the necessary centripetal force for circular motion. The formula for centripetal force is  $F_c = \frac{mv^2}{r}$ , where  $m$  is the mass of the car,  $v$  is the velocity, and  $r$  is the radius of the curve. The radius  $r$  can be calculated from the length of the track and the angle of the bank using  $r = \frac{L}{2\sin(\theta)}$ , where  $L$  is the length of the track and  $\theta$  is the angle of the bank. Substituting  $r$  into the centripetal force formula and equating it to the component of the gravitational force parallel to the surface ( $mg\sin(\theta)$ ), we can solve for the minimum velocity  $v$ . Given that the length of the track  $L = 1$  mile and the angle of the bank  $\theta = 45^\circ$ , the radius  $r = \frac{1}{2\sin(45^\circ)} = \frac{1}{2} \cdot \frac{1}{\sqrt{2}/2} = \frac{1}{\sqrt{2}}$ . Therefore, the minimum velocity  $v$  is given by  $v = \sqrt{\frac{2mg\sin(\theta)}{m}} = \sqrt{2g\sin(\theta)}$ . Since  $\sin(45^\circ) = \frac{\sqrt{2}}{2}$ , we have  $v = \sqrt{2g \cdot \frac{\sqrt{2}}{2}} = \sqrt{g\sqrt{2}}$ . Using  $g \approx 32 \text{ ft/s}^2$ , we find  $v \approx \sqrt{32 \cdot \sqrt{2}} \approx \sqrt{60.8} \approx 7.8 \text{ ft/s}$ . Converting this to miles per hour (since 1 mile = 5280 feet), we get  $v \approx 7.8 \cdot \frac{3600}{5280} \approx 51.4 \text{ mph}$ . Therefore, the car must be traveling at least 51.4 mph to avoid falling off the track.

Final Answer: The car must be traveling at least 51.4 mph to avoid falling off the track.

**Figure 14:** This figure presents the second of two output cases from Qwen-2.5-3B, trained with the calibrated 85%-accurate RM. The model demonstrates reasoning to solve a physics problem.



**Figure 15:** Llama-3.1-8B results for Experiments 1 and 2: Accuracy on three test sets during training (Llama-3.1-8B).

## B Human evaluation

### B.1 Guidelines

We recruited three graduate students with expertise in model evaluation. Each evaluator spent approximately 8 hours completing all tasks and earned \$70 USD. The human evaluation was granted by our institute, with the payment slightly above the standard wage for graduate students working in AI companies in our country. Below is the guideline for human evaluators:

#### Guideline for Evaluating Responses:

Your task is to help me determine which of two responses is more helpful in addressing the user's latest request. First, I will provide you with the conversation history between the user and the AI assistant, where the last statement is the user's most recent request. Then, I will show you the two assistant replies: response #1 and response #2. Please evaluate which response is better based on factors such as helpfulness, the amount of information provided, the thoroughness of the reasoning, and overall coverage of the user's needs. Please check two responses carefully, do not casually answer that #1 is better. **\*\*Make sure to clearly state whether '#1 is better' or '#2 is better' or 'tie' AT THE END OF YOUR ANSWER\*\*.**

Here is the conversation history:

`{chat}`

Response #1:

`{res1}`

Response #2:

`{res2}`

**Figure 16:** The evaluation prompt for GPT, designed according to the core guidelines for human annotators. The placeholders will be replaced with user-assistant chat history and two models' responses.

Your task is to determine which of the two responses better addresses the user's latest request.

#### Steps to Follow:

- **Review the Conversation History:** Carefully read the conversation history provided. The user's most recent question will be the last message, and that is the request you need to evaluate the responses against.
- **Examine the Two Responses:** You will be presented with two possible replies from two AI assistants (Response #1 and Response #2).
- **Criteria for Evaluation:** Evaluate each response based on the following factors:
  - **Helpfulness:** Does the response directly answer the user's request? Is it practical and useful?
  - **Amount of Information:** Does the response provide sufficient details to address the request thoroughly?
  - **Clarity and Coherence:** Is the response easy to understand, and does it present information logically?
  - **Thoroughness:** Does the response cover all aspects of the user's request? Is anything missing or incomplete?
- **Avoid Quick Judgment:** We will randomize the response order from two models. You cannot infer which one is always better based on the order. Also, don't assume one response is better simply because it's shorter or longer.

After evaluating both responses, decide which one is more helpful overall. You can choose #1 is better, #2 is better, or they tie with each other. Write the evaluation, as well as reasons.

## B.2 Results and inter-annotator agreement

In Figure 17, we present the averaged human evaluation results for Experiments 3 and 4 in Section 3. Each figure also reports inter-evaluator agreement  $\kappa$ , with all experiments demonstrating moderate ( $0.4 < \kappa \leq 0.6$ ) to substantial ( $0.6 < \kappa \leq 0.8$ ) consistency among evaluators. A key distinction between human evaluations and those from GPT-4o is that human judges exhibit stronger discriminatory ability, resulting in fewer comparisons marked as "ties." Nonetheless, the overall conclusions—such as the net win ratios and the impact of calibration—align closely with the GPT-based evaluations. Therefore, we do not repeat Takeaways and conclusions here.



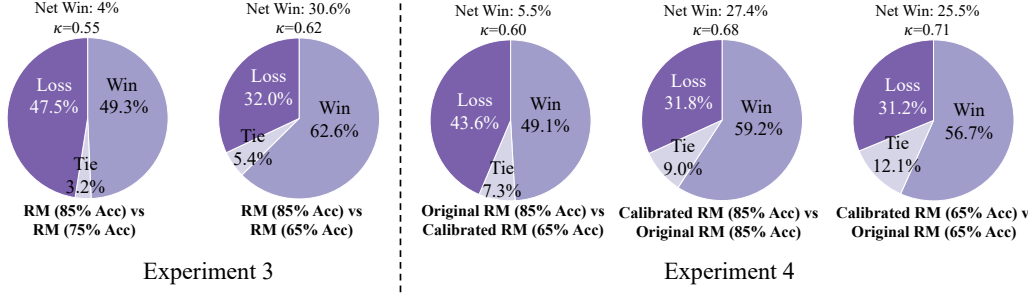


Figure 17: Human evaluation results and agreements.

## C Case studies

Figures 19 and 20 show outputs from Qwen-2.5-7B trained with the calibrated and original 85%-accurate RMs, respectively. With RPR, the generated code and comments are more detailed, and both the main function and interaction loop are more comprehensive compared to the single test case produced by models trained without RPR. The reasoning process is also more thoroughly articulated.

Figures 21 and 22 illustrate Qwen-2.5-7B trained with calibrated and original 65%-accurate RMs, respectively. Compared to models trained with 85%-accurate RMs, both outputs here fail to explicitly move the model to the GPU. However, the model trained with the calibrated 65%-accurate RM correctly implements a chatbot using the transformers pipeline API, which implicitly moves the model to the GPU. As a result, the model trained with the original 65%-accurate RM performs slightly worse in terms of helpfulness. It is uncommon for an assistant to build a chatbot using the transformers pipeline—an approach that is both concise and effective—suggesting that Qwen models have acquired substantial knowledge during pretraining.

## D Limitations, broader impacts and safety issues

Due to resource limitations, we limit our experiments to models with up to 7–8 billion parameters. However, because our results are consistent even for smaller Qwen-2.5-3B models, and given that the 14B and 70B Qwen models exhibit stronger inherent reasoning abilities, the scalability of our takeaways might not be a concern, as the core premise of these findings relies on strong reasoning capabilities in pre-trained models.

Regarding broader impacts, we hope this paper will inspire future advancements in post-training techniques. It is possible that efficient tuning methods could yield similar outcomes with RL. Additionally, we highlight the importance of continuing efforts to enhance fundamental reasoning abilities during the pre-training stage. This paper presents findings and insights into post-training LLMs using RL with noisy rewards, which do not raise safety concerns.

## E RPR weight in calibration

In Section 3.3, we set  $\tau = 0.5$  since the RMs are binary classifiers. For the scaling factor  $\alpha$ , we initially used a value of 1, but observed little difference in training dynamics—such as average response length—compared to using the original RMs. We then experimented with values of  $\alpha$  ranging from 1 to 0.1. Notably, reducing  $\alpha$  to 0.1 led to a significant increase in average response length (see Figure 18), suggesting a shift in the model’s reasoning behavior. Based on these findings, we set  $\alpha = 0.1$ .

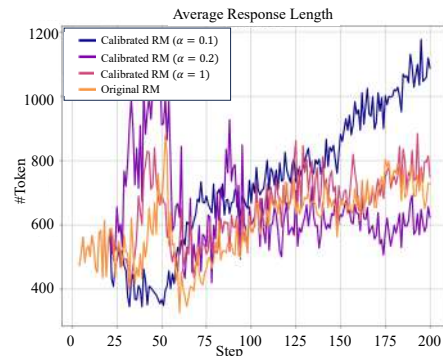


Figure 18:  $\alpha = 0.1$  enables Qwen’s effective reasoning in HelpSteer3 task. These experiments use the 85%-accurate RM.

**Question:** Generate python code for creating a chatbot using a large language model

**Answer:** To create a chatbot using a large language model, we need to follow a structured approach that involves several steps, including choosing the right model, preparing the data, setting up the chatbot framework, and testing the chatbot. Below, I will outline the reasoning process and provide a Python code example that demonstrates how to create a simple chatbot using the Hugging Face Transformers library, which provides access to various large language models. This example will use the GPT-2 model as an example, but you can replace it with other models as needed.

#### ### Reasoning Process

1. **Choose the Right Large Language Model**: Select a suitable pre-trained large language model. For this example, we'll use the GPT-2 model, which is known for its ability to generate human-like text. Other options include GPT-3, BERT, or other models depending on the specific requirements and the tasks the chatbot will perform.
2. **Set Up the Environment**: Ensure you have the necessary libraries installed. This includes the Hugging Face Transformers library for accessing the pre-trained models and tokenizers, and possibly other libraries for additional functionalities like handling user input and output.
3. **Load the Pre-Trained Model and Tokenizer**: Use the Transformers library to load the pre-trained model and its corresponding tokenizer. The tokenizer is crucial for converting text into tokens that the model can understand and for converting the model's outputs back into human-readable text.
4. **Define the Chatbot Logic**: Implement the logic for handling user input, generating responses using the model, and managing the conversation flow. This might involve simple rules-based systems or more complex natural language understanding (NLU) techniques.
5. **Test and Iterate**: Test the chatbot with various inputs to ensure it behaves as expected. Iterate on the model selection, data preparation, and chatbot logic based on the test results and any feedback.

#### ### Summary of Reasoning

The process of creating a chatbot using a large language model involves selecting an appropriate model, setting up the necessary environment, loading the model and tokenizer, defining the chatbot's interaction logic, and thoroughly testing the chatbot to refine its performance. The key is to leverage the capabilities of the large language model for generating coherent and contextually relevant responses while ensuring the chatbot is user-friendly and effective in its intended use case.

#### ### Python Code Example

Here is a Python code example that demonstrates these steps using the Hugging Face Transformers library to create a simple chatbot with the GPT-2 model:

```
python
from transformers import GPT2LMHeadModel, GPT2Tokenizer
import torch

Step 1: Choose the right large language model
model_name = 'gpt2' # You can change this to other models like 'gpt2-medium', 'gpt2-large', etc.

Step 2: Set up the environment
Ensure you have the necessary libraries installed: transformers, torch

Step 3: Load the pre-trained model and tokenizer
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)

Move model to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

Step 4: Define the chatbot logic
def generate_response(prompt):
 # Encode the user's input
 input_ids = tokenizer.encode(prompt, return_tensors='pt').to(device)

 # Generate text until the output length (which includes the input length) reaches 50
 output = model.generate(input_ids, max_length=50, num_return_sequences=1, no_repeat_ngram_size=2)

 # Decode the generated tokens and return the response
 response = tokenizer.decode(output[0], skip_special_tokens=True)
 return response

Step 5: Test and interact with the chatbot
if __name__ == "__main__":
 print("Chatbot: Hello! How can I assist you today?")
 while True:
 user_input = input("You: ")
 if user_input.lower() in ['exit', 'quit']:
 print("Chatbot: Goodbye!")
 break
 response = generate_response(user_input)
 print(f"Chatbot: {response}")
 ...
```

#### ### Final Answer

To create a chatbot using a large language model like GPT-2, you first need to choose the appropriate model based on your requirements. Then, set up your Python environment with the necessary libraries, particularly the Hugging Face Transformers library. Load the pre-trained model and tokenizer using the 'GPT2LMHeadModel' and 'GPT2Tokenizer' classes, respectively. Define the chatbot's interaction logic in a function that takes user input, encodes it, generates a response using the model, and decodes the model's output back into text. Implement a simple loop to continuously interact with the chatbot, allowing users to input messages and receive responses from the chatbot. Test the chatbot thoroughly to ensure it meets your expectations and refine it based on feedback and performance. The provided Python code sets up a basic framework for achieving this, using GPT-2 as the example model, and can be extended or modified according to specific needs and the capabilities of other large language models.

**Figure 19:** How to create a chatbot using an LLM: the answer from Qwen-2.5-7B trained with the calibrated 85%-accurate RM.

**Question:** Generate python code for creating a chatbot using a large language model

**Answer:** To create a chatbot using a large language model in Python, we can leverage libraries such as `transformers` by Hugging Face, which provides easy access to various pre-trained language models including GPT-3 compatible models (like GPT-2 or larger models in the GPT series, BERT, etc.). Below is a step-by-step reasoning process followed by the Python code to create a simple chatbot:

### Reasoning Process:

1. **Choose a Language Model**: Select a pre-trained language model from the `transformers` library. For this example, we'll use `GPT-2` due to its balance between capability and ease of use.
2. **Import Necessary Libraries**: Import `transformers` and `torch` (for handling tensors and computations).
3. **Load the Model and Tokenizer**: Load the GPT-2 model and its associated tokenizer. The tokenizer is crucial for converting text inputs into tokens that the model can understand.
4. **Define the Chatbot Functionality**: Create a function that takes user input, tokenizes it, feeds it through the model to get a response, and then tokenizes and decodes the model's output back into text.
5. **Handle Conversation Context**: While simple, for this example, we won't maintain conversation context but in a production setting, you might want to feed the previous interaction history to the model to make the chat more coherent.

### Python Code:

```
```python
import torch
from transformers import GPT2Tokenizer, GPT2LMHeadModel

# Step 1: Load the GPT-2 model and tokenizer
model_name = 'gpt2' # Using gpt2, can also use 'gpt2-medium', 'gpt2-large', 'gpt2-xl' for more capacity
model = GPT2LMHeadModel.from_pretrained(model_name)
tokenizer = GPT2Tokenizer.from_pretrained(model_name)

# Ensure the model is set to evaluation mode and moved to the appropriate device (CPU or GPU)
model.eval()
if torch.cuda.is_available():
    model.to(torch.device("cuda"))

# Step 2: Define the chat function
def chat_with_bot(user_input):
    # Tokenize the user input
    input_ids = tokenizer.encode(user_input, return_tensors='pt')

    # Generate a response
    # Here we generate 100 tokens (you can adjust this based on expected response length)
    output = model.generate(input_ids, max_length=input_ids.shape[1] + 100, num_return_sequences=1)

    # Decode the response back to text
    bot_response = tokenizer.decode(output[0], skip_special_tokens=True)

    return bot_response

# Example usage
user_message = "Hello, how are you?"
response = chat_with_bot(user_message)
print("Bot says:", response)
```
```

### Summary of Reasoning and Answer:

In this solution, we first imported the necessary components from the `transformers` library, specifically the GPT-2 model and its tokenizer. We loaded these into our script ensuring they are ready for inference, with an option to use a GPU if available for speedup. The `chat\_with\_bot` function encapsulates the interaction: it tokenizes the user's input, uses the model to generate a response by extending the input sequence, and finally decodes the model's output back into human-readable text. This approach leverages the model's ability to generate text that continues from a given prompt, effectively simulating a chatbot response. Note that the quality and relevance of the responses depend heavily on the pre-training of the GPT-2 model and the specific tuning or fine-tuning it might have undergone for chatbot applications.

**Figure 20:** How to create a chatbot using an LLM: the answer from Qwen-2.5-7B trained with the original 85%-accurate RM.

**Question:** Generate python code for creating a chatbot using a large language model

**Answer:** To create a chatbot using a large language model in Python, we can follow these steps:

1. **Select a Language Model:** We'll use a pre-trained language model from Hugging Face's `transformers` library, such as `gpt-2` or `gpt-j`. For this example, we'll use `gpt-2`.

2. **Set Up the Environment:** Ensure you have Python installed along with the necessary libraries. You can install them using pip if they are not already installed:

```
```bash
pip install transformers gradio
```
```

3. **Create the Chatbot:** We'll write a Python script that sets up the chatbot using the selected language model, handles user input, generates responses, and displays them in a user-friendly way using `gradio`.

Here's a complete Python script that accomplishes this:

```
```python
import gradio as gr
from transformers import pipeline

def generate_response(user_input):
    # Load the pre-trained model for text generation
    text_generator = pipeline('text-generation', model='gpt-2')

    # Generate a response using the model
    response = text_generator(user_input, max_length=150, num_return_sequences=1)

    # Extract and return the generated text
    return response[0]['generated_text']

# Set up the Gradio interface
iface = gr.Interface(
    fn=generate_response, # The function to call when input changes
    inputs="text",        # Type of input (in this case, text)
    outputs="text",        # Type of output (text)
    title="GPT-2 Chatbot", # Title of the interface
    description="Talk to a chatbot powered by the GPT-2 model." # Description of the interface
)

# Launch the interface
iface.launch()
```
```

### Explanation of the Code:

- **Import Libraries:** We import `gradio` for creating the web interface and `pipeline` from `transformers` for easily accessing the language model.
- **Define Response Generation Function:** The `generate\_response` function takes user input, uses the `gpt-2` model to generate a response, and returns the generated text.
- **Set Up Gradio Interface:** We define the Gradio interface using `gr.Interface`. We specify the function to be called (`generate\_response`), the type of input (`text`), and the type of output (`text`). We also add a title and description to the interface.
- **Launch the Interface:** Finally, we call `iface.launch()` to start the web interface where users can interact with the chatbot.

This script sets up a basic chatbot using the GPT-2 model through a simple web interface. Users can type in questions or statements, and the chatbot will generate responses based on the model's predictions. This example can be extended and customized further based on specific requirements, such as integrating more sophisticated input handling, improving the model's parameters, or adding more advanced features to the chatbot.

**Figure 21:** How to create a chatbot using an LLM: the answer from Qwen-2.5-7B trained with the calibrated 65%-accurate RM.

**Question:** Generate python code for creating a chatbot using a large language model

**Answer:** To create a chatbot using a large language model in Python, you can follow these steps. This example will use the Hugging Face Transformers library, which provides easy access to a wide range of pre-trained models. We'll use the `gpt-2` model for this demonstration, but you can replace it with any other model from the Hugging Face Model Hub that suits your needs.

#### ### Step 1: Setting up the Environment

First, ensure you have Python installed along with the necessary libraries. You'll need `transformers` by Hugging Face and `torch` for handling the model computations. You can install them using pip if they are not already installed:

```
```bash
pip install transformers torch
```
```

#### ### Step 2: Importing Libraries and Loading the Model

Import the required libraries and load the pre-trained model. Here's how you can do it:

```
```python
import torch
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Load pre-trained model and tokenizer
model_name = 'gpt2' # You can change this to any other model name from the Hugging Face Model Hub
model = GPT2LMHeadModel.from_pretrained(model_name)
tokenizer = GPT2Tokenizer.from_pretrained(model_name)

# Set the model to evaluation mode
model.eval()
```
```

#### ### Step 3: Creating a Basic Chat Function

Now, let's create a simple chat function. This function will take user input, tokenize it, generate a response using the model, de-tokenize the response, and print it out. Note that the quality of responses will depend heavily on the model used and might require fine-tuning for better performance on specific tasks or domains.

```
```python
def chat_with_model(user_input):
    # Tokenize the user input
    input_ids = tokenizer.encode(user_input + tokenizer.eos_token, return_tensors='pt')

    # Generate a response
    output = model.generate(input_ids, max_length=150, num_return_sequences=1, no_repeat_ngram_size=2)

    # Decode and print the response
    response = tokenizer.decode(output[0], skip_special_tokens=True)
    print("Chatbot:", response)

# Example usage
user_input = "Hello, how are you?"
chat_with_model(user_input)
```
```

#### ### Summary of Reasoning:

1. **Environment Setup**: Ensure all required libraries are installed and accessible.
2. **Model Loading**: Use the Hugging Face Transformers library to load a pre-trained large language model (e.g., `gpt-2`). This step includes both the model weights and the tokenizer needed to convert text into a format the model can understand and vice versa.
3. **Model Configuration**: Set the model to evaluation mode to ensure it operates in a way that is suitable for generating responses rather than training.
4. **Chat Function**: Create a function that takes user input, processes it through the model to generate a response, and then presents that response. This involves tokenizing the input, generating text based on the model, and de-tokenizing the output to human-readable form.

This basic chatbot can be expanded by adding features such as context handling, more sophisticated response generation, or integration with other services for a more interactive experience.</answer>

**Figure 22:** How to create a chatbot using an LLM: the answer from Qwen-2.5-7B trained with the original 65%-accurate RM.