Error Detection and Correction for Interpretable Mathematics in Large Language Models

Yijin Yang[1], Cristina Cornelio[2], Mario Leiva[3], Paulo Shakarian[4]
[1]Arizona State University, Tempe, AZ
[2]Samsung AI, Cambridge, UK
[3]Universidad Nacional del Sur, Buenos Aires, Argentina
[4]Syracuse University, Syracuse, NY
yyang491@asu.edu, c.cornelio@samsung.com, mario.leiva@cs.uns.edu.ar, pashakar@syr.edu

Abstract

Recent large language models (LLMs) have demonstrated the ability to perform explicit multi-step reasoning such as chain-of-thought prompting. However, their intermediate steps often contain errors that can propagate leading to inaccurate final predictions. Additionally, LLMs still struggle with hallucinations and often fail to adhere to prescribed output formats, which is particularly problematic for tasks like generating mathematical expressions or source code. This work introduces EDCIM (**E**rror **D**etection and **C**orrection for **I**nterpretable **M**athematics), a method for detecting and correcting these errors in interpretable mathematics tasks, where the model must generate the exact functional form that explicitly solve the problem (expressed in natural language) rather than a black-box solution. EDCIM uses LLMs to generate a system of equations for a given problem, followed by a symbolic error-detection framework that identifies errors and provides targeted feedback for LLM-based correction. To optimize efficiency, EDCIM integrates lightweight, open-source LLMs with more powerful proprietary models, balancing cost and accuracy. This balance is controlled by a single hyperparameter, allowing users to control the trade-off based on their cost and accuracy requirements. Experimental results across different datasets show that EDCIM significantly reduces both computational and financial costs, while maintaining, and even improving, prediction accuracy when the balance is properly configured.

Code — https://github.com/lab-v2/EDCIM
Extended version — https://arxiv.org/abs/2508.03500

Introduction

Large language models (LLMs) have shown significant potential for converting unstructured natural language (NL) text into structured formats suitable for downstream tasks, such as equation solvers or code compilers. This capability has been applied to a wide range of tasks, including solving math problems (Lewkowycz et al. 2022), combinatorial optimization problems (Ye et al. 2023; Zhao et al. 2023), and declarative logic (He-Yueya et al. 2023). However, despite these advances, LLMs often produce intermediate steps with errors, which can propagate to the final output and lead to inaccurate predictions. This is particularly problematic for structured outputs like mathematical equations, where even minor errors can result in completely incorrect solutions. Moreover, LLMs are prone to hallucinations and frequently struggle to adhere to strict output formats. To address these challenges, recent work has focused on re-prompting methods to reduce errors and improve the reliability of LLM outputs (Gou et al. 2023; An et al. 2023). However, these methods often require multiple re-prompting steps, as they do not have explicit error detection, resulting in significant resource consumption and high costs, especially when using powerful state-of-the-art LLMs.

To address these challenges, we introduce EDCIM (**E**rror **D**etection and **C**orrection for **I**nterpretable **M**athematics), a novel error detection and correction framework specifically designed for interpretable mathematical reasoning and inspired by concepts from meta-cognition ("thinking about thinking") (Flavell 1976; Didolkar et al. 2024). Unlike prior methods (Gou et al. 2023), which assume every response is potentially incorrect and thus re-query every sample, EDCIM uses a more selective approach. It combines lightweight, open-source LLMs for initial response generation with more powerful, cloud-based models for targeted error correction, reducing overall costs while maintaining high accuracy. Alternatively, it can also be used to enhance the performance of an LLM, whether local or cloud-based, by enabling it to iteratively correct and refine its own outputs. This approach is made possible via the use of symbolic Error Detection Rules, adapted from Kricheli et al. (2024), which identify specific error patterns in the generated equations before deciding whether to trigger a correction via re-prompting. This approach also brings the benefit of explainability, as it provides explicit feedback on the potential mistakes made by the LLM.

A key feature of EDCIM is its ability to balance the cost and accuracy via a single hyperparameter, which directly controls the re-prompt rate. This flexibility is fundamental for practical applications, allowing users to adapt to different computational budgets. For example, in large-scale experiments or industrial systems that involve millions of LLM queries, controlling cost and latency becomes as important as accuracy. Unlike prior methods with limited re-prompting control, EDCIM offers a simple and effective way to manage this trade-off. Our experiments show that using local models (Phi3) for initial solutions, combined with selective cloud-based (DeepSeek or GPT4o) queries for error correction, significantly reduces costs with minimal accuracy loss. Specifically, EDCIM re-prompts in only about one-third of cases while maintaining over 90% of the full re-prompt accuracy. Another major contribution of our work is EDCIM's improved correction quality: even when full correction is not possible, it consistently brings generated equations closer to the ground truth. Finally, although EDCIM was designed for mathematical reasoning, our approach can be extended to other structured tasks, such as declarative logic, program synthesis, and combinatorial optimization.

LLMs are computationally intensive, consuming significant energy and contributing to the carbon footprint of AI systems. By providing a flexible framework that allows users to control the balance between computational cost and accuracy, EDCIM enables more efficient use of these models, reducing their environmental impact. Additionally, by improving the reliability of generated outputs through targeted error detection and correction, our approach enhances the safety of AI systems in applications where even small errors can lead to significant consequences.

Limitations. EDCIM performance depends on the selection of the error detection rules, which currently is done manually by a domain expert. This can be particularly difficult when the task under consideration is very difficult or under-defined, where explicit error patterns are harder to define. Moreover, the error detector requires supervised data for learning which are the relevant rules. However, the number of samples required is minimal ($\sim$100). Additionally, the final solution relies on symbolic solvers, which are known to have scalability limitations.

Related work. Imposing constraints on neural networks and correcting their outputs is a challenging and widely studied topic in AI (Cornelio et al. 2023). This is especially true for LLMs, where recent research has focused on correcting their outputs across many use cases (Kamoi et al. 2024; Mishra et al. 2024; Pan et al. 2023b; Upadhyaya and Sridharamurthy 2024). Most relevant to our work is CRITIC (Gou et al. 2023), a framework that, inspired by human reasoning, enables LLMs to self-correct by simulating a critique-and-revise loop with structured reasoning and feedback from external tools, such as calculators or code interpreters. Importantly, CRITIC assumes that all the responses are incorrect and asks an LLM to revise its answers iteratively multiple times without distinguishing whether the answers are truly incorrect or not. Several recent methods have expanded upon or complemented the principles behind CRITIC. For instance, REFINER (Paul et al. 2023) introduced a similar self-refinement mechanism but focused on open-ended questions and factual generation, using internal consistency checks and a feedback model. Self-Refine (Madaan et al. 2023) adopts an edit-based feedback loop, where the model iteratively improves its answers based on automatically generated feedback.

Program-aided reasoning is another relevant line of work. Unlike PAL and MathCoder, which always apply verification, EDCIM uses a metacognitive EDR layer to selectively trigger correction, allowing finer resource control.

There has been a variety of approaches proposed to detect errors in LLM outputs, specifically focused on hallucinations (Huang et al. 2023; Su et al. 2024; Friel and Sanyal 2023). However, there has been limited work on detecting errors in the context of using LLMs to convert unstructured text into structured format for down-stream tasks. Works on such translation have noted high error rates involved in creating mathematical formulas (Imani, Du, and Shrivastava 2023; Yamauchi et al. 2023), combinatorial problems (Ye et al. 2023), and declarative logic programs (Pan et al. 2023a)(Gilpin et al. 2018). However, none of these works has focused on specific mechanisms to detect errors. Specific to the creation of mathematical formulae from unstructured text (focus of this paper) the work of Shakarian et al. (2023) shows that certain characteristics of math word problems can be used to predict LLM errors with a simple classification model while the work of Ngu, Lee, and Shakarian (2024) estimates LLM uncertainty by computing metrics like entropy and Gini impurity over multiple responses to the same prompt. We build on these ideas to create a broader framework for both error detection and correction, allowing the system to learn which empirically observed error conditions are most relevant for a particular dataset. The work on symbolic error detection rules has been applied in a variety of tasks including image classification (Kricheli et al. 2024), movement trajectory classification (Xi et al. 2023) and time-series prediction (Lee et al. 2024). However, to our knowledge it has not been applied to the output of LLM's.

The EDCIM Method

Our method, EDCIM (**E**rror **D**etection and **C**orrection for **I**nterpretable **M**athematics), focuses on the task of solving mathematical problems by generating explicit systems of equations, which can then be used to compute the final answer via a symbolic solver. This task is significantly more challenging than simply generating the final numerical result, as it requires the model to produce interpretable outputs that humans can fully understand and verify. Our system follows a 4-step process. First, an LLM generates an initial system of equations based on the natural language (NL) description of a mathematical problem provided as input. In the second step, a symbolic error detection model analyzes this initial output, identifying errors and producing insights for each detected one. These insights are then used in the third step, where an LLM leverages this feedback to produce a revised, corrected version of the equation system. Finally, the corrected equations are passed to a symbolic solver, which computes the final numerical solution. See Figure 1 for an overview.

More formally, EDCIM takes as input a natural language text T, which is first processed by a large language model, $\mathcal{LLM}_1$ to generate a set of equations, X_1 ($\mathcal{LLM}_1 : T \mapsto X_1$). This initial system of equations is then analyzed by an error detection module, EDR, which identifies potential errors and produces a corresponding set of explanations and suggestions. EDR output is then converted into natural language and concatenated in the text J ($EDR_{NL} : X_1 \mapsto J$). In the next step, the justifications J are concatenated to the original text T and the initial equations X_1 forming the input for a second large

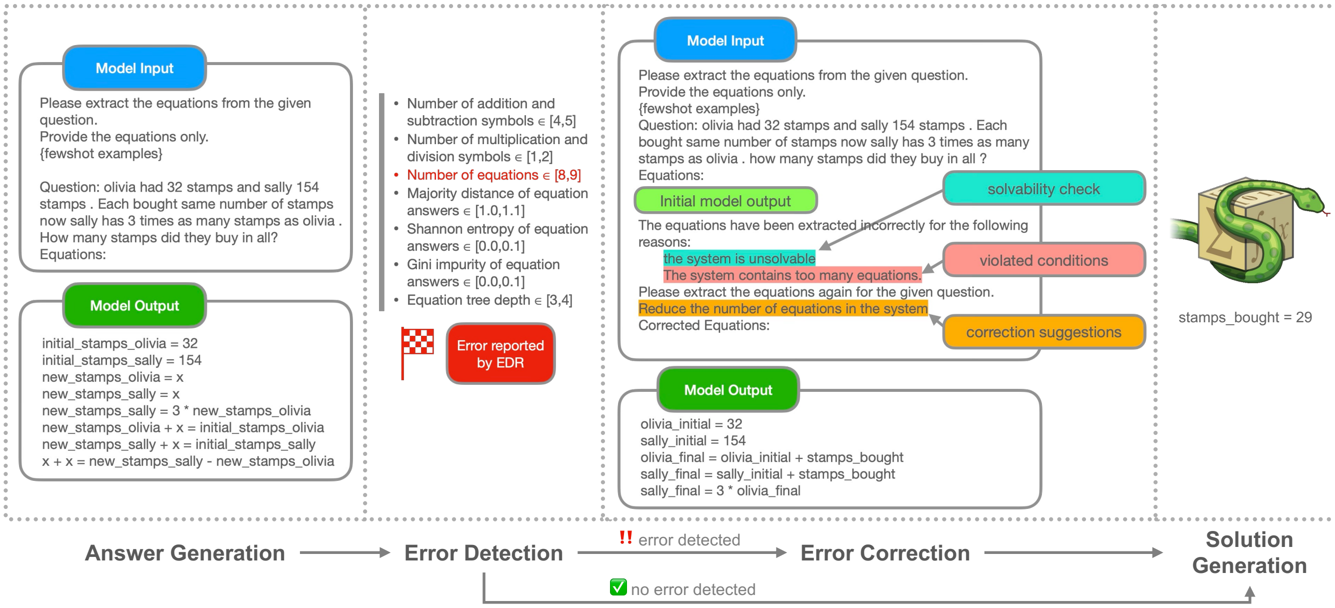


Figure 1: Overview of our system: (1) an LLM generates an initial system of equations from the problem description; (2) symbolic error detection identifies errors and produces justifications and guidance; (3) an LLM uses this information for context-aware error correction; and (4) the corrected equations are passed to a symbolic solver to generate the final numerical solution.

language model, $\mathcal{LLM}_2$, which generates a revised set of equations, X_2 ($\mathcal{LLM}_2 : (T, X_1, J) \mapsto X_2$). Finally, this corrected system of equations X_2 is passed to a symbolic solver to compute the final numerical solution, S ($solve : X_2 \mapsto S$). The complete system can be summarized as: $S = solve(\ \mathcal{LLM}_2(T,\ \mathcal{L}_1(T),\ EDR_{NL}(\mathcal{LLM}_1(T))\ )$. In what follows, we present a detailed overview of each component in our framework, followed by the implementation details used in our experiments.

Answer Generation

Answer generation is the first step in our pipeline, where a large language model $\mathcal{LLM}_1$ is employed to transform the initial, unstructured natural language text of the mathematical problem into a structured set of equations. The goal is to produce a complete system of equations X_1 that accurately captures the mathematical relationships described in the input text, as proposed in recent works for mathematical (Gou et al. 2023) or logical reasoning (Pan et al. 2023a). An example of this process can be found in Figure 1. This is achieved by providing the LLM with carefully designed prompts (see Appendix B of the ext. version) that include custom instructions and few-shot examples to guide the model towards the desired output format. In particular, we instruct the LLM to produce equations in the same format required by the downstream symbolic solver, ensuring compatibility.

To guide the LLM in generating outputs and to respect the required syntax, we incorporate a set of few-shot examples. These examples are manually curated to cover a broad range of mathematical word problem (MWP) types, inspired by external sources such as high school math textbooks. Specifically, we reviewed several textbooks and noticed that MWPs are often organized into chapters based on problem types. Following this approach, we categorized the MWPs into classes and selected a representative example for each category, including: (1) geometric; (2) numeric operations (addition, subtraction, multiplication, and division); (3) monetary; (4) chemistry solutions and mixtures; (5) age; (6) rate: speed, distance and time (e.g., $speed * time = distance$); and (7) rate: work and time (e.g., $rate * time = work\_done$). Details in Appendix A of the ext. version. However, despite providing explicit syntax guidelines and curated few-shot examples, the outputs generated by the LLM might still not always match the required format. To address this, we employ a parser to clean, simplify, standardize, and interpret the generated equations, ensuring they are properly formatted to match the expected solver syntax.

Error Detection

In the second step, we leverage EDR (Kricheli et al. 2024) to evaluate the system of equations X_1 generated by the LLM and identify potential errors. EDR (Metacognitive Error Detection Rules) is a neuro-symbolic approach that combines human-designed rules with statistical supervision for error detection. EDR framework takes a set of predefined error detection rules, and if the preconditions of one or more of these rules are satisfied, the generated equations are flagged as containing errors. The specific rules that are relevant for a given dataset are identified in advance during training, starting from a manually defined set of candidates. The learning process is described in what follows.

Using the first-order logic notation introduced by Xi et al. (2023) and by Kricheli et al. (2024), we denote one or more LLM responses to a prompt for a given query as X. We say some condition (denoted $cond \in C$) is true for sample X as $cond(X)$. In the domain of generating mathematical equations, an example condition for a single sample could be the number of addition operations exceeding a certain amount. An example where X is a set of LLM responses (to the same prompt) is that the entropy of the set of responses falls within a certain range. We denote as C' the subset of conditions C that are known to cause errors (e.g., identified by analyzing responses generated from a set of prompts sent to the LLM, where the ground truth is known). Based on this subset, we can specify a set of error detection rules as follows: $cond \in C' : error(X) \leftarrow cond(X)$.

Using ideas from the literature (Ngu, Lee, and Shakarian 2024; Shakarian et al. 2023), we first define the set of candidate conditions C that capture potential error patterns. We then identify a subset of conditions C' most likely to cause errors on a specific dataset. This subset is learned by evaluating the relevance of each candidate condition on a training set. A key feature of this approach is that when a rule executes on a given LLM response, we have precise knowledge of which conditions were present. This information is later leveraged to develop custom prompts that enable fine-grained and informed LLM-based correction.

Algorithm 1 reviews the DetRuleLearn rule-learner of Xi et al. (2023), adapted for our use case. Given N training samples, for a subset $C' \subset C$, let $POS_{C'}$ denote the number of samples with at least one condition in C' causes the LLM to produce an error, and $NEG_{C'} = N - POS_{C'}$. The Recall Reduction Threshold hyperparameter, $\varepsilon \in (0, 1]$, is a critical parameter for error detection as it controls the balance between detection coverage and precision. A higher value of ε allows for a more aggressive detection strategy, resulting in a larger number of detected errors but potentially increasing the rate of false positives. In contrast, a lower ε value prioritizes precision, reducing the rate of false positives but potentially missing some errors. DetRuleLearn leverages ideas from constrained submodular optimization and we refer the reader to Xi et al. (2023) and Shakarian, Simari, and Bastian (2025) for a formal treatment of this algorithm in a classification setting.

Algorithm 1: DetRuleLearn (Xi et al. 2023)

```
Input: Recall reduction threshold ε, Condition set C
Output: Subset of conditions C'
C' := ∅
C* := {c ∈ C s.t. NEG_{c} ≤ ε · N }
while C* ≠ ∅ do
    c_best = arg max_{c∈C*} POS_{C'∪{c}}
    Add c_best to DC_i
    C* := {c ∈ C \ C' s.t. NEG_{C'∪{c}} ≤ ε · N }
end while
return C'
```

Error correction. A key component of our approach is the ability to generate custom, context-aware prompts based on detected errors, using a set of interpretable Error Detection Rules (EDRs). These rules not only identify potential failure modes in the output of the LLM (e.g., overuse of mathematical symbols, incorrect structure, too much diversity among multiple queries) but also provide instructional feedback that can be directly translated into new prompts. The correction process begins by constructing a new prompt, which is formed by concatenating the original query T, the initial solution set X_1, and the insights provided by the triggered error detection rules. This prompt is then passed to an LLM, allowing it to generate a revised solution while being informed about the specific causes of the detected errors in a fine-grained manner.

The insights provided by EDR are converted in two main natural language components: violated conditions and recovery suggestions. Violated conditions describe the error detection rules that were triggered during the evaluation of the initial solution. These are converted in natural language by using predefined templates that we manually created when defining the set C of candidate rules for the EDC framework. Recovery suggestions provide targeted guidance for correcting the identified errors. These, also defined using natural language templates, are created during the initial definition of the candidate detection rules. For example, if the error is "The system contains too many equations", the corresponding recovery suggestion might be "Reduce the number of equations in the system". Like the violated conditions, these templates are manually prepared, but it is important to note that they could also be automatically generated using an LLM. In addition to the recovery suggestions, we include a fixed one-shot example from the training set, which shows how to correct errors based on the triggered rules for a specific instance. The corrections for this example were manually written, and cover only a subset of the full set of conditions considered in our framework.

The final correction prompt is then constructed by concatenating, in the following order, the original query T, the initial system of equations X_1, the violated conditions block and the recovery suggestions including the few-shot example. This prompt is then used to guide the LLM in producing the corrected solution X_2. As with the answer generation step, the corrected output is then parsed to ensure its consistency with the required syntax. In Figure 1, we show an example of the re-prompt for our running example, where the dynamic template components are marked by different colors. It is important to note that EDCIM only attempts to apply a correction if an error is detected. If no errors are identified, the original system of equations remains unchanged, resulting in $X_1 = X_2$.

Solution Generation

In the final step of our framework, we rely on an external solver that takes in input the corrected system of equations X_2, produced by $\mathcal{LLM}_2$, and outputs the final numerical solution S. The symbolic solver guarantees a correct solution to the input system, provided that the system is well-defined and consistent. In cases where the input system is unsolvable, the solver returns an empty set, indicating no possible solutions. If the system instead is ill-defined (e.g., syntax errors), the solver will produce an error. Both of these cases will be tagged as incorrect solutions. If the system of equations is under-determined, the solver identifies multiple valid solutions. In this case, for evaluation purposes, we check if the ground-truth solution appears in this set.

Local Open-Source LLMs vs. Cloud-Based Proprietary LLMs

Our framework combines the advantages of lightweight, open-source LLMs running locally with the superior performance of larger, proprietary models accessed via API. The goal is to have an effective balance between computational cost, financial efficiency, and predictive accuracy. We refer to small, open-source models deployed on local hardware with low computational requirements as *local LLMs*. These models offer zero additional inference costs and offline accessibility, though their performance is typically limited by smaller model sizes and constrained computational resources. In contrast, we refer to powerful, proprietary models hosted on commercial platforms as *cloud-based LLMs*. These models leverage extensive computational resources, providing superior reasoning and problem-solving capabilities. However, they come with substantial usage costs, making them less cost-effective for high-volume tasks (such as big datasets).

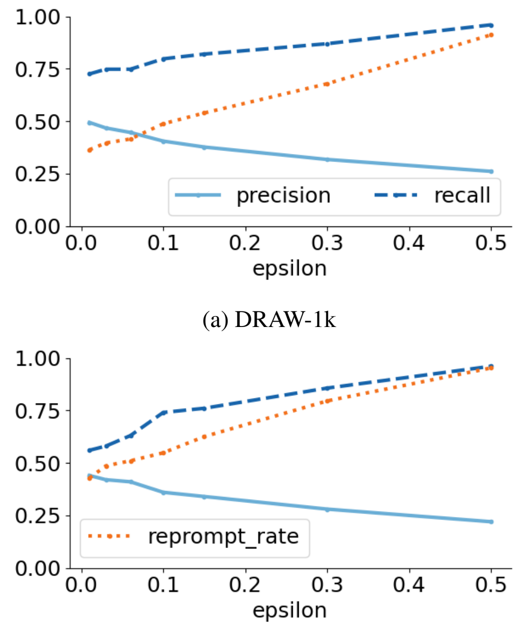
The goal of our approach is to optimally combine these two types of LLMs to maximize accuracy while minimizing cost. The key setting that we considered is where a local LLM is used to generate an initial, approximate solution. If this initial output is correct, it is accepted, reducing computational and financial costs, but if errors are detected, it is refined using a more powerful cloud-based LLM. Another relevant setting is where we enhance the performance of an LLM, whether local or cloud-based, by enabling it to iteratively correct and refine its own outputs.

Experimental Set-up

For our experiments, we utilize different LLM models: For the local-based, we use the `Phi-3 Mini 128K` model (Abdin et al. 2024), a compact, open-source LLM specifically optimized for reasoning tasks; For the cloud-based models, we submit API requests to `GPT4o` (OpenAI) (Achiam et al. 2023) and `DeepSeek-V2-R1` (DeepSeek) (Liu et al. 2024), both of which are larger, more powerful models known for their advanced reasoning capabilities. The answer generation is performed 10 times for each sample to evaluate the diversity among responses. The answer correction is only called if an error has been identified and the LLM is only queried once for each sample. All local model inference experiments were conducted on an NVIDIA A100 GPU.

We use SymPy (Meurer et al. 2017) as our symbolic solver, a Python library for symbolic mathematics that we use to solve the mathematical equations generated by the LLMs. It also serves as the backbone for our custom parser, which attempts to fix common syntax errors to correctly load the equations. This involves tasks like equation manipulation and simplification. While our parser handles the majority of cases, it is not 100% accurate, meaning that a small number of outputs may still be incorrectly formatted and thus classified as errors.

In the generation prompt, we instruct the LLM to act as a math assistant that converts word problems into raw mathematical equations. The prompt specifies that the output should consist solely of the equations, without any explanations, labels, or additional text, with each equation on a separate line. Variable names should be limited to letters and underscores, and only SymPy-compatible mathematical operators ($+$, $-$, $*$, $/$, $=$) are permitted (e.g., $age\_sarah = 2 * age\_brother \backslash n\ age\_sarah + age\_brother = 27$). Details in Appendix B of the ext. version.



(a) DRAW-1k

(b) GSM-8k

Figure 2: Effect of ε on detection precision, recall and re-prompt rate. Higher ε values increase EDCIM's detector recall but reduce precision. Zero re-prompting means only local correction. Results are shown for DRAW-1K (a) and GSM-8K (b) using Phi-3 as generator and GPT-4o as corrector.

Detection Rules

The set C of candidate conditions is defined starting from a collection of general rule categories, or meta-rules, which capture common error patterns identified in the literature (Xi et al. 2023). Each condition within the set C is a specific grounding, or parameterization, of a meta-rule, created by assigning numerical values to the intervals boundaries that define that rule. For example, the meta-rule "Number of equations $\in [a, b]$" can be grounded in several conditions in C such as "Number of equations $\in [4, 5]$" or "Number of equations $\in [8, 9]$".

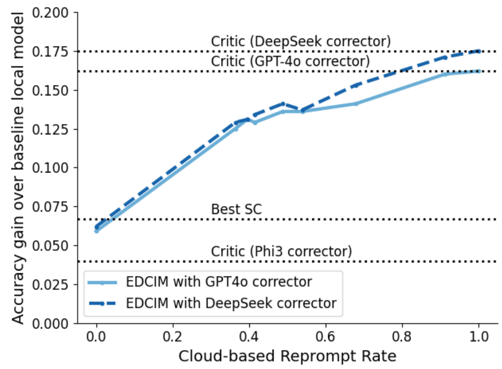
In this work we consider two primary categories of meta-rules, identified based on empirical observations and domain knowledge. (1) *Equation Complexity Measures* describe structural aspects of the equations, such as the depth of the equation parsing tree and the number of operators. More complex equations, with a deeper tree or with a higher number of mathematical symbols, correspond to more difficult questions that thus carry a higher likelihood of containing an error. In this category we define different types of meta-rules, including the average depth of the equations parsing tree, the number of additions and subtractions, the number of multiplications and divisions, and the total number of equations. (2) *Diversity Measures* analyze the response consistency when querying LLMs multiple times on the same problem. If a model produces highly variable outputs across multiple queries on the same input problem, it indicates a lack of confidence in its responses, increasing the likelihood of errors. In this category we have different types of meta-rules, such as Shannon Entropy (a measure of the variability in the LLM's responses), Gini Impurity (a measure of the likelihood of incorrect classification within a set of responses), and Jaccard distance to the core set of solutions (i.e., the Jaccard distance between the solutions of each system of equations and the largest common core solution set for individual variables, computed across multiple responses).

Each category defines multiple fine-grained rules by grounding of its intervals parameters (as mentioned above), forming the pool C of candidate detection rules. During training, EDR error detector learns the optimal parameter intervals for each meta-rule, selecting the ranges that most effectively distinguish correct from incorrect outputs. Thus, from the pool of candidate conditions C, a subset C' is learned, containing the conditions most likely to indicate errors for a specific dataset. This learnt configuration is then used to classify responses in the test set.

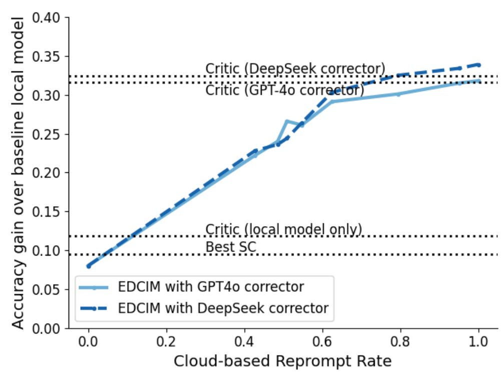
Datasets and Baselines

We conduct our experiments on two widely used mathematical benchmark datasets: DRAW-1K (Upadhyay and Chang 2016) and GSM-8K (Cobbe et al. 2021). DRAW-1K (DiveRse Algebra Word problems) contains 1000 algebra word problems crawled from algebra.com. Each problem is annotated with both the ground-truth formulae leading to the final numerical solution and the solution itself. GSM-8K (Grade School Math problems) is a dataset of 8500 math word problems written by human authors and released by OpenAI. Each problem is accompanied by a detailed solution that includes the intermediate reasoning steps needed to reach the final answer, from which both the ground-truth equations and the final numerical solution can be recovered. Given that our error detection method, since symbolic, does not rely on large training sets, we modified the official train-test splits into a 1:9 train-test ratio, which we found to be the most effective configuration based on preliminary experiments (see Appendix C of the ext. version). While state-of-the-art models achieve high performance on these benchmarks, our focus is to validate a cost-saving, controllable correction methodology.

We compare EDCIM against different methods: (1) **LLM only**: running an LLM one time per sample; (2) **SC**: running an LLM multiple times (10 times in our experiments) and adopting self-consistency (SC) which aggregates the multiple answers via majority voting; (3) **SC+Solv.**: a refined variant of SC that performs majority voting only among solvable answers; and (4) **CRITIC** (Gou et al. 2023): a framework that self-corrects an LLM output by performing one or more LLM-critique-and-revise loops with feedback from external tools (e.g., Python interpreters). In our experiments we re-prompt each sample once.



(a) DRAW-1k



(b) GSM-8k

Figure 3: Trade-off between accuracy gain over the baseline local model Phi-3 (answer generator) and re-prompt rate, as cloud-based re-prompt rate (of the error corrector model) varies (controlled by ε). Results are shown for DRAW-1K dataset (a), and GSM-8K dataset (b).

Experimental Results

Our experimental results can be summarized as follows: (1) Using local models for initial solutions and selective cloud-based queries for error correction significantly reduces costs, with only a small trade-off in accuracy: EDCIM re-prompts only about one-third of cases, unlike state-of-the-art that re-query every sample; (2) EDCIM provides flexibility by allowing users to adjust the trade-off between cost and performance through a single hyperparameter ε, providing full control over this balance; (3) Even when the corrector LLM is unable to fully correct the system of equations, it still improves the overall quality, bringing the equations closer to the ground truth.

Method	Answer Generator	Error Corrector	DRAW-1k		GSM-8k	
			ACC	re-prompt %	ACC	re-prompt %
LLM only	GPT4o	-	91.6	-	84.4	-
LLM only	DeepSeek	-	92.4	-	86.2	-
LLM only	Phi3	-	75.2	-	52.2	-
SC	Phi3	-	78.8	-	60.4	-
SC+Solv.	Phi3	-	81.9	-	61.7	-
CRITIC	Phi3	Phi3	79.2	-	64	-
	Phi3	GPT4o	91.4	100	83.8	100
	Phi3	DeepSeek	92.7	100	84.6	100
EDCIM	Phi3	Phi3	78.8	-	66.2	-
	Phi3	GPT4o	85.7	36	74.4	43
	Phi3	DeepSeek	87.8	36	75.0	43

Table 1: Comparison of our method EDCIM with baselines (LLM only, SC and SC-solvable) and the state-of-the-art CRITIC (Gou et al. 2023) framework using all considered LLMs (Phi-3, GPT4o, and DeepSeek) and optimal value of ε. Reported metrics include Accuracy (ACC) and cloud-based re-prompt rate (percentage). EDCIM achieves the best trade-off between these two metrics.

Effect of Hyperparameter Selection for Re-prompt Rate Control

Figure 2 illustrates how the ε hyperparameter in EDCIM's detector affects precision, recall, and re-prompt rate, using Phi-3 as answer generator and GPT4o as error corrector. This parameter controls the sensitivity of error detection, creating a trade-off where higher ε values increase recall by detecting more errors, but at the cost of reduced precision due to more false positives. In contrast, lower ε values improve precision but reduce recall, as more errors go undetected. The hyperparameter ε provides flexibility by allowing users to adjust the trade-off between cost and performance. This is achieved by indirectly controlling the re-prompt rate, as higher ε values increase the likelihood of re-prompting, leading to higher accuracy at a greater cost. Moreover, as shown in Figure 2 the relationship between ε and the re-prompt rate is approximately linear, making this control highly predictable. Generally, directly controlling the re-prompt rate is challenging, and existing state-of-the-art systems lack effective mechanisms for this. In contrast, our approach, which uses EDC error detector to indirectly manage the re-prompt rate via the ε parameter, offers the ability to balance between computational expense and performance.

Figure 3 shows the comparative performance of EDCIM, the baselines, and CRITIC against the local model (Phi-3) on DRAW-1K and GSM-8K datasets, as the re-prompt rates vary (indirectly with ε). It shows the trade-off between comparative accuracy (calculated as the difference between the ACC of each model and the ACC of the local model, Phi-3) and the average number of cloud-based queries. We achieve the different re-prompt rates by sampling the values of ε by grid search in $(0, 0.5]$ with more density in low values (and manually adding 100% re-prompt rate). From this figure, we can observe that as the re-prompt rate increases, the comparative accuracy shows a linear improvement. Notably, at a 100% re-prompt rate, our method slightly outperforms CRITIC.

Given the results of these two experiments, we choose $\varepsilon = 0.1$ as the optimal value for both DRAW-1K and GSM-8K, as it has a good balance between precision and recall across these two datasets. Further improvements can be obtained by fine-tuning ε value separately for each dataset.

System Performance at $\varepsilon = 0.1$

Table 1 reports the comparison, in terms of Accuracy (ACC) and cloud-based re-prompt rate, of EDCIM with the baselines and the state-of-the-art CRITIC framework. We used all considered LLMs (Phi-3, GPT4o, and DeepSeek) and the optimal value of ε defined in Section . Our results indicate that EDCIM selectively re-prompts only about one third of cases, significantly reducing reliance on expensive cloud-based queries when compared to CRITIC, which re-prompts each sample. Re-prompting in 100% cases achieves high accuracy but at substantial computational and financial cost, while using only local models avoids cloud-based queries but results in lower accuracy. EDCIM optimally balances local and cloud-based queries, extracting maximal value from cost-free local queries before resorting to cloud-based corrections. Overall, EDCIM demonstrates superior cost efficiency while maintaining competitive accuracy. Moreover, on the GSM-8k dataset, EDCIM outperforms the state-of-the-art when only the local Phi-3 model is used for both answer generation and correction.

Solution Improvement Analysis

We analyze the extent to which EDCIM can improve the quality of the generated equations, even in cases where it fails to produce a fully correct result. To evaluate the quality of the generated equations we measure the distance between the ground truth equations and both the original and corrected equations. We define a distance metric, *Equation Distance* $EqD(X, Y)$ between two systems of equations X and Y which is computed as the average of the following three metrics: (1) The distance between their numerical solutions; (2) The structural distance, measured as $1 - \frac{|N_1 - N_2|}{\max N1, N2}$

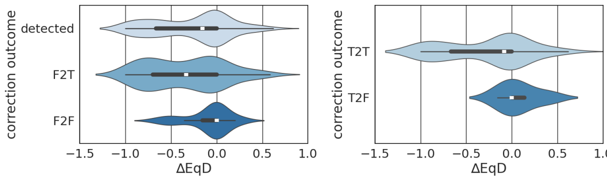


Figure 4: Change in Equation Distance ΔEqD, before and after correction on DRAW-1K. The middle bar represents the median. Negative values indicate equations moved closer to the ground truth. Results use Phi-3 for answer generator and GPT4o for error correction and $\varepsilon = 0.1$.

where N_1 and N_2 are the total number of nodes in the parsing trees of the two systems; and (3) The complexity distance, calculated as $\frac{|O_1 - O_2|}{\max O1,O2}$ where O_1 and O_2 are the number of operators in the two systems.

We then evaluate the improvement in distance to the ground truth system of equations, defined as the difference $\Delta EqD = EqD(X_2, X^*) - EqD(X_1, X^*)$ where X_1 is the initial set of equations, X_2 is the corrected set (as defined in Section), and X^* is the ground truth. Negative values indicate that the correction step moved the equations closer to the ground truth, while positive values indicate the opposite.

Figure 4 shows the values of ΔEqD when using Phi-3 as answer generator and GPT4o as error corrector on DRAW-1K dataset. The results show that re-prompting generally produces equations closer to the ground truth. For this analysis, we categorize the correction outcomes as follows: (1) **F2T (False-to-True)**, where the initial response was incorrect but corrected to the correct solution; (2) **T2F (True-to-False)**, where a correct response was incorrectly modified; (3) **T2T (True-to-True)**, where the response remained correct; (4) **F2F (False-to-False)**, where the response remained incorrect; and (5) all generated equations flagged as errors by EDCIM's detector, regardless of their correctness.

We observed significant improvement in the quality of solutions in all detections, with the bulk of improvement occurring in successfully corrected errors (F2T). However, we also noted improvement in incorrect detection cases (T2T) where results from EDCIM were also found to be closer to the ground truth, likely refining equation structure without altering correctness. F2F cases are particularly interesting; while they remain incorrect, our analysis shows that the correction step more often improves the solution quality than degrades it. This suggests that even unsuccessful corrections can provide incremental value, reinforcing the overall effectiveness of our framework. We note that false-positive detections improperly corrected (T2F, carrying the risk of overcorrection) were infrequent in the analysis. Overall, these results confirm the effectiveness of our correction framework in reducing errors, even if only partially. To validate this finding, we conducted a study using several alternative methods to measure equation distance, including multiple graph embedding techniques and vector distance functions. Details in Appx. F, ext. version.

Ablation Studies

We conduct a series of ablation studies to evaluate the actual impact of different components in our framework. Specifically, we analyze the effects of various detector models, rule types, and prompt designs on EDCIM performance. Details in Appendix D of the ext. version.

Impact of Error Detector Type. We evaluated the impact of different error detection models, including full re-prompting, ground truth-based correction, equations solvability checks, and EDR without and with equations solvability checks. Our results indicate that integrating both learned error detection and solvability checks, as used in EDCIM, provides the most accurate and comprehensive correction, reducing false negatives and capturing a broader range of error types.

Impact of Prompt Type. We modified the prompt by removing prompt components (e.g., few-shot examples, correction suggestions, etc.) resulting in different settings, including the standard method employed in EDCIM and static prompts that do not take into account dynamic, example-dependent cues. The results indicate that the standard setting used in EDCIM provides the highest overall performance; violated conditions and correction suggestions lead to very similar additional information and that static prompts lead to the worst performance overall, highlighting the importance of targeted example-dependent feedback.

Impact of Rule Type. We excluded some rules meta-rules category (see Section) from the set C to see if they are both impacting error detection accuracy. The results show that using both rule types increases the number of detected errors, resulting in higher re-prompt rates and overall accuracy, despite a slight reduction in precision.

Details in Appendix D of the ext. version.

Final Remarks and Future Work Discussion

To conclude, in this work, we introduced EDCIM, an error detection and correction framework for LLMs that uses a symbolic-based and interpretable error detection mechanism(EDRs). Unlike prior work, EDCIM selectively triggers corrections, allowing users to control the balance between cost and accuracy through a single hyperparameter. Our results show that this approach significantly reduces costs, with only a small trade-off in accuracy, and even partial corrections tend to improve the quality of the generated solutions. While our experiments focused on math problems, this approach is broadly applicable to other structured outputs such as logic, combinatorial optimization, and program synthesis. Future directions include automatic EDR learning, deeper integration with symbolic solvers, and extending the framework to new tasks like theorem proving and complex knowledge extraction. We also plan to assess performance on more challenging benchmarks, such as the MATH dataset, with the latest generation of LLMs. Notably, our initial pilot experiments suggest EDCIM can also improve state-of-the-art models (e.g., DeepSeek and GPT-4o), a direction best explored on these more complex datasets where performance is not already near-saturated.

Acknowledgments

This research was supported by DARPA cooperative agreement HR00112420370 (MCAI) and by Army Research Office (ARO) grant W911NF-24-1-0007.

References

Abdin, M.; Aneja, J.; Awadalla, H.; Awadallah, A.; Awan, A. A.; Bach, N.; Bahree, A.; Bakhtiari, A.; Bao, J.; Behl, H.; et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

An, S.; Ma, Z.; Lin, Z.; Zheng, N.; Lou, J.-G.; and Chen, W. 2023. Learning from mistakes makes llm better reasoner. *arXiv preprint arXiv:2310.20689*.

Cobbe, K.; Kosaraju, V.; Bavarian, M.; Chen, M.; Jun, H.; Kaiser, L.; Plappert, M.; Tworek, J.; Hilton, J.; Nakano, R.; Hesse, C.; and Schulman, J. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168*.

Cornelio, C.; Stuehmer, J.; Hu, S. X.; and Hospedales, T. 2023. Learning where and when to reason in neuro-symbolic inference. In *The Eleventh International Conference on Learning Representations (ICLR)*.

Didolkar, A.; Goyal, A.; Ke, N. R.; Guo, S.; Valko, M.; Lillicrap, T.; Jimenez Rezende, D.; Bengio, Y.; Mozer, M. C.; and Arora, S. 2024. Metacognitive capabilities of llms: An exploration in mathematical problem solving. *Advances in Neural Information Processing Systems*, 37: 19783–19812.

Flavell, J. H. 1976. Metacognitive aspects of problem solving. In *The nature of intelligence*, 231–236. Routledge.

Friel, R.; and Sanyal, A. 2023. Chainpoll: A high efficacy method for llm hallucination detection. *arXiv preprint arXiv:2310.18344*.

Gilpin, L. H.; Bau, D.; Yuan, B. Z.; Bajwa, A.; Specter, M.; and Kagal, L. 2018. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, 80–89. IEEE.

Gou, Z.; Shao, Z.; Gong, Y.; Shen, Y.; Yang, Y.; Duan, N.; and Chen, W. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.

He-Yueya, J.; Poesia, G.; Wang, R. E.; and Goodman, N. D. 2023. Solving math word problems by combining language models with symbolic solvers. *arXiv preprint arXiv:2304.09102*.

Huang, L.; Yu, W.; Ma, W.; Zhong, W.; Feng, Z.; Wang, H.; Chen, Q.; Peng, W.; Feng, X.; Qin, B.; et al. 2023. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions, 2023. *arXiv preprint arXiv:2311.05232*.

Imani, S.; Du, L.; and Shrivastava, H. 2023. Mathprompter: Mathematical reasoning using large language models. *arXiv preprint arXiv:2303.05398*.

Kamoi, R.; Zhang, Y.; Zhang, N.; Han, J.; and Zhang, R. 2024. When can llms actually correct their own mistakes? a critical survey of self-correction of llms. *Transactions of the Association for Computational Linguistics*, 12: 1417–1440.

Kricheli, J. S.; Vo, K.; Datta, A.; Ozgur, S.; and Shakarian, P. 2024. Error detection and constraint recovery in hierarchical multi-label classification without prior knowledge. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, 3842–3846.

Lee, N.; Ngu, N.; Sahdev, H. S.; Motaganahall, P.; Chowdhury, A. M. S.; Xi, B.; and Shakarian, P. 2024. Metal Price Spike Prediction via a Neurosymbolic Ensemble Approach. *arXiv preprint arXiv:2410.12785*.

Lewkowycz, A.; Andreassen, A.; Dohan, D.; Dyer, E.; Michalewski, H.; Ramasesh, V.; Slone, A.; Anil, C.; Schlag, I.; Gutman-Solo, T.; et al. 2022. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35: 3843–3857.

Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; et al. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36: 46534–46594.

Meurer, A.; Smith, C. P.; Paprocki, M.; Čertík, O.; Kirpichev, S. B.; Rocklin, M.; Kumar, A.; Ivanov, S.; Moore, J. K.; Singh, S.; Rathnayake, T.; Vig, S.; Granger, B. E.; Muller, R. P.; Bonazzi, F.; Gupta, H.; Vats, S.; Johansson, F.; Pedregosa, F.; Curry, M. J.; Terrel, A. R.; Roučka, v.; Saboo, A.; Fernando, I.; Kulal, S.; Cimrman, R.; and Scopatz, A. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3: e103.

Mishra, K.; Soliman, T.; Ramakrishna, A.; Galstyan, A.; and Kumar, A. 2024. Correcting Language Model Outputs by Editing Salient Layers. In *Findings of the Association for Computational Linguistics: EACL 2024*, 1295–1305.

Ngu, N.; Lee, N.; and Shakarian, P. 2024. Diversity measures: Domain-independent proxies for failure in language model queries. In *2024 IEEE 18th International Conference on Semantic Computing (ICSC)*, 176–182. IEEE.

Pan, L.; Albalak, A.; Wang, X.; and Wang, W. Y. 2023a. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. *arXiv preprint arXiv:2305.12295*.

Pan, L.; Saxon, M.; Xu, W.; Nathani, D.; Wang, X.; and Wang, W. Y. 2023b. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188*.

Paul, D.; Ismayilzada, M.; Peyrard, M.; Borges, B.; Bosselut, A.; West, R.; and Faltings, B. 2023. Refiner: Reasoning feedback on intermediate representations. *arXiv preprint arXiv:2304.01904*.

Shakarian, P.; Koyyalamudi, A.; Ngu, N.; and Mareedu, L. 2023. An independent evaluation of ChatGPT on mathematical word problems (MWP). *arXiv preprint arXiv:2302.13814*.

Shakarian, P.; Simari, G. I.; and Bastian, N. D. 2025. Probabilistic Foundations for Metacognition via Hybrid-AI. *arXiv preprint arXiv:2502.05398*.

Su, W.; Wang, C.; Ai, Q.; Hu, Y.; Wu, Z.; Zhou, Y.; and Liu, Y. 2024. Unsupervised real-time hallucination detection based on the internal states of large language models. *arXiv preprint arXiv:2403.06448*.

Upadhyay, S.; and Chang, M.-W. 2016. Annotating derivations: A new evaluation strategy and dataset for algebra word problems. *arXiv preprint arXiv:1609.07197*.

Upadhyaya, N.; and Sridharamurthy, R. 2024. Internalized Self-Correction for Large Language Models. *arXiv preprint arXiv:2412.16653*.

Xi, B.; Scaria, K.; Bavikadi, D.; and Shakarian, P. 2023. Rule-based error detection and correction to operationalize movement trajectory classification. *arXiv preprint arXiv:2308.14250*.

Yamauchi, R.; Sonoda, S.; Sannai, A.; and Kumagai, W. 2023. LPML: llm-prompting markup language for mathematical reasoning. *arXiv preprint arXiv:2309.13078*.

Ye, X.; Chen, Q.; Dillig, I.; and Durrett, G. 2023. Satlm: Satisfiability-aided language models using declarative prompting. *Advances in Neural Information Processing Systems*, 36: 45548–45580.

Zhao, Y.; Zhang, H.; Si, S.; Nan, L.; Tang, X.; and Cohan, A. 2023. Investigating Table-to-Text Generation Capabilities of LLMs in Real-World Information Seeking Scenarios. *arXiv preprint arXiv:2305.14987*.