Do LLMs Understand Constraint Programming? Zero-Shot Constraint Programming Model Generation Using LLMs

Yuliang Song and Eldan Cohen

Department of Mechanical and Industrial Engineering,
University of Toronto, Toronto, Canada
yl.song@mail.utoronto.ca, ecohen@mie.utoronto.ca

Abstract. Large language models (LLMs) have gained significant attention for their ability to solve complex tasks such as coding and reasoning. In this work, we aim to evaluate their ability to generate constraint programming (CP) models in a zero-shot setting, emphasizing model correctness and conformity to user-specified output formats. We propose a novel, iterative approach for zero-shot CP modeling that translates natural language problem descriptions into valid CP models and supports solution extraction to pre-defined output formats to facilitate effective adoption by domain experts and enable automated performance evaluation. To evaluate our approach, we introduce the Constraint Programming Evaluation (CPEVAL) benchmark[1], derived from a diverse set of CP problems in CSPLib, coupled with an automated evaluation suite for large-scale assessment. We augment CPEVAL with paraphrased variants to assess robustness across linguistic variation and mitigate bias in the evaluation due to data memorization. Our extensive experiments across eight prominent LLMs and two CP modeling languages, MiniZinc and PyCSP3, show that our proposed iterative Two-Step method significantly enhances model correctness and conformity to user-specified output formats. Furthermore, we observe that larger LLMs demonstrate superior performance, with DeepSeek-R1 emerging as the top performer across both CP languages. We also observe that LLMs generally perform better in MiniZinc than in PyCSP3.

Keywords: Constraint Programming · Large Language Models · Benchmark · MiniZinc

1 Introduction

Constraint Programming (CP) has proven to be a powerful paradigm for formulating and solving complex combinatorial problems across diverse domains [29]. Traditionally, when a project arises, domain experts articulate the problem requirements, while CP experts formalize these into constraint models and implement them in code [12]. However, the expertise required to formulate and

[1] We will make our benchmark publicly available upon acceptance.

encode CP models has been widely recognized as a bottleneck, limiting the broader adoption of CP [26, 23, 33]. This challenge highlights the need for automated, adaptive tools to simplify the process of generating CP models from natural language descriptions.

In recent years, there has been growing interest in addressing this bottleneck by harnessing large language models (LLMs) for CP modeling [23, 5]. Pre-trained on extensive web-scale corpora, LLMs have demonstrated remarkable capabilities in question answering, code generation, and reasoning [10, 38, 16]. This positions them as promising candidates for the "holy grail" of programming: a scenario in which domain experts simply provide a natural language description of a problem, and an LLM-based modeling assistant then produces the corresponding CP model [13]. Yet, despite their potential and often impressive outputs, LLMs do not guarantee correctness, consistency, or even relevance [18, 32, 1].

Existing evaluations commonly rely on publicly available problems, raising concerns that LLMs might recall memorized solutions from training rather than genuinely modeling the problem [7, 36]. To enable a more rigorous evaluation of LLM capabilities, we propose a benchmark for CP modeling tasks that assesses LLMs' internal knowledge without relying on similar problems or reference models to guide the modeling process. Moreover, domain experts require not only correct solutions but also outputs that conform to specific formatting instructions, facilitating straightforward interpretation, verification, and integration into their workflows. To address this, we introduce a CP modeling workflow emphasizing both model correctness and conformity to user-specified output formats.

In this work, we present the first comprehensive zero-shot CP modeling approach using LLMs, validated across eight prominent LLMs, two widely-used modeling languages, and a comprehensive benchmark involving linguistic variations. Specifically, we make the following contributions:

1. We propose a novel, iterative approach for zero-shot CP modeling that translates natural language problem descriptions into valid CP models and supports solution extraction to pre-defined output formats to facilitate effective adoption by domain experts and enable automated performance evaluation.
2. We present a new constraint programming evaluation (CPEVAL) benchmark comprised of diverse CP problems from CSPLib, augmented with paraphrased variants to assess robustness across linguistic variation and mitigate the impact of data memorization. This benchmark also includes an automated evaluation suite to support large-scale evaluation of model correctness and conformity to user-specified output formats.
3. We conduct extensive experimental analysis across eight prominent LLMs and two popular CP modeling languages, PyCSP3 [20] and MiniZinc [25], and show: (1) our proposed approach significantly enhances model correctness and output format conformity; Notably, DeepSeek-R1 excels in both MiniZinc and PyCSP3. (2) Although LLMs exhibit varying levels of CP modeling capabilities, they all experience performance degradation on paraphrased tasks, indicating data memorization may bias evaluations and underscoring the need to evaluate LLMs beyond publicly available benchmarks.

2 Background

2.1 Constraint Programming

CP is a powerful paradigm for solving combinatorial problems by representing them in terms of decision variables, their domains, and constraints that define relationships among these variables [34]. A classical constraint satisfaction problem (CSP) involves finding assignments of values to a set of variables X from their respective domains D such that all constraints C are satisfied. Extending this framework, a constraint optimization problem (COP) incorporates an objective function O to optimize while still satisfying all specified constraints. We denote a CP model as $\mathcal{M}$ and it's solution as $\mathcal{A}$ such that $\mathcal{A} = \mathcal{M}(X, C, D, O)$.

2.2 Large Language Models

Language Models (LMs) are probabilistic models parameterized by θ and trained on extensive corpora to predict the probability distribution of the next token, conditioned on a given prompt and the sequence of tokens generated so far. Given a prompt $I = (x_1, x_2, \ldots, x_n)$ of length n, where each x_i is a token in the input sequence, the model predicts each token y_t in the output sequence at time step t by conditioning its probability on the input prompt I and the previously generated tokens $y_{<t} = (y_1, y_2, \ldots, y_{t-1})$. The joint probability of a sequence $y = (y_1, y_2, \ldots, y_l)$, where l is the maximum allowed length of the sequence is:

$$p_\theta(y \mid I) = \prod_{t=1}^{l} p_\theta(y_t \mid y_{<t}, I) \tag{1}$$

LMs generate text in an autoregressive manner by selecting the next token y_t at each time step t based on the conditional distribution $p_\theta(y_t \mid y_{<t}, I)$, typically via sampling. Given logits $\phi(y_t \mid y_{<t}, I)$ for each possible next token y_t, the conditional probability is computed as:

$$p_\theta(y_t \mid y_{<t}, I) = \frac{\exp\left(\frac{\phi(y_t|y_{<t},I)}{T}\right)}{\sum_{y'} \exp\left(\frac{\phi(y'|y_{<t},I)}{T}\right)} \tag{2}$$

where T is the sampling temperature. A value of $T < 1$ sharpens the distribution, making the model more deterministic by favoring high-probability tokens. Conversely, a higher temperature of $T > 1$ flattens the distribution, encouraging the model to explore a broader range of potential outputs [10].

LLMs improve over the capabilities of standard LMs by significantly increasing the complexity of the model and its training data. Typically built on transformer architectures, LLMs incorporate billions of parameters and are trained on massive and diverse text corpora sourced from the web and specialized domains. This scaling enables LLMs to capture nuanced linguistic patterns, extensive world knowledge, and advanced reasoning capabilities [18]. In practice,

prompting acts as the interface to guide LLM behavior through natural language instructions, where zero-shot methods rely solely on task descriptions and the LLM's pre-trained knowledge, while few-shot methods provide contextual exemplars to implicitly define task patterns, enabling in-context learning [8].

Self-Improvement. Using LLMs to solve complex programming challenges in a single attempt often proves difficult [28]. To address this limitation, previous work has explored various forms of self-improvement techniques, which prompted the model to critically review and correspondingly improve its own outputs [22]. Under this framework, the model first generates an initial response and then iteratively refines it using additional feedback. This feedback may be derived from external sources, such as automated tests in code generation [11] or analysis of error messages [39]. Alternatively, feedback can be generated internally, for instance, through self-produced test cases [9] or self-assessment [35].

3 Zero-Shot CP Modeling with LLMs

We propose a workflow that leverages LLMs for automated CP modeling. Our system interprets task descriptions in natural language, translates them into CP models, and iteratively tests and refines the generated code.

3.1 Problem Definition

The input to our system consists of: (1) $\mathbf{P_{NL}}$: A natural-language description of the problem; (2) $\mathbf{P_{param}}$: A concise description of input parameters, specifying their meanings, data formats, and types; (3) $\mathbf{P_{OF}}$: Specifications of the expected output formats, namely a list of required variables and their description and data type, as well as an example output. Subsequently, our system, denoted as $\mathcal{F}$, transforms these components into a CP model $\mathcal{M}$, formally, $\mathcal{M} = \mathcal{F}(P_{NL}, P_{param}, P_{OF})$. See Fig. 1 for example P_{NL}, P_{OF}, and P_{param} for the well-known N-Queens problem.

The motivation for including explicit output format specification stems from the variety of valid equivalent ways to model the same problem (e.g., disjunctive vs. time-indexed formulation for scheduling problems [6]). Consequently, the representation of solutions can vary significantly, placing an additional burden on users who must interpret the generated model with potentially unfamiliar solution representations.

3.2 Iterative Modeling Workflow

We consider two workflows for iterative generation. First, we introduce the *Direct Instruction* method, which takes the problem context and explicitly instructs the LLM to generate code that solves the problem and outputs the solution in the user-specified output format. However, enforcing a complex output format

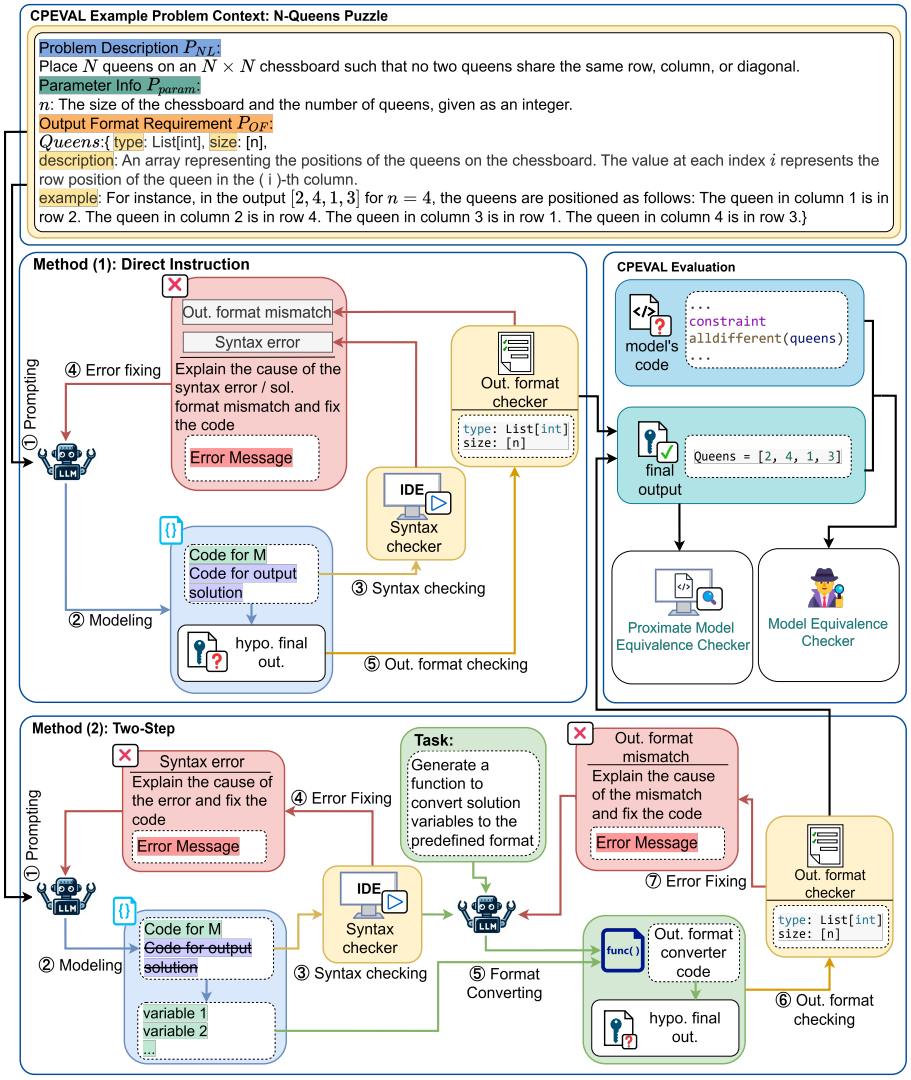

Fig. 1: Illustration of the proposed methods using the classic N-Queens problem from the CPEVAL benchmark.

increases the difficulty of the generation task, especially when the required format has an intricate structure (e.g., scheduling timetable or game board layout). To address this, we propose a *Two-Step* method, where the first step focuses on generating the CP model, and the second focuses on extracting the solution variables and transforming them into the user-specified output format. For effective output format validation, we implement an automated output format checker that takes the data types from P_{OF}, assesses the output's conformity to the

prescribed requirements, and produces error reports when it fails to meet the expected format.

Direct Instruction (DI) Fig. 1 (Method 1) illustrates the direct instruction workflow. The first step is to prompt the LLM with the problem context P, instructing it to generate a CP model in the chosen modeling language and output a final solution in the user-specified format P_{OF}. The generated code is extracted and compiled in an IDE with runtime information collected. If compilation fails, a self-improvement process (§3.3) is triggered to correct the code and reattempt compilation. Once the generated code passes the syntax check, it is executed, and the solver status is extracted upon completion or timeout. If the solver returns UNKNOWN or UNSATISFIABLE, the model is deemed semantically incorrect, and the generation workflow is aborted. Otherwise, if the solver returns a SATISFIABLE or OPTIMAL status, an output format checker evaluates whether the data types of the output variables align with the user-specified formats P_{OF}. Subsequently, if discrepancies are detected, an error message detailing the discrepancies is sent to the LLM, prompting it to revise the output format through the self-improvement process. The generated code is accepted once its outputs conform to the user-specified output format.

Two-Step Method (2S) Fig. 1 (Method 2) presents the Two-Step Method workflow. In the *modeling stage*, the problem context (P_{NL}, P_{param}) and user-specified output format (P_{OF}) are provided to the LLM for modeling. However, the LLM is explicitly instructed to only consider the output format requirements but not to generate any code to comply with them. The hypothesis code is then evaluated for syntax correctness, and if it fails, an iterative self-improvement process is triggered for debugging. Upon successfully passing the syntax check, the solver must return a SATISFIABLE or OPTIMAL status; otherwise, the model is deemed semantically incorrect, and the workflow is aborted.

Once the model is solved, all decision variables used in the model are saved to a local file, and the workflow proceeds to the *formatting stage*. The LLM is now instructed to generate Python code that transforms the stored decision variables into the user-specified format. This step requires the LLM to interpret the problem context and may involve generating code for additional calculations or adjustments to ensure compliance with P_{OF}. Similar to the DI method, the generated output is then validated by an output format checker, and if mismatches arise, a self-improvement process is triggered to correct them.

3.3 Self-Improving

We employ an iterative self-debugging approach following [11], as illustrated in Appendix C. This process targets two types of code defects: syntax errors identified through IDE compilation and output format mismatches detected by the output format checker. In the initial step, we input the defective code, along with any error messages, to the same LLM that generated the original code.

In case of syntax errors, the IDE's runtime error messages are used to guide debugging. In cases of output format mismatches detected by the output format checker, a message outlining discrepancies between the generated output and the expected format is provided to facilitate refinement. Subsequently, the LLM is instructed to provide a concise explanation of the error's cause and to produce a revised version of the code which is then executed in the corresponding IDE. If new errors are encountered, the updated code and corresponding error messages are fed back to the LLM for further review and correction. This iterative process continues until the hypothesis code passes its corresponding checker or a user-specified self-improving attempt limit is reached.

4 Experimental Setup

4.1 Models

We employed seven prominent open-source models including Llama-3.3-70B[2], DeepSeek-V3-685B [21] and it's reasoner variant, DeepSeek-R1 [15], QWen2.5-70B and QWen2.5-Coder-32B [37], Phi-3.5 mini (3.8B) [2], and Phi-4 (14.7B) [3], as well as the closed-source model ChatGPT-4o (2024-08-06)[3].

4.2 Datasets

We employ CSPLib [14] as the primary source of problems to construct the CPEVAL benchmark. Each problem consists of a natural language description, reference models in various CP languages, and instance data files, when applicable.

Due to time and computational cost constraints associated with large-scale evaluation, problems were selected based on criteria facilitating evaluation purposes. We included all problems that have a reference MiniZinc model and can be solved within 10 minutes. As a result, the CPEVAL dataset comprises 30 problems, including 9 COPs and 21 CSPs across seven categories and varying levels of complexity. Each problem in CPEVAL comprises three key components:

Problem Description ($\mathbf{P_{NL}}$) We pre-process the problem descriptions to exclude any images, references, unrelated information, and example solving steps.

Input Parameters ($\mathbf{P_{param}}$) The input parameters are derived from the CSPLib parameter file, each accompanied by a description of its data type, structure, and meaning. We verify each problem instance for validity; if multiple instances are available, we select up to three of the simplest ones based on computational complexity (e.g., preferring a 4-queens instance over a 100-queens instance). During modeling, the LLM is instructed to generate code that loads these parameters from the file, conditioned on this description. This approach prevents long, complex parameter structures from distracting the LLM and consuming unnecessary tokens, enabling the evaluation of more complex problems.

[2] https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/
[3] https://openai.com/index/gpt-4o-system-card/

Required Output Format ($\mathbf{P_{OF}}$) For each CSPLib problem, we provide a predefined output format requirement that states the data type, structure, and representation of all needed output variables.

4.3 Paraphrase Generation

The original CSPLib problems are widely known, and their solutions are publicly available, raising concerns that evaluation problems have been used to train the LLM, which could lead to over-estimation of models' performance due to memorization of the solution rather than CP modeling capabilities. To assess whether LLMs can effectively interpret a given problem context and generate correct CP models, we therefore opt to evaluate LLMs' performance under linguistic variations. Specifically, an LLM is instructed to paraphrase the original problem description to simulate how a user might request a modeling service from a CP expert. The goal is to introduce varied linguistic expressions and simulate different linguistic framings, allowing for the evaluation of the LLM on problems presented in unfamiliar or diverse linguistic forms, all while preserving the original underlying semantics. To ensure a diverse set of paraphrases that could effectively challenge the LLM's understanding and generalization capabilities, we introduce two paraphrasing styles: *Precision* mode, which minimally alters semantics while preserving technical accuracy, and *Colloquial* mode, which uses a casual, conversational tone closer to a layperson's request for assistance. For each original problem, we employ Claude-3.5-Sonnet[4] to generate three paraphrased versions in each style, with each paraphrase treated as a distinct problem instance, resulting in 180 paraphrased problems overall. Since generating paraphrases with the same LLM being evaluated could bias the paraphrases toward that model's language patterns and understanding, we excluded the Claude series from the evaluation. We note that the paraphrased descriptions are not guaranteed to be semantically equivalent to the original ones. In Appendix B, we analyze their semantic similarity using established metrics from the literature. A thorough human evaluation of semantic fidelity is left to future work.

4.4 Implementation Details

We evaluated the Direct Instruction Method and the Two-Step Method across the CP modeling languages PyCSP3 (using the ACE solver) and MiniZinc (using Gecode). A timeout of 10 minutes was set for each solver. For both syntax and output format errors, the number of self-improvement attempts was limited to 3. The Direct Instruction method was also compared against a Standalone Mode, where the LLM receives the same instruction prompt but is allowed only a single attempt to deliver the required output without iterative refinement.

For all LLMs, we set the temperature to zero for deterministic decoding and generate one model per problem. However, we observed that ChatGPT-4o, with a temperature setting of zero, did not follow greedy decoding, consistent with

[4] https://www.anthropic.com/claude/sonnet

prior findings [27]. To mitigate bias to a single given sample, we sampled multiple models per problem from ChatGPT-4o and reported average performance across samples. Specifically, five models were sampled for each original problem, and three models were sampled for each paraphrased problem.

We did not report model generation durations because models were accessed through different public cloud API providers with potentially varied infrastructure, making direct wall-clock time comparison unreliable.

4.5 Evaluation

To evaluate the system's performance in both modeling capability and generating output that aligns with user requirements, we focus on two criteria: (1) Output Format Alignment, which checks whether the generated output complies with the user-specified format; and (2) Model Equivalence, which verifies whether each generated model is semantically equivalent to a reference model M. We begin by introducing an output format alignment measurement, followed by a manual model equivalence check. However, manually interpreting and aligning diverse output formats is expensive and only feasible on a small scale. To enable large-scale evaluation, we also present an automated, unit-test-based approach that closely approximates our manual evaluation.

Output Format Alignment The required output variables are evaluated using the output format checker described in(§3.2). The output produced by a generated model is deemed to align with the required output format P_{OF} if it passes the output format checker, denoted as $FC(P_{OF}, \mathcal{A}) = 1$; otherwise, 0. The output format alignment rate (OFAR) across all N problems is then defined as:

$$\text{OFAR} = \frac{1}{N} \sum_{i=1}^{N} FC(P_{OF}, \mathcal{A}). \tag{3}$$

Manual Model Equivalence Checker We consider two CP models to be equivalent if they have semantically equivalent constraints and produce semantically equivalent solutions. To assess model equivalence, we performed manual evaluation from two perspectives: (1) Model alignment: we interpret the problem description, extract its key logical components, and determine whether the generated model aligns with the reference model in terms of constraints and objectives; (2) Solution verification: we then extract the final assignment of decision variables from the generated model and manually map them to the decision variables used in the reference model. Next, we insert these mapped values into the reference model to verify consistency with its constraints. Note that the model equivalence check does not require compliance with any specific output format; if the mapped assignment satisfies the reference model's constraints, it is considered a pass. Both perspectives must pass for a generated model to be considered equivalent, denoted as $\mathbb{I}(\hat{M}_i \equiv M_i) = 1$; otherwise, 0. Based on this, the *manual*

model equivalence rate (MMER), representing the fraction of generated models equivalent to their reference models across all N problems, is then defined as:

$$\text{MMER} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}(\hat{M}_i \equiv M_i) \tag{4}$$

Automated Model Equivalence Checker We approximate the human evaluation process by creating evaluation scripts that contain a set of unit tests for each problem. Specifically, these evaluation scripts require the generated code to output solutions in a predefined format and verify whether the solutions satisfy the problem's constraints and logic as outlined in the problem description. For COPs, the unit tests also recalculate the objective value from the output solution to verify optimality. If it passes the checker, it is deemed equivalent to the reference model, denoted as $\mathbb{PI}(\hat{M}_i \equiv M_i) = 1$; otherwise, 0. The *automated model equivalence rate* (AMER) is defined as:

$$\text{AMER} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{PI}(\hat{M}_i \equiv M_i) \tag{5}$$

5 Results

In Section 5.1, we present results from a small-scale manual evaluation on the original CPEVAL problems, validating the effectiveness of the automated model equivalence checker with the manual approach. As described in §4.5, we also examine output format alignment under our proposed methods to assess the conformity to the user-requested format. Then, in Section 5.2, we perform a large-scale automated evaluation across eight LLMs, evaluating their performance on the original CPEVAL problems as well as the paraphrased variants.

Table 1: Evaluation results on original problems using ChatGPT-4o with 5 samples per problem. Method marked with † indicates a standalone model without self-improvement.

Method	**Lang.**	**MMER** (%)	**OFAR** (%)	**MMER ∩ OFAR** (%)	**AMER** (%)	**Match** (Jaccard)
Direct Instr.†	mzn	50	49	31	31	1
	pycsp3	13	19	13	13	1
Direct Instr.	mzn	69	63	55	55	1
	pycsp3	53	60	53	53	1
Two-Step	mzn	67	79	67	66	0.98
	pycsp3	54	63	54	54	1

5.1 Small Scale Manual Evaluation

We start with a small-scale evaluation on the original CPEVAL problems using ChatGPT-4o with manual model equivalence checking, as presented in Table 1. Here, MMER∩OFAR denotes the proportion of hypothesis models that pass both the manual model equivalence check and the output format checker. "Match" denotes the Jaccard similarity between the set of problem instances passing MMER ∩ OFAR and the set of problem instances passing AMER.

Comparison of Modeling Methods Direct Instruction† (ChatGPT-4o in standalone mode) exhibits relatively low performance in generating semantically correct models in both MiniZinc and PyCSP3, with MMER scores of 50% and 13%, respectively. Upon investigation into the failed cases, we observed that this poor performance is primarily due to syntax errors stemming from the absence of a self-improvement process. Additionally, its ability to align with the predefined output format is weak, resulting in the lowest MMER ∩ OFAR score for both MiniZinc and PyCSP3. In contrast, Direct Instruction with self-refinement demonstrates a notable improvement in MMER scores for MiniZinc (50% → 69%) and PyCSP3 (19% → 53%), along with a significantly higher MMER ∩ OFAR scores for MiniZinc (31% → 55%) and PyCSP3 (13% → 53%).

While the Two-Step Method achieves comparable MMER scores to Direct Instruction in MiniZinc (67% vs. 69%), it is significantly more effective at ensuring correct output formatting, leading to a significantly higher OFAR score than Direct Instruction (79% vs. 63%). Additionally, the Two-Step Method consistently outperforms Direct Instruction in OFAR across all evaluated CP modeling languages and LLMs (Appendix D). Therefore, we focus on the Two-Step Method in the following sections.

Effectiveness of Automated Evaluation We gauge the effectiveness of the automated evaluation by comparing the alignment between the MMER ∩ OFAR score (models that are semantically correct and deliver solution in the predefined format) and the AMER score. Notably, across all methods and CP languages, AMER aligns closely with MMER ∩ OFAR. We also observe consistently high Match scores, with a value of 1 across all methods and CP modeling languages, except for the Two-Step Method on MiniZinc (0.98). This indicates that our automated model equivalence checker reliably identifies models that are semantically equivalent to their reference counterparts and output solutions in the predefined format.

5.2 Large Scale Automated Evaluation

Table 2 presents the results of our large-scale automated (unit test-based) evaluation for original and paraphrased problems across eight prominent LLMs. Overall, we observe that larger models tend to achieve higher AMER scores.

Table 2: Results of automated evaluation on original and paraphrased problems using the Two-Step Method.

Model	Params	Lang.	Original		Paraphrased	
			OFAR (%)	**AMER** (%)	**OFAR** (%)	**AMER** (%)
DeepSeek-R1	685B	MZN	80	80	80	74
		PYCSP3	67	63	61	54
DeepSeek-V3	685B	MZN	80	70	76	58
		PYCSP3	60	50	62	51
ChatGPT-4o	Unknown	MZN	79	66	76	65
		PYCSP3	63	54	59	47
llama3.3	70B	MZN	67	57	57	45
		PYCSP3	43	30	33	23
QWen2.5	70B	MZN	63	60	59	49
		PYCSP3	47	33	46	33
QWen2.5-Coder	32B	MZN	63	47	54	41
		PYCSP3	53	43	45	30
Phi-4	14.7B	MZN	20	17	16	13
		PYCSP3	37	20	33	26
Phi-3.5 mini	3.8B	MZN	0	0	3	0
		PYCSP3	7	0	9	2

Among all LLMs, DeepSeek-R1 attains the highest AMER score with MiniZinc (80%) and PyCSP3 (63%), followed by DeepSeek-V3 (70%) for MiniZinc and ChatGPT-4o (54%) for PyCSP3. Interestingly, QWen2.5-Coder outperforms Llama3.3 and QWen2.5, both significantly larger, on PyCSP3, demonstrating the potential benefits of code-related pre-training in Python code generation tasks. The smaller model Phi-4 obtains significantly lower AMER scores, whereas the even smaller Phi-3.5 mini struggles to generate valid CP models.

CP Languages Comparison We observe that modeling with MiniZinc consistently outperforms PyCSP3 in terms of the AMER score across all LLMs. This suggests that while LLMs demonstrate strong proficiency in generating Python code, this ability does not directly translate into effective CP modeling with PyCSP3. Moreover, we observe that the robustness of models to linguistic variations varies across the two modeling languages, as discussed below.

Performance on Paraphrased Problems All LLMs exhibit degradation in AMER when moving from the original problems to the paraphrased problems, though the extent of degradation varies. DeepSeek-R1 achieves the highest AMER for the original problems and experiences a moderate drop on paraphrased tasks for both MiniZinc (80% $\rightarrow$ 74%) and PyCSP3 (63% to 54%), though it still outperforms other LLMs. DeepSeek-V3, another strong performer, undergoes a notable performance drop on MiniZinc for paraphrased problems (70%$\rightarrow$58%) but maintains relatively stable AMER scores for PyCSP3 (50% $\rightarrow$ 51%). In contrast, ChatGPT-4o's AMER on paraphrased tasks with MiniZ-

inc remains closely aligned with its performance on the original problems (66% → 65%), while its results for PyCSP3 shows greater sensitivity to paraphrased problems with a significant drop in AMER (54% → 47%).

Among the remaining open-source models, Llama3.3-70B shows a consistent performance drop across all CP languages, while QWen2.5-70B demonstrates a notable decline on MiniZinc (60% → 49%) but remains stable on PyCSP3 (33%). Conversely, QWen2.5-Coder-32B experiences a significant AMER reduction on both MiniZinc (47% → 41%) and PyCSP3 (43% → 30%), highlighting differences in sensitivity across problem formulations and languages. Phi-4 drops on MiniZinc (17% → 13%) but improves on PyCSP3 (20% → 26%).

6 Related Works

Several studies have explored using LLMs to generate code for modeling and solving mathematical optimization problems from natural language descriptions. Ramamonjison et al. [30] proposed a workflow for assisting linear programming (LP) problem formulation based on natural language descriptions, followed by the introduction of the NL4Opt competition benchmark focused on LP problems [31]. Ahmaditeshnizi et al. [4] presented a workflow for modeling LP problems and proposed NLP4Opt, a benchmark encompassing LP and mixed-integer linear programming (MILP). In addition, Mostajabdaveh et al. [24] proposed a benchmark for LP, MILP, and quadratic programming (QP) with Zimpl, along with a multi-agent modeling system evaluated on that benchmark. Hao et al. [17] extract planning tasks from natural language and encode them as SMT optimization models to compute plans.

In the context of CP model generation from natural language descriptions, research remains comparatively limited. Almonacid et al. [5] performed a preliminary exploration of using ChatGPT-3.5 to generate MiniZinc code. However, different from our work, they do not focus on generating correct CP models for optimization problems based on natural language description. Instead, their approach is focused on prompting LLMs to generate MiniZinc code templates (e.g., "A source code with an array of discrete variables with domain and without constraints"). Tsouros et al. [33] presented a position paper discussing potential strategies for LLM-based CP modeling, though these ideas were not implemented or evaluated. Additionally, Lawless et al. [19] studied the use of LLMs to generate CP-based scheduling constraints in an interactive meeting scheduling system. The most relevant prior work is by Michailidis et al. [23], which focuses on CP modeling in CPMpy using a few-shot approach supplemented by retrieval-augmented generation (RAG). During modeling, their method provides the LLM with several similar problems and corresponding solution models to guide generation. In contrast, our workflow is designed for zero-shot CP model generation, relying solely on the LLM's internal knowledge augmented with an iterative modeling workflow that effectively boosts modeling performance without requiring similar example problems or models. Moreover, our comprehensive evaluation spans eight state-of-the-art LLMs, two CP modeling languages, and

attempts to mitigate evaluation bias due to data memorization, an aspect that was not considered in prior work on CP model generation.

7 Conclusion and Discussion

In this paper, we propose methods for leveraging LLMs for CP modeling, emphasizing both model correctness and conformity to output format specification. We also introduce CPEVAL, a benchmark with an automated evaluation framework designed to assess LLMs' performance in CP modeling. Through an empirical evaluation of our proposed workflows across eight prominent LLMs and two popular CP modeling languages, we observe that (1) our two-step method with iterative self-improvement significantly enhances the performance of LLMs and (2) DeepSeek-R1 outperforms all other evaluated models, and MiniZinc generally achieves better results than PyCSP3 across most LLMs. By paraphrasing the original CSPLib problem descriptions, we further examine the robustness of LLMs when presented with varied linguistic expressions and framings. Although all models exhibit a notable performance degradation on paraphrased problems, the extent of this degradation varies across both models and CP languages.

Our work raises several promising directions for future research: (1) Developing specialized prompting strategies that incorporate CP-specific knowledge to guide LLM in CP modeling can enhance the performance of our approach. (2) Transferring knowledge from larger LLMs with strong modeling and coding capabilities to enhance the performance of smaller, more efficient LLMs in CP tasks. (3) Extending CPEVAL with additional problems would enable more comprehensive evaluations of LLMs' capabilities in CP tasks. (4) Building on our iterative modeling framework and the CPEVAL suite, future work could explore reinforcement learning-based fine-tuning of LLMs using structured feedback signals (e.g., syntax errors, semantic failures, and output format mismatches) as a form of supervision.

References

1. Abbasi-Yadkori, Y., Kuzborskij, I., György, A., Szepesvari, C.: To believe or not to believe your llm: Iterativeprompting for estimating epistemic uncertainty. In: NeurIPS (2024)
2. Abdin, M., Aneja, J., Awadalla, H., Awadallah, A., Awan, A.A., Bach, N., Bahree, A., Bakhtiari, A., Bao, J., Behl, H., et al.: Phi-3 technical report: A highly capable language model locally on your phone. arXiv:2404.14219 (2024)
3. Abdin, M., Aneja, J., Behl, H., Bubeck, S., Eldan, R., Gunasekar, S., Harrison, M., Hewett, R.J., Javaheripi, M., Kauffmann, P., et al.: Phi-4 technical report. arXiv:2412.08905 (2024)
4. AhmadiTeshnizi, A., Gao, W., Udell, M.: Optimus: Optimization modeling using mip solvers and large language models. arXiv:2310.06116 (2023)
5. Almonacid, B.: Towards an automatic optimisation model generator assisted with generative pre-trained transformer. arXiv:2305.05811 (2023)

6. Azem, S., Aggoune, R., Dauzère-Pérès, S.: Disjunctive and time-indexed formulations for non-preemptive job shop scheduling with resource availability constraints. In: IEEE IEEM. pp. 787–791 (2007)
7. Balloccu, S., Schmidtová, P., Lango, M., Dušek, O.: Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms. In: EACL. pp. 67–93 (2024)
8. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)
9. Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.G., Chen, W.: Codet: Code generation with generated tests. In: ICLR (2023)
10. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv:2107.03374 (2021)
11. Chen, X., Lin, M., Schärli, N., Zhou, D.: Teaching large language models to self-debug. In: ICLR (2024)
12. Epstein, S.L., Freuder, E.C.: Collaborative learning for constraint solving. In: CP. pp. 46–60. Springer (2001)
13. Freuder, E.: In pursuit of the holy grail. ACM CSUR **28**(4es), 63–es (1996)
14. Gent, I.P., Walsh, T.: Csplib: a benchmark library for constraints. In: CP. pp. 480–481 (1999)
15. Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al.: Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv:2501.12948 (2025)
16. Hao, S., Gu, Y., Ma, H., Hong, J., Wang, Z., Wang, D., Hu, Z.: Reasoning with language model is planning with world model. In: EMNLP. pp. 8154–8173 (2023)
17. Hao, Y., Zhang, Y., Fan, C.: Planning anything with rigor: General-purpose zero-shot planning with llm-based formalized programming. arXiv preprint arXiv:2410.12112 (2024)
18. Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B., et al.: A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. ACM TOIS (2023)
19. Lawless, C., Schoeffer, J., Le, L., Rowan, K., Sen, S., St. Hill, C., Suh, J., Sarrafzadeh, B.: "i want it that way": Enabling interactive decision support using large language models and constraint programming. ACM TIIS **14**(3), 1–33 (2024)
20. Lecoutre, C., Szczepanski, N.: Pycsp3: modeling combinatorial constrained problems in python. arXiv:2009.00326 (2020)
21. Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al.: Deepseek-v3 technical report. arXiv:2412.19437 (2024)
22. Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al.: Self-refine: Iterative refinement with self-feedback. NeurIPS **36** (2024)
23. Michailidis, K., Tsouros, D., Guns, T.: Constraint modelling with llms using in-context learning. In: CP (2024)
24. Mostajabdaveh, M., Yu, T.T., Ramamonjison, R., Carenini, G., Zhou, Z., Zhang, Y.: Optimization modeling and verification from problem specifications using a multi-agent multi-stage llm framework. INFOR: Information Systems and Operational Research **62**(4), 599–617 (2024)
25. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: CP. pp. 529–543 (2007)

26. O'Sullivan, B.: Automated modelling and solving in constraint programming. In: AAAI. pp. 1493–1497 (2010)
27. Ouyang, S., Zhang, J.M., Harman, M., Wang, M.: An empirical study of the non-determinism of chatgpt in code generation. ACM TOSEM (2024)
28. Pan, L., Saxon, M., Xu, W., Nathani, D., Wang, X., Wang, W.Y.: Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. arXiv:2308.03188 (2023)
29. Petropoulos, F., Laporte, G., Aktas, E., Alumur, S.A., Archetti, C., Ayhan, H., Battarra, M., Bennell, J.A., Bourjolly, J.M., Boylan, J.E., et al.: Operational research: methods and applications. JORS **75**(3), 423–617 (2024)
30. Ramamonjison, R., Li, H., Yu, T., He, S., Rengan, V., Banitalebi-Dehkordi, A., Zhou, Z., Zhang, Y.: Augmenting operations research with auto-formulation of optimization models from problem descriptions. In: EMNLP. pp. 29–62 (2022)
31. Ramamonjison, R., Yu, T., Li, R., Li, H., Carenini, G., Ghaddar, B., He, S., Mostajabdaveh, M., Banitalebi-Dehkordi, A., Zhou, Z., et al.: Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In: NeurIPS 2022 Competition Track. pp. 189–203. PMLR (2023)
32. Simhi, A., Herzig, J., Szpektor, I., Belinkov, Y.: Distinguishing ignorance from error in llm hallucinations. arXiv:2410.22071 (2024)
33. Tsouros, D., Verhaeghe, H., Kadıoğlu, S., Guns, T.: Holy grail 2.0: From natural language to constraint models. arXiv:2308.01589 (2023)
34. Van Hentenryck, P., van Hoeve, W.J.: Constraint Programming, pp. 1–16. Springer International Publishing (2020)
35. Wang, Y., Zeng, J., Liu, X., Meng, F., Zhou, J., Zhang, M.: Taste: Teaching large language models to translate through self-reflection. arXiv:2406.08434 (2024)
36. Xu, R., Wang, Z., Fan, R.Z., Liu, P.: Benchmarking benchmark leakage in large language models. arXiv:2404.18824 (2024)
37. Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., et al.: Qwen2. 5 technical report. arXiv:2412.15115 (2024)
38. Yu, F., Zhang, H., Tiwari, P., Wang, B.: Natural language reasoning, a survey. ACM CSUR **56**(12), 1–39 (2024)
39. Zhong, L., Wang, Z., Shang, J.: Ldb: A large language model debugger via verifying runtime execution step-by-step. arXiv:2402.16906 (2024)

A CPEVAL Original Problems

Table 3 provides an overview of the CPEVAL problems selected from CSPLib.

Table 3: Overview of problems included in the CPEVAL dataset.

ID	Title	Category
1	Car Sequencing	Scheduling and related problems
2	Template Design	Bin packing Partitioning and related problems
3	Quasigroup Existence	Combinatorial mathematics
6	Golomb rulers	Combinatorial mathematics
7	All-Interval Series	Combinatorial mathematics
8	Vessel Loading	Design and configuration
10	Social Golfers Problem	Scheduling and related problems
12	Nonogram	Games and puzzles
14	Solitaire Battleships	Games and puzzles
15	Schur's Lemma	Combinatorial mathematics
16	Traffic Lights	Unclassified
18	Water Bucket Problem	Bin packing Partitioning and related problems
19	Magic Squares	Combinatorial mathematics
22	Bus Driver Scheduling	Scheduling and related problems
23	Magic Hexagon	Games and puzzles Design and configuration Combinatorial mathematics
24	Langford's number problem	Combinatorial mathematics
28	Balanced Incomplete Block Designs	Design and configuration Combinatorial mathematics
32	Maximum density still life	Games and puzzles
34	Warehouse Location Problem	Design and configuration Logistics
39	The Rehearsal Problem	Scheduling and related problems
41	The n-Fractions Puzzle	Games and puzzles Combinatorial mathematics
44	Steiner triple systems	Combinatorial mathematics
49	Number Partitioning	Combinatorial mathematics
50	Diamond-free Degree Sequences	Combinatorial mathematics
53	Graceful Graphs	Combinatorial mathematics
54	N-Queens	Games and puzzles
56	Synchronous Optical Networking (SONET) Problem	Network design
57	Killer Sudoku	Games and puzzles
67	Quasigroup Completion	Combinatorial mathematics
74	Maximum Clique	Combinatorial mathematics

B Paraphrase Problems

B.1 Paraphrase Modes

We define two paraphrase modes to generate paraphrases in diverse styles, both designed to maintain the original problem logic while introducing varied linguistic expressions and framings.

Precise Mode : The precise mode instructs the paraphrase LLM to rephrase the problem descriptions with minimal semantic variation, preserving the original meaning and technical accuracy without introducing any new information that or altering the problem's intent. This approach generates professional and formal rephrasing that retrain the technical rigor of the original descriptions while introducing subtle variations in linguistic expressions and framings.

Colloquial Mode : The colloquial mode instructs the paraphrase LLM to rephrase the problem descriptions in a casual and conversational tone, simulating how laypersons who lack domain-specific terminology might describe complex problems when seeking assistance. Unlike original CSPLib problems, which are typically expressed in formal and precise language, these paraphrases introduce relatively greater linguistic variation, incorporating colloquial expressions and simplified linguistic structures.

B.2 Paraphrase Generation

The original problem descriptions were paraphrased using claude-3.5-sonnet-20240620. For each original problem, twenty paraphrases were generated in each mode (*Precision* and *Colloquial*). Figure 2 illustrates the prompt employed for paraphrase generation.

B.3 Semantical Analysis of Paraphrase Problems

We evaluate the paraphrased problems from two perspectives: each paraphrase should exhibit linguistic variation from the original problem descriptions while preserving semantical similarity. Table 4 shows BLEU[5] scores and BERTScore[6] values for paraphrased problem descriptions compared to their original counterparts, along with self-BLEU scores to assess similarity among paraphrases within the same mode. The low BLEU scores indicate minimal n-gram overlap between the paraphrased and original descriptions, confirming that the paraphrases exhibit substantial lexical variation. Meanwhile, the relatively high BERTScores suggest that the paraphrases retain strong semantic similarity to the original descriptions.

[5] Papineni et al. (2002), "Bleu: a method for automatic evaluation of machine translation"

[6] Zhang et al. (2019), "Bertscore: Evaluating text generation with bert"

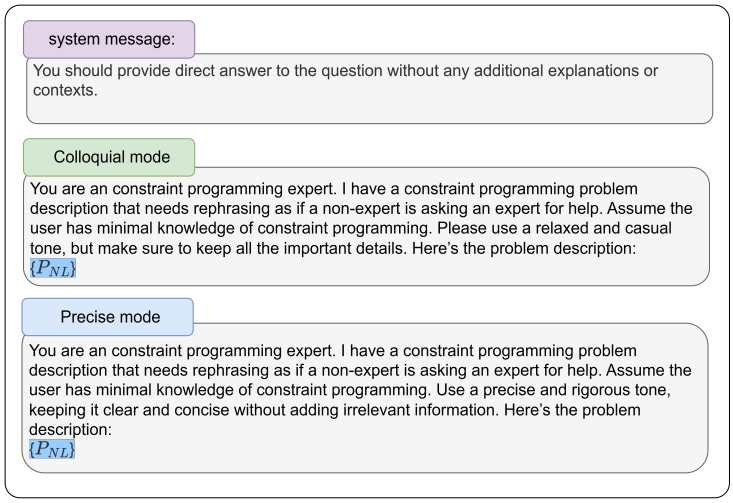

Fig. 2: Prompt used for paraphrase generation for the *Colloquial* mode and *Precise* mode

Table 4: Average content similarity metrics for problem descriptions. Paraphrased descriptions are compared to the original problem description, both collectively (Combined) and separately by mode.

Mode	BLEU	Self-BLEU	BERTScore
Combined	0.0476	0.1901	0.5869
Colloquial	0.0465	0.3254	0.5869
Precise	0.0488	0.3181	0.6150

C Self-Improvement

Figure 3 illustrates the iterative self-improvement process on example errors. In part (a), the LLM receives a syntax error message from the IDE, while in part (b), it receives an output format mismatch error message. In both cases, the LLM produces a revised version of the code and re-executes it in the IDE. This process continues iteratively until the corresponding checker is passed or the self-improvement attempt limit is reached.

D Large-Scale Evaluation

D.1 Augmented Output Format Alignment

Table 5 corresponds to Table 2 but includes results for both the Direct Instruction and Two-Step methods. According to the OFAR scores in Table 5, the Two-Step method demonstrates significantly better performance across all LLMs

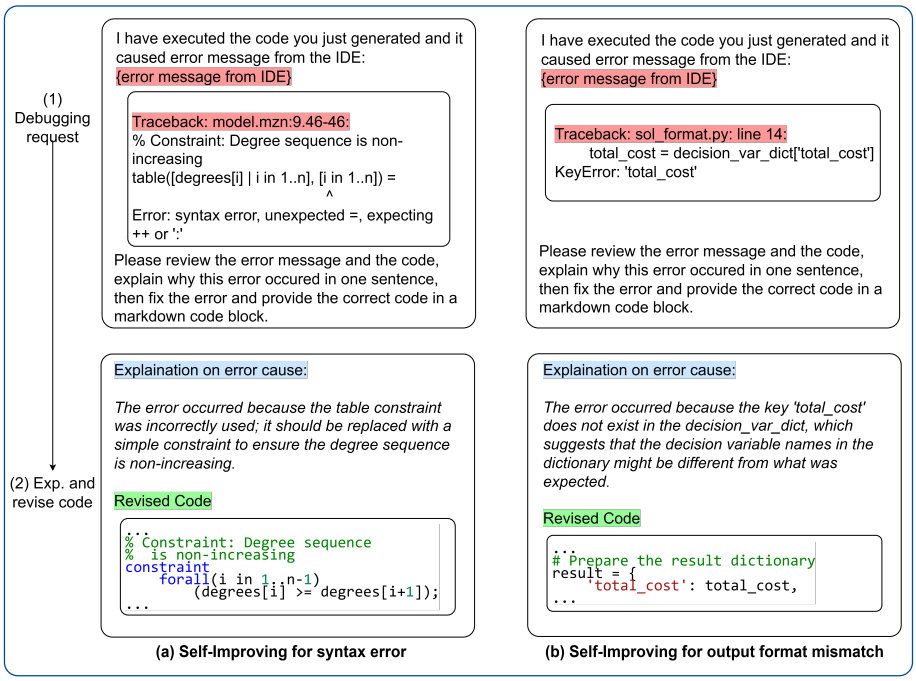

Fig. 3: Examples of Self-Improvement. Block (a) addresses syntax errors, while Block (b) handles output format mismatches based on feedback from the output format checker.

and CP languages than the Direct Instruction method in OFAR scores. This improvement is particularly pronounced with MiniZinc for medium-sized LLMs. For instance, Llama-3.3-70B and QWen2.5-70B nearly double their OFAR scores compared to the Direct Instruction method, increasing from 30% $\rightarrow$ 67% and 37% $\rightarrow$ 63%, respectively. For PyCSP3, the Two-Step method also significantly improves the OFAR scores of open-source models, though the improvement is less substantial than with MiniZinc.

Table 5: Evaluation of all LLMs across CP modeling languages on both original and paraphrased problems.

				Original		**Paraphrased**	
Model	**Params**	**Lang.**	**Methods**	**OFAR** (%)	**PMER** (%)	**OFAR** (%)	**PMER** (%)
DeepSeek-R1	685B	MZN	DI	70	70	71	66
			2S	80	80	80	74
		PYCSP3	DI	63	57	63	54
			2S	67	63	61	54
DeepSeek-V3	685B	MZN	DI	70	57	63	48
			2S	80	70	76	58
		PYCSP3	DI	63	50	62	50
			2S	60	50	62	51
ChatGPT-4o	Unknown	MZN	DI	63	55	63	56
			2S	79	66	76	65
		PYCSP3	DI	60	52	58	47
			2S	63	54	59	47
llama3.3-70B	70B	MZN	DI	30	20	30	22
			2S	67	57	57	45
		PYCSP3	DI	30	23	26	17
			2S	43	30	33	23
QWen2.5-70B	70B	MZN	DI	37	33	36	33
			2S	63	60	59	49
		PYCSP3	DI	33	20	29	17
			2S	47	33	46	33
QWen2.5-Coder	32B	MZN	DI	43	30	41	34
			2S	63	47	54	41
		PYCSP3	DI	20	20	21	17
			2s	53	43	45	30
Phi-4	14.7B	MZN	DI	7	3	4	4
			2S	20	17	16	13
		PYCSP3	DI	17	13	11	6
			2S	37	20	33	26
Phi-3.5 mini	3.8B	MZN	DI	0	0	2	0
			2S	0	0	3	0
		PYCSP3	DI	0	0	0	1
			2S	7	0	9	2