WHEN LLMS MEET NONLINEAR OPTIMIZATION: NED-TREE FOR END-TO-END OR MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Automating Operations Research (OR) modeling from natural language to executable solver code remains difficult when problem descriptions contain nonlinear expressions. Existing frameworks can often handle linear problems, but nonlinear terms introduce an additional semantic gap: mathematically plausible expressions must still be converted into solver-compatible expressions. To bridge this gap, we propose a structured LLM modeling framework centered on the Nonlinear Element Decomposition Tree (NED-Tree). NED-Tree organizes code generation around OR-specific construction requirements: objective and constraint relations, auxiliary-variable dependencies, operator domains, and solver-compatible operations. The framework (1) leverages dual-granularity element extraction to build traceable modeling elements from natural language, (2) exploits NED-Tree to decompose complex nonlinear expressions into linear backbones, auxiliary definitions, and atomic operator mappings, and (3) maps the decomposed representation into executable solver code with a repair stage. We also construct NEXTOR, a benchmark designed to evaluate nonlinear OR modeling under long descriptions, redundant information, and diverse nonlinear structures. Experiments on 10 benchmarks show that our framework achieves the best average pass@1 accuracy of 72.51%. On the NEXTOR nonlinear split, the framework reaches 92.11% accuracy and 100.00% pass rate, while also preserving the best accuracy and pass rate on the linear split. Code and data are available[1].

1 INTRODUCTION

Applying Operations Research (OR) methods to decisions described in natural language requires recovering a mathematical model and translating it into executable solver code (Saban & Weintraub, 2021; Qian & Yu, 2021; Trimborn et al., 2020; Jiang et al., 2025). This process arises in supply chain optimization (Corbett & DeCroix, 2001; Li et al., 2023), network deployment (Hong et al., 2016), and job-shop scheduling (Brandimarte, 1993). Practitioners must identify decision variables and parameters, formulate objectives and constraints, and implement these relations consistently in code. Refining the model and its implementation is technically demanding and time-consuming, creating a barrier for non-experts (Ghiani et al., 2022; Cornuejols et al., 2018; Rao, 2019).

Recent LLM-based methods automate this process through two complementary directions: fine-tuning on optimization data (Huang et al., 2025; Yang et al., 2024; Lu et al., 2025; Jiang et al., 2025) and prompt-based agent frameworks that divide modeling into reasoning, formulation, and coding stages (Xiao et al., 2024; AhmadiTeshnizi et al., 2024). Benchmarks have also expanded from linear tasks (Ramamonjison et al., 2023) to broader optimization scenarios (Huang et al., 2024; Xiao et al., 2024; Huang et al., 2025; Yang et al., 2025). Despite this progress, reliably translating nonlinear relations into solver code remains challenging: powers, ratios, exponentials, and logarithms introduce operator compositions and domain restrictions that must remain consistent during code generation. For example, implementing $\cos(\log x)$ requires preserving the composition order and $x > 0$, while its enclosing constraint must use a valid solver interface. Preserving these relationships during code generation requires linking expression composition to explicit solver construction steps.

Our observations in Section 3 identify two failure modes in nonlinear modeling. First, LLMs may omit variables, hallucinate parameters, or lose the correspondence between source descriptions and

[1] https://anonymous.4open.science/r/NEXTOR

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

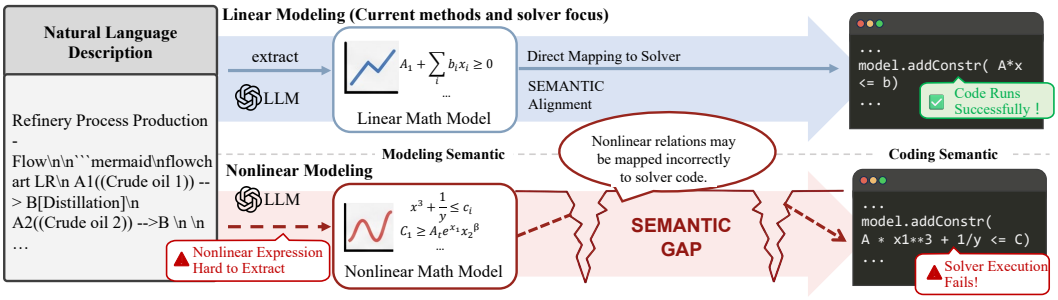


Figure 1: The nonlinear semantic gap in end-to-end LLM OR modeling. Linear relations (top) can often be mapped directly to solver code. For nonlinear problems (bottom), errors can arise when recovering modeling elements from natural language (Type I) or when constructing solver code for nonlinear relations (Type II). The latter requires preserving operator dependencies, domain restrictions, and the enclosing relation through valid solver interfaces.

extracted modeling elements. These modeling-semantic errors (Type I) arise during problem understanding and formulation. Alongside the connections between modeling elements used in existing frameworks (AhmadiTeshnizi et al., 2024), retaining sentence-level links makes their source descriptions explicit. Second, errors arise when the agent implements nonlinear relations as code, either by misrepresenting the relations or by using incompatible solver interfaces (Type II). Modern solvers already provide nonlinear interfaces (Gurobi Optimization, LLC, 2024); the remaining challenge is to construct the intended relations correctly through those interfaces. Figure 1 locates these failures at the two stages of the natural-language-to-code pipeline.

These observations motivate a framework centered on the Nonlinear Element Decomposition Tree (NED-Tree). Before code generation, NED-Tree separates each supported nonlinear expression into a Linear Backbone retaining the outer relation and auxiliary definitions exposing its local nonlinear operations. These definitions, their dependencies, and their atomic solver mappings give the CodingAgent explicit construction steps, with domain conditions carried through to code generation. NED-Tree thus complements the solver's expression support by organizing how the agent constructs the nonlinear model. Upstream, dual-granularity element extraction links modeling elements to the problem description and filters nonlinear expressions for decomposition. Downstream, code mapping implements the decomposed representation, and solver feedback supports repair of programming-level errors using the same structured context. Together, these stages organize code generation around OR modeling requirements: consistent modeling elements, explicit nonlinear relations, and solver-compatible constructions.

Our main contributions are as follows: 1) We characterize the nonlinear semantic gap through modeling-semantic and nonlinear code-semantic failures, and develop an end-to-end framework combining traceable extraction, nonlinear decomposition, and code mapping with repair; 2) We introduce the Nonlinear Element Decomposition Tree (NED-Tree), a decomposition representation for the CodingAgent that exposes the outer relation, auxiliary definitions, and atomic solver mappings as explicit construction steps before code generation; 3) We construct **NEXTOR**, a challenging evaluation environment with long-context OR descriptions and diverse nonlinear structures, complementing existing modeling benchmarks; 4) Evaluation on 10 benchmarks yields an average accuracy of 72.51%, including 92.11% on the NEXTOR nonlinear split. Ablations and controlled diagnostics assess component roles, expression preservation, and solver-interface mapping.

2 Related Work

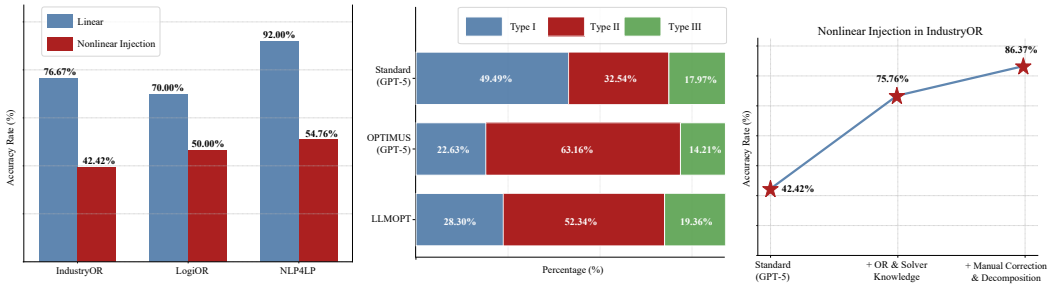
Automatic Modeling Framework for OR. LLM-based OR modeling follows two primary routes: prompt-based agent frameworks that decompose modeling into reasoning, formulation, and coding, with iterative reflection, variable-constraint relationships, and code repair (Yang et al., 2023; Xiao et al., 2024; AhmadiTeshnizi et al., 2024; Zhang & Luo, 2025; Liu et al., 2025; Song et al., 2026), and specialized models trained with synthesized optimization data (Ma et al., 2024; Huang et al., 2025; Yang et al., 2024; Lu et al., 2025; Jiang et al., 2025). CIR represents modeling rules and

IR2Solve compiles ModelIR (Lyu et al., 2026; Zhu et al., 2026). Tree-based decoders such as GTS (Xie & Sun, 2019) and Graph2Tree (Zhang et al., 2020) preserve compositional relations in math word problems, but target compact arithmetic expressions rather than solver-executable optimization models. Modern solvers also provide nonlinear expressions and composite expression trees through `NLExpr` and `addGenConstrNL` (Gurobi Optimization, LLC, 2024). NED-Tree builds on structured expression representation and complements these solver interfaces at the coding stage: it supplies the CodingAgent with the outer relation, auxiliary definitions, and atomic mappings needed to construct the model. Its focus is the explicit construction information presented to the agent before execution, evaluated against native and direct-AST paths in Section 5.4.

Evaluation Benchmarks for OR. OR benchmarks have expanded from linear programming (Ramamonjison et al., 2023) to longer descriptions, implicit constraints, MILP, and industrial contexts (Xiao et al., 2024; AhmadiTeshnizi et al., 2024; Huang et al., 2024; 2025). Recent datasets further cover nonlinear, tabular, and logic-heavy modeling (Yang et al., 2024; Lu et al., 2025; Yang et al., 2025). They cover diverse descriptions, applications, and mathematical structures.

3 Motivated Observations

In this section, we present key observations that motivate our decomposition method. (1) Existing LLM-based OR modeling methods often exhibit a substantial performance decline when nonlinear expressions are introduced. (2) The resulting failures are concentrated in modeling semantic errors and nonlinear code semantic errors. (3) The performance of LLMs in nonlinear OR modeling can benefit from OR knowledge and solver API guidance, with further gains observed from manual formulation correction and nonlinear decomposition in a separate motivation diagnostic. The experiments in this part are conducted on 300 controlled instances constructed from IndustryOR (Huang et al., 2025), LogiOR (Yang et al., 2025), and NLP4LP (AhmadiTeshnizi et al., 2024); complete data construction, modeling approaches, and evaluation settings are provided in Appendix D.



(a) Linear/Nonlinear Injection Average Accuracy (b) Proportion of 3 Error Types in Nonlinear Injection (c) Accuracy of Deeper OR Knowledge

Figure 2: Observational experiments on nonlinear modeling in existing methods. (a) OE1: Average accuracy on the original linear instances and their nonlinear variants across three benchmarks. (b) OE2: Proportions of three error types. Type I denotes modeling semantic errors, Type II denotes nonlinear code semantic errors, and Type III denotes other coding errors. (c) OE3: A separate manual motivation diagnostic on nonlinear IndustryOR, comparing the standard setting, OR knowledge and solver API guidance, and added manual formulation correction and nonlinear decomposition.

Nonlinear perturbations reduce modeling accuracy. Figure 2(a) shows that accuracy on the nonlinear variants drops by 28.57% to 44.67% relative to the original linear instances across the three benchmarks. Nonlinear expressions thus challenge end-to-end OR modeling.

Type I and Type II errors dominate the observed failures. Figure 2(b) shows that Type I and Type II errors jointly account for 80.64%–85.79% of the failed nonlinear cases. Type I captures missing or incorrect modeling elements, whereas Type II captures unreliable mappings from nonlinear relations to solver code. These results motivate traceable element extraction and explicit nonlinear mapping.

Manual assistance motivates structured modeling and decomposition. OE3 is a separate manually assisted motivation diagnostic on the IndustryOR nonlinear setting. OR knowledge and solver API guidance improve accuracy from 42.42% to 75.76%, while further correcting the mathematical formulation and manually decomposing its nonlinear expressions raises accuracy to 86.37% (Fig-

ure 2(c)). The final condition uses human-checked formulations, including corrections to variables, parameters, objectives, and constraints. The additional 10.61 percentage points therefore reflect the combined manual intervention; they do not isolate the effect of decomposition or measure the gain of automated NED-Tree. Section 5 separately evaluates the automated framework and its components.

Together, these observations motivate dual-granularity element extraction for Type I errors and NED-Tree for the Type II nonlinear modeling-to-code semantic gap, as detailed in Section 4.

4 LLM NONLINEAR OPTIMIZATION MODELING FRAMEWORK

The overall framework is illustrated in Figure 3. Its core component is the Nonlinear Element Decomposition Tree (NED-Tree), a solver-compatible nonlinear decomposition layer that organizes complex nonlinear terms as a Linear Backbone l_F, a New Definition Set D_{new}, and introduced auxiliary variables $\mathcal{V}_{new}$. Dual-Granularity Element Extraction collects traceable problem elements and identifies the problem type, including whether the extracted nonlinear elements trigger NED-Tree (Section 4.2). NED-Tree then separates each complex nonlinear expression into its main relational structure and atomic nonlinear operations (Section 4.3). Code mapping declares $\mathcal{V}_{new}$ and maps l_F and D_{new} to solver code; repair then addresses programming-level execution errors (Section 4.4).

4.1 DEFINITION OF NONLINEAR TERMS

Given an OR problem instance p, we specialize the corresponding optimization model as follows,

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{s.t.} \quad g_i(\mathbf{x}; \beta_i) \leq 0, \quad i = 1, \ldots, N, \tag{1}$$

where f represents the objective function, $\mathbf{x} \in \mathbb{R}^n$ denotes the decision variables, g_i signifies the i-th constraint function, and β_i stands for the corresponding parameters. Both f and g_i may contain nonlinear terms and complex nonlinear nesting or logical conditional branches.

Continuous atomic operators $\mathcal{A}_{cont}$. They encompass non-nested unary functions with unit coefficients and binary products, including $\exp(\cdot)$, $\ln(\cdot)$, $(\cdot)^k$, $\alpha^{(\cdot)}$, and basic bilinear products. **Logical atomic operators $\mathcal{A}_{logic}$.** They represent conditional branches and piecewise relations. NED-Tree expands the conditions and branch relations into explicit logical or auxiliary definitions, which the CodingAgent maps to solver constraints.

A complex nonlinear term is a function expression obtained through a finite number of linear and composite operations on these nonlinear atomic operators:

$$\mathcal{C}(\mathbf{x}) = \Phi(\mathcal{A}_1, \ldots, \mathcal{A}_k, \mathbf{x}). \tag{2}$$

Here, Φ represents the outer linear backbone and nested combinatorial logic. When the expression occurs in a constraint, it can be written as $\mathcal{C}(\mathbf{x}) \leq 0$. Any term resulting from the decomposition of a complex nonlinear term is referred to as a sub-nonlinear term. Modern solvers provide native interfaces for supported composite nonlinear expressions. NED-Tree organizes the end-to-end agent's mapping into atomic operators, auxiliary definitions, and the original relational structure before code generation. The end-to-end experiments use the `NEDTree-4R` implementation. Its tested scalar path covers addition, multiplication, powers, division and square roots through powers, exponential, logarithmic, sine, and cosine expressions. The controlled comparison in Section 5.4 evaluates scalar expressions; conditional-branch expansion is part of the end-to-end modeling pipeline. Appendix E provides complete operator definitions, the tested operator set, and required domain conditions.

4.2 DUAL-GRANULARITY ELEMENT EXTRACTION AND MODELING

Motivated by OE2 in Section 3, we propose a dual-granularity sentence-by-sentence extraction strategy for nonlinear modeling. OR models require globally consistent variable and parameter roles across sentence-level objectives and constraints. As shown in Figure 3(a), the strategy first performs Global bird's-eye extraction and then Local sentence-by-sentence extraction.

Global bird's-eye extraction. A large language model scans the complete natural-language problem description $p = \{s_1, \ldots, s_I\}$ and extracts the global parameter set P, the decision variable set $E^{(v)}$, and macroscopic information Info_{base}. In particular, Info_{base} records the problem type and determines whether downstream nonlinear decomposition is required.

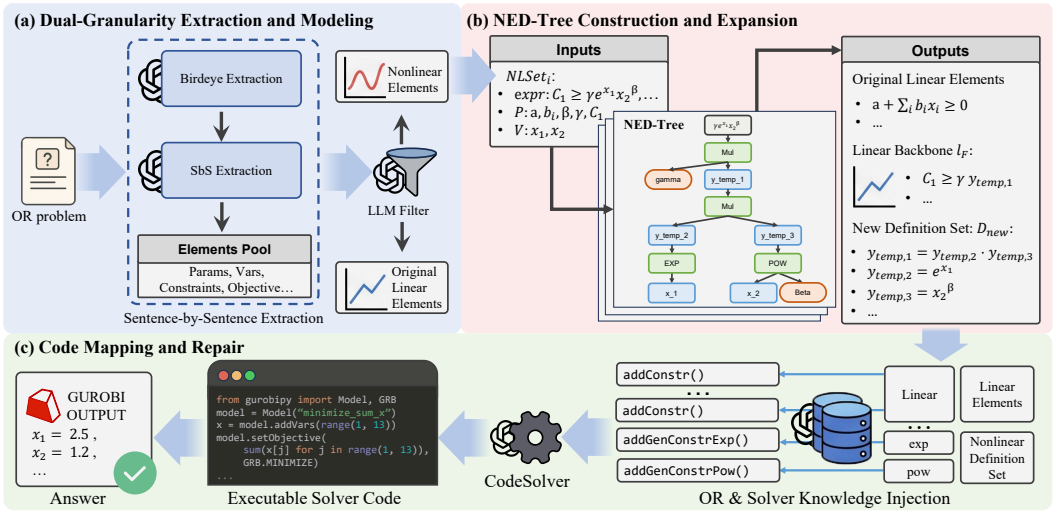


Figure 3: The NED-Tree framework contains three stages: (a) Dual-granularity extraction, which builds traceable variables, parameters, constraints, objectives, and problem-type information; (b) NED-Tree, which converts nonlinear expressions into the Linear Backbone l_F, the New Definition Set D_{new}, and introduced auxiliary variables $\mathcal{V}_{new}$; and (c) Code mapping and repair, which generates executable code from these outputs and repairs programming-level errors.

Local sentence-by-sentence extraction. Given the extracted variables and parameters, the large language model scans each sentence s_i and maps it into a constraint set $E_i^{(c)}$ and an objective function $E_i^{(o)}$. Detected conditional structures are stored in the same structured element pool. The two extraction stages and the subsequent filtering process are summarized as

$$\begin{aligned} \{P, E^{(v)}, \text{Info}_{base}\} &= LLM_{\text{Birdeye}}(p), \\ \{E_i^{(c)}, E_i^{(o)}\} &= LLM_{\text{SbS}}(s_i), \\ E_{Pool} = \{P, E^{(v)}, \text{Info}_{base}\} \cup \bigcup_{i=1}^{I} \{E_i^{(c)}, E_i^{(o)}\} &\xrightarrow{\text{Filter}} \{\text{NLSet}, E'_{Pool}\}. \end{aligned} \tag{3}$$

Detected nonlinear expressions are collected into NLSet and sent to NED-Tree. A linear element pool is mapped directly to solver code. The complete extraction equations and the treatment of conditional constraints are provided in Appendix F.

4.3 Nonlinear Element Decomposition Tree Construction and Expansion

After dual-granularity extraction, the framework obtains the nonlinear expression set NLSet. A nonempty NLSet activates NED-Tree, which separates each nonlinear expression's outer relation from its local operations in a solver-compatible representation (Figure 3(b)).

Given expression $expr$, parameters $\mathcal{P}$, and variables $\mathcal{V}$, NED-Tree returns

$$l_F, D_{new}, \mathcal{V}_{new} = \text{NEDTree}(expr, \mathcal{P}, \mathcal{V}), \tag{4}$$

where l_F denotes the Linear Backbone, D_{new} denotes the New Definition Set, and $\mathcal{V}_{new}$ denotes the introduced auxiliary variables. For a scalar expression, the Linear Backbone retains the objective or outer constraint relation after nonlinear sub-expressions are replaced by auxiliary variables. It is affine in the original and auxiliary variables, while the nonlinear operations remain in their defining relations. The New Definition Set stores their definitions and expands composite nonlinear structures into atomic nonlinear operations. The outputs l_F and D_{new} are mapped and solved together; the complete formulation generally remains nonlinear.

Construction of the Linear Backbone. NED-Tree first normalizes the input expression. For a constraint containing a supported relational operator $\bowtie \in \{\geq, \leq, =\}$, it constructs

$$E_{root} : \text{LHS} - \text{RHS} \bowtie 0. \tag{5}$$

For a nonlinear objective, E_{root} is the objective expression. NED-Tree retains linear terms in l_F. When it encounters a complex nonlinear sub-expression, it introduces an auxiliary variable y_i, replaces that sub-expression in l_F, and places the pair (y_i, E_i) in a pending queue. Thus, l_F retains the main relational structure, and the queued pairs retain the nonlinear structures to be expanded.

Expansion of the New Definition Set. NED-Tree repeatedly processes the pending queue. A supported atomic nonlinear operation is added to D_{new} with its required domain conditions. A composite nonlinear structure is further decomposed by introducing auxiliary variables. Definitions formed by linear operations are retained in D_{new}. Expressions outside the supported rules return UNSUPPORTED. NED-Tree applies three major transformation rules. First, binary chain splitting rewrites a multivariate product $\prod_{i=1}^{n} v_i$ as a sequence of binary product definitions. Second, operator decoupling assigns the inner function of a composite expression such as $\cos(\log(x))$ to an auxiliary variable and defines the outer function over that variable. Third, equivalent structural transformations rewrite supported expressions under their required domain conditions; for a known positive numeric base γ, the parameter-base variable exponent γ^x is transformed into $\exp(x \log \gamma)$. Required domain conditions are recorded and passed to code mapping; their mathematical scope and execution requirements are detailed in Appendix E. Algorithm 1 in Appendix G summarizes the scalar construction and expansion process.

For example, consider $\alpha + \gamma \cos(\log(x_3)) \geq 10$. NED-Tree obtains

$$\begin{aligned} l_F &: \alpha + \gamma y_1 - 10 \geq 0, \\ D_{new} &: \{y_2 = \log(x_3),\ y_1 = \cos(y_2)\}, \\ \mathcal{V}_{new} &: \{y_1, y_2\}. \end{aligned} \tag{6}$$

It also records the required domain condition $x_3 > 0$. This example illustrates the resulting Linear Backbone, New Definition Set, and auxiliary variables. Complete transformation steps and the longer example are provided in Appendix G.

4.4 CODE MAPPING AND REPAIR

After NED-Tree construction, the framework obtains the Linear Backbone l_F, the New Definition Set D_{new}, and introduced auxiliary variables $\mathcal{V}_{new}$. As shown in Figure 3(c), code mapping first declares $\mathcal{V}_{new}$ and maps l_F to an objective function or a standard solver constraint. It then traverses D_{new} in the order of its definitions: definitions formed by linear operations are encoded directly, and atomic nonlinear definitions are mapped to the corresponding solver-supported operations. This process relies on the solver's existing nonlinear capability and provides the agent with a structured mapping from nonlinear expressions to solver code. Finally, CodeSolver executes the generated code, collects syntax and solver API errors, and uses diagnostic feedback to regenerate code from the same structured context. Examples and the complete procedure are provided in Appendix H.

5 EXPERIMENTS

5.1 EXPERIMENT SETUPS

Dataset. We evaluate our method on the 10 OR modeling benchmarks in Table 1: (1) NLP4LP (AhmadiTeshnizi et al., 2024), (2) NL4OPT (Ramamonjison et al., 2023), (3) ComplexOR, (4) MAMO (Easy) and (5) MAMO (Complex) (Huang et al., 2024), (6) IndustryOR (Huang et al., 2025), (7) OptMATH (Lu et al., 2025), (8) OptiBench (Yang et al., 2024), (9) OR-LLM-Agent benchmark (Zhang & Luo, 2025), and (10) NEXTOR. NLP4LP, NL4OPT, and MAMO cover foundational linear programming; ComplexOR and IndustryOR contain industrial grade problems with complex constraints; OptMATH and OptiBench include nonlinear mathematical expressions; and OR-LLM-Agent includes linear optimization problems with tabular inputs. NEXTOR contains 76 challenging linear(38 cases) and nonlinear optimization problems(38 cases), long descriptions, and realistic text redundancy. Appendix K and Table 9 provide detailed benchmark coverage, and Figure 5 reports NEXTOR statistics. Appendix M and Figure 6 describe the NEXTOR synthesis process.

Baselines. We compare our method with four categories of baselines: (1) **Non-Reasoning Models**, including DeepSeek-V3 and GPT-4o (Hurst et al., 2024); (2) **Reasoning Models**, including

Table 1: Pass@1 accuracy (%). **Red bold** and underlined values mark the best and second-best results. † denotes published results; ‡ marks ComplexOR answers from LLMOPT's repository (Jiang et al., 2025). The best-performing OptiBench and OptMATH weights are unavailable. Method AVG covers available datasets. IR (percentage points) compares ours with each dataset's strongest baseline in the indicated group (FT: fine-tuning); its AVG averages these ten differences.

Method	NLP4LP	NL4OPT	ComplexOR‡	MAMO (Easy)	MAMO (Complex)	IndustryOR	OptMATH	OptiBench	OR-LLM-Agent	NEXTOR	AVG
Non-Reasoning Models											
DeepSeek-V3	73.96	87.82	66.67	88.30	40.28	33.00	32.60	64.79	61.44	26.31	57.52
GPT-4o	76.03	81.30	66.67	88.19	33.17	34.00	34.94	58.51	40.96	18.42	53.22
Reasoning Models											
DeepSeek-R1	76.44	90.00	66.67	82.97	47.39	33.00	40.96	74.21	68.67	40.79	62.11
GPT-o4-mini	76.85	88.26	72.22	89.26	43.12	34.00	**45.18**	64.13	72.29	52.63	63.79
Prompt-based Methods											
Chain-of-Experts	†53.1	†64.2	†25.9	85.28	42.18	33.00	39.7	69.26	67.47	51.32	53.15
Optimus	†72.0	†78.8	†66.67	89.11	44.08	32.00	19.28	64.96	48.19	18.42	53.35
OR-LLM-Agent	81.40	88.26	66.67	89.20	47.86	35.00	**45.18**	69.75	74.70	52.63	65.07
Fine-tuning Methods											
ORLM (best)	68.60	†86.50	55.56	†82.30	†37.40	†38.00	9.03	54.55	12.05	11.84	45.58
LLMOPT	†**83.80**	†93.00	†72.70	†**97.00**	†**68.00**	†**46.00**	†40.00	†66.44	13.25	1.32	58.15
OPTBENCH	-	-	-	-	-	-	-	†66.10	-	-	66.10
OptMATH	-	†**95.90**	-	†89.90	†54.10	-	†34.70	-	-	-	68.65
Ours											
Ours	80.16	94.35	**83.33**	90.49	49.52	40.00	42.16	**86.86**	**81.93**	**76.32**	**72.51**
Improvement over strongest comparable baselines											
IR: non-FT (pp)	-1.24	+4.35	+11.11	+1.23	+1.66	+5.00	-3.02	+12.65	+7.23	+23.69	+6.27
IR: FT (pp)	-3.64	-1.55	+10.63	-6.51	-18.48	-6.00	+2.16	+20.42	+68.68	+64.48	+13.02

Table 2: Comparison of AC and PR metrics between linear and nonlinear tasks on NEXTOR.

Method	Nonlinear		Linear	
	AC	PR	AC	PR
DeepSeek-V3	23.68%	31.58%	28.95%	71.05%
GPT-4o	21.05%	26.32%	15.79%	52.63%
DeepSeek-R1	39.47%	55.26%	42.11%	92.11%
GPT-5	60.53%	89.47%	44.74%	89.47%
Chain-of-Experts	52.63%	76.32%	50.00%	92.11%
OR-LLM-Agent	60.53%	81.58%	44.74%	89.47%
LLMOPT	0.00%	13.16%	2.63%	10.53%
ORLM	21.05%	52.63%	2.63%	44.74%
Ours	**92.11%**	**100.00%**	**60.53%**	**100.00%**

DeepSeek-R1 (Guo et al., 2025), GPT-o4-mini, and GPT-5 (Singh et al., 2025); (3) **Prompt-based Methods**, including Chain of Experts (Xiao et al., 2024), OPTIMUS (AhmadiTeshnizi et al., 2024), and OR-LLM-Agent (Zhang & Luo, 2025); and (4) **Fine-tuning Methods**, including ORLM (Huang et al., 2025), OPTBENCH (Yang et al., 2024), OptMATH (Lu et al., 2025), and LLMOPT (Jiang et al., 2025). API-based models and prompt-based methods share the same inference configuration and solver backend unless otherwise specified. Results marked with † are cited from the corresponding papers; the remaining results are evaluated under our experimental pipeline. The extraction pipeline determines each test instance's problem type from its natural-language description. Fairness details are provided in Appendix K.

Metric and Implementation. We use Accuracy (pass@1, AC) and Pass Rate (PR) as complementary metrics. AC evaluates the final answer under each benchmark's checking rule; NEXTOR uses a 1% relative-error tolerance, with exact matching for zero references. PR measures normal program termination, including solver time limits, and does not require an optimal solution. Ours and our reproductions of Chain-of-Experts, OPTIMUS, and OR-LLM-Agent use GPT-5. One pass@1 attempt is a complete pipeline invocation, including its internal stages; CodeSolver uses a repair limit of $K = 3$. We use API inference with a temperature of 0.2 and a maximum of 8192 output tokens per call, Python 3.12, and Gurobi 12.0.2 on a server with an NVIDIA GeForce RTX 5090 GPU (32 GB). We repeat the end-to-end experiments 10 times and report the mean results. Appendix K defines the metrics and pipeline comparison protocol.

We evaluate overall modeling performance (RQ1), component contributions (RQ2), and expression preservation and solver-interface mapping (RQ3).

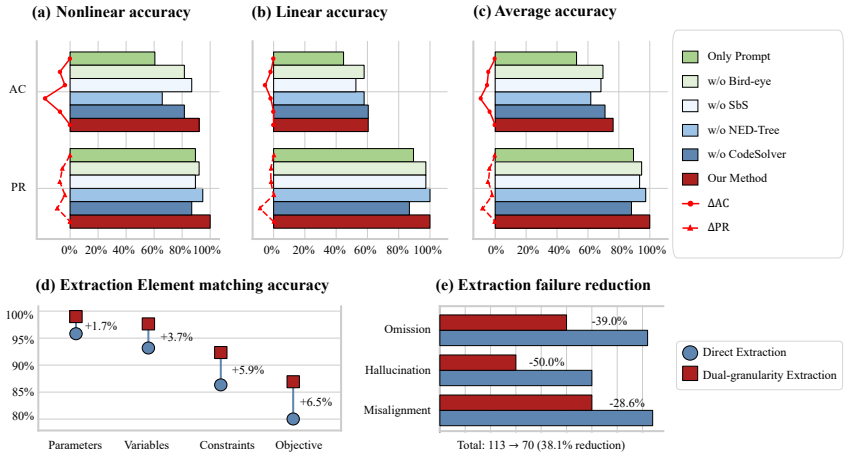


Figure 4: Ablation study on NEXTOR. Panels (a)–(c) report the end-to-end AC and PR of different ablated variants on nonlinear, linear, and average settings, respectively. The red curves indicate the performance gaps of AC and PR between each ablated variant and the full framework. Panels (d)–(e) provide a fine-grained analysis of the dual-granularity extraction module. Panel (d) compares element matching accuracy for parameters, variables, constraints, and objectives, while panel (e) reports the reduction of omission, hallucination, and sentence-element misalignment errors.

5.2 RQ1: How Does Our Method Perform across Multiple Benchmarks and in Nonlinear Modeling?

Table 1 compares complete pipelines across 10 OR benchmarks; Table 2 separates nonlinear and linear NEXTOR tasks. RQ2 examines component contributions.

Overall performance across benchmarks. The NED-Tree framework achieves a mean accuracy of 72.51% across the 10 benchmarks, the highest average among methods with results reported on all ten. Our method achieves the best results on four datasets and the second-best results on four other datasets. Across the reported entries, the NED-Tree framework delivers strong performance on nonlinear, linear, and industrial OR benchmarks.

Performance on NEXTOR and nonlinear modeling. Table 2 reports the complete results on the nonlinear and linear splits of NEXTOR. On the nonlinear split, our framework achieves 92.11% AC and 100.00% PR, exceeding the strongest baseline AC by 31.58 percentage points. On the linear split, our framework achieves 60.53% AC and 100.00% PR, exceeding the strongest baseline by 10.53 percentage points in AC and 7.89 percentage points in PR. Before code generation, NED-Tree decomposes nonlinear terms into auxiliary definitions and atomic operator mappings, directly targeting the Type II nonlinear code semantic errors identified in Section 3. The results establish strong performance on both nonlinear and linear modeling tasks.

5.3 RQ2: What Roles Do the Components Play?

Impact of Dual-Granularity Element Extraction. Removing either Global bird's-eye extraction or Local sentence-by-sentence extraction degrades the reliability of problem-element construction and increases Type I modeling semantic errors identified in Section 3. Removing Global bird's-eye extraction increases parameter–variable confusion and missed nonlinear trigger signals. Removing Local sentence-by-sentence extraction increases omissions in sentence–element correspondence.

Impact of NED-Tree. Removing NED-Tree causes the largest degradation on nonlinear tasks, with a 26.32-percentage-point drop in AC. The observed failures occur when powers, products, fractional terms, and indicator logic are mapped to solver code, corresponding to the Type II nonlinear code semantic errors identified in Section 3. The CodingAgent and the solver's existing nonlinear capability remain fixed in this ablation, and this variant removes the NED-Tree representation path.

Impact of CodeSolver. Removing CodeSolver reduces PR, showing its contribution to execution repair and program executability (Figure 4(a)–(c)).

Table 3: Expression preservation at 30 fixed points (agreement within 10^{-7} across all three paths and the source). Direct AST and NED-Tree detect domain violations before execution.

Expression pattern	Valid matches	Out-of-domain	Max. abs. deviation	Recorded domain
Product	$5/5$	$0/0$	2.22×10^{-16}	None
Positive fractional power	$4/4$	$1/1$	0	$x_1 \geq 0$
Division/negative power	$4/4$	$1/1$	0	$x_2 \neq 0$
Logarithm	$3/3$	$2/2$	1.78×10^{-15}	$x_1 > 0$
Square root	$4/4$	$1/1$	0	$x_1 \geq 0$
Nested expression	$3/3$	$2/2$	0	$x_1 x_2 > 0;\ x_3 \neq 0$
Total	$23/23$	$7/7$	1.78×10^{-15}	Pattern-specific

Table 4: One-shot coding diagnostic on ten fixed-value nonlinear constraints: constraint preservation, successful execution, Type II errors, and Representation Path adherence.

Representation Path	Constraint Preserved	Program Run	Type II Errors	Path Adherence
Native nonlinear expression	$10/10$	$9/10$	$1/10$	$10/10$
Direct AST expression	$10/10$	$9/10$	$1/10$	$10/10$
NED-Tree	$10/10$	$10/10$	$0/10$	$10/10$

5.4 RQ3: Is NED-Tree Reliable and Effective for Nonlinear Solver Mapping?

RQ3 evaluates semantic preservation and solver-interface mapping. Gurobi 12.0.2 provides native nonlinear interfaces; NED-Tree makes outer relations, auxiliary definitions, variable and parameter bindings, dependencies, and domain conditions explicit for the CodingAgent. First, with source expressions fixed and no LLM, we compare Native, Direct AST, and NED-Tree at 30 points from six expression patterns. Second, we vary only the Representation Path across 30 one-shot programs, holding the CodingAgent, ten fixed-value constraints, and the Gurobi construction contract constant. Native uses independently written complete constraints; Direct AST maps each constraint as one nonlinear expression; NED-Tree shares its parser and domain collector and additionally produces l_F and D_{new}. Full protocols appear in Appendices L and J.

Semantic preservation. All 23 in-domain points in Table 3 agree across paths and with the source expression, with maximum absolute deviations of 1.78×10^{-15}. Direct AST and NED-Tree also identify all $7/7$ domain violations before execution. Thus, l_F and D_{new} preserve the tested values and domain conditions. Conditional equivalence argument is in Appendix I.

Solver-interface mapping. All paths preserve the supplied constraints and assigned representations in $10/10$ cases. NED-Tree is the only path with $10/10$ Program Run and $0/10$ Type II errors; both Native and Direct AST achieve $9/10$ Program Run with one Type II error. In c02, Direct AST constructs $x_1^{3/2}$ but submits the unsupported `NLExpr >= constant` relation; NED-Tree instead records $y = x_1^{3/2}$ in D_{new} and applies $y - \beta \geq 0$ through l_F. This localizes the observed Direct AST failure to solver-interface mapping.

Expression preservation, controlled mapping, and the 26.32-percentage-point ablation drop jointly support NED-Tree's role in agent-level nonlinear construction. The open-source decomposition component runs without Gurobi and passes l_F and D_{new} to a solver-specific mapping layer; Appendix N further evaluates transfer from Gurobi to COPT.

6 Conclusion

We present an LLM-based framework that connects traceable element extraction, NED-Tree decomposition, and code generation with repair. NED-Tree exposes the Linear Backbone, auxiliary definitions, and atomic solver mappings as explicit construction steps. Evaluation across 10 benchmarks demonstrates strong nonlinear and linear modeling performance, while controlled diagnostics verify expression preservation and solver-interface mapping. These results support organizing LLM code generation around the relational structure and solver requirements of nonlinear OR models.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Scalable optimization modeling with (mi) lp solvers and large language models, 2024.

Paolo Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, pp. 157–183, Sep 1993.

Charles J Corbett and Gregory A DeCroix. Shared-savings contracts for indirect materials in supply chains: Channel profits and environmental impacts. *Management science*, 47(7):881–893, 2001.

Gerard Cornuejols, Javier Peña, and Reha Tütüncü. *Optimization methods in finance*. Cambridge University Press, 2018.

Gianpaolo Ghiani, Gilbert Laporte, and Roberto Musmanno. *Introduction to Logistics Systems Management: With Microsoft Excel and Python Examples*. John Wiley & Sons, 2022.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, Version 12.0. `https://docs.gurobi.com/_/downloads/optimizer/en/12.0/pdf/`, 2024. Accessed 2026-08-29.

David Ke Hong, Yadi Ma, Sujata Banerjee, and Z. Morley Mao. Incremental deployment of sdn in hybrid enterprise and isp networks. In *Proceedings of the Symposium on SDN Research*, pp. 1–7, Mar 2016.

Yunquan Hu. *Operations Research Tutorial*. Tsinghua University Press, Beijing, China, 5th edition, 2018. ISBN 978-7-302-48125-6. Original in Chinese.

Yunquan Hu. *Operations Research Exercises*. Tsinghua University Press, Beijing, China, 5th edition, 2019. ISBN 978-7-302-52398-7. Original in Chinese.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. Orlm: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 2025.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. Mamo: a mathematical modeling benchmark with solvers, 2024.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card, 2024.

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. Llmopt: Learning to define and solve general optimization problems from scratch. In *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR)*, Singapore, Singapore, 2025.

Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. Large language models for supply chain optimization, 2023.

Haoyang Liu, Jie Wang, Yuyang Cai, Xiongwei Han, Yufei Kuang, and Jianye Hao. Optitree: Hierarchical thoughts generation with tree search for llm optimization modeling. *arXiv preprint arXiv:2510.22192*, 2025.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. Optmath: A scalable bidirectional data synthesis framework for optimization modeling, 2025.

Zhongyuan Lyu, Shuoyu Hu, Lujie Liu, Hongxia Yang, and Ming Li. Canonical intermediate representation for LLM-based optimization problem formulation and code generation. *arXiv preprint arXiv:2602.02029*, 2026.

Zeyuan Ma, Hongshu Guo, Jiacheng Chen, Guojun Peng, Zhiguang Cao, Yining Ma, and Yue-Jiao Gong. Llamoco: Instruction tuning of large language models for optimization code generation, 2024.

Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, et al. A survey of context engineering for large language models, 2025.

Hong Qian and Yang Yu. Derivative-free reinforcement learning: A review. *Frontiers of Computer Science*, 15(6):156336, 2021.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, et al. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In *NeurIPS 2022 Competition Track*, pp. 189–203. PMLR, 2023.

Singiresu S Rao. *Engineering optimization: theory and practice*. John Wiley & Sons, 2019.

Daniela Saban and Gabriel Y Weintraub. Procurement mechanisms for assortments of differentiated products. *Operations Research*, 69(3):795–820, Sep 2021.

Philipp Schmid. The new skill in ai is not prompting, it's context engineering, 2025. URL `https://www.philschmid.de/context-engineering`.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card, 2025.

Yang Song, Anoushka Vyas, Zirui Wei, Sina Khoshfetrat Pakazad, Henrik Ohlsson, and Graham Neubig. NEMO: Execution-aware optimization modeling via autonomous coding agents. *arXiv preprint arXiv:2601.21372*, 2026.

Simon Trimborn, Mingyang Li, and Wolfgang Karl Härdle. Investing with cryptocurrencies—a liquidity constrained investment approach. *Journal of Financial Econometrics*, 18(2):280–306, 2020.

Laurence A. Wolsey. *Integer Programming*. John Wiley & Sons, Hoboken, NJ, USA, 2nd edition, 2020. ISBN 978-1-119-60653-6.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, et al. Chain-of-experts: When LLMs meet complex operations research problems. In *The Twelfth International Conference on Learning Representations*, 2024.

Zhipeng Xie and Shichao Sun. A goal-driven tree-structured neural model for math word problems. In *Ijcai*, pp. 5299–5305, 2019.

Beinuo Yang, Qishen Zhou, Junyi Li, Chenxing Su, and Simon Hu. Automated optimization modeling through expert-guided large language model reasoning. *arXiv preprint arXiv:2508.14410*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2023.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. Optibench meets resocratic: Measure and improve llms for optimization modeling, 2024.

Bowen Zhang and Pengcheng Luo. Or-llm-agent: Automating modeling and solving of operations research optimization problem with reasoning large language model, 2025.

Jipeng Zhang, Lei Wang, Roy Ka-Wei Lee, Yi Bin, Yan Wang, Jie Shao, and Ee-Peng Lim. Graph-to-tree learning for solving math word problems. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pp. 3928–3937, 2020.

Penglin Zhu, Linhai Zhang, Jungang Xu, Xinchi Wei, and Xiuqi Wu. IR2Solve: Structured intermediate representations for cost-efficient optimization autoformulation. *arXiv preprint arXiv:2608.02641*, 2026.

Table 5: Composition of the 300 instances used in the motivated observations.

Category	Number of instances
Linear	100
Non-quadratic Powers	94
Fractional/Rational	50
Logic/Indicator	56
Total	300

A AI USE STATEMENT

This study uses large language models, including GPT and DeepSeek models, for optimization problem element extraction, solver code generation, and execution error repair, and to assist in synthesizing and augmenting NEXTOR problem descriptions. The corresponding procedures are described in Section 4 and Appendix M. During manuscript preparation, we use generative AI to assist with language editing and consistency checks, and to obtain feedback on methodological exposition, experimental design, and result interpretation. The authors are responsible for reviewing AI-assisted content and deciding whether to incorporate it, and take responsibility for the final text, technical claims, experimental conclusions, and accompanying materials.

B ETHICS STATEMENT

This study aims to improve the automated modeling of optimization problems described in natural language. Automatically generated optimization models may contain omitted constraints, semantic discrepancies, or inappropriate modeling assumptions. Before such models are used in actual decision-making, domain experts should therefore verify the consistency among the problem description, mathematical model, and solver results. For applications such as resource allocation that may affect relevant groups, the representativeness of the data, the formulation of constraints, and the fairness of decision outcomes should also be assessed in the specific application context. The accuracy and program pass rates reported in this paper reflect performance on the evaluated benchmarks and do not guarantee outcomes in real-world deployment.

C REPRODUCIBILITY STATEMENT

Section 4 and the corresponding appendices describe the NED-Tree construction procedure, operator coverage, domain conditions, and solver code mapping process. Section 5 and the corresponding appendices provide the model and solver configurations, evaluation metrics, and comparison protocols. Appendix I presents the conditional equivalence argument and its assumptions, Appendix M describes the NEXTOR construction process, and Appendix Q provides prompt examples. The settings for the end-to-end benchmark experiments and controlled coding diagnostics are described separately to help readers understand and reproduce what each experiment evaluates.

D DETAILS OF MOTIVATED OBSERVATIONS

D.1 CONSTRUCTION OF THE NONLINEAR VARIANTS

We sample 100 linear OR problems of simple and general difficulty from IndustryOR (Huang et al., 2025), LogiOR (Yang et al., 2025), and NLP4LP (AhmadiTeshnizi et al., 2024). We preserve the main decision problem and manually modify selected expressions to introduce nonlinear structures. Together with the 100 original linear instances, the resulting 200 nonlinear variants form the 300-instance set used in OE1–OE3. Table 5 summarizes its composition.

Linear. This category contains the 100 original linear problems and serves as the reference setting in OE1.

Table 6: Average accuracy on the original linear instances and their nonlinear variants in OE1.

Benchmark	Linear	Nonlinear variants
IndustryOR	76.67%	42.42%
LogiOR	70.00%	50.00%
NLP4LP	92.00%	54.76%

Non-quadratic Powers. This category contains 94 variants in which selected linear terms in the objective or constraints are replaced by nonlinear power expressions. For example, a term such as $3x$ may be replaced by $3x^3$. The remaining variables, constraints, and modeling objective are preserved unless they must be adjusted to maintain the modified relation.

Fractional/Rational. This category contains 50 variants involving ratios or average-value relations. Examples include replacing $x + y$ with x/y or introducing an average-cost constraint of the form

$$\frac{\sum_i cost_i x_i}{\sum_i x_i} \leq \bar{c}.$$

Such expressions require the denominator and its domain condition to remain explicit. More generally, a relation of the form

$$\frac{F(x)}{G(x)} \leq k$$

is considered together with the corresponding condition $G(x) \neq 0$.

Logic/Indicator. This category contains 56 variants involving piecewise functions or conditional relations. One example is a fixed cost activated when a decision variable satisfies a condition such as $x > 50$. These instances require the logical condition and the associated modeling relation to be preserved when the model is transformed into solver code.

D.2 Evaluation Protocol

We evaluate three representative modeling approaches. The first is direct LLM inference using GPT-5 (Singh et al., 2025). The second is the prompt-based OPTIMUS framework (AhmadiTeshnizi et al., 2024), also using GPT-5 as its underlying model. The third is the fine-tuned optimization model LLMOPT (Jiang et al., 2025).

The experiments follow the same inference configuration used in Section 5.1. All LLM-based inference uses a temperature of 0.2 and a maximum output length of 8192 tokens. The generated programs are evaluated using Python 3.12 and Gurobi 12.0.2. The experiments are conducted on a server equipped with an NVIDIA GeForce RTX 5090 GPU with 32 GB of memory. We conduct 10 repeated experiments using a fixed seed of 42 and report the mean results.

D.3 OE1: Linear and Nonlinear Modeling Accuracy

OE1 compares the accuracy of the three modeling approaches on the original linear instances and the corresponding nonlinear variants. The nonlinear variants preserve the main problem logic while introducing the nonlinear structures described above. Table 6 reports the values visualized in Figure 2(a).

Across the three benchmarks, the nonlinear variants consistently obtain lower modeling accuracy than the corresponding linear instances. OE1 therefore establishes the empirical performance gap examined by the subsequent error analysis.

D.4 OE2: Error Classification

We inspect the failed nonlinear cases and assign each failure to one of three error types according to the stage at which the generated output becomes incorrect.

Table 7: Proportions of the three error types among failed nonlinear cases in OE2.

Approach	Type I	Type II	Type III
Standard GPT-5	49.49%	32.54%	17.97%
OPTIMUS	22.63%	63.16%	14.21%
LLMOPT	28.30%	52.34%	19.36%

Type I: Modeling semantic errors. Type I includes missing or incorrectly extracted variables, parameters, objectives, or constraints. It also includes cases in which an extracted modeling element does not preserve its intended relation to the source description.

Type II: Nonlinear code semantic errors. Type II includes cases in which the nonlinear modeling relation is not reliably implemented through solver-compatible expressions or API calls. The generated code may be syntactically complete while failing to preserve the intended nonlinear relation, variable binding, expression dependency, or domain condition.

Type III: Other coding errors. Type III contains other coding or execution failures, including incomplete Python code blocks, missing imports, or unrelated execution errors.

Related element-extraction, context-handling, and decomposition issues have also been discussed in prior studies (Mei et al., 2025; Schmid, 2025; Liu et al., 2025; AhmadiTeshnizi et al., 2024). OE2 focuses specifically on how these errors appear in the controlled nonlinear variants.

Table 7 reports the error proportions visualized in Figure 2(b).

Type I and Type II jointly account for 82.03%, 85.79%, and 80.64% of the failed cases for Standard GPT-5, OPTIMUS, and LLMOPT, respectively. These results localize most observed failures to problem understanding and the mapping from nonlinear modeling semantics to solver code.

D.5 OE3: OR and Solver Guidance with Manual Formulation Correction and Nonlinear Decomposition

OE3 is a separate manually assisted motivation diagnostic on the IndustryOR nonlinear setting and compares three cumulative conditions. Manual formulation correction and nonlinear decomposition are confined to this diagnostic; the automated NED-Tree framework is evaluated separately in Section 5.

Standard setting. The modeling approach receives the original problem description and follows the standard modeling procedure without additional OR knowledge, solver API guidance, or nonlinear element decomposition.

OR knowledge and solver API guidance. The prompt is supplemented with relevant OR modeling knowledge and solver API documentation intended to help the model select and compose appropriate solver interfaces.

Manual formulation correction and nonlinear decomposition. On top of the preceding setting, human review checks and corrects variables, parameters, objectives, and constraints to obtain a correct mathematical formulation. Complex nonlinear expressions from this human-checked formulation are then manually decomposed into simpler expressions before being provided to the model. The decomposition preserves the corrected nonlinear relations while making their internal structure explicit. This condition combines formulation correction with nonlinear decomposition.

Table 8 reports the values visualized in Figure 2(c).

OR knowledge and solver API guidance produce a substantial improvement over the standard setting. Adding manual formulation correction and nonlinear decomposition provides a further improvement of 10.61 percentage points. The 86.37% accuracy and this incremental gain characterize the combined manual intervention; they do not isolate the effect of decomposition or measure the

Table 8: Accuracy under the three cumulative settings in the OE3 motivation diagnostic. The final condition combines manual formulation correction and nonlinear decomposition.

Setting	Accuracy
Standard setting	42.42%
+ OR knowledge and solver API guidance	75.76%
+ Manual formulation correction and nonlinear decomposition	86.37%

gain of automated NED-Tree. This diagnostic motivates explicit formulation construction and nonlinear decomposition in the automated framework evaluated separately in Section 5.

E Definition of Nonlinear Terms

Given an OR problem $\mathcal{P}$, we specialize the corresponding optimization model as follows,

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad s.t.\ g_i(\mathbf{x};\beta_i) \leq 0, \text{for } i = 1,\ldots,N, \tag{7}$$

where f represents the objective function, $\mathbf{x} \in \mathbb{R}^n$ denotes the decision variables, g_i signifies the i-th constraint function, and β_i stands for the corresponding parameters. Both f and g_i may contain nonlinear terms. In practical commercial scenarios, f and g_i frequently involve highly complex nonlinear nesting or logical conditional branches.

Continuous atomic operators $\mathcal{A}_{cont}$. They encompass non-nested unary functions with unit coefficients and binary products. Examples include elementary functions such as $\exp(\cdot)$, $\ln(\cdot)$, $(\cdot)^k$, and $\alpha^{(\cdot)}$, as well as non-differentiable functions such as $|\cdot|$ and basic bilinear products such as $x_1 \cdot x_2$. The current `NEDTree-4R` implementation covers scalar addition, multiplication, powers, division and square roots through powers, exponential, logarithmic, sine, and cosine expressions. Absolute-value expressions are outside its tested operator set.

Logical atomic operators $\mathcal{A}_{logic}$. They represent conditional branches and piecewise relations. The `NEDTree-4R` implementation used in the end-to-end experiments introduces a binary auxiliary variable to represent the truth of a detected condition, together with logical definitions specifying its equivalence to that condition. It then decomposes nonlinear expressions within the branch into auxiliary definitions and retains the branch constraint under the corresponding indicator. The CodingAgent maps the condition bindings and branch constraints to the solver's indicator-constraint interfaces. This representation makes both the condition and the branch's nonlinear dependencies explicit before code generation. The controlled comparison in Section 5.4 evaluates only the listed scalar expression patterns; it does not evaluate conditional-branch expansion.

A complex nonlinear term refers to a function expression obtained through a finite number of linear and composite operations on the aforementioned nonlinear atomic operators:

$$\mathcal{C}(\mathbf{x}) = \Phi(\mathcal{A}_1,\ldots,\mathcal{A}_k,\mathbf{x}) \leq 0, \tag{8}$$

where Φ represents the outer linear backbone and nested combinatorial logic. Any term resulting from the decomposition of a complex nonlinear term is referred to as a sub-nonlinear term. Modern solvers such as Gurobi provide native interfaces for supported composite nonlinear expressions. NED-Tree does not create this solver capability; it organizes the end-to-end agent's mapping into the atomic operator set, auxiliary definitions, and the original relational structure before code generation.

For the tested scalar path, logarithms require positive arguments, square roots require nonnegative arguments, and divisions and negative powers require nonzero denominators or bases. A parameter-base variable exponent γ^x is transformed into $\exp(x\log\gamma)$ only when the known numeric value γ is positive. These domain conditions are recorded during NED-Tree construction and passed to code mapping.

F Dual-Granularity Element Extraction and Modeling

Motivated by OE2 in Section 3, the framework applies Global bird's-eye extraction followed by Local sentence-by-sentence extraction. For the complete natural-language problem description $p =$

$\{s_1, \ldots, s_I\}$, Global bird's-eye extraction concentrates on the global parameter set P, the decision variable set $E^{(v)}$, and macroscopic information Info_{base}, including the problem type:

$$\left\{P, E^{(v)}, \mathrm{Info}_{base}\right\} = LLM_{\mathrm{Birdeye}}(p). \tag{9}$$

At a macroscopic level, this stage isolates tabular data and numerical declarations from logical constraints to attenuate parameter hallucinations during subsequent generation.

Equipped with the extracted variables and parameters, Local sentence-by-sentence extraction performs a fine-grained scan. For each sentence s_i, the system maps the sentence into a constraint set $E_i^{(c)}$ and an objective function $E_i^{(o)}$:

$$\left\{E_i^{(c)}, E_i^{(o)}\right\} = LLM_{\mathrm{SbS}}(s_i). \tag{10}$$

When a conditional structure is detected within a sentence, the extraction module records the corresponding conditional constraint:

$$c_{\mathrm{cond},i} \sim \mathbb{I}(F(X) \leq 0) \implies G(X) \leq 0. \tag{11}$$

All parameters, variables, constraints, objectives, and problem-type information are deposited into the structured element pool E_{Pool}. The system filters this pool according to the extracted expressions and the problem-type signal in Info_{base}:

$$E_{Pool} = \left\{P, E^{(v)}, \mathrm{Info}_{base}\right\} \cup \bigcup_{i=1}^{I} \left\{E_i^{(c)}, E_i^{(o)}\right\} \xrightarrow{\mathrm{Filter}} \left\{\mathrm{NLSet}, E'_{Pool}\right\}. \tag{12}$$

If nonlinear expressions are detected, they are collected into NLSet and sent to NED-Tree; otherwise, the framework bypasses nonlinear decomposition and directly maps the linear element pool to solver code.

G Nonlinear Element Decomposition Tree Construction and Expansion

The construction of the Linear Backbone l_F and the New Definition Set D_{new} is coupled through a queue-based procedure. For a constraint containing a solver-supported relational operator $\bowtie \in \{\geq, \leq, =\}$, NED-Tree first normalizes the expression into

$$E_{root} : \mathrm{LHS} - \mathrm{RHS} \bowtie 0. \tag{13}$$

Starting from E_{root}, NED-Tree keeps linear terms in l_F. When a complex nonlinear sub-expression is encountered, the algorithm introduces a new auxiliary variable, replaces the sub-expression in l_F, and inserts the corresponding definition into the pending queue. The expansion stage repeatedly processes this queue until every definition is solver-compatible and atomic or an unsupported state is returned.

Every auxiliary variable on the left-hand side of a definition in D_{new} belongs to $\mathcal{V}_{new}$.

The three major transformation rules operate as follows. For a multivariate product $\prod_{i=1}^{n} v_i$, binary chain splitting rewrites the expression into binary product definitions. For example, $x_1x_2x_3$ is decomposed into $y_1 = x_1x_2$ and $y_2 = y_1x_3$. For a composite function such as $\cos(\log(x))$, operator decoupling produces $y_1 = \cos(y_2)$ and $y_2 = \log(x)$. For a parameter-base variable exponent, the equivalent structural transformation $\gamma^x = \exp(x \log \gamma)$ is applied when the required domain condition $\gamma > 0$ holds.

The representative nonlinear expression used in the main construction is

$$\alpha + \beta \cdot 3^{x_1} \cdot \exp(2x_2) + \gamma \cdot \cos(\log(x_3)) \geq 10. \tag{14}$$

NED-Tree first extracts the Linear Backbone

$$l_F : \alpha + \beta y_2 + \gamma y_1 - 10 \geq 0, \tag{15}$$

Algorithm 1 NED-Tree Construction and Expansion for Scalar Expressions

Input: Nonlinear expression $expr$, parameter set $\mathcal{P}$, variable set $\mathcal{V}$
Output: Linear Backbone l_F, New Definition Set D_{new}, auxiliary variables $\mathcal{V}_{new}$, and required domain restrictions, or an unsupported state

Initialize $D_{new} \leftarrow \emptyset$, $\mathcal{V}_{new} \leftarrow \emptyset$, and queue $Q \leftarrow \emptyset$
if $expr$ contains $\bowtie \in \{\geq, \leq, =\}$ **then**
  Construct $E_{root} \leftarrow \text{LHS} - \text{RHS}$ and retain $\bowtie$
else if $expr$ contains no relational operator **then**
  Set $E_{root} \leftarrow expr$
else
  return UNSUPPORTED
end if
Record the required domain restrictions in $expr$
if a required domain condition is violated by a known value **then**
  return UNSUPPORTED
end if
Extract the linear backbone l_F from E_{root}
Replace each nonlinear sub-expression in l_F with a new auxiliary variable
Insert each new variable into $\mathcal{V}_{new}$ and push its pair (y, E) into Q
while Q is not empty **do**
  Pop an auxiliary pair (y, E) from Q
  if E is formed by linear operations **then**
    Replace each nonlinear sub-expression by a new auxiliary variable, insert it into $\mathcal{V}_{new}$, and push its pair into Q
    Add the resulting definition to D_{new}
  else if E is a supported composite function $f(g(x))$ **then**
    Introduce y_{sub} for $g(x)$ and insert it into $\mathcal{V}_{new}$
    Push $(y_{sub}, g(x))$ into Q and add $y = f(y_{sub})$ to D_{new}
  else if E is a multivariate product $\prod_i v_i$ **then**
    Apply binary chain splitting and update D_{new} and Q
    Insert every new auxiliary variable into $\mathcal{V}_{new}$
  else if E matches α^x for a known numeric value α **then**
    if $\alpha > 0$ **then**
      Introduce y_{sub} for $x \log \alpha$ and insert it into $\mathcal{V}_{new}$
      Push $(y_{sub}, x \log \alpha)$ into Q and add $y = \exp(y_{sub})$ to D_{new}
    else
      return UNSUPPORTED
    end if
  else if E is a supported atomic operator with a known domain rule **then**
    Introduce and queue an auxiliary variable for each non-constant argument that is not already a variable, and insert it into $\mathcal{V}_{new}$
    Add the resulting atomic definition to D_{new}
  else
    return UNSUPPORTED
  end if
end while
Arrange D_{new} so that each definition follows the definitions it uses
return l_F, D_{new}, $\mathcal{V}_{new}$, together with the required domain restrictions

and then constructs the New Definition Set

$$\begin{aligned} D_{new} = \{ & y_1 = \cos(y_3), \\ & y_2 = y_4 \cdot y_5, \\ & y_3 = \log(x_3), \\ & y_4 = \exp(y_6), \\ & y_5 = \exp(y_7), \\ & y_6 = x_1 \log 3, \\ & y_7 = 2x_2 \}. \end{aligned} \tag{16}$$

Every introduced variable is inserted into $\mathcal{V}_{new}$, and the condition $x_3 > 0$ required by $\log(x_3)$ is retained. The Linear Backbone alone is not the model: l_F and D_{new} together represent the original nonlinear relation.

H Code Mapping and Repair

After NED-Tree construction, code mapping traverses l_F and D_{new}. The main relational structure in l_F is mapped using conventional constraint construction interfaces such as `model.addConstr`. Each atomic definition in D_{new} is mapped to the corresponding solver-supported operation. In Gurobi, power operations can be mapped to `model.addGenConstrPow`, exponential operations to `model.addGenConstrExp`, and logarithmic operations to `model.addGenConstrLog`. For conditional structures, the CodingAgent maps the explicit logical or auxiliary definitions produced by NED-Tree to solver constraints, including indicator constraints. Recorded domain conditions are passed as explicit requirements to code generation, and unsupported relations or operators are returned as unsupported rather than silently mapped.

After the initial mapping, the generated program is passed to CodeSolver, which performs a bounded generate-execute-repair loop. CodeSolver repairs syntax errors, missing imports, and solver API incompatibilities, but it does not revise the extracted mathematical model or alter the NED-Tree definitions.

Algorithm 2 CodeSolver: Generate, Execute, and Repair

Input: Structured context $\mathcal{Z}$, max iterations K, CodeGenerator, ErrorDiagnostic
Output: Generated code $Code$, execution state $ExecState$, and execution feedback e

$Code \leftarrow \text{CodeGenerator}(\mathcal{Z})$
$(ExecState, e) \leftarrow \texttt{exec}(Code)$
$t \leftarrow 0$
while $ExecState \neq Success$ **and** $t < K$ **do**
  $Advice \leftarrow \text{ErrorDiagnostic}(Code, ExecState, e)$
  $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{Code, Advice\}$
  $Code \leftarrow \text{CodeGenerator}(\mathcal{Z})$
  $(ExecState, e) \leftarrow \texttt{exec}(Code)$
  $t \leftarrow t + 1$
end while
return $(Code, ExecState, e)$

The repair loop terminates when the code runs successfully or the retry budget is exhausted. Its purpose is to improve executability at the programming level while preserving the mathematical structure represented by l_F and D_{new}.

I Conditional Equivalence of the NED-Tree Decomposition

This appendix states the limited mathematical guarantee used by the mechanism discussion. It concerns the decomposition itself, not the global optimality of a numerical solver or the correctness of an upstream natural-language formulation.

Proposition. Consider a scalar expression over registered variables and parameters that uses only the implemented NED-Tree operations described in Section 4.3. Suppose that NED-Tree terminates without returning UNSUPPORTED; every introduced auxiliary variable has exactly one definition in D_{new}; the dependencies among these definitions are acyclic; all recorded domain restrictions hold and are enforced by code mapping; and every atomic solver mapping implements the same real-valued operation as its corresponding definition. Then, for every assignment of the original variables, there is a unique assignment of the auxiliary variables satisfying D_{new}. Under this assignment, evaluating the Linear Backbone together with D_{new} gives the same value and relational truth value as the original expression. Consequently, projecting the decomposed relation onto the original variables yields the same feasible set as the original relation within the stated domain.

Proof sketch. Order the auxiliary definitions topologically. A leaf definition contains only registered variables, parameters, or a supported atomic operation. By the atomic-mapping assumption and the recorded domain restrictions, assigning its left-hand-side variable the value of its right-hand side preserves that sub-expression. Assume that the first k definitions preserve the sub-expressions they replace. The $(k+1)$-st definition depends only on registered symbols or earlier auxiliary variables; substituting their preserved values therefore gives exactly the value of the original sub-expression at this node. Induction over the topological order establishes the claim for every definition in D_{new}. Substituting the resulting auxiliary values into l_F reconstructs the original root expression, so its equality or inequality has the same truth value. Uniqueness follows because each auxiliary variable is defined once by a deterministic operation over already assigned symbols. The same argument applies when the root expression is used as an objective: replacing it by its linked auxiliary value leaves the objective value unchanged.

The proposition is conditional. It does not cover unsupported operators, cyclic or incomplete definitions, missing domain enforcement, floating-point tolerances, or whether a solver returns a global optimum for a nonconvex model. The controlled comparison in Section 5.4 provides executable pointwise evidence for the tested product, power, division, logarithm, square-root, and nested-expression patterns.

J AGENT-LEVEL NATIVE, DIRECT-AST, AND NED-TREE DIAGNOSTIC

The pointwise comparison in Section 5.4 evaluates representation preservation independently of an LLM. We complement it with ten paired fixed-value nonlinear constraint cases. The supplied variable values, parameters, objective, constraint, and Gurobi 12.0.2 construction contract are held fixed, while the same controlled coding-stage agent receives the Native nonlinear expression, the Direct AST expression, or the NED-Tree Linear Backbone and New Definition Set. Each of the 30 programs uses one `gpt-5.1` completion, permits at most one execution, uses no repair, and retains the provider-default temperature. Table 4 reports the aggregate results.

All three Representation Paths preserve the ten supplied constraints and follow their assigned representation. NED-Tree achieves $10/10$ Program Run with $0/10$ Type II errors. Native achieves $9/10$ Program Run with $1/10$ Type II error, and Direct AST achieves $9/10$ Program Run with $1/10$ Type II error. Native's failed program does not correctly translate the nonlinear expression together with its outer constraint relation into solver code; this constraint-mapping failure is classified as Type II. In case c02, Direct AST constructs $x_1^{3/2}$ but directly submits an `NLExpr >= constant` relation. NED-Tree instead maps $y = x_1^{3/2}$ as an atomic definition in the New Definition Set and implements $y - \beta \geq 0$ from the Linear Backbone. Native succeeds in case c02 through a top-level result variable. This case explains the observed Direct AST Type II solver-interface error.

The diagnostic isolates nonlinear-constraint construction rather than optimization search by fixing the supplied variables. It uses ten paired cases, one completion per program, provider-default temperature, and no repair. Prompt lengths differ because the three representations contain different structured content. The experiment provides controlled mechanism evidence, while the full-framework evaluation and NED-Tree ablation provide the end-to-end evidence.

K EXPERIMENTAL SETUP

The end-to-end benchmark comparison is conducted on a single GPU server equipped with an NVIDIA GeForce RTX 5090 GPU with 32 GB of memory. For all LLM-based methods in this comparison, we use their APIs for inference with a uniform temperature of 0.2 and a maximum token count of 8192. We write the code in Python 3.12 and use Gurobi 12.0.2 as the solver. The comparison uses a fixed seed of 42 over 10 repeated experiments and reports the mean results. Variances are omitted from the main comparison table due to paper space limitations.

K.1 SETTINGS AND METRICS

We use two complementary metrics to evaluate different stages of the natural-language-to-solver pipeline. Accuracy measures final-answer agreement under each benchmark's checking rule. Pass Rate measures whether the generated solver program is executable.

Accuracy (pass@1). For each problem instance i, let $\hat{a}_i$ denote the answer returned by the generated solver program and let a_i denote the ground-truth answer. We define

$$\mathbb{I}_{AC}(i) = \mathbb{I}\left[\mathrm{Exec}(i) = 1 \wedge \mathrm{Match}(\hat{a}_i, a_i) = 1\right]. \tag{17}$$

Given a test set $\mathcal{D}$, pass@1 accuracy is

$$\mathrm{AC} = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}_{AC}(i). \tag{18}$$

Pass Rate (PR). For each instance, $\mathbb{I}_{PR}(i)$ is 1 if the generated program executes and returns a valid solver status:

$$\mathbb{I}_{PR}(i) = \mathbb{I}\left[\mathrm{Exec}(i) = 1\right], \qquad \mathrm{PR} = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}_{PR}(i). \tag{19}$$

PR measures execution robustness; AC measures end-to-end answer agreement under the benchmark checking rule.

K.2 BASELINES AND BENCHMARKS

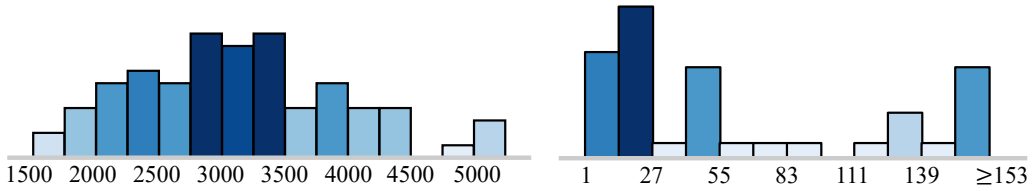
Baselines. We compare with four categories of representative methods: Non-Reasoning Models, including DeepSeek-V3 and GPT-4o (Hurst et al., 2024); Reasoning Models, including DeepSeek-R1 (Guo et al., 2025), GPT-o4-mini, and GPT-5 (Singh et al., 2025), with the evaluated subset identified in each table; Prompt-based Methods, including Chain of Experts (Xiao et al., 2024), OPTIMUS (AhmadiTeshnizi et al., 2024), and OR-LLM-Agent (Zhang & Luo, 2025); and Fine-tuning Methods, including ORLM (Huang et al., 2025), OPTBENCH (Yang et al., 2024), OptMATH (Lu et al., 2025), and LLMOPT (Jiang et al., 2025).

Benchmarks. The evaluation covers 10 OR benchmarks spanning a wide range of difficulty. These problems range from foundational linear programming problems, such as NL4OPT, NLP4LP, and MAMO, to industrial-grade problems with complex constraints, such as IndustryOR and ComplexOR. To assess the capability of handling nonlinear semantics, the evaluation includes tasks involving highly nonlinear mathematical expressions in OPTIBENCH and OptMATH, together with NEXTOR. NEXTOR contains challenging linear programming and nonlinear optimization problems as well as instances with long descriptions and real-text redundancy. It jointly covers quadratic and conic structures, high-order powers and fractional terms, exponential and logarithmic terms, long descriptions, tabular inputs, multimodal descriptions, redundant content, and reference solver code.

Baselines and Fairness Protocol. For API-based models and prompt-based methods, we use the same inference configuration and solver backend unless otherwise specified. Results marked with † are cited from the corresponding papers, and the remaining results are evaluated under our experimental pipeline. Several fine-tuned baselines have incomplete availability of weights, training data, or nonlinear training splits. Reported averages for these methods cover their available datasets. Table 1 reports the GPT-o4-mini reasoning-model row, and Table 2 reports the GPT-5 reasoning-model row. The extraction pipeline supplies the problem-type judgment used by our method.

Table 9: Comparison of NEXTOR with other benchmarks. HP, FP, and ELP denote high-order powers, fractions and fractional powers, and exponentials and logarithms, respectively. ✔ and ✘ indicate coverage and absence, respectively. Multimodality includes multiple languages, character images, and Mermaid drawings; Redundant Content includes noise content and background information.

Dataset	Linear	Non-linear			Table	Multimodality	Reference Code	Redundant Content	AvgLen
		QP & SOCP	HP & FP	ELP					
NLP4LP	✔	✘	✘	✘	✘	✘	✘	✘	536
NL4OPT	✔	✘	✘	✘	✘	✘	✘	✘	534
ComplexOR	✔	✘	✘	✘	✘	✘	✔	✘	1273
MAMO (H)	✔	✘	✘	✘	✘	✘	✘	✘	1724
ORllmAgent	✔	✘	✘	✘	✔	✘	✘	✘	1220
OptiBench	✔	✔	✘	✘	✔	✘	✘	✘	686
IndustryOR	✔	✘	✘	✘	✔	✘	✘	✘	1185
OPTMATH	✔	✔	✘	✘	✔	✘	✘	✘	2974
NEXTOR	✔	✔	✔	✔	✔	✔	✔	✔	**3144**



(a) Question length distribution **(b) Variable Numbers Distribution**

Figure 5: Statistics of the NEXTOR benchmark. Panel (a) shows the question-length distribution, and panel (b) shows the distribution of variable numbers.

L Detailed Settings for RQ3

This diagnostic fixes the upstream expression and evaluates NED-Tree decomposition independently of end-to-end modeling. We compare three Representation Paths under Gurobi 12.0.2. The **Native nonlinear-expression path** uses an independently written Gurobi construction for the complete expression. The **Direct AST path** maps the complete expression to one Gurobi nonlinear expression. The **NED-Tree path** uses the same whitelist parser and domain collector as the Direct AST path and converts the expression into the Linear Backbone and the New Definition Set before solver mapping.

We preselect six expression patterns and evaluate five fixed points per pattern. The resulting 30 points contain 23 points inside the recorded domains and seven points outside them. At each in-domain point, the original variables are fixed, and the source expression is compared with all three Gurobi paths. A valid match requires all three paths to reach an optimal status and agree with the source expression within 10^{-7}. For each out-of-domain point, the Direct AST and NED-Tree paths apply the same explicit domain check before solver execution. Table 3 reports the pointwise results; Appendix I gives the conditional equivalence argument for the supported rules.

M NEXTOR Benchmark Synthesis Method

NEXTOR is constructed to evaluate models on complex OR problems and to assess the effectiveness of NED-Tree. The benchmark synthesis method comprises the Forward Step, the NEXT Step, and the Verification Step, based on four Guiding Principles.

M.1 NEXTOR Guiding Principles

NEXTOR follows four principles. **Coverage Extension** spans diverse nonlinear and complex problem types. **Complexity Elevation** increases the number of variables and constraints to reflect real-world decision-making. **Realism Augmentation** adds redundant text, unstructured data, and multimodal information to otherwise clean mathematical descriptions. **Verifiability** pairs instances with reference answers and solver code for offline auditing.

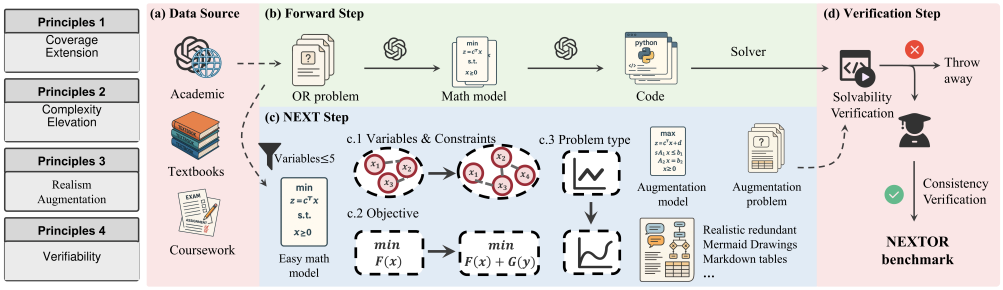


Figure 6: NEXTOR Synthesis Method. The method collects data from three primary channels and then applies the Forward Step, NEXT Step, and Verification Step according to four Guiding Principles.

M.2 NEXTOR Synthesis Method

Data Source. Raw data are obtained from three primary channels: source materials searched using OpenAI's Deep Research, established textbooks (Hu, 2018; 2019; Wolsey, 2020), and university course assignments and exams. All real names are removed and replaced with pseudonyms to avoid personally identifying information, and the data are screened to exclude offensive content. Only publicly available material is collected, and no private or personally identifying information is retained.

Forward Step. The main objective of the Forward Step is to solve and standardize the problem seeds. It formally defines the decision variables, objective function, and constraints; writes standard Python solver code, primarily using Gurobi; and executes the code to obtain a deterministic optimal solution as the ground truth. The output is an internal seed dataset in which each instance contains a natural-language description, a structured mathematical model, standard solver code, and the ground-truth answer.

NEXT Step. The NEXT Step augments problem seeds to evaluate LLM-based modeling ability. Problems with at most five variables are selected for augmentation, motivated by the observation that more than 80% of the problems in OPTIBench contain fewer than five variables (Yang et al., 2024). This stage proceeds along two dimensions. *Complexity Enhancement* increases the intrinsic mathematical difficulty by expanding the number of decision variables and constraints, introducing reasonable modifications or perturbations to the objective function, and promoting basic problem types to more advanced forms, such as from linear programming (LP) to fractional programming. *Realism Augmentation* simulates real-world information environments by injecting redundant information, including irrelevant background stories and distracting data points, and incorporating multimodal or structured information, such as Markdown tables and Mermaid charts. These augmentations increase the requirements for information extraction, filtering, and robustness to noise.

Verification Step. The verification stage combines solver-based solvability checks with expert reviews of consistency between each problem description and its reference answer. Reference code and answer targets are checked offline during dataset construction and auditing. Appendix P documents a known issue identified through this process.

N Solver Transferability Analysis

Figure 7 shows that the performance differences remain small on NL4OPT and NEXTOR when the backend solver is switched from Gurobi to COPT. The NED-Tree representation helps decouple nonlinear semantic reasoning from backend-specific code generation. After nonlinear expressions are decomposed into auxiliary definitions and atomic operator mappings, the solver-specific code-generation layer can be adapted from Gurobi to COPT with minor performance changes on the tested benchmarks.

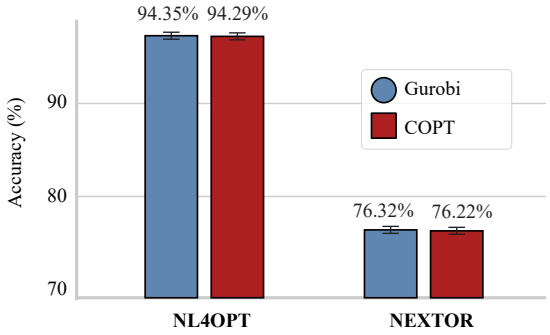


Figure 7: Performance under different solver backends on NL4OPT and NEXTOR. Error bars show the reported standard deviations.

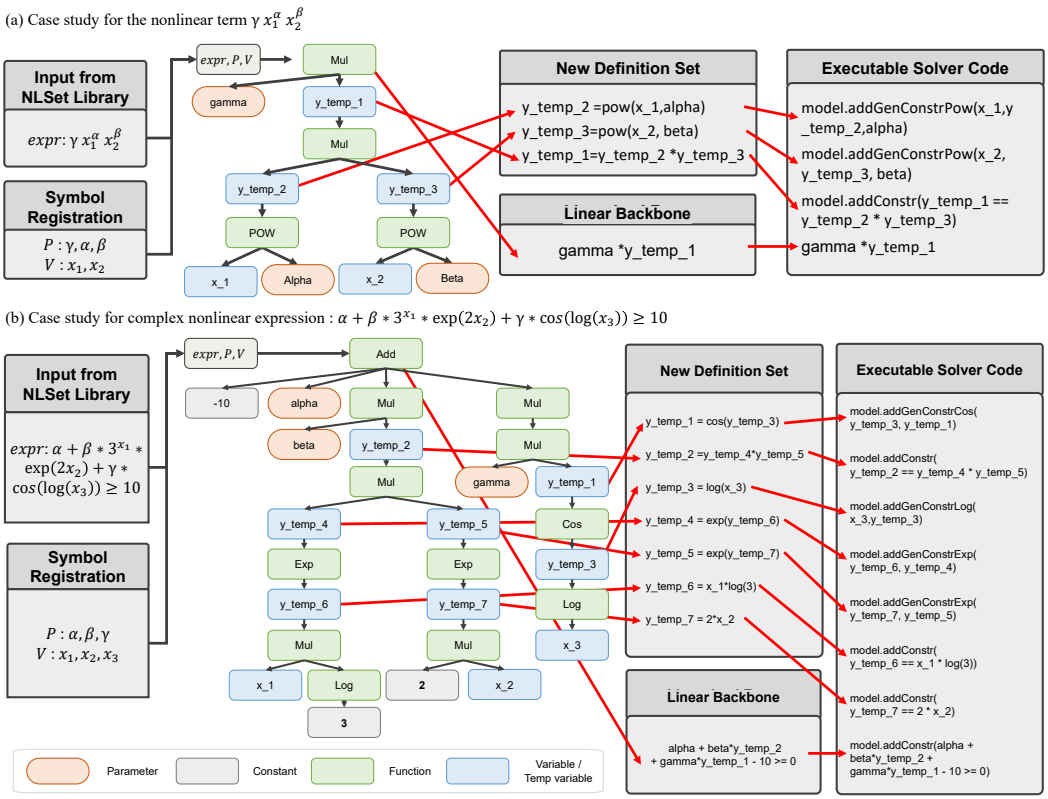


Figure 8: Case studies of NED-Tree decomposition. Panel (a) shows a power-product nonlinear term, and panel (b) shows an expression containing exponential, logarithmic, trigonometric, and product structures.

O Case Study

For the nonlinear term $\gamma x_1^{\alpha} x_2^{\beta}$, the input declares x_1 and x_2 as variables and γ, α, and β as parameters. NED-Tree first performs symbol registration and then recursively constructs the decomposition tree. The power terms x_1^{α} and x_2^{β} are identified as atomic nonlinear units and mapped to auxiliary definitions such as `y_temp_2 = pow(x_1, alpha)` and `y_temp_3 = pow(x_2, beta)`. Their product is represented by `y_temp_1 = y_temp_2 * y_temp_3`, and the Linear Backbone contains $\gamma y_{\text{temp},1}$. This allows the code agent to call `model.addGenConstrPow` for the leaves and standard product constraints for the internal node.

For $\alpha + \beta 3^{x_1} \exp(2x_2) + \gamma \cos(\log(x_3)) \geq 10$, NED-Tree separates logarithm, cosine, exponential, and product operations into ordered auxiliary definitions. The parameter-base power 3^{x_1} is rewritten through the equivalent exponential form with the intermediate term $x_1 \log 3$, and $\exp(2x_2)$ and $\cos(\log(x_3))$ are separated into ordered auxiliary variables. The Linear Backbone is $\alpha + \beta y_{\text{temp},2} + \gamma y_{\text{temp},1} - 10 \geq 0$ together with the New Definition Set. This transformation uses the known positive base 3, and the mapped model retains $x_3 > 0$ for $\log(x_3)$.

P Failure and sensitivity

In the original bridge-beam design case of the NEXTOR nonlinear split, the statement requests the value of x_1, and the ground-truth answer is $10{,}392{,}304.84$, which corresponds to an objective-scale quantity; the optimized x_1 is approximately 173.205. The case introduces the bending-stress relation $\sigma_{\max} = \frac{3}{4}\frac{M}{x_1 x_2^2}$ and omits the material yield limit σ_y. One generated program assumed $\sigma_y = 1$ and returned a scaled bending-moment value of $13{,}856{,}406.46$, which failed the answer check. Another run used a different normalization and matched the ground truth with $10{,}392{,}304.85$. This case remains in the reported 38-problem split under its original evaluation version; its statement has since been corrected without recomputing the reported results. Checks of answer targets, physical constants, and reference quantities are offline dataset audits, and test-time agents do not receive reference answers. A numerical match on this legacy instance therefore does not establish a semantically correct formulation.

Q Prompts for LLMs

Runtime values are replaced by angle-bracket placeholders. The local extraction and CodingAgent boxes are abridged and labeled accordingly.

Q.1 Global Bird's-eye Extraction

Global Bird's-eye Extraction Prompt

```
[System]
You are an expert mathematical modeler and an optimization professor.
Read the following Operations Research (OR) problem description in its entirety.

Extract all elements directly and output a structured JSON containing:
1. "Parameters_List": List of given constants/parameters (Name, Meaning, Type, Value).
2. "Variables_List": List of decision variables (Symbol, Meaning, Type, Range).
3. "Constraint_Table": List of constraints (Constraint Name, Mathematical expressions).
4. "Objective": The optimization goal (Objective_sentence, Mathematical_expressions).
5. "Problem_Type": 'MILP' or 'NLP'.

Do not extract sentence by sentence. Extract the global information directly.
Output strictly in JSON format without any markdown wrappers or additional text.

[User]
Here is the problem description:
----------------------------------------
<PROBLEM_DESCRIPTION>
Output the result mentioned above.
```

Q.2 Local Sentence-by-Sentence Extraction

Local Sentence-by-Sentence Extraction Prompt (Abridged)

```
[Factor extraction -- System]
Scan the problem sentence by sentence. For each numbered sentence, extract
variables and constraints or mark it as "No constraints". Return
Sentence_Scanning, Variables_List, Constraint_Table, Objective, and Problem_Type.

[Parameter extraction -- System]
```

```
Scan every sentence and table for parameters with explicit values, or mark it
as "No Values". Return a Parameters_List with names, types, and exact values.

[User]
Here is the problem description:
----------------------------------------
<PROBLEM_DESCRIPTION>
Output the requested results.
```

Q.3 CODINGAGENT

CodingAgent Prompt for NLP (Abridged)

```
[System]
You are a Python program expert in operations research and optimization, with
proficiency in Gurobi Python coding. Given an NLP optimization problem, its
identified variables, background, constraints, objective, and validated
mathematical model, generate complete, executable Gurobi code that:
1. imports the required packages and defines all parameter data;
2. creates the decision variables and required auxiliary variables;
3. sets the objective and adds all constraints, including general and
   indicator constraints;
4. preserves required domain conditions for nonlinear expressions;
5. solves the model and prints FinalAnswer in the required format.
Output only a fenced Python code block.

[User]
Now, use this information to solve the mathematical problem.
Problem: <PROBLEM>
Context: <VARIABLES_CONSTRAINTS_OBJECTIVE_AND_PROBLEM_TYPE>
You must strictly use the values in the Parameters List.
Parameters List: <PARAMETERS_LIST>

[Assistant]
Math model and coding advice:
<LINEAR_BACKBONE_AND_NEW_DEFINITION_SET>

[User]
Based on the above, write complete and reliable Python code using Gurobi to
solve this operations research optimization problem. The first line must be
"import gurobipy as gp", and the Gurobi model variable must be named "model".
```

Q.4 ONLY PROMPT

Only Prompt

```
[System]
You are an operations optimization expert.
Please construct a mathematical model based on the operational optimization
problem provided by the user, and write complete and reliable Python code to
solve the operational optimization problem using Gurobi.
Please include necessary model construction, variable definition, constraint
addition, objective function setting, solution, and result output in the code.
Output in the form of ```python
{code}
```, without the need for code description.

[User]
Problem: <PROBLEM>
```
```