OPT-JUDGER: FINE-GRAINED EVALUATION OF LLM-BASED OPTIMIZATION MODELING VIA FEASIBILITY VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are increasingly used to generate optimization models from natural language descriptions, motivating the development of methods to evaluate their correctness. However, existing evaluators still struggle to detect constraint errors in generated models and provide limited feedback for diagnosing them. These limitations hinder accurate assessment of LLMs' ability to translate problem requirements into optimization models and offer little guidance for improvement. To address these limitations, we propose Opt-Judger, a solver-backed framework for fine-grained evaluation of LLM-based optimization modeling. Opt-Judger evaluates whether the constraints in generated models faithfully capture the problem requirements by checking whether they permit the same decisions as a reference model representing these requirements. It uses feasibility verification to search for a witness point, which represents a decision that is feasible under one model but infeasible under the other. This point reveals constraint violations that indicate mismatches with the problem requirements and supports error localization. We also construct Opt-Judger-Bench using equivalent reformulations and transformations that simulate common modeling errors. Extensive experiments show that Opt-Judger accurately identifies genuine constraint errors across benchmarks and consistently outperforms existing evaluators on challenging cases.

1 INTRODUCTION

Optimization modeling is a key step in applying operations research to real-world decision problems across diverse domains, from route planning (Vander Wiel & Sahinidis, 1995) and power system scheduling (Liu et al., 2023) to collateral management (Goblet et al., 2024). A formulation expresses problem requirements through decision variables, constraints, and an objective, specifying which decisions are allowed and what should be optimized (Williams, 2013). Once the formulation is correct, modern solvers can often search for high-quality or optimal solutions. The difficult part is writing a correct formulation, because it requires domain understanding and mathematical skill. Even a small constraint error can alter the feasible region, admitting invalid decisions or excluding valid ones. Recent work uses large language models (LLMs) to reduce this modeling burden by generating and refining formulations and solver code from natural language descriptions (AhmadiTeshnizi et al., 2024; Huang et al., 2025; Astorga et al., 2025; Liu et al., 2025; Zhou et al., 2025).

As LLM-based optimization modeling systems become more capable, evaluating them is essential for assessing their ability to translate problem requirements into correct optimization models. Benchmarks such as NL4Opt, MAMO, IndustryOR, and OptiBench have been developed to support this assessment (Ramamonjison et al., 2023; Huang et al., 2024; 2025; Yang et al., 2025). These resources make it possible to compare models across a broader range of optimization modeling tasks. Existing methods for evaluating generated models include solution-based evaluation (AhmadiTeshnizi et al., 2024; Huang et al., 2024; Zhai et al., 2025) and structure-based evaluation (Xing et al., 2024; Wang et al., 2024). Solution-based methods compare optimal objective values (AhmadiTeshnizi et al., 2024; Huang et al., 2024) or check whether optimal solutions remain feasible and optimal in the reference model (Zhai et al., 2025). However, agreement on optimal results can mask constraint errors that change the feasible region without affecting the current optimum. Structure-based methods compare formulation components or graph representations of variables and constraints (Xing et al., 2024;

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.

Wang et al., 2024). However, equivalent formulations may differ structurally through slack-variable introduction, variable decomposition, or redundant constraints and may therefore be incorrectly judged as modeling errors. Moreover, most evaluation methods provide limited feedback beyond correctness scores or equivalence labels, making it difficult to localize errors to specific constraints.

Thus, we argue that evaluation should assess whether the constraints in generated models faithfully capture the problem requirements while distinguishing genuine modeling errors from harmless reformulations. Our insight is that constraints determine which decisions a model permits. Harmless reformulations preserve these decisions despite differences in mathematical representation, whereas constraint errors may admit invalid decisions or exclude valid ones. Comparing the decision sets permitted by a generated model and a reference model representing the problem requirements therefore provides a basis for making this distinction. This comparison presents two challenges: (1) Different formulations may represent the same decisions using different variables and auxiliary constructions, preventing direct comparison of their feasible regions. (2) Feasible decision sets are implicitly defined by constraints rather than explicitly listed. Establishing their equivalence requires verifying that every decision satisfying one constraint system also satisfies the other, and vice versa.

To address these challenges, we propose Opt-Judger, a solver-backed framework for fine-grained evaluation of LLM-based optimization modeling. Opt-Judger first aligns the generated and reference models in a shared space of problem decisions, accounting for differences in variable representations and projecting out supported auxiliary variables. It then checks feasible-region equivalence through bidirectional feasibility verification, searching for a witness point that represents a decision permitted by one model but excluded by the other. This point provides concrete evidence of a mismatch and identifies the violated constraints, supporting error localization at the constraint level. An LLM-assisted attribution step links this evidence to the likely affected problem requirements. The verification applies to successfully parsed and aligned models within the supported formulation classes and numerical tolerances.

We also construct Opt-Judger-Bench using equivalent reformulations and controlled transformations that simulate common modeling errors. Experimental results demonstrate that Opt-Judger achieves 100.0% judgment accuracy on both EquivaFormulation and Opt-Judger-Bench. Opt-Judger outperforms existing baselines, particularly in detecting constraint errors that leave optimal results unchanged. Further evaluation on NLP4LP reveals such errors in models generated by a range of LLMs and modeling methods, highlighting the importance of assessing formulation correctness beyond objective-value agreement.

2 Related work

2.1 Optimization modeling

Optimization modeling turns a real-world decision problem into variables, constraints, and an objective. Traditional optimization modeling relies on human experts to understand the task, identify the decision space, and write a correct formulation. Recent work studies how large language models can support this process. Early benchmark work such as NL4Opt focuses on extracting optimization formulations from natural language descriptions (Ramamonjison et al., 2023). Agent-based systems use LLMs to generate formulations and solver code, revise generated models with solver feedback, or coordinate multiple roles for formulation, planning, coding, and debugging (AhmadiTeshnizi et al., 2024; Astorga et al., 2025; Thind et al., 2025). Search-based and experience-based methods explore intermediate reasoning steps and reuse lessons from previous attempts (Liu et al., 2025; Kong et al., 2025; Zhou et al., 2025). Another line of work trains or adapts language models for operations research tasks through instruction tuning, synthetic data, preference alignment, or solver-informed reinforcement learning on linear, mixed-integer, and industrial optimization problems (Ma et al., 2024; Huang et al., 2025; Jiang et al., 2025; Lu et al., 2025; Chen et al., 2025).

2.2 Evaluation for optimization modeling

Evaluation resources for optimization modeling have grown quickly. NL4Opt introduces a benchmark for natural language optimization formulation (Ramamonjison et al., 2023). OptiMUS builds NLP4LP and evaluates LLMs on scalable optimization modeling tasks (AhmadiTeshnizi et al., 2024). MAMO introduces a solver-backed benchmark for mathematical modeling (Huang et al., 2024). ORLM

includes IndustryOR for industrial optimization problems (Huang et al., 2025). OptMATH provides large-scale synthetic data and a benchmark for optimization modeling (Lu et al., 2025). OptiBench studies broader optimization modeling settings and evaluation protocols (Yang et al., 2025).

Existing methods for evaluating generated models include solution-based evaluation and structure-based evaluation. Solution-based approaches assess generated models using their optimization results. OptiMUS, MAMO, and OptiBench use execution-based evaluation to compare the resulting objective values with reference answers (AhmadiTeshnizi et al., 2024; Huang et al., 2024; Yang et al., 2025). EquivaMap extends solution-based verification by inferring variable mappings and checking the feasibility and optimality of mapped solutions (Zhai et al., 2025). Structure-based approaches compare formulation components or graph representations of variables and constraints. Representative methods include graph edit distance for linear programming formulations (Xing et al., 2024) and graph-based equivalence detection (Wang et al., 2024). In this work, we compare feasible decision sets in an aligned business-variable space and use solver-generated witness points to support constraint-level error diagnosis, while making the required alignment and solver assumptions explicit.

3 Motivating Observations

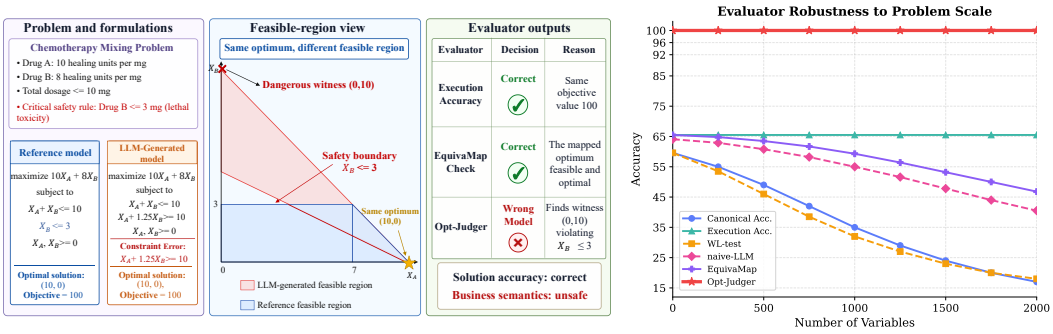

Figure 1: The observations.Left: Motivating examples illustrating the semantic blind spot of execution-based evaluation.Right: The robustness gap of existing evaluators under increasing problem scale.

Motivation 1: Matching optimal results does not imply constraint correctness Evaluation based on matching optimal solutions or objective values can overlook constraint errors in generated models. As illustrated in the left panel of Figure 1, the generated formulation yields the same optimal solution as the reference model, even though its constraints bear little relation to the requirements specified in the problem description. The two formulations permit largely different sets of feasible decisions.

Motivation 2: Harmless reformulations make direct structure comparison unreliable LLMs often generate structurally diverse but mathematically equivalent formulations. Common cases include introducing slack variables, decomposing variables, and adding redundant constraints. These reformulations change the textual or graph structure of the model while preserving the set of feasible decisions over the original decision variables. Consequently, direct structure comparison can mistake harmless reformulations for genuine modeling errors.

Motivation 3: Existing evaluators lack robustness to problem scale We fix the constraint error type and increase the number of variables to examine evaluation robustness. As shown in the right panel of Figure 1, several existing evaluators exhibit declining accuracy, while Opt-Judger maintains stable accuracy across the tested sizes. Moreover, Opt-Judger requires less solver time than solving the original models on larger instances, further supporting its scalability to larger formulations. Appendix E provides the experimental details and analysis.

4 Opt-Judger Evaluation Framework

Figure 2 illustrates the overall framework of Opt-Judger. Opt-Judger evaluates whether generated constraints faithfully capture the problem requirements by comparing the decisions allowed by the generated and reference models. It first aligns the formulations in a shared business-variable space

and projects out auxiliary variables (Section 4.2). A lightweight objective-matching check screens out basic objective errors before constraint verification. It then performs bidirectional feasibility verification to search for a witness point that is feasible under one model but infeasible under the other (Section 4.3). When found, this point reveals constraint violations and supports constraint-level error localization and LLM-assisted attribution to the relevant problem requirements (Section 4.5).

4.1 Evaluation objective and notation

Opt-Judger takes a natural-language problem description, a reference model representing the problem requirements, and a generated model as input. Both models are parsed into explicit solver constraints and aligned in a shared business-variable space to accommodate harmless reformulations. Before comparing constraints, Opt-Judger checks objective consistency by comparing coefficient vectors over aligned variables after normalizing both objectives to a common optimization direction. If this check fails, it reports an objective mismatch. Otherwise, it assesses whether the two models permit the same decisions by verifying feasible-region equivalence in the shared space.

Definition 1 (Business variables)**.** *Business variables directly describe decisions in the original problem statement. Auxiliary variables, such as slack, split, or linearization variables, are introduced only to express the same constraints. We denote the aligned business-variable domain by $\mathcal{X}$.*

We write an optimization model M as

$$\min_{u \in \mathcal{U}} f(u) \quad \text{subject to} \quad g_i(u) \le 0,\ i = 1, \dots, m, \qquad h_j(u) = 0,\ j = 1, \dots, r. \tag{1}$$

Here u contains all variables used in the formulation, including both business variables and auxiliary variables. The feasible region of the model is the set of assignments that satisfy all constraints:

$$\mathcal{P}(M) = \{u \in \mathcal{U} \mid g_i(u) \le 0,\ i = 1, \dots, m,\ h_j(u) = 0,\ j = 1, \dots, r\}. \tag{2}$$

The objective determines which feasible decisions are optimal, while the constraints determine which decisions are allowed. Opt-Judger compares these feasible decision sets to assess whether generated constraints faithfully capture the problem requirements and identify constraint errors.

Let M_{ref} denote the reference model and M_{gen} denote the generated model. Let $\mathcal{X}$ denote the business variable space. After alignment and projection to this space, define the projected feasible regions as

$$\mathcal{F}_{\text{ref}} = \text{proj}_x(\mathcal{P}(M_{\text{ref}})), \qquad \mathcal{F}_{\text{gen}} = \text{proj}_x(\mathcal{P}(M_{\text{gen}})). \tag{3}$$

From this point on, $\mathcal{F}_{\text{ref}}$ and $\mathcal{F}_{\text{gen}}$ always denote feasible regions in the business variable space. Opt-Judger treats the generated model as constraint-equivalent to the reference model when it preserves the same business decisions:

$$\mathcal{F}_{\text{ref}} = \mathcal{F}_{\text{gen}}. \tag{4}$$

> **Evaluation target.** The core problem is to compare the two projected feasible regions in Eq. equation 3. The generated model is constraint-equivalent only when Eq. equation 4 holds.

This definition targets the constraint semantics of the model. If the two feasible regions are the same in $\mathcal{X}$, then the two models allow the same business decisions, even if they use different variable names, auxiliary variables, or algebraic forms. If the two feasible regions are different, then at least one business decision is treated differently by the two models. In that case, the generated model changes the original constraint semantics.

4.2 Aligning models to business variables

The problem setup above assumes that both models can be compared in the same business-variable space. In practice, valid reformulations may use different variable names, auxiliary variables, or algebraic forms, causing their feasible regions to be defined over different variable spaces. Opt-Judger therefore constructs the shared business-variable space through variable alignment and projection.

For variable alignment, Opt-Judger uses a set-based symbolic representation. Rather than expanding all instance-level values, it represents each variable group by its modeling role. For each group,

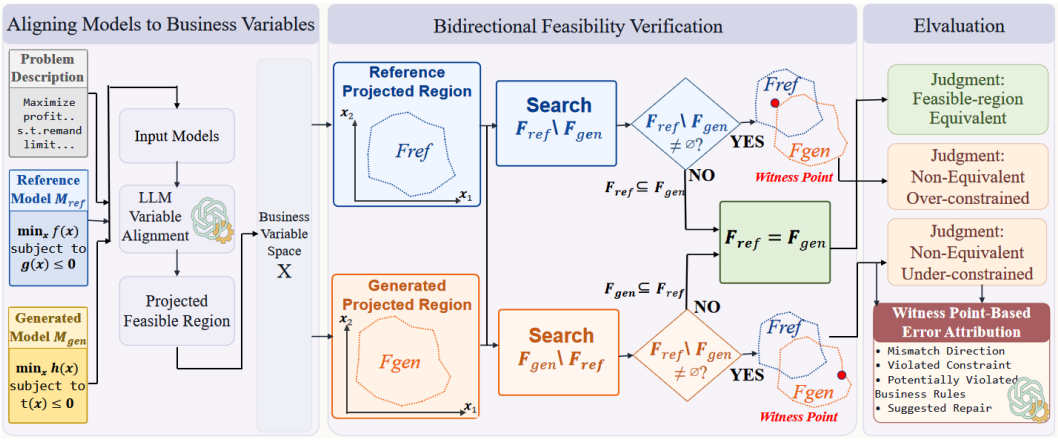

Figure 2: The overview of Opt-Judger. (1) Opt-Judger aligns the reference and generated formulations into a shared business-variable space. (2) Opt-Judger performs bidirectional feasibility verification by searching for witness points in both mismatch directions. (3) Opt-Judger outputs the feasible-region equivalence judgment and uses witness points to provide error attribution and repair guidance.

Opt-Judger extracts its textual description, the constraints in which it appears, the explanations of those constraints, and whether it appears in the objective. These structured features are provided to an LLM, which proposes mappings between variables in the reference model and variables in the generated model. The LLM is used only for semantic alignment; the final equivalence decision is made by solver-backed verification.

After alignment, Opt-Judger projects each model onto the aligned business variables. Suppose a model has business variables x, continuous auxiliary variables z, and feasible region

$$\mathcal{P} = \{(x, z) \mid Ax + Bz \le b,\ Ex + Hz = e,\ x \in \mathcal{X},\ z \in \mathcal{Z}\}. \tag{5}$$

The feasible region used for comparison is its projection onto x:

$$\mathrm{proj}_x(\mathcal{P}) = \{x \in \mathcal{X} \mid \exists z \in \mathcal{Z},\ Ax + Bz \le b,\ Ex + Hz = e\}. \tag{6}$$

Opt-Judger eliminates explicit auxiliary definitions by substitution or Gaussian elimination over equalities, applies Fourier–Motzkin elimination to implicitly constrained continuous auxiliary variables, and handles integer auxiliary variables through integer-preserving encodings when available. If an integer auxiliary is globally unbounded or cannot be locally eliminated, Opt-Judger reports a projection failure. This step only affects variables outside the business-variable space; the equivalence judgment is still defined over the aligned business variables. The full projection procedure is given in Appendix C. Its output is a feasible region over business variables only, as in Eq. equation 6.

4.3 Searching for Feasible-Region Mismatch

After alignment and projection, $\mathcal{F}_{\text{ref}}$ and $\mathcal{F}_{\text{gen}}$ denote the projected feasible regions over the same business variable space $\mathcal{X}$, as defined in Eq. equation 3. Opt-Judger checks feasible-region equivalence by constructing feasibility problems over these projected feasible regions. We first define the regions where the two models disagree.

Definition 2 (Mismatch regions)**.** *Given two feasible regions $\mathcal{F}_{\text{ref}}, \mathcal{F}_{\text{gen}} \subseteq \mathcal{X}$ in the same business variable space, define*

$$\Omega_{\text{ref-only}} = \mathcal{F}_{\text{ref}} \cap (\mathcal{X} \setminus \mathcal{F}_{\text{gen}}), \qquad \Omega_{\text{gen-only}} = \mathcal{F}_{\text{gen}} \cap (\mathcal{X} \setminus \mathcal{F}_{\text{ref}}). \tag{7}$$

The first mismatch region contains business decisions that are feasible under the reference model but infeasible under the generated model. The second mismatch region contains business decisions that are feasible under the generated model but infeasible under the reference model.

Proposition 1 (Feasible-region equivalence by empty mismatch regions)**.** *The generated model and the reference model have the same feasible region in the business variable space if and only if both*

mismatch regions are empty:

$$\mathcal{F}_{\text{ref}} = \mathcal{F}_{\text{gen}} \quad \textit{holds exactly when} \quad \begin{cases} \Omega_{\text{ref-only}} = \emptyset, \\ \Omega_{\text{gen-only}} = \emptyset. \end{cases} \tag{8}$$

Eq. equation 8 turns feasible-region equivalence into two feasibility questions. Opt-Judger does not need to directly compare two feasible regions. It only needs to check whether $\Omega_{\text{ref-only}}$ or $\Omega_{\text{gen-only}}$ contains a feasible point. We call such a feasible point a witness point, because it gives a concrete business decision on which the two models disagree. A reference-only witness indicates an over-constrained error mode, while a generated-only witness indicates an under-constrained error mode. Both modes can occur for the same generated model. If both mismatch regions are empty, then the two feasible regions are equivalent in the business variable space.

4.4 Turning mismatch search into solver problems

The mismatch regions in Eq. equation 7 contain set complements, such as $\mathcal{X} \setminus \mathcal{F}_{\text{gen}}$. A solver cannot directly take this set expression as input. Opt-Judger therefore turns each mismatch search into a standard feasibility problem. It keeps the constraints of one model and asks the solver to find a point that violates at least one constraint of the other model. Assume that the projected feasible regions in Eq. equation 3 are represented as

$$\mathcal{F}_{\text{ref}} = \{x \in \mathcal{X} \mid A_{\text{ref}} x \le b_{\text{ref}}\}, \qquad \mathcal{F}_{\text{gen}} = \{x \in \mathcal{X} \mid A_{\text{gen}} x \le b_{\text{gen}}\}. \tag{9}$$

The first feasibility problem searches for a point in $\Omega_{\text{ref-only}}$, namely a business decision that is feasible under the reference model but infeasible under the generated model. The second feasibility problem searches for a point in $\Omega_{\text{gen-only}}$, namely a business decision that is feasible under the generated model but infeasible under the reference model.

The key step is to express "infeasible under the other model" in a form that a solver can handle. A point is outside a feasible region when it violates at least one constraint. With numerical solvers, Opt-Judger searches for an ϵ-separated mismatch, where a violated inequality must be violated by at least $\epsilon > 0$. For a constraint $c_i^\top x \le d_i$, it introduces a binary indicator y_i:

$$c_i^\top x \ge d_i + \epsilon - M_i(1 - y_i), \tag{10}$$

When $y_i = 1$, the point violates the constraint by at least ϵ; when $y_i = 0$, the big-M term relaxes the requirement. A constraint-wise M_i is valid if

$$M_i \ge d_i + \epsilon - \min_{x \in \mathcal{X}} c_i^\top x, \tag{11}$$

A global big-M can be used when it upper-bounds all constraint-wise valid values in Eq. 11. An equality $h(x) = 0$ is treated as two possible violations, $h(x) \ge \epsilon$ or $h(x) \le -\epsilon$, using binary indicators. The condition

$$\sum_i y_i \ge 1 \tag{12}$$

requires at least one constraint of the other model to be violated.

Definition 3 (Auxiliary feasibility problems)**.** *Let $a_{\text{ref},j}^\top x \le b_{\text{ref},j}$ and $a_{\text{gen},i}^\top x \le b_{\text{gen},i}$ denote constraints of the reference model and the generated model. With this notation, Opt-Judger constructs two auxiliary feasibility problems:*

$$\begin{aligned} &\textit{reference-only problem} \quad \begin{cases} \textit{find} & x,\ y^{\text{gen}} \\ \textit{s.t.} & A_{\text{ref}} x \le b_{\text{ref}}, \\ & a_{\text{gen},i}^\top x \ge b_{\text{gen},i} + \epsilon - M_i^{\text{gen}}(1 - y_i^{\text{gen}}),\ \forall i, \\ & \sum_i y_i^{\text{gen}} \ge 1, \quad x \in \mathcal{X}, \quad y_i^{\text{gen}} \in \{0,1\}; \end{cases} \\ &\textit{generated-only problem} \quad \begin{cases} \textit{find} & x,\ y^{\text{ref}} \\ \textit{s.t.} & A_{\text{gen}} x \le b_{\text{gen}}, \\ & a_{\text{ref},j}^\top x \ge b_{\text{ref},j} + \epsilon - M_j^{\text{ref}}(1 - y_j^{\text{ref}}),\ \forall j, \\ & \sum_j y_j^{\text{ref}} \ge 1, \quad x \in \mathcal{X}, \quad y_j^{\text{ref}} \in \{0,1\}. \end{cases} \end{aligned} \tag{13}$$

The first problem in Eq. equation 13 is feasible exactly when $\Omega_{\text{ref-only}}$ contains a witness point, under the chosen feasibility tolerance. We discuss sensitivity to M and ϵ in Appendix D. The second problem is feasible exactly when $\Omega_{\text{gen-only}}$ contains a witness point under the same conditions. Thus Opt-Judger replaces direct set comparison with the two auxiliary feasibility problems in Eq. equation 13.

4.5 GENERATING FEEDBACK FROM WITNESS POINTS

The solver output provides the basis for both equivalence judgment and feedback. If both auxiliary feasibility problems are infeasible, Opt-Judger reports equivalence in the business-variable space. Otherwise, it records the witness point, its mismatch direction, and the violated constraint identified by the binary indicators in Eq. equation 13. A reference-only witness indicates that the generated model is over-constrained, whereas a generated-only witness indicates that it is under-constrained. Together with the problem description and the related model fragments, this solver-grounded evidence is provided to an LLM-based attribution step. The output is a short diagnostic message containing the mismatch direction, the violated constraint, the likely business rule, and a suggested correction target. This diagnostic message is intended to guide repair rather than to serve as an independent proof of the natural language error. The final equivalence decision is made by the solver under the stated assumptions, while the LLM only explains the solver-generated witness in terms of the original business rule. We discuss implementation scope and failure reporting in Appendix B.

5 OPT-JUDGER-BENCH

We construct Opt-Judger-Bench by encoding common constraint errors as controlled transformation operators. Seed instances are drawn from OptMATH, which pairs natural-language problems with reference Gurobi implementations, after excluding infeasible, unbounded, unparsable, and unsupported cases. We apply the operators to generate candidate models and automatically label whether they contain constraint errors. The benchmark covers Semantics-Preserving Transformations, Latent Misalignment Transformations, and Explicit Error Transformations, incorporating harmless reformulations alongside constraint errors. Due to limited space, we present a detailed analysis of Opt-Judger-Bench in Appendix A.

6 EXPERIMENTS

6.1 EXPERIMENT SETUPS

Benchmark We evaluate Opt-Judger on EquivaFormulation (Zhai et al., 2025) and Opt-Judger-Bench. EquivaFormulation contains equivalent and non-equivalent formulations generated through controlled transformations. Opt-Judger-Bench covers semantics-preserving reformulations and common constraint errors, including those that leave optimal results unchanged. Together, they assess whether evaluators detect constraint errors while accepting harmless reformulations.

Baselines We compare our method with representative baselines from two categories. Specification-matching methods include (1) Canonical matching (Ramamonjison et al., 2023), which compares variables, objectives, and constraints at the declaration level, and (2) WL-test (Wang et al., 2024), which represents each formulation as a variable-constraint bipartite graph and evaluates structural equivalence through graph-based matching. We also evaluate result- or solution-based methods, including (3) Execution accuracy (AhmadiTeshnizi et al., 2024; Huang et al., 2024; Yang et al., 2025), which solves both formulations and compares their optimal objective values or optimal solutions, and (4) EquivaMap-style mapping verification (Zhai et al., 2025), where an LLM first infers a variable mapping between the two formulations and the verifier then checks whether the mapped optimal solution remains feasible and optimal under the reference model. For completeness, we further include (5) Naive-LLM (Zhai et al., 2025), a direct LLM-based judge that prompts an LLM to decide whether the candidate formulation is equivalent to the reference formulation without solver-backed feasible-region verification.

Metric Under the objective-consistency check, we use feasible-region equivalence as the evaluation metric. A generated formulation is considered correct if its projected feasible region over the aligned business variables is identical to that of the reference formulation.

Implementation Opt-Judger uses GPT-4o(OpenAI, 2024) for semantic variable alignment and witness-based attribution. EquivaMap and Naive-LLM use the same LLM backbone. All LLM calls use deterministic decoding, and all solver-backed verification steps are implemented with Gurobi.

Table 1: Accuracy of equivalence judgments on EquivaFormulation and Opt-Judger-Bench.

Dataset	Subset / Transformations	Canonical Acc.	Execution Acc.	WL-test	Naive-LLM	EquivaMap	Opt-Judger
Equiva-Formulation	**Overall**	**34.3**	**77.1**	**34.3**	**63.9**	**100.0**	**100.0**
	Equivalent Transformations						
	Substitute Objective Functions	0	100	0	91.2	100	100
	Add Slack Variables	0	100	0	36.1	100	100
	Base-10 Representation	0	100	0	53.1	100	100
	Add Valid Inequalities	0	100	0	3.3	100	100
	Rescaling	0	0	0	69.9	100	100
	Linear Combinations	0	100	0	24.4	100	100
	Non-Equivalent Transformations						
	Random Order	100	100	100	98.7	100	100
	Loose Constraints	100	100	100	93.6	100	100
	Feasibility	100	0	100	100	100	100
Opt-Judger-Bench	**Overall**	**59.6**	**65.5**	**59.6**	**64.1**	**65.5**	**100.0**
	Semantics-Preserving Transformations						
	Add Slack Variables	0	100	0	82.5	100	100
	Variable Decomposition	0	100	0	48.2	100	100
	Add Redundant Constraints	0	100	0	63.2	100	100
	Latent Misalignment Transformations						
	Loose Non-binding Constraints	100	0	100	23.5	0	100
	Safe-zone Tightening	100	0	100	88.4	0	100
	Binding Constraint Rotation	100	0	100	89.6	0	100
	Explicit Error Transformations						
	Loose Binding Constraints	100	100	100	19.6	100	100
	Fatal Tightening	100	100	100	96.9	100	100

6.2 Performance of Opt-Judger

As shown in Table 1, Opt-Judger achieves consistently strong performance on both EquivaFormulation and Opt-Judger-Bench. The results on Opt-Judger-Bench further reveal three observations. First, compared with solution-based evaluation, Opt-Judger is more effective at detecting latent constraint errors. While solution-based methods overlook Latent Misalignment Transformations that leave optimal results unchanged, Opt-Judger detects these errors through bidirectional feasibility verification. Second, Opt-Judger demonstrates greater robustness than structure-based evaluation methods on Semantics-Preserving Transformations. Even when harmless reformulations alter the model structure, Opt-Judger correctly recognizes that the models permit the same decisions. Third, Opt-Judger correctly detects all constraint errors introduced by Explicit Error Transformations, maintaining consistent accuracy across the three categories.

6.3 Feasible-region Evaluation of Modeling Methods

We evaluate representative modeling methods on NLP4LP, which provides reference models, using Execution Accuracy, EquivaMap, and Opt-Judger. Appendix G provides method details. As shown in Table 2, Opt-Judger yields lower estimates of modeling accuracy than both Execution Accuracy and EquivaMap across all methods, revealing constraint errors in formulations that recover the correct optimal value. Such errors may admit decisions that violate the problem requirements or exclude valid ones without affecting the current optimum. This gap suggests that solution-based evaluation can overestimate modeling capabilities by overlooking constraint errors that leave optimal results unchanged. Opt-Judger assesses these errors by checking whether generated and reference models permit the same decisions.

6.4 Ablation Studies

We evaluate the impact of framework components on feasible-region equivalence evaluation, as shown in Table 3. Compared with the full Opt-Judger, removing variable projection leads to failure on structural reformulations, confirming its role in aligning models within a shared business-variable space. Similarly, relying on a single mismatch direction introduces critical blind spots: reference-only search misses under-constrained errors, while generated-only search fails to detect over-constrained errors. These results highlight the importance of variable projection and bidirectional feasibility verification for assessing whether generated constraints faithfully capture the problem requirements.

Table 2: Modeling accuracy under different evaluators. Red arrows indicate potential overestimation relative to Opt-Judger in percentage points.

Method	Execution Acc.	EquivaMap	Opt-Judger
High-Capacity LLMs			
GPT-4o	69.6 ↑ 7.1	69.6 ↑ 7.1	62.5
OpenAI-o3	75.1 ↑ 4.3	75.1 ↑ 4.3	70.8
DeepSeek-V3	70.0 ↑ 4.0	68.0 ↑ 2.0	66.0
DeepSeek-V4-Flash	78.7 ↑ 6.4	78.4 ↑ 6.1	72.3
Prompt-based Modeling Methods			
COE	53.0 ↑ 9.5	48.5 ↑ 5.0	43.5
OptiMUS	69.6 ↑ 4.0	69.3 ↑ 3.7	65.6
Finetuned Modeling Methods			
ORLM	68.4 ↑ 5.9	68.0 ↑ 5.5	62.5
LLMOPT	70.4 ↑ 3.2	69.3 ↑ 2.1	67.2
OptMATH	77.9 ↑ 4.8	77.6 ↑ 4.5	73.1
SIRL	81.4 ↑ 5.1	81.4 ↑ 5.1	76.3

Table 3: Ablation study of the components on Opt-Judger.

Subset / Transformations	w/o Variable Projection	Generated-only Mismatch	Reference-only Mismatch	Opt-Judger Full
Opt-Judger-Bench (Overall)	**86.1**	**72.7**	**67.2**	**100.0**
Semantics-Preserving Transformations				
Add Slack Variables	0	100	100	100
Variable Decomposition	100	100	100	100
Add Redundant Constraints	100	100	100	100
Latent Misalignment Transformations				
Loose Non-binding Constraints	100	100	0	100
Safe-zone Tightening	100	0	100	100
Binding Constraint Rotation	100	80.2	19.8	100
Explicit Error Transformations				
Loose Binding Constraints	100	100	0	100
Fatal Tightening	100	0	100	100

6.5 Analysis on the Evaluation Time

We measure the average runtime per generated formulation on Opt-Judger-Bench. As shown in Table 4, Opt-Judger incurs additional runtime relative to lightweight evaluators, while remaining comparable to other LLM-based evaluators. Its cost is dominated by semantic variable alignment, whereas solver-backed mismatch search adds little overhead, indicating that feasible-region verification is lightweight after alignment. This added cost is reasonable for offline equivalence checking given the improved reliability.

Table 4: Average runtime per formulation.

Method	Time (s)	Accuracy
Execution Acc.	0.02	65.5
WL-test	5.45	59.6
EquivaMap	12.93	65.5
Naive-LLM	11.84	64.1
Opt-Judger (alignment)	12.58	–
Opt-Judger (search)	0.03	–
Opt-Judger (full)	12.61	**100.0**

7 Conclusion

We introduce Opt-Judger, a solver-backed framework for fine-grained evaluation of LLM-generated optimization models. By comparing feasible decisions in an aligned business-variable space, it distinguishes constraint errors from harmless reformulations and provides witness points for error diagnosis. We also construct Opt-Judger-Bench using controlled transformations. Our findings show that these comparisons support a more comprehensive assessment of modeling capabilities, while witness points provide concrete evidence to guide model improvement.

AI USE STATEMENT

In this work, we used generative AI tools to assist with dataset generation, software implementation, experimental support, result interpretation, and manuscript editing. Specifically, generative AI was used to generate portions of the benchmark data, suggest experimental parameters, create or edit software code, assist in interpreting experimental results, and improve the clarity and readability of the manuscript.

Generative AI was not used to develop the theoretical models or conceptual framework of this work, formulate mathematical claims, construct or assist with proofs, propose or refine research hypotheses, or determine the core research methodology and experimental design.

All AI-assisted outputs were reviewed and verified by the authors. Generated benchmark instances were checked according to the benchmark construction and validation procedures described in the paper. We also manually inspected and tested AI-assisted code, independently validated suggested experimental parameters, and carefully revised AI-assisted text to ensure scientific accuracy, rigor, and consistency with the authors' original research ideas. The authors take full responsibility for the final content of this work, including the dataset, scientific claims, experiments, code, and artifacts produced with the assistance of generative AI.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 577–596. PMLR, 2024. URL https://proceedings.mlr.press/v235/ahmaditeshnizi24a.html.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 1864–1886. PMLR, 2025. URL https://proceedings.mlr.press/v267/astorga25a.html.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed RL: Grounding large language models for authentic optimization modeling. In *Advances in Neural Information Processing Systems*, 2025.

Jordan Goblet, Fabien Willame, and Jan Grauls. A mixed-integer linear programming approach for collateral management. *Decision Analytics Journal*, 10:100429, 2024. doi: 10.1016/j.dajour.2024.100429. URL https://www.sciencedirect.com/science/article/pii/S277266222400033X.

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*, 73(6):2986–3009, 2025.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. MAMO: A mathematical modeling benchmark with solvers. *arXiv preprint arXiv:2405.13144*, 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*, 2025.

Minwei Kong, Ao Qu, Xiaotong Guo, Wenbin Ouyang, Chonghe Jiang, Han Zheng, Yining Ma, Dingyi Zhuang, Yuhan Tang, Junyi Li, Shenhao Wang, Haris Koutsopoulos, Hai Wang, Cathy Wu, and Jinhua Zhao. AlphaOPT: Formulating optimization programs with self-improving LLM experience library. *arXiv preprint arXiv:2510.18428*, 2025.

Zhong Li, Hongliang Lu, Tao Wei, Yuxuan Chen, Wenyu Liu, Yuan Lan, Fan Zhang, and Zaiwen Wen. Constructing industrial-scale optimization modeling benchmark. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0Q7adIsv6p.

Chunyang Liu, Houzhen Ma, Hengxu Zhang, Xiaohan Shi, and Fang Shi. A MILP-based battery degradation model for economic scheduling of power system. *IEEE Transactions on Sustainable Energy*, 14(2):1000–1009, 2023. doi: 10.1109/TSTE.2022.3232370. URL https://ieeexplore.ieee.org/document/9999506/.

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. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

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. *arXiv preprint arXiv:2403.01131*, 2024.

OpenAI. GPT-4o system card. *arXiv preprint arXiv:2410.21276*, 2024. URL https://arxiv.org/abs/2410.21276.

Rindra Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *Proceedings of Machine Learning Research*, 2023.

Raghav Thind, Youran Sun, Ling Liang, and Haizhao Yang. OptimAI: Optimization from natural language using LLM-powered AI agents. *arXiv preprint arXiv:2504.16918*, 2025.

Russ J. Vander Wiel and Nikolaos V. Sahinidis. Heuristic bounds and test problem generation for the time-dependent traveling salesman problem. *Transportation Science*, 29(2):167–183, 1995.

Zhuohan Wang, Ziwei Zhu, Yizhou Han, Yufeng Lin, Zhihang Lin, Ruoyu Sun, and Tian Ding. OptiBench: Benchmarking large language models in optimization modeling with equivalence-detection evaluation. *OpenReview preprint*, 2024.

H. Paul Williams. *Model Building in Mathematical Programming*. John Wiley & Sons, fifth edition, 2013. ISBN 9781118443330.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. Chain-of-experts: When LLMs meet complex operations research problems. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=HobyL1B9CZ.

Linzi Xing, Xinglu Wang, Yuxi Feng, Zhenan Fan, Jing Xiong, Zhijiang Guo, Xiaojin Fu, Rindra Ramamonjison, Mahdi Mostajabdaveh, Xiongwei Han, Zirui Zhou, and Yong Zhang. Towards human-aligned evaluation for linear programming word problems. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pp. 16550–16556. ELRA and ICCL, 2024. URL https://aclanthology.org/2024.lrec-main.1438/.

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. In *Proceedings of the Thirteenth International Conference on Learning Representations*, 2025.

Haotian Zhai, Connor Lawless, Ellen Vitercik, and Liu Leqi. EquivaMap: Leveraging LLMs for automatic equivalence checking of optimization formulations. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 74288–74305. PMLR, 2025. URL https://proceedings.mlr.press/v267/zhai25a.html.

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. StepORLM: A self-evolving framework with generative process supervision for operations research language models. *arXiv preprint arXiv:2509.22558*, 2025.

A OPT-JUDGER-BENCH CONSTRUCTION AND DOCUMENTATION

This appendix provides additional details about the construction of Opt-Judger-Bench. The benchmark is designed to evaluate whether an automatic evaluator can judge feasible-region equivalence in the business-variable space, rather than merely comparing objective values, solver outputs, or surface-level formulation structure. Each benchmark instance starts from a natural-language optimization problem and an executable reference formulation. We then apply controlled transformation operators to produce candidate formulations whose equivalence labels are known by construction.

A.1 DESIGN GOALS AND DATA PREPARATION

The benchmark has three goals. First, it separates feasible-region equivalence from surface similarity. A generated model may introduce auxiliary variables, rewrite constraints, or add redundant constraints while still preserving the same business decisions. Second, it exposes the blind spot of result-based evaluation. A model may give the same objective value or the same optimal solution on one instance while still accepting or rejecting the wrong business decisions elsewhere. Third, it supports error attribution. When a model is not equivalent, the benchmark identifies the constraint error that was introduced, so that evaluation can provide feedback beyond a binary score.

Before applying transformations, we clean the base data to make the labels well defined. We remove instances whose solver status is infeasible or unbounded and keep only instances with a finite optimum. This avoids cases where objective-value comparison is ambiguous. We also apply anti-cheating preprocessing so that the benchmark does not reward memorizing code patterns. The order of parameters, variable declarations, and constraints is randomly shuffled. Variable names are replaced with dynamic names rather than common symbols such as x, y, or m. The natural language description is then rewritten according to the new names, so that the text and the model remain aligned while simple surface shortcuts are weakened.

A.2 TRANSFORMATION CATEGORIES

The transformation operators are organized into three categories according to their effect on the business feasible region and on the current optimum.

Semantics-preserving transformations. These transformations change the algebraic or structural form of the formulation while preserving the projected feasible region over the business variables. They are equivalent transformations and should be accepted by a semantics-aware evaluator. This category includes adding slack variables, decomposing variables, and adding redundant constraints.

Latent misalignment transformations. These transformations change the feasible region but are constructed so that the current optimal solution or optimal objective value can remain unchanged. They create challenging non-equivalent cases that result-based evaluation can easily miss. This category includes loosening non-binding constraints, safe-zone tightening, and binding constraint rotation.

Explicit error transformations. These transformations introduce visible modeling errors that change the feasible region and typically also affect the current optimization result. They serve as direct negative cases. This category includes loosening binding constraints and fatal tightening.

A.3 TRANSFORMATION OPERATORS

We present the transformation operators used to construct Opt-Judger-Bench in Table 5. For each operator, we report its construction procedure, a representative before/after example, and the feasible-region equivalence label in the projected business-variable space. The labels are determined by the transformation semantics: semantics-preserving transformations preserve the projected feasible region by construction, while latent misalignment and explicit error transformations deliberately change it. This design yields reliable ground-truth labels for testing whether evaluators can separate equivalent reformulations from feasible-region-altering modeling errors. The **Size** column further reports the number of transformed formulations generated by each operator.

Table 5: Overview of Equivalent and Non-equivalent Transformation Operators in Opt-Judger-Bench.

Transformation Name	How It Is Transformed	Example (Before/After)	Equivalent?	Size
Add Slack Variables	Transform inequality constraints into equalities by adding non-negative slack variables.	**Before:** $Ax \leq b$ **After:** $Ax + s = b,\ s \geq 0$	Yes	228
Variable Decomposition	Decompose a variable into the difference of two non-negative variables.	**Before:** $x \in \mathbb{R}$ **After:** $x = x^+ - x^-,\ x^+, x^- \geq 0$	Yes	224
Add Redundant Constraints	Add algebraically dependent or geometrically redundant constraints that do not remove any feasible solutions.	**Before:** $\{x + y \leq 5\}$ **After:** $\{x + y \leq 5,\ 2x + 2y \leq 10\}$	Yes	212
Loose Non-binding Constraints	Remove constraints with positive slack at the optimal solution, thereby enlarging the feasible region while preserving the current optimum.	**Before:** $\max_{x,y} x - y$ s.t. $x \leq 1,\ 0 \leq y \leq 10,$ **After:** s.t. $x \leq 1,\ 0 \leq y,$	No	204
Safe-zone Tightening	Add artificial bounds far from the optimal solution that do not affect the current optimum but alter the feasible region.	**Before:** $\max_{x} x$ s.t. $x \leq 1$ **After:** s.t. $0 \leq x \leq 1$	No	189
Binding Constraint Rotation	Rotate the normal vector of a binding constraint while keeping the current optimal point on the rotated boundary.	**Before:** $\max_{x,y} x$ s.t. $x \leq 1,\ x + y \leq 1,$ **After:** s.t. $x \leq 1,\ x + 2y \leq 1,$	No	173
Loose Binding Constraints	Remove constraints that are tight at the optimal solution, causing the optimum to shift significantly.	**Before:** $x + y = 5$ **After:** remove $x + y = 5$	No	189
Fatal Tightening	Add a constraint that directly conflicts with the original optimal solution.	**Before:** optimal x^* feasible **After:** add $x \leq x^* - 1$	No	223

A.4 Statistics of Opt-Judger-Bench

Opt-Judger-Bench is constructed from a filtered seed set of source optimization problems whose reference formulations can be parsed, executed, and verified by the solver-backed evaluation pipeline. After removing infeasible, unbounded, unparsable, and unsupported instances, the benchmark contains 241 source problems and 1,642 transformed formulations. The resulting benchmark includes LP and MILP formulations with binary, integer, and continuous variables, as well as moderately large constraint sets and natural-language specifications. The detailed statistics of the filtered seed problems and their transformed formulations are reported in Table 6.

Table 6: Statistics of Opt-Judger-Bench.

Category	Attribute	Value
Scale	Source Problems	241
	Transformed Problems	1642
Variable Type	Avg. Binary Variables	13
	Avg. Integer Variables	4
	Avg. Continuous Variables	27
	Avg. Variables	44
Structure	Avg. Constraints	41
Text	Avg. Problem Length	3346

A.5 Analysis on the Opt-Judger Benchmark

We analyze the domain coverage and optimization-type distribution of Opt-Judger-Bench in Figure 3. The filtered seed problems span diverse application domains, with Manufacturing, Supply Chain, Healthcare, and Transportation forming the largest groups. The remaining problems cover Energy, Logistics, Services, and other domains, indicating that the benchmark is not concentrated in a single

application scenario. The problem-type breakdown further shows that the transformed formulations include MILP, IP, and LP instances across all transformation operators. MILP instances constitute the largest portion for most operators, followed by IP and LP problems, which reflects the common use of discrete decision variables in real-world optimization modeling. Importantly, each operator is applied to multiple formulation types rather than to a single homogeneous problem class. This diversity helps evaluate whether an automatic equivalence checker can robustly handle different optimization structures and application domains.

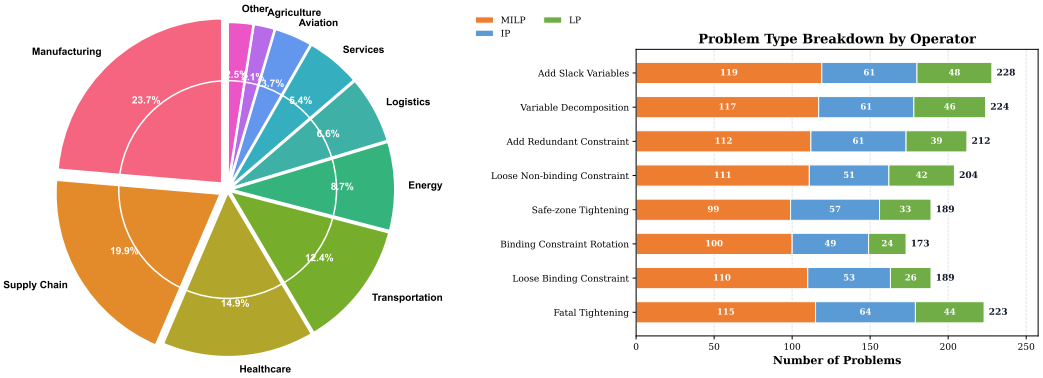

Figure 3: Distribution analysis of Opt-Judger-Bench. Left: application-domain distribution of the filtered source problems. Right: problem-type breakdown across transformation operators.

B Implementation Scope and Failure Reporting

This appendix clarifies the operational scope of Opt-Judger and describes how unsupported cases are reported. Opt-Judger is designed as a solver-backed evaluator for comparing the constraint semantics of two optimization formulations in the aligned business-variable space. Its equivalence judgment is rigorous only under explicit parsing, and alignment assumptions. When these assumptions are not satisfied, Opt-Judger reports a prerequisite failure rather than returning an unsupported equivalence judgment.

B.1 Verification Scope

Opt-Judger takes as input a natural-language problem description, a reference formulation, and a generated formulation. The reference formulation is treated as the intended target formulation. Opt-Judger does not attempt to verify whether the reference formulation itself correctly captures the natural-language problem. Instead, it evaluates whether the generated formulation preserves the same feasible decisions as the reference formulation after both models are projected onto the aligned business-variable space.

The current implementation focuses on formulations that can be parsed into explicit solver constraints and compared through linear or mixed-integer linear feasibility problems. This scope covers a broad class of LP, IP, and MILP formulations, including formulations with auxiliary variables, slack variables, decomposed variables, and redundant constraints. For such cases, Opt-Judger can align business variables, eliminate or encode auxiliary variables, and perform bidirectional witness search to determine feasible-region equivalence.

Models outside this scope are not forced into an equivalence or non-equivalence decision. In these cases, Opt-Judger returns a failure status that identifies the unsupported stage.

B.2 Assumptions

Opt-Judger relies on the following assumptions.

Explicit constraint representation. Both the reference and generated formulations must be converted into explicit variables, domains, objectives, and constraints. The solver-backed verification

step compares mathematical constraints rather than raw text or unexecuted code. If the formulation cannot be parsed into a supported constraint representation, Opt-Judger reports a parsing failure.

Shared business-variable space. The two formulations must admit a reliable alignment between their business variables. Auxiliary variables may differ across formulations, but the original decision variables that represent business decisions must be mapped into a common space. If Opt-Judger cannot establish a reliable alignment for the business variables, the feasible-region comparison is undefined and an alignment failure is reported.

B.3 Failure Reporting

Opt-Judger separates equivalence judgments from prerequisite failures. An equivalence judgment is returned only when the verification pipeline successfully constructs and solves both mismatch-search problems. If a prerequisite step fails, the system returns a diagnostic status indicating the stage at which verification became unsupported. Table 7 summarizes the main failure types.

Table 7: Failure types reported by Opt-Judger before equivalence verification.

Failure Type	Trigger	Reported Status	Reason for Separate Reporting
Parsing failure	The formulation cannot be converted into explicit variables, domains, objectives, and constraints.	`PARSE_FAIL`	The mathematical object to be compared is not available.
Execution failure	The solver model cannot be built or executed due to invalid code, missing data, or solver errors.	`EXEC_FAIL`	Solver-backed verification requires executable constraint representations.
Alignment failure	The business variables of the reference and generated formulations cannot be reliably mapped into a shared space.	`ALIGN_FAIL`	Feasible-region equivalence is undefined without a common business-variable space.
Projection failure	An integer auxiliary lacks finite bounds or cannot be locally eliminated while preserving the projected feasible region.	`PROJ_FAIL`	Feasible-region equivalence cannot be verified without a projected constraint representation over the business variables.
Unsupported model class	The formulation contains nonlinear, nonconvex, disjunctive, or solver-specific constructs outside the current verification scope.	`UNSUPPORTED`	The current implementation does not provide an exact equivalence checker for this model class.

C Projection to the Business-Variable Space

This appendix provides additional details about the projection step used in Opt-Judger. The purpose of projection is to remove auxiliary variables introduced by a formulation while preserving the feasible decisions over the original business variables. This step is necessary because two equivalent formulations may use different slack variables, split variables, linearization variables, or other auxiliary constructs. Opt-Judger therefore compares formulations only after projecting both models onto the aligned business-variable space.

C.1 Problem Setting

Let x denote the aligned business variables and let z denote auxiliary variables. For a linear formulation with auxiliary variables, the feasible region can be written as

$$\mathcal{P} = \{(x, z) \mid Ax + Bz \leq b,\ Ex + Hz = e,\ x \in \mathcal{X},\ z \in \mathcal{Z}\}. \tag{14}$$

The projected feasible region used for equivalence checking is

$$\mathrm{proj}_x(\mathcal{P}) = \{x \in \mathcal{X} \mid \exists z \in \mathcal{Z},\ Ax + Bz \leq b,\ Ex + Hz = e\}. \tag{15}$$

The goal of the projection procedure is to construct an equivalent constraint representation over x only, whenever this can be done within the supported linear verification scope.

C.2 PROJECTION PROCEDURE

Opt-Judger applies projection in two main stages. First, it removes auxiliary variables that are explicitly defined by equalities. Second, it eliminates the remaining auxiliary variables that appear in inequality constraints. When such variables carry integrality requirements, Opt-Judger uses integer-preserving encodings when such encodings are available, so that the elimination preserves the feasible region over the business variables.

Eliminating explicit auxiliary definitions. Many auxiliary variables are introduced through explicit equalities, such as slack-variable definitions, split-variable definitions, or linearization definitions. When an auxiliary variable can be solved from an equality without changing the feasible set, Opt-Judger substitutes the expression into all other constraints and removes the auxiliary variable. For example, an equality $z = a^\top x + c$ allows every occurrence of z to be replaced by $a^\top x + c$. For a system of linear equalities, Opt-Judger applies Gaussian elimination to identify eliminable auxiliary variables and substitute them into the remaining constraints.

Eliminating auxiliary variables in inequalities. After explicit substitutions, some auxiliary variables may remain in inequality constraints. For each such variable z_k, Opt-Judger first rewrites the constraints involving z_k into a form suitable for projection. If z_k is continuous, the constraints are rewritten as lower and upper bounds on z_k. If z_k is subject to integrality requirements, Opt-Judger uses an integer-preserving encoding before elimination when available, so that the encoded constraints preserve the same feasible assignments over the business variables.

For the continuous bound-elimination step, suppose the constraints imply lower bounds $z_k \geq \ell_i(x, z_{-k})$ and upper bounds $z_k \leq u_j(x, z_{-k})$. Eliminating z_k requires every lower bound to be no larger than every upper bound, yielding constraints

$$\ell_i(x, z_{-k}) \leq u_j(x, z_{-k}), \qquad \forall i, j. \tag{16}$$

Constraints that do not contain z_k are retained. This operation is repeated until all auxiliary variables outside the business-variable space have either been eliminated or encoded into the projected representation. Finally, Opt-Judger removes duplicate constraints and optionally prunes redundant inequalities.

C.3 ALGORITHM

Algorithm. The procedure maintains a constraint system over the current variables and removes auxiliary variables one at a time.

Algorithm 1 Exact projection to the business variable space

Require: Inequalities $Ax + Bz \leq b$, equalities $Ex + Hz = e$, business variables x, continuous auxiliary variables z
Ensure: Constraints over x defining $\mathrm{proj}_x(\mathcal{P})$
1: Initialize $\mathcal{C} \leftarrow \{Ax + Bz \leq b,\ Ex + Hz = e\}$
2: Use Gaussian elimination on the equality subsystem to solve explicit auxiliary definitions $z_k = \ell_k(x, z_{-k})$ whenever possible
3: **for all** solved auxiliary variables z_k **do**
4: Substitute ℓ_k into every constraint in $\mathcal{C}$
5: Remove z_k from the auxiliary variable set
6: **end for**
7: **for all** remaining auxiliary variables z_k **do**
8: Rewrite each inequality containing z_k as either $z_k \leq u_s(x, z_{-k})$ or $z_k \geq l_t(x, z_{-k})$
9: **for all** lower bounds l_t and upper bounds u_s **do**
10: Add the inequality $l_t(x, z_{-k}) \leq u_s(x, z_{-k})$ to $\mathcal{C}$
11: **end for**
12: Keep constraints that do not contain z_k and remove old constraints that contain z_k
13: Remove duplicate constraints and certified redundant constraints when detected
14: **end for**
15: **return** the remaining constraints over x

C.4 CORRECTNESS

We justify the projection procedure using two standard elimination theorems. The first theorem states that Gaussian elimination over equality-defined auxiliary variables preserves the projection onto the remaining variables. The second theorem states that Fourier–Motzkin elimination exactly preserves the projection of a linear system over continuous variables. Auxiliary variables with integrality requirements are handled through integer-preserving encodings before projection, so that their elimination preserves the feasible assignments over the business variables. Based on these results, we prove the correctness of Algorithm 1.

Theorem 1 (Gaussian elimination)**.** *Consider a feasible region with linear equality-defined auxiliary variables:*

$$\mathcal{P} = \{(x, z, w) \mid E_x x + E_z z + E_w w = e,\ Gx + Hz + Kw \le h\}, \tag{17}$$

where x denotes business variables, z denotes auxiliary variables to be eliminated, and w denotes the remaining variables. Suppose Gaussian elimination on the equality subsystem expresses z as an affine function of (x, w):

$$z = R_x x + R_w w + r. \tag{18}$$

Let $\mathcal{P}'$ be obtained by substituting this expression into all remaining constraints and removing z and its defining equalities. Then

$$\mathrm{proj}_{x,w}(\mathcal{P}) = \mathcal{P}'. \tag{19}$$

Consequently,

$$\mathrm{proj}_x(\mathcal{P}) = \mathrm{proj}_x(\mathcal{P}'). \tag{20}$$

Proof. Gaussian elimination applies elementary row operations to the equality subsystem, which preserve the solution set of the linear equalities. Under the stated assumption, the eliminated auxiliary variables z are represented as affine functions of the remaining variables (x, w). For any feasible point $(x, z, w) \in \mathcal{P}$, substituting $z = R_x x + R_w w + r$ into the remaining constraints yields a feasible point $(x, w) \in \mathcal{P}'$. Conversely, for any $(x, w) \in \mathcal{P}'$, setting $z = R_x x + R_w w + r$ satisfies the original equality subsystem and all substituted constraints, so $(x, z, w) \in \mathcal{P}$. Thus the feasible assignments over (x, w) are identical before and after Gaussian elimination, and projecting further onto x gives the same set. □

Theorem 2 (Fourier–Motzkin elimination)**.** *Consider a polyhedron over variables (x, z) where z is a continuous auxiliary variable:*

$$\mathcal{P} = \{(x, z) \mid A_0 x \le b_0,\ z \ge \ell_i(x),\ i \in \mathcal{L},\ z \le u_j(x),\ j \in \mathcal{U}\}. \tag{21}$$

Let $\mathcal{P}'$ be the system over x defined by

$$\mathcal{P}' = \{x \mid A_0 x \le b_0,\ \ell_i(x) \le u_j(x),\ \forall i \in \mathcal{L},\ j \in \mathcal{U}\}. \tag{22}$$

Then

$$\mathrm{proj}_x(\mathcal{P}) = \mathcal{P}'. \tag{23}$$

Proof. If $(x, z) \in \mathcal{P}$, then z satisfies every lower bound and every upper bound. Therefore, for all $i \in \mathcal{L}$ and $j \in \mathcal{U}$, we have

$$\ell_i(x) \le z \le u_j(x), \tag{24}$$

which implies $\ell_i(x) \le u_j(x)$. Hence $x \in \mathcal{P}'$.

Conversely, suppose $x \in \mathcal{P}'$. Then all retained constraints $A_0 x \le b_0$ hold, and every lower bound is no larger than every upper bound. If both lower and upper bounds exist, the interval

$$\left[\max_{i \in \mathcal{L}} \ell_i(x),\ \min_{j \in \mathcal{U}} u_j(x)\right] \tag{25}$$

is nonempty, and choosing any z in this interval satisfies all constraints involving z. If only lower bounds or only upper bounds exist, a feasible z can be chosen because z is continuous and unbounded in the missing direction. If no constraint contains z, any value of z is feasible. Therefore, there exists a z such that $(x, z) \in \mathcal{P}$, which proves $x \in \mathrm{proj}_x(\mathcal{P})$. □

Proposition 2 (Correctness of business-variable projection)**.** *When auxiliary variables are eliminated by Gaussian elimination, Fourier–Motzkin elimination, or integer-preserving encoding, Algorithm 1 preserves the projected feasible region over the business variables.*

Proof. By Theorem 1, Gaussian elimination over equality-defined auxiliary variables preserves the feasible assignments over the remaining variables. By Theorem 2, each Fourier–Motzkin elimination step exactly preserves the projection after removing one continuous auxiliary variable. For auxiliary variables with integrality requirements, the integer-preserving encoding preserves the same feasible assignments over the business variables by construction. Since each step preserves the projection onto the remaining variables, their sequential composition preserves the final projection onto the business-variable space. Therefore, Algorithm 1 returns a constraint system equivalent to the original formulation projected onto the business variables. □

C.5 Complexity

The cost of projection depends on the number and structure of auxiliary variables. Gaussian elimination over an equality subsystem with q variables has dense complexity $O(q^3)$, plus the cost of substituting eliminated variables into the remaining constraints. For auxiliary variables that remain in inequalities, Fourier–Motzkin elimination can generate up to pr new inequalities when a variable has p lower bounds and r upper bounds, leading to worst-case exponential growth when many such variables are eliminated. If integrality requirements are present, Opt-Judger first applies an integer-preserving encoding, whose cost depends on the encoding size and the number of involved constraints. In practice, most reformulation-induced auxiliary variables are removed during equality elimination, and duplicate or redundant inequalities are pruned after projection, making the procedure tractable for the benchmark instances considered in this work.

D Sensitivity Analysis of Big-M, ϵ, and Error Attribution

This appendix reports additional experimental analyses on the complement-encoding hyperparameters and the attribution module of Opt-Judger. We first describe the experimental settings for the big-M constants and the violation tolerance ϵ used in witness search. We then analyze the error introduced by the two big-M settings and by ϵ, followed by an analysis of attribution performance.

D.1 Experimental Settings

Opt-Judger uses complement encoding to search for witness points that are feasible in one projected feasible region but violate at least one constraint of the other. The exactness of this encoding requires the inactive violation-indicator constraints to be relaxed by valid big-M constants, as specified in Eq. 11. In our experiments, instead of assigning a separate big-M value to each encoded constraint, we use a single global big-M implementation constant, chosen to be larger than the maximum constraint-wise value observed after projection and benchmark preprocessing. Constraint violations are evaluated with an ϵ-separated threshold to avoid numerical ambiguity near constraint boundaries. The same hyperparameter settings are used for both reference-only and generated-only witness searches, as summarized in Table 8.

Table 8: Hyperparameter settings for complement encoding and witness search.

Hyperparameter	Value	Description
Big-M constant	10^4	Relaxes inactive violation-indicator constraints in the complement encoding.
Violation threshold ϵ	10^{-4}	Minimum violation margin required for a constraint to be treated as violated.

D.2 Sensitivity to Big-M and ϵ

Impact of Big-M As shown in the left subfigure of Figure 4, the verification accuracy is sensitive to the value of the big-M constant. When M is too small, inactive violation indicators are not sufficiently relaxed, which can prevent the solver from finding valid witness points and lead to

incorrect equivalence judgments. As M increases, the complement encoding becomes valid and the accuracy quickly stabilizes. This result supports our default setting of $M = 10^4$.

Impact of ϵ As shown in the right subfigure of Figure 4, Opt-Judger remains robust across different values of the violation threshold ϵ. A large ϵ may ignore small but real feasible-region mismatches, while a sufficiently small ϵ allows the solver to detect finer constraint violations. The accuracy becomes stable when ϵ is small enough. In our experiments, we set $\epsilon = 10^{-4}$.

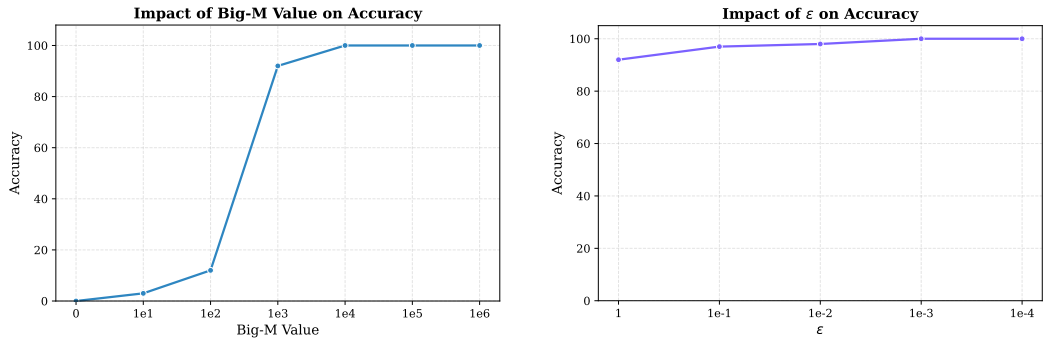

Figure 4: Sensitivity analysis. Left: impact of the big-M value on equivalence-verification accuracy. Right: impact of the violation threshold ϵ on equivalence-verification accuracy.

D.3 Analysis on the Error Attribution

Error Attribution Accuracy As shown in the left subfigure of Figure 5, we evaluate the error attribution hit rate on non-equivalent transformations. Opt-Judger achieves accurate attribution across all transformation types, including both Latent Misalignment Transformations and Explicit Error Transformations. This demonstrates that the witness point generated by bidirectional feasible-region verification provides a reliable signal for identifying the source of the mismatch. In particular, even when the optimal solution remains unchanged, Opt-Judger can still localize the violated feasible-region boundary rather than relying on objective-value differences.

Repair Success Rate As illustrated in the right subfigure of Figure 5, we further evaluate whether the attributed errors can help the model repair incorrect formulations. Compared with naive self-correction, witness-driven self-correction achieves consistently higher repair accuracy across all Transformation types. This improvement shows that Opt-Judger provides actionable feedback: the witness point indicates which business decision is treated differently by the reference and generated models, and the violated constraint boundary guides the model toward correcting the feasible region. These results confirm that Opt-Judger not only detects non-equivalence, but also supports effective model self-correction through grounded diagnostic feedback.

E Analysis of Evaluation Time and Robustness

We examine two aspects of evaluation at larger scales. The first experiment measures judgment accuracy as the number of variables increases. The second compares the solver effort of bidirectional verification with that of solving large-scale optimization models. These experiments address accuracy and computational cost separately.

E.1 Evaluation Accuracy across Problem Scales

Experimental design. We consider the explicit and latent errors defined in our benchmark and hold the constraint error type fixed for each sample. Inspired by the indexed modeling structures described in MIPLIB-NL (Li et al., 2026), we increase the number of decision variables while retaining the specified error type. At each scale, we evaluate 10 samples and compare the constraint error detection accuracy of Canonical matching, Execution Accuracy, WL-test, Naive-LLM, EquivaMap, and Opt-Judger. The results are shown in the right panel of Figure 1.

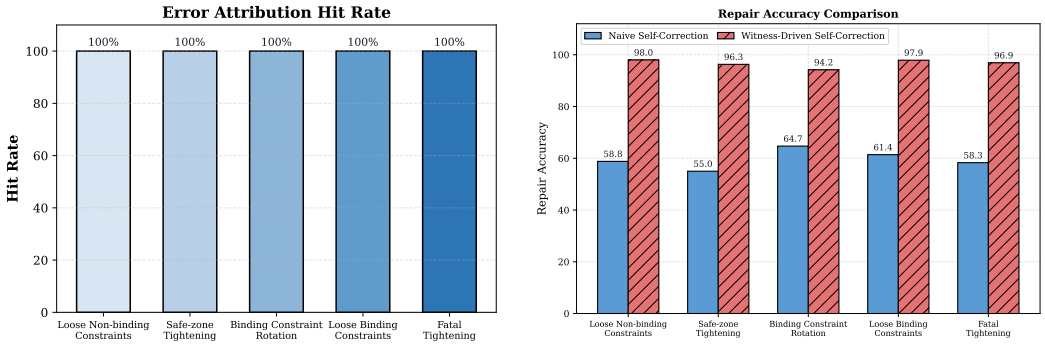

Figure 5: Left: error attribution hit rate across non-equivalent transformation types. Right: repair accuracy comparison between naive self-correction and witness-driven self-correction.

Results and interpretation. As the formulation size increases, the accuracy of Canonical Matching, WL-test, Naive-LLM, and EquivaMap gradually decreases. This trend suggests that larger formulations make structural and semantic comparison more difficult: repeated variables and constraints introduce greater representation ambiguity, while latent constraint errors become increasingly sparse relative to the full model and are therefore easier to overlook. In contrast, Opt-Judger maintains 100% accuracy over the tested range because, after variable alignment, it converts model comparison into solver-backed feasibility verification and explicitly searches for decisions that distinguish the two formulations. These results indicate that Opt-Judger is more robust to increasing formulation complexity within the evaluated range.

E.2 Solver Effort on Large-Scale Instances

Experimental design. We further evaluate solver effort on selected MIPLIB-NL instances with more than 10,000 business variables. MIPLIB-NL provides natural-language specifications and reference formulations reconstructed from industrial-scale mixed-integer programs (Li et al., 2026). We compare solving the original models, which obtains the optimal results used in solution-based evaluation, with bidirectional feasibility verification. Both procedures use Gurobi. We report the mean runtime, branch-and-bound (B&B) nodes, simplex iterations, and presolve row-removal percentage in Table 9, with verification runtime measured as the total time for both search directions.

Table 9: Comparison of solver effort on large-scale MIPLIB-NL instances.

Procedure	Runtime (s)	B&B nodes	Simplex iterations	Presolve row removal
Solving original models	1,881	184,361	6,837,566	4.67%
Bidirectional verification	59	2,350	17,901	82.11%

Results and interpretation. As shown in Table 9, bidirectional verification requires less solver effort than solving the original models on the tested instances. In practice, verification tests whether a decision permitted by one formulation is excluded by the other, rather than finding and certifying an optimal solution. After alignment, the two formulations may share substantial constraint structure. Combining the constraints of one formulation with a requirement to violate the other can expose contradictions during presolve, tighten variable bounds, and reduce branch-and-bound search. When a feasible witness is found, it establishes a mismatch without an optimality proof; when no witness exists, the shared structure may help the solver certify infeasibility. These mechanisms are consistent with the higher presolve row-removal rate and lower search counts reported in the table.

F Robustness across LLM Backbones

Experimental setup. We examine the sensitivity of Opt-Judger to its LLM backbone on EquivaFormulation and Opt-Judger-Bench. In addition to the default GPT-4o backbone, we evaluate Qwen3-32B, GPT-4o-mini, and GPT-5.1. Each backbone is evaluated over five independent test-time

inference runs. Table 10 reports the mean equivalence judgment accuracy and sample standard deviation across these runs.

Table 10: Mean judgment accuracy across five independent inference runs, with sample standard deviations in parentheses.

LLM Backbone	EquivaFormulation	Opt-Judger-Bench
Qwen3-32B	99.6 (0.12)	99.1 (0.10)
GPT-4o-mini	99.9 (0.05)	99.6 (0.12)
GPT-4o	100.0 (0.0)	100.0 (0.05)
GPT-5.1	100.0 (0.0)	100.0 (0.0)

Results and analysis. As shown in Table 10, Opt-Judger maintains high judgment accuracy across the evaluated LLM backbones, with only small reductions for Qwen3-32B and GPT-4o-mini. In Opt-Judger, the LLM aligns variables according to their modeling roles, while the solver determines whether the aligned formulations permit the same decisions. Incorrect variable alignments can therefore affect the judgment, but the subsequent feasibility verification does not rely on the LLM's ability to reason about formulation equivalence. This separation is consistent with the limited accuracy variation across backbones observed in the experiment.

G Description of Optimization Modeling Methods

G.1 Prompt-based Modeling Methods

COE (Chain of Experts) (Xiao et al., 2024) The chain of experts method is a multi-agent framework designed to handle the complexity of mathematical programming. This method primarily utilizes `GPT-4` as its core reasoning engine. It decomposes the modeling process into sequential stages assigned to specialized agents, such as a modeler and a programmer. By separating problem analysis from code implementation, it ensures that the mathematical structure is logically sound before the solver-ready code is generated.

OptiMUS (AhmadiTeshnizi et al., 2024) OptiMUS is a modular agent system that focuses on formulating and solving large-scale optimization problems. It utilizes a state-machine architecture to process long-context problem descriptions by breaking them into manageable components. A distinguishing feature of this method is its automated feedback loop, which uses error messages from external solvers to iteratively debug and refine the generated mathematical formulations and code.

G.2 Finetuned Modeling Methods

ORLM (Huang et al., 2025) ORLM is a framework focused on training open-source models specifically for operations research. It is developed by finetuning base architectures such as LLaMA-3-8B on a specialized dataset called OR-Instruct. The training process emphasizes the mapping between natural language and formal optimization structures, enabling the model to accurately identify decision variables and complex indexing patterns in industrial scenarios.

LLMOPT (Jiang et al., 2025) LLMOPT is a learning-based framework designed to improve the generalization of optimization modeling. It is typically based on a finetuned Qwen3-14B architecture. The method employs multi-instruction tuning to convert business requirements into a structured five-element formulation consisting of sets, parameters, variables, objectives, and constraints. This structured approach helps the model maintain logical consistency and reduces hallucinations when generating code for different problem types.

OptMATH (Lu et al., 2025) OptMATH is a bidirectional data synthesis framework and model series designed for high-quality optimization modeling. It utilizes the Qwen2.5-32B architecture, trained on a large-scale verified dataset. The framework ensures mathematical consistency through a back-translation and rejection sampling process, allowing the models to achieve high accuracy in translating complex word problems into precise mathematical models.

SIRL (Solver-Informed Reinforcement Learning) (Chen et al., 2025) SIRL is a reinforcement learning framework that improves the authenticity and executability of optimization models. It is typically implemented on top of base models like Qwen2.5-7B and Qwen2.5-32B. Unlike supervised training, SIRL uses external solvers as verifiers to provide direct reward signals based on the syntactic validity and solution quality of the generated code. This self-improvement loop enables the model to learn from execution feedback and generate more reliable optimization formulations.

H EXAMPLES

H.1 WITNESS POINT-BASED ERROR ATTRIBUTION EXAMPLE

Witness Point-Based Error Attribution Example

Business Problem Description. A public-health agency distributes temperature-sensitive vaccines from a central depot to a city hospital C and a rural clinic R over two delivery days. Let $x_{t,C}$ and $x_{t,R}$ denote the number of vaccine boxes delivered on day $t \in \{1, 2\}$. The depot can ship at most 120 boxes per day. The rural clinic has a small cold room and can safely receive at most 40 boxes on each day. The goal is to maximize the delivered priority-weighted vaccine volume:

$$\max \ 8x_{1,C} + 12x_{1,R} + 8x_{2,C} + 12x_{2,R}.$$

Reference Formulation.

$$\begin{aligned} \max \quad & 8x_{1,C} + 12x_{1,R} + 8x_{2,C} + 12x_{2,R} \\ \text{s.t.} \quad & x_{1,C} + x_{1,R} \le 120, \\ & x_{2,C} + x_{2,R} \le 120, \\ & x_{1,R} \le 40, \\ & x_{2,R} \le 40, \\ & x_{t,C}, x_{t,R} \ge 0, \quad t \in \{1, 2\}. \end{aligned}$$

Erroneous Generated Formulation. The generated formulation incorrectly aggregates the rural clinic's daily cold-room limit into a two-day total limit:

$$\begin{aligned} \max \quad & 8x_{1,C} + 12x_{1,R} + 8x_{2,C} + 12x_{2,R} \\ \text{s.t.} \quad & x_{1,C} + x_{1,R} \le 120, \\ & x_{2,C} + x_{2,R} \le 120, \\ & x_{1,R} + x_{2,R} \le 80, \\ & x_{t,C}, x_{t,R} \ge 0, \quad t \in \{1, 2\}. \end{aligned}$$

Witness Point. Opt-Judger finds a generated-only witness:

$$(x_{1,C}, x_{1,R}, x_{2,C}, x_{2,R}) = (40, 80, 120, 0).$$

This assignment satisfies the generated formulation:

$$40 + 80 \le 120, \qquad 120 + 0 \le 120, \qquad 80 + 0 \le 80,$$

but violates the reference cold-room constraint on day 1:

$$x_{1,R} = 80 > 40.$$

Attribution Report.

- **Mismatch Direction:** Under-constrained.
- **Violated Constraint:** The witness violates the day-1 rural cold-room capacity constraint $x_{1,R} \le 40$ by 40 boxes.
- **Potentially Violated Business Rule:** The rural clinic's cold-room capacity is a per-day operational limit, not a cumulative two-day budget. Aggregating $x_{1,R}$ and $x_{2,R}$ allows the model to overload the clinic on a single day, which may cause vaccine spoilage and invalid delivery decisions.
- **Suggested Correction Target:** Replace the aggregate constraint $x_{1,R} + x_{2,R} \le 80$ with separate daily constraints $x_{1,R} \le 40$ and $x_{2,R} \le 40$.

H.2 Framework Execution Flow Example

Framework Execution Flow Example

Business Problem Description. A factory produces two products, A and B. Let x_A and x_B denote their production quantities. Product A consumes 2 units of labor and 1 unit of material, while product B consumes 1 unit of labor and 2 units of material. The factory has at most 100 units of labor and 80 units of material. The goal is to maximize total profit:

$$\max\ 5x_A + 4x_B.$$

Reference Gurobi Model.

```
model = gp.Model("reference")
xA = model.addVar(lb=0, name="xA")
xB = model.addVar(lb=0, name="xB")
sM = model.addVar(lb=0, name="material_slack")
model.setObjective(5*xA + 4*xB, GRB.MAXIMIZE)
model.addConstr(2*xA + xB <= 100, name="labor")
model.addConstr(xA + 2*xB + sM == 80, name="material")
```

Generated Gurobi Model.

```
model = gp.Model("generated")
yA = model.addVar(lb=0, name="prod_A")
yB = model.addVar(lb=0, name="prod_B")
model.setObjective(5*yA + 4*yB, GRB.MAXIMIZE)
model.addConstr(2*yA + yB <= 100, name="labor")
model.addConstr(yA + 2*yB <= 100, name="material")
```

Step 1: Variable Mapping.

```
{
"xA": [
{
"constant": 1.0,
"variable": "prod_A"
}
],
"xB": [
{
"constant": 1.0,
"variable": "prod_B"
}
]
}
```

Step 2: Objective Comparison in the Business Space.

$$5x_A + 4x_B \equiv 5y_A + 4y_B.$$

Objective Check: Pass.

Step 3: Projected Models after Auxiliary Elimination.

```
Projecting the mapped reference model into the
business-variable space...


Eliminating auxiliary variables: material_slack


Rebuilding projected Gurobi polyhedron...

Model: reference_mapped_pure
Objective sense: Maximize
```

```
Variables:  2
Linear constraints:  4

[Objective]
Maximize 5.0 prod_A + 4.0 prod_B

[Variables]
prod_A: type=Continuous, bounds=[0, inf]
prod_B: type=Continuous, bounds=[0, inf]

[Constraints]
prod_A_lb:  -1.0 prod_A <= 0
prod_B_lb:  -1.0 prod_B <= 0
labor:  2.0 prod_A + 1.0 prod_B <= 100
material_projected:  1.0 prod_A + 2.0 prod_B <= 80



Projecting the evaluated model into the business-variable
space...


No auxiliary variables detected.


Rebuilding projected Gurobi polyhedron...

Model:  generated_mapped_pure
Objective sense:  Maximize
Variables:  2
Linear constraints:  4

[Objective]
Maximize 5.0 prod_A + 4.0 prod_B

[Variables]
prod_A: type=Continuous, bounds=[0, inf]
prod_B: type=Continuous, bounds=[0, inf]

[Constraints]
prod_A_lb:  -1.0 prod_A <= 0
prod_B_lb:  -1.0 prod_B <= 0
labor:  2.0 prod_A + 1.0 prod_B <= 100
material:  1.0 prod_A + 2.0 prod_B <= 100
```

Step 4: Constructed Witness-Search Models.

```
[*] Constructing reference-only witness search...
reference_only:  F_ref AND NOT(F_gen)
status:  infeasible

[*] Constructing generated-only witness search...
generated_only:  F_gen AND NOT(F_ref)
status:  feasible
```

Step 5: Witness Point.

[*] Reference Diagnosis : PASS No reference-only witness was found.

[*] Generated Diagnosis: ERROR TYPE = Under-Constrained

Witness point = {"prod_A": 0.0, "prod_B": 50.0}
Generated model check = PASS
material = prod_A + 2*prod_B = 100.0000 <= 100
Reference model check = FAIL
material = xA + 2*xB = 100.0000 > 80

Step 6: Attribution Report.

- **Mismatch Direction:** Under-constrained.
- **Violated Constraint:** The witness violates the material-capacity constraint

$$x_A + 2x_B \leq 80$$

by 20 units.

- **Potentially Violated Business Rule:** The factory's material capacity is a hard resource limit. The generated formulation incorrectly allows material consumption up to 100 units, while the business description states that only 80 units of material are available. This makes the generated model accept production plans that exceed the available material capacity.
- **Suggested Correction Target:** Replace the generated constraint

$$y_A + 2y_B \leq 100$$

with the correct material-capacity constraint

$$y_A + 2y_B \leq 80.$$

I Prompts for LLMs

I.1 Prompts for Variable Alignment

Prompts for Variable Alignment

```
You are an AI language model assisting in mapping variables between
    two optimization problems by analyzing their roles in constraints
    and the objective function.

Your top priority is to recover a linear mapping such that, after
    substitution, the transformed constraints from Problem 1 align
    with the constraints in Problem 2 as closely as possible.
The objective is a secondary signal used to validate or break ties
    between candidate mappings.

You are allowed to return a linear combination of multiple Problem 2
    variables.
Do not assume the mapping is one-to-one.

**Problem Context:**
{nl_description}

**Problem 1 Objective:**
{objective1['formulation']}

**Variable from Problem 1:**
- **Name:** {var_name1}
- **Description:** {var_desc1}
- **Constraints involving {var_name1}:**
  - Description: {constraint['description']}
    Formulation: {constraint['formulation']}
```

```
- **In Objective Function:** {Yes/No}

**Variables from Problem 2:**
- **Name:** {var_name2}
  **Description:** {var_desc2}
  **Constraints involving {var_name2}:**
    - Description: {constraint['description']}
      Formulation: {constraint['formulation']}
  **In Objective Function:** {Yes/No}

**Problem 2 Objective:**
{objective2['formulation']}

Based on the above information, map the variable '{var_name1}' from
    Problem 1 to variables in Problem 2.

Very important:
You must prioritize transformed-constraint alignment first.
After substituting your mapping into Problem 1, the coefficients and
    structure of the relevant constraints should align with Problem 2
    whenever possible.
Only after that should you use the objective to confirm the mapping or
     break ties between otherwise plausible candidates.
If a single Problem 2 variable does not align both the objective and
    the constraints, you must try a linear combination of multiple
    Problem 2 variables.

Priority order:
1. Agreement across multiple transformed constraints is the strongest
    signal.
2. Agreement of the transformed objective is the second strongest
    signal.
3. Variable-name similarity is weak and must never override
    coefficient alignment.

The returned mapping must satisfy this substitution direction:

    {var_name1} = sum_i constant_i * (variable_i from Problem 2)

This means the constants are chosen so that if we substitute
    {var_name1} <- sum_i constant_i * y_i
into Problem 1, then the resulting relevant constraints should match
    Problem 2 as closely as possible, and the objective should be used
     as a secondary consistency check.

Do not return the reciprocal coefficient.
Do not force a single-variable mapping if a linear combination fits
    better.

Algebra rule:
If Problem 1 contains
    a * {var_name1}
and the corresponding role in Problem 2 contains
    b1 * y1 + b2 * y2 + ...
then you must return

    {var_name1} = (b1 / a) * y1 + (b2 / a) * y2 + ...

because substituting into Problem 1 gives

    a * {var_name1} -> b1 * y1 + b2 * y2 + ...

Example:
```

```
- If Problem 1 has 100 * x and Problem 2 has 50 * y, return x = 0.5 *
    y
- If Problem 1 has 50 * x and Problem 2 has 25 * y, return x = 0.5 * y
- In the above cases, returning 2.0 would be wrong because that is the
    reciprocal
- If Problem 1 has 300 * x and Problem 2 has 300 * y2 - 300 * y1,
    return x = 1.0 * y2 + (-1.0) * y1
- If one mapping fits more transformed constraints, prefer it even if
    another mapping matches the objective slightly better
- If one candidate makes the transformed objective exactly equal to
    Problem 2 and another candidate only loosely matches names, choose
    the exact objective match
- If a candidate has a similar name but does not align the transformed
    coefficients, do not choose it
- If the objective suggests a linear combination and the constraints
    also support it, return the linear combination

Self-check before answering:
1. Check whether '{var_name1}' appears in the objective of Problem 1.
2. First try a single-variable mapping from Problem 2.
3. Substitute the mapping back into the Problem 1 constraints
    involving '{var_name1}' and check whether the transformed
    coefficients align Problem 2.
4. If that does not align the constraints well enough, try a two-
    variable linear combination.
5. Compare candidate mappings by how well they align the transformed
    constraints.
6. Use the objective only as a secondary validation or tie-breaker.
7. If the evidence is inconsistent or weak, return "none" instead of
    guessing.

Output requirements:
- Return only JSON
- Do not include any explanation
- Do not default to 1.0
- Do not return the reciprocal coefficient
- It is valid to return more than one item in the list
- Prefer the mapping that best aligns the transformed constraints; use
    the objective as secondary validation
- If objective alignment and transformed-constraint alignment disagree
    strongly, return "none" instead of forcing a bad mapping

Required format:
{
  "{var_name1}": [
    {
      "constant": constant_value_1,
      "variable": "variable_name_1"
    }
  ]
}

For a linear combination, return multiple items in the list, for
    example:
{
  "{var_name1}": [
    {
      "constant": 1.0,
      "variable": "y2"
    },
    {
      "constant": -1.0,
      "variable": "y1"
```

```
    }
  ]
}

If no reliable mapping exists, return:
{
  "{var_name1}": [
    {
      "constant": "none",
      "variable": "none"
    }
  ]
}
```

I.2 Prompts for Error Attribution

Prompts for Error Attribution

```
ATTRIBUTION_PROMPT = """
You are an expert-level specialist in Operations Research and
optimization-model diagnosis.
Do not decide equivalence by yourself.  The equivalence judgment has
already been made by the solver.
Use only the provided witness point, violated constraint, mismatch
direction, and related business description.

Your report must follow four fields:
1.  Mismatch Direction
2.  Violated Constraint
3.  Potentially Violated Business Rule
4.  Suggested Correction Target

Input:
1.  BUSINESS PROBLEM: {business_problem}
2.  ERROR CONSTRAINTS: {error_constraints}
3.  WITNESS POINT: {witness_point}
4.  GENERATED MODEL: {generated_model}

Return only the final attribution report in the following JSON format:
{
"mismatch_direction":  "<under-constrained or over-constrained>",
"violated_constraint":  "<constraint violated by the witness point>",
"potentially_violated_business_rule":  "<business rule corresponding
to the violated constraint>",
"suggested_correction_target":  "<concrete constraint or formulation
fragment to repair>"
}

Please make sure the output is valid JSON and can be parsed by
json.loads().
Do not include explanations outside the JSON object.
"""
```