CONSTRAINT COVERAGE GUIDED HIDDEN ERRORS EXPOSURE FOR AUTOMATED OPTIMIZATION MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimization modeling translates real-world requirements into mathematical models and executable solver code, and large language models (LLMs) have shown promise in automating this process from natural-language descriptions. However, existing evaluations typically test a generated optimization model only on the original numerical instance and judge correctness by its optimal objective value. From a software-testing perspective, this resembles executing a program on a single input and checking only its final output: constraints irrelevant to the original optimum may remain untested. We introduce a constraint-coverage-guided validation paradigm for automated linear programming modeling, which assesses whether each target inequality constraint contributes to establishing optimality in at least one test instance. Inspired by coverage-guided software testing and inverse optimization, our framework treats constraints as test targets and searches prescribed parameter domains for problem variants that exercise previously uncovered constraints. It then reconstructs the generated variants as complete natural-language problems, producing an augmented test suite of paired problem descriptions and reference models. Experiments on public optimization-modeling benchmarks show that the augmented test suites substantially increase constraint coverage and expose hidden constraint-level modeling errors in optimization models produced by automated optimization modeling systems, even when these formulations yield correct objective values on the original instances. These results demonstrate that constraint coverage provides a more diagnostic complement to conventional answer-based evaluation.

1 INTRODUCTION

Optimization modeling translates real-world decision requirements into mathematical formulations that can be solved by optimization solvers. Traditionally, this process requires substantial domain knowledge and modeling expertise, creating a significant barrier to the broader adoption of optimization-based decision making. Recent advances in large language models (LLMs) have motivated growing interest in automated optimization modeling, where natural-language problem descriptions are converted into mathematical formulations and executable solver code Xiao et al. (2025). Early studies explored structured auto-formulation pipelines that translate natural-language problem descriptions into formal optimization representations Ramamonjison et al. (2022a;b). Existing approaches can be broadly grouped into two paradigms: prompt-based systems guide general-purpose LLMs through task decomposition or structured multi-agent collaboration Ahmaditeshnizi et al. (2024); Xiao et al. (2024); Yang et al. (2026), whereas fine-tuning-based methods adapt LLMs using optimization-specific training data and modeling-oriented supervision Jiang et al. (2025); Huang et al. (2025a); Chen et al. (2026).

As these systems increasingly produce executable optimization models, evaluating the reliability of their outputs has become essential. Existing systems and benchmarks commonly execute the generated solver code and compare the resulting solution or optimal objective value with a reference answer for the original problem instance Ahmaditeshnizi et al. (2024); Jiang et al. (2025); Huang et al. (2025a); Yang et al. (2026). In this paper, we refer to this evaluation approach as single-instance answer-based evaluation. However, in software testing, test adequacy is typically assessed

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

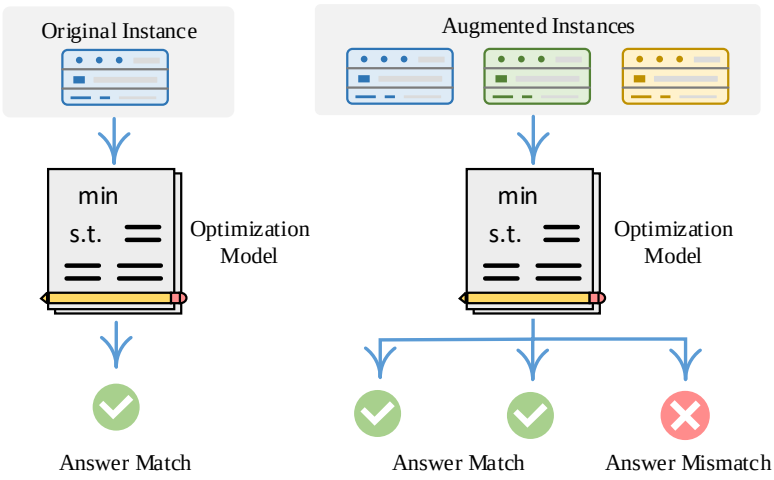


Figure 1: Comparison between single-instance answer-based evaluation and the proposed constraint-coverage-guided evaluation. The former checks a generated model only on the original numerical instance, whereas the latter evaluates it on augmented instances designed to evaluate constraints.

using a suite of test cases and coverage criteria that measure whether relevant program structures or behaviors have been exercised Ammann & Offutt (2016); Manès et al. (2021). Coverage-guided testing goes a step further by using coverage feedback to generate new inputs that exercise previously uncovered program behaviors Böhme et al. (2019). From this perspective, evaluating an optimization model only on its original instance is analogous to testing a program with a single input: it observes the model under only one parameter setting. A constraint that is inactive or does not have an independent effect on the optimum under this setting may be omitted or misformulated without changing the reported answer. Figure 1 illustrates the difference between single-instance execution-based evaluation and our coverage-guided validation setting: the former checks the optimization model only on the original instance, whereas the latter evaluates the same modeling result on augmented instances designed to exercise additional constraints.

Motivated by this testing perspective, we instantiate the constraint-coverage-guided validation paradigm through a test-suite augmentation framework for automated optimization modeling. The framework treats each target constraint in an optimization model as a test target and regards it as covered when it independently contributes to establishing optimality in at least one test instance. Drawing on the reverse-design principle of inverse optimization Ahuja & Orlin (2001); Chan et al. (2025), the framework generates additional test instances by modifying objective and constraint parameters within their admissible domains, such that constraints not exercised by the original instance can influence the optimum. The generated model parameters are then mapped back to their corresponding problem parameters and reconstructed as complete natural-language problem descriptions. Together with the original problems, these generated variants form an augmented test suite that evaluates automated modeling systems across multiple parameter settings, increasing the opportunity to expose constraint errors hidden by single-instance answer-based evaluation.

We evaluate the proposed framework on five public optimization-modeling benchmarks, focusing on their linear programs. Across all the benchmarks, the generated instances improve constraint coverage over the original test suites. The augmented test suites further expose several undetected constraint-level modeling errors in generated optimization models that return correct answers on the corresponding original instances. These cases show that a correct answer on the original instance does not guarantee that every constraint has been modeled correctly, and that constraint coverage provides information beyond single-instance answer-based evaluation. Our main contributions are summarized as follows:

- We introduce the constraint-coverage-guided validation paradigm for automated optimization modeling. It extends single-instance answer-based evaluation by testing whether each target constraint contributes to establishing optimality in at least one test instance.

- We develop an automated test-suite augmentation framework for LPs that draws on coverage-guided testing and inverse optimization. The framework generates instances targeting uncovered constraints and improves constraint coverage on existing optimization-modeling datasets.
- We use the augmented test suites to evaluate automated modeling methods and expose hidden constraint-level modeling errors. The results show that optimization models can return correct answers on original instances while still containing constraint modeling errors.

2 Related Work

2.1 Linear Programming

This work focuses on finite-dimensional linear programming with continuous decision variables. Following the standard linear-optimization notation Bertsimas & Tsitsiklis (1997), linear programming takes the form:

$$P: \quad \min_{x \in \mathbb{R}^n} \quad c^\top x \qquad \text{s.t.} \quad Ax \le b, \quad Ex = d, \quad \ell \le x \le u, \tag{1}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $E \in \mathbb{R}^{q \times n}$, and $d \in \mathbb{R}^q$. The vectors ℓ and u give the lower and upper bounds on x, respectively. Maximization problems can be written in the same form by negating c. The feasible region of P is X. A point $x \in \mathcal{X}$ is feasible, and $x^\star \in \mathcal{X}$ is optimal if $c^\top x^\star \le c^\top x$ for every $x \in \mathcal{X}$. The corresponding optimal objective value is denoted by $z^\star = c^\top x^\star$. An inequality $a_i^\top x \le b_i$ is active at x if $a_i^\top x = b_i$, and inactive otherwise.

Throughout this work, all considered instances are assumed to be feasible and to attain a finite optimum. Integer restrictions and nonlinear objectives or constraints fall outside the scope of this study.

2.2 Verification of Optimization Models

Most automated optimization modeling methods evaluate a generated optimization model by executing it with an optimization solver and comparing the returned answer with the reference answer Ahmaditeshnizi et al. (2024); Jiang et al. (2025); Huang et al. (2025a); Yang et al. (2026). Such evaluations are commonly conducted on optimization modeling benchmarks, including MAMO, NLP4LP, IndustryOR, OptiBench, and LogiOR Huang et al. (2025b); Ahmaditeshnizi et al. (2024); Huang et al. (2025a); Yang et al. (2025; 2026). For most of these benchmarks, the primary ground-truth signal available for automatic evaluation is the final numerical answer, typically the optimal objective value, rather than a complete reference formulation that supports constraint-level verification. A generated model is therefore considered correct if it executes successfully and reproduces the reference answer on the original instance. Although this answer-based protocol naturally accommodates alternative but equivalent formulations, it may overlook constraint omissions or misformulations that do not change the answer under the original parameter setting.

Recent work has begun to verify generated formulations beyond comparing their optimal objective values on the original instance. One line of work checks whether two formulations are equivalent as complete optimization models. EquivaMap Zhai et al. (2025) uses LLMs to discover mappings between the decision-variable assignments of two formulations and verifies that mapped solutions preserve feasibility and optimality. ORGEval Wang et al. (2025) represents MILP formulations as weighted bipartite graphs and applies a Weisfeiler–Lehman-based isomorphism test with correctness guarantees for symmetric-decomposable instances. Another line of work evaluates formulations behaviorally by generating problem-level tests or model-specific mutations Zadorojniy et al. (2025). These studies provide stronger evidence than comparing a single optimal objective value. However, they do not explicitly focus on which reference constraints have been exercised by the test instances. Our work is complementary: it treats each target inequality constraint as a coverage target and generates admissible LP instances in which previously uncovered constraints participate in establishing optimality.

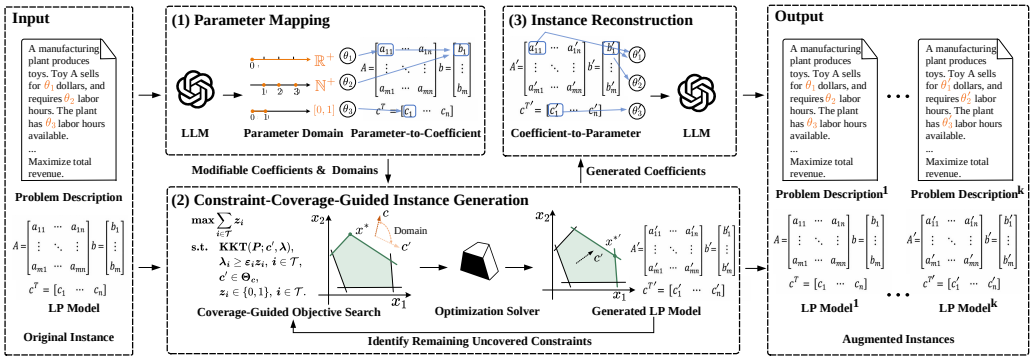


Figure 2: Overview of the proposed constraint-coverage-guided test-suite augmentation framework. Given an original natural-language problem and its reference LP, (1) Parameter Mapping identifies problem parameters, their admissible domains, and their correspondences with modifiable formulation coefficients; (2) Constraint-Coverage-Guided Instance Generation produces new LP instances targeting previously uncovered constraints; and (3) Instance Reconstruction maps the generated formulation coefficients back to problem parameters and reconstructs the corresponding problem context. The problem–LP pairs together form the augmented test suite.

3 METHOD

3.1 FRAMEWORK OVERVIEW

As shown in Figure 2, the framework consists of three stages. *Parameter Mapping* and *Instance Reconstruction* use LLM assistance to establish parameter-to-coefficient mappings and to reconstruct generated parameter settings as complete natural-language problems, respectively. At the core, *Constraint-Coverage-Guided Instance Generation* uses an optimization model to generate instances targeting uncovered constraints. The remainder of this section follows the three stages in Figure 2: parameter mapping, constraint-coverage-guided instance generation, and instance reconstruction.

3.2 PARAMETER MAPPING

The first stage establishes parameter-to-coefficient mappings between problem parameters and modifiable formulation coefficients. These mappings are necessary because arbitrary changes to formulation coefficients may not correspond to meaningful variants of the problem. For example, changing an objective coefficient may imply a negative unit cost, while changing a right-hand side may imply a fractional number of vehicles or workers. Such variants are mathematically valid LP instances but invalid problem instances. To avoid this mismatch, we execute the formulation code under dynamic data-flow tracking Schwartz et al. (2010). During execution, the provenance of problem parameters is propagated through arithmetic operations and modeling expressions, allowing us to record which problem parameters affect each formulation coefficient. Besides, we prompt an LLM to analyze the problem description to infer the semantic type and admissible domain of each problem parameter, such as positive or nonnegative quantities, positive integers, and proportions restricted to $[0, 1]$. The inferred domains collectively define the admissible domain Θ for problem parameters, which is used by constraint-coverage-guided instance generation. A worked example of dynamic data-flow tracing and the prompt used for semantic domain inference are provided in supplementary material.

3.3 CONSTRAINT-COVERAGE-GUIDED INSTANCE GENERATION

Constraint coverage. Given a reference LP model, let $\mathcal{I} = \{1, \ldots, m\}$ index the linear inequality constraints represented by the rows of $Ax \leq b$. We refer to these inequalities as *target constraints* and treat each $i \in \mathcal{I}$ as a coverage target. Equality constraints and variable bounds are excluded from $\mathcal{I}$: equality constraints are active at every feasible point Boyd & Vandenberghe (2004), while variable bounds are treated as specifications of the variable domains rather than as coverage targets.

Algorithm 1 Constraint-Coverage-Guided Generation

Require: Parametric LP model $P(\theta)$, original parameter setting θ^0, and admissible parameter domain Θ
Ensure: Generated test suite $\mathcal{S}$ and covered constraint set $\mathcal{C}$

```
Let I be the indices of explicit inequalities                    ▷ Coverage targets
C, S ← ∅; T^0 ← I
P^0 ← P(θ^0); k ← 0                                              ▷ Initialize with the original instance
repeat
    Solve P^k to obtain A_I^k
    Construct W^k and D^k                                        ▷ Partition active targets
    C ← C ∪ W^k
    S ← S ∪ {P^k}
    for all a ∈ D^k ∩ T^k do
        Perturb RHS/LHS to get P^{k,a}                           ▷ R/LHS search
        P^{k,a} ← CGOS(P^{k,a}, {a})                             ▷ Objective search
        if a coverage witness (P^{k,a}, W^{k,a}) is found then
            C ← C ∪ W^{k,a}
            S ← S ∪ {P^{k,a}}
        end if
    end for
    T^{k+1} ← T^k \ (A_I^k ∪ C)
    P^{k+1} ← CGOS(P(θ^0), T^{k+1})                              ▷ Objective search
    k ← k + 1
until P^k = ∅
return (S, C)
```

For an LP instance P with an optimal solution $x^\star$, we first identify the target constraints that are tight at the optimum:

$$\mathcal{A}(x^\star) = \left\{ i \;\middle|\; a_i^\top x^\star = b_i \right\}. \tag{2}$$

Although constraints in $\mathcal{A}(x^\star)$ are tight at the optimum, tightness alone does not imply coverage. In a degenerate LP solution, multiple active constraints may describe the same local face or normal space, so some active constraints can be linearly dependent on others and do not provide an independent optimality certificate Bertsimas & Tsitsiklis (1997). Under a given problem-parameter setting, target constraints fall into the following cases:

- *Covered targets* $\mathcal{W}(x^\star)$: active target constraints that enter the target-prioritized linearly independent working set.
- *Dependent active targets* $\mathcal{D}(x^\star)$: active target constraints that are not included in $\mathcal{W}(x^\star)$, i.e., $\mathcal{D}(x^\star) = \mathcal{A}(x^\star) \setminus \mathcal{W}(x^\star)$.
- *Inactive targets* $\mathcal{R}(x^\star)$: target constraints that are not tight at $x^\star$ but may define the feasible region away from the current optimum.
- *Globally redundant targets* $\mathcal{G}$: target constraints that are implied by the remaining constraints under the current parameter setting and therefore do not change the feasible region.

We count a target constraint as covered by P only if it belongs to $\mathcal{W}(x^\star)$. For a test suite $\mathcal{S}$, constraint coverage is the fraction of target constraints that are covered by at least one instance in $\mathcal{S}$.

Generation procedure. Using the parameter mapping from the previous subsection, we generate test instances in the parameterized LP space. Let $P(\theta)$ denote the reference LP, where θ^0 is the original parameter setting and Θ is the admissible domain inferred for modifiable problem parameters. Covering a target constraint may require controlling both the objective vector and the geometry of the feasible polyhedron. The objective vector exposes an optimal face, whereas the constraint parameters shape the feasible polyhedron and hence determine which target constraints can be active and linearly independent at an optimal solution.

Algorithm 1 summarizes our generation procedure. The original LP constitutes the initial instance. In the outer iteration, *Coverage-Guided Objective Search* (CGOS) changes only the objective coef-

ficients of $P(\theta^0)$ to explore an alternative optimal working set while preserving the feasible region. Solving P^k yields the target-prioritized working set $\mathcal{W}^k$, while the remaining active targets form $\mathcal{D}^k$. For each unprocessed target $a \in \mathcal{D}^k \cap \mathcal{T}^k$, the inner search perturbs its admissible RHS/LHS parameters from θ^0 and applies CGOS again to determine an objective for the perturbed LP. A target is added to the covered set only when a certified witness places it in the target-prioritized linearly independent working set. The procedure continues until no further admissible coverage witness is found.

For a candidate LP P and a target set $\mathcal{T}$, CGOS keeps the feasible region fixed and searches for an admissible objective vector c' under which as many targets in $\mathcal{T}$ as possible receive positive dual multipliers in an optimality certificate. We formulate CGOS as

$$\begin{aligned} \mathrm{CGOS}(P,\mathcal{T}): \quad \max \quad & \sum_{i\in\mathcal{T}} z_i \\ \text{s.t.} \quad & \mathrm{KKT}(P;c',\lambda), \quad \lambda_i \geq \varepsilon_i z_i \ (i\in\mathcal{T}), \quad c' \in \Theta_c, \quad z_i \in \{0,1\} \ (i\in\mathcal{T}). \end{aligned} \tag{3}$$

Here, Θ_c is the admissible domain of the objective vector, and λ_i denotes the dual multiplier of target inequality i. The shorthand $\mathrm{KKT}(P;c',\lambda)$ denotes the existence of a primal–dual tuple satisfying the KKT conditions of P under objective c', with λ collecting the multipliers of the target inequalities Boyd & Vandenberghe (2004). The binary variable z_i indicates whether target i is selected by CGOS. The constraint $\lambda_i \geq \varepsilon_i z_i$ requires a selected target to have a strictly positive dual multiplier, where $\varepsilon_i > 0$ is a small numerical threshold. Thus, CGOS favors objectives whose optimality certificates involve more target constraints. Solving this model yields a modified objective vector together with the corresponding optimal objective value and optimal solution.

3.4 Instance Reconstruction

Writing generated parameter settings back into natural-language problem descriptions is not a direct numerical substitution process, because a formulation coefficient may depend on a combination of multiple problem parameters. Given the generated formulation coefficients and the parameter-to-coefficient mappings, we construct a dependency graph over the affected problem parameters and prompt an LLM to update them in topological order. The LLM modifies only the affected numerical values in the original problem description. The problem context and optimization task are preserved. Each reconstructed problem is paired with its generated optimization model and optimal answer to form a complete test instance. The reconstruction prompt and implementation details are provided in supplementary material.

4 Experiments

To analyze the performance of the proposed constraint-coverage-guided framework, our experiments aim to answer the following three questions:

RQ1: How much does the proposed generation procedure improve constraint coverage over the original instances?

RQ2: What hidden constraint-level modeling errors can the resulting augmented test suites expose beyond conventional answer-based evaluation?

RQ3: How does the proposed generation procedure affect test-suite generation efficiency?

4.1 Experimental Setup

Datasets. We evaluate our framework on five public optimization-modeling benchmarks: NLP4LP Ahmaditeshnizi et al. (2024), IndustryOR Huang et al. (2025a), LogiOR Yang et al. (2026), ReSocratic Yang et al. (2025), and MAMO Huang et al. (2025b). Since our method targets linear programs, we filter each benchmark to retain instances whose reference formulations are linear programs with explicit inequality constraints. This yields 48 instances from NLP4LP, 17 from IndustryOR, 21 from LogiOR, 70 from MAMO (including both EasyLP and ComplexLP), and 1,844 from ReSocratic.

Evaluated modeling methods. We evaluate automated optimization modeling methods from three categories:

- **Zero-shot LLMs.** We use GPT-5-mini OpenAI (2025) and DeepSeek-V4-Pro DeepSeek-AI (2026) as direct zero-shot baselines. These models receive the original problem description and are prompted to produce an executable optimization formulation code.
- **Prompt-based methods.** We evaluate OptiMUS-v0.3 AhmadiTeshnizi et al. (2024) and ORThought Yang et al. (2026). OptiMUS-v0.3 is a modular LLM-based system for modeling. It uses a modular decomposition, including a connection graph, to process objectives and constraints, generate and debug solver code, and refine outputs using execution feedback. ORThought uses a structured dual-agent reasoning framework that incorporates expert-level modeling principles through chain-of-thought reasoning, aiming to reduce redundant autonomous-agent interactions while improving complex-constraint handling.
- **Fine-tuned methods.** We evaluate ORLM Huang et al. (2025a) and SIRL Chen et al. (2026). ORLM trains open-source LLMs for optimization modeling and solver-code generation using the semi-automatically synthesized OR-Instruct data. SIRL trains LLMs through solver-informed reinforcement learning, where external optimization solvers provide verifiable rewards by checking the generated code and the associated LP representation for syntax, feasibility, and solution quality.

Evaluation metrics. For coverage evaluation, we report *constraint coverage*. For a test suite $\mathcal{S}$ and the set of target inequality constraints $\mathcal{I}$, let $\mathcal{C}(\mathcal{S}) \subseteq \mathcal{I}$ denote the target constraints covered by at least one instance in $\mathcal{S}$, according to the coverage criterion. Constraint coverage is computed as

$$\text{constraint coverage}(\mathcal{S}) = |\mathcal{C}(\mathcal{S})|/|\mathcal{I}|.$$

At the dataset level, $\mathcal{I}$ is the union of target inequality constraints across all retained problems in a dataset, and $\mathcal{C}(\mathcal{S})$ is the subset covered by the corresponding test suite.

For hidden-error exposure, we report the *hidden constraint error count*. For a modeling method m, let $\mathcal{O}_m$ be the set of generated formulations, and let $\mathcal{A}_m \subseteq \mathcal{O}_m$ denote the formulations that return the correct answer on the original instances. We evaluate each formulation in $\mathcal{A}_m$ on the generated variants in the corresponding augmented test suite. A formulation is counted as exposing a hidden constraint error if it is correct on the original instance but fails on at least one generated variant due to a constraint omission or misformulation. We report this result as h/a, where $a = |\mathcal{A}_m|$ is the number of original-answer-correct formulations and h is the number of such formulations exposed by the augmented test suite.

In the ablation study, we report the generated test-suite size required to match the same constraint coverage. This metric counts the number of saved generated instances needed to reach the coverage achieved by our method. Since the ablated method may not reach this coverage within the time limit, averages are computed only on the reached problems, using the same problem subset for both methods. Under the same coverage requirement, a smaller generated test suite indicates more efficient instance generation.

Implementation Details. In our framework, GPT-5-mini OpenAI (2025) is used in Step (1) for semantic parameter-domain inference and in Step (3) for natural-language instance reconstruction. Gurobi is used in Step (2) as the optimization solver for the coverage-guided optimization subproblems. In the constraint-coverage experiment, we run the proposed framework to generate the augmented test suites and compare their constraint coverage with that of the original instances. In the hidden-error experiment, we evaluate each automated modeling method under the configuration reported in its original paper. We focus on formulations that are correct on the original instances. For these formulations, we automatically substitute the numerical parameters from the augmented test suites into the same generated modeling code, solve the instantiated candidate models, and compare their optimal objective values with the corresponding reference answers. Only failures caused by constraint errors are counted as hidden constraint errors. Additional implementation details are provided in supplementary material.

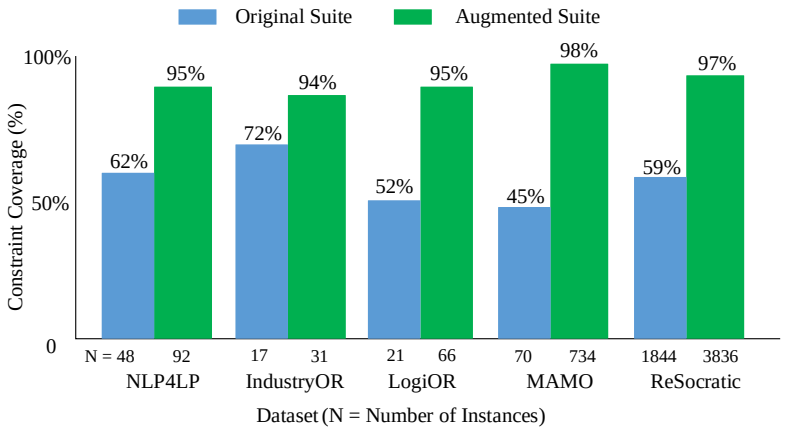


Figure 3: Comparison of constraint coverage between the original and augmented test suites across five datasets. Blue bars show the coverage achieved by the original test suite, while green bars show the coverage achieved after augmenting the original instances with generated instances. The percentages above the bars indicate constraint coverage. The value of N below each bar denotes the number of instances in the corresponding suite.

Table 1: Hidden constraint errors exposed by the augmented test suites. Each entry reports h/a, where a is the number of formulations that return the correct answer on the original instance, and h is the number of such formulations that fail on augmented instances due to formulation errors. We abbreviate IndustryOR as IndOR and ReSocratic as ReSoc.

Method	NLP4LP	IndOR	LogiOR	MAMO	ReSoc
Zero-shot LLMs					
GPT-5-mini	0/43	0/13	1/17	1/51	0/1510
DeepSeek-V4	0/30	0/11	0/14	1/56	2/1474
Prompt-based Methods					
OptiMUS-v0.3	4/36	0/10	1/17	0/44	11/1502
ORThought	1/39	0/8	1/10	3/35	15/1641
Fine-tuned Methods					
ORLM	1/33	0/5	0/5	1/14	8/1338
SIRL	1/42	0/8	0/4	1/29	6/1562

4.2 Results

Constraint coverage comparison. Figure 3 compares the coverage achieved by the original test suites and the augmented test suites. The original instances provide only partial constraint coverage, ranging from 45% to 72% across the five datasets. After augmentation, all datasets exceed 90% coverage, with the best result reaching 98%. The gains are consistent across datasets of different scales. Smaller datasets such as NLP4LP and IndustryOR show substantial improvements despite having relatively few original instances, while larger datasets such as ReSocratic also benefit strongly from coverage-guided augmentation. Datasets with more constraints per problem, such as LogiOR and MAMO, also reach high coverage after augmentation. This suggests that the generation procedure remains effective not only for datasets containing many instances, but also for problems containing more coverage targets. Overall, the augmented test suites provide broader constraint-level evidence than original instances alone.

Hidden constraint error exposure. Table 1 reports the hidden errors exposed by the augmented test suites across datasets and modeling methods. Each cell counts original-answer-correct formulations that fail on generated variant because of a constraint-level modeling error. Thus, these are not ordinary answer mismatches, but cases that conventional answer-based evaluation would regard as correct. The exposed errors appear across zero-shot LLMs, prompt-based methods, and fine-tuned methods, suggesting that this failure mode is not tied to a single implementation style. Prompt re-

Table 2: Ablation of the constraint-coverage-guided (CCG) instance generation module. Each entry reports the average number of generated instances required to match the same constraint coverage as our method. The numbers in parentheses under each dataset name indicate how many problems the ablated variant reaches within the budget; averages for both methods are computed on these problems.

Method	NLP4LP (46/48)	IndOR (16/17)	LogiOR (20/21)	ReSoc (1777/1844)	MAMO (46/70)
Ours	1.87	1.69	3.20	2.06	6.46
w/o CCG	1.96	1.98	3.56	2.13	7.36

finement or fine-tuning can improve performance on the original numerical instance, but it does not necessarily ensure that every relevant constraint is faithfully modeled. The reason is that an error constraint may be inactive, or active but linearly dependent, under the original parameter setting, and therefore may not affect the optimal objective value. Our test suite turns such hidden constraint mistakes into observable failures by making the corresponding constraints participate in establishing optimality. These results demonstrate the diagnostic value of augmented test suites as a complement to conventional answer-based evaluation.

4.3 Ablation Study

To evaluate the contribution of the proposed constraint-coverage-guided instance generation module, we replace it with a waypoint-guided random mutation strategy Padhye et al. (2019), a common strategy in coverage-guided fuzzing. Both variants use the same admissible parameter domains, constraint-coverage criterion, and per-problem budget. Implementation details are provided in the supplementary material. Among the problems for which the ablated variant reaches the same constraint coverage as our method, Table 2 shows that our method requires fewer generated instances across all datasets. The reached-problem counts further show that random mutation fails to match this coverage within the budget on a subset of problems. The difference is most pronounced on MAMO, which contains problems with relatively many constraints. This result is consistent with random mutation requiring greater search effort to exercise the remaining coverage targets within the fixed budget. Overall, the proposed generation module reduces the required test-suite size while enabling more problems to reach the target coverage.

5 Conclusion

We introduced constraint coverage as a diagnostic criterion for automated optimization modeling, extending evaluation beyond the optimal answer of a single numerical instance. Experiments on public optimization-modeling datasets show that the augmented test suites substantially increase constraint coverage and expose hidden constraint-level modeling errors in models judged correct on the original instances. Ablation results further support the efficiency of the proposed instance generation procedure. These findings demonstrate the value of constraint coverage as a practical complement to conventional answer-based evaluation. The generated problem variants also expand existing optimization-modeling datasets, providing broader test suites for evaluating formulation correctness.

As a test-based criterion, constraint coverage remains diagnostic and does not constitute a formal proof of formulation correctness. Besides, the current study focuses on linear programming, future work will extend the approach to mixed-integer and nonlinear programs, it will require coverage criteria and generation mechanisms that account for integrality and nonlinear optimality conditions.

AI USE STATEMENT

We used generative AI tools in two roles. First, large language models, including GPT-5-mini and DeepSeek-V4-Pro, were used as explicit components of the experimental framework and as evaluated automated optimization-modeling systems. Their roles in semantic parameter-domain inference, natural-language instance reconstruction, and automated optimization modeling are described in the main paper and supplementary material. The generated problem variants are part of the proposed experimental protocol and were validated by the solver-based procedures described in the paper. Second, we used generative AI tools during manuscript preparation to assist with language editing, structural organization, and LaTeX formatting. The authors reviewed and verified the final mathematical formulations, algorithm descriptions, citations, experimental settings, numerical results, tables, figures, and claims. Generative AI tools were not used to fabricate experimental results, citations, or scientific claims, and no AI-generated figure was included without author review. The authors take full responsibility for the final content of this submission.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, Herman Brunborg, Shayan Talaei, Connor Lawless, and Madeleine Udell. OptiMUS-0.3: Using large language models to model and solve optimization problems at scale, 2024.

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 of *Proceedings of Machine Learning Research*, pp. 577–596. PMLR, 2024. URL https://proceedings.mlr.press/v235/ahmaditeshnizi24a.html.

Ravindra K. Ahuja and James B. Orlin. Inverse optimization. *Operations Research*, 49(5):771–783, 2001. doi: 10.1287/opre.49.5.771.10607. URL https://doi.org/10.1287/opre.49.5.771.10607.

Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2 edition, 2016. doi: 10.1017/9781316771273.

Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019. doi: 10.1109/TSE.2017.2785841.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

Timothy C. Y. Chan, Rafid Mahmood, and Ian Yihang Zhu. Inverse optimization: Theory and applications. *Operations Research*, 73(2):1046–1074, 2025. doi: 10.1287/opre.2022.0382. URL https://doi.org/10.1287/opre.2022.0382.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed rl: Grounding large language models for authentic optimization modeling. *Advances in Neural Information Processing Systems*, 38:106027–106069, 2026.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

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, 2025a. doi: 10.1287/opre.2024.1233. URL https://doi.org/10.1287/opre.2024.1233.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. LLMs for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 2678–2710. Association for

Computational Linguistics, 4 2025b. doi: 10.18653/v1/2025.findings-naacl.146. URL https://aclanthology.org/2025.findings-naacl.146/.

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 *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=9OMvtboTJg.

Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

OpenAI. GPT-5 mini Model. OpenAI API documentation, 2025. URL https://developers.openai.com/api/docs/models/gpt-5-mini. Accessed: 2026-07-22.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

Rindra Ramamonjison, Haley Li, Timothy Yu, Shiqi He, Vishnu Rengan, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. Augmenting operations research with auto-formulation of optimization models from problem descriptions. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 29–62, 2022a.

Rindranirina 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 the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pp. 189–203. PMLR, 2022b. URL https://proceedings.mlr.press/v220/ramamonjison23a.html.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pp. 317–331, 2010. doi: 10.1109/SP.2010.26.

Zhuohan Wang, Ziwei Zhu, Ziniu Li, Congliang Chen, Yizhou Han, Yufeng Lin, Zhihang Lin, Angyang Gu, Xinglin Hu, Ruoyu Sun, and Tian Ding. ORGEval: Graph-theoretic evaluation of LLMs in optimization modeling, 2025. URL https://arxiv.org/abs/2510.27610. arXiv preprint arXiv:2510.27610.

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.

Ziyang Xiao, Jingrong Xie, Lilin Xu, Shisi Guan, Jingyan Zhu, Xiongwei Han, Xiaojin Fu, WingYin Yu, Han Wu, Wei Shi, Qingcan Kang, Jiahui Duan, Tao Zhong, Mingxuan Yuan, Jia Zeng, Yuan Wang, Gang Chen, and Dongxiang Zhang. A survey of optimization modeling meets llms: Progress and future directions. In James Kwok (ed.), *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pp. 10742–10750. International Joint Conferences on Artificial Intelligence Organization, 8 2025. doi: 10.24963/ijcai.2025/1192. URL https://doi.org/10.24963/ijcai.2025/1192. Survey Track.

Beinuo Yang, Qishen Zhou, Junyi Li, Chenxing Su, Panagiotis Angeloudis, and Simon Hu. OR-Thought: Benchmarking and automating logistics optimization modeling via structured LLM reasoning. *Artificial Intelligence for Transportation*, 6:100059, 2026. doi: 10.1016/j.ait.2026.100059. URL https://doi.org/10.1016/j.ait.2026.100059.

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 *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=fsDZwS49uY.

Alexander Zadorojniy, Segev Wasserkrug, and Eitan Farchi. An agent-based framework for the automatic validation of mathematical optimization models, 2025. URL https://arxiv.org/abs/2511.16383. arXiv preprint arXiv:2511.16383.

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 of *Proceedings of Machine Learning Research*, pp. 74288–74305. PMLR, 2025. URL https://proceedings.mlr.press/v267/zhai25a.html.

A Overview of Supplementary Material

This appendix provides supplementary material for our work. It is organized into three main sections, summarized below.

- **Methodological and Implementation Details (Appendix B).** Appendix B.1 presents a worked example of Dynamic Data-Flow Tracing and provides the prompt used for LLM-assisted semantic-domain inference of source parameters. Appendix B.2 gives a detailed formulation of the Coverage-Guided Objective Search (CGOS) model. Appendix B.3 provides pseudocode describing the complete Constraint-Coverage-Guided Generation Procedure. Appendix B.4 provides the prompt used by the LLM to reconstruct generated source-parameter updates as complete natural-language problem instances.
- **Experimental Details and Additional Results (Appendix C).** Appendix C.1 describes the procedure used to identify continuous-variable LP instances from the five evaluated benchmarks. Appendix C.2 reports the evaluated modeling methods, model checkpoints, inference configurations, and the prompt used for the zero-shot baselines. Appendix C.3 provides the implementation details and additional results of the waypoint-guided random mutation ablation, including results from repeated runs with different random seeds.
- **Detailed Case Study (Appendix D).** Appendix D.1 provides the complete problem description and network data of the representative LogiOR instance. Appendix D.2 compares the correct shared arc-capacity constraint with the erroneous per-product capacity bounds and shows their corresponding implementations. Appendix D.3 reports the generated transportation-cost modifications and compares the resulting solutions and objective values, demonstrating how the augmented instance exposes the hidden constraint error.

B Methodological and Implementation Details

B.1 Parameter Mapping

B.1.1 Dynamic Data-Flow Tracing

The main paper describes the role of parameter mapping in the overall framework. Here, we provide the implementation representation and a worked trace. The reference formulation code is executed with each numeric source parameter wrapped as a traced scalar. A lightweight symbolic model records the resulting LP coefficients without invoking the optimization solver.

Consider the following simplified formulation fragment:

```
def solve(capacity=70, cost=2):
    x_cola = model.addVar(name="x_cola")
    x_juice = model.addVar(name="x_juice")
    x_water = model.addVar(name="x_water")

    model.setObjective(
        cost * x_cola + cost * x_juice + cost * x_water,
        GRB.MINIMIZE)

    model.addConstr(
        x_cola + x_juice + x_water <= capacity)
```

During arithmetic and model construction, each traced scalar propagates its source path and linear contribution. For the normalized constraint

$$x_{\text{cola}} + x_{\text{juice}} + x_{\text{water}} - \texttt{capacity} \leq 0,$$

the resulting dependencies are

Source parameter	LP coefficient	Weight
`cost`	$c_{x_{\text{cola}}}$	1
`cost`	$c_{x_{\text{juice}}}$	1
`cost`	$c_{x_{\text{water}}}$	1
`capacity`	normalized row constant	-1

The tracing results are serialized as a structured mapping artifact. Each source parameter is identified by its path in the input data structure, and each affected LP coefficient is recorded together with the corresponding affine contribution. The artifact contains:

- **Source parameters:** numerical source parameters extracted from the reference solver-function inputs and indexed by their paths in the input data structure.
- **Objective mapping:** source parameters contributing to each objective LP coefficient, together with their recorded contributions.
- **Constraint mapping:** source parameters contributing to the constant term and variable LP coefficients of each normalized constraint.
- **Variable and bound mapping:** variable records and source parameters contributing to finite lower or upper bounds, if any.
- **Source metadata:** basic metadata associated with the source parameters, including their default Python types and whether their default values are integral.

More formally, let $\mathcal{S}_j$ denote the source parameters linked to an LP coefficient q_j. The traced affine dependency is represented as

$$q_j(\theta) = \gamma_j + \sum_{p \in \mathcal{S}_j} \beta_{jp} p,$$

where β_{jp} is the recorded contribution of source parameter p to LP coefficient q_j. If p is updated from p to $p' = p + \delta$, every linked LP coefficient is updated as

$$q'_j = q_j + \beta_{jp}\delta, \qquad j : \ p \in \mathcal{S}_j.$$

Therefore, LP coefficients linked to the same source parameter are updated consistently according to the recorded mapping.

B.1.2 Semantic Domain Inference Prompt

The semantic domain inference step uses an LLM only to annotate admissible domains of source parameters. It does not modify the LP model, generate test instances, or certify constraint coverage. The complete prompt template used in this step is shown below.

```
You are annotating source parameters for an LP word-problem generator.

Task:
For the listed sources, decide whether the heuristic domain should be overridden by a
     harder semantic domain from the word problem and code defaults.

Important:
- This is about problem parameters, not decision variables.
- Infer every domain and bound from the supplied problem, code, model, source
  value, and source metadata. Do not apply a predefined domain based only on a
  parameter-name token, Python numeric type, or the sign of its default value.
- Return strict JSON only with key source_domains.
- source_domains must contain exactly one record for EVERY source in the input.
- Do not omit a source and do not return sources that are absent from the input.
- Each record must have keys: source, domain, lower_bound, upper_bound, step, hard,
     confidence, evidence.
- Set hard to true because these domains are enforced during instance generation.
- Keep evidence short. Do not repeat the full question text.

Original question:
```text
<ORIGINAL_QUESTION>
```

Reference solver code:
```python
<REFERENCE_SOLVER_CODE>
```

Reference model:
```text
<REFERENCE_MODEL>
```
```

````
Source parameters:
```json
{{SOURCE_PARAMETER_PAYLOAD}}
```
````

B.2 COVERAGE-GUIDED OBJECTIVE SEARCH

Coverage-Guided Objective Search (CGOS) is the objective-search subproblem used in our generation procedure. Given a fixed LP feasible region, a reference objective vector c^0, a set of target inequality constraints $\mathcal{T}$, and modifiable coefficients, CGOS searches for an an objective vector that target constraints can appear in an optimality certificate. The feasible region is fixed during each call. Let the fixed candidate feasible region be represented as

$$Ax \le b, \qquad Ex = d, \qquad \ell \le x \le u,$$

where $Ax \le b$ denotes the explicit inequality constraints in the reference formulation, $Ex = d$ denotes equality constraints, and $\ell \le x \le u$ denotes finite variable bounds. In the CGOS model, finite variable bounds are converted into additional linear inequalities. We denote the resulting full inequality system by

$$\bar{A}x \le \bar{b}.$$

Only the original explicit inequalities are treated as coverage targets. Equality constraints and variable bounds are included in the optimality conditions but are not counted as target constraints.

Let $\mathcal{G}_c$ be the set of source-parameter groups that affect only objective LP coefficients. For each group $g \in \mathcal{G}_c$, let α_g denote the perturbation applied to the corresponding parameter, and let $d_g \in \mathbb{R}^n$ denote the induced change direction in the objective vector. The objective vector used in CGOS is therefore

$$c(\alpha) = c^0 + \sum_{g \in \mathcal{G}_c} d_g \alpha_g.$$

Each perturbation α_g is restricted to the admissible semantic domain inferred for the corresponding source parameter, denoted by

$$\alpha_g \in \Delta_g, \qquad g \in \mathcal{G}_c.$$

The domain Δ_g encodes the allowed lower and upper bounds of the source-parameter change and, when applicable, integrality restrictions.

For each row r of $\bar{A}x \le \bar{b}$, let $s_r \ge 0$ be the slack variable and let $\lambda_r \ge 0$ be the associated dual multiplier. Let ν denote the free dual multipliers for the equality constraints. CGOS imposes the KKT conditions for the LP under the objective vector $c(\alpha)$:

$$\bar{A}x + s = \bar{b}, \tag{4}$$
$$Ex = d, \tag{5}$$
$$s \ge 0, \tag{6}$$
$$\lambda \ge 0, \tag{7}$$
$$c(\alpha) + \bar{A}^\top \lambda + E^\top \nu = 0, \tag{8}$$
$$\lambda_r s_r = 0, \qquad r = 1, \dots, \bar{m}. \tag{9}$$

Equation equation 4 enforces primal feasibility for the full inequality system, where the slack vector s converts inequalities into equalities. Equation equation 5 enforces equality constraints. Equations equation 6 and equation 7 impose nonnegativity of primal slacks and inequality dual multipliers, respectively. Equation equation 8 is the stationarity condition under the objective vector induced by source-parameter perturbations. Equation equation 9 enforces complementary slackness between each inequality row and its dual multiplier. For compactness, let $\mathcal{KKT}(P, c(\alpha))$ denote the set of tuples (x, s, λ, ν) satisfying Eqs. equation 4–equation 9 for the fixed feasible region P under objective vector $c(\alpha)$.

For each target constraint $i \in \mathcal{T}$, CGOS introduces a binary variable $z_i \in \{0, 1\}$. Since strict positivity of a dual multiplier cannot be directly encoded in a mixed-integer linear model, we use the numerical condition

$$\lambda_i \ge \frac{\epsilon_{\text{search}}}{\rho_i} z_i, \qquad i \in \mathcal{T},$$

Algorithm 2 Outer Frontier Search

Require: Original LP $P(\theta^0)$, uncovered targets $\mathcal{T}$
Ensure: Generated LP $P(\theta^k)$, working set $\mathcal{W}^k$, dependent active set $\mathcal{D}^k$

$P(\theta^k) \leftarrow \mathrm{CGOS}(P(\theta^0), \mathcal{T})$ ▷ Objective search
if $P(\theta^k) = \varnothing$ **then**
  return $\varnothing$
end if
$\mathcal{A}^k \leftarrow \mathrm{Solver}(P(\theta^k))$ ▷ Active constraints at optimum
Initialize $\mathcal{W}^k \leftarrow \varnothing$, $G \leftarrow E$ ▷ rank space initialized by equality rows
for all $a \in \mathcal{A}^k$ **do**
  if $\mathrm{rank}\left(\begin{bmatrix} G \\ a^\top \end{bmatrix}\right) > \mathrm{rank}(G)$ **then** ▷ Increase rank
    $\mathcal{W}^k \leftarrow \mathcal{W}^k \cup \{a\}$, $G \leftarrow \begin{bmatrix} G \\ a^\top \end{bmatrix}$
  end if
end for
$\mathcal{D}^k \leftarrow \mathcal{A}^k \setminus \mathcal{W}^k$ ▷ active but linearly dependent target constraints
return $P(\theta^k), \mathcal{W}^k, \mathcal{D}^k$

where $\epsilon_{\mathrm{search}} > 0$ is an implementation threshold and $\rho_i = \max\{1, \|a_i\|_\infty\}$ normalizes the threshold by the scale of the corresponding constraint row. This normalization makes the test less sensitive to equivalent rescalings of the same inequality constraint.

The CGOS model is formulated as

$$\begin{aligned} \max_{x,\alpha,s,\lambda,\nu,z} \quad & \sum_{i\in\mathcal{T}} z_i \\ \text{s.t.} \quad & (x, s, \lambda, \nu) \in \mathcal{K}KT(P, c(\alpha)), \\ & \lambda_i \geq \frac{\epsilon_{\mathrm{search}}}{\rho_i} z_i, \quad i \in \mathcal{T}, \\ & \alpha_g \in \Delta_g, \quad g \in \mathcal{G}_c, \\ & z_i \in \{0, 1\}, \quad i \in \mathcal{T}. \end{aligned}$$

When multiple perturbations cover the same number of target constraints, we use a secondary objective that minimizes the total absolute source-parameter change:

$$\min \sum_{g\in\mathcal{G}_c} |\alpha_g|.$$

This secondary objective favors smaller admissible source-parameter perturbations while preserving the primary objective of maximizing the number of target constraints supported by positive dual multipliers.

The complementarity constraints are implemented through a standard big-M mixed-integer linearization. For each inequality row r, CGOS introduces a binary activity variable y_r and imposes

$$s_r \leq M_r^s (1 - y_r), \qquad \lambda_r \leq M^\lambda y_r.$$

The slack bound M_r^s is computed from variable-wise lower and upper bounds over the fixed feasible region, and M^λ is a fixed upper bound on dual multipliers. In our implementation, the variable-wise bounds are obtained by separately minimizing and maximizing each variable over the fixed feasible region before constructing the CGOS model.

CGOS returns the induced objective vector $c(\alpha)$, the source-parameter perturbations $\{\alpha_g\}$, and a primal witness x. This output is not used as a final coverage certificate by itself. We subsequently validate the witness by checking feasibility and optimality against an independent LP solve under the induced objective vector. At the validated optimum, we recompute active target constraints and construct the target-prioritized linearly independent working set defined in the main paper. A target constraint is counted as covered only if it enters this independently verified working set. Thus, CGOS serves as a candidate-generation model, while the final coverage decision is made by an independent solver-based certification step.

Algorithm 3 RHS/LHS Perturbation

Require: Original LP $P(\theta^0)$, target constraint a, source-parameter groups $\mathcal{G}$
Ensure: Generated LP $P(\theta^a)$, working set $\mathcal{W}^a$

$\mathcal{G}_b(a) \leftarrow \{g \in \mathcal{G} \mid g \text{ modifies the RHS of } a\}$ ▷ RHS-related groups
for all $g \in \mathcal{G}_b(a)$ **do**
  $\Delta_g^b(a) \leftarrow \mathrm{RHS}(P(\theta^0), a, g)$ ▷ admissible RHS perturbations
  for all $\delta \in \Delta_g^b(a)$ **do**
    $P(\theta^{a,\delta}) \leftarrow \mathrm{Perturb}(P(\theta^0), g, \delta)$
    $P(\theta^a), \mathcal{W}^a \leftarrow \mathrm{CGOS}(P(\theta^{a,\delta}), \{a\})$
    if $P(\theta^a) \neq \varnothing$ and $a \in \mathcal{W}^a$ **then**
      return $P(\theta^a), \mathcal{W}^a$
    end if
  end for
end for
$\mathcal{G}_A(a) \leftarrow \{g \in \mathcal{G} \mid g \text{ modifies the LHS of } a\}$ ▷ LHS-related groups
for all $g \in \mathcal{G}_A(a)$ **do**
  $\Delta_g^A(a) \leftarrow \mathrm{LHS}(P(\theta^0), a, g)$ ▷ admissible LHS perturbations
  for all $\delta \in \Delta_g^A(a)$ **do**
    $P(\theta^{a,\delta}) \leftarrow \mathrm{Perturb}(P(\theta^0), g, \delta)$
    $P(\theta^a), \mathcal{W}^a \leftarrow \mathrm{CGOS}(P(\theta^{a,\delta}), \{a\})$
    if $P(\theta^a) \neq \varnothing$ and $a \in \mathcal{W}^a$ **then**
      return $P(\theta^a), \mathcal{W}^a$
    end if
  end for
end for
return $\varnothing$

B.3 Detailed Constraint-Coverage-Guided Generation Procedure

Algorithm in the main paper summarizes the overall generation loop. This section expands the implementation-level details abbreviated in the main algorithm, including outer objective search, target-specific , and the certification rule used to accept generated candidates. These subroutines use the same coverage criterion as the main paper: a target constraint is counted as covered only when it enters the target-prioritized linearly independent working set at a certified optimum.

As described in Algorithm 2, the outer search finds a new optimal frontier while preserving the original feasible region. It applies CGOS to the original LP with the current uncovered target set $\mathcal{T}$, allowing only source-parameter groups that affect the objective coefficients. The generated LP $P(\theta^k)$ is then solved to obtain the active target set $\mathcal{A}^k$. Starting from the rank space spanned by equality rows, the algorithm scans active target constraints and selects those whose normal vectors increase the rank, forming the working set $\mathcal{W}^k$. The remaining active targets form the dependent active set $\mathcal{D}^k = \mathcal{A}^k \setminus \mathcal{W}^k$. Constraints in $\mathcal{W}^k$ are added to the covered set, while constraints in $\mathcal{D}^k$ are passed to the inner target-specific search.

As described in Algorithm 3, the inner search is applied to a dependent active target a. It first tries source-parameter groups that modify the right-hand side of the target row. If no accepted instance is found, it then tries source-parameter groups that modify coefficients in the left-hand side of the target row. Each candidate perturbation must satisfy the admissible semantic domain of the corresponding source parameter. After a perturbation fixes the candidate feasible region, the objective is selected by the same objective-search routine, using the singleton target set $\{a\}$.

Candidate certification is shared by both outer and inner searches. The CGOS model returns an induced objective and a witness solution, but this output is not itself used as the final coverage decision. We independently solve the candidate LP under the induced objective and check that the witness is feasible and attains the same optimal objective value within numerical tolerance. The active explicit inequalities and active variable bounds are then recomputed at the certified witness, and the target-prioritized linearly independent working set is constructed using the same rule as in the main paper. A candidate is accepted only when at least one target constraint under consideration enters this working set. The generation procedure stops when the outer search can no longer

assign any remaining target to a new certified frontier. After outer search is exhausted, the same target-specific procedure is applied to unresolved targets. Constraints that still cannot be certified as covered are left uncovered rather than forced through parameter changes outside their admissible domains.

B.4 Instance Reconstruction

The following prompt is used to reconstruct the natural-language problem description after the source parameters have been updated. It restricts the reconstruction to numerical changes supported by the generated source-parameter updates.

```
You are a careful operations-research dataset editor. You rewrite word problems
by changing only numeric parameter values that are explicitly listed in
source_parameter_updates.

We generated a new LP instance by changing source/code parameters. Rewrite the
original word problem into the new instance.

Important:
- Before rewriting, use symbolic_sources_for_math_changes together with
    source_parameter_updates to construct an internal dependency graph from the changed
    formulation coefficients to the affected problem parameters. Process the affected
    parameters in a topological order of this graph. When one coefficient depends on
    multiple source parameters, or one source parameter affects multiple coefficients,
    treat them as coupled dependencies and apply all listed target values consistently.
- Preserve the original wording, structure, entities, units, and optimization question as
    much as possible.
- Treat source_parameter_updates as the candidate reconstruction plan derived from the
    generated LP. Confirm each candidate against the dependency graph and math_changes,
    then include the confirmed values in parameter_updates. Only change numeric values
    supported by confirmed candidates. Do not silently replace a candidate with a
    different value: if a candidate is inconsistent, leave that value unchanged in the
    question and report it in unmapped_source_updates and audit_notes.
- Treat source_domain as a hard validity requirement. In particular, do not write
    fractional values for integer, nonnegative_integer, or positive_integer source
    parameters.
- Do not add explanations inside rewritten_question.
- Do not wrap rewritten_question in Markdown fences or any code block.
- If a source update cannot be located in the question text, keep the question unchanged
    for that value and report it in unmapped_source_updates.
- Return strict JSON only, with keys: rewritten_question, parameter_updates,
    unmapped_source_updates, audit_notes.

Original question:
```text
{{ORIGINAL_QUESTION}}
```

Reference solver code:
```python
{{REFERENCE_SOLVER_CODE}}
```

Reference mathematical model:
```text
{{REFERENCE_MATHEMATICAL_MODEL}}
```

Generated instance data:
```json
{{GENERATED_INSTANCE_DATA}}
```
```

C Experimental Details and Additional Results

C.1 Data filtering

We construct the LP subset by applying a consistent definition of linear programming across all five benchmarks. Dataset-provided type labels are used only to identify candidate instances; the final classification is based on the natural-language description, the reference formulation, its implementation, and solver-based inspection. An instance is retained only if (i) all decision variables are continuous, (ii) the objective function and all constraints are linear, and (iii) the reference

implementation can be solved to optimality. We exclude instances containing binary or integer variables, quadratic or other nonlinear expressions, indicator or general constraints, or missing or non-executable reference implementations.

Table 3: LP instances retained by dataset filtering.

Dataset	LP instances
NLP4LP	48
IndustryOR	17
LogiOR	21
MAMO	70
ReSocratic	1,844

The filtering procedure consists of three stages: an LLM-assisted semantic check, a static code check, and a solver-based execution check. First, the semantic check determines whether the decision quantities are implicitly required to be integral. Second, the static check identifies discrete or nonlinear components in the reference implementation. Finally, the surviving reference models are executed and inspected with Gurobi to confirm that they satisfy the definition of LP and can be solved to optimality.

For the semantic check, we use DeepSeek-V4-Pro to inspect the natural-language description of each candidate instance. The model is prompted to determine whether the decision quantities may take fractional values or must be integral. For example, quantities representing vehicles, workers, machines, or indivisible items may be required to take integer values even when the supplied implementation declares the corresponding variables as continuous. Instances identified as having such semantic integrality requirements are conservatively excluded because their intended formulations are MIPs rather than LPs. For the static code check, we examine variable declarations, objective expressions, constraint expressions, and solver-specific interfaces for discrete or nonlinear model components. Each reference model that passes both the semantic and static checks is then built and solved with Gurobi. During this execution check, we verify that the resulting model contains no integer or binary variables, quadratic objective terms, or quadratic constraints. An instance is retained only if Gurobi reports an optimal solution. The counts in Table 3 report the numbers of instances identified as LPs before any subsequent experiment-specific preprocessing.

C.2 Experimental Configurations

We evaluate six automated optimization modeling methods. We evaluate DeepSeek-V4-Pro and GPT-5-mini as direct zero-shot baselines, together with OptiMUS-0.3, ORThought, ORLM, and SIRL. Unless otherwise stated, we follow the prompts, workflows, and inference configurations provided in the corresponding papers. We use DeepSeek-V4-Pro as the language-model backend for OptiMUS-0.3, ORLM-LLaMA-3-8B as the checkpoint for ORLM, and SIRL-Qwen2.5-7B-Gurobi as the checkpoint for SIRL. The following prompt is used to zero-shot baselines:

```
You are an expert in operations research, mathematical optimization, and Gurobi Python
     modeling. Your response must always begin with a complete executable Python code block
      that defines solve.

Formulate and solve the optimization problem below.
Problem description:
{question}

Return executable Gurobipy code and a mathematical model. The executable code is mandatory
     and must be the first block in the response.

The Python code must satisfy all of the following requirements:
1. Define exactly one top-level function named `solve`.
2. Put the numerical problem data in named function arguments whose default values are
     taken directly from the problem description whenever practical.
3. Build and optimize the model entirely inside `solve`.
4. Return the optimal objective value as a Python `float`; return `None` when no finite
     optimal solution exists.
5. Do not call `solve` and do not include an example invocation.
```

````
Output exactly the following two fenced blocks and no other text. The output must follow
    this format:

```python
[complete executable Python code]
```
```model
[mathematical model corresponding exactly to the code]
```
````

For the descriptive comparison with EquivaMap, we use the manually confirmed hidden-error formulation pairs evaluated in the main paper. For each pair, the evaluated formulation returns the same optimal objective value as the reference formulation on the original instance, but contains a verified constraint error that is exposed by our test suite. The generated variant is used only to establish the formulation error and is not provided to EquivaMap. EquivaMap compares the reference and evaluated formulations under the original numerical setting.

We adapt both formulations in each pair to the input format required by EquivaMap. The adapted inputs preserve the model parameters, decision-variable families and their index structures, constraints, and objective, while also providing executable formulations for solution mapping and verification. Following the evaluation protocol of EquivaMap, we use GPT-4o with temperature 0 and perform three independent mapping attempts for each formulation pair. A pair is classified as equivalent if any attempt produces a mapping under which the mapped optimizer of the evaluated formulation is feasible and optimal for the reference formulation. It is classified as non-equivalent only when none of the three attempts establishes such a verified mapping. Because all evaluated pairs contain confirmed formulation errors, a non-equivalent result is recorded as a formulation flagged as incorrect, whereas an equivalent result is recorded as a formulation not flagged by EquivaMap. These outcomes describe behavior on the selected cohort and do not estimate general detection accuracy.

For a descriptive comparison on this selected hidden-error cohort, let $\mathcal{E}$ denote the formulations identified by the augmented test suites as containing constraint-level modeling errors. For a validation method v, we report the fraction of formulations in $\mathcal{E}$ that it flags as incorrect:

$$\text{selected-cohort flag rate}(v) = |\{f \in \mathcal{E} : v(f) = \text{wrong}\}|/|\mathcal{E}|.$$

This quantity describes the behavior of a validation method on the selected cohort $\mathcal{E}$; it is not an estimate of general detection accuracy, recall, or false-negative rate.

Table 4: Validation outcomes on the selected hidden-error cohort identified by the augmented test suites. Each percentage reports the fraction of the selected formulations that a validation method flags as incorrect. This cohort-conditioned comparison is descriptive and is not an estimate of general detection accuracy.

Validation Method	Answer-based	EquivaMap	Ours
Flagged as incorrect	0%	44%	100%

Table 4 provides a descriptive comparison on the hidden-error formulations identified above, with answer-based validation included as the conventional baseline. Because the cohort is defined by failure on at least one augmented instance, the entries should not be interpreted as general detection accuracy. Answer-based validation flags 0% of these cases by construction, since all formulations in the cohort return the correct answer on the original instance. EquivaMap flags 44% of the selected cases, while the augmented suites flag all cases in this cohort by the same selection rule. The comparison therefore characterizes validation outcomes on the suite-exposed cases rather than establishing general superiority or estimating false-negative rates. A representative example illustrating this difference is provided in the supplementary material.

C.3 Additional Ablation Results

We replace the constraint-coverage-guided (CCG) instance generation module with waypoint-guided random mutation. The mutation variant is independently run five times for each problem

Table 5: Stability of waypoint-guided random mutation across five independent runs. Each entry reports the mean and standard deviation of the number of problems reaching the target coverage, followed by the total number of evaluated problems.

	NLP4LP	IndOR	LogiOR	ReSoc	MAMO
Reached Problems	46.00 ± 0.00/48	16.00 ± 0.00/17	19.80 ± 0.45/21	1757.40 ± 13.79/1844	39.00 ± 4.18/70

Table 6: Detailed ablation results across five independent runs. Each entry reports the average total test-suite size among problems reaching the target coverage, followed by the number of such problems.

Run	NLP4LP	IndOR	LogiOR	ReSoc	MAMO
w/o CCG (Seed 0)	1.97/46	1.94/16	3.55/20	2.12/1762	6.02/42
w/o CCG (Seed 1)	1.87/46	2.00/16	3.37/19	2.13/1763	5.98/41
w/o CCG (Seed 2)	1.85/46	2.00/16	3.50/20	2.13/1775	6.86/43
w/o CCG (Seed 3)	2.06/46	2.00/16	3.60/20	2.12/1746	4.41/34
w/o CCG (Seed 4)	1.93/46	2.00/16	3.60/20	2.13/1741	4.63/35
Ours	1.87/48	1.69/17	3.20/21	2.06/1844	6.46/70

using different random seeds, with a time budget of 300 seconds per run. For each problem p, the constraint coverage achieved by our complete method defines the target coverage. A run is considered successful if it reaches this target within the budget. Intermediate waypoints may guide subsequent mutations, but only instances that cover at least one previously uncovered target constraint are included in the generated test suite. As reported in the main paper, CCG requires fewer generated instances to reach the same constraint coverage and enables more problems to reach the target coverage within the fixed budget.

Table 5 summarizes the stability of waypoint-guided random mutation across five random seeds. The number of problems reaching the target coverage remains stable on NLP4LP and IndustryOR, while LogiOR and ReSocratic exhibit modest seed-dependent variation. The variation is substantially more pronounced on MAMO, where only 39.00 ± 4.18 of 70 problems reach the target. These results indicate that coverage generation based on random mutation can become less reliable within a fixed budget when the remaining coverage targets are more difficult to exercise.

Table 6 further reports the detailed results of the five mutation runs together with those of CCG. The smaller average test-suite sizes of Seeds 3 and 4 on MAMO should not be interpreted as greater efficiency, because these runs reach the target on substantially fewer problems and therefore retain a comparatively easier subset. Under the same per-problem budget, CCG enables more problems to reach the target coverage and, under the matched-coverage comparison reported in the main paper, requires fewer test instances. This advantage is consistent with the design of CGOS, which explicitly searches for objective functions that allow multiple uncovered target constraints to be exercised jointly by a generated instance.

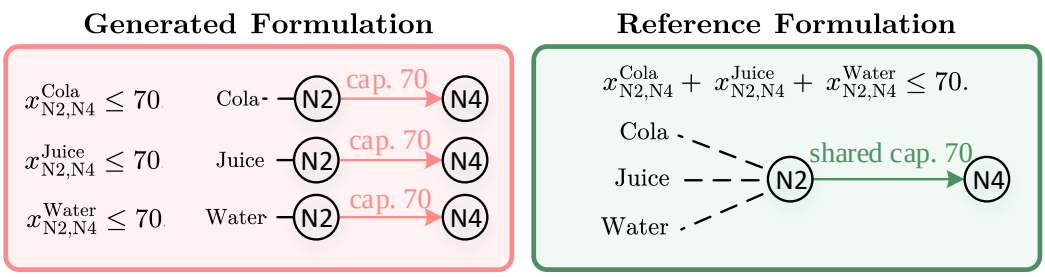
D Detailed Case Study

This section provides additional details for the representative hidden-error case discussed in the main paper. We first present the original problem description and transportation network. We then compare the reference and evaluated formulations, and finally explain how the coverage-guided parameter modification exposes the hidden capacity error.

Figure 4 presents a representative hidden constraint error exposed by the augmented test suite. The case is drawn from LogiOR and models the transportation of Cola, Juice, and Water through a capacitated network. Let $\mathcal{P}$ denote the three products and $x_{p,N2,N4}$ the flow of product p on arc N2–N4. The reference and evaluated formulations encode the capacity of this arc differently:

$$\text{Reference:} \quad \sum_{p\in\mathcal{P}} x_{p,N2,N4} \le 70,$$

$$\text{Evaluated:} \quad x_{p,N2,N4} \le 70, \qquad \forall p \in \mathcal{P}.$$



Test instance	x^{Cola}	x^{Juice}	x^{Water}	Constraints check
Original optimum	30	20	10	$30+20+10=60\le 70$: error hidden
Augmented optimum	40	30	20	$40+30+20=90>70$: error exposed

Figure 4: Case study on LogiOR. The left panel shows the generated mathematical formulation, while the right panel shows the correct reference formulation. The bottom table compares the optimal solutions under the original instance and the generated instance.

The reference constraint requires all products to share a total capacity of 70, whereas the evaluated formulation assigns the full capacity separately to each product. This is an aggregation error that removes the coupling among product flows and enlarges the feasible region. Nevertheless, the resulting model remains mathematically feasible and can be solved normally, so neither code execution nor the solver status reveals the error. It is also invisible to answer-based validation on the original instance. At the original optimum, the aggregate flow on arc N2–N4 is 60, leaving a slack of 10 in the intended shared-capacity constraint. The optimal solution therefore belongs to the feasible regions of both formulations. Although the evaluated formulation admits additional solutions, none of them improves the objective under the original transportation costs. Consequently, the evaluated formulation returns the same optimal objective value as the reference formulation and is judged correct by conventional evaluation. This agreement only establishes that the erroneous relaxation does not affect the optimum at this particular parameter setting; it does not verify that the capacity relationship has been modeled faithfully.

Constraint-coverage-guided generation identifies the shared-capacity constraint as an uncovered target and changes only three admissible transportation-cost parameters to make it relevant to optimality. Under the modified costs, the evaluated formulation routes a total flow of 90 through arc N2–N4, exceeding the intended capacity by 20. This solution is feasible for the relaxed formulation but infeasible for the reference formulation, causing their optimal answers to diverge. The generated instance thus exposes the latent constraint error without changing the network structure, demands, or capacities. By making the shared-capacity constraint relevant to optimality, the cost perturbation converts a previously untested constraint into a test that distinguishes the two formulations. This illustrates how coverage-guided generation provides constraint-level diagnostic evidence even when the original answers agree. Complete details are provided in the supplementary material.

D.1 Original Problem Instance

The original problem description is reproduced below. Capacities are measured in pallets, and transportation costs are measured in dollars per pallet.

Original Problem Description

A logistics company needs to optimize the delivery of three types of drinks (Cola, Juice, and Mineral Water) from two production facilities (P1 and P2) to three retail stores (S1, S2, S3) via a transportation network with five intermediate nodes (N1–N5). The directed arcs between nodes have capacity limits, measured in pallets, and unit transportation costs, measured in dollars per pallet.

The network connections, capacities, and costs are as follows:

- $P1 \to N1$: capacity 80, cost 2;
- $P1 \to N2$: capacity 60, cost 3;
- $P2 \to N3$: capacity 90, cost 2;
- $N1 \to N3$: capacity 50, cost 1;
- $N2 \to N4$: capacity 70, cost 2;
- $N3 \to N5$: capacity 100, cost 3;
- $N4 \to S1$: unlimited capacity, cost 1;
- $N4 \to S2$: unlimited capacity, cost 2;
- $N5 \to S2$: unlimited capacity, cost 1;
- $N5 \to S3$: unlimited capacity, cost 2.

The demand requirements are Cola ($S1$: 30, $S2$: 20, $S3$: 10), Juice ($S1$: 15, $S2$: 25, $S3$: 5), and Mineral Water ($S1$: 10, $S2$: 15, $S3$: 20). The production capacities are $P1$ (Cola: 40, Juice: 30, Water: 20) and $P2$ (Cola: 30, Juice: 25, Water: 40). The goal is to satisfy all demands while minimizing the total transportation cost.

Figure 5 formalizes the transportation network. Each arc label reports the pair (u_{ij}, c_{ij}), where u_{ij} is the total capacity of arc (i, j) and c_{ij} is its unit transportation cost. The highlighted arc $N2 \to N4$ is the arc whose shared-capacity relationship is misrepresented by the evaluated formulation.

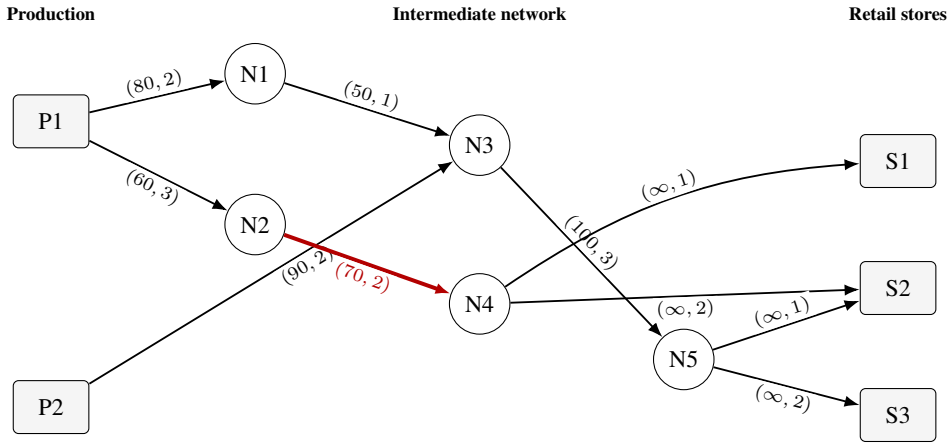


Figure 5: Transportation network of the original instance. Each arc label gives (capacity, unit cost). The highlighted arc has a total capacity of 70 pallets shared by all three products.

For completeness, Table 7 summarizes the product-specific supply limits and demands. A dash indicates that the corresponding node is not a source or demand node for that row.

Table 7: Product-specific supply limits and demands in the original instance. All quantities are measured in pallets.

Type	Node	Cola	Juice	Water
Supply limit	P1	40	30	20
	P2	30	25	40
Demand	S1	30	15	10
	S2	20	25	15
	S3	10	5	20

Reference formulation

```
import gurobipy as gp
from gurobipy import GRB


def solve_beverage_distribution(
    nodes=[...],
    commodities=['Cola', 'Juice', 'Water'],
    arcs_data=[...],
    demand={...},
    supply={...}
):
    model = gp.Model("BeverageDistribution")

    arcs, capacity, cost = gp.multidict({
        (i, j): [cap, c]
        for (i, j, cap, c) in arcs_data
    })

    flow = model.addVars(
        arcs, commodities, lb=0, name="flow"
    )

    # Supply, demand, and
    # flow-conservation constraints
    ...

    # One shared capacity per arc
    for i, j in arcs:
        if capacity[i, j] != float('inf'):
            model.addConstr(
                gp.quicksum(
                    flow[i, j, k]
                    for k in commodities
                )
                <= capacity[i, j]
            )

    model.optimize()
    ...
```

Evaluated formulation

```
def optimize_drink_distribution(
    production_capacity_P1={...},
    production_capacity_P2={...},
    demand={...},
    arc_capacities={...},
    arc_costs={...}
):
    import gurobipy as gp
    from gurobipy import GRB

    products = ['Cola', 'Juice', 'Water']

    m = gp.Model("DrinkDistribution")

    x = {}

    # Arc capacity is copied to every
    # product-specific variable
    for (i, j) in arc_costs:
        for k in products:
            var_name = f"x_{i}_{j}_{k}"
            cap = arc_capacities[(i, j)]
            ub = (
                cap
                if cap != float('inf')
                else 1e9
            )
            x[(i, j, k)] = m.addVar(
                lb=0,
                ub=ub,
                name=var_name
            )

    m.update()
    # Objective, supply, demand, and flow-
      conservation constraints
    ...

    m.optimize()
    ...
```

Figure 6: Code-level comparison of the reference and evaluated formulations. Repeated data definitions and modeling components unrelated to the capacity error are omitted. The reference formulation imposes one aggregate capacity constraint on each finite arc, whereas the evaluated formulation assigns the full arc capacity independently to every product-specific flow variable.

D.2 Formulation Difference

Figure 6 compares the relevant implementation fragments of the reference and evaluated formulations. The shaded code regions isolate their different implementations of finite arc capacities. The reference formulation creates nonnegative product-specific flow variables and then imposes one aggregate capacity constraint for each finite arc:

$$\sum_{k \in \mathcal{K}} x_{ijk} \leq u_{ij}.$$

Thus, all products transported through arc (i, j) share the same total capacity u_{ij}. The evaluated formulation instead assigns the arc capacity directly as the upper bound of every product-specific flow variable:

$$0 \leq x_{ijk} \leq u_{ij}, \qquad \forall k \in \mathcal{K}.$$

This difference removes the intended coupling among the three product flows. In particular, the evaluated formulation can admit an aggregate flow greater than 70, provided that the flow of each individual product remains below 70. It is therefore a relaxation of the reference formulation. Nevertheless, it remains mathematically feasible and executable, so neither successful code execution nor the solver status alone reveals the error. Under the original numerical setting, the aggregate flow on $N2 \rightarrow N4$ is 60, leaving a slack of 10 in the intended shared-capacity constraint. Consequently, the optimal solution is feasible for both formulations, and both return the same objective value of 940.0. The formulation difference is therefore hidden from single-instance answer-based evaluation.

D.3 Coverage-Guided Parameter Modification and Error Exposure

The shared-capacity constraint on arc $N2 \rightarrow N4$ is not exercised by the original optimum because its aggregate flow is only 60, below the capacity of 70. To make this constraint relevant to optimality,

the coverage-guided generation procedure modifies only three admissible unit transportation costs. Table 8 reports these changes.

Table 8: Transportation-cost parameters modified in the generated test instance. Capacities, supplies, demands, and the network structure remain unchanged.

Arc	Original cost	Generated cost
$P2 \to N3$	2.0	4.0
$N2 \to N4$	2.0	1.5
$N4 \to S2$	2.0	1.5

These changes make transportation through $N2 \to N4$ relatively more attractive. Because each cost parameter is shared by the three product-specific objective coefficients on its corresponding arc, changing an arc cost updates the costs of Cola, Juice, and Water flows together. Importantly, this modification changes only objective coefficients. The arc $N2 \to N4$ retains its original capacity of 70, and all supply limits and demand requirements remain unchanged. Therefore, the generated problem preserves the original transportation structure and semantics while presenting the two formulations with a different admissible numerical setting.

Table 9 compares the two formulations under the original and generated parameter settings. The flow tuple reports the Cola, Juice, and Water flows on $N2 \to N4$ in the evaluated formulation.

Table 9: Exposure of the shared-capacity error. The flow tuple gives the evaluated formulation's product-specific flows on the target arc.

Test instance	Reference objective	Evaluated objective	Flow tuple on $N2 \to N4$	Capacity check
Original	940.0	940.0	$(30, 20, 10)$	$60 \leq 70$
Generated	1057.5	1027.5	$(40, 30, 20)$	$90 > 70$

Under the original costs, the evaluated formulation routes 60 pallets through $N2 \to N4$, which satisfies the intended capacity constraint. Although the evaluated feasible region is larger, the additional solutions admitted by this relaxation do not improve the objective under the original costs. Both formulations consequently return 940.0. Under the generated costs, however, the evaluated formulation routes 90 pallets through the same arc. Each individual product flow remains below its incorrectly assigned upper bound of 70, so this solution is feasible for the evaluated formulation. Its aggregate flow, however, exceeds the intended shared capacity by 20 pallets and is therefore infeasible for the reference formulation.

The reference formulation must respect the shared capacity and obtains an optimal objective value of 1057.5, whereas the evaluated formulation exploits the erroneous relaxation and obtains the lower value 1027.5. The resulting objective mismatch exposes the hidden modeling error. In the generated reference instance, the previously uncovered shared-capacity constraint becomes relevant to optimality and provides a certified coverage witness.

D.4 EquivaMap Result and Failure Analysis

We further apply EquivaMap to the reference and evaluated formulations under the original numerical setting. In all three independent mapping attempts, EquivaMap identifies the same identity mapping of variables:

Variable mapping returned by EquivaMap

```
{
  "flow": [{
    "constant": 1,
    "variable": "x"
  }]
}
```

The returned mapping expresses each variable in the reference family `flow` as a linear combination of variables in the evaluated formulation. Here, the coefficient of `x` is 1, and no other variable family appears in the mapping. Since the two variable families have compatible arc–product indices, the mapping assigns each reference component $\texttt{flow}[i, j, k]$ the value of the corresponding evaluated component $\texttt{x}[i, j, k]$. EquivaMap then maps and fixes all 30 product–arc flow components of the evaluated formulation's original-instance optimizer in the reference formulation. The mapped solution remains feasible and optimal for the reference formulation and attains the same objective value of 940.0. Consequently, all three attempts classify the two formulations as equivalent, despite the confirmed shared-capacity modeling error.

The successful mapping is valid for the particular optimizer obtained under the original parameter setting. Specifically, EquivaMap verifies that

$$x^{\star}_{\mathrm{eval}}(\theta^0) \xrightarrow{f} \hat{x} \in \mathrm{Opt}\big(P_{\mathrm{ref}}(\theta^0)\big),$$

where f is the identity mapping above. This condition holds in the original instance because the optimal value of evaluated formulation happens to satisfy all intended capacity constraints. In particular, its aggregate flow on $N2 \to N4$ is 60, which does not exceed the correct capacity of 70. However, the existence of such an original-instance mapping witness does not establish that the two formulations have the same feasible region, nor does it establish their behavioral equivalence over the admissible parameter domain. In particular, it does not imply

$$\mathcal{F}_{\mathrm{eval}}(\theta) = \mathcal{F}_{\mathrm{ref}}(\theta), \qquad \forall \theta \in \Theta,$$

where $\mathcal{F}_{\mathrm{eval}}(\theta)$ and $\mathcal{F}_{\mathrm{ref}}(\theta)$ denote the feasible regions of the evaluated and reference formulations, respectively. Because the evaluated formulation replaces each shared arc-capacity constraint with independent variable upper bounds, its feasible region contains additional solutions that overload an arc in aggregate. These invalid solutions are not visited by the optimal value under the original costs, and therefore remain invisible to the original-instance mapping verification.