HALLUCINATIONS IN LLM-BASED OPTIMIZATION MODELING: DEFINITION, TAXONOMY, DETECTION BENCHMARKS, AND REFERENCE DETECTORS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly used to translate natural-language optimization problems into mathematical formulations and solver code. Most existing work evaluates the resulting models by comparing the objective value produced by generated solver code with a reference value. However, matching the reference objective value is not a reliable test of correctness, as it can mask unintended changes to optimization semantics, such as changes to the intended decisions, feasible region, objective meaning, or solver behavior. We define an *optimization-modeling hallucination* as a deviation from the intended optimization semantics. We develop, to our knowledge, the first fine-grained hallucination taxonomy specifically for optimization modeling, spanning objective, variable, constraint, and implementation failures. Building on this definition and taxonomy, we formulate *optimization-modeling hallucination detection* as structured prediction over tuples comprising a problem description, symbolic model, and solver code. To support detector development and evaluation, we construct development and test suites comprising more than $12{,}000$ detection instances, with disjoint source-problem sets. The test suite contains correct examples for measuring false alarms, controlled changes for measuring localization, and LLM-generated outputs for measuring detection of naturally occurring errors. Using these suites, we design and evaluate six detectors for optimization-modeling hallucinations. Among them, OptArgus is a multi-agent detector in which separate agents inspect the objective, variables, constraints, and solver code before combining their findings into a final diagnosis. Experiments show that OptArgus offers a strong overall trade-off across the three evaluation settings, combining the highest controlled-error localization scores among the six detectors with competitive false-alarm control and natural-error detection, while producing concise diagnoses. Together, these contributions establish optimization-modeling hallucination detection as a concrete empirical problem and foundation for more reliable optimization modeling.

1 INTRODUCTION

Optimization modeling (Bisschop, 2006; Hart et al., 2017) concerns the specification of a decision problem as a mathematical program, typically by defining decision variables, constraints that characterize feasible decisions, and an objective function that captures the desired notion of optimality. In practice, a stakeholder describes the task, operational context, and business rules in natural language; domain specialists interpret the application semantics; operations-research (OR) experts formulate the corresponding mathematical model; and technically trained modelers implement it in solver code. This workflow underlies decision support in logistics (Bartolacci et al., 2012), supply chains (Garcia & You, 2015), transportation (Gkiotsalitis, 2022), energy (Dincer et al., 2017), finance (Cornuejols et al., 2018), and healthcare (Rais & Viana, 2011). Despite its importance, conventional optimization modeling faces three persistent barriers (Fan et al., 2026; Xiao et al., 2025): it relies on scarce cross-disciplinary expertise, incurs substantial time and financial costs for formulation, validation, and implementation, and remains insufficiently agile because even modest changes in requirements, assumptions, or data interfaces can require revisions to both model and code. These limitations make **automatic optimization modeling** an attractive research direction.

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.

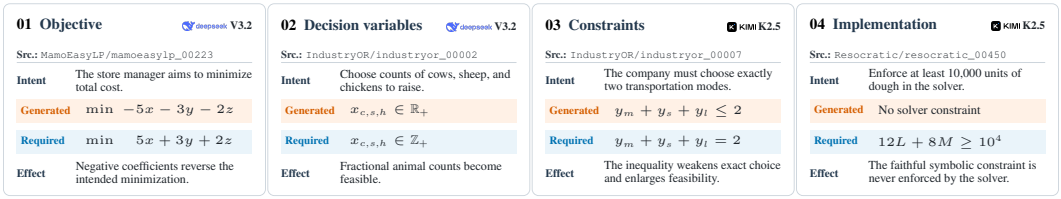


Figure 1: Four examples of optimization-modeling hallucinations. Each contrasts a generated fragment (orange) with the required semantics (blue). The *objective panel* reverses optimization intent through sign errors, the *variable panel* relaxes an integer-count decision into a continuous one, the *constraint panel* weakens an exact rule into an at-most rule, and the *implementation panel* preserves the symbolic requirement in appearance while failing to materialize it in solver code.

Recent studies (Xiao et al., 2024; Ahmaditeshnizi et al., 2024; Huang et al., 2025a; Lu et al., 2025; Chen et al., 2025) have explored the use of LLMs for automatic optimization modeling, where a model translates a natural-language problem description into a mathematical formulation and solver-ready code. These systems report promising results across diverse benchmarks (Ramamonjison et al., 2022; Huang et al., 2025c; Yang et al., 2025; Li et al., 2026b; Chen et al., 2026), suggesting that end-to-end LLM-based optimization modeling is becoming increasingly capable. A common limitation, however, is that success is often judged primarily by whether the generated solver code reproduces the reference objective value. For optimization modeling, this criterion is too weak: even when objective values match, the generated formulation and solver code may not faithfully encode the intended decision problem. In this paper, we refer to such structurally incorrect but superficially plausible outputs as **optimization-modeling hallucinations**. Figure 1 illustrates this gap. Such outputs can appear plausible, executable, and numerically correct while changing the intended decisions, feasible region, objective meaning, or solver behavior. Empirically, Table 1 (see the last column) shows that even among outputs matching the reference objective value, hallucinations remain at rates of up to 27.1% for some evaluated systems. In high-stakes settings, failing to detect such hallucinations can lead to systematically wrong allocations, infeasible or noncompliant plans, and unwarranted confidence in decision-support systems precisely where reliability matters most.

This motivates a different view of LLM-based automatic optimization modeling. Rather than treating objective-value agreement as the primary success criterion, we ask a more fundamental question: *Do the generated symbolic model M and solver implementation S faithfully realize the optimization structure specified by the natural-language problem description P?* This perspective connects our setting to broader work on hallucination detection (Huang et al., 2025b; Zhang et al., 2025; Li et al., 2025; Liu & Fang, 2025), while highlighting what makes optimization modeling distinctive: both the symbolic model and solver implementation must preserve the decisions, objective, and constraints specified in the problem description. Building on our definition, we develop, to our knowledge, the first fine-grained hallucination taxonomy specifically for optimization modeling, spanning objective, variable, constraint, and implementation failures. The taxonomy contains 83 entries: 18 objective, 18 variable, 31 constraint, and 16 implementation. Figure 2 summarizes the taxonomy; Table 18 and App. D provide the complete inventory, decision boundaries, and examples. With the definition and taxonomy in place, we formulate *optimization-modeling hallucination detection* as structured prediction over tuples (P, M, S), distinguishing semantic errors from equivalent reformulations and quality warnings that do not change the intended optimization problem (e.g., unnecessary redundancy or missed efficiency opportunities).

To support hallucination-detector development and evaluation, we introduce, to our knowledge, the first systematic development and test suites for this task. We curate 200 development problems and 484 disjoint test problems from five NL2Opt benchmarks: IndustryOR, MamoComplexLP, MamoEasyLP, NL4Opt, and ReSocratic. For each seed problem, we retain a human-validated reference formulation and solver program and collect the unedited outputs of 13 modeling systems as natural outputs. We also create controlled cases by applying a designated change to a reference artifact and recording its target type and location. The 200 development problems yield 200 reference cases, 1383 controlled cases, and 2600 natural outputs for detector design and configuration. The 484 test problems yield 484 reference-clean artifacts for restraint measurement, 1266 controlled cases for

localization, and 6292 natural outputs for natural-error detection, with label provenance detailed in App. B.4.1. Together, the two suites contain more than 12,000 instances.

Using these suites, we construct and evaluate six detectors for optimization-modeling hallucinations. They differ in whether an artifact is checked once or repeatedly, by one agent or by agents assigned to different components, and with or without the taxonomy during review. Among them, OptArgus is a multi-agent detector in which separate agents inspect the objective, variables, constraints, and solver code before combining their findings into a final diagnosis. The experiments reveal distinct strengths: OptArgus leads controlled localization with concise reports, independent specialists report the fewest clean-set findings, and generic taxonomy-aware review attains the highest natural-error F1.

In summary, our contributions are fourfold: (1) **Reliability gap and detection problem.** We demonstrate why objective-value matching is insufficient through a reference-assisted audit of 16 automatic optimization-modeling systems and formulate optimization-modeling hallucination detection as structured prediction over tuples comprising a problem description, symbolic model, and solver code (see Secs. 3 and 5); (2) **Fine-grained taxonomy.** We develop, to our knowledge, the first fine-grained hallucination taxonomy specifically for optimization modeling, spanning objective, variable, constraint, and implementation failures while distinguishing semantic errors from equivalent reformulations and quality warnings (see Sec. 4 and App. D); (3) **Development and test suites.** We construct more than 12,000 detection instances across development and test suites with disjoint source-problem sets, including complementary tests of false-alarm restraint, controlled localization, and detection of naturally occurring errors (see Sec. 5 and App. B); and (4) **Reference-detector study.** We construct and evaluate six detectors spanning different auditing strategies and characterize their trade-offs in false-alarm restraint, error localization, natural-error detection, and token use (see Secs. 6–7).

2 Related Work

LLMs for Optimization Modeling. Natural-language-to-optimization modeling (NL2Opt) translates verbal problem specifications into mathematical models and executable solver code (Xiao et al., 2025). Prompt- and agent-based methods include CoE (Xiao et al., 2024), OptiMUS (Ahmaditeshnizi et al., 2024), and the multi-agent framework of Mostajabdaveh et al. (2024); fine-tuned methods include ORLM (Huang et al., 2025a), OptMath (Lu et al., 2025), and LLMOPT (Jiang et al., 2025); and verification-aware methods include SIRL (Chen et al., 2025), StepORLM (Zhou et al., 2026), Opt-Verifier (Liu et al., 2026), and PEARL (Lu et al., 2026), which use solver-informed rewards, process supervision, dual-side checks, and interactive solver feedback, respectively.

Existing NL2Opt Benchmarks and Evaluation. Representative benchmarks include NL4Opt (Ramamonjison et al., 2022), IndustryOR (Huang et al., 2025a), MAMO (Huang et al., 2025c), OptiBench (Wang et al., 2024), ReSocratic (Yang et al., 2025), OptMath (Lu et al., 2025), Bench4Opt (Wang et al., 2026), ProOPF (Shen et al., 2026), MIPLIB-NL (Li et al., 2026b), OPT-Engine (Chen et al., 2026), and MM-OptBench (Li et al., 2026a). They evaluate generated models against reference answers or formulations through execution, objective-value agreement, or formulation similarity, but do not annotate hallucinations for detector development and evaluation.

Error Diagnosis and Structural Verification. Closest to our task are efforts that diagnose modeling errors or verify structural correctness beyond conventional metrics. Refai & Ahmed (2025) compare recovered variables, constraints, and mathematical expressions with reference formulations, but do not define taxonomy-grounded detection over complete (P, M, S) tuples. Zhang (2025) classifies attribute, syntax, and logical errors in generated code and studies mitigation, rather than detection across complete tuples. ORGEval and EquivaMap test symbolic-formulation equivalence through graph isomorphism or semantic mappings (Wang et al., 2026; Zhai et al., 2025), not consistency among descriptions, formulations, and implementations. Falsification-based verification perturbs quantities such as capacities or costs and tests whether models behave as their stated roles require, certifying some errors without reference formulations (Li & Hai, 2026); its guarantees cover only faults exposed by these tests. To our knowledge, none provides a fine-grained hallucination taxonomy for optimization modeling that spans objective, variable, constraint, and implementation failures, nor a unified task for diagnosing and localizing these failures over complete (P, M, S) tuples.

3 THE RELIABILITY GAP IN AUTOMATIC OPTIMIZATION MODELING

Table 1: Reference-assisted audit of 16 optimization-modeling systems. Objective-Value Match reports agreement with the reference value (App. B.1). Full-Correct reports tuples labeled structurally correct; Hallucination Ratio reports those labeled erroneous. Hallucination within Matches reports errors among objective-matched outputs. These updated audit labels differ from the historical detector-evaluation labels; Gemini-3.1-Pro annotations remain provisional (App. B.4.1).

Class	Model	Param. Scale	Objective-Value Match	Full-Correct	Hallucination Ratio	Hallucination within Matches
General open-weight	DeepSeek-v3.2	685B	92.1% (446/484)	85.7% (415/484)	14.3% (69/484)	7.8% (35/446)
	GLM-4.5-air	106B total / 12B active	84.1% (407/484)	69.4% (336/484)	30.6% (148/484)	17.9% (73/407)
	Kimi-K2.5	1T total / 32B active	92.1% (446/484)	85.5% (414/484)	14.5% (70/484)	9.0% (40/446)
	MiniMax-M2.5	229B	88.0% (426/484)	84.3% (408/484)	15.7% (76/484)	11.0% (47/426)
	Qwen3-4b	4.0B	16.1% (78/484)	13.6% (66/484)	86.4% (418/484)	23.1% (18/78)
	Qwen3-8b	8.0B	78.5% (380/484)	72.1% (349/484)	27.9% (135/484)	8.9% (34/380)
	Qwen3-32b	32.0B	72.1% (349/484)	53.5% (259/484)	46.5% (225/484)	26.4% (92/349)
General proprietary	Claude-Sonnet-4.6	undisclosed	92.4% (447/484)	84.9% (411/484)	15.1% (73/484)	8.7% (39/447)
	Gemini-2.5-flash-lite	undisclosed	82.0% (397/484)	67.6% (327/484)	32.4% (157/484)	18.6% (74/397)
	Gemini-3.1-Pro	undisclosed	94.6% (458/484)	88.6% (429/484)	11.4% (55/484)	7.0% (32/458)
	GPT-5.4	undisclosed	95.2% (461/484)	90.3% (437/484)	9.7% (47/484)	5.6% (26/461)
	Qwen3.5-Plus	undisclosed	91.7% (444/484)	85.5% (414/484)	14.5% (70/484)	7.7% (34/444)
Fine-tuned models	LLMOPT-Qwen2.5-14B	14.0B	77.9% (377/484)	65.9% (319/484)	34.1% (165/484)	20.7% (78/377)
	OptMATH-Qwen2.5-7B	7.0B	82.6% (400/484)	70.9% (343/484)	29.1% (141/484)	16.0% (64/400)
	ORLM-LLaMA-3-8B	8.0B	67.8% (328/484)	50.4% (244/484)	49.6% (240/484)	27.1% (89/328)
	SIRL-Qwen2.5-7B	7.0B	85.5% (414/484)	72.1% (349/484)	27.9% (135/484)	16.7% (69/414)

We begin by quantifying the reliability gap between objective-value agreement and structural correctness. Specifically, we ask how often matching the reference objective value hides errors in generated formulations and solver code. To answer this question, we audit the symbolic formulations and solver implementations generated by 16 LLM-based optimization-modeling systems for the same set of 484 benchmark problems. Full structural correctness requires both the formulation and implementation to faithfully represent the original problem. The panel includes twelve general-purpose models and four models fine-tuned for optimization modeling. We make this assessment using a reference formulation and implementation verified by human experts. We record whether the complete artifact is correct and, if not, whether it contains errors in the objective, decision variables, constraints, or solver implementation (see App. Table 6 for the category-level breakdown). Details of the problem set, model panel, generation settings, and annotation procedure appear in Apps. B.2.1 and B.4.1.

Objective-value matching overstates correctness. Table 1 shows that objective-value matching overstates structural correctness for every system. The gap remains even for the systems with the highest full-correct rates: among outputs with matching objective values, 5.6% from GPT-5.4, 7.0% from Gemini-3.1-Pro, and 8.7% from Claude-Sonnet-4.6 still contain hallucinations. For several other systems, the corresponding rate is substantially higher, reaching 18.6% for Gemini-2.5-flash-lite, 26.4% for Qwen3-32b, and 27.1% for ORLM-LLaMA-3-8B. Thus, objective-value matching alone is not a reliable criterion for evaluating the correctness of formulations and solver code.

Different systems fail in different ways. App. Table 6 reports error rates across the four major hallucination categories: objective, decision-variable, constraint, and solver-implementation errors. Two patterns emerge. First, objective errors are rare: the highest rate is 7.2% for ORLM-LLaMA-3-8B, while all other systems are at or below 2.7%. Second, the dominant error type varies by system. Implementation errors dominate Qwen3-4b at 82.0%, variable errors dominate Qwen3-32b at 38.8%, and constraint errors dominate ORLM-LLaMA-3-8B at 33.7%. These findings motivate detectors that inspect all four components, localize errors, and distinguish genuine errors from formulations that differ in form but encode the same problem.

4 DEFINITION AND TAXONOMY OF HALLUCINATIONS

Definition and semantic boundary. We consider three artifacts: a natural-language problem description P, a symbolic model M, and a solver implementation S. Let $M = (O, V, C, A)$, where O is the objective, V the decision variables, C the constraints, and A auxiliary objects such as parameters, sets, and indices. We write the complete tuple as $G = (P, M, S)$. Let $\mathfrak{M}(P)$ denote the set of formulations that faithfully encode the decision semantics of P, allowing different notation and equivalent reformulations. Let $\mathfrak{S}(M)$ analogously denote implementations that faithfully realize M.

We define

$$\mathcal{R}_{\text{sym}}(P, M) = \mathbf{1}[M \in \mathfrak{M}(P)], \qquad \mathcal{R}_{\text{imp}}(M, S) = \mathbf{1}[S \in \mathfrak{S}(M)].$$

Overall structural consistency is $\mathcal{R}(P, M, S) = \mathcal{R}_{\text{sym}}(P, M) \wedge \mathcal{R}_{\text{imp}}(M, S)$.

Definition 1 (Optimization-modeling hallucination)**.** An *optimization-modeling hallucination* is an evidence-grounded inconsistency in $G = (P, M, S)$ that makes $\mathcal{R}(P, M, S) = 0$ by changing or leaving undefined at least one intended decision, the feasible-decision set, the preference ordering or objective value assigned to feasible decisions, or the solver code's behavior or reported result.

The definition is semantic rather than syntactic or purely behavioral. A valid finding must provide a *semantic witness* showing how the intended optimization problem has changed. Matching the reference objective value on one instance is not sufficient: a formulation may omit a currently non-binding constraint or relax an integer variable without changing that value, and executable solver code may implement a different model. Conversely, variable renaming, reordered constraints, auxiliary variables, solver-specific sign transformations, and more substantial equivalent reformulations are not hallucinations when they preserve the intended decisions, feasibility, and objective.

Changes that affect only style or efficiency also fall outside the definition. Redundant variables, unbroken symmetry, and unexploited sparsity may warrant attention, but without a semantic witness they are quality warnings rather than hallucinations. The same boundary applies to modeling choices such as Big-M: a valid but loose constant is a warning, whereas an invalid value that removes intended feasible decisions is a hallucination. App. C.4 evaluates this boundary using paired equivalent and minimally non-equivalent reformulations.

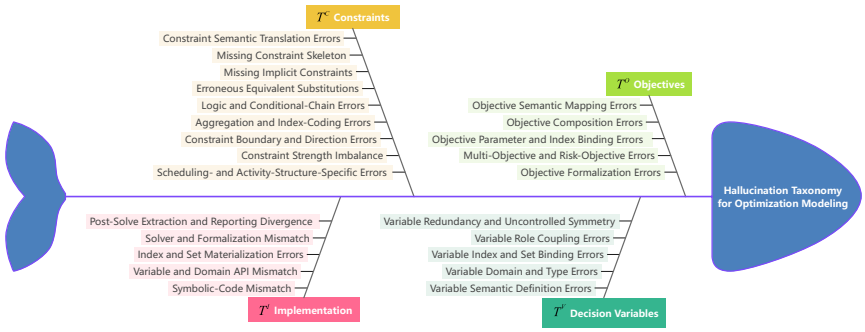


Figure 2: Compact view of the optimization-modeling hallucination taxonomy. The four major families comprise 24 subcategories and 83 specific entries. Table 18 lists every entry, and App. D documents their definitions and decision boundaries.

A fine-grained taxonomy for optimization modeling. To make this definition operational, we develop, to our knowledge, the first fine-grained hallucination taxonomy specifically for optimization modeling through OR-expert analysis and iterative refinement. The taxonomy has three levels: major families identify the part of the modeling process in which semantics first change, subcategories group related failure mechanisms, and specific entries provide labels for annotation and detection. The four major families cover the objective ($\mathcal{T}^O$), decision variables ($\mathcal{T}^V$), constraints ($\mathcal{T}^C$), and solver implementation ($\mathcal{T}^I$). We treat them as disjoint and write $\mathcal{T} = \mathcal{T}^O \sqcup \mathcal{T}^V \sqcup \mathcal{T}^C \sqcup \mathcal{T}^I$.

The hierarchy contains 24 subcategories and 83 specific entries: 18 objective, 18 decision-variable, 31 constraint, and 16 implementation entries. Of these, 80 describe potential semantic errors and three are unconditional quality warnings. Let $\mathcal{W}$ denote the warning set and $\mathcal{T}_H = \mathcal{T} \setminus \mathcal{W}$ the semantic label space. The taxonomy is a label space, not a checklist: assigning any label in $\mathcal{T}_H$ still requires case-specific evidence that the intended optimization semantics have changed.

Four major families. *1) Objective failures (18 entries).* These alter what the model optimizes or how it represents preferences, covering semantic mapping, objective composition, parameter/index binding, multi-objective and risk formulations, and piecewise, nonlinear, nonsmooth, or fractional objectives. *2) Decision-variable failures (18 entries).* These concern variable meanings and dimensions, domains and bounds, index sets, and coupling among variable roles. Redundancy and symmetry entries include the three quality warnings excluded from $\mathcal{T}_H$. *3) Constraint failures (31 entries).* These alter

the feasible region through mistranslated rules or quantifiers, missing explicit or implicit structure, invalid substitutions, faulty logic or Big-M constructions, aggregation/indexing errors, incorrect boundary conditions, inequality directions, or constraint strength, and scheduling-specific mistakes. *4) Implementation failures (*16 *entries).* These occur when solver code misrepresents the symbolic model: objectives, variables, or constraints may be omitted or altered; API domains, bounds, index expansion, solver selection, or linearization may be incorrect; or solutions may be extracted or reported incorrectly. Table 18 lists the catalog; App. D gives each subcategory's decision boundary and one representative entry with its definition, erroneous form, example, and consequence.

5 Hallucination Detection: Task, Data, and Metrics

Formal task definition. For a completed tuple $G = (P, M, S)$, let E^O, E^V, E^C, and E^I denote the objective, decision-variable, constraint, and implementation elements to be audited. These sets include both elements present in the generated artifacts and elements required by P whose absence may itself be an error. Let $m(e) \in \{O, V, C, I\}$ denote the family of element e, and $\mathcal{T}^{m(e)}$ the set of taxonomy labels in that family. Given the semantic taxonomy $\mathcal{T}_H$ from Sec. 4, we define the element set, admissible element–label pairs, and gold hallucination set as

$$E(G) = E^O \sqcup E^V \sqcup E^C \sqcup E^I, \tag{1}$$

$$\mathcal{U}(G) = \{(e, t) : e \in E(G),\ t \in \mathcal{T}_H \cap \mathcal{T}^{m(e)}\}, \tag{2}$$

$$\mathcal{H}(G) = \{(e, t) \in \mathcal{U}(G) : y(e, t) = 1\}, \tag{3}$$

where $y(e, t) = 1$ only when the evidence establishes that element e exhibits hallucination type t. A detector maps G to a possibly empty ranked list $\widehat{\mathcal{H}}(G)$ of pairs in $\mathcal{U}(G)$ and provides supporting evidence for each prediction. Thus, the task jointly requires artifact-level detection, localization of the affected element, and classification at the major-family, subcategory, and specific-type levels. The empty list represents a prediction of no hallucination. App. B.1 specifies the element scope, false alarms and misses, and the relationship to objective-value agreement.

Development and test suites. We construct separate development and test suites from five NL2Opt benchmarks: IndustryOR, MamoComplexLP, MamoEasyLP, NL4Opt, and ReSocratic. A *seed problem* is a natural-language optimization problem paired with a human-validated reference formulation and solver implementation. For each seed, we retain the reference tuple and collect one unedited formulation and solver program from each of 13 modeling systems as natural cases. We also create controlled cases using taxonomy-guided mutations, recording the target type and location; construction limitations are detailed in App. B.3.1.

Table 2: Composition and use of development and test suites with disjoint seed problems.

Suite	Seed problems	Reference	Controlled	Natural	Use
Development	200	200	1383	2600	Detector design and configuration
Test	484	484	1266	6292	Final evaluation

Together, the two suites contain 12,225 instances, with no seed problem shared between them. We use the development suite to design and configure the main detectors, fixing their settings before test evaluation. Full construction and annotation protocols appear in App. B; the anonymized code and data are available at `https://anonymous.4open.science/r/OptArgus-FA44`.

Test sets and metrics. The test suite contains three test sets; metric abbreviations in parentheses match Table 3. *1) Reference-clean set.* The 484 tuples carry clean labels (App. B.2.1). `EmptyReportRate` (*Empty*) is the fraction of empty reports; `MeanFindings_clean` (*Avg.FP*) counts findings per artifact, not a false-positive rate. *2) Controlled set.* This set contains 1266 cases, each with one designated taxonomy error or warning and a recorded target location. `Top-1 MajorCategoryHit` (*Major@1*), `Top-1 SubcategoryHit` (*Sub@1*), and `Top-1 SpecificTypeHit` (*Type@1*) measure whether the first finding matches the introduced issue's label at the corresponding taxonomy level. `MeanFindings` (*Avg.Find.*) is the average number of findings per artifact. *3) Natural set.* This set contains 6292 unedited outputs generated by 13 modeling systems on the 484 test seed problems. Because errors may overlap, we use archived artifact- and major-category labels (App. B.4.1). All natural-set columns report F1: `Halluc-F1` (*Halluc*) measures artifact-level error detection; `Objective-F1`, `Variable-F1`, `Constraint-F1`, and

`Implementation-F1` (*Obj*, *Var*, *Con*, and *Impl*, respectively) measure category-level detection; and `Macro-F1` (*Macro*) and `Micro-F1` (*Micro*) summarize performance across the four categories. We do not report natural subtype F1 because the annotations do not support that resolution. The three sets are complementary: the reference-clean set measures reporting restraint, the controlled set measures precise localization, and the natural set tests open-ended errors in generated artifacts. No single score captures all three requirements. Appendices B.2.2, B.3.2, and B.4.2 define the metrics for the reference-clean, controlled, and natural sets, respectively. Appendix C.6 discusses evaluation limitations.

6 Design of Reference Detectors

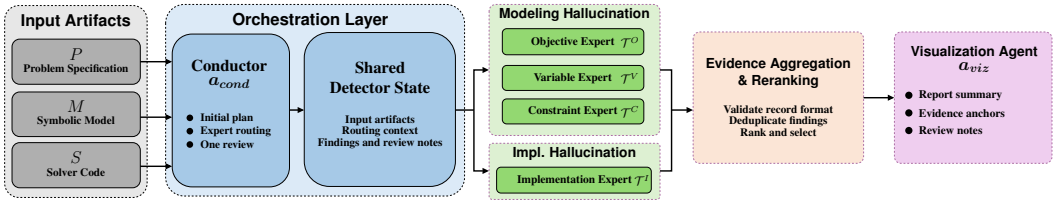


Figure 3: OptArgus workflow. A conductor selects among objective, variable, constraint, and implementation specialists. Selected specialists return structured findings, which the conductor reviews once for overlaps and dependencies. Deterministic aggregation and reranking select the final findings; the visualization agent presents them with evidence and review notes. App. A details scout-assisted routing and reporting (Fig. 4).

Optimization-modeling hallucination detection spans three representations of the same decision problem: the problem description P states the business intent, the symbolic model M formalizes it, and the solver code S implements it. Objective, variable, and constraint failures are primarily exposed by comparing P with M, whereas implementation failures require comparing M with S. These judgments may interact: a suspicious objective term may stem from an incorrectly defined variable, while an apparent code defect may reproduce an upstream modeling error.

This suggests a simple division of labor: reviewers handle distinct checks, and their findings are reviewed together for cross-component dependencies. We build **OptArgus** around this multi-agent design (Guo et al., 2024). As summarized in Fig. 3, a conductor reads the audit tuple and selects among four specialists for objective, variable, constraint, and implementation errors. Selected specialists return structured findings with taxonomy labels and supporting evidence, and the conductor reviews them for dependencies and overlaps. OptArgus deduplicates and ranks the findings into a short diagnosis, returning an empty report if none is sufficiently supported. A visualization agent presents the diagnosis and evidence; agent roles, routing, and prompts are detailed in App. A.

OptArgus is one way to organize the audit. We develop five additional detectors to explore lower-cost review, repeated verification, and alternative uses of specialist roles and taxonomy guidance. Single-Agent serves as a low-cost option; for the other four, we select fixed call schedules on the development suite to target token use close to OptArgus. Different kinds of review consume different numbers of tokens, so comparable budgets need not imply equal call counts. On a paired evaluation panel, their mean token use ranges from 0.895–$1.058\times$ that of OptArgus (App. Table 13).

1) Single-Agent Detector. This detector makes one call that receives (P, M, S) and the taxonomy, checks all four error categories, and directly returns the structured report. *2) Multi-Pass Self-Consistency.* It repeats that same complete audit five times, then merges, deduplicates, and ranks the filtered findings. *3) Independent Specialist Fan-Out.* This detector runs the objective, variable, constraint, and implementation specialists once each, then assigns three additional specialist passes using a fixed, content-independent schedule, for seven calls in total. The specialists do not exchange outputs; their findings are merged deterministically. *4) Generic Taxonomy-Aware Verifier.* This detector performs five full-tuple audits with fixed complementary inspection orders. Every pass sees the taxonomy and checks all four categories, but none has a specialist role; the five outputs are then merged. *5) Generic Taxonomy-Blind Verifier.* This detector separates discovery from classification:

seven calls search (P, M, S) for supported issues without seeing the taxonomy; a subsequent label-only call is instructed to map at most two surviving issues to taxonomy labels without adding claims. These are complete detector architectures with different organizing principles; the component studies in Sec. 7 instead examine changes within OptArgus. App. C.1 compares all six architectures (Table 9) and gives their full protocols and prompts.

7 Experimental Evaluation

Research questions. Building on the semantic definition and taxonomy, we use six reference detectors to examine what the proposed benchmarks reveal about hallucination detection. **RQ1.** How often do detectors flag reference-clean artifacts? **RQ2.** How precisely can they diagnose errors within the taxonomy? **RQ3.** Does success on controlled errors transfer to natural outputs? **RQ4.** Which OptArgus components support accurate, concise diagnoses, and at what cost?

Experimental setup. We evaluate the six detectors on the three test sets (Sec. 5) with DeepSeek-V3.2, identical (P, M, S) inputs, and a common ranked output format. Settings for this main comparison are selected on the development suite and fixed before testing. Table 3 includes four OptArgus variants and all 1266 controlled cases. App. B.3.3 reports warning-excluded scores and paired confidence intervals; App. B.4.3 gives natural-set breakdowns by generator and problem source. App. C covers detector protocols, ablations, efficiency, and backbone sensitivity.

Table 3: Results on the three test sets; metric abbreviations follow Sec. 5. Blue rows show the six reference detectors; gray rows show OptArgus variants. Within blue rows, bold and underlining mark each column's best and second-best values. App. B.4.3 gives natural-set breakdowns by generator and source; App. C covers detector protocols, ablations, efficiency, and backbone sensitivity.

Backbone	Method	Reference-clean		Controlled catalog				Natural benchmark (F1 ↑)						
		Empty ↑	Avg.FP ↓	Major@1 ↑	Sub@1 ↑	Type@1 ↑	Avg.Find. ↓	Halluc	Macro	Micro	Obj	Var	Con	Impl
DeepSeek (V3.2)	Single-Agent	0.483	0.556	0.723	0.431	0.339	2.151	0.521	0.382	0.379	0.404	0.328	0.504	0.294
	Multi-pass self-consistency	0.915	0.103	0.691	0.421	0.367	2.384	0.609	0.526	0.528	0.496	0.478	0.648	0.481
	Independent specialist fan-out	**0.946**	**0.064**	0.639	0.390	0.356	2.061	0.514	0.470	0.421	**0.602**	0.355	0.591	0.334
	Generic taxonomy-aware verifier	0.895	0.147	0.721	0.457	0.401	2.714	**0.645**	**0.554**	**0.563**	0.511	**0.544**	**0.664**	**0.497**
	Generic taxonomy-blind verifier	0.742	0.269	0.717	0.400	0.358	**1.191**	0.594	0.454	0.462	0.411	0.397	0.582	0.427
	OptArgus	0.853	0.159	**0.767**	**0.473**	**0.403**	1.224	0.617	0.512	0.486	0.541	0.495	0.579	0.431
	w/o specialist calibration	0.791	0.219	0.569	0.320	0.228	1.136	0.605	0.373	0.466	0.020	0.474	0.583	0.417
	w/o routing rescue	0.829	0.182	0.672	0.385	0.327	1.186	0.598	0.469	0.449	0.473	0.427	0.562	0.415
	w/o final rerank	0.636	0.438	0.636	0.395	0.341	2.617	0.591	0.513	0.477	0.556	0.449	0.613	0.435
	with all experts	0.880	0.122	0.788	0.479	0.416	1.184	0.617	0.499	0.484	0.504	0.501	0.546	0.444

RQ1: Reporting restraint on reference-clean artifacts. *All five multi-call detectors leave more reference-clean artifacts unflagged than Single-Agent.* On the 484 reference-clean artifacts, Single-Agent returns empty reports in 48.3% of cases, compared with 74.2%–94.6% for the multi-call designs (Table 3). Independent specialists are the most restrained at 94.6%, followed by multi-pass self-consistency at 91.5%; OptArgus returns empty reports in 85.3% of cases. The number of findings tells a consistent story: Single-Agent reports 0.556 findings per artifact, compared with 0.064 for independent specialists and 0.159 for OptArgus. Thus, additional review calls need not produce more concerns: all five multi-call designs return both fewer nonempty reports and fewer findings on this set. This is an important requirement for a detector intended to support human review. Each reported concern requires attention, so evaluation should measure not only whether a detector finds errors, but also when it refrains from reporting them. The reference-clean set makes this behavior measurable, with separate metrics for the frequency of nonempty reports and the number of findings. Restraint alone is not sufficient, however: an always-empty detector would perform well here while missing every error. RQ2 and RQ3 therefore examine the complementary requirements of precise diagnosis and natural-error detection.

RQ2: Diagnosing errors precisely. *OptArgus leads the six detectors at all three taxonomy levels while returning relatively few findings.* On the 1266 controlled cases, its Major@1, Sub@1, and Type@1 scores are 0.767, 0.473, and 0.403, respectively, compared with 0.723, 0.431, and 0.339 for Single-Agent (Table 3). The taxonomy-aware verifier nearly matches OptArgus's Type@1 (0.401), but returns 2.714 findings per case rather than 1.224. Thus, a longer list of findings need not provide a more accurate top-ranked diagnosis. Specific-type diagnosis remains harder than category recognition:

OptArgus identifies the specific type in only 40.3% of cases with its top-ranked finding. A detector may flag a constraint issue without distinguishing a missing requirement from an incorrectly expressed one. Our taxonomy makes this distinction explicit, and the controlled benchmark tests it by scoring the same top-ranked finding at each level. Specific labels help reviewers focus their inspection, while each additional finding requires attention. The benchmark therefore measures both the accuracy of the top-ranked label and the number of findings to review.

RQ3: Detecting natural errors. *The taxonomy-aware verifier leads all three aggregate F1 scores on natural outputs.* On the 6292 unedited outputs, its Macro-F1 is 0.554, compared with 0.526 for multi-pass self-consistency and 0.512 for OptArgus. It also achieves the highest Halluc-F1 (0.645) and Micro-F1 (0.563) among the six detectors (Table 3). At the category level, it leads on variable, constraint, and implementation errors, while independent specialists achieve the highest Objective-F1 (0.602). The overall ranking also differs from RQ2, where OptArgus leads at all three taxonomy levels. The two settings test complementary abilities. A controlled case asks whether the top-ranked finding matches the label of one known change. A natural output may contain several errors across different categories, so evaluation checks both whether an artifact contains errors and which major categories are affected. Together with RQ1, these results show why all three test sets are needed: reference-clean artifacts measure when detectors refrain from reporting issues, controlled cases measure diagnostic specificity, and natural outputs measure detection across potentially overlapping error categories. Evaluating these abilities separately reveals differences that a single aggregate score would hide.

RQ4: Understanding OptArgus's design. *Specialist guidance and final selection improve OptArgus's controlled diagnosis.* Whereas RQ1–RQ3 compare complete detectors, we use OptArgus's modular workflow to examine specialist guidance, routing, and final selection within one architecture. These ablations help explain its strong controlled diagnosis and concise reports. Removing specialist calibration—the additional branch-specific instructions—lowers controlled Type@1 from 0.403 to 0.228, even with the same agent roles and base prompts. Removing final reranking lowers Type@1 to 0.341 and more than doubles the findings per controlled case, from 1.224 to 2.617 (Table 3). Final selection also reduces findings on reference-clean artifacts, although natural Macro-F1 is nearly unchanged without it. Calling every expert brings a different trade-off. The All Experts variant raises controlled Type@1 to 0.416, but does not improve any aggregate natural-set F1 score and uses 22.9% more tokens than routed OptArgus on the paired efficiency panel (App. Table 13). Within the evaluated OptArgus configuration, these results highlight three design choices: how specialists are guided, which findings are retained, and whether additional calls justify their token cost.

Discussion. These results connect the paper's contributions: the semantic definition specifies what constitutes an optimization-modeling hallucination, the taxonomy distinguishes levels of diagnosis, and the complementary benchmarks make different detector failures observable. The reference detectors instantiate this task and reveal open challenges; no single design resolves all of them. Their current role is to guide inspection, not certify correctness. A supplementary paired-reformulation test examines whether OptArgus leaves equivalent formulations unflagged while flagging their minimally non-equivalent counterparts (App. C.4). In an exploratory downstream study, a repairer uses OptArgus's reports to produce executable candidates, demonstrating how structured diagnoses can be integrated into a repair workflow (App. C.5).

8 Conclusion

Matching a reference objective value can conceal semantic errors in LLM-generated optimization models, as our audit of 16 modeling systems demonstrates. We turn this reliability gap into a structured detection task over problem specifications, mathematical formulations, and solver code, supported by a fine-grained taxonomy and seed-disjoint development and test suites containing more than $12{,}000$ instances. We develop six reference detectors, showing that avoiding false alarms, localizing controlled errors, and detecting natural errors are distinct capabilities. Among them, OptArgus combines leading controlled-localization scores with concise reports, but no detector is best across all three settings. Distinguishing valid reformulations from genuine errors remains a key challenge. Together, these resources provide a common basis for developing and evaluating detectors that check whether generated formulations and code preserve the intended optimization problem.

AI USE STATEMENT

Generative AI systems are core experimental subjects and components of this work: they are used to generate optimization-modeling artifacts for the natural benchmark and as the backbones of the evaluated detectors. Generative AI tools were also used to polish the manuscript's language. Experimental artifacts and detector outputs were evaluated using the procedures described in the paper and appendix. The authors reviewed the language edits and take responsibility for the final content of this work.

ETHICS STATEMENT

The main ethical risk is over-reliance on an imperfect auditing system in consequential decision-support settings. False negatives may allow structurally incorrect optimization models to enter downstream workflows, while false positives may waste expert effort or reduce trust in valid models. The reference detectors are intended as auditing aids rather than standalone correctness certificates. Applications involving private or sensitive operational data should use appropriate privacy, security, and access-control safeguards.

REPRODUCIBILITY STATEMENT

We describe OptArgus's implementation and representative prompts in App. A, dataset construction, annotation, and evaluation metrics in App. B, and detector protocols and additional analyses in App. C. Anonymized code and data are available at `https://anonymous.4open.science/r/OptArgus-FA44`. The repository includes the complete taxonomy, frozen prompts, and manifests recording source identifiers, annotation rationales, and human-validation status.

REFERENCES

Ali Ahmaditeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Scalable optimization modeling with (mi) lp solvers and large language models. In *International Conference on Machine Learning*, pp. 577–596. PMLR, 2024.

Michael R Bartolacci, Larry J LeBlanc, Yasanur Kayikci, and Thomas A Grossman. Optimization modeling for logistics: options and implementations. *Journal of Business Logistics*, 33(2):118–127, 2012.

Johannes Bisschop. *AIMMS optimization modeling*. Lulu. com, 2006.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed rl: Grounding large language models for authentic optimization modeling. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Yitian Chen, Cheng Cheng, Yinan Sun, Zi Ling, and Dongdong Ge. OPT-Engine: Benchmarking the limits of LLMs in optimization modeling via complexity scaling. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. arXiv:2601.19924.

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

Ibrahim Dincer, Marc A Rosen, and Pouria Ahmadi. *Optimization of energy systems*. John Wiley & Sons, 2017.

Zhenan Fan, Bissan Ghaddar, Xinglu Wang, Linzi Xing, Yong Zhang, and Zirui Zhou. Artificial intelligence for optimization: Unleashing the potential of parameter generation, model formulation, and solution methods. *European Journal of Operational Research*, 332(1):1–30, 2026. doi: 10.1016/j.ejor.2025.08.029.

Daniel J Garcia and Fengqi You. Supply chain design and optimization: Challenges and opportunities. *Computers & Chemical Engineering*, 81:153–170, 2015.

Konstantinos Gkiotsalitis. *Public transport optimization*. Springer, 2022.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: a survey of progress and challenges. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pp. 8048–8057, 2024.

William E. Hart, Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, and John D. Siirola. *Pyomo—Optimization Modeling in Python*, volume 67. Springer, second edition, 2017.

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.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55, 2025b.

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, 2025c.

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.

Haifeng Li and Mo Hai. Falsification-based verification of LLM-generated optimization models: Sound test batteries and their detection limits. *arXiv preprint arXiv:2607.16646*, 2026.

Ruosen Li, Ziming Luo, and Xinya Du. FG-PRM: Fine-grained hallucination detection and mitigation in language model mathematical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 4247–4278. Association for Computational Linguistics, November 2025.

Zhong Li, Qi Huang, Yuxuan Zhu, Mohammad Mohammadi Amiri, Niki van Stein, Thomas Bäck, Matthijs van Leeuwen, Zaiwen Wen, and Lincen Yang. MM-OptBench: A solver-grounded benchmark for multimodal optimization modeling. *arXiv preprint arXiv:2605.12154*, 2026a.

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*, 2026b. URL https://openreview.net/forum?id=0Q7adIsv6p.

Haoyang Liu, Jie Wang, Boxuan Niu, Xiongwei Han, Yian Xu, Mingxuan Ye, Zijie Geng, Fangzhou Zhu, Tao Zhong, Mingxuan Yuan, and Jianye Hao. Opt-Verifier: Unleashing the power of LLMs for optimization modeling via dual-side verification. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. arXiv:2605.29556.

MingShan Liu and Jialing Fang. Enhancing mathematical reasoning in large language models with self-consistency-based hallucination detection. *arXiv preprint arXiv:2504.09440*, 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 *International Conference on Machine Learning*, pp. 40769–40802. PMLR, 2025.

Hongliang Lu, Zhong Li, Yuxuan Chen, Yuan Lan, Fan Zhang, and Zaiwen Wen. PEARL: Solver-in-the-loop interactive optimization modeling from natural language. *arXiv preprint arXiv:2607.18256*, 2026.

Mahdi Mostajabdaveh, Timothy T. Yu, Rindranirina Ramamonjison, Giuseppe Carenini, Zirui Zhou, and Yong Zhang. Optimization modeling and verification from problem specifications using a multi-agent multi-stage LLM framework. *INFOR: Information Systems and Operational Research*, 62(4):599–617, 2024. doi: 10.1080/03155986.2024.2381306.

Abdur Rais and Ana Viana. Operations research in healthcare: a survey. *International transactions in operational research*, 18(1):1–31, 2011.

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 Marco Ciccone, Gustavo Stolovitzky, and Jacob Albrecht (eds.), *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pp. 189–203. PMLR, 28 Nov–09 Dec 2022. URL https://proceedings.mlr.press/v220/ramamonjison23a.html.

Dania Refai and Moataz Ahmed. Peering inside the black box: Uncovering llm errors in optimization modelling through component-level evaluation. *arXiv preprint arXiv:2510.16943*, 2025.

Chao Shen, Zihan Guo, Xu Wan, Zhenghao Yang, Yifan Zhang, Wengi Huang, Jie Song, Zongyan Zhang, and Mingyang Sun. Proopf: Benchmarking and improving llms for professional-grade power systems optimization modeling. *arXiv preprint arXiv:2602.03070*, 2026.

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, 2024. URL https://openreview.net/forum?id=KD9F5Ap878.

Zhuohan Wang, Ziwei Zhu, Ziniu Li, Congliang Chen, Zhihang Lin, MingZhe Yang, Yizhou Han, Yufeng Lin, Angyang Gu, Xinglin Hu, Ruoyu Sun, and Tian Ding. Certified evaluation for LLMs in optimization modeling: From graph isomorphism to formulation isomorphism. In *ICML 2026 Workshop on Combining Theory and Benchmarks: Towards A Virtuous Cycle to Understand and Guarantee Foundation Model Performance*, 2026. URL https://openreview.net/forum?id=jEudMU2Ppd.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, et al. Chain-of-experts: When llms meet complex operations research problems. In *The twelfth international conference on learning representations*, 2024.

Ziyang Xiao, Jingrong Xie, Lilin Xu, Shisi Guan, Jingyan Zhu, Xiongwei Han, Xiaojin Fu, WingYin Yu, Han Wu, Wei Shi, et al. A survey of optimization modeling meets llms: progress and future directions. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pp. 10742–10750, 2025.

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.

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.

Guoyun Zhang. Large language model enabled mathematical modeling. *arXiv preprint arXiv:2510.19895*, 2025.

Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. Siren's song in the ai ocean: A survey on hallucination in large language models. *Computational Linguistics*, 51(4):1373–1418, 2025.

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. StepORLM: A self-evolving framework with generative process supervision for operations research language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ZrgxU8WMmG.

Appendix for *Hallucinations in LLM-Based Optimization Modeling: Definition, Taxonomy, Detection Benchmarks, and Reference Detectors*

TABLE OF CONTENTS

A OptArgus Design and Prompt Templates
- A.1 Design Rationale
- A.2 Inputs and Shared State
- A.3 Agent Roles and Responsibilities
- A.4 Specialist Routing and Review
- A.5 Aggregation, Reranking, and Abstention
- A.6 Inference Procedure
- A.7 Representative Prompts and Full Archive

B Benchmark Construction and Evaluation Metrics
- B.1 Audit Scope and Objective-Value Agreement
- B.2 Reference-Clean Set
  - B.2.1 Construction and Validation
  - B.2.2 Metrics
- B.3 Controlled Set
  - B.3.1 Construction, Coverage, and Validation
  - B.3.2 Metrics
  - B.3.3 Uncertainty and Warning Exclusion
- B.4 Natural Set
  - B.4.1 Generators, Annotation, and Label Versions
  - B.4.2 Metrics
  - B.4.3 Results by Generator and Problem Source

C Detector Protocols and Additional Analyses
- C.1 Multi-Call Reference Detectors
- C.2 Component Ablations and Inference Cost
- C.3 Alternative-Backbone Evaluation
- C.4 Paired Semantic-Equivalence Stress Test
- C.5 Exploratory Diagnosis-Guided Repair
- C.6 Limitations and Scope

D Complete Optimization-Modeling Hallucination Taxonomy
- D.1 Objective-Function Modeling Hallucinations
- D.2 Decision-Variable Modeling Hallucinations
- D.3 Constraint Modeling Hallucinations
- D.4 Implementation Hallucinations

A OPTARGUS DESIGN AND PROMPT TEMPLATES

This appendix describes the main DeepSeek-V3.2 OptArgus design introduced in Sec. 6. Figure 4 summarizes its main components, and the following subsections explain the shared state, agent roles, routing, evidence consolidation, and inference procedure. App. A.7 includes two representative prompt templates; the complete frozen prompts and source code are provided in the anonymous artifact.

A.1 DESIGN RATIONALE

OptArgus is organized around three design choices.

Match specialists to error families. Objective, variable, and constraint errors require comparing the problem description P with the symbolic model M, whereas implementation errors concern whether the solver code S faithfully realizes M. OptArgus assigns these checks to four specialists aligned

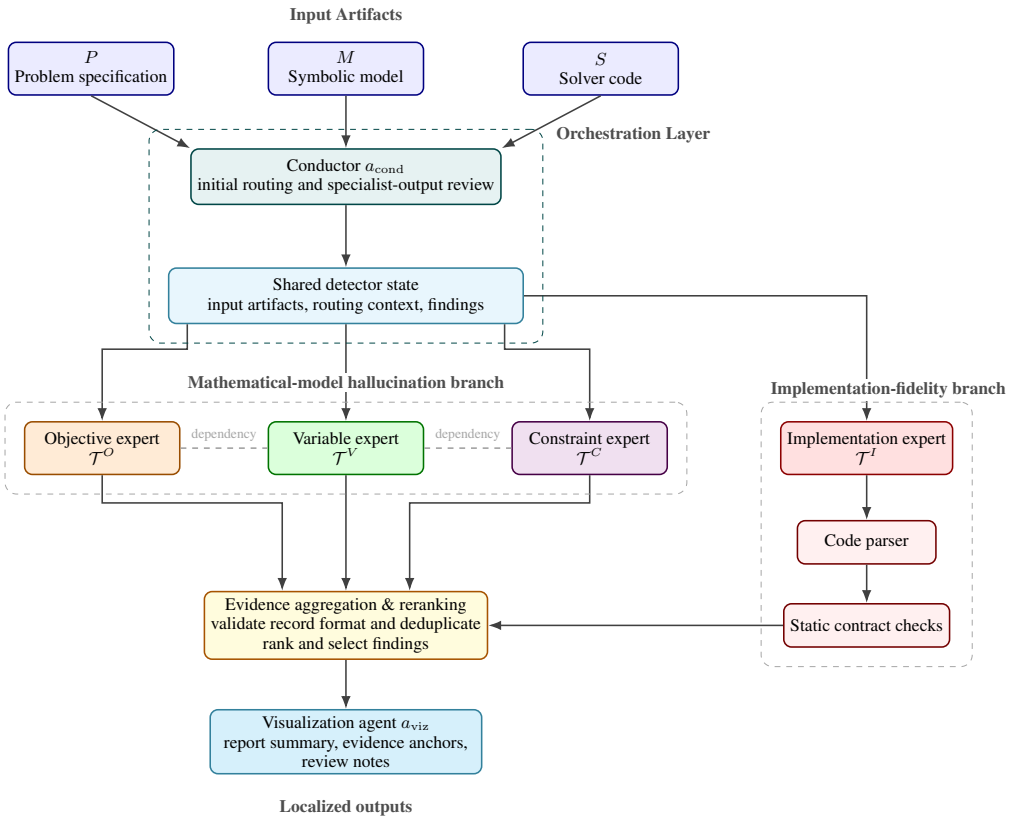


Figure 4: Component-level view of OptArgus. The conductor coordinates specialist selection and review through shared state. Objective, variable, and constraint specialists compare the symbolic model with the problem description using the taxonomy in App. D. The implementation branch combines symbolic-to-code comparison with static code-contract checks; it does not execute the solver program. Dashed links illustrate dependencies considered during conductor review, not direct communication between specialists. Deterministic aggregation and reranking produce the final findings, which the visualization node presents with their supporting evidence.

with the taxonomy. Each specialist proposes labels supported by evidence from its input artifacts; the taxonomy guides the search but does not itself establish that an error exists.

Select relevant specialists and review their findings. Routing combines the conductor's initial plan with a preliminary audit (the scout) and fixed heuristic checks, which can add branches omitted from the first plan. After the selected specialists run, the conductor reviews their outputs for overlaps and cross-component dependencies. This review runs once; it does not trigger another round of specialist calls. Selective routing limits model calls but may miss useful checks. App. C.2 compares this cost–coverage trade-off with calling all four experts.

Consolidate evidence before reporting. Specialist outputs are candidates, not separate final decisions. The implementation branch also supplies static code-contract evidence, such as syntax errors or invalid API references. Deterministic aggregation and reranking validate the candidates, remove duplicates, and select a short list of findings. These rules are fixed before test evaluation. An empty report means that no finding was retained, not that the tuple has been proved correct.

A.2 Inputs and Shared State

OptArgus keeps the input tuple and intermediate audit results in a shared state. Each stage reads the information it needs and records outputs for later stages. The problem description P, symbolic model M, and solver code S remain available as text; the evaluated implementation does not convert them into explicit symbolic, code, or cross-artifact dependency graphs.

Inputs and audit plan. The conductor reads (P, M, S) and produces a structured semantic summary of the intended objective, entities, parameters, index sets, hard requirements, soft preferences, implementation-sensitive details, and assumptions. This summary guides the audit but is an LLM-generated interpretation, not a verified representation of the problem. The state also records the selected specialists, routing rationale, and audit focus. Fixed heuristic checks and a preliminary full-tuple audit, called the scout, can add specialists before their calls begin. The scout's findings are retained as candidate concerns for further checking.

Specialist context and findings. The objective, variable, and constraint specialists receive P and M, together with the conductor's summary, routing guidance, and selected scout hypotheses. The implementation specialist receives M and S, routing guidance, and scout hypotheses, but not the raw problem description or the conductor's semantic summary. Its guidance can nevertheless contain information derived from P. Specialists do not receive one another's outputs. Any findings they return include proposed taxonomy labels, support scores, and evidence anchors, and are collected in the shared state.

Review and reporting. After the specialists finish, the conductor reviews their combined outputs alongside (P, M, S) and the semantic summary, recording overlaps and possible cross-component dependencies. This review occurs once and does not trigger further specialist calls. Aggregation checks that findings follow the output schema and merges them with candidates admitted by fixed rules or recovered from the scout. Deterministic reranking then selects the final list using these candidates, routing decisions, and scout findings. The review notes are preserved for reporting but are not inputs to this reranker. The state retains the intermediate candidates, final findings, and report, so the diagnosis can be traced back to its supporting evidence.

A.3 Agent Roles and Responsibilities

OptArgus uses LLMs to inspect artifacts and fixed rules to select the final findings. The conductor plans and reviews the audit, the scout suggests possible issues, and four specialists check different error families. The visualization agent summarizes the selected findings for the report.

Conductor. The conductor reads (P, M, S), summarizes the intended problem, and selects relevant specialists. Fixed heuristic checks and a separate scout call can add specialists before inspection begins. After the selected specialists finish, one conductor call reviews their combined outputs for overlaps and dependencies. Its notes are stored for reporting; they do not trigger further specialist calls.

Objective expert. The objective expert compares the objective in M with the goal stated in P. It checks the optimization direction, terms, coefficients, and index ranges, as well as whether the formula expresses the intended optimization goal. Checking how the code implements that objective is the implementation specialist's responsibility.

Variable expert. The variable expert checks what each decision variable represents, its domain, and its indices. Its prompt distinguishes quantities that may be continuous, such as divisible allocations or mixtures, from decisions that require integer or binary values, such as trips, facility choices, or assignments. The prompt asks for evidence from the problem description before treating a domain choice as an error; generic wording such as "how many" is not sufficient by itself.

Constraint expert. The constraint expert checks whether M expresses the requirements in P. It looks for missing relationships, incorrect aggregation, omitted policy rules, boundary errors, and faulty logical encodings. Its prompt distinguishes, for example, a requirement on a pooled total from separate requirements on each object, and a missing capacity relationship from an incorrect initial or terminal condition.

Implementation branch. The implementation specialist compares S with M, checking objective direction, constraint translation, and the construction of indexed variables and constraints. Separate static checks flag certain API errors, placeholder code, and Python syntax errors as additional candidates. Prompt guidance and filtering rules seek to avoid flagging equivalent code conventions, such as minimizing the negative of a maximization objective or expressing bounds in different ways. These checks inspect source code; they do not execute the solver program or certify its runtime behavior.

Final selection. Unlike the LLM-based reviewers, the final selector uses fixed rules. Aggregation and reranking merge candidates, suppress redundant findings, and return an empty list if none is retained. Conductor-review notes accompany the report but do not enter the reranker (App. A.2).

Visualization agent. The final LLM call summarizes the selected findings with a headline, an overview, and suggested review or repair priorities. The report renderer combines this summary with the selected findings, taxonomy labels, support scores, evidence anchors, and conductor-review notes. This stage presents the findings; it does not change the selected list.

A.4 Specialist Routing and Review

OptArgus selects specialists before detailed inspection begins. The conductor proposes an initial selection from (P, M, S), and fixed heuristic checks can add specialists. If this selection is empty, all four are included. The scout then performs a preliminary audit and can add specialists when its proposed error types and support scores meet fixed criteria. Each selected specialist runs once. Selective routing can reduce calls, but may omit a useful specialist check.

Routing cues suggest what to inspect; they do not establish that an error exists. For example, different objective directions in P and M can trigger an objective check, while a continuous domain for explicitly indivisible decisions can trigger a variable check. Differences in objective direction between M and S, suspicious API references, or placeholder code can trigger an implementation check. Mentions of pooled totals provide guidance on distinguishing shared requirements from separate per-object bounds; they do not by themselves activate the constraint expert. The specialists are instructed to interpret these cues in context and allow valid reformulations, such as implementing maximization by minimizing a negated objective.

After inspection, the conductor reviews the specialist outputs in a single call. The review looks for related concerns, such as an objective finding that depends on a variable definition or a code finding that reflects an error already present in M. These observations are saved as report notes. They neither trigger further specialist calls nor enter the final reranker. Final selection instead follows the fixed aggregation and ranking rules described in App. A.5.

A.5 Aggregation, Reranking, and Abstention

OptArgus first aggregates candidate findings, then selects and ranks those to include in the final report. Each specialist candidate identifies a potentially problematic element and provides taxonomy labels, a proposed verdict, a support score, and evidence. Its core fields are summarized as

$$f = (e, b, s, t, v, c, \mathcal{Z}), \tag{4}$$

where e is the affected element; b, s, and t are the major category, subcategory, and specific type; v is the proposed verdict; $c \in [0, 1]$ is the support score; and $\mathcal{Z}$ contains evidence anchors and explanations. Full records also include severity and metadata identifying proposed root causes and duplicates.

Aggregation checks that candidate records follow the required format and combines specialist outputs with enabled rule-based and scout-rescue candidates. It normalizes taxonomy labels, removes duplicates, and applies fixed filters to weakly supported findings or secondary symptoms of another issue. These checks organize and filter the candidate pool; validating a record's format does not establish that its diagnosis is correct.

The final reranker uses support scores, severity, routing context, specificity, and scout evidence to select a short, ordered list from the remaining candidates. Its rules are chosen on the development suite and frozen before test evaluation; conductor-review notes are not used in this selection. Reranking is distinct from *specialist calibration*, the branch-specific prompt guidance studied in App. C.2. Neither performs statistical probability calibration: the field `confidence` stores an agent-reported or rule-assigned support score, not a validated probability of correctness.

If no candidate is retained, OptArgus returns an empty finding list. This abstention from reporting an error is evaluated as a prediction of no hallucination, not as a separate uncertainty class or a guarantee that (P, M, S) is correct. The final-reranking ablation retains aggregation but bypasses this last selection stage, measuring its effect on false alarms, localization, and report length (App. C.2).

A.6 Inference Procedure

Algorithm 1 summarizes the evaluated workflow. Routing is completed before the selected specialists run, and their outputs receive one conductor review. The review produces reporting notes; deterministic aggregation and reranking select the findings. There is no loop that calls specialists again after review, and no solver execution during detection.

Algorithm 1 Inference procedure of OptArgus.

Require: Problem specification P, symbolic model M, solver code S, taxonomy $\mathcal{T}$
Ensure: Ranked finding list $F(P, M, S)$, empty when no candidate is retained

1: Initialize the shared state with the input texts (P, M, S)
2: Run the conductor to produce a semantic summary and select an initial set of specialists
3: Apply fixed heuristic checks and run the scout; supplement routing and retain scout hypotheses
4: **for all** selected specialists a **do**
5: Run expert a once and append its structured candidates to the raw pool $\mathcal{C}^{\text{raw}}$
6: **end for**
7: Run one conductor review of the specialist outputs and store its notes for reporting
8: Validate candidate schemas, add enabled rule-based and scout-rescue findings, and deduplicate the pool
9: Apply the frozen filtering and ranking rules using routing context and scout evidence (App. A.5)
10: Suppress derivative or overly generic findings and retain the final ranked list
11: Let $F(P, M, S)$ be the resulting ranked list; if no candidate survives, set $F(P, M, S) = \varnothing$
12: Emit the final auditable report with evidence anchors, taxonomy labels, support scores, and localized explanations

A.7 Representative Prompts and Full Archive

These two abridged examples summarize selected instructions for initial specialist routing and constraint auditing. They are not verbatim runtime prompts; the evaluated inputs are specified by the released templates and source code.

The anonymous artifact indexes Markdown templates for detection, generation, reference construction, and model-assisted annotation under `resources/prompts/`. The scout template, output schemas, and code-added specialist guidance are in `src/langgraph_detector/`. Baseline runners supply their output contracts and call schedules. The archived final-judge template is not used by the deterministic reranker.

Conductor Prompt (Abridged Summary)

```
You are the Conductor agent in a taxonomy-grounded multi-agent
hallucination detector for optimization modeling.

Your responsibilities:
1. Extract a faithful semantic schema from the natural-language problem.
2. Separate hard requirements from soft preferences.
3. Summarize the objective, entities, parameters, index sets, and
   implementation-sensitive details.
4. Propose an initial selection of specialist modules.
5. Produce routing guidance for the selected specialists.

Important rules:
- Do not invent missing business rules; record explicit assumptions when the
  statement is underspecified.
- Stay neutral: prepare the audit rather than deciding correctness.
- Route conservatively, normally to one or two modules rather than all four.
- Route to implementation when symbolic-to-code divergence is plausible.
- Route to variable for decision-object, dimension, domain, or type risks.
- Route to constraint for missing, mistranslated, boundary, share, policy,
  strength, or aggregation risks.
- Route to objective for direction, binding, omission, insertion, or
  composition risks.
- When text specifies pooled limits such as total, combined, or overall and
  the model may have localized them, flag aggregation-level risk.

Return only the fields required by the structured schema.
```

Constraint-Specialist Prompt (Abridged Summary)

```
You are the Constraint Expert in a taxonomy-grounded hallucination detector
for optimization modeling.

Taxonomy:
{taxonomy_block}

Audit instructions:
1. Compare textual rules with symbolic constraints.
2. Audit semantic translation, missing skeletons and implicit rules,
   logical chains, aggregation/indexing, boundaries, strength, and
   scheduling-specific structure.
3. Use the most specific supported subtype and quote short evidence spans.
4. Emit atomic root causes, not downstream paraphrases.
5. If the primary failure is in variable definition or solver-only code,
   record a dependency rather than forcing a constraint label.
6. Treat pooled totals and cardinality rules as aggregation structure; do not
   replace them with separate per-item bounds.
7. Do not demand an extra row when an equivalent constraint is already
   enforced through domains, bounds, or an algebraically equivalent encoding.
8. Do not report stylistic or robustness suggestions as hallucinations
   unless they create an actual semantic divergence.
9. Abstain when a concern rests only on an ambiguous or stricter reading,
   rather than a clear semantic contradiction.

Output instructions:
- Return at most {max_findings} findings, preferably zero to three.
- Provide direct textual or symbolic evidence and a concrete repair.
- Return only the fields required by the structured schema.
```

The inventory also explains prompt reuse across the comparison methods. Multi-pass and taxonomy-aware verification reuse the frozen Single-Agent template; independent specialists use the base expert templates without OptArgus's calibration addenda; taxonomy-blind verification uses discovery followed by a label-only mapper. Full protocols appear in App. C.1. The release manifest records file checksums for the templates and source used to reconstruct the evaluated inputs.

B Benchmark Construction and Evaluation Metrics

The detection task is defined in Sec. 5. Here we describe how the development and test suites are constructed, how their labels are validated, and how detector outputs are scored. The three test sets serve different purposes: reference-clean tuples measure reporting restraint, controlled cases test diagnosis of a single recorded mutation, and unedited model outputs test natural-error detection using archived labels (App. B.4.1). We report these scores separately because they measure complementary capabilities.

Development and test splits. We use 200 development seeds and 484 test seeds, with no shared seed problems or derived artifacts. Both splits draw from five sources. Their development/test seed counts are IndustryOR (20/10), MamoComplexLP (45/36), MamoEasyLP (45/229), NL4Opt (45/69), and ReSocratic (45/140).

The development suite contains 200 reference cases, 1383 controlled cases, and 2600 natural outputs from 13 generators, totaling 4183 instances. The reference cases comprise 198 ordinary clean cases and two ambiguity-stress cases. The latter are retained for separate diagnostic analysis and excluded from ordinary clean false-alarm scoring. The test-set composition is summarized in Table 2.

Detector configuration. We use the development suite to choose prompts, temperatures, output schemas, and inference budgets, as well as routing, aggregation, specialist-calibration, and reranking rules. The six main DeepSeek-V3.2 configurations are fixed before test evaluation and are not revised using test results. Sharing a taxonomy and source families across splits does not establish generalization beyond the evaluated distribution (App. C.6).

B.1 Audit Scope and Objective-Value Agreement

Audit scope. The element sets in Sec. 5 cover existing elements as well as required but missing ones. E^O, E^V, and E^C include objectives, variables, and constraints present in M, as well as

those required by P but missing from M. E^I similarly covers existing code elements and missing implementations of symbolic-model requirements. An omission means that required semantics are absent; using an equivalent formulation or implementation is not an error merely because it differs from the reference.

At the formal element–type level, a predicted pair outside $\mathcal{H}(G)$ is a false alarm, and a gold pair absent from $\widehat{\mathcal{H}}(G)$ is a miss. The reported metrics use the available annotation resolution: controlled Top-1 scores compare taxonomy labels, while natural-set F1 measures artifact- and major-category detection. Neither directly measures exact element–type recovery or independently validates a finding's explanation.

Objective-value agreement. This is a numerical check, not a semantic correctness test. Let $f(\widehat{S})$ be the recorded objective value for the generated implementation and $f(S^*)$ the audited reference value associated with reference implementation S^* for the same seed. For finite values, the numerical comparison uses

$$V_{\mathrm{obj}}(P, \widehat{M}, \widehat{S}) = \mathbf{1}\Big[\Big|f(\widehat{S}) - f(S^*)\Big| \le \max\Big\{\delta_{\mathrm{abs}},\, \delta_{\mathrm{rel}} \max\Big(|f(\widehat{S})|, |f(S^*)|\Big)\Big\}\Big] \tag{5}$$

as implemented by `math.isclose`. All 16 systems in Table 1 use $\delta_{\mathrm{abs}} = \delta_{\mathrm{rel}} = 10^{-6}$, applied to the recorded objective values and the corresponding audited reference values. Recorded execution failures and missing or non-finite objective values count as nonmatches. The rounding-tolerant repair diagnostic in App. C.5 uses a separate rule.

Why agreement is insufficient. A matching value establishes neither that $\widehat{M}$ faithfully represents P nor that $\widehat{S}$ faithfully implements $\widehat{M}$:

$$V_{\mathrm{obj}}(P, \widehat{M}, \widehat{S}) = 1 \;\not\Rightarrow\; \mathcal{R}_{\mathrm{sym}}(P, \widehat{M}) = 1, \tag{6}$$

$$V_{\mathrm{obj}}(P, \widehat{M}, \widehat{S}) = 1 \;\not\Rightarrow\; \mathcal{R}_{\mathrm{imp}}(\widehat{M}, \widehat{S}) = 1, \tag{7}$$

and therefore

$$V_{\mathrm{obj}}(P, \widehat{M}, \widehat{S}) = 1 \;\not\Rightarrow\; \mathcal{R}(P, \widehat{M}, \widehat{S}) = 1. \tag{8}$$

For example, weakening "exactly two" to "at most two" can admit unintended decisions even when the optimum still selects two. This violates the P-to-$\widehat{M}$ relation without changing the optimal value. Likewise, code can omit a nonredundant constraint from $\widehat{M}$ yet return the same optimum, violating the $\widehat{M}$-to-$\widehat{S}$ relation. Detection must therefore examine the complete tuple and identify a semantic inconsistency, rather than infer correctness from one matching value. Section 3 demonstrates this gap empirically across 16 systems.

B.2 Reference-Clean Set

The reference-clean set measures how often a detector returns findings on artifacts labeled as correct. Under these labels, the expected output is an empty finding list. The construction and version limitations below qualify its interpretation as a false-alarm test.

B.2.1 Construction and Validation

Sources and reference construction. The set draws on five benchmark families: IndustryOR (Huang et al., 2025a), MamoComplexLP (Huang et al., 2025c), MamoEasyLP (Huang et al., 2025c), NL4Opt (Ramamonjison et al., 2022), and ReSocratic (Yang et al., 2025). Each seed retains its problem statement, available source material, and provenance. Reference construction combines normalization, symbolic-model review, execution checks, targeted repairs, and adjudication of ambiguous cases. The intended checks are whether M represents P and whether S implements M, as defined in Sec. 4. Execution and objective-value agreement provide supporting evidence, not a proof of either relation (App. B.1). An adjudicated reference may differ from a source-reported answer when that answer conflicts with the problem statement.

Selection and version provenance. The historical pool contains 484 seeds: IndustryOR (10), MamoComplexLP (36), MamoEasyLP (229), NL4Opt (69), and ReSocratic (140). Its manifest records selection of cases receiving an overall-consistent verdict from an earlier detector, followed by deduplication by seed. Thus, this is not a detector-independent sample of correct artifacts. Archived clean-run manifests point to this pool, and the reported scores use its archived clean labels.

The released clean files retain the historical artifacts and labels. A later reference audit revised the formulations and implementations of five cases: three inventory-balance cases and two cases involving variable domains or activation bounds. These revisions are not incorporated into the released clean files. Moreover, the archived OptArgus clean reports do not contain input snapshots or hashes that would establish byte-for-byte identity between the released files and the inputs used in those runs. These selection and version limitations mean that a returned finding cannot automatically be treated as an independently confirmed false alarm.

B.2.2 METRICS

Let $\mathcal{C}_{\text{clean}} = \{1, \ldots, N_{\text{clean}}\}$ index the $N_{\text{clean}} = 484$ reference-clean cases, and let F_i be the final finding list after detector-specific post-processing. We report

$$\text{EmptyReportRate} = \frac{1}{N_{\text{clean}}} \sum_{i=1}^{N_{\text{clean}}} \mathbf{1}[F_i = \varnothing], \tag{9}$$

$$\text{MeanFindings}_{\text{clean}} = \frac{1}{N_{\text{clean}}} \sum_{i=1}^{N_{\text{clean}}} |F_i|. \tag{10}$$

EmptyReportRate is the fraction of cases with no finding; higher values indicate greater restraint. $\text{MeanFindings}_{\text{clean}}$ counts all returned findings per case; lower values indicate shorter reports. It is a count, not a probability, and can exceed one. Neither metric applies an additional confidence cutoff after the final list is produced. The two distinguish how often a detector raises a finding from how many findings it returns: detectors can flag the same number of cases yet produce different report lengths. Both scores are conditional on the archived clean labels and the limitations above. They do not measure recall on erroneous artifacts or establish robustness to equivalent reformulations.

B.3 CONTROLLED SET

The controlled set tests whether a detector can identify a designated change to an optimization artifact. Each case records the target's major category, subcategory, and specific type from the taxonomy in App. D. We compare the detector's first finding with these labels at all three levels. This design provides a known diagnostic target, but one assigned target does not guarantee that the artifact contains no other error. We describe construction and its limitations before defining the metrics.

B.3.1 CONSTRUCTION, COVERAGE, AND VALIDATION

Table 4: Controlled-set coverage of the 83-entry catalog, which includes quality warnings. "Full" means 30 constructed cases, "Partial" means 1–29, and "Not built" means no case under the released construction rules.

Major Category	Catalog Entries	Built Cases	Full	Partial	Not built
Objective	18	271	9	1	8
Variable	18	355	9	6	3
Constraint	31	257	5	13	13
Implementation	16	383	12	3	1
Total	83	1266	35	23	25

Source artifacts and target labels. Construction draws on the same 484 seed identifiers as the reference-clean set; the 1266 controlled cases use 275 distinct seeds. The pipeline combines previously constructed cases with new recipe-based mutations. Some base artifacts are recovered from earlier injected versions by reversing a previous change. Thus, sharing a seed identifier does not

imply that a controlled case was derived from the exact reference tuple currently released as clean (App. B.2.1). Each recipe assigns a target label and location; these labels come from construction, not from detector predictions.

Sampling and coverage. The target is up to 30 cases per catalog entry. Broadly applicable recipes, such as reversing the optimization direction or relaxing a discrete variable, use benchmark-family-stratified sampling with fixed-seed randomization within each family. Specialized recipes, such as Big-M or state–control changes, search for artifacts that meet their documented eligibility conditions. A shortfall therefore reflects these construction rules and the available artifacts, not an inherent inability to instantiate that error. Table 4 reports 1266 cases across 58 realized entries: 35 have 30 cases, 23 have fewer, and 25 catalog entries have no constructed case.

Quality warnings and reported scores. The main comparison in Table 3 uses all 1266 cases. Of these, 30 are labeled "Unnecessary Redundant Variables" and 3 "Failure to Exploit Sparsity," both quality warnings rather than semantic errors. The third unconditional warning, "Unhandled Symmetric Variables," has no constructed case. Table 5 separately reports scores after excluding these 33 warning-labeled cases, leaving 1233 cases across 56 types. This is a label-based exclusion, not an independent validation of every remaining case. For the mixed entry "Big-M Too Loose or Too Tight," the 10 constructed cases target overly tight bounds. A loose but valid encoding remains a warning unless evidence establishes a semantic change.

Validation limits. Construction records document recipe-specific checks and representative solver checks, but do not establish error isolation for every released case. For example, a ReSocratic seed `00085` case targets an activation bound reduced from 20 to 10, yet its symbolic model also assigns a raw-material coefficient of 6 where the problem specifies 3. This additional mismatch is not represented by the assigned Big-M label. Such residual errors can affect which issue a detector reports first. The scores therefore measure agreement with designated target labels, not exhaustive diagnosis of independently verified single-error artifacts.

B.3.2 METRICS

Top-1 scores measure whether the first finding matches the assigned target label at each taxonomy level. We also report the number of findings to distinguish label accuracy from report brevity. Neither measure independently verifies the finding's location or supporting explanation.

Let $\mathcal{C} = \{1, \ldots, N\}$ be the evaluated case set and let

$$F_i = (f_{i1}, \ldots, f_{i|F_i|})$$

be the final evaluated finding list for case i, in the order produced by the detector's output processing. Let b_i, s_i, and t_i be the target's major category, subcategory, and specific type. Writing $\mathrm{top}(F_i) = f_{i1}$ when $|F_i| > 0$, we report

$$\mathrm{Top1MajorCategoryHit} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[F_i \neq \varnothing \wedge \mathrm{branch}(\mathrm{top}(F_i)) = b_i], \tag{11}$$

$$\mathrm{Top1SubcategoryHit} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[F_i \neq \varnothing \wedge \mathrm{branch}(\mathrm{top}(F_i)) = b_i \wedge \mathrm{subcat}(\mathrm{top}(F_i)) = s_i], \tag{12}$$

$$\mathrm{Top1SpecificTypeHit} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[F_i \neq \varnothing \wedge \mathrm{branch}(\mathrm{top}(F_i)) = b_i \wedge \mathrm{type}(\mathrm{top}(F_i)) = t_i], \tag{13}$$

$$\mathrm{MeanFindings} = \frac{1}{N}\sum_{i=1}^{N} |F_i|. \tag{14}$$

The scorer compares the final labels directly, without reranking findings or applying an additional confidence threshold. Major@1 requires a matching major category; Sub@1 requires matching major-category and subcategory labels; Type@1 requires matching major-category and specific-type labels, without a separate subcategory check. An empty completed report scores zero on all three.

Each score lies in $[0, 1]$ and measures top-ranked label accuracy, not the precision of all findings or the correctness of their evidence. Operational failures are tracked separately from completed empty reports.

$\mathrm{MeanFindings}$ is the average length of the final finding list, with an empty report contributing zero. It counts findings, not words, false positives, or computational cost. Fewer findings indicate a shorter list, but are useful only alongside diagnostic accuracy: returning nothing minimizes this metric without identifying the target. Conversely, additional findings are not necessarily incorrect merely because they fall outside the single assigned label.

B.3.3 Uncertainty and Warning Exclusion

The paired OptArgus–Single-Agent comparisons here use $10{,}000$ bootstrap resamples. Clean cases are paired by seed; controlled cases are clustered by source seed, keeping all mutations from a sampled seed together. On the 484 reference-clean cases, the EmptyReportRate difference is $+0.370$ with 95% CI $[0.322, 0.419]$. Table 5 compares the full catalog with the warning-excluded subset. It also reports type-macro scores, which average within each realized type and then weight all types equally. These intervals quantify sampling uncertainty under the assigned labels, not uncertainty about label completeness or artifact validity.

Table 5: Controlled results with and without quality-warning cases. Parenthesized intervals are paired source-seed-clustered 95% CIs for OptArgus minus Single-Agent.

Slice	Method	MajorCategory	Subcategory	SpecificType
All catalog entries (n=1266)	Single-Agent	.723	.431	.339
	OptArgus	.767	.473	.403
	Difference	+.044 ([.017,.073])	+.042 ([.011,.072])	+.064 ([.036,.091])
Warnings excluded (n=1233)	Single-Agent	.718	.420	.325
	OptArgus	.765	.465	.394
	Difference	+.047 ([.018,.076])	+.045 ([.013,.075])	+.069 ([.041,.097])
Type-macro, warnings excluded (56 types)	Single-Agent	.738	.408	.322
	OptArgus	.785	.422	.351

B.4 Natural Set

The natural set evaluates errors in model-generated formulations and solver code, without injecting a designated defect. An output may be correct or contain several errors, so we evaluate both whether an error is present and which major categories are affected. Results measure agreement with the recorded annotations for the evaluated models and problems; they do not establish performance on unseen generators or problem families.

B.4.1 Generators, Annotation, and Label Versions

Generators. The descriptive audit covers 16 systems on the same 484 seed problems. Twelve are general-purpose models: seven open-weight systems—`DeepSeek-v3.2`, `GLM-4.5-air`, `Kimi-K2.5`, `MiniMax-M2.5`, `Qwen3-4b`, `Qwen3-8b`, and `Qwen3-32b`—and five proprietary systems—`Claude-Sonnet-4.6`, `Gemini-2.5-flash-lite`, `Gemini-3.1-Pro`, `GPT-5.4`, and `Qwen3.5-Plus`. Open-weight refers to checkpoint availability, not disclosure of training data or the full training pipeline. Four further systems are adapted to optimization modeling: `LLMOPT-Qwen2.5-14B` (Jiang et al., 2025), `OptMATH-Qwen2.5-7B` (Lu et al., 2025), `ORLM-LLaMA-3-8B` (Huang et al., 2025a), and `SIRL-Qwen2.5-7B` (Chen et al., 2025). These released models allow us to examine errors in outputs from optimization-adapted and general-purpose systems.

Descriptive audit versus detector evaluation. The descriptive audit contains $7744 = 16 \times 484$ outputs. Detector evaluation retains the original 13 generators: the nine general-purpose systems other than GPT-5.4, Claude-Sonnet-4.6, and Gemini-3.1-Pro, plus the four optimization-adapted systems. Its aggregate scores, confidence intervals, and natural-case efficiency measurements use the $6292 = 13 \times 484$-output pool. The three later extensions are descriptive only; they are not additional detector tests. The two analyses also use different label snapshots, as detailed below.

Extension generation. Each added system retains one independently generated output per seed, without reference or label inputs or iterative model-based repair. GPT-5.4 uses low reasoning effort; Claude-Sonnet-4.6 uses adaptive thinking at low effort and structured output; Gemini uses the `gemini-3.1-pro-preview` API model with low thinking and a native output schema. Five failed Claude requests were resubmitted after upstream timeouts. Discarded Gemini preflight requests are excluded; 39 retained Gemini outputs underwent representation-only normalization of index/bound serialization or framework-identifier casing. These collection steps do not constitute semantic repair.

Annotation policy. Reference-assisted model reviews compare each generated formulation and program with the problem and reference artifacts. They record an artifact-level correctness judgment, labels for objective, variable, constraint, and implementation errors, and a rationale. The policy attributes an error to the layer where it arises: code that merely reproduces an upstream symbolic error does not receive an additional implementation label. Distinct errors in both layers may receive both labels.

Label versions. Tables 1 and 6 use the later reference-consistent audit, whereas detector scores retain the historical evaluation labels. The overlapping generator–seed pairs therefore need not have identical labels. For example, the later audit completed 94 previously null GLM-4.5-Air records, of which 41 were positive, changing its positive count from 107 to 148. The historical evaluation conversion treated those null labels as negative. Thus, Table 7 retains 107, not 148. The reported detector scores have not been recomputed against the revised labels and remain subject to this label-version limitation.

Review status and scope. Model-assisted review and author approval are not equivalent to independent expert adjudication of every artifact. In particular, the retained Gemini-3.1-Pro audit records still mark its 484 labels as provisional pending review; its descriptive error rates should be read accordingly. We do not report independent double annotation or inter-rater agreement. Natural-set evaluation is limited to artifact and major-category labels, not exact subtypes. The model panel is not exhaustive, and its error rates are conditional on the recorded model versions, prompts, reference revisions, and seed problems.

Table 6 reports category-level error rates and counts from the updated descriptive audit. Categories can overlap within an artifact, so their counts need not sum to the "Any Hallucination" total. The general-purpose rows are split into open-weight and proprietary systems, followed by the optimization-adapted models.

Table 6: Major-category composition under the updated descriptive audit, distinct from the historical detector-evaluation labels. Percentages and counts use 484 artifacts per generator; Any Hallucination is the category union. Gemini-3.1-Pro annotations remain provisional (App. B.4.1). Rows follow Table 1.

Class	Model	Objective	Variable	Constraint	Implementation	Any Hallucination
General open-weight	`DeepSeek-v3.2`	0.4% (2/484)	10.1% (49/484)	3.7% (18/484)	0.0% (0/484)	14.3% (69/484)
	`GLM-4.5-air`	1.2% (6/484)	23.3% (113/484)	6.8% (33/484)	3.3% (16/484)	30.6% (148/484)
	`Kimi-K2.5`	0.2% (1/484)	8.7% (42/484)	5.0% (24/484)	1.2% (6/484)	14.5% (70/484)
	`MiniMax-M2.5`	0.4% (2/484)	9.1% (44/484)	5.8% (28/484)	0.6% (3/484)	15.7% (76/484)
	`Qwen3-4b`	2.7% (13/484)	19.0% (92/484)	8.3% (40/484)	82.0% (397/484)	86.4% (418/484)
	`Qwen3-8b`	1.0% (5/484)	9.7% (47/484)	4.1% (20/484)	16.5% (80/484)	27.9% (135/484)
	`Qwen3-32b`	0.8% (4/484)	38.8% (188/484)	3.9% (19/484)	12.2% (59/484)	46.5% (225/484)
General proprietary	`Claude-Sonnet-4.6`	0.2% (1/484)	12.6% (61/484)	2.3% (11/484)	0.0% (0/484)	15.1% (73/484)
	`Gemini-2.5-flash-lite`	1.0% (5/484)	26.4% (128/484)	6.6% (32/484)	4.5% (22/484)	32.4% (157/484)
	`Gemini-3.1-Pro`	0.8% (4/484)	7.4% (36/484)	3.1% (15/484)	0.0% (0/484)	11.4% (55/484)
	`GPT-5.4`	0.4% (2/484)	8.1% (39/484)	1.2% (6/484)	0.0% (0/484)	9.7% (47/484)
	`Qwen3.5-Plus`	0.0% (0/484)	11.6% (56/484)	3.1% (15/484)	0.0% (0/484)	14.5% (70/484)
Fine-tuned models	`LLMOPT-Qwen2.5-14B`	1.0% (5/484)	21.7% (105/484)	6.8% (33/484)	11.8% (57/484)	34.1% (165/484)
	`OptMATH-Qwen2.5-7B`	2.1% (10/484)	12.0% (58/484)	15.7% (76/484)	3.7% (18/484)	29.1% (141/484)
	`ORLM-LLaMA-3-8B`	7.2% (35/484)	18.6% (90/484)	33.7% (163/484)	2.3% (11/484)	49.6% (240/484)
	`SIRL-Qwen2.5-7B`	0.8% (4/484)	14.0% (68/484)	16.1% (78/484)	1.2% (6/484)	27.9% (135/484)

The category breakdown shows why an audit must look beyond the objective. Variable and constraint labels are more frequent than objective labels for most systems. For Claude-Sonnet-4.6, for example, 61 of the 73 artifacts labeled erroneous contain a variable error, while only one contains an objective error. The dominant category also varies by generator: implementation for `Qwen3-4b` (82.0%), variables for `Qwen3-32b` (38.8%), and constraints for `ORLM-LLaMA-3-8B` (33.7%). These are

category-level counts, not counts of artifacts with only that error. They motivate broad semantic checks; the separate objective-matching analysis in Table 1 establishes why a matching scalar value is insufficient.

The optimization-adapted systems also differ substantially in their error profiles. Neither adaptation to optimization modeling nor performance on one error category establishes reliability across the full modeling pipeline. These observations describe the evaluated systems under the recorded annotations, not all models in either class.

B.4.2 METRICS

Natural-set scoring asks two questions: does the output contain an error, and which major categories are affected? Because several categories can apply to one output, category detection is multilabel. We report artifact-level F1, F1 for each of the four categories, and macro- and micro-F1 across categories. All detector scores use the historical evaluation labels described above.

Formally, let $\mathcal{A} = \{1, \ldots, N_{\text{nat}}\}$ be the artifact set, and let

$$\mathcal{B} = \{\text{obj}, \text{var}, \text{con}, \text{impl}\} \tag{15}$$

denote the four major categories. For each artifact $i \in \mathcal{A}$ and category $b \in \mathcal{B}$, let $y_i^{(b)} \in \{0, 1\}$ be the archived evaluation label. The prediction $\hat{y}_i^{(b)}$ is one when the detector's final report contains a finding marked `hallucinated` in category b, and zero otherwise. Artifact-level labels and predictions are defined as

$$y_i^{(\text{hall})} = \mathbf{1}\left[\text{is_incorrect}_i \vee \sum_{b \in \mathcal{B}} y_i^{(b)} > 0\right], \qquad \hat{y}_i^{(\text{hall})} = \mathbf{1}\left[\sum_{b \in \mathcal{B}} \hat{y}_i^{(b)} > 0\right]. \tag{16}$$

Thus $y_i^{(\text{hall})} = 1$ whenever the reference annotation marks artifact i incorrect or assigns at least one positive category, and $\hat{y}_i^{(\text{hall})} = 1$ whenever the detector predicts at least one category. For any annotation channel $\ell \in \mathcal{B} \cup \{\text{hall}\}$, define

$$\text{TP}_\ell = \sum_{i=1}^{N_{\text{nat}}} \mathbf{1}[y_i^{(\ell)} = 1 \wedge \hat{y}_i^{(\ell)} = 1], \tag{17}$$

$$\text{FP}_\ell = \sum_{i=1}^{N_{\text{nat}}} \mathbf{1}[y_i^{(\ell)} = 0 \wedge \hat{y}_i^{(\ell)} = 1], \tag{18}$$

$$\text{FN}_\ell = \sum_{i=1}^{N_{\text{nat}}} \mathbf{1}[y_i^{(\ell)} = 1 \wedge \hat{y}_i^{(\ell)} = 0]. \tag{19}$$

The corresponding F1 score is

$$\text{F1}_\ell = \frac{2\,\text{TP}_\ell}{2\,\text{TP}_\ell + \text{FP}_\ell + \text{FN}_\ell}. \tag{20}$$

We report

$$\text{Halluc-F1} = \text{F1}_{\text{hall}}, \tag{21}$$

major-category-specific scores

$$\text{Objective-F1} = \text{F1}_{\text{obj}}, \tag{22}$$

$$\text{Variable-F1} = \text{F1}_{\text{var}}, \tag{23}$$

$$\text{Constraint-F1} = \text{F1}_{\text{con}}, \tag{24}$$

$$\text{Implementation-F1} = \text{F1}_{\text{impl}}, \tag{25}$$

and the aggregate metrics

$$\text{MajorCategoryMacro-F1} = \frac{1}{4}\sum_{b \in \mathcal{B}} \text{F1}_b, \tag{26}$$

$$\text{MajorCategoryMicro-F1} = \frac{2\sum_{b \in \mathcal{B}} \text{TP}_b}{2\sum_{b \in \mathcal{B}} \text{TP}_b + \sum_{b \in \mathcal{B}} \text{FP}_b + \sum_{b \in \mathcal{B}} \text{FN}_b}. \tag{27}$$

Halluc-F1 measures agreement on whether any error is present; category-specific F1 scores show which error families are detected. Macro-F1 gives each of the four categories equal weight. Micro-F1 pools their true positives, false positives, and false negatives before computing F1, so frequent categories have more influence. Neither aggregate is uniformly preferable: together they distinguish balanced category performance from performance dominated by common labels. When a precision, recall, or F1 denominator is zero, the score is set to zero; Macro-F1 still averages all four categories.

All scores lie in $[0, 1]$, with higher values indicating closer agreement with the archived labels. They do not validate exact error types, the evidence in individual findings, or generalization beyond the evaluated data.

B.4.3 Results by Generator and Problem Source

We compare OptArgus with Single-Agent by generator and problem source using the historical evaluation labels. Predictions are paired on the same artifacts. For confidence intervals, we draw $10{,}000$ bootstrap samples of the 484 source seeds, keeping all generator outputs from a sampled seed together. This preserves their within-seed dependence. The historical identifiers `qwen3_plus` and `qwen3_5_plus` are merged into one Qwen3.5-Plus group. Tables 7 and 8 report the breakdowns; we also examine sensitivity to excluding Qwen3-4B. These intervals reflect sampling uncertainty under the archived labels, not uncertainty from label revisions.

Table 7: Natural-set detector results by generator under the historical evaluation labels. Positive counts need not match the updated audit in Table 6. SA and OA denote Single-Agent and OptArgus. P, R, and F_1 are artifact-level precision, recall, and F1; Macro averages F1 over the four error categories. The final row averages each column equally over the 13 generators, rather than pooling their predictions.

Generator	Positive	SA F_1	OA F_1	Δ	OA P	OA R	SA Macro	OA Macro
General-purpose, open-weight								
`DeepSeek-v3.2`	91	0.202	0.405	0.203	0.478	0.352	0.142	0.222
`GLM-4.5-Air`	107	0.407	0.571	0.164	0.487	0.692	0.237	0.558
`Kimi-K2.5`	72	0.200	0.279	0.079	0.297	0.264	0.164	0.213
`MiniMax-M2.5`	100	0.178	0.335	0.158	0.352	0.320	0.149	0.273
`Qwen3-4B`	419	0.528	0.936	0.408	0.949	0.924	0.395	0.629
`Qwen3-8B`	135	0.610	0.695	0.085	0.726	0.667	0.391	0.494
`Qwen3-32B`	225	0.722	0.797	0.075	0.759	0.840	0.371	0.517
General-purpose, proprietary								
`Gemini-2.5-Flash-Lite`	156	0.510	0.658	0.149	0.644	0.673	0.298	0.424
`Qwen3.5-Plus`	98	0.170	0.422	0.251	0.448	0.398	0.062	0.164
Fine-tuned systems								
`LLMOPT-Qwen2.5-14B`	165	0.474	0.446	-0.028	0.525	0.388	0.285	0.318
`OptMATH-Qwen2.5-7B`	140	0.591	0.449	-0.142	0.289	1.000	0.483	0.402
`ORLM-LLaMA-3-8B`	240	0.768	0.771	0.003	0.900	0.675	0.490	0.467
`SIRL-Qwen2.5-7B`	136	0.472	0.439	-0.033	0.281	1.000	0.313	0.340
Macro over generators	–	0.449	0.554	0.106	0.549	0.630	0.291	0.386

Table 8: Natural-set detector results by problem source under the historical evaluation labels. Positive counts artifacts labeled as containing an error. SA and OA denote Single-Agent and OptArgus; Macro is major-category macro-F1.

Source family	N	Positive	SA F_1	OA F_1	Δ	OA Macro
IndustryOR	130	44	0.630	0.714	0.084	0.510
MamoComplexLP	468	314	0.336	0.548	0.212	0.476
MamoEasyLP	2977	656	0.579	0.583	0.005	0.480
NL4Opt	897	366	0.565	0.674	0.109	0.604
ReSocratic	1820	704	0.492	0.641	0.149	0.528

OptArgus's advantage is not uniform across generators. Its mean Halluc-F1 over the 13 systems is 0.554, versus 0.449 for Single-Agent, but it scores lower on LLMOPT, OptMATH, and SIRL and is nearly tied on ORLM. On OptMATH and SIRL, perfect recall comes with low precision (0.289 and 0.281), showing the cost of flagging many artifacts. Excluding Qwen3-4B reduces the pooled Halluc-F1 advantage from $+0.097$ to $+0.031$ (95% CI $[0.002, 0.061]$). The Micro-F1 difference becomes

$+0.024$ ($[-0.006, 0.053]$), while Implementation-F1 favors Single-Agent by 0.156 (OptArgus minus Single-Agent: -0.156, $[-0.216, -0.096]$). Thus, the aggregate gain depends partly on the generator mix and does not imply improvement on every source of natural errors.

C Detector Protocols and Additional Analyses

This appendix details the four multi-call alternatives to OptArgus and Single-Agent. It then examines OptArgus's components, inference cost, an alternative-backbone configuration, and behavior on equivalent and minimally non-equivalent formulations, followed by an exploratory diagnosis-guided repair study. Limitations appear at the end. The main DeepSeek-V3.2 protocols are configured on the development suite and frozen before test evaluation.

C.1 Multi-Call Reference Detectors

Table 9: Six reference-detector designs. All receive the same (P, M, S) tuple and return findings in a common format; their call schedules, intermediate outputs, and selection rules differ.

Detector	Review plan	Taxonomy use	Coordination and output
Single-Agent	One full-tuple audit	During audit	Direct structured report
Multi-Pass Self-Consistency	Five full-tuple audits with the same prompt	During audit	Filtered union; no agreement threshold
Specialist Fan-Out	Four specialists plus three fixed repeats	During audit	No interaction; category-preserving merge
Taxonomy-Aware Verifier	Five full-tuple audits with different inspection orders	During audit	Filtered union; no agreement threshold
Taxonomy-Blind Verifier	Seven discovery calls and one label call	Mapping only	Cross-pass support; shortlist of at most two issues
OptArgus	Initial plan, scout, and routed specialists	During audit	One conductor review; rule-based final selection

Shared protocol. No detector weights are trained or fine-tuned. Single-Agent and OptArgus use temperature zero; the four multi-call alternatives use the temperatures specified below. The four designs organize multiple LLM calls around the same detection task (Table 9). All use DeepSeek-V3.2 with thinking enabled and a per-call thinking cap of 4096 tokens. They receive the same (P, M, S) tuple and return a common final finding format, but use different intermediate schemas and output limits. No gold annotations or manual corrections are supplied during inference. Fixed filters check record structure, require evidence fields, and reject specified placeholder, hypothetical, or self-negating claims; these checks do not establish semantic correctness. Provider errors, invalid structured responses, or incomplete call schedules are detector-run failures, whereas valid empty reports remain scored outputs. Raw responses and provider traces are retained for audit.

Development settings and budget matching. Prompts, call schedules, and aggregation rules were selected using the 4183-artifact development suite. A common development panel was used to target token use close to OptArgus, rather than equal call counts. Development checks covered structured-output validity, call counts, and behavior on clean and erroneous examples. Settings were then frozen for testing. Budget matching is approximate; measured test-panel usage is reported in Table 13.

Multi-pass self-consistency. This detector repeats the frozen Single-Agent audit five times at temperature 0.2. Each call checks the complete tuple independently, without specialist roles or a conductor. Findings that pass the schema and rule-based checks are pooled, deduplicated, ordered by severity and support score, and capped at four. Despite the method name, aggregation uses a union rather than majority voting: a finding from one pass can enter the report. This design examines repeated full-tuple review without specialization.

Independent specialist fan-out. The four specialists each inspect every artifact once at temperature zero. Three receive one additional pass each, giving seven calls in total. They are selected by the three lowest SHA-256 values of `artifact_id:module:flat-cost-repeat-v1`, without using artifact content or predictions. Each pass receives its base specialist prompt and taxonomy, without OptArgus's calibration addenda, and is instructed to return at most one finding. There is no conductor, scout, or exchange of specialist outputs. A deterministic merge removes repeated findings and retains at most four, reserving one slot for each represented category before filling any remaining slots. This is specialist review without orchestration, not the All Experts ablation, which retains OptArgus's prompt guidance, conductor, and final selection rules.

Generic taxonomy-aware verifier. This detector also makes five independent full-tuple audits at temperature 0.2, using the frozen Single-Agent prompt and complete taxonomy. Unlike Multi-Pass, it adds a different inspection order to each call: objective-first, implementation-first, variable-first, constraint-first, or backward from code to model to problem. Every call still checks all four categories; none has a specialist role. The same rule-based filtering and union aggregation retain at most four findings, with no minimum cross-pass agreement requirement. The distinguishing feature is the complementary review order.

Generic taxonomy-blind verifier. Seven discovery calls at temperature zero combine end-to-end and component-focused reviews without receiving the taxonomy. Each is instructed to return at most one unlabeled issue. Fixed element- and text-similarity rules group recurring issues; only groups supported by at least two calls are eligible for a shortlist of at most two. An eighth, temperature-zero call maps these issues to taxonomy labels. It receives the shortlist and taxonomy, but not the original tuple, and is instructed to preserve the evidence without adding or strengthening claims. Post-processing checks source-issue references, taxonomy labels, and affected elements before deduplication; it does not prove that every mapped explanation preserves the source meaning. This design separates discovery from taxonomy-based labeling.

Scope of comparison. Single-Agent is a low-cost reference and is not budget matched to the multi-call designs. These methods compare complete review workflows: prompts, call schedules, and finding-selection rules vary together. Their results therefore do not isolate the effects of specialization, coordination, or taxonomy access. The component ablations below instead change individual parts of OptArgus. Frozen templates and prompt-assembly code are documented in App. A.7.

C.2 Component Ablations and Inference Cost

We compare the full OptArgus configuration with four variants that change specialist prompt guidance, routing, or final finding selection (Table 10). Here, *specialist calibration* means branch-specific prompt addenda, not probability calibration. We assess each change within the fixed OptArgus workflow, including its effects on information passed downstream. We then examine inference cost on a shared, paired evaluation panel.

Table 10: OptArgus ablation definitions. A checkmark marks a retained component; a dash marks removal or bypass. For All Experts, "n/a" refers only to adding omitted specialist roles: the scout and heuristic signals are still retained.

Variant	Specialist calibration	Selective routing	Routing rescue	Final reranking	Variant definition
OptArgus	✓	✓	✓	✓	Full system: calibrated specialists, selective routing with rescue, and deterministic final reranking.
OptArgus w/o Specialist Calibration	–	✓	✓	✓	Removes only the calibration addenda; retains base specialist templates, the taxonomy, and downstream rules.
OptArgus w/o Routing Rescue	✓	✓	–	✓	Skips heuristic routing checks and the scout call; the retained reranker also loses these signals.
OptArgus w/o Final Rerank	✓	✓	✓	–	Emits all post-aggregation findings in severity/confidence order; aggregation filtering and deduplication remain.
OptArgus w/ All Experts	✓	–	n/a	✓	Runs all four specialists; retains the scout, heuristic checks, and final reranking.

Specialist calibration. Removing the calibration addenda lowers controlled Type@1 from 0.403 to 0.228 and natural Objective-F1 from 0.541 to 0.020 (Table 11). The base specialist prompts still specify each role and provide taxonomy guidance, while routing, aggregation, and reranking rules remain unchanged. The comparison therefore concerns the additional branch-specific instructions, not specialization versus generic review. Removing these instructions affects categories differently: objective detection shows the largest natural-set decline, whereas Constraint-F1 slightly increases.

The uncalibrated variant also emits fewer findings per controlled case (1.136 vs. 1.224), but this reduction accompanies much weaker localization.

Table 11: Component ablations across reference-clean, full-catalog controlled (1266 cases, including warnings), and historical-label natural settings. Metric abbreviations follow Table 3. Parenthesized deltas are relative to OptArgus; blue and red indicate better and worse values under the metric direction, not statistical significance.

	Reference-clean		Controlled catalog				Natural (F1)						
Method	Empty ↑	Avg.FP ↓	Major@1 ↑	Sub@1 ↑	Type@1 ↑	Avg.Find. ↓	Halluc ↑	Macro ↑	Micro ↑	Obj ↑	Var ↑	Con ↑	Impl ↑
OptArgus	0.853	0.159	0.767	0.473	0.403	1.224	0.617	0.512	0.486	0.541	0.495	0.579	0.431
OptArgus w/o Specialist Calibration	0.791 (-0.062)	0.219 (+0.060)	0.569 (-0.198)	0.320 (-0.153)	0.228 (-0.175)	1.136 (-0.088)	0.605 (-0.012)	0.373 (-0.139)	0.466 (-0.020)	0.020 (-0.521)	0.474 (-0.021)	0.583 (+0.004)	0.417 (-0.014)
OptArgus w/o Routing Rescue	0.829 (-0.024)	0.182 (+0.023)	0.672 (-0.095)	0.385 (-0.088)	0.327 (-0.076)	1.186 (-0.038)	0.598 (-0.019)	0.469 (-0.043)	0.449 (-0.037)	0.473 (-0.068)	0.427 (-0.068)	0.562 (-0.017)	0.415 (-0.016)
OptArgus w/o Final Rerank	0.636 (-0.217)	0.438 (+0.279)	0.636 (-0.131)	0.395 (-0.078)	0.341 (-0.062)	2.617 (+1.393)	0.591 (-0.026)	0.513 (+0.001)	0.477 (-0.009)	0.556 (+0.015)	0.449 (-0.046)	0.613 (+0.034)	0.435 (+0.004)
OptArgus w/ All Experts	0.880 (+0.027)	0.122 (-0.037)	0.788 (+0.021)	0.479 (+0.006)	0.416 (+0.013)	1.184 (-0.040)	0.617	0.499 (-0.013)	0.484 (-0.002)	0.504 (-0.037)	0.501 (+0.006)	0.546 (-0.033)	0.444 (+0.013)

Routing rescue. Disabling heuristic routing checks and the scout lowers controlled Type@1 from 0.403 to 0.327 and natural Macro-F1 from 0.512 to 0.469. This variant retains specialist calibration and final reranking, but changes more than which specialists are selected: the reranker also receives neither scout findings nor heuristic routing notes. The scores therefore reflect the combined contribution of these supplementary checks and their downstream signals, rather than the isolated effect of adding omitted specialist branches.

Final reranking. Bypassing the final reranker retains aggregation, including its filtering and deduplication, and emits the remaining findings in severity/confidence order. Empty reports are still possible. Without this final selection stage, mean false findings per clean artifact rise from 0.159 to 0.438, and findings per controlled case more than double from 1.224 to 2.617. Controlled Type@1 also falls from 0.403 to 0.341. On natural outputs, however, Macro-F1 is nearly unchanged (0.513 vs. 0.512), while Halluc-F1 and Micro-F1 decline. The clearest benefit is therefore fewer false alarms and a more concise, better-ranked diagnosis list, not a uniform improvement in natural-set F1.

All-expert invocation. Running all four specialists improves the clean and controlled scores, including Type@1 (0.416 vs. 0.403). These gains do not carry over to aggregate natural-set F1: Halluc-F1 is unchanged, while Macro-F1 is lower (0.499 vs. 0.512) and Micro-F1 is nearly unchanged (0.484 vs. 0.486). All Experts retains the scout and heuristic signals, but no longer needs them to add specialist roles. It tests exhaustive expert invocation, not an upper bound on detection coverage. Calling every expert therefore does not yield a uniformly better detector; its additional inference cost is quantified below.

Inference cost and latency. Efficiency comparisons use a fixed, paired panel of 150 test artifacts (50 reference-clean, 50 controlled, and 50 natural), with the same artifacts evaluated by every method. All methods use DeepSeek-V3.2 through SiliconFlow, with thinking enabled and a 4096-token per-call thinking cap. The panel gives the three settings equal weight. Usage covers algorithmically successful attempts, excluding operational failures and duplicates, and is therefore not the total billed cost. Tables 12 and 13 report provider-measured tokens and distinguish inference work from elapsed time.

Table 12: Thinking-enabled efficiency of routed OptArgus and OptArgus w/ All Experts on the paired evaluation panel. Both variants use the same DeepSeek-V3.2 backbone, thinking cap, role prompts, and output contracts. "Specialist calls" counts the four expert roles; "LLM calls" includes all graph calls. Token columns report provider-measured per-artifact usage. "LLM sec" sums model-call latency, "Wall sec" is end-to-end runtime, and "P95 wall sec" is its 95th percentile. Lower is better; negative savings indicate an increase.

Method	Specialist calls / inst. ↓	LLM calls / inst. ↓	Input tok / inst. ↓	Reasoning tok / inst. ↓	Visible tok / inst. ↓	Total tok / inst. ↓	LLM sec / inst. ↓	Wall sec / inst. ↓	P95 wall sec ↓
OptArgus	2.740	6.773	17,252.0	4,527.9	1,547.0	23,327.0	356.97	300.66	508.92
OptArgus w/ All Experts	4.000	8.000	21,937.6	4,966.9	1,758.0	28,662.6	362.98	274.75	462.80
Savings of OptArgus vs. w/ All Experts	31.5%	15.3%	21.4%	8.8%	12.0%	18.6%	1.7%	−9.4%	−10.0%

Routed OptArgus invokes 2.740 specialists per artifact on average, compared with 4.000 for All Experts (Table 12). It uses 31.5% fewer specialist calls, 15.3% fewer total LLM calls, and 18.6% fewer total tokens (23,327.0 vs. 28,662.6). These savings do not translate into faster execution on this panel: mean wall time is 300.66 seconds for routed OptArgus versus 274.75 seconds for All

Experts, with P95 times of 508.92 and 462.80 seconds. Timing also depends on concurrency and provider load. The supported efficiency claim is lower call and token usage, not lower end-to-end latency.

Table 13 extends the comparison to all six reference detectors and All Experts on the same artifacts.

Table 13: Thinking-enabled efficiency on the paired evaluation panel. Calls and provider-measured token counts are means per artifact. Token ratios normalize mean total tokens to routed OptArgus on the same artifacts. Visible tokens exclude reasoning; total tokens comprise input plus provider-reported completion usage.

Method	Calls	Input	Reasoning	Visible	Total	Token ratio
Single-Agent	1.000	3,357.6	740.8	104.7	4,203.0	0.180×
OptArgus	6.773	17,252.0	4,527.9	1,547.0	23,327.0	1.000×
OptArgus w/ All Experts	8.000	21,937.6	4,966.9	1,758.0	28,662.6	1.229×
Multi-pass self-consistency	5.000	16,787.8	3,565.1	529.4	20,882.3	0.895×
Independent specialist fan-out	7.000	19,602.0	4,261.0	810.0	24,672.9	1.058×
Generic taxonomy-aware verifier	5.000	17,040.7	3,806.7	624.5	21,471.9	0.920×
Generic taxonomy-blind verifier	8.000	14,574.2	5,906.5	931.2	21,411.8	0.918×

The four other multi-call detectors use 0.895–1.058× OptArgus's total tokens, while Single-Agent uses 0.180× and All Experts 1.229×. Thus, the multi-call reference detectors have broadly comparable, but not identical, token budgets; Single-Agent is a substantially cheaper reference. These are comparisons of complete detector designs, not isolated effects of individual mechanisms.

Call counts alone do not determine token cost: taxonomy-blind verification makes more calls and uses more reasoning tokens than taxonomy-aware verification, yet uses slightly fewer total tokens because its inputs are shorter.

C.3 Alternative-Backbone Evaluation

We compare Single-Agent and OptArgus using Qwen3-Max-Preview on the same three test sets as the main DeepSeek-V3.2 experiments. Both detectors use the same Qwen provider route at temperature zero, with thinking disabled. They share the benchmark inputs, taxonomy, output schema, and evaluation metrics. This supplementary study compares two Qwen-based configurations; it does not repeat the full six-detector comparison.

The supplementary OptArgus configuration uses Qwen-specific post-processing selected after the initial evaluation: archived responses are reprocessed to include schema-valid scout findings before deterministic filtering and reranking, without additional LLM calls.

Table 14: Qwen3-Max-Preview results on the reference-clean set and full controlled catalog (including warnings). Metric abbreviations follow Table 3. Bold marks the better observed score; parenthesized differences are computed from displayed, rounded scores. Bold and color do not indicate statistical significance.

	Reference-clean (484 cases)		**Controlled catalog (1266 cases)**			
Method	Empty ↑	Avg.FP ↓	Major@1 ↑	Sub@1 ↑	Type@1 ↑	Avg.Find. ↓
Single-Agent	0.603	0.409	0.704	0.450	0.368	1.882
OptArgus	**0.727** (+0.124)	**0.285** (-0.124)	**0.716** (+0.012)	**0.467** (+0.017)	**0.369** (+0.001)	**1.420** (-0.462)

Table 15: Qwen3-Max-Preview results on the 6292 natural artifacts with historical evaluation labels. OptArgus and score annotations follow Table 14. Here, n counts gold-positive artifacts in each category; categories may overlap.

Method	Halluc-F1 ↑	Objective-F1 ↑ (n=92)	Variable-F1 ↑ (n=1126)	Constraint-F1 ↑ (n=571)	Implementation-F1 ↑ (n=675)	MajorCategory Macro-F1 ↑	MajorCategory Micro-F1 ↑
Single-Agent	0.495	0.578	0.300	0.519	0.251	0.412	0.365
OptArgus	**0.598** (+0.103)	**0.630** (+0.052)	**0.438** (+0.138)	**0.525** (+0.006)	**0.436** (+0.185)	**0.507** (+0.095)	**0.460** (+0.095)

Results and interpretation. With Qwen3-Max-Preview, OptArgus returns empty reports on 72.7% of reference-clean cases, compared with 60.3% for Single-Agent, and produces shorter reports

(Table 14). It also reports fewer findings per controlled case (1.420 vs. 1.882). However, controlled Type@1 is nearly tied (0.369 vs. 0.368): more concise reporting does not translate into a clear advantage in identifying the exact error type.

On natural outputs, Macro-F1 is higher (0.507 vs. 0.412), as are the other reported F1 scores (Table 15). The category-level differences vary, with the largest increase in implementation detection and the smallest in constraint detection. The clearest observed benefits are reporting restraint, report concision, and natural-error detection. These results compare the two evaluated Qwen configurations rather than isolate a backbone effect.

C.4 Paired Semantic-Equivalence Stress Test

Motivation and scope. Under our definition, a valid reformulation is not a hallucination merely because it differs from a reference formulation. A detector should accept such a rewrite while flagging a similar-looking version that changes the intended optimization problem. We test this distinction with matched pairs of equivalent and minimally non-equivalent formulations. This is a targeted diagnostic of OptArgus, not a fourth general-purpose benchmark or a comparison of detector architectures.

Paired construction. We use 64 solver-valid source seeds, disjoint from the 200 development seeds and 484 main test seeds: 23 MamoEasyLP, 27 ReSocratic, 13 NL4Opt, and one IndustryOR problem. Each seed yields two artifacts with the same natural-language problem. The equivalent member, E^+, preserves the original semantics; the non-equivalent member, E^-, introduces one controlled semantic defect into that rewrite. The term "near-equivalent" in the released records refers only to the small edit, not to approximate semantic equivalence. The 128 artifacts cover eight transformation families, with eight pairs per family. The implemented constructions are:

1. **Variable renaming:** a consistent variable permutation in E^+, versus an inconsistent coefficient or bound mapping in one renamed constraint in E^-.
2. **Sign/scaling:** multiplication of a constraint by -2 with the corresponding bound and direction changes, versus an incorrectly tightened transformed bound.
3. **Slack/auxiliary variables:** replacement of an inequality by an equality with a nonnegative slack variable, versus an incorrect positive lower bound on that slack.
4. **Aggregation:** addition of the sum of two inequalities while retaining both original constraints, versus replacing the originals with their sum and thereby relaxing the feasible set.
5. **Indicator/Big-M:** a Big-M encoding with its binary activation variable fixed to one, versus allowing the variable to take zero and relax the original constraint.
6. **Redundant constraints:** addition of a positively scaled duplicate constraint, versus tightening its bound so that it is no longer redundant.
7. **Solver materialization:** replacement of a variable bound by an equivalent explicit constraint, versus an incorrect endpoint in that constraint. Both versions are represented in the model and code; this is not a model–code mismatch.
8. **Objective decomposition:** decomposition into two components that sum to the original objective, versus omission, rescaling, or sign reversal of a coefficient during recomposition.

Each pair records a transformation-specific mapping or certificate for E^+ and a feasible-set counterexample or objective-preference reversal for E^-. For auxiliary-variable constructions, equivalence is defined after projection onto the original decision variables. Construction also checks that E^+ preserves the audited reference optimum, but this numerical check alone does not establish equivalence. A separate validation script checks file integrity and the presence of the recorded evidence, executes both solver programs, and compares their outputs with the construction records. It does not independently prove the recorded equivalence claims or re-verify the witnesses. All 64 pairs pass these checks before the reported evaluation; construction and validation scripts do not read detector predictions.

Evaluation and metrics. We evaluate the DeepSeek-V3.2 OptArgus reference configuration at temperature zero. Each artifact is processed separately: the detector receives (P, M, S) and its usual

taxonomy, but not the label, certificate, witness, or paired artifact. A positive prediction means that the final aggregated report contains at least one finding. Let $\widehat{y}_i^{+}$ and $\widehat{y}_i^{-}$ denote these binary predictions for pair i, with $n = 64$. The superscripts identify the two variants; E^-, not E^+, is the hallucination-positive member. We report

$$\text{EquivalentFPR} = \frac{1}{n}\sum_{i=1}^{n}\widehat{y}_i^{+}, \tag{28}$$

$$\text{NonEquivalentRecall} = \frac{1}{n}\sum_{i=1}^{n}\widehat{y}_i^{-}, \tag{29}$$

$$\text{PairBalancedAccuracy} = \frac{1}{2n}\sum_{i=1}^{n}\left[(1-\widehat{y}_i^{+}) + \widehat{y}_i^{-}\right]. \tag{30}$$

We also report the fraction of pairs with both decisions correct: no finding on E^+ and at least one finding on E^-. These are artifact-level detection metrics; they do not check whether a reported finding identifies the inserted defect. Overall 95% confidence intervals use $20{,}000$ nonparametric bootstrap resamples of the 64 intact seed pairs. Family-level values are descriptive because each family contains only eight pairs.

Table 16: Paired semantic-equivalence diagnostic for DeepSeek-V3.2 OptArgus. Equivalent FPR measures false alarms on E^+; non-equivalent recall measures detection of E^-. Pair-balanced accuracy averages the two correct-decision rates, whereas the final column requires both decisions in a pair to be correct. Each family has only eight pairs; its scores are descriptive, not robust rankings.

Transformation family	Pairs	Equivalent FPR ↓	Non-equivalent recall ↑	Pair-balanced accuracy ↑	Both members correct ↑
Aggregation	8	0.750	1.000	0.625	0.250
Indicator/Big-M	8	0.875	1.000	0.562	0.125
Objective decomposition	8	0.125	1.000	0.938	0.875
Redundant constraints	8	0.250	1.000	0.875	0.750
Sign/scaling	8	0.250	0.500	0.625	0.375
Slack/auxiliary variables	8	0.500	0.875	0.688	0.375
Solver materialization	8	0.125	0.875	0.875	0.750
Variable renaming	8	0.375	0.625	0.625	0.250
Overall	**64**	**0.406**	**0.859**	**0.727**	**0.469**

Results. Table 16 shows that detecting errors does not ensure acceptance of valid rewrites. OptArgus flags 85.9% of the non-equivalent artifacts (95% CI $[76.6\%, 93.8\%]$), but also flags 40.6% of the equivalent artifacts ($[28.1\%, 53.1\%]$). Pair-balanced accuracy is 72.7% ($[66.4\%, 78.9\%]$), yet both decisions are correct for only 46.9% of pairs. The latter measure exposes a distinction that recall alone misses: a detector can flag an erroneous rewrite while also rejecting its valid counterpart.

The family results illustrate two failure modes. For Indicator/Big-M and aggregation, all eight erroneous artifacts are flagged, but so are 87.5% and 75.0% of their valid counterparts. High recall therefore coexists with poor discrimination in these samples. Conversely, recall is only 0.500 for sign/scaling and 0.625 for variable renaming, showing that small semantic defects can also go unflagged. These observations describe the recorded decisions; they do not establish which internal reasoning step failed or a general ranking of transformation difficulty.

Interpretation and limitations. This diagnostic reinforces the distinction between raising a useful suspicion and establishing semantic incorrectness. OptArgus reports require further validation; neither a finding nor an empty report is a correctness certificate. The evidence is limited to one detector, eight pairs per family, an unbalanced source mix, and one controlled defect per erroneous artifact. In particular, the Indicator/Big-M cases test fixed-activation rewrites, not general Big-M reformulations. The paired intervals reflect variation over the sampled seeds, not uncertainty in the transformation design or recorded equivalence evidence. Broader conclusions require independently constructed pairs, more diverse reformulations, and additional detectors or dedicated equivalence-checking methods.

Inference cost. All 128 artifacts have successful final reports. Their recorded usage totals 790 LLM calls and 2,696,690 tokens, averaging 6.17 calls and 21,068 tokens per artifact. These figures describe this diagnostic evaluation and are separate from the cost-matching calibration in App. C.1.

C.5 Exploratory Diagnosis-Guided Repair

Question and scope. Can a separate LLM use OptArgus reports to produce revised formulations and solver code? We examine this downstream use in an exploratory study. OptArgus supplies diagnostic information; the repairer performs the edits. We report operational outcomes, not validated semantic repair success or an improvement over repair without diagnoses.

Cases and repair protocol. The panel contains 100 controlled cases, sampled with a fixed random seed within each branch: 25 objective, 25 variable, 25 constraint, and 25 implementation cases. It spans 73 source seeds, five source families (IndustryOR, MamoComplexLP, MamoEasyLP, NL4Opt, and ReSocratic), and 44 specific error types. Selection does not depend on whether the detector's diagnosis is correct.

We reuse archived input artifacts and diagnoses from an earlier OptArgus configuration, rather than regenerating reports with the main configuration evaluated in this paper. The repairer receives (P, M, S), the diagnostic and routing summaries, and the selected findings. No separate benchmark label, injection metadata, or clean target is supplied. However, some archived artifact texts retain construction annotations describing the injected change. Thus, the inputs cannot be treated as free of mutation hints.

A fixed DeepSeek-V3.2 repairer makes one formal request per case at temperature zero, with hidden reasoning disabled and a requested 4096-token output cap. The shared prompt asks for minimally changed, internally consistent replacements of M and S, treating the diagnosis as advice to verify rather than an oracle. A response is protocol-valid if it passes the required output-format and provider-contract checks; this does not establish that the repair is correct.

Automated checks. For protocol-valid responses, we check Python compilation and apply a static filter for disallowed imports and operations before execution in a separate process with time limits. The execution check requires the program to complete and return the expected dictionary; it does not certify solver optimality or semantic correctness. Reported objectives are compared with those of the executed reference programs. The archived agreement rule is rounding-tolerant: it uses relative and absolute tolerances of 10^{-6} and also accepts nearest-integer agreement. Normalized edit size is $1 - \mathrm{SequenceMatcher.ratio}$ on the concatenated original and revised (M, S) texts, not the literal fraction of characters changed.

Scoring subsets. Compilation and execution are reported over the 98 protocol-valid OptArgus-guided responses. Objective agreement uses the archived scorer's 97-case common-response subset: cases with valid responses under all three original conditions (no diagnosis, Single-Agent, and OptArgus). The denominator is therefore determined by response availability, not reference-replay failures. Only the OptArgus-guided condition is reported here; a failed repaired execution counts as disagreement. Edit size, tokens, and latency are averaged over the 98 valid responses. Tokens and latency cover the repair call only, excluding diagnosis generation.

Table 17: Operational outcomes of exploratory OptArgus-guided repair. Denominators are shown explicitly: 100 requests, 98 protocol-valid responses, and 97 cases in the common-response scoring subset. Objective agreement includes integer-rounding matches; none of these checks establishes semantic repair success. Means use the 98 valid responses, and token/latency figures exclude diagnosis generation.

Method	Protocol-valid response ↑	Python compiles ↑	Solver executes ↑	Reference-objective agreement ↑	Mean normalized edit size	Mean total tokens	Mean latency (seconds)
OptArgus-guided repair	98/100	98/98	95/98	83/97	0.228	4,877	14.2

Results and interpretation. The repairer returns 98 protocol-valid responses from 100 requests. All 98 compile, and 95 pass the execution check. In the 97-case scoring subset, 83 satisfy the rounding-tolerant objective-agreement rule. Mean normalized edit size is 0.228, with 4,877 tokens and 14.2 seconds per valid response (Table 17). These outcomes show that the archived reports can be integrated into a repair workflow that usually produces executable candidates.

They do not establish that the injected error was removed, that M and S preserve the intended problem, or that no new error was introduced. Those conclusions require semantic validation of the repaired tuple, beyond execution and objective agreement. The earlier detector configuration, retained construction hints, and absence of a reported comparative analysis also prevent attributing these outcomes to the final OptArgus design or to the diagnosis alone. The supported conclusion is operational feasibility, not proven repair quality or a causal repair benefit.

C.6 Limitations and Scope

Results are limited to the evaluated problems and generators; the controlled catalog covers 58 of 83 taxonomy entries. Detector-based clean-set selection, reference and label-version differences, and incomplete annotation validation also limit interpretation (App. B). Controlled scores measure target-label agreement, not whether every reported explanation is correct; natural scores measure detection at artifact and major-category levels, not exact error types.

The reference-detector comparisons evaluate complete systems, not isolated design choices. The repair study demonstrates workflow feasibility, not improved repair quality (App. C.5).

Distinguishing valid reformulations from genuine errors remains difficult (App. C.4). These detectors support expert review rather than certify correctness. Their diagnoses and any resulting repairs require independent semantic validation.

D Complete Optimization-Modeling Hallucination Taxonomy

We developed the taxonomy through OR-expert analysis and iterative refinement of optimization-modeling failures. Its organization follows where the intended semantics first change: the objective, decision variables, constraints, or solver implementation. Within each major family, subcategories group related failure mechanisms, and specific entries provide operational labels for annotation and detection. This produces 24 subcategories and 83 entries: 18 objective, 18 decision-variable, 31 constraint, and 16 implementation entries.

The taxonomy is designed around semantic consequences rather than surface differences. A catalog match is not by itself a positive label; the evidence must show a changed or undefined intended decision, feasible set, preference ordering or value, or executable behavior. Equivalent reformulations are excluded. Three entries, "Unnecessary Redundant Variables," "Unhandled Symmetric Variables," and "Failure to Exploit Sparsity," are unconditional quality warnings and are excluded from the semantic label space $\mathcal{T}_H$. Any independent semantic error receives the corresponding semantic label, not a warning label. Other modeling choices, such as a loose Big-M value, are treated as warnings unless the case-specific evidence establishes a semantic change.

The first three families describe deviations already present in the symbolic formulation M relative to the problem specification P. The implementation family describes additional deviations introduced by solver code S relative to M, whether or not M also contains an upstream error. Some entries describe specific mechanisms, while others describe broader effects such as weakened constraints; these descriptions can overlap. Controlled cases retain the label of their designated construction change, rather than imposing a new priority order among overlapping descriptions.

Table 18 lists the complete hierarchy. The remainder of this appendix gives the decision boundary for each subcategory and one fully documented representative entry for each of the 24 subcategories, including its definition, typical erroneous form, example, and consequence. Full documentation for all 83 entries is released in the anonymous online artifact as a standalone PDF, a content-equivalent Markdown file, a concise index, and a machine-readable hierarchy.

Table 18: The complete 83-entry optimization-modeling hallucination taxonomy. Superscript W marks quality warnings outside $\mathcal{T}_H$; App. B.3.3 distinguishes full-catalog and warning-excluded scores. All other entries require a case-specific semantic witness.

Major Category	Subcategory	Specific Hallucination Type
Objective-Function Modeling Hallucinations	Objective Semantic Mapping Errors	1.1.1 Wrong Optimization Direction 1.1.2 Objective Substitution 1.1.3 Softening a Hard Constraint 1.1.4 Turning a Preference into a Hard Constraint
	Objective Composition Errors	1.2.1 Omission of a Key Objective Term 1.2.2 Introduction of a Spurious Objective Term 1.2.3 Repeated Counting of an Objective Term 1.2.4 Wrong Sign on a Local Objective Term
	Objective Parameter and Index Binding Errors	1.3.1 Coefficient Misbinding 1.3.2 Wrong Index Range 1.3.3 Misaligned Time Indices 1.3.4 Wrong Unit or Scale Binding
	Multi-Objective and Risk-Objective Errors	1.4.1 Wrong Multi-Objective Aggregation Mechanism 1.4.2 Replacing a Risk or Extreme-Value Objective 1.4.3 Wrong Weight Setting or Interpretation
	Objective Formalization Errors	1.5.1 Smoothing Piecewise or Stepwise Cost into a Continuous Linear Cost 1.5.2 Incorrect Handling of Nonlinear or Nonsmooth Objective Terms 1.5.3 Fractional Objective Errors
Decision-Variable Modeling Hallucinations	Variable Semantic Definition Errors	2.1.1 Wrong Variable Object 2.1.2 Missing a Key Dimension 2.1.3 Wrong Aggregation Level of Variables 2.1.4 State-Control Confusion
	Variable Domain and Type Errors	2.2.1 Relaxing a Discrete Variable into a Continuous Variable 2.2.2 Forcing a Continuous Variable to Be Discrete 2.2.3 Wrong Value Range 2.2.4 Wrong Sign Domain
	Variable Index and Set Binding Errors	2.3.1 Omitted Index Set 2.3.2 Fabricated Index Set 2.3.3 Wrong Subscript Binding 2.3.4 Missing Index Dependence
	Variable Role Coupling Errors	2.4.1 Variable Not Taking Effect 2.4.2 Duplicate Variable Roles 2.4.3 Missing Coupling Between Master and Auxiliary Variables
	Variable Redundancy and Uncontrolled Symmetry	2.5.1 Unnecessary Redundant Variables^{W} 2.5.2 Unhandled Symmetric Variables^{W} 2.5.3 Failure to Exploit Sparsity^{W}
Constraint Modeling Hallucinations	Constraint Semantic Translation Errors	3.1.1 Wrong Interpretation of the Rule 3.1.2 Wrong Interpretation of Quantifiers 3.1.3 Confusing Resource Amount with Time Feasibility 3.1.4 Constraint Type Mismatch
	Missing Constraint Skeleton	3.2.1 Missing Balance, Conservation, or Recursion Constraints 3.2.2 Missing Chain or Path Structure Constraints 3.2.3 Missing Capacity or Coverage Skeleton 3.2.4 Missing Initial or Terminal Conditions
	Missing Implicit Constraints	3.3.1 Missing Default Business Rules 3.3.2 Missing Must-Include Object Constraints 3.3.3 Missing Legal-Action Set Restrictions
	Erroneous Equivalent Substitutions	3.4.1 Replacing an Equality with an Inequality 3.4.2 Using Local or Partial Conditions to Replace a Full Structure 3.4.3 Treating an Approximate Constraint as an Exact One
	Logic and Conditional-Chain Errors	3.5.1 Big-M Direction Error 3.5.2 Big-M Too Loose or Too Tight 3.5.3 Incomplete Logical Linearization 3.5.4 Broken Conditional Chains
	Aggregation and Index-Coding Errors	3.6.1 Wrong Aggregation Level 3.6.2 Wrong Summation Range 3.6.3 Wrong Subscripts When Calling a Constraint 3.6.4 Repeated or Missing Aggregation
	Constraint Boundary and Direction Errors	3.7.1 Wrong Inequality Direction 3.7.2 Swapped Upper and Lower Bounds 3.7.3 Misplaced Boundary Conditions 3.7.4 Misuse of Threshold or Boundary Parameters
	Constraint Strength Imbalance	3.8.1 Constraint Too Weak 3.8.2 Constraint Too Strong 3.8.3 Slack or Tolerance Parameters on the Wrong Scale
	Scheduling- and Activity-Structure-Specific Errors	3.9.1 Wrong Encoding of Precedence Relations 3.9.2 Wrong Structure for Mode Selection and Optional Activities
Implementation Hallucinations	Symbolic-Code Mismatch	4.1.1 Wrong Objective Sense in Code 4.1.2 Omitted Constraint Materialization 4.1.3 Missing Variable Registration 4.1.4 Stale or Divergent Coefficient Mapping
	Variable and Domain API Mismatch	4.2.1 Wrong API Variable Type 4.2.2 Wrong Bounds in Code 4.2.3 Wrong Index Domain Materialization
	Index and Set Materialization Errors	4.3.1 Dropped Loop or Index Expansion 4.3.2 Partial Set Expansion 4.3.3 Filtered Index Loss
	Solver and Formalization Mismatch	4.4.1 Missing Linearization in Code 4.4.2 Incompatible Solver Selection 4.4.3 Incorrect Big-M or Numerical Option in Code
	Post-Solve Extraction and Reporting Divergence	4.5.1 Wrong Solution Variable Readout 4.5.2 Misreported Objective Value 4.5.3 Stale Result Object or Wrong Post-Processing

D.1 Objective-Function Modeling Hallucinations

D.1.1 Objective Semantic Mapping Errors

Definition. This category concerns failures to map the intent of a natural-language task—what should be optimized, in which direction it should be optimized, and which requirements are mandatory

versus merely preferable—into the correct objective function and objective role in a mathematical program. The result is an objective whose notion of optimality no longer matches the original decision intent.

Boundary. The focus here is the semantic role of the objective: whether the model understood what should be optimized and what should remain a hard requirement. If the error lies in omitted or redundant cost terms, repeated counting, or a local sign flip inside the objective, it belongs to Sec. D.1.2. If the issue is the misbinding of coefficients, time indices, object indices, or units, it belongs to Sec. D.1.3. If the error concerns multi-objective aggregation, risk preference, or the interpretation of weights, it belongs to Sec. D.1.4. If the mathematical realization of a piecewise, extreme-value, absolute-value, bilinear, or fractional objective is wrong, it belongs to Sec. D.1.5. In short, Sec. D.1.1. asks whether the semantic target of optimization is correct, not whether the target was expanded, indexed, or encoded correctly.

1.1.1 Wrong Optimization Direction

Definition. The original problem asks to minimize or maximize a performance metric, but the generated model reverses that direction, so the solver systematically searches in the opposite direction of the real decision goal.

Typical erroneous form. The intended objective is $\min f(x)$, but the model is written as $\max f(x)$.

Example. In a regional power-purchasing plan, total purchasing cost should be minimized subject to load balance and reserve constraints: $\min \sum_t \sum_g c_{gt} p_{gt}$, where p_{gt} is the purchased power or dispatch output of unit g in period t, and c_{gt} is the corresponding electricity price or marginal cost. If the formulation is mistakenly written as $\max \sum_t \sum_g c_{gt} p_{gt}$, the solver will prefer expensive generators first. All constraints may still be satisfied formally, yet the optimization direction is exactly opposite to the true dispatch goal.

Consequence. This error flips the criterion of optimality itself and makes the solver prefer "worse is better." It does not create a small local deviation; it distorts the global direction of the decision. Cost-minimization becomes cost amplification, profit-maximization becomes profit suppression, and the computed solution may be mathematically optimal while being close to worst possible from a business standpoint.

D.1.2 Objective Composition Errors

Definition. The model preserves the overall optimization direction, yet the objective fails to include the right collection of costs, rewards, penalties, or bonuses at the term level. Typical manifestations are omitted terms, irrelevant extra terms, repeated counting, or locally reversed signs. The result is a distorted ranking among candidate decisions.

Boundary. This section asks which terms should appear in the objective and whether they are included correctly. If the error concerns the overall direction of optimization, the role of objective versus constraints, or parameter-variable binding, it belongs to Secs. D.1.1 or D.1.3 instead.

1.2.1 Omission of a Key Objective Term

Definition. A cost, reward, or penalty term that should appear in the objective is dropped. A certain behavior therefore becomes effectively free or unrewarded, and the model is systematically biased toward unrealistic "optimal" solutions.

Typical erroneous form. The intended objective contains two or more components, such as $\min\{f_1(x) + f_2(x)\}$, but the symbolic formulation only uses $\min f_1(x)$, thereby omitting the key term $f_2(x)$.

Example. In workforce planning, hiring cost, firing cost, and wage cost may all matter: $\min \sum_t \big(c_t^h H_t + c_t^f F_t + c_t^w W_t\big)$. Here H_t, F_t, and W_t denote hires, layoffs, and workforce level in period t. If the firing-cost term is omitted, the model treats layoffs as free and may repeatedly fire and rehire workers to regulate capacity.

Consequence. The omitted behavior is systematically underpriced, so the solver overuses it. Even with correct constraints, the "optimal" plan may show unreasonable over-firing, excessive switching,

frequent start-stop cycles, or disregard for tardiness. The deeper problem is that the objective no longer evaluates solutions according to the true economic criterion.

D.1.3 Objective Parameter and Index Binding Errors

Definition. Cost, reward, and penalty parameters do appear in the model, but they are bound to the wrong variables, indices, time points, or units. The relevant objective term exists only in appearance; in effect it acts on the wrong decision object.

Boundary. This section concerns cases in which an objective term is present but attached to the wrong target. If a term is omitted, added redundantly, counted repeatedly, or locally sign-reversed, the issue belongs to Sec. D.1.2. If the problem is with the overall optimization direction, objective substitution, or objective-constraint role confusion, it belongs to Sec. D.1.1.

1.3.1 Coefficient Misbinding

Definition. A cost, reward, or penalty coefficient is multiplied by the wrong variable, so the economic meaning associated with one decision object is mistakenly transferred to another.

Typical erroneous form. The intended expression is $\min \sum_i c_i x_i$, but the formulation uses $\min \sum_i c_i y_i$, even though the business meaning of c_i corresponds to x_i, not to y_i.

Example. In urban distribution, vehicle activation cost should be charged per activated vehicle. Let $y_k \in \{0, 1\}$ indicate whether vehicle k is used, F_k its fixed activation cost, and x_{ijk} whether vehicle k traverses arc (i, j). The fixed-cost term should be $\sum_k F_k y_k$. If the formulation instead charges the fixed cost through a sum over the traversed arcs of each vehicle, the model effectively turns a per-vehicle cost into a per-arc cost.

Consequence. The marginal-cost structure of the decision is changed. The model no longer compares "whether to activate one vehicle" but instead compares "whether taking another arc will trigger a fixed cost that should not exist." This can produce overly compressed fleet sizes, unnaturally short routes, or other preference patterns with no business justification.

D.1.4 Multi-Objective and Risk-Objective Errors

Definition. When a problem contains multiple objectives, hierarchical preferences, risk-aversion principles, or extreme-value performance measures, the generated model may fail to preserve the intended combination mechanism, priority structure, or risk representation. It may aggregate, average, replace, or reweight the objectives incorrectly, so the returned "optimal" solution is not optimal for the original decision semantics.

Boundary. This section concerns how multiple objectives are combined, ordered, and made to express risk preference. Errors in the direction, object, term composition, or parameter binding of a single objective belong to Secs. D.1.1,D.1.2,D.1.3. Errors in the concrete mathematical coding of piecewise, fractional, absolute-value, or extreme-value forms belong to Sec. D.1.5.

1.4.1 Wrong Multi-Objective Aggregation Mechanism

Definition. The original problem specifies a particular relation among objectives—for example lexicographic optimization, hierarchical optimization, stagewise optimization, weighted-sum optimization, or an ϵ-constraint method—but the generated model mixes them up or replaces one mechanism with another non-equivalent one.

Typical erroneous form. The intended model solves first $\min f_1(x)$, and then, within the set of f_1-optimal solutions, solves $\min f_2(x)$. The formulation instead writes a single weighted objective such as $\min\big(f_1(x) + \lambda f_2(x)\big)$ with an arbitrary λ, even though this is generally not equivalent to lexicographic optimization.

Example. In airport-recovery scheduling, suppose the true requirement is first to minimize the maximum flight delay and only then to minimize total delay. A correct lexicographic formulation solves $\min z \quad \text{s.t. } d_f \le z,\ \forall f$, and then, after obtaining the optimal value $z^\star$, solves $\min \sum_{f \in F} d_f \quad \text{s.t. } d_f \le z^\star,\ \forall f$. If the formulation instead minimizes a weighted sum $z + \lambda \sum_f d_f$ with an arbitrary λ, the resulting optimization problem is fundamentally different.

Consequence. The priority structure among objectives is destroyed. The model may accept a noticeably worse worst-case delay just to shave a little off total delay, or sacrifice a primary business criterion in exchange for a small improvement in a secondary one.

D.1.5 Objective Formalization Errors

Definition. The semantic target of the objective is broadly correct and the relevant terms have been identified, but the mathematical form used in the optimization model fails to preserve the original functional structure. Piecewise, extreme-value, absolute-value, bilinear, and fractional objectives may be linearized, replaced, or transformed incorrectly. The error lies not in optimizing the wrong thing, but in implementing the shape of the right thing incorrectly.

Boundary. This section concerns the correctness of the mathematical form of the objective. Errors in direction, object, or objective-constraint role belong to Sec. D.1.1; errors in omitted or redundant terms, repeated counting, or local sign problems belong to Sec. D.1.2; errors in parameter, index, time, or unit binding belong to Sec. D.1.3; and errors in multi-objective mechanism or risk preference belong to Sec. D.1.4.

1.5.1 Smoothing Piecewise or Stepwise Cost into a Continuous Linear Cost

Definition. The original cost structure contains fixed activation fees, stepped charges, piecewise-linear cost, or batch-triggered cost jumps, but the generated model smooths the structure into a simple linear term and ignores the discontinuous jumps that occur when another batch, vehicle, or machine must be activated.

Typical erroneous form. A transport or production cost that should look like "fixed cost plus variable cost" or a piecewise interval charge is replaced by an average linear unit cost. For example, a model of the form $\min \sum_k F_k y_k + \sum_k c_k q_k$ is replaced by $\min \sum_k \bar{c}_k q_k$, thereby smearing the fixed or stepped component into the unit cost.

Example. In long-haul transport, suppose one standard truck carries at most Q tons, each activated truck costs F, and variable shipping cost is c per ton. If the transported quantity is x, then the true cost includes the integer number of trucks needed and a capacity relation such as $x \le Qn$ with $n \in \mathbb{Z}_+$. If the formulation instead uses a purely linear cost in x, the integer jump in truck activation disappears.

Consequence. The replacement can underprice or overprice activation, depending on the chosen linear rate and quantity. It may therefore distort the choice of batch sizes, vehicles, or machine starts. Underestimating the required vehicle count is not inevitable; it also depends on whether the capacity and integrality constraints are preserved.

D.2 Decision-Variable Modeling Hallucinations

D.2.1 Variable Semantic Definition Errors

Definition. This category covers failures to map decision objects, system states, and control actions in a natural-language task into decision variables with the correct semantic meaning. The issue is not simply that the variable type or bound is wrong; it is that the variable already misrepresents the modeling object at the definition stage, so even a formally complete objective and constraint set may still be built on semantically wrong variables.

Boundary. The focus here is what the variable is supposed to represent. If the issue is variable type, bounds, or allowed values, it belongs to Sec. D.2.2. If it concerns index sets, subscript binding, or missing scenario dimensions, it belongs to Sec. D.2.3. If the problem is missing coupling, consistency, or master-auxiliary binding among variables, it belongs to Sec. D.2.4.

2.1.1 Wrong Variable Object

Definition. The formulation defines a variable for the wrong decision object. A problem that needs a selection variable, assignment variable, flow variable, or state variable is modeled with a different kind of variable that has a different semantics.

Typical erroneous form. A discrete-choice decision should be represented by a binary variable such as $x_{ijk} \in \{0, 1\}$, indicating whether vehicle k traverses arc (i, j), but the formulation defines only a continuous flow variable $f_{ijk} \geq 0$, and assumes that "positive flow means the arc is chosen." This replaces a path-choice decision with a splittable-flow interpretation.

Example. In a last-mile routing problem, each vehicle should form a discrete route. A typical variable is $x_{ijk} = 1$ if vehicle k travels from node i to node j, and 0 otherwise. If the formulation defines only arc flow $f_{ij} \geq 0$ and interprets "has flow" as "uses the arc," the model no longer describes which arcs vehicles traverse; it describes how commodity flow passes through a network. The former is route choice, the latter is closer to network flow.

Consequence. The decision object of the entire problem is changed. A discrete route can become a splittable flow, a machine on/off decision can become a continuous output variable, and a state-transition problem can become a static allocation problem. The solver is then solving a different problem created by the wrong variable object.

D.2.2 Variable Domain and Type Errors

Definition. Here the variable corresponds to the right modeling object, but the set of values it is allowed to take is wrong. A variable that should be binary, integer, nonnegative, bounded, or sign-free is placed in the wrong mathematical domain, thereby reshaping the feasible region directly.

Boundary. The issue is not what the variable means, but what values it may take. Errors in variable semantics belong to Sec. D.2.1. Missing coupling or logical consistency among variables belongs to Sec. D.2.4. Errors in constraints rather than variable declarations belong to Sec. D.3.

2.2.1 Relaxing a Discrete Variable into a Continuous Variable

Definition. A variable that should take discrete values—binary, integer, counting, or categorical levels—is incorrectly modeled as continuous, allowing meaningless intermediate values such as "half a warehouse" or "0.7 of a truck."

Typical erroneous form. A variable that should satisfy $y_i \in \{0, 1\}$ or $n_i \in \mathbb{Z}_+$ is modeled as $0 \leq y_i \leq 1$ or $n_i \geq 0$ only, thereby relaxing a discrete choice problem into a continuous one.

Example. In facility location, $y_j \in \{0, 1\}$ indicates whether warehouse j is opened. If the formulation instead uses $0 \leq y_j \leq 1$, the model may return values such as 0.4 or 0.7, which amount to "partially opened warehouses" with no business meaning.

Consequence. The model becomes an artificially relaxed version of the original discrete problem, often producing overly optimistic lower bounds or non-implementable fractional solutions.

D.2.3 Variable Index and Set Binding Errors

Definition. The basic semantics and domain of the variable may both be correct, but the variable is defined over the wrong object set, with the wrong indices, or with missing contextual dependence. The variable therefore exists on the wrong object domain and the model describes the wrong individuals, roles, scenarios, or layers.

Boundary. The key question is over which objects and indices the variable should be defined. Errors in variable meaning belong to Sec. D.2.1, errors in domain belong to Sec. D.2.2, and errors in objective or constraint summation belong elsewhere. In contrast to Sec. D.2.1, the variable object here is usually broadly correct; the problem lies in the index structure on which it is instantiated.

2.3.1 Omitted Index Set

Definition. An entire class of objects that should have its own variables is omitted from the variable definition, so the model contains no decision representation for that class at all.

Typical erroneous form. The intended model should define variables for two or more object sets, for example one variable family for captains and another for first officers, but the formulation builds variables for only one of them.

Example. In crew scheduling, a duty chain may require both captains and first officers. If the model defines assignment variables only for captains and omits the corresponding first-officer variables entirely, it can represent captain assignment but not the full crew configuration.

Consequence. One whole category of actors or resources disappears from the model. The optimization may still produce a formal solution, but a crucial role or resource turns out never to have been assigned.

D.2.4 VARIABLE ROLE COUPLING ERRORS

Definition. Individual variables may each have the right semantics, type, and index structure, but the activation, mapping, consistency, subordination, or support relations that should connect multiple variables are not established correctly. The problem is no longer "what each variable is," but whether the variables work together in the right roles.

Boundary. This section concerns how multiple variables coordinate with one another. Errors in variable meaning, domain, or indexing belong to Secs. D.2.1–D.2.3. If the issue is the absence of a core structural constraint rather than a missing variable-to-variable role relation, it belongs to Sec. D.3.

2.4.1 Variable Not Taking Effect

Definition. A variable is declared to represent a required decision or relation, but the formulation does not connect it to the objective or constraints needed to realize that role, and no equivalent encoding supplies the missing relation. An unused variable alone is only a redundancy warning, not evidence of a semantic error.

Typical erroneous form. The formulation declares a decision variable but omits the links required for its intended role. The required decision or relation is consequently absent from the model, rather than merely represented by a different variable or equivalent expression.

Example. In flight-recovery scheduling, suppose a connection variable y_{fg} indicates whether flight g follows flight f, and the task requires each selected connection to be temporally compatible. If y_{fg} is never linked to flight times and no other constraint enforces that compatibility, the model permits connections that the specification forbids.

Consequence. The variable name suggests that a required decision or relation is represented, but its missing links leave the intended feasibility or objective semantics unenforced.

D.2.5 VARIABLE REDUNDANCY AND UNCONTROLLED SYMMETRY

Definition. The model broadly captures the right decision objects, but the variable design fails to exploit problem structure. It introduces unnecessary extra variables, ignores opportunities to merge or compress them, or leaves large sets of perfectly symmetric variables untreated. The resulting model is bloated, filled with equivalent branches, and more vulnerable to later indexing or constraint errors.

Boundary. These are model-quality warnings, not semantic hallucinations by themselves. They enter $\mathcal{W}$, not $\mathcal{T}_H$; controlled scores are reported both with and without warning cases (App. B.3.3). If an over-expanded variable domain additionally permits an illegal action, that separate feasible-set change should receive an appropriate semantic variable or constraint label.

2.5.1 Unnecessary Redundant Variables

Definition. Extra variables are introduced even though they carry no new business semantics and serve no clear purpose such as linearization, logical encoding, decomposition, or numerical stabilization. They merely duplicate information already present in the model.

Typical erroneous form. A main variable already captures the decision state, but the formulation adds another variable family that neither introduces new structure nor performs a necessary intermediate modeling function.

Example. In facility location with distribution, suppose y_i already indicates whether facility i is open and x_{ij} already captures its assignments or flows. If the formulation adds another binary variable z_i

also meaning whether facility i is used, yet z_i is not needed for any extra logic or decomposition, it is just a duplicate encoding of the same fact.

Consequence. The model becomes larger without gaining expressive power. Consistency constraints, index management, debugging difficulty, and solve effort all increase, and the redundant design may even hide whether the truly necessary structure has been encoded correctly.

D.3 Constraint Modeling Hallucinations

D.3.1 Constraint Semantic Translation Errors

Definition. A business rule, logical condition, coverage requirement, feasibility criterion, or eligibility restriction from the natural-language task is indeed recognized as something that should become a constraint, but the mathematical constraint does not preserve the original semantics. The constraint exists, yet it no longer means the same rule.

Boundary. The question here is what rule the natural-language statement was translated into. If a necessary class of constraints was not built at all, the issue belongs to Sec. D.3.2. If the constraint exists but its index, boundary, strength, or direction is wrong, the issue belongs to later sections. If the real problem lies in variable definitions rather than in the translation of the rule, it belongs to Sec. D.2.

3.1.1 Wrong Interpretation of the Rule

Definition. The model recognizes that a business rule should become a constraint, but misunderstands the operative meaning of the rule itself—what is allowed, what is forbidden, or what process structure is required. The resulting constraint describes another set of feasible actions.

Typical erroneous form. A rule saying that only certain actions are allowed, or that an action may occur only under a particular condition, is translated into a broader or simply different relation. Examples include turning "can only substitute for A" into "can substitute for anything," or turning "must be continuous" into "total amount is sufficient."

Example. In maintenance scheduling, once major repair starts on a machine, it must continue without interruption until completion. If $x_{mt} \in \{0, 1\}$ indicates whether machine m is under maintenance in period t, this continuity requirement cannot be expressed by a total-duration condition alone such as $\sum_t x_{mt} = L_m$, where L_m is the required maintenance length. That condition allows maintenance to be scattered across nonconsecutive dates. It describes "maintenance lasted this long in total," not "maintenance was continuous."

Consequence. The feasible action logic of the original problem is replaced by a weaker or otherwise different one. The model may satisfy aggregate totals while violating process semantics, producing maintenance splits, broken tasks, forbidden substitutions, or other operationally infeasible plans.

D.3.2 Missing Constraint Skeleton

Definition. A class of core constraints that determines the basic topology of the feasible region is missing, or only fragments of it are present. Variables therefore lack the fundamental links required for balance, continuity, coverage, capacity, or boundary conditions. The model may still run, but it no longer represents the basic operating mechanism of the original problem.

Boundary. The focus is whether the structural backbone of the model exists at all. If a rule has been written but semantically mistranslated, it belongs to Sec. D.3.1. If the backbone exists but is indexed, bounded, or directed incorrectly, the issue belongs to later sections. Sec. D.3.2 asks whether the frame was built, not whether a built frame is slightly crooked.

3.2.1 Missing Balance, Conservation, or Recursion Constraints

Definition. The model omits constraints that should connect adjacent periods, nodes, or states through balance, conservation, or recursion. The state evolution no longer follows the logic of inflow, outflow, and stock change.

Typical erroneous form. Relations such as $I_t = I_{t-1} + q_t - d_t$, $\sum_j x_{ji} - \sum_j x_{ij} = b_i$, or $s_{t+1} = f(s_t, u_t)$ are missing, while local capacity or single-period requirements are retained.

Example. In multi-period inventory planning, production q_t, demand d_t, and ending inventory I_t should be tied by inventory recursion. If the model contains only capacity limits on q_t and local demand satisfaction inequalities, but no inventory recursion, inventory in different periods is no longer linked and may appear "from nowhere."

Consequence. Inventory, energy, cash, material, bed occupancy, or queue length cease to be governed by history. The model can produce pseudo-feasible plans such as having stock in a period without previous production or inflow.

D.3.3 Missing Implicit Constraints

Definition. The model omits a requirement supported by the problem description, an explicit domain assumption, or a logical consequence of those premises, even though the requirement is not stated as a formula. Common practice alone does not establish that a constraint is required for the given problem.

Boundary. A finding must identify the premise that requires the missing rule and show that no equivalent encoding enforces it. If the key structural backbone is missing, the issue belongs to Sec. D.3.2. If the rule was included but given the wrong direction, location, or threshold, the issue belongs to Sec. D.3.7.

3.3.1 Missing Default Business Rules

Definition. The formulation omits an operational or organizational rule required by the stated task or its explicit assumptions. The omission permits decisions that violate that requirement; an unstated industry convention is not sufficient evidence.

Typical erroneous form. The formulation enforces task assignment or coverage but omits a task-required rule, such as return to base or a specified minimum rest interval, without an equivalent constraint elsewhere.

Example. Suppose a crew-scheduling task explicitly requires each duty chain to start and end at the crew's base. A formulation that covers every flight but omits those endpoint requirements can admit a chain ending at another airport. Merely assigning each crew a home base would not, by itself, establish a return requirement.

Consequence. The model returns plans that look fully covered on paper but cannot actually be executed. These errors are often hidden because the objective value and coverage rate still look good.

D.3.4 Erroneous Equivalent Substitutions

Definition. In an attempt to simplify, linearize, localize, or engineer a constraint, the model replaces the original condition with something that looks similar but is not actually equivalent. The substitute preserves only part of the semantics, so the feasible region is silently changed.

Boundary. The concern here is not the absence of a constraint but a misleading stand-in for it. If the rule itself was misunderstood, the issue belongs to Sec. D.3.1. If the backbone is missing, it belongs to Sec. D.3.2. If the problem is specifically a broken logical encoding or Big-M implementation, it belongs to Sec. D.3.5.

3.4.1 Replacing an Equality with an Inequality

Definition. A relation that should hold exactly or be perfectly balanced is replaced by an upper-bound or lower-bound inequality. The precise feasibility condition is thereby relaxed or skewed.

Typical erroneous form. A condition such as $\sum_{a \in A_t} x_{at} = 1$ is written as $\sum_{a \in A_t} x_{at} \leq 1$ or $\sum_{a \in A_t} x_{at} \geq 1$. The former permits doing nothing; the latter permits repeated selection.

Example. In equipment control, each period may require exactly one operating mode. If $x_{mt} \in \{0, 1\}$ indicates whether mode m is selected in period t, then the correct relation is $\sum_m x_{mt} = 1, \quad \forall t$. Using ≤ 1 allows empty mode selection; using ≥ 1 allows multiple modes simultaneously.

Consequence. A precisely defined state or action relation is replaced by an incomplete one, allowing the solver to exploit "do nothing" or "do several at once" loopholes.

D.3.5 Logic and Conditional-Chain Errors

Definition. The model attempts to encode implications, equivalences, or multi-step trigger chains using binary variables, Big-M constraints, activation bounds, or linearization, but the encoding changes the intended logic. Examples include reversed implications, invalid constants that change feasibility, a missing direction of an equivalence, or a broken conditional chain. A loose but valid Big-M is only a quality warning unless case-specific evidence establishes semantic failure.

Boundary. This section concerns how logical implications are encoded mathematically. If the rule itself was misunderstood, the problem belongs to Sec. D.3.1. If the backbone of constraints is missing, the issue belongs to Sec. D.3.2. If a non-equivalent stand-in replaced the original condition, the issue belongs to Sec. D.3.4.

3.5.1 Big-M Direction Error

Definition. A Big-M implication is written with the trigger direction reversed, or with the logical control variable tied to the wrong side of the relaxed constraint. The encoded logic is therefore the opposite of what was intended.

Typical erroneous form. To express "when $y = 1$, enforce $a^\top x \le b$; when $y = 0$, do not enforce it," one typically writes $a^\top x \le b + M(1 - y)$. If the formulation writes $a^\top x \le b + My$, then the constraint is loose when $y = 1$ and tight when $y = 0$, which is exactly the opposite logic.

Example. Suppose $y_t \in \{0, 1\}$ indicates whether overtime mode is turned on in period t, and only then may working hours exceed the regular limit H. A natural formulation is $h_t \le H + My_t$. If the formulation uses $h_t \le H + M(1 - y_t)$, the model allows extra hours when overtime is off and tightens the constraint when overtime is on.

Consequence. A variable intended to activate a condition ends up deactivating it. The model then rewards the wrong status: extra resources may appear when a mode is not selected, and selected options may become more constrained rather than less.

D.3.6 Aggregation and Index-Coding Errors

Definition. The business rule behind a constraint may have been recognized correctly, but when the rule is written as sums, grouped constraints, per-object expansions, or cross-level aggregations, the aggregation level, summation range, or subscript binding is wrong. The constraint may look familiar while applying to the wrong objects or the wrong statistical level.

Boundary. The issue here is how the constraint is written over objects and indices. If the semantic rule itself is wrong, the issue belongs to Sec. D.3.1. If the core structure is missing, it belongs to Sec. D.3.2. If the problem is Big-M direction or logical propagation, it belongs to Sec. D.3.5.

3.6.1 Wrong Aggregation Level

Definition. A requirement is enforced at the wrong level: separate per-object constraints are incorrectly pooled, or a shared total requirement is incorrectly imposed on each object. Either direction can change the intended feasible set.

Typical erroneous form. A correct form such as $\sum_i x_{ij} = d_j, \quad \forall j$ is replaced by $\sum_j \sum_i x_{ij} = \sum_j d_j$. The latter enforces only total balance, not per-customer satisfaction.

Example. In warehouse distribution, each customer's demand must be satisfied individually. If the formulation enforces only equality between total shipped volume and total demand, some customers may be overserved while others receive nothing.

Consequence. Local feasibility is falsely replaced by aggregate feasibility. The overall accounting may balance while the plan fails at the level where execution actually matters.

D.3.7 CONSTRAINT BOUNDARY AND DIRECTION ERRORS

Definition. The basic semantics and objects of the constraint may be right, yet the inequality direction, the placement of upper and lower bounds, the point at which a boundary takes effect, or the threshold parameter itself is wrong. The constraint still exists, but it cuts the feasible region in the wrong direction or at the wrong place.

Boundary. This section is about how much is restricted, in which direction, and at which boundary location or time point. Missing default rules belong to Sec. D.3.3, missing skeleton constraints belong to Sec. D.3.2, and wrong summation domains belong to Sec. D.3.6.

3.7.1 Wrong Inequality Direction

Definition. A quantity that should be bounded above, bounded below, guaranteed, or suppressed is constrained with the inequality sign reversed.

Typical erroneous form. A relation intended as $a^\top x \leq b$ is written as $a^\top x \geq b$, or a coverage requirement $\sum_i x_{ij} \geq d_j$ is written as $\sum_i x_{ij} \leq d_j$.

Example. In production scheduling, if line output in period t must not exceed capacity C_t, the correct form is $q_t \leq C_t$. Writing $q_t \geq C_t$ turns a maximum-capacity restriction into a minimum-production requirement.

Consequence. The reversed bound may permit overload, lateness, or undercoverage, or exclude otherwise valid decisions. For example, $q_t \geq C_t$ allows overload but does not force it, since equality remains feasible.

D.3.8 CONSTRAINT STRENGTH IMBALANCE

Definition. A constraint may target the right requirement and objects, yet enforce a weaker or stronger condition than intended. A semantic error requires evidence that the mismatch admits forbidden decisions or excludes required feasible decisions; a merely weak relaxation used for analysis is not sufficient.

Boundary. Strength describes the effect of a constraint, so it can overlap with the more specific mechanisms in Secs. D.3.1–D.3.7, including non-equivalent substitution and aggregation. These examples do not prescribe a new label-priority rule; controlled labels retain their designated construction mechanism.

3.8.1 Constraint Too Weak

Definition. The constraint keeps only a weak shadow of the original restriction, so many plans that should be excluded remain feasible.

Typical erroneous form. The formulation substitutes a weak necessary condition for the required restriction. The surrogate holds for every intended feasible decision but also admits a demonstrably forbidden one.

Example. Suppose binary activity choices must satisfy $y_1 + y_2 \leq 1$ and $y_2 + y_3 \leq 1$ because the indicated pairs are incompatible. Replacing both restrictions with $y_1 + y_2 + y_3 \leq 2$ retains a necessary count bound but admits $(y_1, y_2, y_3) = (1, 1, 0)$, which violates the first incompatibility.

Consequence. The model admits pseudo-feasible plans that look cheaper or better only because they do less work, serve fewer demands, or assume away responsibilities that should have been mandatory.

D.3.9 SCHEDULING- AND ACTIVITY-STRUCTURE-SPECIFIC ERRORS

Definition. In scheduling, project networks, crew-connection problems, multimode operations, or optional-activity settings, the model may have the right basic variables and some local constraints, yet fail to preserve the distinctive topology, mode-switching semantics, and activity-activation logic of those problem classes.

Boundary. This section collects the remaining structural errors/hallucinations that are particularly characteristic of time-ordered and activity-based models. General errors/hallucinations in equivalent

replacement, logic, aggregation, or boundary direction still belong to earlier sections; Sec. D.3.9 adds the especially common distortions that arise in precedence networks and mode-choice structures.

3.9.1 Wrong Encoding of Precedence Relations

Definition. A precedence relation among tasks, activities, flights, jobs, or events is recognized, but its start times, finish times, lags, trigger conditions, predecessor sets, or successor sets are encoded incorrectly. The resulting precedence constraint no longer represents the intended topology.

Typical erroneous form. Common forms include replacing "start after finish" with "start after start," omitting processing or service duration, writing predecessor sets as successor sets, or using $s_j \geq s_i + p_j$ where $s_j \geq s_i + p_i$ is required.

Example. In project scheduling, if activity j may start only after activity i finishes, with start time s_i and duration p_i, then the standard relation is $s_j \geq s_i + p_i$. If the formulation writes $s_j \geq s_i + p_j$, it uses the successor duration instead of the predecessor duration, which can either delay too much or allow too-early starts depending on the durations.

Consequence. The topology of the schedule is corrupted. Tasks may start before their predecessors finish, be delayed for the wrong reason, or distort the critical path and all downstream timing logic.

D.4 Implementation Hallucinations

Definition. This family concerns additional deviations introduced when solver code S instantiates, solves, or reports the symbolic model M. Such deviations can arise through API misuse, dropped loops, wrong objective sense, incompatible solver settings, or incorrect post-solve extraction. They can occur even when M already contains a separate upstream error. In our current detector, the implementation branch checks for these failures through symbolic-to-code review and static code-contract checks.

Boundary. If the objective, variable, or constraint is already wrong in M and the code simply mirrors that error, the root cause belongs to $\mathcal{T}^O$, $\mathcal{T}^V$, or $\mathcal{T}^C$ rather than $\mathcal{T}^I$. If S introduces an independent deviation from M, such as changing a variable domain or misreporting an output, that additional error belongs here. Distinct errors in both layers may therefore receive both labels; an upstream error is not counted twice merely because it propagates to code.

Practical interpretation. The implementation specialist compares the symbolic model M with the solver code S, checking whether variables, objectives, constraints, and solution reporting are faithfully represented. Static code-contract checks supplement this review with cues such as invalid API references, placeholder code, and Python syntax errors. The detector inspects the source without executing the solver program or constructing a code graph; it therefore does not claim to verify runtime behavior.

Why this family is operationally important. An algebraic program can instantiate, solve, or report a different model from the one described by its symbolic formulation. Standard sanity checks may still pass: the code can compile, the solver can return an "optimal" status, and the reported objective can look numerically plausible. What fails is fidelity between the formulation and the executable artifact.

D.4.1 Symbolic-Code Mismatch

Definition. The symbolic model and the solver program encode different mathematical objects. Typical manifestations include wrong objective sense in code, omitted constraint materialization, missing variable registration, and stale coefficient mappings.

Boundary. The issue here is not whether the symbolic model itself is right, but whether code faithfully instantiates it. If the mathematical formulation already omits a required constraint, that is a symbolic constraint hallucination; if the formulation contains the constraint but code never adds it to the solver, it is an implementation hallucination.

4.1.1 Wrong Objective Sense in Code

Definition. The symbolic model specifies one optimization sense, but the solver program materializes the opposite sense or an equivalent sign-flipped objective without a faithful correction.

Typical erroneous form. The symbolic formulation declares $\min c^\top x$, but the code calls a maximize API, negates coefficients without a matching correction, or reports the solved objective with the wrong sign. A common pattern is a solver call such as `setObjective(expr, GRB.MAXIMIZE)` when the symbolic model requires minimization.

Example. Suppose a facility-allocation model minimizes transportation cost $\min \sum_{i,j} c_{ij} x_{ij}$. If the solver code instead sets the same linear expression under a maximize sense, the executable artifact will prefer expensive routes rather than cheap ones. The symbolic model remains correct, but the realized program optimizes the opposite business criterion.

Consequence. This error flips optimality at the executable level. The program may compile, solve, and even return a numerically plausible value, yet every downstream decision is chosen according to the wrong direction of preference.

D.4.2 Variable and Domain API Mismatch

Definition. The solver API assigns the wrong domain, bounds, or admissible index domain to variables relative to the symbolic model. Representative subtypes are wrong API variable type, wrong bounds in code, and wrong index-domain materialization.

Boundary. This category concerns an API realization that changes the symbolic variable declaration, for example by materializing a binary variable as continuous or expanding variables over an overly broad Cartesian product. A separate error elsewhere in M does not exclude this additional implementation error.

4.2.1 Wrong API Variable Type

Definition. The solver API creates a variable with the wrong type, such as continuous instead of binary or integer.

Typical erroneous form. A symbolic binary or integer variable is created in code with a continuous domain, or a continuous variable is incorrectly forced into an integer API type. In practice this often appears through calls such as `addVar(vtype=CONTINUOUS)` for a symbolic binary, or by using a generic nonnegative variable constructor where an integral container was required.

Example. Suppose a shipment-count variable n_{ij} represents the number of truck trips on arc (i, j) and should therefore be integral. If the solver code creates it with a continuous type, the executable model may use 2.4 trips. The symbolic model remains discrete, but the realized API object does not enforce that discreteness.

Consequence. The code changes the feasible set directly. This can produce lower costs, smoother flows, or artificial feasibility that exist only because the solver API no longer respects the symbolic variable type.

D.4.3 Index and Set Materialization Errors

Definition. The symbolic indices are correct, but loops, comprehensions, or filters in code fail to instantiate them correctly. Typical subtypes are dropped loop or index expansion, partial set expansion, and filtered index loss.

Boundary. These are not symbolic aggregation mistakes. The mathematical indices may be fully correct in M, yet the program only instantiates a subset of them, drops one nested loop, or filters legal objects away during code generation.

4.3.1 Dropped Loop or Index Expansion

Definition. A loop, nested index, or repeated expansion required by the symbolic model is omitted when code instantiates the solver object.

Typical erroneous form. A family that should be generated over (i, t), (i, j, k), or every period in a horizon is instantiated only over one dimension because one loop is missing. Typical code

patterns include using a one-level comprehension where a nested comprehension is required, placing a constraint constructor outside the inner loop, or calling a bulk-constructor such as `addConstrs` over an incomplete iterator.

Example. Suppose capacity constraints should hold for every facility i and period t: $\sum_j x_{ijt} \leq C_{it}, \quad \forall i, t$. If code loops over i but forgets the loop over t, then only one period or one default period is materialized in the solver model.

Consequence. The executable model under-enforces an entire slice of the symbolic family. This is a common way for time-coupled or scenario-coupled models to look valid while still missing most of their actual structure.

D.4.4 Solver and Formalization Mismatch

Definition. The code uses an implementation device that is incompatible with the symbolic formulation, such as missing a required linearization, choosing an incompatible solver, or using incorrect numerical options such as a bad Big-M realization.

Boundary. This category should be used when the symbolic form is acceptable in principle but the implementation layer fails to choose a compatible computational realization. It is therefore different from symbolic formalization errors, where the mathematical object itself is written incorrectly before code generation.

4.4.1 Missing Linearization in Code

Definition. The symbolic formulation requires a code-level linearization or auxiliary construction, but the implementation omits it.

Typical erroneous form. The symbolic model assumes auxiliary variables and linking constraints for fixed charges, binary-continuous products, absolute values, or max operators, but the code writes only the base variables and forgets the linearization device.

Example. If a facility-opening model requires $\sum_j x_{ij} \leq U_i y_i$ to linearize the activation of shipments by binary open variable y_i, then omitting that linking device in code allows shipments to occur regardless of whether the facility is opened.

Consequence. The executable model is no longer an equivalent realization of the symbolic one. It solves a relaxed or simply different problem because the intended logical or nonlinear structure was never encoded.

D.4.5 Post-Solve Extraction and Reporting Divergence

Definition. The code solves one model but reads out, reports, or post-processes another object. Typical subtypes include wrong solution variable readout, misreported objective value, and stale result-object or post-processing errors.

Boundary. These errors arise during post-solve extraction or reporting: even if the solver receives and solves the correct model, the reported decision or objective misrepresents its solution.

4.5.1 Wrong Solution Variable Readout

Definition. The code reads, exports, or displays the wrong variable family after solving, so the reported decision object differs from the one optimized.

Typical erroneous form. The solver optimizes one family of decision variables, but the reporting logic extracts auxiliary variables, reduced costs, slack variables, or a different indexed container and presents them as the final decision.

Example. In a routing model with binary arc decisions x_{ij}, reporting node-balance slacks or route-position helper variables as selected arcs misrepresents the intended routing decision.

Consequence. Even when optimization is correct, the reporting layer misrepresents the solution and can lead to incorrect operational decisions.