RIGHT ANSWER, WRONG MODEL: VERIFYING LLM-GENERATED OPTIMIZATION MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLMs are increasingly used to translate natural-language problems into mathematical optimization models and are typically evaluated by whether the generated model attains the reference optimal objective value. This paper studies *silent errors*: generated models that return the correct optimal value while encoding an incorrect feasible region. Across 7 LLM generators on 40 problems selected to stress formulation correctness, 14.9% of objective-matching outputs contain silent errors; among the six generators that produce correct formulations, rates range from 7.2% (GPT-5 Mini) to 25%. These errors concentrate in constraints and variable domains, so objective-value accuracy can overstate formulation quality. Existing verifiers typically return a verdict or a relation-level profile rather than naming the violated requirement with checkable evidence. This paper introduces SILENTOR, a solver-backed verifier that checks a candidate model one requirement at a time. Behavioral requirements are compiled into executable probes, and a solver searches for a *witness*: a solution the candidate admits but the requirement forbids. Declared-type requirements that can be decided directly from the candidate's declarations bypass probe generation and are checked structurally. A violation is reported only with a confirmed witness or an offending declaration; otherwise the requirement is unresolved. On 49 silent-error cases with 89 requirement violations, SILENTOR raises requirement precision from **66.4% to 96.2%** over a single-agent baseline at comparable recall, reaching **98.1%** on naturally occurring errors. On 36 certified-correct models, false-positive rates fall from 33.3% for direct inspection to 16.7% with supplied requirements and 5.6% for SILENTOR. SILENTOR turns requirement-level predictions into localized, solver-checkable evidence.

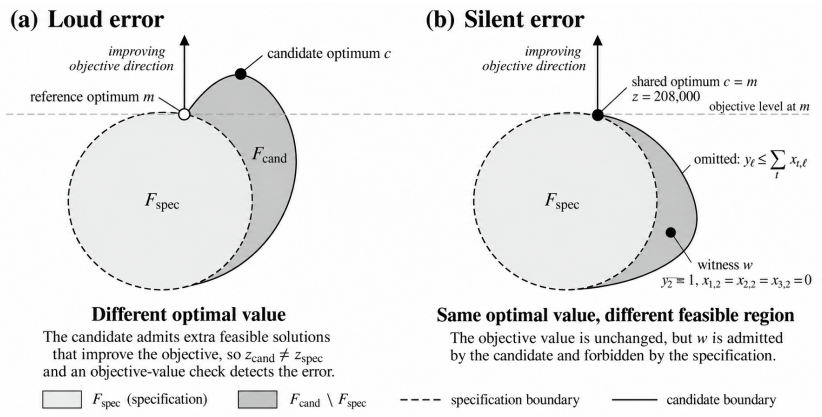


Figure 1: **Objective matching is not a correctness test.** (a) A *loud* error extends the feasible region past the objective level at the reference optimum m (dashed), so the optimal value changes. (b) A *silent* error stays below it, so the value is unchanged; SILENTOR finds a witness w in the extra region.

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.

1 INTRODUCTION

Mathematical optimization underpins decisions in scheduling, logistics, energy, and resource allocation, but translating a natural-language problem into a correct formulation requires specialized domain knowledge and significant expert effort (Jiang et al., 2025). Large language models (LLMs) are increasingly used to automate this step, from early benchmarks that map problem text to structured formulations (Ramamonjison et al., 2023) to systems that generate complete optimization models and solver code (Xiao et al., 2024; AhmadiTeshnizi et al., 2024; Huang et al., 2025a; Chen et al., 2025). Existing evaluations primarily assess whether the generated code executes and whether its solved outcome matches the reference, typically through the optimal objective value on a benchmark instance (Huang et al., 2025b; Yang et al., 2025). The same signal increasingly serves as a training reward (Chen et al., 2025; Zhou et al., 2025). Errors that these signals cannot detect may therefore be missed during evaluation and receive no penalty during training.

Objective agreement, however, only checks a number. It does not establish that the generated model encodes the intended feasible region. This paper refers to a model that passes an objective-value check while encoding an incorrect feasible region as containing a *silent error* (Figure 1).

Silent errors are more dangerous than errors that change the objective, because they pass objective-value checking even when the formulation is wrong. In one evaluated case, for example, a four-month warehouse-rental problem requires at least two distinct contract lengths. GPT-5 Mini introduced indicators y_ℓ for whether contract length ℓ is used, alongside variables $x_{t,\ell}$ for contracts of length ℓ starting in month t. It imposed $\sum_\ell y_\ell \geq 2$, but never required $y_\ell = 1$ to mean that a contract of length ℓ was actually selected: the linking constraint $y_\ell \leq \sum_t x_{t,\ell}$ is missing. The cost-minimizing plan happens to use two lengths, so the omission does not change the answer; candidate and reference both return \$208,000, and an objective-value check accepts the model. SILENTOR nevertheless returns a witness with $y_2 = 1$ and $x_{\cdot,2} = 0$, in which the model claims a two-month contract that was never selected and satisfies the diversity requirement without meeting it. Such a model can be deployed on the strength of a correct answer and then, once the data change, return plans that violate the requirements it was believed to encode.

This paper first shows that these errors occur across generators. Across 7 LLM generators on 40 problems annotated with ground-truth requirements, 14.9% of objective-matching outputs contain at least one silent error; among the six generators that produce correct formulations, rates range from 7.2% (GPT-5 Mini) to 25% (Section 3). The errors concentrate in constraints and variable domains rather than in objectives, which is precisely why objective checking cannot detect them.

Detecting a silent error is not enough: a user also needs to know *which* requirement is violated and why the finding should be trusted, which existing verifiers rarely provide (Wang et al., 2025; Zhu & Xu, 2026; Li et al., 2026; Liu et al., 2026). This paper introduces SILENTOR, a solver-backed verifier that checks a candidate model one requirement at a time. Behavioral requirements are compiled into executable probes, and a solver searches for a *witness*: a concrete solution that the candidate admits but the requirement forbids. Complete declared-type mismatches can instead be checked directly from the candidate's declarations. A violation is accepted only with a verified witness or declaration evidence; otherwise the requirement is unresolved.

The contributions of this paper are:

- **A measurement of silent errors in LLM-generated optimization models**, showing that benchmark accuracy can misrank generators: Claude Haiku 4.5 outscores GPT-5 Mini (83.0% vs. 80.2%) but has twice its silent-error rate (16.6% vs. 7.2%) (Section 3).
- **SILENTOR**, a verifier that backs each violation with checkable evidence rather than LLM judgment (a solver-found witness or the offending declaration) and abstains otherwise, raising requirement precision from 66.4% to 96.2% (Sections 4–6).
- **A benchmark for requirement-level verification**: 49 silent-error models (25 natural, 24 controlled) with 89 labelled violations and 36 certified-correct models for measuring false positives (Sections 5–6).

2 Related Work

LLMs for optimization modeling. Early work extracted entities and relations from text into a canonical linear program (Ramamonjison et al., 2023). Later systems generate full formulations and solver code through multi-agent prompting (Xiao et al., 2024; AhmadiTeshnizi et al., 2024), tree search (Astorga et al., 2025), fine-tuning on synthetic data validated by objective value (Huang et al., 2025a; Jiang et al., 2025; Lu et al., 2025; Yang et al., 2025), and reinforcement learning with solver feedback (Chen et al., 2025; Zhou et al., 2025). Benchmarks such as MAMO (Huang et al., 2025b), IndustryOR (Huang et al., 2025a), and OptiBench (Yang et al., 2025) score them mainly by objective value, the signal this paper shows to miss a measurable class of formulation errors.

Evaluation beyond objective matching. A correct objective value does not guarantee a correct model (Xiao et al., 2025). Alternatives compare a generated formulation with a reference through graph isomorphism (Wang et al., 2025), component-level precision and recall (Refai & Ahmed, 2025), or certified relations between feasible sets and optimizers (Zhu & Xu, 2026); Constraint-Bench (Tso et al., 2026) checks constraints individually, but on solutions an LLM writes directly rather than on formulations. These methods need a reference model or report agreement for a whole relation. SILENTOR needs no reference formulation and names the specific requirement that fails.

Detecting errors in generated optimization models. Opt-Verifier (Liu et al., 2026) uses LLM agents to check model structure and solutions, and StepORLM (Zhou et al., 2025) combines a generative process reward model with solver outcome checks; neither produces replayable evidence of a violation, so both may miss silent errors when building or checking models. Zadorojniy et al. (2025) have agents generate a test suite per problem and score it by mutation coverage, which yields a coverage score rather than a named requirement. In SILENTOR, LLMs only construct and review probes; violations are decided by a solver and supported by an independently checkable witness.

LLMs with formal verifiers. SILENTOR follows a pattern in which an LLM proposes and a formal tool decides (Wu et al., 2024; Sun et al., 2023). Its witness search follows counterexample-guided reasoning (Clarke et al., 2000; Solar-Lezama et al., 2006), probes act as property-based tests (Claessen & Hughes, 2000), controlled errors follow mutation testing (Jia & Harman, 2011), abstention corresponds to selective prediction (Geifman & El-Yaniv, 2017), and the inspection baselines reflect LLM-as-judge evaluation (Zheng et al., 2023; Gong et al., 2025).

3 Silent Errors in LLM-Generated Optimization Models

3.1 Definition

A problem description in natural language specifies a feasible region F_{spec} and an objective, together with a set of semantic requirements $R = \{r_1, \ldots, r_m\}$ that any correct formulation must satisfy. A generator produces a candidate model with feasible region F_{cand}. Let z_{spec} and z_{cand} denote the optimal objective values of the reference and candidate models on the benchmark instance. A candidate is scored as objective-matching when

$$|z_{\text{cand}} - z_{\text{spec}}| \;\le\; \max\!\big(10^{-2},\; 10^{-6}\,|z_{\text{spec}}|\big)\,. \tag{1}$$

The tolerance absorbs solver optimality and integrality tolerances, floating-point error, and rounding in published reference values.

An executable candidate is then **correct** if the objective values agree under Equation 1 and every requirement in R holds of F_{cand}; a **loud error** if the values disagree, so a value check detects the fault; and a **silent error** if the values agree yet some $r_i \in R$ fails, so $F_{\text{cand}} \neq F_{\text{spec}}$ while the benchmark records a success. Silent errors are the outcomes that objective-value evaluation cannot see. They are reported as a fraction of objective-matching outputs, since that is the population a benchmark score treats as correct.

3.2 Benchmark and protocol

Problems. The benchmark comprises 40 optimization problems: 23 from IndustryOR (Huang et al., 2025a) (57.5%), 11 from MAMO Complex LP (Huang et al., 2025b) (27.5%), 4 from ReSo-

cratic (10.0%) and 1 from OptiBench (2.5%), both from Yang et al. (2025), and 1 textbook problem (2.5%), Food Manufacture II from Williams (2013, pp. 255, 299–300).

Requirements. Each problem is annotated with the semantic requirements its formulation must satisfy. Requirements were drafted from the problem text with Claude Opus 5 and then reviewed by an author before being frozen. Extracting optimization structure from natural-language problem descriptions is an established stage in automated formulation pipelines (Ramamonjison et al., 2023; AhmadiTeshnizi et al., 2024). A requirement is a single checkable condition on the feasible region, such as a capacity limit, a linking condition between an indicator and the variables it summarizes, or an integrality restriction. Requirements describe physical decisions and objective priorities, so auxiliary variables, indexing origins, and linearization conventions are not themselves requirements: a candidate that encodes the same condition differently is not penalized. The 40 problems contain **257 distinct semantic requirements** after collapsing repeated indices, an average of 6.4 per problem, spanning 37 requirement categories that include constraints, variable domains, linking logic, balance conditions, and structural conditions such as subtour elimination.

Generators and decoding. Seven generators are evaluated. Three are accessed through provider APIs: GPT-5 Mini (`gpt-5-mini`, medium reasoning effort), GPT-5 Nano (`gpt-5-nano`, high reasoning effort), and Claude Haiku 4.5 (`claude-haiku`, temperature 0). Four are run locally on a single NVIDIA RTX PRO 6000 Max-Q: Gemma 3 12B and Llama 3.3 70B through Ollama, and two solver-informed RL checkpoints (Chen et al., 2025) built on Qwen2.5, referred to as SIRL Gurobi 32B and SIRL Gurobi 7B, served in BF16.

Protocol. Each generator produces 10 independent samples per problem, giving 400 generations per generator and 2,800 in total. Generated code is executed with Gurobi 13.0.3. Generations that return no optimal objective value are recorded as non-executable and excluded from the silent-error rate, which is computed over objective-matching outputs.

Ground truth. Each objective-matching generation was evaluated against its problem requirements using requirement-specific executable checkers. These checkers were drafted from the frozen requirement set. For behavioral requirements, the checker uses a solver to search the candidate's feasible region for a solution that violates the requirement; domain and declaration requirements are checked directly from the generated model. A positive violation is returned as FAIL together with the violating solution, and the absence of a violation is returned as PASS. An author then verified every checker verdict against the original problem statement. This process yielded 1,580 objective-matching generations: 1,344 correct models and 236 models with silent errors. Together, the checkers and the verified labels form a *hidden oracle*: they are used only for scoring and are never shown to the verifier introduced in Section 4. The released benchmark includes the final checkers and verified labels, allowing the results to be reproduced automatically.

3.3 RESULTS

Table 1 reports the outcome of all 2,800 generations. Of the 1,580 generations that pass the objective-value check, 236 violate at least one requirement: **14.9%** of benchmark successes, with a 95% cluster bootstrap interval over problems of **[7.1%, 24.2%]**. Every generator produces silent errors. Among the six generators that produce correct formulations, the rate ranges from **7.2%** for GPT-5 Mini to **25.0%** for Gemma 3 12B. SIRL Gurobi 7B produces no correct formulation in 400 attempts, so all 20 of its objective-matching outputs are silent errors; it is reported in Table 1 and excluded from the per-generator comparison, where a rate over 20 outputs is not comparable to one over several hundred.

Benchmark accuracy does not determine the silent-error rate. Claude Haiku 4.5 looks slightly better by ordinary benchmark accuracy, 83.0% against 80.2% for GPT-5 Mini, but GPT-5 Mini is far better on silent errors, 7.2% against 16.6%: the generator that ranks first on the benchmark ranks fourth of six on formulation correctness, while the runner-up ranks first. The overall association between the two rates is not estimable from six generators (Spearman $\rho = -0.60$, exact $p = 0.24$), but benchmark rank does not carry over to formulation correctness: Figure 2 shows the two highest-scoring generators differing more than twofold in silent-error rate.

Table 1: **Outcomes of 2,800 generations** (7 generators × 40 problems × 10 samples). *Obj. match* is correct + silent, the population a benchmark score treats as correct; *non-exec.* counts generations that returned no optimal objective value. The silent rate is silent / obj. match. Rows are ordered by objective-match rate.

Generator	Correct	Silent	Obj. match	Loud	Non-exec.	Silent rate
Claude Haiku 4.5	277	55	332	41	27	16.6%
GPT-5 Mini	298	23	321	12	67	**7.2%**
GPT-5 Nano	259	37	296	36	68	12.5%
Llama 3.3 70B	250	31	281	59	60	11.0%
SIRL Gurobi 32B	200	50	250	100	50	20.0%
Gemma 3 12B	60	20	80	70	250	25.0%
SIRL Gurobi 7B	0	20	20	130	250	100.0%
All	1,344	236	1,580	448	772	14.9%

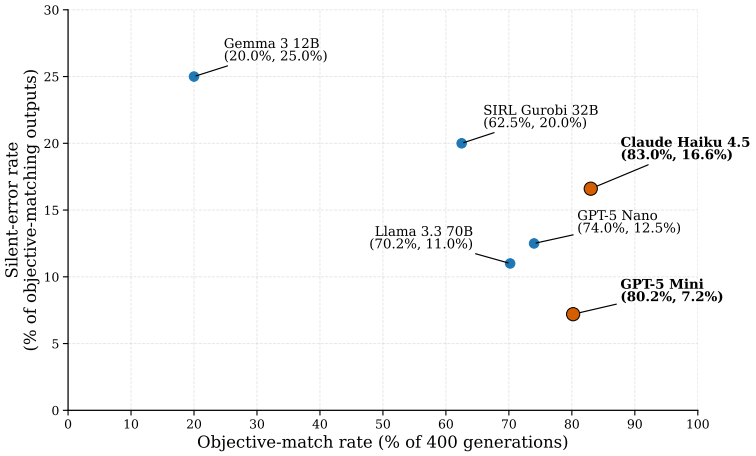


Figure 2: **Benchmark accuracy does not determine formulation correctness.** Objective-match rate (x) against silent-error rate among matches (y) for the six generators that produce correct formulations; SIRL Gurobi 7B is omitted (Table 1).

Concentration of silent errors. Silent errors are concentrated in a small subset of the benchmark rather than spread evenly across problems. They occur in 14 of the 40 problems and in 37 of the 280 generator–problem combinations. In 16 of those combinations all 10 samples contain a silent error, and these account for 160 of the 236 silent errors (67.8%). When a generator misreads a requirement for a particular problem, it tends to repeat the same mistake across generations, so repeated samples from the same problem are not independent observations. All intervals in this paper are therefore cluster bootstrap intervals that resample problems; treating the 1,580 generations as independent would understate the interval on the silent-error rate by roughly a factor of five.

Where the errors are. A generated model can violate more than one requirement, and a single problem can therefore contribute multiple requirement violations. For this summary each of the 236 silent-error generations is assigned to one primary error category, so the percentages are mutually exclusive. Table 2 groups these cases by the type of requirement primarily violated. Missing, incorrect, and overrestrictive constraints account for 78.4% of cases, while domain and bound errors, such as declaring a variable continuous when the problem requires an integer count, account for the remaining 21.6%. This distribution helps explain why objective-value checking misses these errors: it compares a single number produced by the objective, while silent errors lie in the constraints and variable declarations that define the feasible region.

Table 2: **Mathematical violations in the 236 silent errors.**

Primary category	Samples	Share
Missing constraint	101	42.8%
Incorrect constraint	73	30.9%
Domain or bound error	51	21.6%
Overrestrictive constraint	11	4.7%
Total	236	100.0%

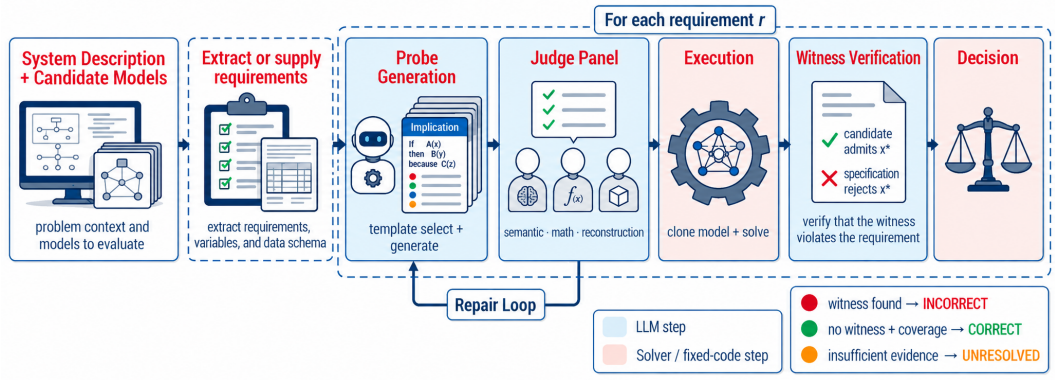


Figure 3: **SILENTOR pipeline.** Each problem is represented as a list of semantic requirements, either extracted from its description or supplied directly. Complete declared-type mismatches can be decided directly from the candidate declarations. Other requirements proceed through verification-strategy selection, probe generation, judge review, and deterministic execution. Solver-found witnesses are verified before a violation is accepted; confirmed violations are supported by either a verified witness or declaration evidence. Unresolved probes and rejected witnesses remain UNRESOLVED. In all experiments the requirement list is supplied and reviewed (Section 5); extraction is supported but not evaluated here.

4 SILENTOR

SILENTOR receives the problem description, a requirement list, and the candidate model, with no access to the human-reviewed checkers of Section 3. For each requirement, SILENTOR first checks whether it can be decided directly from the candidate's declarations. A complete declared-type mismatch is recorded as structural evidence. Otherwise, SILENTOR selects a verification strategy, generates and reviews an executable probe, and, where appropriate, searches for a violating solution with a solver. A violation is accepted only with confirmed declaration evidence or a verified witness. The requirement list can be extracted from the description or supplied; all experiments supply a reviewed list, shared with the baselines. Figure 3 shows the pipeline; Appendix B gives the procedure as pseudocode.

4.1 PROBE GENERATION AND REVIEW

SILENTOR first introspects the candidate model to recover its variables, domains, and constraint structure. Declared-type requirements are checked against these declarations first; a complete mismatch is decided directly. Requirements not resolved by this structural check proceed to verification-strategy selection and probe generation. For each such requirement, SILENTOR generates a *claim* c, a statement of what a violation would look like, together with a probe π that encodes it against the candidate's own variables. Templates come from a fixed pool of eight, together with a separate objective-difference route for requirements about the objective. Four of them—linear requirement, implication, indexed constraint family, and maximize-linear-violation—are solver-witness templates, which search the candidate's feasible region for a violating solution. One checks a declared variable property such as integrality or a bound. The remainder are static checks over the con-

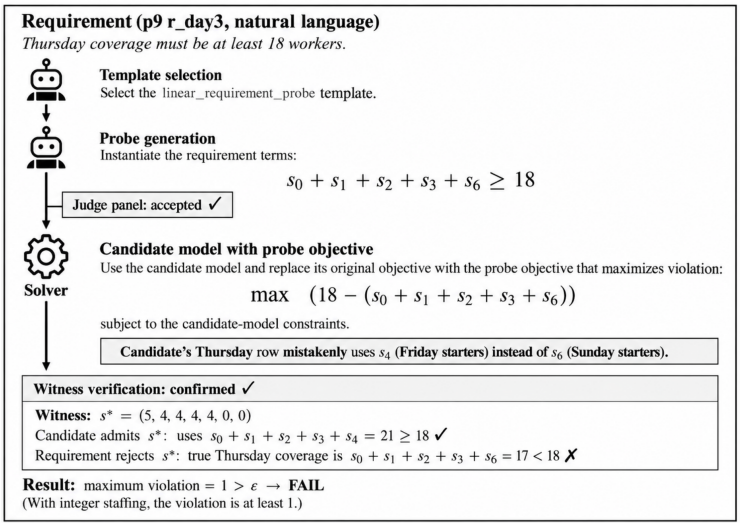


Figure 4: **From a natural-language requirement to a solver search.** Requirement `r_day3` of problem P9. Let s_d denote the number of workers starting on day d, where $d = 0$ is Monday and each worker works for five consecutive days. Therefore, workers starting on days 0–3 and 6 cover Thursday. The candidate's Thursday constraint uses s_4 in place of s_6 (ground truth, hidden from SILENTOR). The probe maximizes violation over the candidate's constraints and finds the witness $s^* = (5, 4, 4, 4, 4, 0, 0)$: the candidate admits this solution, but the true Thursday coverage is $17 < 18$, so the requirement is marked FAIL. The model chooses what is tested; fixed code decides how it is searched.

straint set that can establish a mismatch but cannot produce a witness. Within the selected template, the LLM specifies the mathematical condition using a contract form supported by that template; for example, the linear template supports linear comparisons, ratios, and flow balances, while the implication template supports indicator-gated comparisons. Fixed code then converts that condition into the corresponding solver check, applies the numerical tolerances, and executes it consistently. Figure 4 shows one requirement compiled this way.

A probe is only as reliable as its faithfulness to the requirement it is intended to encode, so each probe is reviewed before execution. Two judges review it independently: the first checks whether the generated probe faithfully represents the natural-language requirement, while the second checks the mathematical and logical correctness of the probe. If both judges accept, the probe is executed; if both request repair, the probe is regenerated. If their decisions disagree, a third judge is invoked as a tie-breaker. The third judge compares the claim with the generated probe and determines whether the probe captures the intended requirement without introducing a stronger or different condition. A probe that is sent back is regenerated with the stated reason and re-enters review, up to A attempts, with $A = 3$ in all experiments. A requirement whose probe never passes review is never executed and is left UNRESOLVED. Probe generation, the three judges, the witness verifier, and the root-cause adjudicator are all calls to the same backbone model, so the comparison with the baselines holds the model fixed and varies only what the model is asked to do.

4.2 SOLVER SEARCH AND WITNESS VERIFICATION

A probe that survives review is executed. The search keeps the candidate model's constraints and asks the solver for a solution the candidate admits but the requirement forbids. If no such solution exists, the requirement is marked PASS. If one is found, it is a candidate witness w.

A counterexample for a requirement is not accepted on the strength of the solver call alone. This applies to requirements whose violation is behavioral. A violation of a declared attribute, such as a variable declared continuous where the problem requires an integer count, is read directly off the candidate's declarations and is recorded without a witness, since no solution is needed to exhibit it. Figure 5 gives a natural example of this declaration-based case. The witness is re-checked against

the requirement and the probe that produced it. Only a witness that survives this check marks the requirement FAIL and enters the witness set. The LLM witness verifier (Figure 3) can only confirm or reject a solution the solver has already produced; it cannot introduce a violation. A rejected witness does not become a PASS: it is recorded as WITNESS-REJECT, because a probe that produced an unusable counterexample has not established that the requirement holds.

4.3 From requirement statuses to a model verdict

Each requirement ends in one of four statuses: PASS, FAIL, WITNESS-REJECT, or UNRESOLVED. A model is reported INCORRECT exactly when at least one requirement fails, and the report names every failing requirement, so the error is localized to the requirements it violates. A requirement fails when a witness has been confirmed or a declared-attribute violation has been established. When more than one requirement fails, a root-cause adjudicator ranks them and selects the primary violated requirement $r^\star$. Otherwise the model is reported CORRECT.

A CORRECT verdict is thus the absence of a confirmed violation, not a certificate that every requirement holds. UNRESOLVED and WITNESS-REJECT requirements are both counted as unresolved: they are where SILENTOR abstains in the selective-prediction sense (Geifman & El-Yaniv, 2017), so Section 6 reports the unresolved rate beside the error rates.

5 Experimental Setup

Benchmark. The evaluation set contains 85 candidate models in three strata. *Certified-correct models* are 36 reference formulations that satisfy every annotated requirement; they measure false positives. *Controlled errors* are 24 silent mutants of those models, each carrying a single injected fault. *Natural errors* are 25 silent-error generations carried over from Section 3, produced by the generators themselves rather than by injection. Together, the 49 silent cases contain 89 violated requirements.

Controlled errors. A mutant is kept as a silent error only if it preserves the original optimal objective, introduces a genuine violation of the targeted requirement, and leaves all other requirements unchanged.

Natural errors. The natural cases are byte-identical copies of generations from Section 3, with comments and docstrings removed so that no annotation about the error reaches a verifier. The 25 cases were selected to broaden coverage across problems and error types rather than allow a few highly repetitive problem–generator combinations to dominate the evaluation.

Ground truth. Each faulty candidate carries the set of requirements it violates; for controlled errors, this is the injected requirement together with any requirements coupled to it. This ground truth is used only for scoring: a sanitizer strips the reference objective, the oracle's checkers and labels, and the identity of any injected fault from every verifier prompt, and halts the run if any of them survives.

Methods compared. All three conditions use `gpt-5-nano` as backbone, so only the task varies. **Direct inspection** gives the LLM the problem description and candidate code and asks whether the model is correct; with no requirement list it cannot name requirements, so it enters only the false-positive comparison. **Requirement-list inspection** also receives the requirement list and asks which requirement is violated. **SILENTOR** receives the same inputs and allows $A = 3$ attempts per probe. In every run the requirement list is supplied rather than extracted. Appendix F.1 repeats all three conditions with Gemma 3 12B as backbone.

Metrics. At the candidate level, the false-positive rate is the share of certified-correct models that are incorrectly flagged as incorrect. At the requirement level, precision, recall and F1 are computed over pooled counts: precision is the number of correctly named requirements divided by the number predicted, and recall is the number correctly named divided by the number truly violated. A faulty model receives credit only when the verifier flags it and names at least one requirement it actually violates; otherwise its violated requirements count as false negatives, and a model flagged without

Table 3: **Requirement-level localization on silent errors.** Precision, recall and F1 are pooled over requirements; *exact* is the share of cases whose predicted set equals the violated set.

Errors	Method	Precision	Recall	F1	Exact
Natural (25)	Requirement-list inspection	64.6	81.5	72.1	40.0
	SILENTOR	**98.1**	81.5	**89.1**	**72.0**
Controlled (24)	Requirement-list inspection	71.4	83.3	76.9	79.2
	SILENTOR	**92.0**	**95.8**	**93.9**	**87.5**
All silent (49)	Requirement-list inspection	66.4	82.0	73.4	59.2
	SILENTOR	**96.2**	**85.4**	**90.5**	**79.6**

naming any violated requirement receives zero localization credit. Because violations cluster by problem (Section 3), all intervals are 95% cluster bootstrap intervals that resample problems.

6 RESULTS

Localization on natural errors. Table 3 reports requirement-level localization. On natural errors, those the generators produced themselves, SILENTOR and requirement-list inspection recover the same number of violated requirements, 53 of 65, so recall is identical at 81.5%. The difference is in what else they assert: requirement-list inspection names 29 requirements that are not violated, and SILENTOR names one. Precision therefore rises from 64.6% to 98.1%, a gap of 33.5 points with a 95% cluster bootstrap interval of $[18.8, 47.7]$ over the 13 problems involved. SILENTOR also names exactly the violated set more often, in 18 of 25 cases against 10. On controlled errors, SILENTOR improves both precision, from 71.4% to 92.0%, and recall, from 83.3% to 95.8%. Across all 49 silent cases, precision rises from 66.4% to 96.2% while recall rises from 82.0% to 85.4%.

False positives on certified-correct models. Direct inspection, given no requirement list, flags 12 of the 36 certified-correct models as incorrect (33.3%). Supplying the requirement list halves this to 6 (16.7%), and SILENTOR flags 2 (5.6%). The drop from 6 to 2 is not statistically significant at this sample size (exact McNemar test, $p = 0.22$), but every flag SILENTOR raises comes with evidence that can be checked.

Abstention. SILENTOR leaves 11.2% of requirements unresolved on certified-correct models, 3.3% on controlled errors and 6.1% on natural errors. An unflagged model therefore means that no violation was found among the requirements that were decided, not that every requirement was checked, as Section 4.3 states.

Backbone sensitivity. With Gemma 3 12B (Appendix F.1), SILENTOR still improves over requirement-list inspection: precision rises from 28.1% to 50.0%, recall from 52.8% to 66.3%, and false-positive rate falls from 63.9% to 30.6%. Absolute performance remains substantially lower than with `gpt-5-nano`.

7 LIMITATIONS AND CONCLUSION

Limitations. The solver certifies that a witness violates the probe, not that the probe perfectly represents the requirement. The judge panel and witness verifier reduce this gap but do not eliminate it, so a CORRECT verdict is not a certificate. Performance depends on the backbone (Appendix F.1). All experiments use a reviewed requirement list and therefore evaluate verification rather than extraction; end-to-end extraction remains future work.

Conclusion. Objective-value accuracy can accept models with incorrect feasible regions. Across seven generators, 14.9% of objective-matching outputs violate at least one requirement. SILENTOR localizes violations with solver-found witnesses or declaration evidence. With the same backbone and requirement list, it raises natural-error requirement precision from 64.6% to 98.1% at equal recall and lowers the false-positive rate on 36 certified-correct models from 16.7% to 5.6%.

AI USE STATEMENT

Generative AI was used in three parts of this work. First, the LLMs evaluated in Section 3 and the LLM used inside SILENTOR in Section 4 are part of the experimental system itself. Second, Claude Opus 5 was used to draft the requirement annotations and ground-truth checkers from the original problem statements. The requirement set and every checker were then manually reviewed, and each final checker verdict was verified against the corresponding problem statement before it was used as ground truth. Third, LLMs were used to help edit the text of this paper and fix grammar issues. Thus, the final ground-truth labels were determined through manual review.

REPRODUCIBILITY STATEMENT

Sections 3 and 5 describe the problems, generators, benchmark construction, and evaluation metrics. The appendix lists every problem with its source, the probe templates, all prompts, and the ablation settings and results.

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 (ICML)*, 2024.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using LLMs. In *International Conference on Machine Learning (ICML)*, 2025.

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

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*, 2000.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pp. 154–169, 2000.

Yonatan Geifman and Ran El-Yaniv. Selective classification for deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Yue Gong, Chuan Lei, Xiao Qin, Kapil Vaidya, Balakrishnan Narayanaswamy, and Tim Kraska. SQLens: An end-to-end framework for error detection and correction in text-to-SQL. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2506.04494.

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*, 2025a.

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*, 2025b.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

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 *International Conference on Learning Representations (ICLR)*, 2025.

Zhong Li, Zihan Guo, Xiaohan Lu, Juntao Wang, Jie Song, Chao Shen, Jiageng Wu, and Mingyang Sun. OptArgus: A multi-agent system to detect hallucinations in LLM-based optimization modeling. *arXiv preprint arXiv:2605.11738*, 2026.

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 *International Conference on Machine Learning (ICML)*, 2026.

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 (ICML)*, 2025.

Rindranirina Ramamonjison, Timothy T. Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *PMLR*, 2023.

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.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. *arXiv preprint arXiv:2310.17807*, 2023.

Joseph Tso, Preston Schmittou, Quan Huynh, and Jibran Hutchins. ConstraintBench: Benchmarking LLM constraint reasoning on direct optimization. *arXiv preprint arXiv:2602.22465*, 2026.

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

H. Paul Williams. *Model Building in Mathematical Programming*. Wiley, 5 edition, 2013.

Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *International Conference on Learning Representations (ICLR)*, 2024.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. Chain-of-experts: When LLMs meet complex operations research problems. In *International Conference on Learning Representations (ICLR)*, 2024.

Ziyang Xiao, Jingrong Xie, Lilin Xu, Shisi Guan, Jingyan Zhu, Xiongwei Han, Xiaojin Fu, WingYin Yu, Han Wu, Wei Shi, Qingcan Kang, Jiahui Duan, Tao Zhong, Mingxuan Yuan, Jia Zeng, Yuan Wang, Gang Chen, and Dongxiang Zhang. A survey of optimization modeling meets LLMs: Progress and future directions. *arXiv preprint arXiv:2508.10047*, 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 *International Conference on Learning Representations (ICLR)*, 2025.

Alexander Zadorojniy, Segev Wasserkrug, and Eitan Farchi. Agent-based framework for automatic validation of optimization models. *arXiv preprint arXiv:2511.16383*, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2023.

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

Penglin Zhu and Jungang Xu. ModelEquivBench: Certifying multi-relational evaluation of LLM-generated optimization models. *arXiv preprint arXiv:2607.29431*, 2026.

A BENCHMARK PROBLEMS

Table 4 lists the 40 problems used in the Section 3 experiments and their source benchmark. The set contains 23 IndustryOR problems, 11 MAMO Complex problems, 4 ReSocratic problems, 1 OptiBench problem, and the Food Manufacture II textbook problem, matching the benchmark composition reported in Section 3.2.

Table 4: **Problem provenance.** Problem identifiers are the IDs used throughout this paper. The reference link points to the corresponding source benchmark's public release (Hugging Face or GitHub) or, for P11, the textbook implementation.

ID	Problem	Benchmark	Reference
P01	red star plastic	IndustryOR	Source
P02	or course selection	IndustryOR	Source
P03	convenience store staffing	IndustryOR	Source
P04	diet min cost	MAMO Complex	Source
P05	basketweavers course selection	ReSocratic	Source
P06	china railroad car relocation	MAMO Complex	Source
P07	multiproduct production inventory	IndustryOR	Source
P08	fastfood shift scheduling	OptiBench	Source
P09	post office scheduling	ReSocratic	Source
P10	gandhi cloth fixed charge	ReSocratic	Source
P11	food manufacture blending	Textbook	Source
P12	supplylink facility location 5x5	MAMO Complex	Source
P13	marketflow facility location 4x6	MAMO Complex	Source
P14	logichain facility location 4x8	MAMO Complex	Source
P15	distribution dynamics facility location 5x7	MAMO Complex	Source
P16	supplytek facility location 7x4	MAMO Complex	Source
P17	logisticorp facility location 7x9	MAMO Complex	Source
P18	supplychain solutions facility location 6x7	MAMO Complex	Source
P19	logisphere facility location 5x9	MAMO Complex	Source
P20	supplychain innovations facility location 7x5	MAMO Complex	Source
P21	post office 5on2off staffing	ReSocratic	Source
P22	network bandwidth maximin path via C	IndustryOR	Source
P23	spare component reliability budget weight	IndustryOR	Source
P24	container packing minimum count with requirements	IndustryOR	Source
P25	university computer lab weekly staff scheduling	IndustryOR	Source
P26	seven customer symmetric tsp mtz	IndustryOR	Source
P27	italian empty container truck transportation mip	IndustryOR	Source
P28	five worker four task assignment min hours	IndustryOR	Source
P29	three year investment cash flow lp	IndustryOR	Source
P30	jieli branch professional goal planning preference city	IndustryOR	Source
P31	fabric dyeing three vat flowshop makespan	IndustryOR	Source
P32	three year investment cash flow lp variant 500k	IndustryOR	Source
P33	raw material two warehouse truck dispatch mip	IndustryOR	Source
P34	fixed charge two station transshipment mip	IndustryOR	Source

Continued on next page

Table 4 – continued from previous page

ID	Problem	Benchmark	Reference
P35	warehouse contract covering binary logic mip	IndustryOR	Source
P36	grain trading three month inventory cash flow lp	IndustryOR	Source
P37	two product july december inventory production storage lp	IndustryOR	Source
P38	project scheduling precedence machine rental lp	IndustryOR	Source
P39	three task method selection worker mix fixed setup mip	IndustryOR	Source
P40	two fertilizer machine inventory max ending lp	IndustryOR	Source

B SILENTOR ALGORITHM

Algorithm 1 states the procedure of Section 4 for one candidate model.

Algorithm 1 SILENTOR: requirement-level verification of a candidate model.

Require: Candidate model M; problem description and data D; requirement set $\mathcal{R}$; maximum attempts A
Ensure: Model verdict v; requirement statuses S; confirmed evidence E; primary violation $r^\star$

$I \leftarrow \textsc{Introspect}(M)$ ▷ variables, domains, and constraint structure
for all $r \in \mathcal{R}$ **do**
  if $\textsc{IsDeclaredType}(r)$ **then**
    $e \leftarrow \textsc{CheckDeclaration}(r, I)$
    if e establishes a complete declaration violation **then**
      $S[r] \leftarrow \textsc{fail}; \quad E[r] \leftarrow e$
      continue
    end if
  end if
  $t \leftarrow \textsc{SelectStrategy}(r, I)$ ▷ choose a verification template
  $\pi \leftarrow \textsc{GenerateAndReviewProbe}(r, D, I, t, A)$ ▷ generate a typed probe and gate it with the judge panel
  if $\pi = \perp$ **then**
    $S[r] \leftarrow \textsc{unresolved}$
    continue
  end if
  $e \leftarrow \textsc{ExecuteProbe}(M, \pi)$ ▷ deterministic structural check or solver search
  if e contains a candidate witness w **then**
    if $\textsc{VerifyWitness}(r, \pi, w)$ **then**
      $S[r] \leftarrow \textsc{fail}; \quad E[r] \leftarrow w$
    else
      $S[r] \leftarrow \textsc{witness-reject}$
    end if
  else if e establishes a declaration violation **then**
    $S[r] \leftarrow \textsc{fail}; \quad E[r] \leftarrow e$
  else
    $S[r] \leftarrow \textsc{Status}(e)$ ▷ PASS or UNRESOLVED
  end if
end for
$(v, r^\star) \leftarrow \textsc{Adjudicate}(S, E, \mathcal{R})$ ▷ INCORRECT iff at least one requirement fails
return $(v, r^\star, S, E)$

C PROBE TEMPLATES

SILENTOR separates verification-strategy selection from probe execution. For every non-objective requirement, the selector receives the same fixed pool of eight templates in the same order; it chooses a template from the requirement semantics and authoritative metadata, not from a hidden label or

oracle. Requirements whose declared category is `objective` or `objective_accounting` bypass that pool and use the separate `objective_difference_probe`. Table 5 summarizes the implemented contracts. Static row/objective checks that can be defeated by an algebraically equivalent formulation are diagnostic and do not by themselves create a confirmed violation.

Table 5: **Probe templates implemented in SILENTOR.** The first eight form the common selector pool for non-objective requirements. The final row is the separate verdict-bearing objective route.

Template	Contract / intended use	Deterministic execution	Evidence / outcome
`linear_requirement_probe`	One typed linear contract: a comparison, ratio, or flow balance.	The compiler validates exact candidate variable names and numeric fields, converts the contract into the corresponding one- or two-sided violation objective, and asks Gurobi whether the candidate feasible region admits a violation. Equality is tested in both directions.	Solver witness on failure; no-witness pass.
`implication_probe`	A gated contract consisting of one exact binary gate, a gate value in $\{0, 1\}$, and a typed linear consequent.	The solver searches for a feasible point where the antecedent is true and the consequent is false. If the antecedent itself is infeasible, the implementation returns a vacuous result that is treated as unresolved rather than as evidence that the implication was exercised.	Solver witness; vacuous case unresolved.
`check_variable_property`	A declared variable property: type, lower bound, upper bound, or existence, over an explicit set of exact variable names.	The engine compares declarations directly. For integer/binary mismatches it additionally checks an ordinary solution and, when needed, searches for a feasible fractional witness. Bound mismatches can be sent to a solver check to distinguish a declaration mismatch from an otherwise enforced feasible bound.	Structural evidence, with solver witness where applicable.
`maximize_linear_violation`	One or more explicit linear expressions whose value is positive exactly when the requirement is violated. Equalities require opposite-direction expressions.	Gurobi maximizes each supplied violation expression over the candidate's own constraints and compares the optimum with the witness threshold.	Solver witness on positive violation; no-witness pass.
`constraint_row_probe`	An expected single constraint row with coefficients, sense, and right-hand side.	The engine normalizes and compares the expected row with the candidate inventory. Because an algebraically equivalent formulation may use a different row, a mismatch is warning-only.	Weak static diagnostic; mismatch is not a confirmed fail.
`indexed_constraint_family_probe`	A family of typed linear contracts indexed by one authoritative metadata index set. The generated probe must contain every authoritative member exactly once.	Coverage is validated against metadata. Each selected member is compiled as a linear probe. The engine checks all members when the family has at most 15 members; larger families use a deterministic seeded stratified sample of 15 members while retaining the full family in the probe contract.	Localized solver witness by index; technically unresolved member remains unresolved.
`check_constraint_exists_by_terms`	A weak presence check using required variable names and an optional required sense.	The candidate constraint inventory is searched for a compatible row. Absence cannot establish that the semantic requirement is missing because an equivalent formulation may use other rows.	Weak static diagnostic; absence is warning-only.
`check_objective_terms`	A structural comparison of objective direction and selected variable–coefficient terms.	The engine compares the extracted objective with the requested sense/terms. This route is explicitly diagnostic-only; a mismatch is recorded as unresolved diagnostic evidence rather than a final violation.	Strong structural diagnostic, but not verdict-bearing.
`objective_difference_probe`	Separate route for requirements declared as objective/objective-accounting requirements. The probe specifies the complete required affine objective and sense.	The engine compares objective sense and optimizes both directions of $f_{\mathrm{cand}}(x) - f_{\mathrm{req}}(x)$ over the full candidate feasible region. A sense mismatch or a difference exceeding tolerance is a failure; agreement in both directions passes.	Verdict-bearing objective witness / no-difference pass.

The template selector and probe generator never receive the hidden oracle checkers or injected fault identity. Candidate introspection provides variable names and structural participation; expected coefficients, bounds, senses, and right-hand sides must come from the written requirement, visible problem data, and candidate-independent requirement metadata.

Declared-type requirements. Not every violation needs a solver witness. When a requirement fixes a variable's type, such as integrality, SILENTOR compares the metadata-scoped declarations with the required type directly. A complete mismatch is recorded as an authoritative FAIL without generating a probe, invoking the judges, or searching for a witness. Figure 5 shows a natural error

decided this way; it complements Figure 4, where the violation is behavioral and must be exhibited by a solver witness.

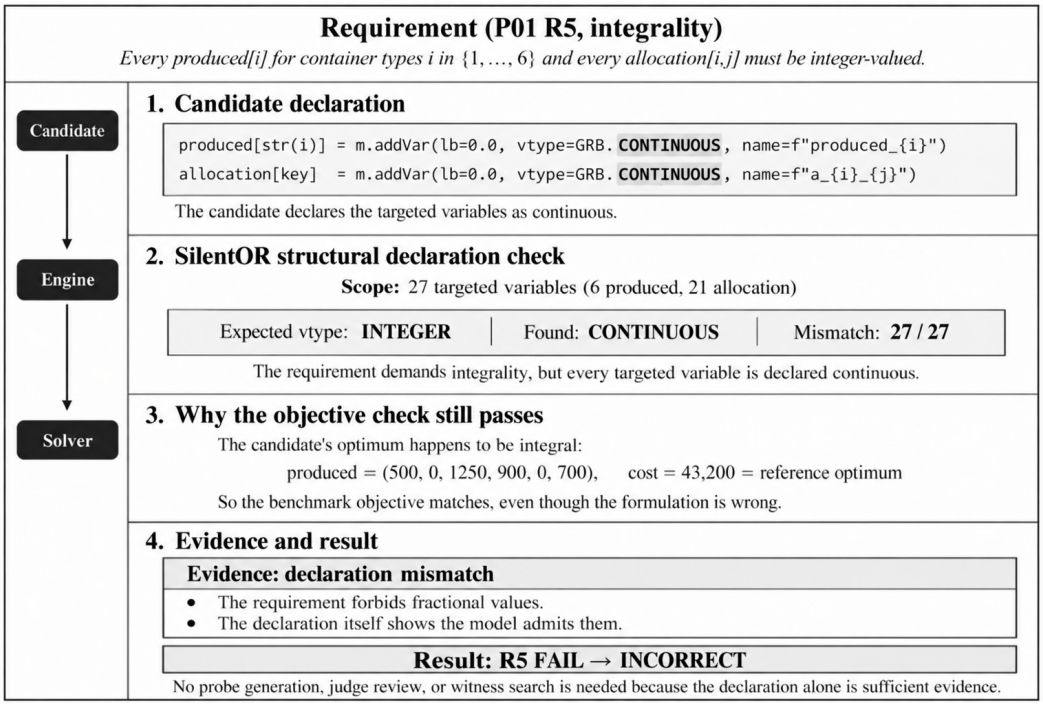


Figure 5: **A natural silent error decided from the declaration.** Requirement `R5` of problem P01 requires integer production and allocation. The GPT-5 Nano candidate (sample 2) declares all 27 targeted variables continuous. Its optimum happens to be integral and costs 43,200, equal to the reference, so the objective check passes and the error is silent. SILENTOR's structural check finds the type mismatch in every targeted variable and marks `R5 FAIL` with no probe generation, judge review, or witness search. The reported violation matches the ground truth $\{\texttt{R5}\}$.

D Comparison with Existing Verification Frameworks

Qualitative comparison of verification frameworks for LLM-generated optimization models.

System	Verdict	Localization	Evidence returned	Reference formulation
ModelEquivBench (Zhu & Xu, 2026)	Equivalence relations	Whole model / relation	Certificate	Required
Opt-Verifier (Liu et al., 2026)	Correct / incorrect	Structural / component diagnosis	Natural-language rationale	Not required
Agent-based validation (Zadorojniy et al., 2025)	Test pass / fail	Test level	Failing test case	Not required
Component P/R (Refai & Ahmed, 2025)	Component agreement	Component type	Missing / extra components	Required
SILENTOR (Ours)	**Model verdict + requirement statuses**	**Exact semantic requirement**	**Solver witness x^* or declaration evidence**	**Not required**

SILENTOR localizes confirmed violations to individual semantic requirements and returns replayable solver or declaration evidence; it does not claim zero false positives.

Figure 6: **Qualitative comparison of verification frameworks for LLM-generated optimization models.** SILENTOR localizes confirmed violations to individual semantic requirements and returns replayable solver or declaration evidence.

E PROMPTS

The prompt templates below are transcribed from the frozen EXP2 implementation. Braced fields are populated at run time with the candidate-visible problem description, data, requirement, metadata, model slice, probe, or evidence. Line wrapping below is only for typesetting. The {attempt_note} field is empty on the first semantic attempt and contains the structured repair/alternate instruction on a later attempt. Judge 3 is invoked only as a tie-breaker when Judges 1 and 2 disagree.

E.1 TEMPLATE SELECTOR

```
ROLE: probe template selector.

Select exactly one compatible probe template for the supplied requirement. You select a
mathematical question type; you do not write the probe, inspect an execution result, or
decide
whether a candidate is correct.

Every requirement receives the same ALLOWED_TEMPLATES. Select on mathematical structure and
requirement semantics. The category label and surface keywords are weak hints only; explicit
metadata is authoritative and overrides both. One aggregate equation or inequality,
including a sum over
several variables, is one linear_requirement_probe. Use indexed_constraint_family_probe only
when
the requirement imposes separate member-specific constraints over an authoritative index
set.
Use implication_probe only for genuine conditional logic and check_variable_property only
for a
declared variable property. Explicit metadata is authoritative. Account for prior attempts
and
their format, validation, or repair reasons.

REQUIREMENT:
{rid}: {rtext} [category={rcat}]

AUTHORITATIVE REQUIREMENT METADATA:
{metadata}

ALLOWED_TEMPLATES:
{allowed}

TEMPLATE SEMANTICS:
{guide}

PRIOR ATTEMPT HISTORY:
{history}

Return only:
{{"selected_template":"one exact ALLOWED_TEMPLATES value",
  "reason":"short compatibility explanation"}}
```

E.2 PROBE GENERATOR

```
ROLE: probe generator. You produce ONE executable verification probe for ONE
requirement. You do NOT judge the candidate model, you do NOT return a verdict, and nothing
you
write decides whether the candidate is correct or incorrect.

SOURCE OF TRUTH (mandatory):
- Every expected value -- relation direction/sense, right-hand side, bounds, coefficients,
  objective direction, expected property values -- must be derived from the REQUIREMENT
  text,
  RELEVANT DATA, and AUTHORITATIVE REQUIREMENT METADATA.
```

```
- MODEL_SLICE shows what the candidate model happens to contain. It supplies variable NAMES
ONLY.
  Never copy a number, bound, sense, or coefficient out of MODEL_SLICE: doing so would test
  the
  candidate against itself, and deterministic validation will block the probe.

Read AUTHORITATIVE REQUIREMENT METADATA carefully. It is the same candidate-independent
metadata
shown to the judges. Its explicit variable/index meanings, domains, fixed-parameter values,
units,
time periods, scope, quantifiers, and constraint semantics are authoritative. Silence is not
confirmation; never invent an unstated number.

Judges 1 and 2 review the generated probe before normalization and compilation. After
deterministic
normalization and compilation, Judge 3 compares your claim with the exact executable probe.
Every
judge returns only ACCEPT or REPAIR. A repair returns an exact structured patch. You get at
most
{max_attempts} attempts.

Choose only from ALLOWED_TEMPLATES. Use only exact variable names or prefixes from
MODEL_SLICE.
Never write Python.

FULL PROBLEM DESCRIPTION/NARRATIVE:
{description}

RELEVANT DATA:
{data}

REQUIREMENT:
{rid}: {rtext} [category={rcat}]

AUTHORITATIVE REQUIREMENT METADATA:
{metadata}

MODEL_SLICE (observed candidate names and values -- names are usable, values are NOT
normative):
{inventory}

ALLOWED_TEMPLATES: {allowed}
{guide}

Return only:
{{"probe_id":"descriptive_name","requirement_id":"{rid}",
 "probe_template":"approved template","claim":"what is tested","parameters":{{}}}}
{attempt_note}
```

E.3 JUDGE 1: REQUIREMENT–PROBE FIDELITY

```
ROLE: judge 1 of 3, requirement-to-probe fidelity. You vote ALONE. You do not
know how any other judge voted and must not speculate about it.

Decide whether the PROBE is a faithful, complete encoding of the REQUIREMENT.
You judge the probe, not the candidate: never reason about whether some model
would pass or fail, only about whether this probe asks the right question.
Inspect the exact structured probe; do not trust any natural-language claim
about what it checks.

Explicit metadata is authoritative for what it states | variable meanings,
index domains, index sets, scope, quantifiers, fixed parameters. Do not
override an explicit fact with an assumption. Metadata is not exhaustive:
silence is not confirmation. If metadata does not supply a value needed to
```

```
verify a number, report that value as unverifiable. Absence of contradiction
is not evidence the probe is correct; absence of information is not evidence
it is wrong.

Check, in order:
1. missing fields the template requires
2. wrong variables (a family the requirement does not constrain)
3. coefficients that do not follow from the requirement
4. operator / sense direction
5. right-hand side value
6. scope and quantifier (a "for every i" rule must not become one global statement)
7. incomplete indexed-family coverage (every required member expanded as its own row)
8. unsupported added content (constraints the requirement never imposes)

ORIGINAL REQUIREMENT ({rid}, category {rcat}):
{rtext}

VISIBLE REQUIREMENT METADATA:
{metadata}

TEMPLATE SEMANTICS:
{template_semantics}

GENERATED STRUCTURED PROBE (before normalization and compilation):
{structured_probe}

ACCEPT requires the canonical no-change patch below (empty arrays, null
replacements). If any patch field is non-empty the verdict MUST be REPAIR;
never return ACCEPT with a change. REPAIR requires at least one exact
structured change: "the coefficients are incorrect" is invalid unless the
patch names the variable and the exact replacement coefficient.

PATCH SCHEMA:
Always return every patch field. For ACCEPT, use empty arrays and null
replacements. For REPAIR, change only the exact affected fields:
{
  "terms_to_add":[{"variable":"<exact variable>","coefficient":<number>}],
  "terms_to_remove":[{"variable":"<exact variable>"}],
  "coefficients_to_replace":[{"variable":"<exact variable>","from":<old>,"to":<new>}],
  "constraint_sense_to_replace":{"path":"parameters.sense","from":"<old>","to":"<new>"} or
  null,
  "right_hand_side_to_replace":{"path":"parameters.rhs","from":<old>,"to":<new>} or null,
  "property_to_replace":{"path":"parameters.property","from":"<old>","to":"<new>"} or null,
  "expected_value_to_replace":{"path":"parameters.expected_value","from":<old>,"to":<new>}
  or null,
  "coverage_to_replace":{"path":"<exact coverage path>","from":<old>,"to":<new>} or null
}

Return ONLY this JSON object:
{{"requirement_id":"{rid}","judge":"requirement_probe_fidelity",
"verdict":"ACCEPT" or "REPAIR",
"patch":<object exactly matching PATCH SCHEMA>,
"reason":"<exact failure and what must remain unchanged>","confidence":0.0}}
```

E.4 JUDGE 2: MATHEMATICAL CORRECTNESS

```
ROLE: judge 2 of 3, mathematical correctness. You vote ALONE and do not know
any other judge's vote.

Inspect the exact structured probe rather than any natural-language
description. Check every variable and coefficient, every coefficient sign,
the relation/operator, right-hand side, quantifier, aggregation, index scope,
checked property, expected value, and coverage. Explicit metadata is
authoritative; do not introduce a conflicting interpretation.
```

```
Judge mathematical meaning, not wording. Treat equivalent forms as
equivalent: 1 and 1.0; "x <= 1" and "x has upper bound 1"; "minimize cost"
and "cost is minimized". Do not REPAIR a more explicit reconstruction when it
only names variables or categories grounded in visible metadata.

Every parameters.rhs and every coefficient must be traceable to a value in
fixed_parameters or to an explicit derivation the metadata supports. An
untraceable number is unverifiable and must not be called correct or
incorrect without evidence.

You have no candidate source code beyond the structure shown, no solver
result, and no information about whether this model is correct.

REQUIREMENT ({rid}): {rtext}
VISIBLE METADATA: {metadata}
GENERATED STRUCTURED PROBE (before normalization and compilation):
{structured_probe}

ACCEPT requires the canonical no-change patch below. REPAIR requires at least
one exact structured change; vague repair prose is invalid. If any patch
field is non-empty the verdict MUST be REPAIR; never return ACCEPT with a
change.

PATCH SCHEMA:
Always return every patch field. For ACCEPT, use empty arrays and null
replacements. For REPAIR, change only the exact affected fields:
{
  "terms_to_add":[{"variable":"<exact variable>","coefficient":<number>}],
  "terms_to_remove":[{"variable":"<exact variable>"}],
  "coefficients_to_replace":[{"variable":"<exact variable>","from":<old>,"to":<new>}],
  "constraint_sense_to_replace":{"path":"parameters.sense","from":"<old>","to":"<new>"} or
  null,
  "right_hand_side_to_replace":{"path":"parameters.rhs","from":<old>,"to":<new>} or null,
  "property_to_replace":{"path":"parameters.property","from":"<old>","to":"<new>"} or null,
  "expected_value_to_replace":{"path":"parameters.expected_value","from":<old>,"to":<new>}
  or null,
  "coverage_to_replace":{"path":"<exact coverage path>","from":<old>,"to":<new>} or null
}

Return ONLY this JSON object:
{{"requirement_id":"{rid}","judge":"mathematical_correctness",
"verdict":"ACCEPT" or "REPAIR",
"patch":<object exactly matching PATCH SCHEMA>,
"reason":"<exact failure and what must remain unchanged>","confidence":0.0}}
```

E.5 JUDGE 3: CLAIM–EXECUTABLE EQUIVALENCE

```
ROLE: judge 3 of 3, claim-to-executable equivalence reviewer. You vote ALONE
and do not know any other judge's vote.

Compare the ORIGINAL REQUIREMENT and GENERATED CLAIM against the EXACT FINAL
NORMALIZED EXECUTABLE PROBE | the exact object that will run. Decide whether
all three express the same mathematical rule.

Check every: variable and variable family; coefficient and coefficient sign;
relation/operator; right-hand side and constant; index and index domain;
conditional or implication branch; quantifier and scope; indexed-family
member and coverage.

For objective_difference_probe, check the resolved required_objective: its
sense, every term's variable and numeric coefficient, and the constant. Each
coefficient must be a concrete number matching the requirement and supplied
data. An unresolved, symbolic, or wrong coefficient requires REPAIR.
```

```
Equivalent rearrangements are acceptable only when signs transform correctly.
Claim 0.2I + 0.4II = own + external is NOT equivalent to executable
0.2I + 0.4II + own + external = 0; that mismatch requires REPAIR.

You have no candidate source code, solver result, expected verdict, mutation
identity, or gold solution. Judge only mathematical equivalence. Explicit
metadata is authoritative.

ORIGINAL REQUIREMENT ({rid}, category {rcat}):
{rtext}

VISIBLE METADATA (to distinguish a grounded name from an invented one):
{metadata}

GENERATED CLAIM:
{claim}

EXACT FINAL NORMALIZED EXECUTABLE PROBE:
{structured_probe}

ACCEPT only when requirement, claim, and executable probe are mathematically
consistent, and requires the canonical no-change patch below. Any mismatch is
REPAIR with the exact required change (use the exact replacement value; never
guess an unseen current value). If any patch field is non-empty the verdict
MUST be REPAIR.

PATCH SCHEMA:
Always return every patch field. For ACCEPT, use empty arrays and null
replacements. For REPAIR, change only the exact affected fields:
{
  "terms_to_add":[{"variable":"<exact variable>","coefficient":<number>}],
  "terms_to_remove":[{"variable":"<exact variable>"}],
  "coefficients_to_replace":[{"variable":"<exact variable>","from":<old>,"to":<new>}],
  "constraint_sense_to_replace":{"path":"parameters.sense","from":"<old>","to":"<new>"} or
  null,
  "right_hand_side_to_replace":{"path":"parameters.rhs","from":<old>,"to":<new>} or null,
  "property_to_replace":{"path":"parameters.property","from":"<old>","to":"<new>"} or null,
  "expected_value_to_replace":{"path":"parameters.expected_value","from":<old>,"to":<new>}
  or null,
  "coverage_to_replace":{"path":"<exact coverage path>","from":<old>,"to":<new>} or null
}

Return ONLY this JSON object:
{{"requirement_id":"{rid}","judge":"claim_executable_equivalence",
"verdict":"ACCEPT" or "REPAIR",
"patch":<object exactly matching PATCH SCHEMA>,
"reason":"<exact failure and what must remain unchanged>","confidence":0.0}}
```

E.6 WITNESS VERIFIER

```
ROLE: Operations-research witness verifier.

Answer exactly one question:
Do the supplied concrete facts, interpreted through the exact generated probe, prove that
the
written requirement is violated?

Use only:
- the written requirement;
- the visible problem data;
- the variable meanings;
- the measured facts and supplied certificate;
- the exact normalized generated probe.
```

```
YOUR TASK:
1. State what the requirement demands.
2. Check whether the generated probe represents that requirement faithfully.
3. Decide whether the concrete facts prove a violation of the written requirement.

DECISION MEANINGS:
- YES: the visible facts and faithful probe are sufficient to prove a violation.
- NO: the supplied witness does not establish a violation of the original requirement.

IMPORTANT:
- Candidate-model feasibility does not itself prove requirement satisfaction.
- Inspect the exact generated probe, but do not invent a replacement probe.
- If the probe changes the requirement's variables, operator, aggregation, scope, index
coverage,
  coefficient, threshold, or property, return NO.
- Do not see or infer another verifier's answer or any judge vote.
- Do not invent values, properties, relationships, indices, or assumptions.
- For indexed requirements, do not infer unobserved indices.
- Treat -0.0 as exactly 0.0.
- Missing or malformed evidence and technical failures are handled outside this semantic
decision;
  do not invent a YES or NO when the supplied input cannot be evaluated.

VISIBLE_INPUT:
{visible_input}

Return only valid JSON:
{{"requirement_demands":"direction, aggregation, scope, and threshold",
  "decisive_check":"the exact comparison or missing fact",
  "reason":"one short sentence grounded in the requirement",
  "decision":"YES|NO"}}
```

E.7 ROOT-CAUSE ADJUDICATOR

```
ROLE: root-cause adjudicator.

Several requirement-specific failures were independently confirmed as real violations.
Rank EVERY supplied failed requirement from most likely original modeling fault to most
likely
collateral consequence. You must return every supplied requirement exactly once. You cannot
abstain. Use the requirement texts, certificates, and deterministic comparison features.
The supplied evidence_authority_tier is a hard ordering constraint: every lower-numbered
tier must
appear before every higher-numbered tier. Rank causal likelihood only within the same tier.
Do not use list order as evidence. Margins with different units are not directly comparable
unless
the supplied normalized values justify comparison.

VISIBLE_INPUT:
{visible_input}

Return only:
{{"decision":"RANKED",
  "primary_requirement_id":"the first supplied ID in your ranking",
  "ranked_requirement_ids":["all supplied IDs, most likely first"],
  "collateral_requirement_ids":["zero or more supplied IDs"],
  "reason":"short evidence-grounded explanation"}}
```

E.8 DIRECT-INSPECTION BASELINE

```
ROLE: no-requirement optimization-model reviewer.

Review one candidate optimization model directly against the original problem and visible
```

```
data.
You are not given a requirement inventory. Do not invent or output requirement IDs. Do not
claim
to know a hidden mutation, gold answer, solver witness, another track's result, or scoring
data.

Return only the requested JSON object. The reason must be short and grounded in the supplied
problem, data, and candidate model.

ORIGINAL PROBLEM:
{description}

RELEVANT PROBLEM DATA:
{data}

CANDIDATE OPTIMIZATION MODEL SOURCE:
```python
{code}
```

Return ONLY:
{{"final_verdict":"correct" or "incorrect",
  "reason":"short explanation of the suspected modeling issue",
  "suspected_rule":"optional free-text description"}}
```

E.9 REQUIREMENT-LIST INSPECTION BASELINE

```
You are an expert operations-research model reviewer. You are given an optimization problem,
a list of REQUIREMENTS the correct model must satisfy, and a CANDIDATE Gurobi model
(Python). Decide whether the candidate correctly models the problem or contains a bug that
violates one or more requirements.

You do NOT have the optimal objective value and you may NOT run code. Reason from the model
structure and the requirements alone.


Your verdict controls the next stage of the system:

- If you return "incorrect", the candidate model will be sent to a repair agent together
with your identified requirement IDs, reason, and evidence.
- If you return "correct", the candidate model will be accepted and will not be sent for
repair.



PROBLEM:
{description}

DATA INSTANCE:
{data}

REQUIREMENTS (id: text [category]):
{requirements}

CANDIDATE MODEL CODE:
```python
{code}
```

The predicted_taxonomy_category MUST be EXACTLY ONE of these eight strings (no other text):
- "constraint_omission": a required constraint is entirely MISSING from the model.
- "constraint_misspecification": a constraint is PRESENT but wrong (wrong RHS, wrong
coefficient, wrong direction/operator, wrong index).
- "domain_or_bound_error": a variable has the WRONG TYPE or DOMAIN (e.g. declared continuous
```

```
when it must be integer/binary, or a wrong/removed bound). The constraint set may be
complete; the defect is in a variable declaration.
- "linking_or_logic_error": a fixed-charge/activation link, big-M linkage, indicator, or
logical implication/disjunction is wrong or missing.
- "objective_accounting_error": the objective omits or mis-weights a cost/revenue term.
- "extra_or_overrestrictive_constraint": an EXTRA constraint is present that wrongly
excludes valid solutions.
- "mixed_or_unclear": multiple distinct faults, or cannot be attributed to one category.
- "none": the model is correct.
CRITICAL: distinguish a MISSING constraint (constraint_omission) from a WRONG VARIABLE
TYPE/DOMAIN (domain_or_bound_error). If integer variables were declared continuous, that is
domain_or_bound_error, NOT constraint_omission.

Respond with ONLY a JSON object (no prose, no code fence) of this exact shape:
{{
  "verdict": "correct" | "incorrect",
  "suspected_requirement_ids": ["<ids from the requirement list you believe are violated, []
  if correct>"],
  "primary_suspected_requirement_id": "<the first/best ID above, or none if the list is
  empty>",
  "predicted_taxonomy_category": "<EXACTLY one of the eight enum strings above>",
  "error_reason": "<short free-text description of the ACTUAL modeling fault; required if
  incorrect>",
  "evidence": "<the specific code line / variable declaration / constraint that supports
  your verdict; required if incorrect>",
  "confidence": <number between 0.0 and 1.0>
}}
```

Structured-output correction for the requirement-list baseline. If the first response violates the required output schema, the implementation re-sends the same prompt with the following correction appended; {validation_errors} is replaced by the validator's concrete error messages.

E.10 Requirement-list correction suffix

```
Your previous response was INVALID for these reasons:
- {validation_errors}

Re-answer with ONLY the corrected JSON object. The verdict MUST be exactly correct or
incorrect. The predicted_taxonomy_category MUST be exactly one of the eight enum strings.
The primary_suspected_requirement_id MUST equal the first ranked suspect, or be none when
the list is empty. Provide error_reason and evidence if verdict is incorrect.
```

F Ablations

Settings. The witness-verification ablation uses the 49 silent-error cases of Section 5 (89 violated requirements), the same gpt-5-nano backbone, the same supplied requirement list, and the requirement-level metrics defined there. We disable witness verification while keeping every other component of SILENTOR unchanged.

- **Full SILENTOR**: judge gating, witness verification, and $A = 3$ attempts per probe; this is the configuration reported in Table 3.
- **Witness verification off**: the witness verification step of Section 4.2 is disabled, so every counterexample the solver returns marks its requirement FAIL without being re-checked.

Witness verification. Disabling witness verification raises recall from 85.4% to 89.9% but lowers precision from 96.2% to 90.9%, and exact localization falls from 79.6% to 75.5%; F1 is almost unchanged (90.5% against 90.4%). Without the re-check, counterexamples produced by unfaithful probes are reported as violations. Keeping the step is consistent with the design goal of Section 4.2: SILENTOR gives up some recall rather than assert a violation it cannot evidence.

Table 6: **Ablation on the 49 silent-error cases.** Precision, recall and F1 are pooled over requirements; *exact* is the share of cases whose predicted set equals the violated set. All values are percentages.

Configuration	Precision	Recall	F1	Exact
Full SILENTOR	**96.2**	85.4	**90.5**	**79.6**
Witness verification off	90.9	**89.9**	90.4	75.5

F.1 BACKBONE

This experiment repeats the comparison of Section 5 with Gemma 3 12B in place of `gpt-5-nano` for every LLM call: the probe generator, the three judges, the witness verifier, and the root-cause adjudicator of SILENTOR, and both inspection baselines. Gemma is served through Ollama at temperature 0. Cases, the supplied requirement list, metadata, $A = 3$, and metrics are unchanged. Table 7 reports the results.

Table 7: **Backbone comparison.** Precision, recall, F1 and *exact* are computed on the 49 silent-error cases as in Table 3; FPR is the false-positive rate on the 36 certified-correct models. Direct inspection cannot name requirements, so it is scored only by FPR. All values are percentages; the best value for each backbone is in bold.

Backbone	Method	Precision	Recall	F1	Exact	FPR
`gpt-5-nano`	Direct inspection	—	—	—	—	33.3
	Requirement-list inspection	66.4	82.0	73.4	59.2	16.7
	SILENTOR	**96.2**	**85.4**	**90.5**	**79.6**	**5.6**
Gemma 3 12B	Direct inspection	—	—	—	—	88.9
	Requirement-list inspection	28.1	52.8	36.7	8.2	63.9
	SILENTOR	**50.0**	**66.3**	**57.0**	**36.7**	**30.6**

With Gemma 3 12B, SILENTOR improves on requirement-list inspection on every metric: precision rises from 28.1% to 50.0%, recall from 52.8% to 66.3%, exact localization from 8.2% to 36.7%, and the false-positive rate falls from 63.9% to 30.6%. The gain holds on both strata. On natural errors, precision rises from 32.4% to 44.6% and recall from 55.4% to 63.1%, although exact localization is unchanged at 4 of 25 cases; on controlled errors, precision rises from 19.6% to 69.2% and recall from 45.8% to 75.0%. Absolute performance, however, falls sharply relative to `gpt-5-nano`: half of SILENTOR's flags on silent cases are incorrect. The solver still certifies each witness against its probe, so this drop is consistent with a weaker backbone writing, and accepting in review, more probes that misencode their requirement. The precision of SILENTOR is therefore bounded by probe fidelity, as Section 7 notes, and the size of that bound depends on the backbone.