# ReLoop: Structured Modeling and Behavioral Verification for Reliable LLM-Based Optimization

**Junbo Jacob Lian**[1,2,3,*] **Yujun Sun**[2,*] **Huiling Chen**[4]
**Chaoyu Zhang**[5] **Hanzhang Qin**[1,†] **Chung-Piaw Teo**[1,†]
[1] Institute of Operations Research and Analytics, National University of Singapore
[2] McCormick School of Engineering, Northwestern University
[3] Wenzhou Buyi Pharmacy Chain Co., Ltd.
[4] College of Computer Science and Artificial Intelligence, Wenzhou University
[5] Department of Decision Analytics and Operations, City University of Hong Kong
{jacoblian, yujunsun2026}@u.northwestern.edu, chenhuiling.jlu@gmail.com
cy.zhang@cityu.edu.hk, {hzqin, bizteocp}@nus.edu.sg

## Abstract

Large language models (LLMs) can translate natural-language problem descriptions into optimization code, but the code is prone to *silent failures*: it executes and returns a solver-feasible solution while encoding a semantically incorrect formulation. On compositional problems, the resulting feasibility–correctness gap reaches 90 percentage points. We introduce **ReLoop**, which combines two mechanisms. *Structured generation* decomposes code production into a four-stage reasoning chain (understand, formalize, synthesize, verify) to reduce formulation errors during generation. *Behavioral verification* detects the errors that remain by testing whether the formulation responds correctly to solver-based parameter perturbation, a signal that comes from the solver rather than from LLM self-review and requires no ground truth. The two mechanisms address different error structures: structured generation gives the largest gain on compositional problems (**+8.5pp** accuracy on RetailOpt-190 with Claude Opus 4.6), and behavioral verification gives its largest gain on localized defects (**+4.4pp** on MAMO-ComplexLP). With diagnostic execution recovery, ReLoop reaches **100% executable code** on Claude Opus 4.6, and relative to direct generation it raises or preserves every reported metric of the three chat-tuned foundation models on all three benchmarks. For the narrowly fine-tuned SFT model we test, the chain-of-thought prompt conflicts with its learned output format and lowers its accuracy on MAMO-ComplexLP; we document and analyze this interaction. We release **RetailOpt-190**, 190 compositional retail optimization scenarios in which several constraints interact. Code and benchmark: https://github.com/junbolian/ReLoop.

## 1 Introduction

Optimization solvers cannot tell a correct model from an incorrect one that happens to be feasible. For LLM-based optimization, this produces *silent failures*: code that executes without errors and returns solver-feasible solutions, yet encodes a semantically incorrect formulation. An inventory model that ignores perishability generates plans that hold expired stock, and a routing model that omits time-window constraints dispatches trucks that arrive after customers leave. In both cases the solver reports an "optimal" solution to the wrong problem.

[*] Co-first authors. [†] Corresponding authors.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Silent failures are frequent on compositional problems, in which several constraints interact (capacity, perishability, substitution, lead times). On such problems, DeepSeek-V3.2 reaches 91.1% solver feasibility but only 0.5% formulation correctness, a 90-point *feasibility–correctness gap* (§5.2). Existing verification does not detect these failures. Solver feedback catches syntax errors but not missing constraints, and LLM self-critique [1, 2] inherits the reasoning gaps that caused the errors. Environment-guided refinement [3] uses execution feedback but cannot distinguish a feasible but wrong model from a correct one, and execution-based reranking [4] requires ground-truth solutions, which are unavailable for a new problem.

Silent failures arise at the *modeling* stage, when the LLM translates the natural-language problem into a mathematical formulation, and they persist because these post-hoc checks cannot detect them. We therefore address them at two levels.

*At the generation level*, an expert operations research modeler does not write code directly: the modeler decomposes the problem, writes the mathematical model with an explicit type decision for each variable, and then implements and cross-checks it. We encode this workflow as a four-stage chain-of-thought prompt (understand, formalize, synthesize, verify), which reduces formulation errors before any external verification runs.

*At the verification level*, a correct optimization model satisfies behavioral invariants that can be tested without ground truth. (i) *Constraint presence*: shrinking a capacity limit to near zero should change the optimal objective, so no change indicates that the constraint is likely missing. (ii) *Objective completeness*: perturbing a cost coefficient should shift the optimal objective, so no change indicates that the term is likely omitted. Declarative self-review inherits the errors of the generating LLM [2], whereas solver-based perturbation provides an *external semantic signal*, because the response is computed by the solver, not judged by the LLM.

**ReLoop** implements both levels with three components: (1) *structured generation* with the four-stage prompt (§3.2); (2) *two-layer behavioral verification*, consisting of execution recovery with IIS-based diagnostics (L1) and solver-based perturbation testing (L2); and (3) *diagnosis-guided repair* with a regression guard.

**Contributions.**

1. We propose *structured generation* for optimization code, which decomposes modeling into four stages with explicit variable-type reasoning and a grounded completeness check. On compositional problems, structured reasoning is the largest source of accuracy gains for the foundation models; we attribute this gain to explicit structured reasoning in general rather than to the OR-specific staging (§3.2, §5.4).
2. We formalize *behavioral verification*, which tests whether a formulation responds correctly to solver-based parameter perturbation. To our knowledge, it is the first approach that uses solver-response perturbation as an external, non-LLM semantic signal for detecting silent failures without ground truth. It is the largest single source of accuracy gains on problems with localized defects (§3.3, §5.4).
3. We design **ReLoop**, a pipeline that combines structured generation, execution recovery, and behavioral testing with regression-guarded repair, and we release **RetailOpt-190**, 190 compositional retail optimization scenarios (§3.3, §4).
4. A cross-benchmark ablation shows that structured generation and behavioral verification are *complementary*, since each gives the larger gain under a different error structure, and that CoT can conflict with domain-specific fine-tuning (§5).

## 2 Related Work

**LLM-Based Optimization Modeling.** Methods that translate natural language into mathematical programs use prompting [5, 6], supervised fine-tuning [7–9], reinforcement learning with solver feedback [10], and agentic workflows [11]. All of them improve *generation quality*, through training or prompting, and are complementary to ReLoop. ReLoop contributes a domain-specific structured generation strategy, which is prompt-based and applies to any model, and adds post-generation *behavioral verification*, which detects when a feasible model is semantically incorrect.

Table 1: Verification approaches for LLM-generated optimization. †Requires ground-truth labels.

| Method | Detects Silent Failures | External Signal | No Ground Truth | Iterative Repair |
|---|---|---|---|---|
| Solver feedback [10] | ✗ | ✓ | ✓ | ✗ |
| OptiChat [12] | ✗ | ✓ | ✓ | ✓ |
| Self-Refine [1] | ✗ | ✗ | ✓ | ✓ |
| Reflexion [3] | ✗ | ✓ | ✓ | ✓ |
| SAC-Opt [14] | ✓ | ✗ | ✓ | ✓ |
| LEVER† [4] | ✓ | ✓ | ✗ | ✗ |
| **ReLoop (ours)** | ✓ | ✓ | ✓ | ✓ |

**Verification and Self-Correction.** Table 1 groups existing verification methods by paradigm, and each paradigm has a limitation for optimization. *Solver-based feedback* [10, 12] cannot detect a model that is feasible but wrong, which is exactly a silent failure. *LLM self-critique* [1] fails because, without external feedback, a model cannot identify errors caused by its own reasoning gaps [2], and these are the gaps that produced the silent failure. *Environment-guided refinement* [3] incorporates execution feedback, but the solver status and the objective value do not distinguish a correct model from a feasible but wrong one. *Execution-based reranking* relies on training-time labels [4] or generated test cases [13]; in optimization, ground-truth programs are scarce, and generating reliable test cases requires the semantic understanding that the LLM lacks. *Semantic-anchor alignment* (SAC-Opt) [14] reconstructs semantic anchors from the generated code, aligns them with anchors extracted from the problem, and applies corrections where the two sets disagree; this backward-guided check operates in the LLM's semantic space. Because the LLM reconstructs both anchor sets, their agreement can depend on the same misreading that caused an error, which is the correlated-error mechanism identified for self-review [2]. The two approaches are complementary: SAC-Opt compares semantic representations produced by the LLM, whereas ReLoop tests solver behavior. ReLoop uses solver-based perturbation as an external semantic signal, which provides the independent feedback that Huang et al. [2] identify as necessary for reliable correction.

**Sensitivity Analysis and Program Verification.** Sensitivity analysis studies how the optimum changes with the parameters [15, 16]. We use it for verification: instead of *interpreting* sensitivity, we test whether it *exists*, because a parameter that should affect the optimum but produces zero sensitivity indicates a missing component. Testing expected sensitivity patterns is an instance of metamorphic testing [17–19]: Assumption 1 defines metamorphic relations specific to optimization, with the solver as the test oracle. Classical program verification requires formal specifications [20, 21]; our approach does not, because it tests properties that any correct optimization model satisfies.

**Benchmarks.** Existing benchmarks, including NL4Opt [22], MAMO [23], IndustryOR [7], and OptMATH [9], mostly test constraints in isolation: they evaluate whether a model can encode a single constraint type rather than several interacting ones. RetailOpt-190 instead composes 8 scenario families in which multiple constraints must bind jointly (§4).

## 3 Method

ReLoop combines two mechanisms. *Structured generation* (§3.2) decomposes code production into a four-stage reasoning chain that follows expert modeling practice, which reduces formulation errors. *Behavioral verification* (§3.3) detects the errors that remain by testing how the formulation responds to solver-based perturbation. Execution recovery (L1) and diagnosis-guided repair (§3.4) complete the pipeline. Figure 1 shows the architecture, and Algorithm 1 (Appendix E) states it formally.

### 3.1 Problem Statement

Let $x$ denote a natural-language optimization problem description containing all parameter values.

**Definition 1** (Semantic Correctness)**.** Code $C$ is *semantically correct* for $x$ if the optimization model encoded by $C$ captures all constraints and objective terms specified in $x$, i.e., the feasible region and objective function match the intended formulation.

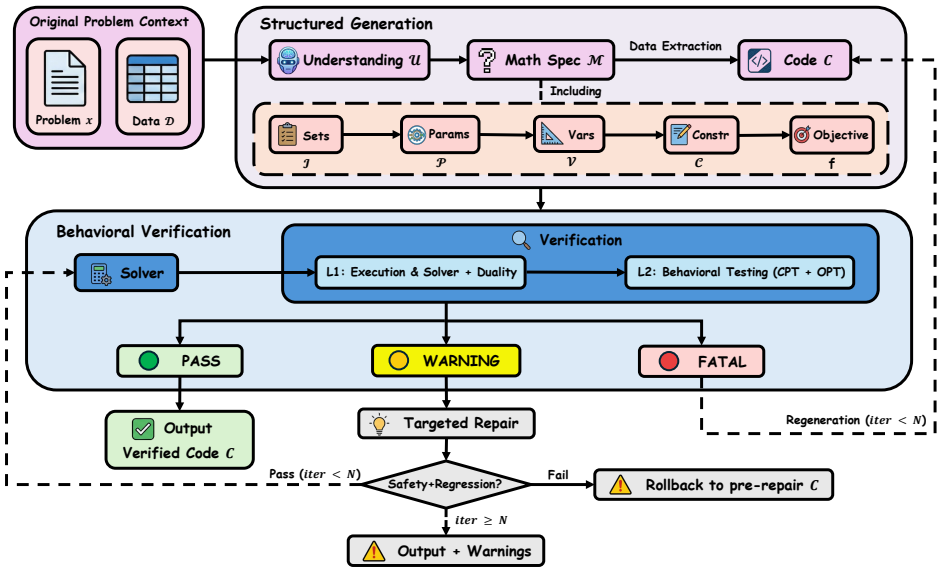


Figure 1: ReLoop overview. **Structured Generation** follows expert modeling practice: understand the problem, formalize the mathematical model with explicit variable-type reasoning, synthesize Gurobi code with data extraction, and self-verify completeness. **Behavioral Verification**: L1 checks execution correctness (FATAL blocks output); L2 tests constraint (CPT) and objective (OPT) presence via solver-based perturbation (WARNING/PASS). **Diagnosis-Guided Repair**: FATAL triggers diagnosis-specific regeneration (syntax tracebacks, IIS, or unbounded rays); WARNING triggers targeted repair with regression rollback. When the budget $N$ is exhausted, ReLoop returns the best verified code.

**Definition 2** (Silent Failure)**.** Code $C$ is a *silent failure* if it (i) executes without error, (ii) returns a solver-feasible solution, yet (iii) is not semantically correct.

Standard software testing checks execution success and solution feasibility, so it cannot detect silent failures. Our verification approach uses a behavioral invariant of correct optimization models:

**Assumption 1** (Perturbation Sensitivity)**.** *Let $z^*(\theta)$ denote the optimal objective of a correctly formulated model parameterized by $\theta$, and let the response ratio to a perturbation $\theta_i \to \tilde{\theta}_i$ be $r_i = |z^*(\tilde{\theta}_i) - z^*(\theta)| \,/\, |z^*(\theta)|$. Assume $\theta_i$ governs a component that (i) binds, or can be forced to bind under sufficiently extreme perturbation, and (ii) carries a non-negligible objective contribution. Then $r_i \gg 0$ under extreme perturbation. This is a design assumption rather than a theorem; the graduated thresholds of §3.3.2 operationalize it, and two guards in the main pipeline handle its edge cases: when $z^*(\theta) = 0$ the implementation falls back to the absolute change $|z^*(\tilde{\theta}_i) - z^*(\theta)|$, and perturbation-induced infeasibility is treated as a strong* PASS *signal (Appendix E).*

The contrapositive gives our detection criterion: if perturbing a parameter that *should* affect the objective produces a negligible change, the corresponding model component is likely absent from the generated code. Extreme perturbation can also force natively inactive constraints to bind (e.g., scaling capacity by $\times 0.001$), so the test detects both natively active and conditionally bindable components.

### 3.2 Structured Generation

Direct code generation combines understanding, formalization, and implementation in a single output, so an error at any stage passes into the code unchecked. We decompose generation into four stages executed in a single LLM call:

$$x \xrightarrow{\text{understand}} \mathcal{U} \xrightarrow{\text{formalize}} \mathcal{M} \xrightarrow{\text{synthesize}} \hat{C} \xrightarrow{\text{verify}} C \tag{1}$$

**Stage 1 (Understand).** Extract the objective direction, decision variables, constraints, and parameters from $x$, producing a structured interpretation $\mathcal{U}$ that makes implicit assumptions explicit.

Table 2: Severity matrix across verification layers. (†) Reference only: marked "do not fix."

| Layer | Check | Severity | Action |
|---|---|---|---|
| L1: Execution (blocking) | Syntax / runtime error | FATAL | Regenerate |
| | Infeasible | FATAL | Regen. (IIS) |
| | Unbounded | FATAL | Regen. (ray) |
| | Duality gap† | INFO | None |
| L2: CPT (diagnostic) | Missing ($r < 5\%$) | WARNING | Repair |
| | Uncertain ($5\% \leq r \leq 30\%$)† | INFO | None |
| L2: OPT (diagnostic) | Missing ($r < 5\%$) | WARNING | Repair |
| | Uncertain ($5\% \leq r \leq 30\%$)† | INFO | None |

**Stage 2 (Formalize).** Transform $\mathcal{U}$ into a mathematical specification $\mathcal{M} = (\mathcal{I}, \mathcal{P}, \mathcal{V}, \mathcal{C}, f)$: index sets $\mathcal{I}$, parameters $\mathcal{P}$, decision variables $\mathcal{V}$ with explicit type reasoning, constraints $\mathcal{C}$, and objective $f$. The prompt requires the LLM to decide from the physical context whether each variable is continuous, integer, or binary (e.g., "can you order 2.7 pallets?"). This step targets a common failure in which models default to continuous variables for discrete decisions and thereby relax integrality.

**Stage 3 (Synthesize).** Generate executable Gurobi code $\hat{C}$ from $\mathcal{M}$ that reads all data through `data["key"]` dictionary lookups rather than hardcoded literals. L2 depends on this structure, because its perturbation tests modify parameter values in the runtime data dictionary. We therefore first prompt the LLM to extract all numerical parameters from $x$ into a JSON dictionary, and the generated code references this pre-loaded dictionary. If extraction fails (e.g., malformed JSON), we fall back to self-contained generation, in which the LLM embeds the data in the code; the perturbation engine detects and handles both patterns (Appendix E.4).

**Stage 4 (Verify Completeness).** Cross-check the generated code $\hat{C}$ against the original problem $x$: are all cost and revenue terms in the objective, are all constraints implemented, and are all data values accessed correctly? If the check finds a discrepancy, the model fixes the code before returning the final $C$, so these errors are corrected before verification. Stage 4 is a *grounded* completeness check rather than open-ended self-critique, because it compares the code with the original problem text $x$ through an explicit checklist. The negative results on LLM self-correction concern identifying reasoning errors *without external feedback* [2]. The external, non-LLM signal in ReLoop is L2's solver response (§3.3.2).

### 3.3 Two-Layer Behavioral Verification

Two principles govern the architecture. (1) **Only L1 blocks output.** L2 provides diagnostics but never discards a valid solution, so the pipeline always returns a solution that has passed L1. (2) **Conservative repair.** Only high-confidence issues (WARNING) trigger repair; ambiguous signals (INFO) are logged for reference only, so false positives do not trigger repairs that introduce regressions. Table 2 summarizes the severity matrix.

#### 3.3.1 L1: Execution Verification (Blocking)

L1 sequentially checks syntax (AST parsing), runtime completion, and solver status. It is the *only blocking layer*: a FATAL result triggers regeneration with error feedback (up to $N$ attempts). When L1 passes, we record the baseline objective $z^* = \text{OPT}(C)$.

For INFEASIBLE status, L1 computes the *Irreducible Inconsistent Subsystem* (IIS), a minimal set of conflicting constraints, and passes their names and bounds to the LLM. The regeneration prompt therefore states which constraints conflict instead of only reporting infeasibility. For UNBOUNDED status, L1 reports the variables of the unbounded ray to guide the correction of variable bounds.

Table 3: Detection scope of L2 behavioral testing.

| Defect | L2 behavior |
|---|---|
| Missing parameter-governed constraint | Detected (CPT) |
| Missing objective term | Detected (OPT) |
| Present-but-wrong constraint reusing the same parameters | Partial / not detected |
| Coefficient-magnitude errors | Not detected |
| Equivalent-looking but structurally different decompositions | Not detected |
| Errors inducing infeasibility or unboundedness | Handled by L1 (IIS / rays) |

#### 3.3.2 L2: Behavioral Testing (CPT + OPT)

L2 detects missing formulation components, that is, constraints or objective terms that the problem description requires but the generated code lacks. L2 avoids the self-consistency limit of LLM self-review [1, 2] by assigning detection to the *solver*. The LLM only extracts candidate constraints and objective terms, and detection uses parameter perturbation and the solver's response, which do not depend on the LLM's reasoning.

**Detection mechanism.** L2 implements Assumption 1 with two symmetric modules. **Constraint Presence Testing (CPT)** targets missing constraints: for each candidate extracted by the LLM (annotated with physical type: capacity, demand, etc.), CPT applies extreme perturbation to the governing parameter (capacity $\times 0.001$, demand $\times 100$, other $\times 0.01$). **Objective Presence Testing (OPT)** targets missing cost/revenue terms with analogous perturbation (cost $\times 0.001$, revenue $\times 100$, other $\times 0.01$).

**Graduated thresholds.** Both modules measure the objective change ratio $r = |z' - z^*| \,/\, |z^*|$ and classify:

- $r < \tau_\ell = 5\%$: WARNING (likely *missing*), which triggers repair.
- $\tau_\ell \leq r \leq \tau_h = 30\%$: INFO (*uncertain*), which is logged without repair.
- $r > \tau_h$ or infeasibility: PASS (*present*), which confirms that the component is active.

The asymmetric $[\tau_\ell, \tau_h]$ buffer favors under-detection over over-repair, because false positives trigger repairs that can cause regressions; accuracy varies by $<1\%$ across $\tau_\ell \in [1\%, 10\%]$ and $\tau_h \in [20\%, 50\%]$.

**Detection scope.** Table 3 lists which defects the mechanism can and cannot detect, and Appendix I measures these boundaries on defects whose labels are known by construction.

### 3.4 Diagnosis-Guided Repair

All layers output `Diagnostic` objects in a common format that specify the severity, the target component, and the evidence (e.g., "constraint `capacity_limit`: perturbation $\times 0.001$ caused only 0.3% objective change"). FATAL triggers *regeneration* with the error context (IIS constraints, unbounded rays); WARNING triggers *targeted repair* guided by the specific L2 diagnostics, and INFO items are explicitly marked "do not fix" to prevent over-correction.

**Safety and regression guards.** Three mechanisms prevent regressions caused by repair. (1) A *safety check* blocks reassignment of data variables and dangerous imports, because we observed repair LLMs fabricating values that corrupt the problem. (2) A *regression guard* rolls back any repair that crashes, degrades the solver status, or shifts the objective by $>\tau_r = 4\%$. It mainly catches failures of the repair step (data fabrication, fundamental restructuring) rather than legitimate constraint additions, because adding a component that L2 flagged as a localized defect typically shifts the objective by less than $\tau_r$. (3) A *skip guard* bypasses repair when no WARNING exists. Algorithm 1 (Appendix E) states the procedure.

Table 4: Comparison with existing benchmarks.

| Benchmark | Domain | Inst. | Multi-period | Compositional | Data-Code Sep. |
|---|---|---|---|---|---|
| NL4Opt [22] | Generic | 289 | Few | ✗ | ✗ |
| MAMO [23] | Generic | ∼800 | Some | ✗ | ✗ |
| IndustryOR [7] | Mixed | 100 | Some | ✗ | ✗ |
| OptMATH [9] | Mixed | ∼360 | Some | ✗ | ✗ |
| **RetailOpt-190** | **Retail** | 190 | **All** | ✓ | ✓ |

Table 5: RetailOpt-190 scenario families ($38 \times 5 = 190$ instances).

| ID | Family | Arch. | Key Mechanisms |
|---|---|---|---|
| F1 | Core Operations | 4 | Multi-period inventory, perishability, lost sales |
| F2 | Assortment & Substitution | 6 | Product substitution, promotions, price bands |
| F3 | Resource Constraints | 4 | Storage bottleneck, supply limits, volumetric |
| F4 | Demand Dynamics | 6 | Demand surge, supply risk, quality holds |
| F5 | Feasibility Stress | 4 | Impossible demand, storage overflow, service traps |
| F6 | Discrete Logistics | 4 | Lead time, MOQ, pack size, fixed order cost |
| F7 | Network & Multi-Echelon | 6 | Transshipment, hub-spoke, multi-sourcing |
| F8 | Omni-channel | 4 | Reverse logistics, labor, ship-from-store |

## 4 RetailOpt-190 Benchmark

**RetailOpt-190** is a benchmark for *compositional constraint reasoning* (Table 4). All 190 instances are multi-period retail inventory optimization problems in which a retailer decides *how much to order*, *when to order*, and *how to allocate* products across periods to minimize total cost subject to interacting operational constraints. Each instance has 3 products, 5 locations, and 20 periods, except that one archetype uses 52 periods and one multi-echelon archetype uses 6 locations. An instance specifies 20+ parameters (demand forecasts, unit costs, storage capacities, shelf lives, lead times, supplier limits) and 10–30 constraints that must bind jointly. The difficulty comes from constraint *interactions* rather than linguistic complexity: for example, perishability limits effective inventory, which tightens storage constraints and forces earlier ordering that conflicts with lead times. Each instance is self-contained, since the natural-language description specifies all parameters, and a correct solution requires translating them into a coherent mathematical program. RetailOpt-190 has two roles in our study. First, it exposes the compositional feasibility–correctness gap, on which structured generation gives the largest gain. Second, its reference formulations support the injected-defect characterization of L2 in Appendix I, which maps the boundaries of behavioral verification.

**Construction.** The benchmark spans eight scenario families (F1–F8) built by *progressive composition* (Table 5). F1–F4 introduce core retail mechanisms (inventory dynamics, substitution, resource limits, demand variability) with increasing constraint interactions, and F5 tests feasibility boundaries with deliberately tight or infeasible configurations. F6 introduces discrete variables (MOQ, pack sizes) that require integrality reasoning, and F7–F8 extend the setting to multi-location networks that require multi-echelon coordination. Each of the 38 archetypes has 5 numerical variants ($\pm 15\%$, deterministic seeds), which gives 190 instances that vary the parameters while holding the structure fixed. Ground-truth optimal values are computed by Gurobi 11.0 [24] applied to hand-crafted formulations for each archetype (Appendix A).

**Evaluation.** Each instance is presented as a *data-embedded* prompt with the JSON data inline, as in prior benchmarks [7, 22, 23]. The model generates self-contained Python code, which is executed with a 60-second solver time limit. An instance is correct if its feasibility status matches the ground truth and $|y_{\text{pred}} - y_{\text{ref}}|/|y_{\text{ref}}| < \epsilon$. We report accuracy at two tiers. The *strict* tier ($\epsilon = 10^{-4}$) requires near-identical objectives, so all constraints and coefficients must be correct. The *practical* tier ($\epsilon = 10^{-2}$) also accepts near-correct solutions with minor coefficient discrepancies, that is, formulations that are structurally right but numerically imprecise. The MIP family F6 uses $\epsilon = 10^{-2}$ for both tiers because of the solver tolerance. Cross-benchmark experiments use $\epsilon = 10^{-6}$, following Chen et al. [10].

Table 6: Main results on RetailOpt-190 (pass@1, greedy decoding, $N = 3$).

| | | **Exec%** | | | **Acc%**($\epsilon = 10^{-4}$) | | | **Acc%**($\epsilon = 10^{-2}$) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Type** | **Model** | Base | CoT | ReLoop | Base | CoT | ReLoop | Base | CoT | ReLoop |
| *Foundation* | Claude Opus 4.6 | 72.1 | 93.7 | **100.0** | 22.6 | 31.1 | **31.1** | 26.8 | 34.7 | **35.3** |
| | DeepSeek-V3.2 | 91.1 | 53.2 | **97.4** | 0.5 | 3.7 | **5.8** | 3.7 | 5.8 | **11.1** |
| | Qwen3-32B | 0.0 | 0.0 | **2.1** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| *Offline SFT* | OptMATH-32B | 2.6 | 2.6 | **17.9** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | **0.5** |
| *Online RL* | SIRL-32B | 0.0 | 0.0 | **1.6** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

# 5 Experiments and Results

## 5.1 Setup

**Models.** We evaluate five models from three paradigms. The *foundation LLMs* are Claude Opus 4.6 (Anthropic's frontier model), DeepSeek-V3.2 [25] (671B MoE), and Qwen3-32B [26] (dense 32B). The *offline SFT* model is OptMATH-Qwen2.5-32B [9], fine-tuned on $\sim$17K curated optimization problems, and the *online RL* model is SIRL-Qwen2.5-32B [10], trained with solver execution status as a verifiable reward. Each model is evaluated under three configurations: *Base* (direct generation, or the model's own format), *CoT* (structured chain-of-thought), and *ReLoop* (CoT + L1–L2 verification with repair, $N = 3$). All runs use greedy decoding (temperature 0, pass@1). ReLoop adds $\sim 3\times$ the base cost in LLM tokens, and solver calls complete in seconds (Appendix E).

**Benchmarks.** RetailOpt-190 (§4), MAMO-ComplexLP (203 instances from the challenging subset of MAMO) [23], and IndustryOR (100 real-world instances) [7].

**Metrics.** On RetailOpt-190: **Exec%** (fraction producing a solver-feasible solution), **Acc%**($\epsilon = 10^{-4}$) (strict formulation correctness), and **Acc%**($\epsilon = 10^{-2}$) (practical accuracy). Cross-benchmark: **Acc%**($\epsilon = 10^{-6}$) following Chen et al. [10].

## 5.2 Main Results on RetailOpt-190

Table 6 shows that **feasibility is a poor proxy for correctness**. DeepSeek reaches 91.1% solver feasibility but only 0.5% correctness, a 90-point gap, and a large gap persists after ReLoop (Claude: 100% Exec, 31.1% Acc, so two-thirds of the runs remain silent failures).

The models fail in different ways, and ReLoop preserves or improves every foundation-model metric relative to Base. On Claude, CoT gives the largest accuracy gain (+8.5pp). On DeepSeek, CoT lowers execution (91.1%$\rightarrow$53.2%) because the four-stage decomposition produces intermediate mathematical notation that does not translate into valid Gurobi syntax. L1 fully recovers this loss through its FATAL feedback (syntax tracebacks for parser errors, IIS for infeasibility, unbounded rays for unboundedness). Claude's accuracy stays at 31.1% because most of the remaining failures are *structural* silent failures: problem decompositions that differ from the reference yet are internally consistent, which perturbation cannot detect. The 32B models reach near-zero accuracy, consistent with RetailOpt-190's compositional structure (20+ interacting constraints across multi-period, multi-product, multi-location dimensions) exceeding their reasoning capacity. ReLoop still improves their execution (OptMATH: 2.6%$\rightarrow$17.9%; SIRL: 0.0%$\rightarrow$1.6%), which suggests that strong models benefit from CoT structuring and weaker models from L1 crash recovery. L2 adds gains for the foundation models without lowering any foundation-model metric (its largest effect is on MAMO's localized defects, §5.4); the non-blocking, regression-guarded design limits the losses from incorrect repairs. For Claude, a paired exact sign test supports the CoT effect (20 corrected vs. 4 regressed, $p \approx 0.002$). Because the 190 instances are 38 archetypes $\times$ 5 numerical variants, we also run an archetype-level cluster bootstrap (38 clusters, 10,000 resamples), which gives a 95% confidence interval of $[1.6, 16.8]$pp for the CoT gain.

Table 7: Cross-benchmark generalization (Acc%, $\epsilon = 10^{-6}$, pass@1).

| | | MAMO-ComplexLP | | | IndustryOR | | |
|---|---|---|---|---|---|---|---|
| **Type** | **Model** | Base | CoT | +ReLoop | Base | CoT | +ReLoop |
| *Foundation* | Claude Opus 4.6 | 70.4 | 73.9 | **79.8** | 66.0 | 66.0 | **68.0** |
| | DeepSeek-V3.2 | 60.1 | 59.6 | **62.6** | 50.0 | 54.0 | **62.0** |
| | Qwen3-32B | 40.4 | 37.4 | **46.3** | 43.0 | 43.0 | **46.0** |
| *Offline SFT* | OptMATH-32B | **56.2** | 30.0 | 31.0 | **34.0** | 31.0 | **34.0** |
| *Online RL* | SIRL-32B | 53.2 | 46.8 | **54.2** | 40.0 | 40.0 | **43.0** |

Table 8: Ablation across benchmarks (pass@1). Each row adds one component; L2 = CPT + OPT behavioral testing.

| | RetailOpt-190 | | | | | | MAMO-ComplexLP ($\epsilon = 10^{-6}$) | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Claude Opus 4.6 | | | DeepSeek-V3.2 | | | Claude Opus 4.6 | | DeepSeek-V3.2 | |
| **Config** | **Exec** | **Acc$_{10^{-4}}$** | **Acc$_{10^{-2}}$** | **Exec** | **Acc$_{10^{-4}}$** | **Acc$_{10^{-2}}$** | **Exec** | **Acc** | **Exec** | **Acc** |
| Direct | 72.1 | 22.6 | 26.8 | 91.1 | 0.5 | 3.7 | 94.1 | 70.4 | 93.6 | 60.1 |
| +CoT | 93.7 | 31.1 | 34.7 | 53.2 | 3.7 | 5.8 | 95.6 | 73.9 | 87.7 | 59.6 |
| +CoT+L1 | 99.5 | 31.1 | 35.3 | 97.4 | 5.8 | 10.5 | 98.0 | 75.4 | 88.7 | 60.6 |
| +CoT+L1+L2 | **100.0** | **31.1** | **35.3** | **97.4** | **5.8** | **11.1** | **98.0** | **79.8** | **88.7** | **62.6** |

The gap between the strict and practical tiers ($\epsilon = 10^{-4}$ vs. $10^{-2}$) is also informative: DeepSeek's larger spread (5.8% vs. 11.1%) suggests near-correct solutions with coefficient errors, whereas Claude's smaller spread (31.1% vs. 35.3%) suggests mostly structural errors.

### 5.3 Cross-Benchmark Generalization

To test generalization beyond the retail domain of RetailOpt-190, we evaluate on two external benchmarks, neither of which was used during development (Table 7; composition audited in Appendix K). MAMO-ComplexLP contains LP/MILP problems with well-isolated constraints ($\sim$459 tokens on average), and IndustryOR contains real-world problems that require multi-step reasoning.

ReLoop improves all three foundation models and SIRL on both benchmarks without domain-specific tuning. SIRL, trained with solver execution feedback via reinforcement learning, also improves: CoT alone lowers its MAMO accuracy, but unlike OptMATH it ends above Base once verification and repair are added. The exception is OptMATH on MAMO, whose accuracy falls from 56.2% (Base) to 30.0% under CoT: the SFT model's rigid "problem$\rightarrow$code" generation pattern does not accommodate our four-stage reasoning template, so 84 instances crash and 65 previously correct solutions become incorrect.[1] All three 32B models stay well below frontier performance (best: 46.3% on MAMO, 46.0% on IndustryOR), which suggests that 32B-scale models lack the reasoning capacity for industrial optimization problems of this complexity, even with verification. The size of the gain depends on the error structure. DeepSeek gains +12.0pp on IndustryOR, +4.0pp from CoT and +8.0pp from L1, with no marginal gain from L2 (50.0/54.0/62.0/62.0 across Direct/+CoT/+CoT+L1/+ReLoop; Claude: 66.0/66.0/68.0/68.0). Claude gains +4.4pp on MAMO through L2 behavioral testing (11 corrected vs. 2 regressed, $p \approx 0.02$), and all 11 corrections restore integrality that continuous relaxation had dropped (Appendix J).

### 5.4 Ablation Studies

Table 8 isolates each component's marginal contribution across two benchmarks.

**Direct $\rightarrow$ +CoT.** On Claude, structured generation gives the largest accuracy gain (+8.5pp; 20 corrected, 4 regressed). We attribute this gain to explicit structured reasoning in general; separating

[1]The high Base score on MAMO may also reflect training–evaluation overlap, since OptMATH's $\sim$17K curated corpus could include publicly available benchmark problems. On IndustryOR, the Base–CoT gap is much smaller (34.0%$\rightarrow$31.0%).

the OR-specific staging from generic step-by-step reasoning would require a vanilla chain-of-thought control, which we leave to future work. On DeepSeek, CoT lowers execution (91.1%→53.2%) because the structured format produces invalid code, which motivates L1 as a dedicated recovery layer.

**+CoT → +CoT+L1.** L1 accounts for most of the execution recovery (+44.2pp Exec for DeepSeek, +5.8pp for Claude) and also improves accuracy. When L1 detects infeasibility, the IIS identifies a minimal set of conflicting constraints (e.g., "`capacity_limit` conflicts with `demand_fulfillment`"), so the LLM receives the specific conflict and can reformulate the affected constraints instead of retrying without information.

**+CoT+L1 → +CoT+L1+L2.** L2's contribution depends on the error structure. On MAMO, L2 is the *largest single accuracy contributor* for Claude (+4.4pp; 11 corrected, 2 regressed) and adds +2.0pp for DeepSeek (4 corrected, 0 regressed), because the simpler problem structures produce *localized errors*, that is, missing constraints or objective terms, which are the defects that perturbation testing detects. On RetailOpt-190, L2 adds execution recovery (Claude: 99.5%→100.0%) and practical accuracy (DeepSeek: 10.5%→11.1% at $\epsilon = 10^{-2}$), but strict accuracy ($\epsilon = 10^{-4}$) is unchanged because the errors are mostly structural: incorrect decompositions that still respond plausibly to perturbation. Two appendix analyses support this reading. In an expert-verified audit of 55 residual failures from both benchmarks, 42 of the 43 true modeling errors are structural (Appendix J). In an injected-defect study, the detector has 99.2% recall on 240 construction-labeled knockouts and a 26.7% false-positive rate on candidates in intact code (Appendix I). On IndustryOR, the relative objective deviation $|z_{\text{pred}} - z_{\text{ref}}|/|z_{\text{ref}}|$ over all non-crashed instances is bimodal: 34% deviate by less than 1% (small coefficient differences that perturbation cannot detect), and 47% deviate by more than 10% (structural misunderstandings that 3 repair iterations do not fix), so almost no instances fall in the range that repair can correct. Per-family results are in Appendix H.

## 6 Conclusion

ReLoop addresses silent failures in LLM-generated optimization code with *structured generation*, which reduces formulation errors during generation, and *behavioral verification*, which detects the remaining errors through solver-based perturbation without ground truth. The cross-benchmark ablation shows that the two mechanisms address different error structures: structured generation gives the largest gain on the compositional problems of RetailOpt-190 (**+8.5pp** accuracy on Claude Opus 4.6), and behavioral verification gives its largest gain on the localized defects of MAMO-ComplexLP (**+4.4pp** on Claude). Relative to Base, ReLoop preserves or improves every foundation-model metric. The 90-point feasibility–correctness gap shows that solver feasibility does not certify an LLM-generated model, so semantic verification is needed before such models are deployed.

**Limitations.** *Supported formulation class.* The current implementation targets (MI)LPs whose parameters have nonnegative physical meaning (capacities, demands, costs, resource coefficients). Signed or near-zero coefficients rely on the absolute-change fallback of Assumption 1, and nonlinear or piecewise objectives and stochastic or chance-constrained models are out of scope. Robustness to the multiplier is established only for shrink-direction perturbations (Appendix I), because RetailOpt-190 is a cost-minimization benchmark with no test population for the inflate direction. *Format compatibility.* CoT disrupts the learned output patterns of SFT models (84 crashes and 65 regressions for OptMATH on MAMO). *Shared extraction.* L2 uses the generating LLM to extract candidates, so extraction and generation errors may be correlated. The oracle-extraction study separates extraction error from detection error (Appendix I), and the symmetric regression threshold $\tau_r$ is calibrated for localized defects, so repairs of major structural omissions are rolled back and those instances keep the L1 result. *Detection boundaries.* Coefficient-magnitude errors, structurally different decompositions that look equivalent, and present-but-wrong constraints that share parameters are beyond the scope of L2 (Table 3). Solution-level change could be used as a secondary INFO signal, and an IIS-based test that separates intrinsic from formulation-induced infeasibility would extend ReLoop to problems whose correct answer may be "infeasible."

## Acknowledgments and Disclosure of Funding

This research is supported by the National Research Foundation, Singapore and Infocomm Media Development Authority under its Trust Tech Funding Initiative and the National Natural Science Foundation of China. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Infocomm Media Development Authority. The DTC grant number is DTC-IGC-07 (project: Multi-Party Evaluation: Blockchain-Enabled AI Safety and Verification Framework), and the National Natural Science Foundation of China grant numbers are U25A20450 and 62571374. The authors declare no competing interests.

## References

[1] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[2] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[3] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[4] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pages 26106–26128. PMLR, 2023.

[5] Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pages 1015–1029. PMLR, 2024.

[6] 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 *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[7] Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. ORLM: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 73(6):2986–3009, 2025.

[8] Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. LLMOPT: Learning to define and solve general optimization problems from scratch. In *Proceedings of the 13th International Conference on Learning Representations (ICLR)*, 2025.

[9] Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. OptMATH: A scalable bidirectional data synthesis framework for optimization modeling. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

[10] Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed RL: Grounding large language models for authentic optimization modeling. In *Proceedings of the 39th Conference on Neural Information Processing Systems (NeurIPS)*, 2025.

[11] Kuo Liang, Yuhang Lu, Jianming Mao, Shuyi Sun, Chunwei Yang, Congcong Zeng, Xiao Jin, Hanzhang Qin, Ruihao Zhu, and Chung-Piaw Teo. LLM for large-scale optimization model auto-formulation: A lightweight few-shot learning approach. *arXiv preprint arXiv:2601.09635*, 2026.

[12] Hao Chen, Gonzalo Esteban Constante-Flores, Krishna Sri Ipsit Mantri, Sai Madhukiran Kompalli, Akshdeep Singh Ahluwalia, and Can Li. OptiChat: Bridging optimization models and practitioners with large language models. *INFORMS Journal on Data Science*, 2025. doi:10.1287/ijds.2025.0074.

[13] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023.

[14] Yansen Zhang, Qingcan Kang, Yujie Chen, Yufei Wang, Xiongwei Han, Tao Zhong, Mingxuan Yuan, and Chen Ma. SAC-Opt: Semantic anchors for iterative correction in optimization modeling. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. arXiv:2510.05115.

[15] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, MA, 1997.

[16] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. International Series in Operations Research & Management Science. Springer, 5th edition, 2020.

[17] Tsong Yueh Chen, Shing Chi Cheung, and Siu Ming Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[18] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.

[19] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):4:1–4:27, 2018.

[20] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[21] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[22] Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track*, pages 189–203. PMLR, 2023.

[23] Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. MAMO: A mathematical modeling benchmark with solvers. *arXiv preprint arXiv:2405.13144*, 2024.

[24] Gurobi Optimization, LLC. Gurobi optimizer reference manual. https://www.gurobi.com, 2024.

[25] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, et al. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[26] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pages 611–626, 2023.

# A Reference MILP Formulation

This appendix gives the modular MILP that defines the semantic ground truth for RetailOpt-190. We present the notation and decision variables, the objective function, and all constraint families with explicit equations, followed by the *design rationale* for modeling choices that prevent common LLM errors.

## A.1 Notation

**Sets and Indices.**

- $\mathcal{P}$: set of products (SKUs); $\mathcal{L}$: set of locations (distribution centers/stores)
- $\mathcal{T} = \{1, \ldots, T\}$: discrete planning periods (weeks)
- $\mathcal{K}_p = \{1, \ldots, \mathrm{SL}_p\}$: remaining shelf-life buckets for product $p$
- $\mathcal{E}^{\text{sub}} \subseteq \mathcal{P} \times \mathcal{P}$: directed substitution arcs (upward substitution)
- $\mathcal{E}^{\text{tr}} \subseteq \mathcal{L} \times \mathcal{L}$: directed transshipment arcs
- $\mathcal{N}_p^{\text{out}} = \{p' : (p, p') \in \mathcal{E}^{\text{sub}}\}$: products that can serve $p$'s demand
- $\mathcal{N}_p^{\text{in}} = \{p' : (p', p) \in \mathcal{E}^{\text{sub}}\}$: products whose demand $p$ can serve

**Parameters.**

- $d_{p,l,t}$: demand for product $p$ at location $l$ in period $t$, computed as $d_{p,l,t} = \texttt{demand_curve}[p][t-1] \times \texttt{demand_share}[l]$
- $\mathrm{SL}_p$: shelf life of product $p$ in periods
- $\mathrm{LT}_p$: order lead time for product $p$ (periods between placement and receipt)
- $\bar{Q}_{p,t}$: production/procurement capacity for product $p$ in period $t$
- $\bar{C}_l$: cold storage capacity at location $l$ (volume units)
- $\gamma_p$: cold storage usage coefficient per unit of product $p$
- $\bar{H}_{l,t}$: labor capacity at location $l$ in period $t$ (hours)
- $h_p$: labor usage per unit of product $p$ handled
- $\rho_p$: return rate, the fraction of period-$t$ sales returned in period $t{+}1$
- $c_p^{\text{buy}}, c_p^{\text{inv}}, c_p^{\text{waste}}, c_p^{\text{ls}}$: purchasing, holding, waste, and lost-sales costs
- $c^{\text{fix}}$: fixed ordering cost per order placed; $c^{\text{tr}}$: transshipment cost per unit
- MOQ: minimum order quantity; PS: pack size
- $B$: per-period purchasing budget (if active); $\omega$: maximum waste fraction (if active)

**Key Convention.** Remaining-life index $k = \mathrm{SL}_p$ denotes the *freshest* inventory; $k = 1$ denotes inventory expiring at the end of the current period. This FIFO-compatible convention matters because LLMs frequently reverse the ordering, which produces aging dynamics in which fresh inventory expires immediately (§A.7, D2).

## A.2 Decision Variables

Table 9 lists all decision variables in the reference MILP. The formulation uses a *modular activation* pattern: the five core variables ($I$, $y$, $Q$, $W$, $L$) are always created, while optional variables ($S$, $X$, $z$, $n$) are instantiated only when the corresponding mechanism appears in the instance JSON. This follows the solver implementation, where variable creation depends on field checks such as `pack_size > 1` or `len(sub_edges) > 0`.

## A.3 Objective Function

The objective minimizes total supply chain cost over the planning horizon:

$$\min \sum_{p \in \mathcal{P}} \sum_{l \in \mathcal{L}} \sum_{t \in \mathcal{T}} \Bigg[ \underbrace{c_p^{\text{buy}}\, Q_{p,l,t}}_{\text{purchasing}} + \underbrace{c_p^{\text{inv}} \sum_{k=2}^{\mathrm{SL}_p} \left( I_{p,l,t,k} - y_{p,l,t,k} \right)}_{\text{holding}} + \underbrace{c_p^{\text{waste}}\, W_{p,l,t}}_{\text{waste}} + \underbrace{c_p^{\text{ls}}\, L_{p,l,t}}_{\text{lost sales}} \Bigg] + \Phi^{\text{opt}} \quad (2)$$

Table 9: Decision variables. Variables marked with $^\dagger$ are created only when the corresponding mechanism is active (i.e., the relevant JSON field is non-default).

| Variable | Domain | Interpretation |
|---|---|---|
| $I_{p,l,t,k}$ | $\mathbb{R}_{\geq 0}$ | Start-of-period inventory of product $p$ at location $l$ in period $t$ with $k$ periods of remaining life |
| $y_{p,l,t,k}$ | $\mathbb{R}_{\geq 0}$ | Sales from remaining-life bucket $k$ in period $t$ |
| $Q_{p,l,t}$ | $\mathbb{R}_{\geq 0}$ | Order quantity placed for product $p$ at location $l$ in period $t$ |
| $W_{p,l,t}$ | $\mathbb{R}_{\geq 0}$ | Waste (expired units) at end of period $t$ |
| $L_{p,l,t}$ | $\mathbb{R}_{\geq 0}$ | Lost sales (unmet demand) in period $t$ |
| $S^\dagger_{p\to p',l,t}$ | $\mathbb{R}_{\geq 0}$ | Substitution flow: units of $p$'s demand served by $p'$'s inventory (created iff $\mathcal{E}^{\text{sub}} \neq \emptyset$) |
| $X^\dagger_{p,l\to l',t}$ | $\mathbb{R}_{\geq 0}$ | Transshipment flow from $l$ to $l'$ (created iff $\mathcal{E}^{\text{tr}} \neq \emptyset$) |
| $z^\dagger_{p,l,t}$ | $\{0,1\}$ | Binary order trigger (created iff MOQ $> 0$ or $c^{\text{fix}} > 0$) |
| $n^\dagger_{p,l,t}$ | $\mathbb{Z}_{\geq 0}$ | Integer pack count (created iff PS $> 1$) |

where $\Phi^{\text{opt}}$ collects optional cost terms activated by specific mechanism flags:

$$\Phi^{\text{opt}} = \sum_{p,l,t} c^{\text{fix}}\, z_{p,l,t} \cdot \mathbb{1}[c^{\text{fix}} > 0] \;+\; \sum_{p} \sum_{(l,l')\in\mathcal{E}^{\text{tr}}} \sum_{t} c^{\text{tr}}\, X_{p,l\to l',t} \tag{3}$$

**Design Note (Holding Cost Scope).** Holding cost applies only to buckets $k \geq 2$ because bucket $k{=}1$ contains inventory that either sells or expires within the current period, so it *cannot* be carried into the next period. LLMs frequently apply holding cost to all buckets $k \geq 1$, which double-counts the waste penalty on expiring units (D5 in §A.7). The end-of-period inventory in each bucket is $I_{p,l,t,k} - y_{p,l,t,k}$, representing units that survived sales and will age into the next period.

### A.4 Core Constraints

The reference formulation enforces six core constraint families that govern inventory flow across products, locations, and time periods (Table 10). Together they form a *closed conservation system*: every unit entering the system (via ordering) must exit through exactly one of four channels: sales, aging into the next period, expiration, or lost sales. This accounting is what makes silent failures detectable: if an LLM omits or reverses any single constraint, the conservation structure breaks, and the resulting solutions are solver-feasible but semantically inconsistent.

### A.5 Capacity and Resource Constraints

In addition to the six core inventory-flow constraints, the formulation enforces a set of capacity and resource limits. These are modular: each constraint is active only when the corresponding JSON parameter takes a non-trivial value.

**Production Capacity.** Each product's total inflow across all locations is bounded by a time-varying capacity:

$$\sum_{l\in\mathcal{L}} A_{p,l,t} \;\leq\; \bar{Q}_{p,t}, \qquad \forall\, p \in \mathcal{P},\; t \in \mathcal{T} \tag{4}$$

where $A_{p,l,t}$ denotes the arrival quantity (accounting for lead time). This constraint is applied to *delivered* inflow, ensuring lead-time consistency: an order placed in period $t$ arrives in period $t + \text{LT}_p$ and counts against the capacity of the *arrival* period.

**Storage Capacity.** Total volume-weighted on-hand inventory at each location must not exceed cold storage capacity:

$$\sum_{p\in\mathcal{P}} \gamma_p \sum_{k=1}^{\text{SL}_p} I_{p,l,t,k} \;\leq\; \bar{C}_l, \qquad \forall\, l \in \mathcal{L},\; t \in \mathcal{T} \tag{5}$$

Table 10: Core constraint families with explicit equations and design rationale. Each rationale column explains the specific modeling choice; see §A.7 for detailed justification.

| Family | Constraint | Design Rationale |
|---|---|---|
| C1. Initialization | $I_{p,l,1,k} = 0 \quad \forall\, k < \mathrm{SL}_p$<br>$I_{p,l,1,\mathrm{SL}_p} = A_{p,l,1}$ | Without zero initialization for non-fresh buckets, unbounded phantom inventory yields objective $\approx 0$ (D1). |
| C2. Fresh inflow | $I_{p,l,t,\mathrm{SL}_p} = A_{p,l,t} + \Delta^{\mathrm{tr}}_{p,l,t} + R_{p,l,t}$<br>where $A_{p,l,t} = Q_{p,l,t-\mathrm{LT}_p}$ if $t-\mathrm{LT}_p \geq 1$, else 0;<br>$\Delta^{\mathrm{tr}}_{p,l,t} = \sum_{l':(l',l)\in\mathcal{E}^{\mathrm{tr}}} X_{p,l'\to l,t} - \sum_{l':(l,l')\in\mathcal{E}^{\mathrm{tr}}} X_{p,l\to l',t}$;<br>$R_{p,l,t} = \rho_p \sum_k y_{p,l,t-1,k}$ if $t > 1$, else 0. | Fresh inventory enters *only* via ordering, transshipment, and returns. LLMs that subtract sales in this equation effectively double-count demand fulfillment. |
| C3. Aging dynamics | $I_{p,l,t+1,k} = I_{p,l,t,k+1} - y_{p,l,t,k+1}$<br>$\forall\, t \in \{1,\ldots,T-1\},\ k \in \{1,\ldots,\mathrm{SL}_p-1\}$ | Remaining life *decrements* each period: bucket $k{+}1$ today becomes bucket $k$ tomorrow, minus sales. Incrementing $k$ instead causes fresh stock to expire immediately (D2). |
| C4. Expiration | $W_{p,l,t} = I_{p,l,t,1} - y_{p,l,t,1}$ | Only bucket $k{=}1$ can expire. Unsold units from the oldest bucket become waste; waste is *not* carried forward. |
| C5. Sales availability | $y_{p,l,t,k} \leq I_{p,l,t,k} \quad \forall\, k$ | Cannot sell more than on-hand inventory in each remaining-life bucket. Without this, the solver can create phantom sales. |
| C6. Demand conservation | $\sum_{k=1}^{\mathrm{SL}_p} y_{p,l,t,k} + L_{p,l,t} = d_{p,l,t} + S^{\mathrm{in}}_{p,l,t} - S^{\mathrm{out}}_{p,l,t}$<br>with $S^{\mathrm{out}}_{p,l,t} = \sum_{p'\in\mathcal{N}^{\mathrm{out}}_p} S_{p\to p',l,t}$,<br>$S^{\mathrm{in}}_{p,l,t} = \sum_{p'\in\mathcal{N}^{\mathrm{in}}_p} S_{p'\to p,l,t}$,<br>$S^{\mathrm{out}}_{p,l,t} \leq d_{p,l,t}$ (demand-route bound),<br>and $S^{\mathrm{in}}_{p,l,t} \leq \sum_{k=1}^{\mathrm{SL}_p} y_{p,l,t,k}$ (inventory-backed substitution). | Reversing edge direction is a common modeling mistake (D4): $S_{p\to p'}$ means $p$'s demand *exported to* $p'$'s inventory, not the reverse. The demand-route and inventory-backed bounds together prevent unbounded substitution flow. |

Storage usage is measured on *start-of-period* inventory (before sales), because this is the physical space occupied when replenishment decisions are made. Product-specific usage coefficients $\gamma_p$ model volumetric heterogeneity (e.g., bulky premium items in F3 archetypes).

**Labor Capacity.** Total labor consumed for sales handling at each location is bounded:

$$\sum_{p\in\mathcal{P}} h_p \sum_{k=1}^{\mathrm{SL}_p} y_{p,l,t,k} \;\leq\; \bar{H}_{l,t}, \qquad \forall\, l \in \mathcal{L},\ t \in \mathcal{T} \tag{6}$$

Labor is modeled as proportional to units sold (picked and packed). In many archetypes, $\bar{H}_{l,t}$ is set to a large value so the constraint does not bind; it becomes active in F8 archetypes where omni-channel operations increase per-unit labor intensity.

**Per-Period Budget.** When $B \neq \texttt{null}$, total purchasing spend in each period is capped:

$$\sum_{p\in\mathcal{P}} \sum_{l\in\mathcal{L}} c^{\mathrm{buy}}_p\, Q_{p,l,t} + c^{\mathrm{fix}} \sum_{p,l} z_{p,l,t} \;\leq\; B, \qquad \forall\, t \in \mathcal{T} \tag{7}$$

**Global Waste Cap.** When $\omega \neq \texttt{null}$, total waste over the horizon is limited as a fraction of total demand:

$$\sum_{p,l,t} W_{p,l,t} \;\leq\; \omega \cdot \sum_{p,l,t} d_{p,l,t} \tag{8}$$

### A.6 Discrete Procurement Constraints

Family F6 activates integer procurement logic. These constraints introduce mixed-integer structure via binary and integer variables.

**Minimum Order Quantity (MOQ).** When MOQ $> 0$, each order must be either zero or at least MOQ units:

$$Q_{p,l,t} \leq M \cdot z_{p,l,t} \tag{9}$$

$$Q_{p,l,t} \geq \text{MOQ} \cdot z_{p,l,t} \tag{10}$$

where $M = 10^6$ is a big-M constant and $z_{p,l,t} \in \{0, 1\}$ is the binary order trigger. The big-M formulation is standard; conditional constraints are not directly expressible in MILP and require this linearization.

**Pack Size.** When PS $> 1$, order quantities must be integer multiples of the pack size:

$$Q_{p,l,t} = \text{PS} \cdot n_{p,l,t}, \qquad n_{p,l,t} \in \mathbb{Z}_{\geq 0} \tag{11}$$

**Fixed Ordering Cost.** When $c^{\text{fix}} > 0$, the binary trigger $z_{p,l,t}$ is activated (via Eq. 9) and the fixed cost $c^{\text{fix}} \cdot z_{p,l,t}$ enters the objective.

### A.7 Formulation Design Rationale

Several formulation choices in RetailOpt-190 need an explicit justification, because in each case a naïve implementation produces a solver-feasible but semantically incorrect solution.

**D1. Explicit zero initialization ($I_{p,l,1,k} = 0$ for $k < \text{SL}_p$).** The reference formulation initializes all non-fresh inventory buckets to zero in period 1. Without this, the model admits unbounded phantom inventory in non-fresh buckets, yielding objective $\approx 0$. This is a closed-system accounting requirement: every unit in the system must originate from a production decision.

**D2. Aging direction convention ($k$ decrements toward expiration).** Inventory ages by decrementing $k$: freshly produced units enter at $k = \text{SL}_p$ and expire when they reach $k = 1$. The reverse convention (incrementing $k$) would place new arrivals at $k = 1$ and "age" them toward $k = \text{SL}_p + 1$, which does not exist, so no unit would ever expire. The decrementing convention gives every unit a finite shelf life, after which it has either been sold or become waste.

**D3. Guarded temporal boundaries.** Aging constraints are defined only for $t \in \{1, \ldots, T-1\}$, and lead-time arrivals $A_{p,l,t}$ default to zero when $t - \text{LT}_p < 1$. These guards prevent out-of-range indexing (e.g., constraints referencing $t = T+1$ or $Q_{p,l,0}$), which would cause infeasibility or runtime errors rather than silent modeling mistakes.

**D4. Directional substitution edges.** Each edge $(p_{\text{from}}, p_{\text{to}}) \in \mathcal{E}^{\text{sub}}$ is *asymmetric*: $p_{\text{from}}$'s demand can be served by $p_{\text{to}}$'s inventory, but not vice versa. This models upward substitution (e.g., premium stock serving basic demand). The prompt text and JSON schema both encode the direction explicitly via ordered pairs `[p_from, p_to]` to minimize ambiguity. Pilot experiments showed that reversing the substitution direction is one of the most common modeling mistakes ($\sim$35% error rate across four frontier LLMs).

**D5. Holding cost restricted to $k \geq 2$.** Units in the oldest bucket ($k = 1$) that are unsold incur waste cost $c_p^{\text{waste}}$. Applying holding cost $c_p^{\text{inv}}$ to this same bucket would double-count the penalty. The formulation therefore charges holding cost only on end-of-period inventory with $k \geq 2$, which keeps the waste and holding cost terms separate.

## B Scenario Family Design

RetailOpt-190 is constructed around four design principles. **(1) Structural coverage:** each instance activates at least one core retail mechanism (perishability, capacity coupling, substitution, discrete procurement, or network flows), so the benchmark covers a range of formulation challenges.

**(2) Compositionality:** difficulty arises from mechanism *interactions*, not linguistic complexity; a perishability-only instance is straightforward, but perishability coupled with capacity limits and substitution reveals whether models correctly propagate constraints. **(3) Diagnosability:** each scenario preserves recognizable module signatures, enabling precise failure localization through objective-value comparison and error analysis. **(4) Feasibility stress:** several instances are designed so that common modeling errors (e.g., missing inventory balance) lead to infeasibility, providing diagnostic signals beyond objective comparison.

### B.1 Family–Mechanism Design Matrix

The benchmark comprises 38 archetypes across 8 families (Table 11).

Table 11: Compositional design matrix. Each family tests specific mechanism interactions. ✓ = structurally active (affects optimal solution); ○ = present but not binding in default data.

| ID | Family | #Arc. | Perish. | Capacity | Subst. | Discrete | Network | Primary Test |
|---|---|---|---|---|---|---|---|---|
| F1 | Core Operations | 4 | ✓ | ○ | ○ | – | – | Aging dynamics |
| F2 | Assortment | 6 | ✓ | ✓ | ✓ | ○ | – | Substitution × capacity |
| F3 | Resources | 4 | ✓ | ✓ | ○ | – | – | Multi-resource coupling |
| F4 | Dynamics | 6 | ✓ | ✓ | ○ | – | – | Temporal propagation |
| F5 | Feasibility Stress | 4 | ✓ | ✓ | – | – | – | Stress robustness |
| F6 | Discrete Logistics | 4 | ✓ | ○ | ○ | ✓ | – | Integer constraints |
| F7 | Network & Multi-Echelon | 6 | ✓ | ✓ | ○ | ○ | ✓ | Multi-location flow |
| F8 | Omni-channel | 4 | ✓ | ✓ | ○ | – | ○ | Returns × labor |

**Difficulty Progression.** Families F1–F4 test mechanisms with limited interaction; a model may pass by correctly implementing each module independently. Families F5–F8 require *compositional reasoning*: constraints must jointly bind, and errors in one module propagate to others. For example, F7 (Network) requires correct capacity constraints *and* transshipment flows *and* budget limits simultaneously. F5 (Feasibility Stress) deliberately combines stressors so that a single modeling omission (e.g., missing lost-sales slack) renders the problem infeasible, while the ground-truth formulation, which includes all necessary slack variables, remains feasible at a very high cost. As a result, an infeasible status returned by an LLM's code on an F5 instance indicates a structural modeling error rather than a data issue.

### B.2 Archetype Specifications

Each archetype specifies: (1) active modules, (2) parameter modifications relative to the base scenario, and (3) expected binding patterns. Table 12 provides the complete list of all 38 archetypes with their structural modifications.

**Naming Convention.** Each archetype is identified by `retail_{family}_{descriptor}` in the codebase (e.g., `retail_f2_no_substitution`). Table 12 omits the `retail_` prefix for brevity; the suffix directly corresponds to the archetype key in `archetypes.yaml` and the generator function name in `retail_benchmark_generator.py`. Instance files are named `retail_{family}_{descriptor}_v{0-4}.json`.

**Reading the Table.** The "Modification" column describes only the parameters that change relative to the base scenario (§B.4); all unmentioned fields retain their base values. Numeric values (e.g., "$\times 0.3$") are multiplicative factors applied in the generator code. Where specific cost values or per-SKU arrays are listed, they correspond to the order `[SKU_Basic, SKU_Premium, SKU_ShortLife]`.

### B.3 Instance Generation via Controlled Perturbation

Each archetype is expanded into 5 numerical variants (v0–v4), yielding $38 \times 5 = 190$ instances total. Variant v0 uses unperturbed parameters; variants v1–v4 apply controlled stochastic perturbation with intensity $\alpha = 0.15$.

**Perturbation Procedure.** For each variant $v \in \{1, 2, 3, 4\}$ and archetype with base name $n$:

1. Compute a deterministic seed: `seed` = `uint32_le`$\big($`SHA256`$(n$ `"|"` $|\ v)\,[:4]\big)$, where the input string is `"{name}|{v}"` and `uint32_le` reads the first 4 bytes of the digest as an unsigned 32-bit little-endian integer.
2. Initialize a NumPy random generator: `rng` = `default_rng(seed)`
3. Perturb demand curves: $\tilde{d}_{p,t} = \lfloor d_{p,t} \cdot U(1-\alpha, 1+\alpha) \rfloor$ for each product and period
4. Perturb storage capacities: $\tilde{C}_l = C_l \cdot U(1-\alpha, 1+\alpha)$ for each location

where $U(a, b)$ denotes independent draws from Uniform$(a, b)$ and $\alpha = 0.15$.

**Rationale.** The perturbation targets the two parameter groups most likely to change which constraints bind, demand volumes and storage capacities, and keeps the structure fixed (shelf life, network topology, cost ratios). The deterministic hashing makes the instances reproducible across platforms without a fixed global seed.

### B.4 Base Scenario Specification

All 38 archetypes derive from a single base scenario via modular parameter overrides. The base scenario is generated by `get_base_scenario()` in the benchmark generator and describes a moderately complex retail setting: three product tiers with heterogeneous shelf lives, five distribution centers with non-uniform demand shares, and a seasonal demand pattern that peaks mid-horizon. Costs are calibrated so that lost sales dominate waste in the objective, and the optimal plan therefore trades off overstocking (waste) against understocking (lost-sales penalty). Table 13 summarizes the base configuration.

## C Data Formats and Access

### C.1 Prompt Format Comparison

RetailOpt-190 provides two prompt formats per scenario to support different evaluation paradigms (Table 14).

**Default Reporting Format.** We report all results using the *data-embedded* format (`.full.txt`) for fair comparison with NL4Opt [22], MAMO [23], and IndustryOR [7], which embed data directly in prompts. The schema-based format is also available for evaluating LLM data-access capabilities in production-like settings.

**Two Data Delivery Mechanisms.** The schema-based and data-embedded formats convey identical semantic information but differ in how instance data reaches the LLM-generated code:

- **Schema-based (`.scenario.txt`):** The prompt contains only a *type-level* JSON schema (e.g., `"periods": int`, `"demand_curve": {p: [float]}`) and a `[DATA ACCESS]` section instructing the LLM that "The variable `data` is pre-loaded. Do NOT use file I/O." The actual numerical JSON is injected at runtime as a Python dictionary. The LLM must write code that reads from the `data` dict using correct key names, nested access patterns, and 0-to-1 index conversion.
- **Data-embedded (`.full.txt`):** The prompt contains the *complete numerical JSON* inline within a `[DATA]` section. There is no `[DATA SCHEMA]` or `[DATA ACCESS]` section. The LLM must embed the JSON as a string literal and parse it with `json.loads()`, or hardcode the values directly.

This design isolates the effect of data access complexity: the schema-based format tests whether LLMs can correctly read nested JSON structures from type signatures alone, while the data-embedded format tests whether they can extract structure from raw numerical data. Section C.4 provides a concrete side-by-side comparison.

**Prompt Architecture.** The prompt is *moderately scaffolded*. It provides the business narrative, the data, and a set of *structural cues*: variable indexing conventions, the perishable inflow accounting equation aligned with the reference MILP (per-location order quantity $Q_{p,l,t}$, lead-time arrival,

transshipment net flow, and returns), and the aggregate production constraint. It does **not** provide a complete formulation, an objective function specification, the full constraint set, or warnings about common errors. The cues are necessary because fully unscaffolded prompts produce near-zero accuracy on compositional retail problems for all tested models. With moderate scaffolding, the benchmark tests whether an LLM can compose the remaining optimization structure (objective, capacity/demand constraints, integrality reasoning, multi-period coupling) from the natural-language description.

### C.2 JSON Schema

All 190 instances share a universal JSON schema, ensuring that a single solver implementation can process every archetype without format-specific logic. The schema is organized into six groups: structural dimensions (periods, products, locations), perishability parameters (shelf life, lead time), demand specification (curve and share), capacity limits (production, storage, labor), cost coefficients, and network topology. Table 16 provides a complete field reference with types, default values from the base scenario, and semantic descriptions. Fields under `constraints` and `network` are nested objects; safe access via `data.get()` with defaults is required to avoid `KeyError` on archetypes where these fields are absent (see Pattern 3 in §C.3).

### C.3 Critical Data Access Patterns

Three data access patterns cause frequent LLM errors:

**Pattern 1: Demand Indexing Mismatch.** The JSON arrays are 0-indexed, but the optimization model uses 1-indexed periods:

$$d_{p,l,t} = \texttt{demand_curve}[p][t-1] \times \texttt{demand_share}[l], \qquad t \in \{1, \ldots, T\} \tag{12}$$

LLMs frequently access `demand_curve[p][t]` directly, causing an off-by-one error that shifts the entire demand profile by one period.

**Pattern 2: Substitution Edge Semantics.** The JSON encodes `sub_edges` as `[p_from, p_to]`, meaning *$p_{from}$'s demand can be served by $p_{to}$'s inventory*. This is "upward substitution": a premium product serves a basic product's excess demand. In the reference solver, this translates to:

- $\mathcal{N}_p^{\text{out}}$: products whose inventory can serve $p$'s demand ($p$ exports demand to them)
- $\mathcal{N}_p^{\text{in}}$: products whose demand $p$'s inventory serves ($p$ receives their demand)

For example, edge `["SKU_Basic", "SKU_Premium"]` creates $\mathcal{N}^{\text{out}}_{\text{Basic}} = \{\text{Premium}\}$ and $\mathcal{N}^{\text{in}}_{\text{Premium}} = \{\text{Basic}\}$.

**Pattern 3: Nested Network Access.** Network data is nested within the `network` object. Safe access requires:

```
sub_edges = [tuple(e) for e in data.get('network', {}).get('sub_edges', [])]
trans_edges = [tuple(e) for e in data.get('network', {}).get('trans_edges', [])]
```

LLMs that access `data['sub_edges']` directly encounter a `KeyError`, converting a potential silent failure into an execution failure.

### C.4 Worked Example: `retail_f1_52_weeks_v0`

We illustrate the complete data pipeline using the `retail_f1_52_weeks_v0` instance (F1 family, 52-week horizon). This archetype tiles the base 20-period demand and capacity arrays to 52 periods, testing long-horizon accumulation effects while preserving the same constraint structure.

**Instance JSON (Truncated).** Listing 1 shows the JSON structure with arrays abbreviated. The full instance contains 52-element arrays for `demand_curve`, `production_cap`, and `labor_cap`; all other fields are scalars or short dicts.

```
{
  "name": "retail_f1_52_weeks_v0",
  "description": "Standard seasonal retail scenario.",
  "periods": 52,
  "products": ["SKU_Basic", "SKU_Premium", "SKU_ShortLife"],
  "locations": ["DC1", "DC2", "DC3", "DC4", "DC5"],
  "shelf_life": {"SKU_Basic": 10, "SKU_Premium": 8, "SKU_ShortLife": 4},
  "lead_time":  {"SKU_Basic": 0,  "SKU_Premium": 0, "SKU_ShortLife": 0},
  "cold_capacity": {"DC1": 4000, "DC2": 3500, "DC3": 3000,
                    "DC4": 3000, "DC5": 2500},
  "cold_usage": {"SKU_Basic": 1.0, "SKU_Premium": 3.0,
                 "SKU_ShortLife": 1.2},
  "production_cap": {
    "SKU_Basic":     [800, 800, ..., 800],       // 52 elements
    "SKU_Premium":   [400, 400, ..., 400],
    "SKU_ShortLife": [500, 500, ..., 500]
  },
  "labor_cap": {"DC1": [99999.0, ...], ...},     // 5 x 52
  "labor_usage":  {"SKU_Basic": 0.0, ...},
  "return_rate":  {"SKU_Basic": 0.0, ...},
  "demand_curve": {
    "SKU_Basic":     [303, 311, 328, ..., 1300, 1245],
    "SKU_Premium":   [151, 155, 164, ...,  650,  622],
    "SKU_ShortLife": [121, 124, 131, ...,  520,  498]
  },
  "demand_share": {"DC1": 0.25, "DC2": 0.2, "DC3": 0.2,
                   "DC4": 0.2,  "DC5": 0.15},
  "costs": {
    "lost_sales": {"SKU_Basic":50,  "SKU_Premium":80,  "SKU_ShortLife":40},
    "inventory":  {"SKU_Basic":1.0, "SKU_Premium":1.5, "SKU_ShortLife":1.0},
    "waste":      {"SKU_Basic":2.0, "SKU_Premium":3.0, "SKU_ShortLife":2.0},
    "fixed_order": 0.0, "transshipment": 0.5,
    "purchasing": {"SKU_Basic":10, "SKU_Premium":20, "SKU_ShortLife":15}
  },
  "constraints": {"moq": 0, "pack_size": 1,
                  "budget_per_period": null, "waste_limit_pct": null},
  "network": {
    "sub_edges":   [["SKU_Basic", "SKU_Premium"]],
    "trans_edges": []
  }
}
```

Listing 1: Instance JSON for `retail_f1_52_weeks_v0` (arrays truncated for space).

**Schema-Based Prompt (`.scenario.txt`).** Listing 2 shows the schema-based prompt structure. The prompt provides the business narrative with embedded structure cues (perishability equations, substitution semantics), a type-level JSON schema, data access instructions, and the required output format. It does *not* include the actual numerical data; at evaluation time, the JSON is loaded externally and made available as the `data` variable.

```
[SCENARIO]
Family: F1 (Core Operations)
Archetype: retail_f1_52_weeks
Scenario ID: retail_f1_52_weeks_v0

[BUSINESS DESCRIPTION]
Business narrative:
The retailer plans inventory over a full year of weekly decisions,
represented by fifty-two time periods. Exogenous seasonal demand,
local inventories at each distribution center, and per-product
production capacities work as in the core operations baseline.
Customers are never backordered; unmet demand in a week is
immediately lost and penalized. ...

Structure cues:
- ...
- Shelf life: Each product has a shelf life in periods.
  Inventory must be tracked by REMAINING LIFE.

  VARIABLE DEFINITION: I[p,l,t,r] = inventory at START of period t
  with r periods remaining.
  Convention: r=1 is OLDEST (sell first FIFO),
              r=shelf_life[p] is FRESHEST.

  KEY EQUATIONS - implement these conventions consistently with
  the JSON data:
```

```
  (1) Fresh inflow:
      I[p,l,t,SL] = Q[p,l,t-LT[p]] + transshipment_net[p,l,t]
                    + returns[p,l,t]
      - Q[p,l,t] is the per-LOCATION order quantity
      - LT[p] = lead_time[p]; treat Q[p,l,s] as 0 for s < 1
      - Aggregate production constraint:
          sum over locations of Q[p,l,t] <= production_cap[p][t]
      - transshipment_net = inbound - outbound flows on trans_edges
        (zero when network.trans_edges is empty)
      - returns = return_rate[p] * sum_a sales[p,l,t-1,a]
        when t > 1, else 0
      - This is ONLY the fresh-inflow channel; do NOT subtract
        sales here.
  (2) Aging: I[p,l,t+1,r] = I[p,l,t,r+1] - sales[p,l,t,r+1]
      for r=1..SL-1
  (3) Waste: W[p,l,t] = I[p,l,t,1] - sales[p,l,t,1]
  (4) Sales availability: sales[p,l,t,r] <= I[p,l,t,r]
  (5) Inventory holding cost: charged on
      (I[p,l,t,r] - sales[p,l,t,r]) for r >= 2

- Substitution: Edge [p_from, p_to] means p_from's demand can
  be served by p_to's inventory.
  Variable sub[p_from, p_to, l, t] = units of p_from's demand
  fulfilled by p_to.

  Demand fulfillment equation:
  - For p_from: total_sales[p_from] + sub[p_from, p_to]
                + L[p_from] = demand[p_from]
  - For p_to:   total_sales[p_to] - sub[p_from, p_to]
                + L[p_to]   = demand[p_to]

- No transshipment and zero lead times in this scenario.
- The objective is to minimize total cost over the entire year,
  aggregating purchasing, holding, waste, and lost sales costs across all weeks.

[DATA SCHEMA]
{
  "periods": int,       "products": [str, ...],
  "shelf_life": {p: int},  "lead_time": {p: int},
  "demand_curve": {p: [float, ...]},
  "demand_share": {l: float},
  "production_cap": {p: [float, ...]},
  "cold_capacity": {l: float},  "cold_usage": {p: float},
  "costs": { "purchasing": {p: float}, "inventory": {p: float},
             "waste": {p: float}, "lost_sales": {p: float},
             "fixed_order": float, "transshipment": float },
  "constraints": { "moq": float, "pack_size": int,
                   "budget_per_period": float|null,
                   "waste_limit_pct": float|null },
  "network": { "sub_edges": [[p_from, p_to], ...],
               "trans_edges": [[l_from, l_to], ...] }
}

[DATA ACCESS]
- The variable `data` is pre-loaded. Do NOT use file I/O.
- Lists are 0-indexed (period t in model uses index [t-1] in
  data arrays)

CRITICAL - Network edges require tuple conversion for Gurobi:
  sub_edges = [tuple(e) for e in
               data.get('network', {}).get('sub_edges', [])]
  trans_edges = [tuple(e) for e in
               data.get('network', {}).get('trans_edges', [])]

[OUTPUT FORMAT]
- Import: import gurobipy as gp; from gurobipy import GRB
- Set Gurobi params: m.Params.OutputFlag = 0;
  m.Params.Threads = 1; m.Params.Seed = 0
- Print at end:
  print(f"status: {{m.Status}}")
  if m.Status == 2:
      print(f"objective: {{m.ObjVal}}")
- Output ONLY executable Python code. No markdown, no explanations.

[TASK]
Write a GurobiPy script that models and solves this optimization
problem.
```

Listing 2: Schema-based prompt (`.scenario.txt`, abbreviated). The full prompt is 112 lines; sections marked ... indicate omitted material of similar structure.

**Data-Embedded Prompt (`.full.txt`).** The data-embedded format delivers the *same business narrative* as the schema-based prompt but changes **how data reaches the LLM-generated code**. In the schema-based format, the prompt provides only field types (e.g., `"periods": int`) and instructs the LLM to read from a pre-loaded `data` variable at runtime. In the data-embedded format, the prompt contains the *complete numerical JSON* inline, and the LLM must parse it with `json.loads()`. The key structural differences are:

1. **Schema-based** has `[DATA SCHEMA]` (types only) + `[DATA ACCESS]` (tells LLM: "`data` is pre-loaded, do NOT use file I/O").
2. **Data-embedded** replaces both with `[DATA]` containing the full JSON (694 lines for this instance). No `[DATA SCHEMA]` or `[DATA ACCESS]` sections appear.
3. The `[OUTPUT FORMAT]` in the data-embedded format also requires `import json`, and the `[TASK]` explicitly instructs "parse the JSON data above (use `json.loads` on the string)".

Listing 3 shows only the sections that structurally differ from the schema-based prompt (Listing 2). This is the default reporting format, as it enables self-contained evaluation without external file access, which matches the setup of NL4Opt [22], MAMO [23], and IndustryOR [7].

```
[SCENARIO]
Family: F1 (Core Operations)
Archetype: retail_f1_52_weeks
Scenario ID: retail_f1_52_weeks_v0

[BUSINESS DESCRIPTION]
...  (identical to .scenario.txt -- same narrative and equations)

                    *** NO [DATA SCHEMA] section ***
                    *** NO [DATA ACCESS] section ***

[DATA]
The following JSON contains all instance data. Parse it directly
in your code.

```json
{
  "name": "retail_f1_52_weeks_v0",
  "description": "Standard seasonal retail scenario.",
  "periods": 52,
  "products": ["SKU_Basic", "SKU_Premium", "SKU_ShortLife"],
  "locations": ["DC1", "DC2", "DC3", "DC4", "DC5"],
  "shelf_life": {"SKU_Basic": 10, "SKU_Premium": 8, ...},
  "demand_curve": {
    "SKU_Basic": [303, 311, 328, 365, 435, 549, 711, 906,
                  1100, 1245, 1300, 1245, 1100, 906, 711,
                  549, 435, 365, 328, 311,          // period 1-20
                  303, 311, 328, ..., 1300, 1245],   // tiled to 52
    "SKU_Premium": [151, 155, 164, ..., 650, 622],
    "SKU_ShortLife": [121, 124, 131, ..., 520, 498]
  },
  "production_cap": {"SKU_Basic": [800, 800, ..., 800], ...},
  "costs": {
    "lost_sales": {"SKU_Basic": 50.0, ...},
    "inventory": {"SKU_Basic": 1.0, ...},
    ...
  },
  "constraints": {"moq": 0, "pack_size": 1,
                  "budget_per_period": null,
                  "waste_limit_pct": null},
  "network": {"sub_edges": [["SKU_Basic","SKU_Premium"]],
              "trans_edges": []}
}
```

[OUTPUT FORMAT]
- Import: import gurobipy as gp; from gurobipy import GRB;
  import json
- Set Gurobi params: m.Params.OutputFlag = 0;
  m.Params.Threads = 1; m.Params.Seed = 0
- Print at end:
  print(f"status: {{m.Status}}")
  if m.Status == 2:
      print(f"objective: {{m.ObjVal}}")
- Output ONLY executable Python code. No markdown, no explanations.

[TASK]
Write a GurobiPy script that:
```

```
1. Parses the JSON data above (use json.loads on the string)
2. Models and solves the optimization problem
3. Prints status and objective value
```

Listing 3: Data-embedded prompt (`.full.txt`): sections that differ from the schema-based format. The `[BUSINESS DESCRIPTION]` is identical and omitted here.

Table 17 summarizes the structural contrast between the two formats.

# D Solver Configuration and Evaluation Protocol

## D.1 Ground Truth Solver Settings

All 190 ground truth solutions are computed using the Universal Retail Solver (URS) implemented in GurobiPy. Table 18 lists the Gurobi parameters used for ground truth computation. Both the URS and LLM-generated code use `Threads=1` and `Seed=0` to ensure deterministic, single-threaded reproducibility.

## D.2 Accuracy Tolerances

An instance is judged **correct** if both conditions hold:

1. **Status match**: predicted and ground-truth statuses agree (both feasible, or both infeasible).
2. **Objective match**: for feasible instances, $|y_{\text{pred}} - y_{\text{ref}}|/|y_{\text{ref}}| < \epsilon$.

The tolerance $\epsilon$ is family-dependent because problem structure determines the achievable precision within the 60-second time limit (Table 19). Families F1–F5 and F7–F8 produce LP relaxations that are either tight or involve few integer variables, so the solver reaches (near-)optimal solutions quickly. Family F6, in contrast, introduces MOQ binary triggers and pack-size integer variables that create harder branch-and-bound trees; the 60-second limit may yield solutions with residual gaps.

## D.3 Evaluation Metrics

We report three primary metrics; precise mathematical definitions appear in Appendix F.2:

- **Execution Rate (Exec%)** = fraction of instances whose generated code runs without runtime error *and* where the solver returns a feasible solution. Infeasibility, unboundedness, and runtime errors all count as execution failure.
- **Accuracy (Acc%)** = fraction of instances satisfying both feasibility-status match and $|y_{\text{pred}} - y_{\text{ref}}| \,/\, |y_{\text{ref}}| < \epsilon$.
- **Silent Failure Rate (SF%)** = Exec% $-$ Acc%, the absolute gap between solver-feasible execution and correctness.

The silent failure rate is the primary metric of interest: it captures the absolute fraction of *seemingly successful* runs that produce incorrect objective values due to semantic modeling errors.

# E ReLoop Implementation Details

This section provides complete implementation details for the two-layer verification framework described in Section 3.3. All prompts and parameters correspond to the source code in `reloop/`.

**Algorithm 1** ReLoop: Behavioral Verification for LLM-Generated Optimization Code

**Require:** Problem description $x$, LLM $G$, iteration budget $N$
**Ensure:** Verified code $C$, objective $z^*$, solution $\mathbf{x}^*$, status, diagnostics $\mathcal{D}$

**Phase 1: Structured Generation + L1 Verification**

$C \leftarrow \textsc{StructuredGeneration}(G, x)$ ▷ Eq. 1
**for** $i = 1, \ldots, N$ **do** ▷ L1 regeneration loop
  $(\mathit{status}, z^*, \mathbf{x}^*, \mathit{diag}) \leftarrow \text{L1-}\textsc{Verify}(C)$
  **if** $\mathit{status} \neq \textsc{Fatal}$ **then break**
  **end if**
  $C \leftarrow \textsc{Regenerate}(G, x, \mathit{diag})$ ▷ IIS/ray-guided
**end for**
**if** $\mathit{status} = \textsc{Fatal}$ **then return** $(C, \perp, \perp, \textsc{Failed}, \emptyset)$
**end if**

**Phase 2: L2 Behavioral Testing + Diagnosis-Guided Repair**

**for** $j = 1, \ldots, N$ **do** ▷ Diagnostic repair loop
  $\mathcal{D} \leftarrow \text{L2-CPT}(C, z^*, x) \cup \text{L2-OPT}(C, z^*, x)$
  **if** no $d \in \mathcal{D}$ has severity $\textsc{Warning}$ **then return** $(C, z^*, \mathbf{x}^*, \textsc{Verified}, \mathcal{D})$ ▷ Skip guard
  **end if**
  $C' \leftarrow \textsc{TargetedRepair}(G, C, \mathcal{D})$
  **if** $\neg\, \textsc{SafetyCheck}(C')$ **then** ▷ Block data mutation
    $C' \leftarrow \textsc{GuidedRetry}(G, C, \mathcal{D})$ ▷ Free retry, no budget consumed
    **if** $\neg\, \textsc{SafetyCheck}(C')$ **then break** ▷ Halt repair on persistent unsafe output
    **end if**
  **end if**
  $(\mathit{status}', z'^*, \mathbf{x}'^*, _) \leftarrow \text{L1-}\textsc{Verify}(C')$
  **if** $\textsc{Regression}(z'^*, \mathit{status}', z^*, \mathit{status})$ **then** ▷ $|z'^* - z^*|/|z^*| > \tau_r$
    **break** ▷ Rollback: keep $C, z^*, \mathbf{x}^*$
  **end if**
  $C,\ z^*,\ \mathbf{x}^*,\ \mathit{status} \leftarrow C',\ z'^*,\ \mathbf{x}'^*,\ \mathit{status}'$
**end for**
**return** $(C, z^*, \mathbf{x}^*, \mathit{status}, \mathcal{D})$

Table 12: Complete archetype specifications. "Modification" describes the parameter change relative to the base scenario (`get_base_scenario()` in the generator). All 38 archetypes share the same JSON schema and are solved by the same universal MILP.

| Family | Archetype ID | Modification from Base Scenario |
|---|---|---|
| F1 | `f1_base` | Baseline: 20 periods, 3 SKUs, 5 DCs, seasonal Gaussian demand |
| | `f1_high_waste` | Waste cost $\times 20$ for all products |
| | `f1_jit_logic` | Holding cost $\times 20$ for all products |
| | `f1_52_weeks` | Horizon extended to $T{=}52$ (demand/capacity arrays tiled from base) |
| F2 | `f2_no_substitution` | Substitution edges removed ($\mathcal{E}^{\text{sub}} = \emptyset$) |
| | `f2_circular_sub` | Circular substitution ring: Basic→Premium→ShortLife→Basic |
| | `f2_cannibalization` | Basic demand $\times 2$, Basic lost-sales penalty reduced to \$5, storage $\times 0.5$ |
| | `f2_ultra_fresh` | Shelf life reduced to $\{2, 2, 1\}$ periods |
| | `f2_price_band_tight` | Premium purchasing cost $\times 0.8$, Basic $\times 1.1$, Premium lost-sales $\times 2$ |
| | `f2_promo_budget` | Last 4 periods: Basic/ShortLife demand $\times 2$; budget = \$15,000/period |
| F3 | `f3_storage_bottleneck` | Cold capacity $\times 0.3$ at all locations |
| | `f3_volumetric_constraint` | Premium cold usage increased to 15.0 (vs. 3.0 baseline) |
| | `f3_supply_bottleneck` | Production capacity $\times 0.3$; storage set to 999,999 |
| | `f3_unbalanced_network` | DC1 gets 96% of total storage; others get 1% each |
| F4 | `f4_early_stockout` | Production = 0 in periods 1–5 |
| | `f4_peak_failure` | Production = 0 in periods 9–12 (peak demand window) |
| | `f4_demand_surge` | Period-15 demand $\times 4$ (single-period spike) |
| | `f4_quality_hold` | Basic production = 0 from period 11 onward |
| | `f4_robust_variance` | Alternating demand $\times 1.5/\times 0.7$; lost-sales $\times 2.5$ |
| | `f4_supply_risk` | Mid-horizon production $\times 0.4$ for 4 periods; waste cost $\times 3$ |
| F5 | `f5_impossible_demand` | All demand $\times 5$ (far exceeds production capacity) |
| | `f5_strict_service_trap` | Storage $\times 0.1$ (near-zero capacity) |
| | `f5_storage_overflow` | Cold capacity = 0.5 at all locations |
| | `f5_ultimate_stress` | Composite: storage $\times 0.3$ + peak failure (periods 9–12) + no substitution |
| F6 | `f6_lead_time` | Lead times = $\{3, 4, 2\}$ periods per SKU |
| | `f6_moq_binary` | MOQ = 300 units (global) |
| | `f6_fixed_order_cost` | Fixed ordering cost = \$5,000 per order |
| | `f6_pack_size_integer` | Pack size = 100 units |
| F7 | `f7_transshipment` | Full bidirectional transshipment among all 5 DCs |
| | `f7_hub_and_spoke` | Hub (DC1: 50,000 cap) → 4 spokes (500 cap each) |
| | `f7_budget_limit` | Per-period budget = \$10,000 |
| | `f7_multi_sourcing` | Heterogeneous lead times $\{5, 0, 1\}$; holding costs Basic=0.5, Premium=10.0 |
| | `f7_multiechelon_chain` | 3-tier: Plant → 2 DCs → 3 Stores (6 locations; Plant/DC demand = 0) |
| | `f7_ring_routing` | Circular transshipment ring among 5 DCs; storage $\times 0.8$ |
| F8 | `f8_reverse_logistics` | Return rates $\{0.20, 0.10, 0.05\}$ per SKU |
| | `f8_labor_constraint` | Labor cap = 200/period; usage = $\{0.1, 0.2, 0.1\}$ per SKU |
| | `f8_ship_from_store` | Storage $\times 5$; labor cap = 500; high per-unit labor usage $\{0.5, 0.8, 0.6\}$ |
| | `f8_sustainability` | Global waste cap $\leq 2\%$ of total demand |

Table 13: Base scenario parameters shared across all archetypes (before archetype-specific modifications).

| Parameter | Value | Description |
|---|---|---|
| $T$ | 20 | Planning periods |
| $\lvert\mathcal{P}\rvert$ | 3 | SKU_Basic, SKU_Premium, SKU_ShortLife |
| $\lvert\mathcal{L}\rvert$ | 5 | DC1 through DC5 |
| Shelf life | $\{10, 8, 4\}$ | Periods remaining at production |
| Lead time | $\{0, 0, 0\}$ | Same-period arrival (default) |
| Demand curve | Gaussian: $\lfloor 1000 \exp\big(-\frac{(t-10)^2}{18}\big) + 300 \rfloor$ | Seasonal peak at mid-horizon ($t$ is 0-indexed); SKU_Premium $\times 0.5$, SKU_ShortLife $\times 0.4$; values cast to `int` in generator |
| Demand share | $\{0.25, 0.20, 0.20, 0.20, 0.15\}$ | Allocated to DC1–DC5 |
| Production cap | $\{800, 400, 500\}$/period | Per-SKU, constant across periods |
| Storage cap | $\{4000, 3500, 3000, 3000, 2500\}$ | Per-DC cold capacity |
| Cold usage | $\{1.0, 3.0, 1.2\}$ | Volume units per product unit |
| $c^{\text{buy}}$ | $\{10, 20, 15\}$ | Purchasing cost per unit |
| $c^{\text{inv}}$ | $\{1.0, 1.5, 1.0\}$ | Holding cost per unit per period |
| $c^{\text{waste}}$ | $\{2.0, 3.0, 2.0\}$ | Waste cost per unit expired |
| $c^{\text{ls}}$ | $\{50, 80, 40\}$ | Lost-sales penalty per unit |
| $c^{\text{fix}}$ | 0 | No fixed ordering cost |
| $c^{\text{tr}}$ | 0.5 | Transshipment cost per unit |
| Substitution | Basic $\rightarrow$ Premium | Upward: Premium can serve Basic's demand |
| Transshipment | $\emptyset$ | No inter-location movement (default) |
| MOQ, pack size | $0, 1$ | Continuous ordering (default) |

Table 14: Two prompt formats for different evaluation scenarios. The *same semantic information* is conveyed in both formats; only the data delivery method differs.

| Format | Data Location | Content | Use Case | File Suffix |
|---|---|---|---|---|
| Schema-based | External (runtime) | Narrative + JSON schema (types only) | Scalability, production | `.scenario.txt` |
| Data-embedded | In prompt | Narrative + full JSON data | Benchmark comparison | `.full.txt` |

Table 15: Information provided in baseline prompts vs. left for the LLM to derive.

| Provided in Prompt | Left for LLM to Derive |
|---|---|
| Business narrative with structure cues | Complete formulation (variables, objective, constraints) |
| Data schema (field names, types) | Capacity, demand, and budget constraints |
| Data access patterns (indexing, nesting) | Substitution and inventory dynamics constraints |
| Output format specification (GurobiPy) | Boundary conditions / edge cases |
| Inventory inflow accounting (URS-aligned) | Common error warnings |

Table 16: Complete JSON schema field reference. Types: `int` = integer, `float` = real, `str` = string, `[T]` = array of type T, `{K:V}` = dict, `null` = absent/inactive.

| Field | Type | Example | Semantics |
|---|---|---|---|
| `periods` | `int` | 20 | Number of planning periods $T$ |
| `products` | `[str]` | `["SKU_Basic",...]` | Product identifiers |
| `locations` | `[str]` | `["DC1",...]` | Location identifiers |
| `shelf_life` | `{str:int}` | `{"SKU_Basic":10}` | Remaining-life capacity per SKU |
| `lead_time` | `{str:int}` | `{"SKU_Basic":0}` | Order-to-arrival delay (periods) |
| `demand_curve` | `{str:[float]}` | `[303,311,...]` | Aggregate demand per SKU per period (0-indexed) |
| `demand_share` | `{str:float}` | `{"DC1":0.25}` | Location's fraction of aggregate demand |
| `production_cap` | `{str:[float]}` | `[800,...]` | Max production per SKU per period (0-indexed) |
| `cold_capacity` | `{str:float}` | `{"DC1":4000}` | Storage capacity per location |
| `cold_usage` | `{str:float}` | `{"SKU_Basic":1.0}` | Volume units per product unit |
| `labor_cap` | `{str:[float]}` | `[99999,...]` | Labor hours per location per period |
| `labor_usage` | `{str:float}` | `{"SKU_Basic":0.0}` | Labor hours per unit sold |
| `return_rate` | `{str:float}` | `{"SKU_Basic":0.0}` | Fraction of sales returned next period |
| `costs.purchasing` | `{str:float}` | `{"SKU_Basic":10}` | Per-unit buying cost |
| `costs.inventory` | `{str:float}` | `{"SKU_Basic":1.0}` | Per-unit holding cost per period |
| `costs.waste` | `{str:float}` | `{"SKU_Basic":2.0}` | Per-unit expiration penalty |
| `costs.lost_sales` | `{str:float}` | `{"SKU_Basic":50}` | Per-unit lost-demand penalty |
| `costs.fixed_order` | `float` | 0.0 | Fixed cost per order placed |
| `costs.transshipment` | `float` | 0.5 | Per-unit transshipment cost |
| `constraints.moq` | `float` | 0 | Minimum order quantity (0 = inactive) |
| `constraints.pack_size` | `int` | 1 | Pack size (1 = continuous) |
| `constraints.budget_per_period` | `float\|null` | `null` | Per-period purchasing budget |
| `constraints.waste_limit_pct` | `float\|null` | `null` | Max waste as fraction of total demand |
| `network.sub_edges` | `[[str,str]]` | `[["Basic","Premium"]]` | Directed substitution: from's demand served by to's inventory |
| `network.trans_edges` | `[[str,str]]` | `[]` | Directed transshipment arcs between locations |

Table 17: Section-level comparison of the two prompt formats for a single instance. ✓ = present, – = absent.

| Prompt Section | Schema-based | Data-embedded |
|---|---|---|
| `[SCENARIO]` header | ✓ | ✓ |
| `[BUSINESS DESCRIPTION]` | ✓ | ✓ |
| `[DATA SCHEMA]` (types) | ✓ | – |
| `[DATA ACCESS]` (runtime) | ✓ | – |
| `[DATA]` (full JSON inline) | – | ✓ |
| `[OUTPUT FORMAT]` | ✓ | ✓ |
| `[TASK]` | ✓ | ✓ |
| Data delivery method | `data` var pre-loaded | `json.loads()` in code |
| Lines (this instance) | 112 | 765 |

Table 18: Gurobi solver parameters for ground truth computation, as set in `universal_retail_solver.py`.

| Parameter | Value | Purpose |
|---|---|---|
| `TimeLimit` | 60 seconds | Prevent stalling on complex MIPs (F6/F7) |
| `MIPGap` | 1% | Tolerance for near-optimal solutions |
| `Threads` | 1 | Single-threaded for reproducibility |
| `Seed` | 0 | Fixed random seed |
| `OutputFlag` | 0 | Suppress solver output for batch processing |

Table 19: Accuracy tolerances by problem structure.

| **Families** | **Problem Type** | **Tolerance $\epsilon$** | **Rationale** |
|---|---|---|---|
| F1–F5, F7–F8 | LP / easy MIP | $10^{-4}$ (strict) / $10^{-2}$ (practical) | LP relaxation tight; optimal is exact |
| F6 | Hard MIP (MOQ, pack size) | $10^{-2}$ (both tiers) | 60s time limit may yield near-optimal solutions; pack-size rounding creates inherent gaps |

### E.1 Verification Layer Configuration

Table 20 summarizes the hyperparameters for each verification layer.

Table 20: Complete parameter configuration for ReLoop verification layers.

| Layer | Parameter | Value | Description |
|---|---|---|---|
| **L1** Execution | `timeout` | 60 s | Subprocess execution timeout |
| | `max_regenerations` | 3 | Regeneration attempts on FATAL |
| | `duality_gap_threshold` | 0.01 | Primal–dual gap threshold (1%, INFO only) |
| **L2** CPT (Constraint Presence) | `missing_threshold` | 0.05 | $< 5\%$ change $\rightarrow$ WARNING |
| | `uncertain_threshold` | 0.30 | 5–30% change $\rightarrow$ INFO |
| | `max_candidates` | 10 | Max constraints to test per problem |
| **L2** OPT (Objective Presence) | `missing_threshold` | 0.05 | $< 5\%$ change $\rightarrow$ WARNING |
| | `uncertain_threshold` | 0.30 | 5–30% change $\rightarrow$ INFO |
| | `max_candidates` | 10 | Max objective terms to test per problem |
| **Pipeline** | $N$ (repair budget) | 3 | Total repair iterations allowed |
| | $\tau_r$ (regression guard) | 0.04 | Rollback if objective shifts $>4\%$ |

### E.2 Code Generation

ReLoop uses Chain-of-Thought (CoT) prompting with four reasoning stages in a single LLM call. The generation pipeline uses an *extraction-first* strategy with a self-contained fallback:

1. **Try extraction**: An LLM call extracts all numerical parameters from the problem description into a structured JSON dictionary. If successful, a CoT prompt instructs the model to reference this pre-loaded `data` variable (e.g., `data["capacity"]`) rather than embedding values.
2. **Fallback**: If extraction fails (invalid JSON, empty result, or the generated code contains `json.loads`), the pipeline falls back to self-contained generation where the LLM embeds all data directly in the code.

The extraction path enables L2 behavioral testing via data-dict perturbation. When extraction fails, L2 falls back to source-code AST perturbation (Section E.4).

**Chain-of-Thought Prompt (Self-Contained Fallback).** The following prompt is used when data extraction fails. The data-reference variant shares the same four-step structure but adapts Steps 2, 3, and 4 for external data access (see note below).

**Chain-of-Thought Generation Prompt**

```
[System] You are an optimization expert who solves problems with
         step-by-step reasoning.

Solve this optimization problem using chain-of-thought reasoning.

## Problem
{problem_description}

---
## STEP 1: UNDERSTAND THE PROBLEM
First, analyze the problem:
- What is the objective? (minimize cost / maximize profit / etc.)
- What decisions need to be made?
- What constraints exist?
- What parameters are given?

## STEP 2: FORMULATE THE MATHEMATICAL MODEL
Write the formal model:
- Sets and indices
- Parameters (extract all numerical values from the problem
  description)
- Decision variables with domains
  **Variable Type**: For each variable, explicitly decide CONTINUOUS,
  INTEGER, or BINARY. Look for context where fractional values would
  be physically meaningless (e.g., number of trucks, workers to hire,
  items to select). State your choice and reasoning.
- Constraints in mathematical notation
- Objective function

## STEP 3: GENERATE GUROBI CODE
Write self-contained Python code using gurobipy.

**CRITICAL RULES:**
1. Define ALL data within your code (extract numbers from the problem
   description above)
2. Model variable must be named `m`
3. Set `m.Params.OutputFlag = 0`
4. Print exactly: `print(f"status: {m.Status}")` and
   `print(f"objective: {m.ObjVal}")`
5. Implement ALL constraints mentioned in the problem description
   (not just those in Step 2 -- re-read the problem to ensure
   nothing is missed)
6. Include ALL cost/revenue terms from the problem in the objective
   function

**Big-M Guidelines (if using indicator/logical constraints):**
- NEVER hardcode Big-M values like `M = 1e6`
- ALWAYS compute M dynamically from data parameters

**Edge Case Handling:**
- Check array length before iteration
- Avoid division by zero: `max(value, 1e-6)`

## STEP 4: VERIFY COMPLETENESS
Before finalizing, cross-check your code against the original problem:
- Does the objective include EVERY cost/revenue term mentioned in the
  problem?
- Is EVERY constraint from the problem implemented in the code?
- Are all numerical values correctly extracted from the problem
  description?
If anything is missing, fix the code before returning it.

---
Now solve the problem. Show your reasoning for Steps 1-2, then
provide the final code in a ```python block.
```

**Data-Reference Variant.** When extraction succeeds, the prompt differs in three steps:

- **Step 2**: "*Parameters (reference the data keys listed below)*" replaces "*Parameters (extract all numerical values from the problem description)*".
- **Step 3**: "*Write Python code using gurobipy*" (no "self-contained"). Rule 1 becomes "*The `data` variable is PRE-LOADED with the problem data. Do NOT define or redefine `data`. Just use `data["key"]` directly.*" An additional Rule 7 is added: "*Do NOT use `import json` or `json.loads()`. Data is already a Python dict.*"

- A data schema section ("`## Available Data Keys`") is inserted between Steps 3 and 4, listing all keys with types and dimensions (but not values).
- **Step 4**: "*Are data keys accessed correctly?*" replaces "*Are all numerical values correctly extracted from the problem description?*"

### E.3 L1: Execution Verification

L1 is a blocking layer that verifies code executability. It performs four sequential checks:

1. **Syntax check**: Compile-time validation via `compile()`
2. **Execution**: Run code in a sandboxed subprocess with 60 s timeout
3. **Solver status**: Check for INFEASIBLE (with IIS diagnostics), UNBOUNDED (with unbounded ray variables), or TIMEOUT
4. **Duality check**: If OPTIMAL, compare primal–dual gap. Gap $> 1\%$ emits INFO (does *not* trigger repair)

A failure in any of the first three checks emits a FATAL diagnostic and triggers the regeneration loop (up to 3 attempts).

**L1 Regeneration Prompt.** When L1 detects a FATAL error, the pipeline regenerates code using the error message as feedback. The data instructions section is *conditional*: if the code uses an external data dict, it includes the data schema and access rules; if the code is self-contained, it states "*Code is self-contained — all data is defined within the code itself.*"

**L1 Regeneration Prompt**

```
[System] You fix broken optimization code. Ensure the new code is
         syntactically correct and handles all edge cases.

The previous code failed to execute. Generate a new, correct version.

## Problem
{problem_description}

## Previous Code (FAILED)
```python
{failed_code}
```

## Error
{error_message}

{data_instructions}

## Instructions
1. Analyze why the previous code failed
2. Generate completely new code that avoids the error
3. Handle edge cases (empty arrays, division by zero)

Return ONLY the corrected Python code in a ```python block.
```

Where `{data_instructions}` expands to either:

- **Data-dict mode**: "`## Data Structure` / The `data` variable is PRE-DEFINED with these keys: {schema} / **CRITICAL**: Do NOT create `data = {...}`. Just use `data["key"]` directly."
- **Self-contained mode**: "`## Note` / Code is self-contained — all data is defined within the code itself."

### E.4 L2: Constraint Presence Testing (CPT)

L2 CPT tests whether constraints expected from the problem description are actually enforced in the generated code, using solver-based perturbation.

**Constraint Extraction Prompt.** An LLM extracts candidate constraints from the problem description. The prompt provides the list of available data parameter keys (but not values) to help the LLM identify relevant parameters.

**L2 CPT Constraint Extraction Prompt**

```
Analyze this optimization problem and extract the KEY CONSTRAINTS
that should be present in the model.

## Problem Description
{problem_description}

## Available Data Parameters
{list(data.keys())}

## Task
Identify constraints that are REQUIRED by the problem. Focus on:
1. Capacity constraints (resource limits, maximum values)
2. Demand constraints (minimum requirements, must-satisfy conditions)
3. Balance constraints (flow balance, inventory balance)

## Output Format
Return ONLY a JSON array with this exact format:
```json
[
  {"description": "minimum protein requirement",
   "type": "demand", "parameters": ["min_protein"]},
  {"description": "capacity limit on production",
   "type": "capacity", "parameters": ["capacity"]}
]
```

Return ONLY the JSON array, no explanation.
```

**Testing Procedure.** For each candidate constraint (up to 10), the pipeline applies extreme perturbation to the associated parameter and measures objective change. A *dual-strategy* perturbation approach is used:

1. **Strategy 1 (data-dict)**: Perturb the parameter in the external data dictionary and re-execute the original code. Used when the code reads from the `data` variable.
2. **Strategy 2 (source-code fallback)**: If the code embeds data directly in source (detected automatically), or if Strategy 1 produced $< 1\%$ objective change in hybrid mode, the pipeline falls back to AST-based source-code perturbation: it locates the parameter's assignment in the code via fuzzy name matching, modifies the literal value, and re-executes.

The perturbation factor depends on constraint type:

- Capacity constraints: $\times 0.001$ (near-zero)
- Demand constraints: $\times 100$ (extreme increase)
- Other constraints: $\times 0.01$ (1% of original)

Result classification:

1. If perturbation causes INFEASIBLE: constraint is present $\rightarrow$ PASS
2. Compute change ratio: $r = |z_{\text{new}} - z^*| \,/\, |z^*|$ (absolute change used when $|z^*| < \varepsilon$)
3. Classify (consistent with §3.3.2):
   - $r < \tau_\ell = 5\%$: Constraint likely missing $\rightarrow$ WARNING (triggers repair)
   - $\tau_\ell \leq r \leq \tau_h = 30\%$: Uncertain $\rightarrow$ INFO (no repair)
   - $r > \tau_h$ or infeasibility: Constraint present $\rightarrow$ PASS

### E.5 L2: Objective Presence Testing (OPT)

L2 OPT tests whether expected cost and revenue terms are present in the generated objective function, using the same perturbation methodology as CPT.

**Objective Term Extraction Prompt.**

**L2 OPT Extraction Prompt**

```
Analyze this optimization problem and extract the KEY OBJECTIVE
FUNCTION TERMS (cost and revenue components) that should be present
in the model's objective function.

## Problem Description
{problem_description}

## Available Data Parameters
{list(data.keys())}

## Task
Identify cost and revenue terms that MUST appear in the objective
function. Focus on:
1. **Cost terms**: purchasing/procurement cost, holding/storage cost,
   transportation cost, shortage/backorder cost, setup/fixed cost,
   penalty cost
2. **Revenue terms**: sales revenue, demand revenue, return/salvage
   value

For each term, identify which data parameter(s) provide its
coefficient.

## Output Format
Return ONLY a JSON array with this exact format:
```json
[
  {"description": "unit purchasing cost",
   "role": "cost", "parameters": ["unit_cost"]},
  {"description": "sales revenue per unit",
   "role": "revenue", "parameters": ["selling_price"]}
]
```

Return ONLY the JSON array, no explanation.
```

**Testing Procedure.** For each candidate objective term (up to 10), the same dual-strategy perturbation is applied. The perturbation factor depends on the term's role:

- Cost terms: $\times 0.001$ (near-zero)
- Revenue terms: $\times 100$ (extreme increase)
- Other terms: $\times 0.01$ (1% of original)

Classification follows the same thresholds as CPT ($< 5\%$ WARNING, 5–30% INFO, $\geq 30\%$ PASS), except that infeasibility is not expected from objective perturbation and is treated as a skipped test (no result emitted).

### E.6 Repair Pipeline

The repair loop processes unified `Diagnostic` objects produced by L1 and L2. Diagnostics are split into *actionable* issues (`triggers_repair=True`, severity WARNING or above) and *reference-only* items (INFO, likely normal). The repair loop runs up to $N = 3$ iterations and terminates early if:

- Status is VERIFIED with no actionable diagnostics (skip repair entirely)
- Repair produces identical code (no change)
- Repair does not change status or objective (plateau)
- Regression detected: objective shifts $> 4\%$, status degrades, or solution crashes

**Repair Prompt.** The repair prompt is assembled dynamically by `build_repair_prompt()` from the list of `Diagnostic` objects. The data section and safety rules adapt based on whether the code uses an external data dict or is self-contained.

**Repair Prompt (build_repair_prompt)**

```
[System] You are an optimization code repair expert.

CRITICAL RULES:
1. ONLY fix the actionable issues listed in the ISSUES DETECTED
   section
2. Items in REFERENCE ONLY are for context -- DO NOT modify code
   based on them
3. Be conservative -- only make changes that are clearly necessary
4. Preserve all working code -- only change what is broken
5. Do NOT change hardcoded data values unless the diagnostic
   evidence specifically requires it

Fix this optimization code based on the behavioral verification
report.

## Problem
{problem_description}

{data_section}

## Current Code
```python
{code}
```

## Current objective value: {current_obj}

---
## ISSUES DETECTED ({N} actionable)

=== Issue 1 [{layer}] [{severity}] ===
Type: {issue_type}
Target: {target_name}
Evidence: {evidence}

=== Issue 2 ... ===

---
## REFERENCE ONLY (DO NOT FIX)

+--------------------------------------------------------------------+
| Below items are NORMAL in 80%+ of cases.  DO NOT CHANGE.       |
+--------------------------------------------------------------------+

1. [{layer}] {issue_type} -- {target_name}
   {evidence}
   Action: DO NOT FIX (unless 100% certain this is an error)

---
## REPAIR INSTRUCTIONS

1. Read each Issue carefully, especially the Evidence field
2. Identify the root cause in your code for each actionable issue
3. Fix ALL actionable issues above
4. DO NOT fix items in the REFERENCE section -- they are likely
   normal
5. Preserve all working code -- only change what is broken

{safety_rules}

Return the COMPLETE fixed code in a ```python block.
```

Where `{data_section}` and `{safety_rules}` are conditional:

- **Data-dict mode**: The data section shows the schema (keys, types, dimensions but not values). Safety rules prohibit redefining `data`, using `json.loads()`, and mutating data contents.
- **Self-contained mode**: The data section states "Code is self-contained." Safety rules prohibit removing or changing hardcoded data values unless diagnostic evidence requires it.

**Safety Guardrails.** Before accepting repaired code, a safety validator (`repair_safety.py`) checks for dangerous operations using both regex pattern matching and AST analysis:

1. **Data reassignment**: Redefining `data` with a dict literal (`data = {...}`) is blocked. Exception: `data = json.loads(...)` is allowed as it re-parses existing data rather than fabricating values.

2. **Data mutation**: Modifying data contents (`data["key"] = value`) is blocked.
3. **Dangerous imports**: `os` and `subprocess` modules are blocked.

If violations are detected, a guided re-repair is attempted once (prepending violation details to the prompt). This safety retry does *not* consume the repair budget. If the retry also fails validation, the original (pre-repair) code is kept.

# F Evaluation Details

## F.1 Correctness Criteria

An instance is **correct** if:

1. **Status match:** Predicted feasibility equals ground truth
2. **Objective match:** For feasible instances, relative error $< \epsilon$

Table 21: Tolerance thresholds by benchmark and family.

| Benchmark | Families | Problem Type | Tolerance $\epsilon$ |
|---|---|---|---|
| RetailOpt-190 | F1–F5, F7–F8 | LP | $10^{-4}$ (strict) / $10^{-2}$ (practical) |
| RetailOpt-190 | F6 | MIP | $10^{-2}$ (both tiers) |
| MAMO-ComplexLP | – | LP | $10^{-6}$ |
| IndustryOR | – | Mixed | $10^{-6}$ |

RetailOpt-190 reports accuracy at two tolerance tiers to capture different aspects of formulation quality: $\epsilon = 10^{-4}$ measures strict formulation correctness (exact match), while $\epsilon = 10^{-2}$ captures practically correct solutions with minor numerical discrepancies. For MIP family F6, both tiers use $\epsilon = 10^{-2}$ since integer variables under the 60-second time limit may yield near-optimal rather than provably optimal solutions. Cross-benchmark experiments adopt $\epsilon = 10^{-6}$ following the SIRL evaluation protocol [10].

## F.2 Metrics

All metrics are computed over the full dataset of $N$ instances.

**Primary Metrics.**

$$\text{Exec\%} = \frac{|\{i : C_i \text{ executes} \wedge \text{solver returns feasible } z_i\}|}{N} \times 100 \tag{13}$$

$$\text{Acc\%}(\epsilon) = \frac{|\{i : |z_i - z_i^{\text{gt}}|/|z_i^{\text{gt}}| < \epsilon\}|}{N} \times 100 \tag{14}$$

We report Acc% at two tolerance levels for RetailOpt-190: Acc%($\epsilon = 10^{-4}$) and Acc%($\epsilon = 10^{-2}$). For cross-benchmark experiments, we report Acc%($\epsilon = 10^{-6}$) only. All results use pass@1 (single attempt, greedy decoding).

**Derived Metrics (Appendix Tables).** For analyses in the appendix, we also compute:

$$\text{SF\%} = \text{Exec\%} - \text{Acc\%} \quad \text{(silent failure rate)} \tag{15}$$

An instance is considered a *silent failure* if it executes and the solver returns a feasible solution, but the objective does not match ground truth within tolerance. SF% quantifies the feasibility–correctness gap discussed in §5.2.

# G Experimental Setup

## G.1 Model Configuration

**Evaluation Configurations.** All five models are evaluated under three configurations: *Base/Native* (direct generation for foundation models, own fine-tuned format for SFT/RL), *CoT* (structured

Table 22: Model configurations used in experiments. All models use greedy decoding for reproducibility.

| Type | Model | Provider | Temp. | Max Tokens | Notes |
|---|---|---|---|---|---|
| Foundation | Claude Opus 4.6 | Anthropic API | 0.0 | 8192 | |
| Foundation | DeepSeek-V3.2 | DeepSeek API | 0.0 | 8192 | |
| Foundation | Qwen3-32B | Local (vLLM [27]) | 0.0 | 8192 | BF16 |
| Offline SFT | OptMATH-Qwen2.5-32B | Local (vLLM) | 0.0 | 8192 | BF16 |
| Online RL | SIRL-Qwen2.5-32B | Local (vLLM) | 0.0 | 8192 | BF16 |

chain-of-thought with Understand → Formalize → Synthesize → Verify stages), and *ReLoop* (CoT + two-layer behavioral verification with diagnosis-guided repair, $N = 3$).

**L2 Verification LLM.** For L2 behavioral testing (CPT and OPT), the constraint and objective extraction uses the same LLM that generates the code, with temperature $= 0$; cross-model extraction is left for future work.

### G.2 Solver Configuration

Ground truth computed with Gurobi 11.0:

- `TimeLimit`: 60 seconds
- `MIPGap`: 0.01 (1% optimality gap for MIP)
- `Threads`: 1 (single-threaded for reproducibility)
- `Seed`: 0 (fixed random seed)
- `OutputFlag`: 0 (suppress solver logs)

### G.3 Benchmark Details

Table 23: Benchmark statistics.

| Benchmark | Instances | Avg Tokens | Tolerance | Format |
|---|---|---|---|---|
| RetailOpt-190 | 190 | ∼2,900 | $10^{-4}$ / $10^{-2}$ | Data-embedded |
| MAMO-ComplexLP | 203 | ∼459 | $10^{-6}$ | Data-embedded |
| IndustryOR | 100 | ∼267 | $10^{-6}$ | Data-embedded |

All benchmarks use data-embedded format (full data in prompt) for evaluation, ensuring comparability with prior work. RetailOpt-190 also provides a schema-based format used internally by the ReLoop verification pipeline for parameter perturbation.

**Baselines.** All Base, CoT, and ReLoop values in Tables 7 and 8 come from our own runs under the protocol of this appendix, including the Base configurations of DeepSeek-V3.2, Qwen3-32B, OptMATH-32B, and SIRL-32B. For OptMATH-32B and SIRL-32B, our reproduced Base accuracies differ from the values reported by Chen et al. [10]; we report our reproductions so that every configuration is evaluated under the same protocol. For context, Chen et al. [10] (Table 1) report the following results, which we did not rerun: GPT-4 (49.3% / 33.0%), DeepSeek-R1 (67.9% / 45.0%), and OpenAI-o3 (51.2% / 44.0%) on MAMO-ComplexLP / IndustryOR.

### G.4 Reproducibility

All LLM calls use greedy decoding (temperature $= 0$), ensuring deterministic outputs. Solver random seed is fixed at 0. All results are single-run (pass@1); we verified reproducibility on a random subset of 20 instances, observing identical outputs across repeated executions.

## H Per-Family Analysis on RetailOpt-190

Table 24 presents per-family accuracy for all five models. This breakdown reveals which constraint interaction patterns are most challenging and where ReLoop provides the largest gains.

Table 24: Per-family results on RetailOpt-190 (Acc%, $\epsilon = 10^{-4}$). Base = direct generation for foundation models, native format for SFT/RL models. +ReLoop = CoT + L1–L2 for all models.

| | | **Claude Opus 4.6** | | **DeepSeek V3.2** | | **Qwen3-32B** | | **OptMATH-32B** | | **SIRL-32B** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Family** | **#** | Base | +ReLoop | Base | +ReLoop | Base | +ReLoop | Base | +ReLoop | Base | +ReLoop |
| **F1 Core Ops** | **20** | 55.0 | **95.0** | 5.0 | **5.0** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **F2 Assort & Sub** | **30** | 50.0 | **53.3** | 0.0 | **3.3** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **F3 Resource** | **20** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **F4 Demand Dyn** | **30** | 3.3 | **20.0** | 0.0 | **10.0** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **F5 Feasibility** | **20** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **F6 Discrete Log** | **20** | 0.0 | 0.0 | 0.0 | **20.0** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **F7 Network & ME** | **30** | 20.0 | **26.7** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **F8 Omni-channel** | **20** | **50.0** | **50.0** | 0.0 | **10.0** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **Total** | **190** | **22.6** | **31.1** | **0.5** | **5.8** | **0.0** | **0.0** | **0.0** | **0.0** | **0.0** | **0.0** |

The breakdown shows four patterns. F1 (Core Operations) shows the largest absolute gain for Claude (55.0%→95.0%), which suggests that ReLoop is most effective on basic perishable inventory dynamics. F3 (Resource) and F5 (Feasibility Stress) remain at 0% across all models, suggesting that multi-resource coupling and deliberately tight feasibility boundaries exceed current LLM modeling capabilities regardless of verification. In F6 (Discrete Logistics), Claude gains nothing while DeepSeek rises from 0% to 20.0%, which suggests that L1 recovery from crashes caused by integer-variable errors accounts for DeepSeek's gain on these MIP problems. The 32B models (Qwen3, OptMATH, SIRL) score 0% in every family, which is consistent with these models lacking the capacity for compositional retail optimization at this level of complexity.

## I Injected-Defect Characterization of Behavioral Verification

To characterize L2 against construction-known labels, we generated 240 single-component knockout mutants of the RetailOpt-190 reference formulations across all 38 archetypes (one constraint family or objective term removed per mutant via disable switches in the reference solver), plus 76 structural-tier mutants (aging linkage, initial-condition) reported separately. Candidate constraints and objective terms were generated programmatically from the instance schema (*oracle extraction*), decoupling extraction error from detection error; solver settings match the evaluation protocol. None of the 240 omissions causes infeasibility, because deleting a constraint only relaxes the feasible region and deleting an objective term does not affect feasibility. Omission defects are therefore invisible to L1 by construction and form exactly the silent-failure class that L2 targets, whereas commission errors (over-constraining, missing slack, broken syntax) are handled by L1.

**Detection.** L2 detects 238/240 (99.2%): 83/83 removed constraints (CPT) and 155/157 removed objective terms (OPT), including 74/74 components inactive at nominal data and detectable only because extreme perturbation forces them to bind. The two misses share one mechanism: the purchasing coefficient also enters the per-period budget constraint, so perturbing it moves the objective through that constraint even with the objective term absent, which is the parameter-sharing case of Table 3. The intact-code false-positive rate is 26.7% of candidates (64/240; CPT 4/83, OPT 60/157), concentrated on objective terms and arising from a single mechanism: components genuinely present but economically inactive even under perturbation (e.g., cost terms of products the optimum never purchases). Its practical cost is limited by conservative repair (re-adding a present constraint leaves the feasible region unchanged), INFO-level non-repair, and the regression guard.

**Multiplier robustness and taxonomy independence.** Detection recall is 238/240 at each shrink multiplier, $10^{-4}$, $10^{-3}$, and $10^{-2}$, which span three orders of magnitude, and intact-code false positives are 63–64/240 (Table 25). Thirty-eight knockouts have zero-valued optima (lost-sales term removal) and are detected via the absolute-change fallback of Assumption 1. The inflate axis has

no test population on this benchmark (every removable component is capacity- or cost-typed), so robustness is scoped to shrink-direction perturbations. A taxonomy-free control (bidirectional extreme perturbation per parameter, no type labels, max response) attains 238/240 with fewer intact-code false positives (54/240), so detection does not depend on the capacity/demand/cost routing, whose only role is to halve the number of solver calls.

Table 25: Injected-defect detection across shrink multipliers (240 knockouts; 38 intact baselines).

| | **Typed routing** | | | **Taxonomy-free** |
|---|---|---|---|---|
| | $\times 10^{-4}$ | $\times 10^{-3}$ | $\times 10^{-2}$ | (bidirectional) |
| Recall | 238/240 | 238/240 | 238/240 | 238/240 |
| Intact-code FP | 63/240 | 64/240 | 64/240 | 54/240 |

## J Failure-Attribution and Label-Validity Audit

We audited a stratified sample of 55 residual failures (RetailOpt 40 by family; MAMO 15 by |GT| tertile) in two stages. First, an LLM pre-annotated the failures in two independent blind runs (raw agreement 76%, $\kappa = 0.57$). Second, an expert verified the labels, and a second author independently re-verified every non-modeling-error label and every disagreement through independent Gurobi re-solves and adversarial tests against the official reference generator. Every label carries a solver-checkable minimal-flip rationale with recomputed objectives.

**RetailOpt-190.** Of the 40 sampled failures, 39 are true modeling errors and 1 is a numerical or tolerance case. None is a defensible alternative reading: we tested each hypothesized reading by forcing the alternative structure in the reference generator, and the resulting optimum matched neither the ground truth nor the model output. For example, toggling substitution leaves the ground truth at 528,518, while the model returns 488,753, 7.5% below the true optimum, which indicates a relaxed or missing constraint. None is an equivalent formulation penalized by the metric. Of the 43 true modeling errors in the audited sample (both benchmarks), 42 are structural, namely wrong inter-mechanism coupling (22) or missing structural linkage (20), and one is a data-extraction slip. The reference generator reproduces all 190 published ground truths to $10^{-6}$ and matches Appendix A line by line.

**MAMO-ComplexLP.** We tested and rejected the hypothesis that integer-valued ground truths on diet problems with divisible-looking quantities are mislabeled. MAMO applies a consistent convention: countable goods carry integer ground truths (197/203 of ground truths are integer-valued), and genuinely continuous quantities carry fractional ones (all 6 fractional ground truths use continuous units). Independent LP+IP re-solves of all 26 integer-labeled diet instances confirm that each integer value is the intended optimum over discrete items. This convention agrees with the variable-type reasoning of Stage 2, and in all 11 L2-corrected MAMO instances the diagnosis-guided repair restored the integrality dropped by continuous relaxation (variable types changed; constraint sets unchanged); none of these instances involves a modified label. We also disclose that one instance was corrupted by our own preprocessing (ground truth 162 recorded as a $-9999$ sentinel; restored; no effect on scores) and that two isolated data errata are documented, with an effect on the scores of at most two instances.

## K Benchmark Composition Audit

Under the standard definition of revenue management, the allocation or pricing of fixed perishable capacity (booking control, fare classes, markdown pricing), neither MAMO-ComplexLP nor IndustryOR contains revenue-management instances. MAMO's own category metadata labels every instance `complex_lp`, and most problems in both sets are production, blending, transportation or network-flow, scheduling, and assignment problems. A minority carry revenue- or profit-maximization objectives (11/203 in MAMO; 33/100 in IndustryOR), which does not place a problem in the revenue-management class, and the single keyword hit ("contract bookings") was judged by a human reviewer to be a false positive. The MAMO instances corrected by L2 are cost-minimization diet problems with

no retail or revenue-management character; Appendix I measures the detector across the mechanism families of RetailOpt-190.

## NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and Section 1 state four contributions: structured generation as the largest source of accuracy gains for foundation models on compositional problems, behavioral verification as the largest single contributor on localized defects, the ReLoop pipeline with regression-guarded repair, and the RetailOpt-190 benchmark. Sections 5.2–5.4 and Appendices H–K report the supporting results, and the Limitations paragraph in Section 6 states the scope.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The Limitations paragraph in Section 6 states four limitations: the supported formulation class and the shrink-only multiplier evidence, the conflict between CoT and the output format of SFT models, possible error correlation from shared LLM extraction, and the detection boundaries listed in Table 3.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper states no theorems. Assumption 1 is a design assumption with explicit conditions and edge-case guards, and it is evaluated empirically in Section 5.4 and Appendix I.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 3 and Algorithm 1 specify the pipeline; Appendix E gives all prompts and verification parameters; Appendices A–C specify the benchmark; and Appendices F and G give the metrics, tolerances, model versions, decoding parameters, and solver settings.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The ReLoop code and the RetailOpt-190 benchmark are public at https://github.com/junbolian/ReLoop under the license included in the repository; Appendices C and E document the data format, prompts, and configuration used to produce the results.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: No models are trained. All evaluations use greedy decoding (temperature $= 0$, max_tokens $= 8192$) and repair budget $N = 3$; the L2 perturbation factors (capacity $\times 0.001$, demand $\times 100$, cost $\times 0.001$, revenue $\times 100$) are in Appendix E, and the tolerances ($\epsilon \in \{10^{-4}, 10^{-2}, 10^{-6}\}$), benchmark statistics, and Gurobi 11.0 settings are in Appendices F and G.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Section 5.2 reports a paired exact sign test for the CoT gain ($p \approx 0.002$) and an archetype-level cluster bootstrap 95% confidence interval ($[1.6, 16.8]$pp), and Section 5.3 reports a sign test for the L2 gain on MAMO ($p \approx 0.02$). All runs are single-run pass@1 with greedy decoding; repeated runs on 20 instances produced identical outputs (Appendix G).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Claude Opus 4.6 and DeepSeek-V3.2 are accessed through commercial APIs, and ReLoop adds about $3\times$ the base token cost (Section 5.1). The 32B models run locally through vLLM in BF16 (Appendix G); per-instance inference completes in seconds on a single high-memory GPU, and single-threaded Gurobi 11.0 solves complete in seconds.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

Justification: The research involves no human subjects and no personal data, and the released artifacts are synthetic optimization scenarios and a verification framework.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: *Positive*: more reliable LLM-based optimization reduces the risk of confident but wrong recommendations in supply chain, inventory, and logistics decisions. *Negative*: users may place more trust in verified pipelines than their detection scope supports; Table 3 states which errors L2 cannot detect, and the 90-point feasibility–correctness gap in Section 5.2 is a caution against deploying unverified LLM-generated optimization code.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: RetailOpt-190 contains synthetic optimization scenarios with no privacy, security, or dual-use concerns, and ReLoop is a verification framework rather than a generative model; no high-risk artifacts are released.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All evaluated LLMs and external benchmarks are cited (Section 5.1, Appendix G) and used under their terms of use: commercial APIs for the closed-weight models, the licenses on the HuggingFace model cards for the open-weight 32B models, and the licenses of the original publications for the benchmarks. Gurobi 11.0 is used under an academic license.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: RetailOpt-190 is documented in Section 4 and Appendices A–C: the 38 archetypes in 8 families (Appendix B), the reference MILP (Appendix A), the JSON schema and data access patterns (Appendix C), the instance generation procedure (Appendix B.3), and the ground-truth solver settings (Appendix D). It is released with the code at the repository listed above.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The research involves no crowdsourcing and no human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: No human subjects research is conducted, so IRB approval is not applicable.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are the objects of evaluation (five models generate optimization code) and a component of the method: the L2 layer uses an LLM to extract candidate constraints and objective terms for solver-based perturbation testing (Section 3.3.2, Appendix E). All configurations, prompts, and decoding parameters are documented in Appendices C–G.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.