arXiv:2602.15983v2 [cs.SE] 29 Apr 2026

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

Junbo Jacob Lian[1,2,5,*], **Yujun Sun**[1,*], **Huiling Chen**[3], **Chaoyu Zhang**[4], **Hanzhang Qin**[5], **Chung-Piaw Teo**[5,†]

[1] McCormick School of Engineering, Northwestern University
[2] Wenzhou Buyi Pharmacy Chain Co., Ltd.
[3] College of Computer Science and Artificial Intelligence, Wenzhou University
[4] Department of Decision Analytics and Operations, City University of Hong Kong
[5] Institute of Operations Research and Analytics, National University of Singapore

jacoblian@u.northwestern.edu, yujunsun2026@u.northwestern.edu, chenhuiling.jlu@gmail.com
cy.zhang@cityu.edu.hk, hzqin@nus.edu.sg, bizteocp@nus.edu.sg

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

Abstract

Large language models (LLMs) can translate natural language into optimization code, but *silent failures* pose a critical risk: code that executes and returns solver-feasible solutions may encode semantically incorrect formulations—a feasibility–correctness gap reaching 90 percentage points on compositional problems. We introduce **ReLoop**, which addresses this gap through two complementary mechanisms. *Structured generation* decomposes code production into a four-stage reasoning chain (understand, formalize, synthesize, verify), preventing formulation errors at their source. *Behavioral verification* detects errors that survive generation by testing whether the formulation responds correctly to solver-based parameter perturbation—an external semantic signal that bypasses LLM self-review and requires no ground truth. The two mechanisms are complementary by error structure: structured generation drives the largest gains on compositional problems (**+8.5pp** accuracy on RetailOpt-190 with Claude Opus 4.6), while behavioral verification dominates on localized defects (**+4.4pp** on MAMO-ComplexLP, its largest contribution across benchmarks). Combined with diagnostic execution recovery, ReLoop reaches **100% executable code** on Claude Opus 4.6 and consistently improves accuracy on chat-tuned foundation models across three benchmarks; we further identify a known limitation of narrowly-tuned SFT models, whose learned output formats are brittle to chain-of-thought prompts—an interaction we document and analyze. We release **RetailOpt-190**, 190 compositional retail optimization scenarios targeting the multi-constraint interactions where LLMs most frequently fail. Code and benchmark: https://github.com/junbolian/ReLoop.

Keywords Automated Optimization Modeling · Behavioral Verification · Silent Failures · Large Language Models

1 Introduction

Optimization solvers cannot distinguish a correct model from a wrong one that happens to be feasible. This fundamental blindness creates a critical failure mode for LLM-based optimization: *silent failures*—code that executes without errors and returns solver-feasible solutions, yet encodes semantically incorrect formulations. An inventory model ignoring perishability generates plans with

expired stock; a routing model omitting time-window constraints dispatches trucks that arrive after customers leave. In both cases, the solver reports "optimal"—to the wrong problem.

Silent failures are not edge cases. On compositional problems requiring multiple interacting constraints (capacity, perishability, substitution, lead times), state-of-the-art models achieve up to 91.1% solver-feasibility but only 0.5% formulation correctness—a 90-point *feasibility–correctness gap* (§5.2). This gap persists because existing verification cannot detect it: solver feedback catches syntax errors, not missing constraints; LLM self-critique [1, 2] inherits the reasoning gaps that caused errors; environment-guided refinement [3] uses execution feedback but cannot distinguish feasible-yet-wrong from correct; execution-based reranking [4] requires ground-truth solutions unavailable for novel problems.

Key Insight. Silent failures originate at the *modeling* stage—when the LLM translates a natural-language problem into a mathematical formulation—and persist because existing post-hoc checks cannot detect them. We address this gap at two levels.

At the generation level, expert operations research modelers do not directly write code: they decompose the problem, write the mathematical model with explicit variable-type decisions, then implement and cross-check. Encoding this disciplined workflow into a four-stage chain-of-thought prompt—understand, formalize, synthesize, verify—prevents many formulation errors at their source, before any external verification is needed.

At the verification level, correct optimization models satisfy behavioral invariants testable without ground truth: (i) *Constraint presence*: perturbing a capacity limit to near-zero must change the objective—zero effect indicates the constraint is missing; (ii) *Objective completeness*: perturbing a cost coefficient must shift the objective—zero effect indicates the term was omitted. Unlike declarative self-review, which inherits the generating LLM's blind spots [2], solver-based perturbation provides an *external semantic signal* grounded in solver behavior rather than LLM introspection.

We introduce **ReLoop**, operationalizing this two-level approach: (1) *Structured generation* via a four-stage chain-of-thought mirroring expert practice (§3.2); (2) *Two-layer behavioral verification*—execution recovery with IIS-enhanced diagnostics (L1) and solver-based perturbation testing (L2); (3) *Diagnosis-guided repair* with regression protection.

Contributions.

1. We propose *structured generation* for optimization code, decomposing the modeling process into four stages that mirror expert practice—including explicit variable-type reasoning and self-verification—and show it is the primary accuracy driver on complex compositional problems (§3.2, §5.4).
2. We formalize *behavioral verification*—testing whether formulations respond correctly to solver-based parameter perturbation—as the first approach detecting silent failures without ground truth, and show it is the largest single contributor on problems with localized defects (§3.3, §5.4).
3. We design **ReLoop**, integrating structured generation, execution recovery, and behavioral testing into a unified pipeline with regression-guarded repair, and release **RetailOpt-190**, 190 compositional retail optimization scenarios (§3.3, §4).
4. Cross-benchmark ablation reveals that structured generation and behavioral verification are *complementary*: each dominates under different error structures, with gains across all foundation models and three benchmarks, though CoT can conflict with domain-specific fine-tuning (§5).

2 Related Work

LLM-Based Optimization Modeling. Translating natural language into mathematical programs has progressed through prompting [5, 6], supervised fine-tuning [7–9], reinforcement learning with solver feedback [10], and agentic workflows [11]. All optimize *generation quality*—through training or prompting—and are complementary to ReLoop. ReLoop contributes a domain-specific structured generation strategy (prompting-based, applicable to any model) and, uniquely, adds post-generation *behavioral verification*: detecting when a feasible model is semantically incorrect.

Verification and Self-Correction. Existing verification falls into four paradigms (Table 1), each insufficient for optimization. *Solver-based feedback* [10, 12] cannot detect feasible-yet-wrong

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]	✗	✓	✓	✓
LEVER† [4]	✓	✓	✗	✗
ReLoop (ours)	✓	✓	✓	✓

models—the core silent failure problem. *LLM self-critique* [1] fails because models cannot identify errors from their own reasoning gaps without external feedback [2]—the very gaps that produced the silent failure. *Environment-guided refinement* [3] incorporates execution feedback, but solver status and objective value cannot distinguish correct from feasible-yet-wrong. *Execution-based reranking* relies on training-time labels [4] or generated test cases [13]; both face challenges on optimization where ground-truth programs are scarce and reliable test-case generation requires the very semantic understanding the LLM is missing. ReLoop introduces solver-based perturbation as external semantic signal, providing the independent feedback that Huang et al. [2] identify as necessary for reliable correction.

Sensitivity Analysis and Program Verification. Sensitivity analysis studies how optima change with parameters [14, 15]. We repurpose it for verification: instead of *interpreting* sensitivity, we test whether it *exists*—a parameter producing zero sensitivity when it should affect the optimum indicates a missing component. This insight—using expected sensitivity patterns for formulation verification—is, to our knowledge, novel. Classical program verification requires formal specifications [16, 17]; our approach sidesteps this by testing properties any correct optimization must satisfy.

Benchmarks. Existing benchmarks—NL4Opt [18], MAMO [19], IndustryOR [7], OptMATH [9]—primarily test constraints in isolation, evaluating whether a model can encode a single constraint type rather than multiple interacting ones. RetailOpt-190 addresses this through systematic composition of 8 scenario families requiring multiple constraints to jointly bind (§4).

3 Method

ReLoop addresses silent failures through two complementary mechanisms: *structured generation* (§3.2) prevents formulation errors by decomposing code production into a disciplined four-stage reasoning chain that mirrors expert modeling practice, while *behavioral verification* (§3.3) detects errors that survive generation by testing formulation responses to solver-based perturbation. Execution recovery (L1) and diagnosis-guided repair (§3.4) complete the pipeline. Figure 1 illustrates the architecture; Algorithm 1 (Appendix E) formalizes it.

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.

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.

Silent failures are insidious: standard software testing—which checks execution success and solution feasibility—cannot detect them. Our verification approach exploits a behavioral invariant of correct optimization models:

Property 1 (Perturbation Sensitivity)**.** *Let $z^*(\theta)$ denote the optimal objective of a correctly formulated model parameterized by θ, and let θ_i govern a constraint that binds (or can be made to bind) under*

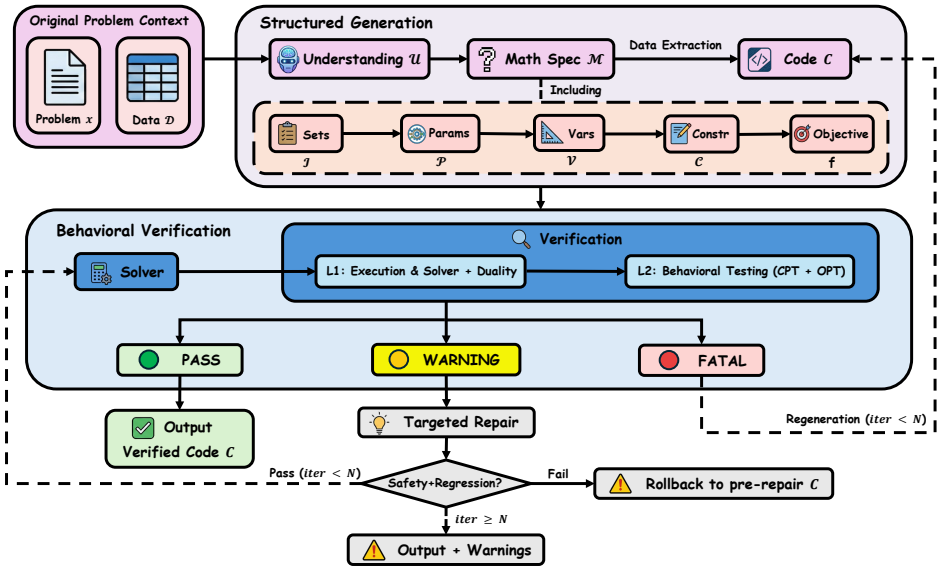

Figure 1: ReLoop overview. **Structured Generation** mirrors expert modeling practice: understand the problem, formalize the mathematical model with explicit variable-type reasoning, synthesize Gurobi code with data extraction, and self-verify completeness. **Behavioral Verification**: L1 checks execution 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. After budget N, ReLoop returns the best verified code.

perturbation, or a non-negligible objective term. Under sufficiently extreme perturbation $\theta_i \to \tilde{\theta}_i$, the objective must change substantially: $|z^*(\tilde{\theta}_i) - z^*(\theta)|/|z^*(\theta)| \gg 0$.[1]

The contrapositive provides our detection criterion: if perturbing a parameter that *should* affect the objective produces 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 conflates understanding, formalization, and implementation into a single output, where errors at each stage propagate silently. 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.

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. Variable type reasoning is particularly important: the prompt requires the LLM to determine whether each variable should be continuous, integer, or binary based on physical context (e.g., "can you order 2.7 pallets?"), addressing the common continuous-relaxation failure where models default to continuous variables for inherently discrete decisions.

[1] When $z^*(\theta) = 0$ our implementation falls back to absolute change, and induced infeasibility under perturbation is treated as a strong PASS signal (Appendix E); these edge cases preserve the detection contrapositive in practice.

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 3 (Synthesize). Generate executable Gurobi code $\hat{C}$ from $\mathcal{M}$, structured to access all data through `data["key"]` dictionary patterns rather than hardcoded literals. This is not merely a style convention—it is essential for L2's perturbation testing, which modifies parameter values in the runtime data dictionary. To support this, we first prompt the LLM to extract all numerical parameters from x into a structured JSON dictionary; the generated code then references this pre-loaded dictionary. If extraction fails (e.g., malformed JSON), we fall back to self-contained generation where the LLM embeds data directly in code; the perturbation engine automatically detects and handles both patterns (Appendix B.3).

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? Are all data values correctly accessed? If discrepancies are found, the model fixes the code before returning the final C. This self-check within the generation call catches errors that would otherwise propagate to verification.

3.3 Two-Layer Behavioral Verification

Our architecture is governed by two principles: (1) **Only L1 blocks output**—L2 provides diagnostics but never discards a valid solution, so the baseline objective z^* from L1 is guaranteed to be returned; (2) **Conservative repair**—only high-confidence issues (WARNING) trigger repair; ambiguous signals (INFO) are logged as reference only, preventing false-positive-driven repairs from introducing 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)—the minimal set of conflicting constraints—and feeds specific constraint names and bounds to the LLM, transforming opaque solver failures into actionable feedback. For UNBOUNDED status, L1 reports unbounded ray variables to guide variable-bound correction.

3.3.2 L2: Behavioral Testing (CPT + OPT)

L2 detects missing formulation components—constraints or objective terms that should be present according to the problem description but are absent from the generated code. L2 sidesteps the well-known self-consistency limit of LLM self-review [1, 2] by routing detection through the *solver*: the LLM only extracts candidate constraints and objective terms, while detection is performed via parameter perturbation and solver response—an external signal independent of the LLM's reasoning.

Detection mechanism. L2 operationalizes Property 1 through two symmetric sub-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

Table 3: Comparison with existing benchmarks.

Benchmark	Domain	Inst.	Multi-period	Compositional	Data-Code Sep.
NL4Opt [18]	Generic	289	Few	✗	✗
MAMO [19]	Generic	∼800	Some	✗	✗
IndustryOR [7]	Mixed	100	Some	✗	✗
OptMATH [9]	Mixed	∼360	Some	✗	✗
RetailOpt-190	**Retail**	190	**All**	✓	✓

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*)—triggers repair.
- $\tau_\ell \leq r \leq \tau_h = 30\%$: INFO (*uncertain*)—logged, no repair.
- $r > \tau_h$ or infeasibility: PASS (*present*)—confirmed active.

The asymmetric $[\tau_\ell, \tau_h]$ buffer reflects a deliberate preference for under-detection over over-repair (false positives trigger regressions); accuracy varies by $<1\%$ across $\tau_\ell \in [1\%, 10\%]$ and $\tau_h \in [20\%, 50\%]$.

3.4 Diagnosis-Guided Repair

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

Safety and regression guards. Three mechanisms prevent repair-induced regression: (1) a *safety check* blocks data-variable reassignment and dangerous imports (we observed repair LLMs fabricating values that corrupt the problem); (2) a *regression guard* rolls back any repair that crashes, regresses solver status, or shifts the objective by $>\tau_r = 4\%$, primarily catching repair pathologies (data fabrication, fundamental restructuring) rather than legitimate constraint additions, whose impact for the localized defects L2 detects is typically smaller than τ_r; (3) a *skip guard* bypasses repair when no WARNING exists. Algorithm 1 (Appendix E) formalizes the procedure.

4 RetailOpt-190 Benchmark

We introduce **RetailOpt-190**, a benchmark targeting *compositional constraint reasoning* (Table 3). All 190 instances are multi-period retail inventory optimization problems where a retailer must decide *how much to order*, *when to order*, and *how to allocate* products across periods to minimize total cost subject to interacting operational constraints. A typical instance involves 3–5 products over 4–8 periods, with 20+ parameters (demand forecasts, unit costs, storage capacities, shelf lives, lead times, supplier limits) and 10–30 constraints that must jointly bind. Difficulty arises from constraint *interactions*—e.g., perishability limits effective inventory, tightening storage constraints, forcing earlier ordering that conflicts with lead times—rather than linguistic complexity. Each instance is self-contained: the natural language description specifies all parameters, and a correct solution requires translating these into a coherent mathematical program.

Construction. The benchmark spans eight scenario families (F1–F8) following a *progressive composition* principle (Table 4): F1–F4 introduce core retail mechanisms (inventory dynamics, substitution, resource limits, demand variability) with increasing constraint interactions; F5 stress-tests feasibility boundaries with deliberately tight or infeasible configurations; F6 introduces discrete

Table 4: 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

variables (MOQ, pack sizes) requiring integrality reasoning; F7–F8 extend to multi-location network settings requiring multi-echelon coordination. Each of 38 archetypes is instantiated with 5 numerical variants ($\pm 15\%$, deterministic seeds), yielding 190 instances that test robustness to parameter changes while holding structure constant. Ground-truth optimal values are computed by Gurobi 11.0 [20] applied to hand-crafted formulations for each archetype (Appendix A).

Evaluation. Each instance is presented as a *data-embedded* prompt with JSON data inline, matching prior benchmarks [7, 18, 19]. The model generates self-contained Python code executed with a 60-second solver time limit. An instance is correct if feasibility status matches ground truth and $|y_{\text{pred}} - y_{\text{ref}}|/|y_{\text{ref}}| < \epsilon$. We report accuracy at two tiers: *strict* ($\epsilon = 10^{-4}$) requires near-identical objectives where all constraints and coefficients must be correct; *practical* ($\epsilon = 10^{-2}$) captures near-correct solutions with minor coefficient discrepancies, revealing formulations that are structurally right but numerically imprecise. MIP family F6 uses $\epsilon = 10^{-2}$ for both tiers due to solver tolerance. Cross-benchmark experiments use $\epsilon = 10^{-6}$ following Chen et al. [10].

5 Experiments and Results

5.1 Setup

Models. We evaluate five models spanning three paradigms: *Foundation LLMs*—Claude Opus 4.6 (Anthropic's frontier model), DeepSeek-V3.2 [21] (671B MoE), and Qwen3-32B [22] (dense 32B); *Offline SFT*—OptMATH-Qwen2.5-32B [9], fine-tuned on $\sim$17K curated optimization problems; *Online RL*—SIRL-Qwen2.5-32B [10], trained with solver execution status as verifiable reward. Each is evaluated under three configurations: *Base* (direct generation / own format), *CoT* (structured chain-of-thought), and *ReLoop* (CoT + L1–L2 verification with repair, $N = 3$). All use greedy decoding (temperature 0, pass@1) for reproducibility. ReLoop adds $\sim 3\times$ base cost in LLM tokens; solver calls complete in seconds (Appendix E).

Benchmarks. RetailOpt-190 (§4), MAMO-ComplexLP (203 instances from the challenging subset of MAMO) [19], 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 5 reveals that **feasibility is a poor proxy for correctness**: DeepSeek achieves 91.1% solver-feasibility but only 0.5% correctness—a 90-point gap that persists even after ReLoop (Claude: 100% Exec, 31.1% Acc—two-thirds remain silent failures).

Models fail in fundamentally different ways, with ReLoop preserving or improving every foundation-model metric combination relative to Base. On Claude, CoT is the primary accuracy driver (+8.5pp); on DeepSeek, CoT *collapses* execution (91.1%→53.2%) because the four-stage decomposition produces intermediate mathematical notation that fails to translate into valid Gurobi syntax—L1 fully recovers this regression through its multi-mode FATAL feedback (syntax tracebacks for parser

Table 5: 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

Table 6: 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**

errors, IIS for infeasibility, unbounded rays for unboundedness). Claude's accuracy plateaus at 31.1% because the remaining two-thirds are predominantly *structural* silent failures: fundamentally different (yet internally consistent) problem decompositions that perturbation cannot detect. The 32B models achieve near-zero accuracy because RetailOpt-190's compositional structure (20+ interacting constraints across multi-period, multi-product, multi-location dimensions) exceeds their reasoning capacity; ReLoop nonetheless improves their execution (OptMATH: 2.6%→17.9%; SIRL: 0.0%→1.6%), confirming that strong models benefit from CoT structuring while weaker models benefit from L1 crash recovery. L2 behavioral verification contributes additional gains across foundation models without degrading any foundation model's performance (its largest impact on MAMO's localized defects, §5.4); this monotonicity is guaranteed by L2's non-blocking, regression-guarded design.

The strict-to-practical gap ($\epsilon = 10^{-4}$ vs. 10^{-2}) provides diagnostic signal: DeepSeek's larger spread (5.8% vs. 11.1%) indicates near-correct solutions with coefficient errors, while Claude's smaller spread (31.1% vs. 35.3%) suggests predominantly structural errors.

5.3 Cross-Benchmark Generalization

To verify generalization beyond RetailOpt-190's retail domain, we evaluate on two external benchmarks neither used during development (Table 6): MAMO-ComplexLP contains shorter-prompt LP/MILP problems (∼459 tokens avg.) with well-isolated constraints; IndustryOR contains longer real-world problems requiring complex 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, benefits because ReLoop's structured generation complements its learned generation patterns rather than conflicting with them (unlike OptMATH). The exception is OptMATH on MAMO: Base accuracy (56.2%) collapses under CoT (30.0%) because the SFT model's rigid "problem→code" generation pattern cannot accommodate our four-stage reasoning template—84 instances crash and 65 previously correct solutions are destroyed.[2] More broadly, all three 32B models plateau well below frontier performance (best: 46.3% MAMO, 46.0% IndustryOR), suggesting that 32B-scale models lack sufficient reasoning capacity for complex industrial optimization even with verification support. Gain magnitude varies

[2]The unusually high Base score on MAMO may also reflect potential training–evaluation overlap, as OptMATH's ∼17K curated corpus could include publicly available benchmark problems. On IndustryOR, the Base–CoT gap is much smaller (34.0%→31.0%).

Table 7: 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**

with error structure: DeepSeek gains +12.0pp on IndustryOR (primarily via L1 execution recovery), while Claude gains +4.4pp on MAMO via L2 behavioral testing.

5.4 Ablation Studies

Table 7 isolates each component's marginal contribution across two benchmarks.

Direct → +CoT. On Claude, structured generation is the primary accuracy driver (+8.5pp; 20 corrected, 4 regressed). On DeepSeek, CoT collapses execution (91.1%→53.2%) because the structured format produces invalid code, motivating L1 as a dedicated recovery layer.

+CoT → +CoT+L1. L1 dominates execution recovery (+44.2pp Exec for DeepSeek, +5.8pp for Claude) and also improves accuracy. The mechanism is specific: when L1 detects infeasibility, IIS analysis identifies the minimal conflicting constraint set (e.g., "`capacity_limit` conflicts with `demand_fulfillment`"), giving the LLM a precise diagnosis that enables targeted reformulation rather than blind retry.

+CoT+L1 → +CoT+L1+L2. L2's contribution depends on 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 simpler problem structures produce *localized errors*—missing constraints or objective terms—precisely the defects perturbation testing can detect. On RetailOpt-190, L2 contributes execution recovery (Claude: 99.5%→100.0%) and practical accuracy gains (DeepSeek: 10.5%→11.1% at $\epsilon = 10^{-2}$), but strict accuracy ($\epsilon = 10^{-4}$) is unchanged because errors are predominantly structural (incorrect decompositions that produce plausible perturbation responses). On IndustryOR, we compute relative objective deviation $|z_{\text{pred}} - z_{\text{ref}}|/|z_{\text{ref}}|$ for all non-crashed instances and find a bimodal distribution: 34% have deviations below 1% (subtle coefficient differences undetectable by perturbation) and 47% exceed 10% (fundamental structural misunderstandings unrepairable in 3 iterations), leaving almost no instances in the correctable range. Per-family breakdowns are in Appendix H.

6 Conclusion

We introduced **ReLoop**, addressing silent failures in LLM-generated optimization code through *structured generation* that prevents formulation errors at their source and *behavioral verification* that detects surviving errors via solver-based perturbation without ground truth. Cross-benchmark ablation confirms their complementarity by error structure: structured generation drives the largest gain on RetailOpt-190's compositional problems (**+8.5pp** accuracy on Claude Opus 4.6); behavioral verification dominates on MAMO-ComplexLP's localized defects (**+4.4pp** on Claude, its largest L2 gain across our benchmarks). ReLoop preserves or improves every foundation-model metric combination through L2's non-blocking, regression-guarded design, and the 90-point feasibility–correctness gap underscores that semantic verification is becoming essential infrastructure for LLM-based optimization.

Limitations. Structured generation assumes format compatibility: CoT disrupts SFT models' learned patterns (84 crashes, 65 regressions on OptMATH/MAMO). L2 shares the generating LLM for constraint extraction, creating potential failure correlation; the symmetric regression threshold τ_r is calibrated for localized defects, conservatively holding major structural omissions to the L1

baseline. Three failure modes remain beyond scope: coefficient magnitude errors, formulation equivalence errors, and unrepresented problem structures.

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

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

[16] 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.

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

[18] 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.

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

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

[21] 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.

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

[23] 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 provides the modular MILP serving as semantic ground truth for RetailOpt-190. We first present the complete notation and decision variables, then give the full objective function and all constraint families with explicit equations. Finally, we discuss *design rationale* explaining why specific modeling choices 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)
- $\mathcal{K}_p = \{1, \ldots, \text{SL}_p\}$: 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 $d_{p,l,t} = \texttt{demand\_curve}[p][t-1] \times \texttt{demand\_share}[l]$
- SL_p: shelf life of product p in periods
- 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)
- γ_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
- ρ_p: return rate—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^{fix}: fixed ordering cost per order placed; c^{tr}: transshipment cost per unit
- MOQ: minimum order quantity; PS: pack size
- B: per-period purchasing budget (if active); ω: maximum waste fraction (if active)

Key Convention. Remaining-life index $k = \text{SL}_p$ denotes the *freshest* inventory; $k = 1$ denotes inventory expiring at the end of the current period. This FIFO-compatible convention is critical: LLMs frequently reverse this ordering, leading to incorrect aging dynamics where fresh inventory expires immediately (§A.7, D2).

A.2 Decision Variables

Table 8 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 mirrors the solver implementation, where variable creation is gated by 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:

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

Table 8: Decision variables. Variables marked with † 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 Φ^{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—it *cannot* be carried overnight. LLMs frequently apply holding cost to all buckets $k \geq 1$, effectively double-counting 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 9). 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 tight accounting is precisely what makes silent failures detectable: if an LLM omits or reverses any single constraint, the conservation structure breaks, producing solutions that 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 9: 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	$I_{p,l,1,k} = 0 \quad \forall\, k < \mathrm{SL}_p$ $I_{p,l,1,\mathrm{SL}_p} = A_{p,l,1}$	Without zero initialization for non-fresh buckets, unbounded phantom inventory yields objective ≈ 0 (D1).
C2. Fresh inflow	$I_{p,l,t,\mathrm{SL}_p} = A_{p,l,t} + \Delta^{\mathrm{tr}}_{p,l,t} + R_{p,l,t}$ where $A_{p,l,t} = Q_{p,l,t-\mathrm{LT}_p}$ if $t - \mathrm{LT}_p \geq 1$, else 0; $\Delta^{\mathrm{tr}}_{p,l,t} = \sum_{l':(l',l)\in\mathcal{E}^{\mathrm{tr}}} X_{p,l'\to l,t} - \sum_{l':(l,l')\in\mathcal{E}^{\mathrm{tr}}} X_{p,l\to l',t}$; $R_{p,l,t} = \rho_p \sum_k y_{p,l,t-1,k}$ 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	$I_{p,l,t+1,k} = I_{p,l,t,k+1} - y_{p,l,t,k+1}$ $\forall\, t \in \{1,\ldots,T-1\},\ k \in \{1,\ldots,\mathrm{SL}_p - 1\}$	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	$\sum_{k=1}^{\mathrm{SL}_p} y_{p,l,t,k} + L_{p,l,t} = d_{p,l,t} + S^{\mathrm{in}}_{p,l,t} - S^{\mathrm{out}}_{p,l,t}$ with $S^{\mathrm{out}}_{p,l,t} = \sum_{p'\in\mathcal{N}^{\mathrm{out}}_p} S_{p\to p',l,t}$, $S^{\mathrm{in}}_{p,l,t} = \sum_{p'\in\mathcal{N}^{\mathrm{in}}_p} S_{p'\to p,l,t}$, $S^{\mathrm{out}}_{p,l,t} \leq d_{p,l,t}$ (demand-route bound), 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.

Note that storage usage is measured on *start-of-period* inventory (before sales), as this represents the physical space occupied at the time replenishment decisions must be made. Product-specific usage coefficients γ_p enable 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 are non-obvious and merit explicit justification. Each addresses a specific modeling subtlety where 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 ≈ 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—effectively preventing expiration. The decrementing convention ensures that every unit faces a finite shelf life and eventually either sells or wastes.

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_{from}'s demand can be served by p_{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^{waste}. Applying holding cost c_p^{inv} to this same bucket would double-count the penalty. The formulation therefore charges holding cost only on end-of-period inventory with $k \geq 2$, cleanly separating the waste and holding cost channels.

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—ensuring breadth across 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 10).

Table 10: 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 with very high cost. This asymmetry provides a strong diagnostic signal: if an LLM's code returns infeasible on an F5 instance, it 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 11 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 11 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., "×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: $\texttt{seed} = \texttt{uint32\_le}\big(\texttt{SHA256}(n \texttt{ "|" } v)[:4]\big)$, 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: $\texttt{rng} = \texttt{default\_rng}(\texttt{seed})$
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 shift constraint binding patterns—demand volumes and storage capacities—while preserving the structural skeleton (shelf life, network topology, cost ratios). The deterministic hashing ensures full reproducibility across platforms without requiring 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 represents 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 peaking at mid-horizon. Costs are calibrated so that lost sales dominate waste in the objective, creating natural tension between overstocking (waste risk) and understocking (lost-sales penalty). Table 12 summarizes the full 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 13).

Default Reporting Format. We report all results using the *data-embedded* format (`.full.txt`) for fair comparison with NL4Opt [18], MAMO [19], and IndustryOR [7], which embed data directly in prompts. The schema-based format is additionally 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 fundamentally 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 navigate 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—but does **not** provide a complete formulation, objective function specification, full constraint set, or common-error warnings. The scaffolding cues are necessary because fully unscaffolded prompts produce near-zero accuracy on compositional retail problems for all tested models; with moderate scaffolding, the test becomes whether LLMs 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 15 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

Two data access patterns cause frequent LLM errors and deserve explicit documentation:

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 data-embedded additionally 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—matching the paradigm of NL4Opt [18], MAMO [19], 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 16 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 17 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 ϵ is family-dependent because problem structure determines the achievable precision within the 60-second time limit (Table 18). 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 11: 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 12: 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^{buy}	$\{10, 20, 15\}$	Purchasing cost per unit
c^{inv}	$\{1.0, 1.5, 1.0\}$	Holding cost per unit per period
c^{waste}	$\{2.0, 3.0, 2.0\}$	Waste cost per unit expired
c^{ls}	$\{50, 80, 40\}$	Lost-sales penalty per unit
c^{fix}	0	No fixed ordering cost
c^{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 13: 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 14: 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 15: 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 16: 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 17: 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 18: Accuracy tolerances by problem structure.

Families	**Problem Type**	**Tolerance** ϵ	**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; pack-size rounding creates inherent gaps

E.1 Verification Layer Configuration

Table 19 summarizes the hyperparameters for each verification layer.

Table 19: 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
	τ_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 employs 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)

Any check failure emits a FATAL diagnostic, triggering 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 20: Tolerance thresholds by benchmark and family.

Benchmark	**Families**	**Problem Type**	**Tolerance ϵ**
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 detailed analysis in appendix tables, we additionally 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 21: 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 [23])	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 $\rightarrow$ Formalize $\rightarrow$ Synthesize $\rightarrow$ 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/objective extraction uses the same LLM that generates the code, with temperature $= 0$. The same LLM that generates the code also performs verification; cross-model verification 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 22: Benchmark statistics.

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

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

Cited Baselines. For MAMO-ComplexLP and IndustryOR, we cite baseline Acc% from Chen et al. [10] (SIRL Table 1) for models where published results are available: DeepSeek-V3.2, Qwen3-32B, OptMATH-32B, and SIRL-32B. We run CoT and +ReLoop configurations ourselves. For additional context, the following baselines are cited in prose but not directly evaluated with ReLoop: GPT-4 (49.3% / 33.0%), DeepSeek-R1 (67.9% / 45.0%), OpenAI-o3 (51.2% / 44.0%) on MAMO-ComplexLP / IndustryOR respectively.

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 23 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 23: 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 per-family breakdown reveals several patterns. F1 (Core Operations) shows the largest absolute gain for Claude (55.0%→95.0%), indicating that ReLoop's structured generation and verification most effectively address 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. F6 (Discrete Logistics) shows an interesting asymmetry: Claude gains nothing while DeepSeek jumps from 0% to 20.0%, indicating that L1 crash recovery from integer-variable errors is the primary contributor for mid-tier models on MIP problems. The 32B models (Qwen3, OptMATH, SIRL) achieve 0% across all families, confirming that these models lack fundamental capacity for compositional retail optimization at this complexity level.

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 (Introduction) state four contributions: (i) structured generation as the primary accuracy driver on compositional problems, (ii) behavioral verification as the largest contributor on localized defects, (iii) the integrated ReLoop pipeline with diagnosis-guided repair, and (iv) the RetailOpt-190 benchmark. Each is substantiated by experimental results in Sections 5.2–5.4 and the appendices.

2. **Limitations**

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

   Answer: [Yes]

   Justification: A dedicated *Limitations* paragraph at the end of Section 6 discusses four concrete limitations: (i) CoT format incompatibility with SFT models (84 crashes, 65 regressions on OptMATH/MAMO), (ii) linear L2 overhead in tested parameters, (iii) potential failure correlation from sharing the generating LLM for constraint extraction, and (iv) three failure modes beyond scope (coefficient magnitude errors, formulation equivalence errors, unrepresented problem structures).

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 does not present formal theorems requiring proof. Property 1 (Perturbation Sensitivity) in Section 3.1 is stated as the structural intuition motivating L2 detection (its contrapositive provides the detection criterion); it is used as a design principle rather than a formally proved result. All design choices are validated empirically through ablation studies (Section 5.4) and per-family analysis (Appendix H).

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: The paper specifies the full ReLoop algorithm (Section 3, Algorithm description in Appendix E), structured generation prompts (Appendix C), L1 and L2 verification configurations (Appendices E), benchmark generation procedure (Section 4, Appendix B), evaluation metrics and tolerances (Appendix F), and full experimental setup including model versions, decoding parameters, and solver settings (Appendix G).

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 paper provides the full algorithmic specification (Section 3, Algorithm 1 in Appendix E), prompt templates and JSON schema (Appendix C), and benchmark generation procedure (Appendix B) sufficient to reproduce results independently. The ReLoop codebase and the RetailOpt-190 benchmark will be released under a permissive license upon acceptance.

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: All evaluations use greedy decoding (temperature $= 0$, max_tokens $= 8192$); ReLoop repair budget $N = 3$; L2 perturbation factors specified in Appendix E (capacity $\times 0.001$, demand $\times 100$, cost $\times 0.001$, revenue $\times 100$); L1 IIS extraction via Gurobi 11.0. Benchmark statistics, accuracy tolerances ($\epsilon \in \{10^{-4}, 10^{-2}, 10^{-6}\}$), and solver settings are listed in Appendix G.

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: [No]

   Justification: All evaluations use greedy decoding (temperature $= 0$), making outputs deterministic; we verified reproducibility on a random subset of 20 instances and observed identical outputs across repeated executions (Appendix G). All reported numbers are single-run pass@1 under deterministic conditions, so stochastic error bars are not applicable. We acknowledge that LLM API behavior can drift over time; results should be reproducible at the time of submission.

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: Foundation model evaluations (Claude Opus, DeepSeek-V3.2) use commercial API calls; ReLoop adds approximately $3\times$ base token cost (Section 5). Local 32B models (Qwen3-32B, OptMATH-32B, SIRL-32B) run via vLLM in BF16 precision; per-instance inference completes in seconds on a single high-memory GPU (Appendix G). Solver calls (Gurobi 11.0, single-threaded) complete in seconds for benchmark instances.

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics?

   Answer: [Yes]

   Justification: The research conforms with the NeurIPS Code of Ethics. No human subjects are involved; no private or personal data are used or collected; the released artifacts (ReLoop framework and RetailOpt-190) consist of synthetic optimization scenarios and a verification framework with no foreseeable harmful applications.

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 impacts*: improving the reliability of LLM-based optimization can benefit operations research deployments (supply chain, inventory, logistics), reducing the risk of silent failures that produce confident-but-wrong recommendations. *Negative impacts*: deployment trust may shift toward verified pipelines while still encoding LLM biases or blind spots; mitigation is provided by the verification mechanism itself, which is non-blocking and regression-guarded. The 90-point feasibility–correctness gap highlighted in Section 5.2 also serves as a caution against deploying unverified LLM-generated optimization code.

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. ReLoop is a verification framework that improves correctness rather than a generative model that poses misuse risks. No high-risk artifacts are released.

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 (Claude Opus, DeepSeek-V3.2, Qwen3-32B, OptMATH-32B, SIRL-32B) and external benchmarks (MAMO-ComplexLP, IndustryOR) are properly cited (Section 5.3, Appendix G) and accessed under their respective public terms of use: commercial APIs for closed-weight LLMs; HuggingFace model cards (which include license terms) for open-weight 32B models; the benchmarks under the licenses specified in their original publications. Gurobi 11.0 is used under an academic license. The released codebase will include a complete `LICENSES.md` itemizing each dependency with its license name.

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, including: 38 archetypes across 8 mechanism families (Appendix B), the modular reference MILP (Appendix A), JSON schema and data access patterns (Appendix C), instance generation procedure, and ground-truth solver settings (Appendix G). The benchmark will be released with full README, schema specification, and reference solutions.

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 does not involve crowdsourcing or human subjects.

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; IRB approval is not applicable.

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?

    Answer: [Yes]

    Justification: LLMs are central to this research as both *subjects of evaluation* (five models spanning foundation, SFT, and RL paradigms generate optimization code) and as *methodological components*: the L2 verification layer uses an LLM to extract constraint and objective references for solver-based perturbation testing (Section 3, Appendix E). All LLM configurations, prompts, and decoding parameters are fully documented in Appendices C–G.