OR-DIFFUSION: STRUCTURAL DATA SYNTHESIS AND HIERARCHICAL VERIFICATION FOR OPTIMIZATION MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models can translate problem descriptions into optimization models, but two challenges limit their utility. First, existing synthesis pipelines mainly paraphrase or perturb parameters, altering descriptions rather than constraint topology. Second, evaluation and training collapse distinct failure modes into a scalar reward, ignoring the natural priority: executability, structural fidelity, feasibility, and optimality, in that order. We introduce OR-Diffusion, which couples structural synthesis with hierarchical training. It synthesizes structurally novel instances via discrete diffusion over graph specifications, each feasible by construction and carrying a solver-certified gold optimum. A Hierarchical Verification Engine (HVE) certifies candidates along four ordered dimensions and reports the weakest failing layer. It guides supervised fine-tuning on executability, structural fidelity, and feasibility, and a preference stage on optimality. A pair enters that stage only when its rejected response passes the first three layers. On six public benchmarks, OR-Diffusion raises an 8B backbone from 57.8% to 78.2% macro optimal-solution rate, surpassing a 671B generalist (DeepSeek-V3, 73.3%) and specialized fine-tuned models (e.g., ORLM-8B, 65.5%), while also improving executability and feasibility. Ablations show that the two components are complementary, yielding a joint gain of 6.5 percentage points, and that diffusion outperforms textual synthesis by 4.2 percentage points. Weakest-layer diagnosis confirms this division of labor: fine-tuning removes most executability failures and leaves structural mismatches as the main residual, and the gated stage raises optimality without reducing feasibility.

1 INTRODUCTION

Optimization modeling (OM) converts a natural-language decision problem into a formal mathematical program solvable by an optimization engine, and is the analytical backbone of operations research (OR) in logistics, manufacturing, energy, and finance (Singh, 2012; Antoniou & Lu, 2007). It still depends heavily on human expertise: a modeler must identify decision variables, objectives, constraints, and domain assumptions, then translate them into solver-executable code. Automating this is an appealing but demanding testbed for large language models (LLMs), and recent work has explored prompt engineering (Ramamonjison et al., 2022), multi-agent reasoning (AhmadiTeshnizi et al., 2024b; Xiao et al., 2024), and domain-specific fine-tuning (Huang et al., 2025; Jiang et al., 2025).

Two challenges limit current progress. First, training data is narrow relative to the hypothesis space of valid formulations. Existing synthesis pipelines operate almost entirely at the textual level—paraphrasing statements, substituting scenarios, or perturbing parameters—and even reverse-synthesis methods such as ReSocratic (Yang et al., 2025b) generate scenarios and expressions sequentially in text space. These operations vary how a problem is described but rarely change its underlying structure: the constraint types, their interactions, and the model skeleton. Models trained on such corpora generalize across phrasings of familiar structures yet remain brittle on unseen combinations of constraints. Second, evaluation and training collapse qualitatively different failure modes into a single scalar reward. A non-executable program is uninformative; a structurally misaligned one solves the wrong problem; a faithful but infeasible one is logically inconsistent; a feasible but suboptimal one is imperfect. These form a strict priority order: executability, structural fidelity, feasibility, optimality.

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

Scalar solver rewards (Chen et al., 2025) collapse this order into one number, which records how good a candidate is but not which layer it failed, so supervision cannot be directed to that layer. When candidates are ranked by such a score, the largest gaps come from the higher layers: most contrasts separate valid code from broken code, re-teaching feasibility instead of refining optimality.

We address both with **OR-Diffusion**, which couples structural data synthesis with hierarchy-aware training. We represent each mixed-integer linear program (MILP) as an attributed bipartite graph over variable and constraint nodes with edges encoding nonzero coefficients—expressive enough to recover solver-executable code, structured enough to support discrete diffusion over formal specifications. Iterative denoising resamples graph components jointly, recombining constraint patterns and objective forms into problems whose structure lies outside the seed distribution. Generation is decoupled from description: each denoised specification is instantiated with coefficients, solved to optimality by an exact solver, and verified before it is verbalized. A witness point makes the feasible region nonempty before any solver call, so feasibility requires no rejected samples and the solver serves only to certify the gold optimum. To our knowledge, this is the first use of diffusion to synthesize modeling *tasks*, rather than to sample solutions to given problems.

To respect this priority, we introduce a Hierarchical Verification Engine (HVE) that certifies a candidate along four ordered dimensions—V_1 (executability) $\succ$ V_2 (structural fidelity) $\succ$ V_3 (feasibility) $\succ$ V_4 (optimality)—and returns the weakest layer at which it first fails. This lexicographic verdict enforces a division of labor: supervised fine-tuning (SFT) instills the first three competences, feasibility through a dedicated pool pairing violating rollouts with feasible references, while a preference stage targets optimality alone under an admission gate on V_1–V_3. The preferred response is a verified gold optimum—off-policy, and therefore a stable anchor—and the rejected one passes the first three layers but is suboptimal, so each pair is a minimal contrast that isolates optimality and never promotes an infeasible candidate. The two components are thus coupled: a preference stage can refine optimality only where the policy emits feasible-but-suboptimal code; structural synthesis supplies prompts that elicit this failure mode, and HVE detects it.

Our contributions are as follows. (i) **Diffusion-based structural synthesis of modeling tasks.** We obtain constraint combinations and model skeletons that textual paraphrasing and parameter perturbation rarely produce, feasible by construction rather than by filtering, and each carrying a verified gold reference. (ii) **Hierarchical verification with weakest-layer diagnosis and feasibility-gated training.** HVE returns the first failing layer rather than a scalar. That layer routes each training failure to its own pool, and preference pairs are admitted only when the rejected response passes the first three layers, so the comparison isolates optimality. (iii) **Empirical validation.** On six public OR modeling benchmarks, OR-Diffusion raises an 8B backbone from 57.8% to 78.2% macro optimal-solution rate, surpassing a 671B generalist (DeepSeek-V3, 73.3%) and specialized fine-tuned models (e.g., ORLM-8B, 65.5%), and improving over a seed-only control on the same backbone (71.7%). A matched 2×2 over corpus and protocol shows that the two components are strongly complementary, with a joint gain of 6.5 percentage points, while diffusion outperforms textual synthesis by 4.2 points at equal corpus size.

2 Related Work

LLMs for optimization modeling and data synthesis. Recent work has increasingly applied LLMs to translate natural-language problem descriptions into optimization formulations or solver-ready code (Ramamonjison et al., 2022; Astorga et al., 2025). Benchmarks such as NL4Opt (Ramamonjison et al., 2022) and MAMO (Huang et al., 2024) standardize evaluation, and subsequent methods improve accuracy through prompting, multi-agent reasoning, and fine-tuning (AhmadiTeshnizi et al., 2024b; Xiao et al., 2024; Huang et al., 2025). Given the scarcity of supervised instances, recent work synthesizes them predominantly at the textual level: template expansion and bidirectional back-translation (Lu et al., 2025), or text-level evolution of seed classes (Li et al., 2024). ReSocratic (Yang et al., 2025b) reverse-synthesizes instances by composing optimization scenarios and expressions, then back-translating them into problem statements. These approaches vary how a problem is *described* but leave its *structure* close to the seed distribution, where structure denotes the set of constraint types, their interactions, and the overall model skeleton. In contrast, OR-Diffusion performs diffusion over formal graph specifications, generating structurally novel yet well-posed instances with solver-verified references and feasibility by construction.

Diffusion models for optimization. Diffusion models have been applied *to* optimization, including constrained sampling over solution spaces (Kong et al., 2025) and graph-based diffusion solvers for combinatorial optimization (Sun & Yang, 2023). These works use diffusion to produce *solutions* to given problems. Our approach is orthogonal to these efforts and, to our knowledge, constitutes the first use of discrete diffusion to synthesize the *problems themselves*, specifically the training instances for description-to-model tasks. Denoising in specification space recombines constraint patterns jointly rather than left-to-right, yielding structural diversity that autoregressive or template-based generation rarely provides.

Verification and execution feedback. Execution feedback is widely used in code generation (Chen et al., 2021), and solver outcomes such as feasibility and solution quality have recently been used as reward signals for LLM-generated optimization models (Chen et al., 2025). These systems generally collapse solver feedback into a scalar outcome. HVE generalizes this to a graded, four-layer verdict evaluated in strict order. By returning the *weakest layer* at which a candidate fails, it provides supervision rich enough to drive both targeted data construction and multi-stage training. In particular, we certify *structural fidelity*—a property no solver can check on its own—as a distinct layer before feasibility, so training can separate a wrong program structure from a merely infeasible or suboptimal one.

Preference optimization and reward design. Direct Preference Optimization (DPO) (Rafailov et al., 2023) and its variants offer a lightweight alternative to RLHF. In verifiable domains, a common approach converts a scalar reward into preference pairs (Rafailov et al., 2024); in optimization modeling, solver-informed RL (Chen et al., 2025) folds syntactic validity, feasibility, and solution quality into one reward. We identify a structural limitation in such additive schemes: a single score does not record which layer a candidate failed, and ranking by it mostly contrasts a correct program with infeasible or crashing code, so the update re-teaches feasibility instead of refining optimality (Section 4). Our design avoids reward weighting altogether: feasibility is an admission gate rather than a term, so infeasible candidates never enter the comparison. Furthermore, our preference pairs are minimally contrastive: the rejected response passes V_1–V_3 and is suboptimal, so the pair contrasts optimality. Anchoring the chosen response to an off-policy verified optimum while drawing the rejected response on-policy combines the stability of off-policy supervision with the distributional relevance of on-policy sampling.

3 METHOD

Our framework has three stages (Figure 1). Conditional discrete diffusion over formal graph specifications synthesizes a structurally diverse corpus, feasible by construction and carrying verified gold references (Section 3.2). The Hierarchical Verification Engine (HVE) certifies a candidate along a lexicographic order of four error layers and reports the weakest one it fails (Section 3.3), then drives supervised fine-tuning for the first three layers (Section 3.4.1) and a feasibility-gated preference stage for the fourth (Section 3.4.2).

3.1 PROBLEM FORMULATION AND GRAPH REPRESENTATION

We study *natural-language-to-optimization modeling*, focusing on mixed-integer linear programs (MILPs). We write $\mathcal{P}$ for the algebraic formulation—objective, constraints, bounds, and the integer/continuous partition—and $\mathcal{V}(\mathcal{P})$ for its solver-executable program (e.g., Gurobi code); Appendix A gives the full MILP form. The learning problem is to estimate $f_\theta : \mathcal{Q} \to \mathcal{V}$ from descriptions to programs; OR-Diffusion improves this mapping by controlling the *structure* of the training data.

To make that structure explicit, we represent each MILP as an attributed bipartite graph $\mathcal{G} = (V, E, \mathcal{A})$ over constraint and variable nodes, following the representation standard in learning-based MILP research (Gasse et al., 2019; Fan et al., 2023). Such graphs are *manipulable*—diffusion can resample variables, constraints, and incidences directly—and *analyzable*, since graph statistics summarize the structural regimes in which a model succeeds or fails. From any valid graph we deterministically recover $\mathcal{P}$ and $\mathcal{V}(\mathcal{P})$; Appendix A gives the construction and validity conditions.

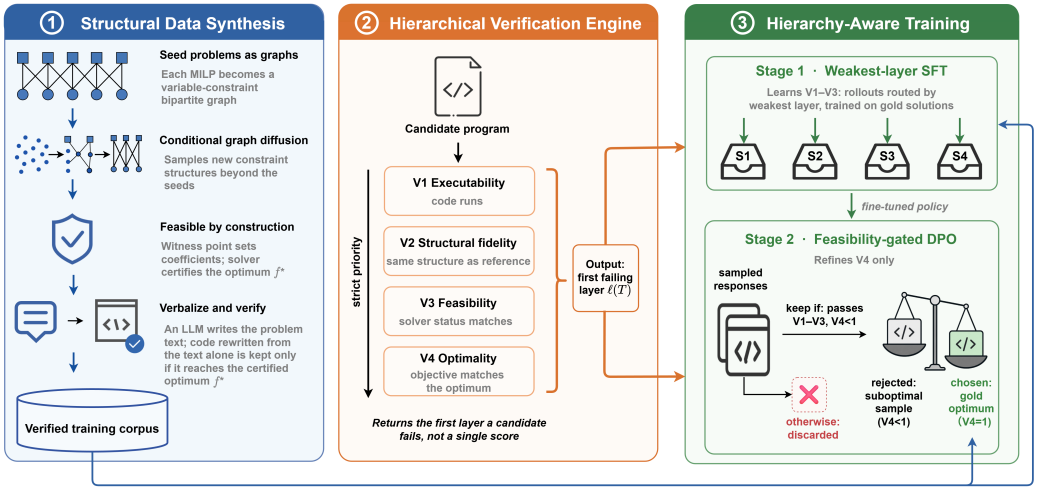


Figure 1: Overview of OR-Diffusion. **(1)** Graph diffusion synthesizes feasible, solver-verified instances. **(2)** HVE returns the first failing layer $\ell(T)$. **(3)** SFT routes failures by $\ell(T)$; gated DPO pairs the gold optimum with a response that passes V_1–V_3 but is suboptimal.

3.2 Data Generation via Conditional Graph Diffusion

We generate training instances by sampling from a conditional discrete diffusion model over graph specifications. Diffusion generates graphs probabilistically from noise; control is exerted through the application-type condition y, the sampled node-count distribution, and a post-hoc verification pipeline.

Seed graphs and conditioning. We build the initial population from the OR-Instruct seed corpus (Huang et al., 2025), extracting bipartite graphs by parsing solver-ready Python (Appendix A). Each seed graph carries a 20-dimensional application-type one-hot vector y (e.g., production planning, scheduling, routing, and portfolio optimization) assigned by an LLM annotator. This vector serves both as the conditioning signal for diffusion and as the retrieval key for few-shot examples later in the pipeline.

Conditional discrete diffusion. We train a discrete diffusion model over these graphs following DiGress (Vignac et al., 2022). The forward process corrupts node and edge types with categorical noise on a fixed schedule, carrying a clean graph $\mathcal{G}_0$ to a fully noisy $\mathcal{G}_T$ whose types are drawn from their empirical marginals. A bipartite graph Transformer parameterizes the reverse process, denoising $\mathcal{G}_t$ to $\mathcal{G}_{t-1}$ conditioned on y and trained with cross-entropy against the true node and edge types. Two inductive biases are built in. *Bipartite projection* removes, after each denoising step, any edge that would join two variable nodes or two constraint nodes, enforcing the variable–constraint structure by construction. *Core protection* marks the original nodes of augmented seed copies as uncorruptible during the forward process, so the model learns to preserve essential structural motifs while varying peripheral ones; it is used only when training the diffusion model, never when sampling.

Sampling draws y and a node count from their empirical distributions, initializes node and edge types from the marginal noise distribution at time T, and runs the reverse process to $t = 0$ under y. Because this resamples graph components *jointly* rather than left-to-right, it can recombine constraint patterns and objective forms into configurations absent from any single seed graph.

Verification and feasibility by construction. Each sampled graph passes a three-step filter before acceptance. (i) Structural checks require at least one variable and one constraint, strict bipartiteness, and no isolated nodes. (ii) Witness-point instantiation draws numerical coefficients under a fixed seed, picks a positive witness x^w within finite variable bounds and integral on integer variables, and sets each right-hand side from $a_i^\top x^w$: plus a slack $s_i > 0$ for $\leq$ rows, minus s_i for $\geq$ rows, and with no slack for equality rows. The feasible region is therefore nonempty *before any solver call*, because x^w satisfies every constraint, and the finite bounds keep the problem bounded. (iii) Gurobi then confirms

the construction and returns the gold objective $f^\star$. Step (iii) therefore certifies rather than filters: no denoised structure is ever discarded for infeasibility, so the accepted pool inherits the diversity of the sampler rather than the bias that rejection sampling would impose.

Realization into training pairs. Each verified graph becomes a pair (q, T_g) in four steps. **Code** instantiates the graph under a deterministic seed into a flat Gurobi script and solves it for $f^\star$. **Reverse** asks DeepSeek-V4-Flash (DeepSeek-AI, 2026) to turn that instance into a fluent description q that embeds every numerical value in the narrative rather than transcribing constraints as equations. **Rewrite** feeds only q back to the same model to regenerate Gurobi code. The objective is compared with $f^\star$ only when that code is feasible, and the candidate is kept only if the values match within 10^{-4}, allowing two retries. **Assemble** formats the survivors as instruction–output pairs (q, T_g), which form $\mathcal{D}_{\text{SFT}}$. The target T_g is *not* the flat template from the Code step but a freshly written solution to the same model, so SFT teaches the model to *formulate* problems rather than to memorize one template per graph. Prompt templates are in Appendix G.

3.3 Hierarchical Verification Engine

HVE maps a candidate response T for an instance with underlying model $\mathcal{P}$ to a profile $\mathbf{V}(T) = (V_1(T), V_2(T), V_3(T), V_4(T))$, where $V_1, V_2, V_3 \in \{0, 1\}$ and $V_4 \in [0, 1]$ is graded, equal to 1 only at the reference optimum. V_1 (*executability*) holds when the output runs under the solver without syntax or runtime error. V_2 (*structural fidelity*) holds when the program has the structure of $\mathcal{P}$ up to variable renaming and constraint reordering: the same variables with the same types, the same constraints with the same incidence and senses, and an objective with the same direction over the same variables. Coefficients, right-hand sides, and bounds are not compared at this layer. Solver status does not imply V_2: a program can be feasible, or even optimal, while encoding a different structure. Conversely, V_2 does not imply V_3: a program with the right structure can carry wrong numbers. The layer therefore routes the SFT pools and is an admission condition for preference pairs. V_3 (*feasibility*) and V_4 (*optimality*) are defined against the reference *solver status*, since a well-posed task may legitimately have no feasible point. Writing $\mathrm{st}(\cdot)$ for the returned status, $V_3(T) = 1$ iff $\mathrm{st}(T) = \mathrm{st}(\mathcal{P})$. If the reference solves to optimality, V_4 grades the attained objective $f(T)$ through the relative gap $g(T) = |f(T) - f^\star|/|f^\star|$ (denominator 1 if $f^\star = 0$): $V_4(T) = 1$ if $g(T) \le 10^{-4}$ and $V_4(T) = \max(0, 1 - g(T))$ otherwise. If the reference is infeasible or unbounded, reproducing that status *is* the answer, so $V_4(T) := V_3(T)$. A program that does not execute has $V_3 = V_4 = 0$, and $V_4 = 0$ whenever $V_3 = 0$; otherwise the layers are scored independently, so V_3 can hold while V_2 fails. This makes V_2, V_3, V_4 one family: agreement with the verified reference on structure, status, and value. Because every reference in Section 4 solves to optimality, V_3 there means that the candidate is feasible and bounded. Tolerances are in Appendix C.

We equip these layers with a strict lexicographic order

$$V_1 \succ V_2 \succ V_3 \succ V_4, \tag{1}$$

so that $T \succ T'$ iff T dominates T' at the first dimension on which they differ. Two consequences drive the rest of the method. First, a failure at a higher-priority layer cannot be offset by any advantage at a lower one: among candidates that agree on V_1 and V_2, one with the wrong status ($V_3 = 0$) is dominated by *every* one with the right status, regardless of objective. A scalar reward respects this order only if its weights are chosen to dominate, and even then it records one number rather than the layer that failed. Second, the informative summary of a candidate is its *weakest layer*

$$\ell(T) = \min\{\, i : V_i(T) < 1 \,\}, \tag{2}$$

with $\ell(T) = \infty$ for a perfect candidate. HVE returns $\ell(T)$ alongside the full profile; we use it both to partition rollouts into training pools and to diagnose the model.

3.4 Two-Stage Training with Weakest-Layer Routing

Both stages run sequentially on the single corpus $\mathcal{D}_{\text{SFT}}$, and the preference stage draws its rollouts from the fine-tuned policy.

3.4.1 SUPERVISED FINE-TUNING WITH WEAKEST-LAYER POOLS

SFT is responsible for the first three competences; the fourth pool, of feasible but suboptimal rollouts, is fitted to the gold as well and then refined by the preference stage. We generate rollouts from a base policy on $\mathcal{D}_{\text{SFT}}$, verify each with HVE, and route it by its weakest layer $\ell(\cdot)$ into one of four pools $\text{S}1, \ldots, \text{S}4$, where S$i$ collects rollouts whose first failure is at layer i. The training target for every prompt is its verified reference T_g; the pools determine how often each failure mode is sampled, so supervision is concentrated on the layers where the policy fails.

The feasibility pool S3 is central (removing it lowers macro Opt@1 from 78.2% to 66.7%; B3 in Section 4): it holds rollouts that are executable and structurally faithful ($V_1 = V_2 = 1$) yet return an infeasible solution ($V_3 = 0$), each paired with the feasible reference. Fine-tuning on S3 is where the model learns to respect the constraints it has correctly transcribed, so that this burden is discharged *before* the preference stage. We minimize

$$\mathcal{L}_{\text{SFT}} = -\,\mathbb{E}_{(q,T_g)\sim\mathcal{D}_{\text{SFT}}}\big[\log \pi_\theta(T_g \mid q)\big], \tag{3}$$

sampling pools with weights inversely proportional to their sizes, subject to a floor of 0.1 on S1 and S2 so that executability and structural failures keep a minimum share of the updates. Pools are filled from base-policy rollouts on training prompts. After SFT, executability failures are largely removed and structural fidelity becomes the dominant residual. This is why the next stage gates on V_2 as well as V_3, and refines only the suboptimality remaining among rollouts that pass the first three layers.

3.4.2 FEASIBILITY-GATED OPTIMALITY-REFINEMENT DPO

All preference pairs are built from the *same* corpus $\mathcal{D}_{\text{SFT}}$: for each prompt q, the verified gold T_g is the positive anchor and responses sampled from the current policy supply the negatives.

Feasibility as an admission gate, not a reward term. Reward-weighting approaches collapse feasibility and optimality into a single scalar (Chen et al., 2025), with the pathology described in Section 1. We instead enforce the order (1) as a hard *admission gate*: a response is eligible only if $V_1 = V_2 = V_3 = 1$, and infeasible responses are discarded outright rather than scored against feasible ones. Realizing the order by *exclusion* rather than weighting keeps every pair within the optimality layer.

Pair construction. The preferred response is fixed to the verified gold T_g of Section 3.2: the program Rewrite wrote from q, which executes, is feasible, and attains $f^\star$. This chosen side is deliberately *off-policy*: unlike a best-of-K sample, the gold reference is a guaranteed optimum and thus a stable, noise-free anchor, removing the variance that on-policy selection would inject. For the rejected side we sample $K = 8$ responses from π_θ, score each with HVE, and form the eligible set

$$\mathcal{F}_q = \big\{\, T \;\big|\; V_1(T) = V_2(T) = V_3(T) = 1,\ V_4(T) < 1 \,\big\}, \tag{4}$$

namely responses that pass the first three layers yet are suboptimal. We take the eligible candidate with the largest relative gap, $T_b = \arg\min_{T\in\mathcal{F}_q} V_4(T)$, retained only if $1 - V_4(T_b) > \kappa$, i.e., its objective is more than $100\kappa\%$ from $f^\star$; smaller gaps carry a weak optimality signal and are dropped. This hard-negative choice maximizes the optimality contrast within each pair. If $\mathcal{F}_q = \varnothing$—every sample fails one of the first three layers or is already optimal—no pair is formed for q. In particular, **an infeasible response is never used as** T_b: both sides of every pair are executable and feasible, and the rejected side also matches the reference structure. We then apply the standard DPO objective with preference $T_g \succ T_b$,

$$\mathcal{L}_{\text{DPO}} = -\,\mathbb{E}_{(q,T_g,T_b)\sim\mathcal{D}_{\text{DPO}}}\left[\log\sigma\left(\beta\left(\log\frac{\pi_\theta(T_g \mid q)}{\pi_{\text{ref}}(T_g \mid q)} - \log\frac{\pi_\theta(T_b \mid q)}{\pi_{\text{ref}}(T_b \mid q)}\right)\right)\right]. \tag{5}$$

A minimal contrast isolating optimality. Because T_b passes the first three layers and T_g attains the verified optimum, each pair is a *minimal contrast* isolating the optimality signal. The model is never asked to reject infeasible or crashing code, a contrast that would re-teach an earlier layer. The division of labor mirrors the hierarchy: the S3 pool teaches the model to be feasible, and this stage teaches it to be optimal *among feasible outputs*. Algorithm 1 in Appendix C gives one round in full.

Weakest-layer diagnosis. Because HVE reports the first failing layer $\ell(\cdot)$ of (2), we summarize a model by the distribution of $\ell(\cdot)$ over an evaluation set; Section 4 reports this as our primary evidence that each stage teaches its intended competence.

4 EXPERIMENTS

This section first measures overall accuracy, then tests three claims, one for each component of the method. Claims (A) and (B) are tested by ablation; claim (C) is tested by a weakest-layer diagnosis. (A) Diffusion over graph specifications should add new constraint structures rather than paraphrases of the seeds, and that structural diversity should improve optimality (Section 3.2). (B) Training in HVE's order—SFT on executability, structural fidelity, and feasibility (V_1–V_3), then feasibility-gated DPO on optimality (V_4)—should outperform a scalar reward (Sections 3.3–3.4.2). (C) Weakest-layer diagnosis should show SFT removing executability failures and leaving structural mismatches as the main residual error, with the preference stage then raising optimality without reducing feasibility (Section 3.4.2).

4.1 EXPERIMENTAL SETUP

Training corpus. We run the pipeline of Section 3.2 on OR-Instruct seeds, yielding $\mathcal{D}_{\text{SFT}} = \{(q, \mathcal{P}, T_g, f^\star)\}$ of size N=3,461 after filtering. Reverse and Rewrite use DeepSeek-V4-Flash (DeepSeek-AI, 2026) with few-shot examples matched by application type (templates in Appendix G); Rewrite scores the objective against $f^\star$ only when the candidate is feasible, and keeps it only if the two values match within relative tolerance 10^{-4}. SFT and DPO are sequential stages on this single corpus, not a closed generation loop, and all synthesis-method controls use the same N, backbone, and training budget. Table 7 (Appendix D) reports one generation round, not the aggregated mix.

Evaluation sets. *Public benchmarks* are the primary comparison: NL4Opt (Ramamonjison et al., 2022), MAMO EasyLP and ComplexLP (Huang et al., 2024), NLP4LP (AhmadiTeshnizi et al., 2024b), ComplexOR (Xiao et al., 2024), and IndustryOR (Huang et al., 2025), on the cleaned split of Xiao et al. (2025). We report Exec@1 (V_1), Feas@1 (V_3: solver status matches the reference), and Opt@1 (V_4). Structural fidelity (V_2) compares the structure of a candidate program with that of the reference program. Public benchmarks release a problem statement and a reference objective, not this gold program, so V_2 is undefined on them and is not reported. The layer remains necessary for training. A solver can accept a program that is feasible, or even optimal, while its variables, constraints, or objective differ from the reference. Weakest-layer SFT routes these mismatches by V_2. A preference pair is admitted only when the rejected response satisfies V_1=V_2=V_3=1, so the pair contrasts optimality. The internal set $\mathcal{D}_{\text{int}}$ is a held-out evaluation set: verified instances that share no graph or problem statement with the training corpus. Each instance includes the gold program structure, so all four verification layers and the weakest-layer diagnosis are defined here. This is the evidence for claim (C). Every reference in both sets solves to optimality.

Baselines. To ensure a thorough comparison, we benchmark against a comprehensive set of baselines in three categories: zero-shot generalists, agentic methods, and specialized fine-tuned LLMs. Zero-shot generalists include GPT-4o (OpenAI, 2026), DeepSeek-V3 (Liu et al., 2024), Qwen3-32B (Yang et al., 2025a), and Qwen2.5-72B-Instruct (Qwen Team, 2024). Prompting and agentic methods include CoT (Wei et al., 2022), CoE (Xiao et al., 2024), and OptiMUS-v0.3 (AhmadiTeshnizi et al., 2024a). Specialized fine-tuned LLMs include ORLM-8B (Huang et al., 2025), LLMOPT-14B (Jiang et al., 2025), and OptMATH-32B (Lu et al., 2025). Same-backbone controls, which we re-run, include Qwen3-8B zero-shot and Qwen3-8B SFT on OR-Instruct seeds only. For OptiMUS-v0.3 and CoE, we use the results reported by Zhang et al. (2026); for CoT and ORLM, we use the results reported by Xiao et al. (2025), whose cleaned split we also evaluate on. For OptMATH and LLMOPT, we report results from the original published papers.

Implementation. The backbone is Qwen3-8B. SFT uses a learning rate of 2×10^{-5} for 3 epochs and pool-balanced sampling, with weights inverse to pool size and a floor of 0.1 on S1 and S2. DPO uses β=0.1, K=8, and κ=0.1; the chosen response T_g is the off-policy gold, and the rejected response $T_b = \arg\min V_4$ is taken among gated candidates with gap above κ. We run one round on

Table 1: Opt@1 (%) on six OR benchmarks (CompOR: ComplexOR; IndOR: IndustryOR). Scores cited from original publications are marked with (*), while missing entries are denoted with (–). Bold marks the best result among fine-tuned models, including same-backbone controls.

Model	Params	NL4Opt	MAMO		NLP4LP	CompOR	IndOR	Avg.
			EasyLP	ComplexLP				
Zero-shot LLMs								
GPT-4o	Closed	73.4	95.2	45.9	87.6	38.9	66.7	68.0
DeepSeek-V3	671B	79.9	96.7	55.0	94.9	44.4	69.0	73.3
Qwen3-32B	32B	76.2	96.9	32.4	90.4	38.9	57.1	65.3
Qwen2.5-72B-Inst	72B	79.9	95.8	33.3	89.9	38.9	61.9	66.6
Prompting and Agentic Methods								
OptiMUS-v0.3	Closed	79.8	92.4	52.1	89.8	52.2	54.3	70.1
CoT	Closed	62.2	49.5	42.3	74.7	39.2	40.5	51.4
CoE	Closed	66.7	94.4	50.6	87.4	57.1	31.2	64.6
Fine-tuned LLMs								
ORLM	8B	73.8	90.4	59.5	76.4	50.0	42.9	65.5
LLMOPT (origin)	14B	80.3*	89.5*	44.1*	73.4*	35.3*	29.0*	58.6*
OptMATH (origin)	32B	95.9*	89.9*	54.1*	–	–	–	–
Same-backbone controls (Qwen3-8B)								
Zero-shot	8B	76.2	95.6	14.4	88.2	22.2	50.0	57.8
Seed SFT (OR-Instruct)	8B	92.1	94.7	64.9	93.8	44.4	40.5	71.7
OR-Diffusion	8B	**96.3**	**97.4**	**65.8**	**97.2**	**55.6**	**57.1**	**78.2**

the epoch-3 SFT rollouts. At inference the model sees only q. Hardware is 8 A100 GPUs; further hyperparameters are in Appendix C.

4.2 Main Results on Public Benchmarks

We report the main results in Table 1 as Opt@1, the greedy Pass@1 rate against gold $f^\star$ at tolerance 10^{-4}. OR-Diffusion sets the best macro-average on these benchmarks, 78.2%, outperforming both vastly larger generalist and agentic models and specialized LLMs. As an 8B model, it surpasses DeepSeek-V3 (671B, 73.3%) and Qwen2.5-72B-Instruct (72B, 66.6%), and even the agentic method OptiMUS-v0.3 (70.1%). Graph-specification diffusion, together with weakest-layer SFT and feasibility-gated DPO, is what enables this compact model to reach more accurate OR modeling than its far larger counterparts.

4.3 Stage-wise Contribution and Weakest-Layer Diagnosis

Table 2 compares three checkpoints of the same backbone. On the six public benchmarks, weakest-layer SFT raises Exec@1, Feas@1, and Opt@1 together, from 81.4%, 79.0%, and 57.8% to 94.0%, 89.1%, and 72.5%. Feasibility-gated DPO then leaves executability nearly unchanged, at 94.5%, and raises Opt@1 further to 78.2%. On $\mathcal{D}_{\text{int}}$, where each layer is scored separately, SFT accounts for most of the executability gain, and DPO improves structural fidelity and optimality without reducing feasibility. The per-layer rates are in the table.

Claim (C) states that SFT should remove executability failures and leave structural fidelity as the main residual error, and that feasibility-gated DPO should then improve optimality without reducing feasibility. Figure 2 (Appendix D) evaluates this claim on $\mathcal{D}_{\text{int}}$, the only split with gold structure, and therefore the split on which the V_2 used to route training and to admit preference pairs can be

Table 2: Stage contribution (%). Public columns are macro-averages; the SFT row is the same checkpoint as B1 in Table 3. Internal $p_i = \Pr(V_i{=}1)$ on $\mathcal{D}_{\text{int}}$ ($n{=}325$).

Checkpoint	Public (macro-avg)			Internal $\mathcal{D}_{\text{int}}$			
	Exec@1	Feas@1	Opt@1	p_1	p_2	p_3	p_4
Base Qwen3-8B	81.4	79.0	57.8	84.3	51.1	79.7	68.6
Weakest-layer SFT	94.0	89.1	72.5	97.5	60.0	91.7	81.8
SFT + gated DPO	94.5	92.3	78.2	97.8	64.3	93.8	85.8

scored. After SFT, the fraction of outputs whose first failure is executability falls from 15.7% to 2.5%, and the remaining errors are mostly structural mismatches. Gated DPO then increases the fraction that passes all four layers from 60.0% to 64.3%, and the executability-failure rate does not rise. An output can mismatch the reference structure and still be feasible, or even optimal, and DPO improves optimality without reducing feasibility. Per-layer rates are given in Appendix D.

4.4 Ablations

Each ablation changes one component at fixed backbone, corpus size N, budget, and evaluation sets (Table 3). On the data side, (A1) runs the same gated-DPO recipe on the OR-Instruct seeds, and (A2) replaces diffusion with paraphrase and scenario-swap synthesis at the same N (SFT only). On the protocol side, (B1) drops the preference stage; (B2) drops weakest-layer routing, training with plain cross-entropy on (q, T_g); (B3) removes the S3 pool of executable-and-faithful-but-infeasible pairs; and (B4) replaces the admission gate with a weighted sum of V_3 and V_4, keeping infeasible rollouts eligible for the rejected side.

Table 3: Ablations (Opt@1 %). One component changed per row at fixed N, backbone, and budget; same benchmarks and macro-average as Table 1.

Variant	NL4Opt	MAMO		NLP4LP	CompOR	IndOR	Avg.
		EasyLP	ComplexLP				
Full OR-Diffusion	**96.3**	**97.4**	**65.8**	**97.2**	**55.6**	**57.1**	**78.2**
A1 w/o Diffusion	88.3	95.4	63.1	93.3	50.0	54.8	74.1
A2 Textual synthesis	86.4	95.8	63.1	87.6	38.9	38.1	68.3
B1 SFT only	90.7	96.1	60.4	90.4	50.0	47.6	72.5
B2 w/o weakest-layer routing	84.6	89.0	36.9	89.9	27.8	45.2	62.2
B3 w/o S3 pool	88.8	93.6	50.5	88.8	33.3	45.2	66.7
B4 w/o feasibility gate	89.3	95.6	56.8	92.1	44.4	40.5	69.8

Where the gains come from. Claim (A) is that diffusion yields structural diversity, not paraphrases, and that this diversity drives optimality. In the matched comparison of Table 5, gated DPO adds 5.7 points on the diffusion corpus and 2.4 points on the seeds, and the full system gains 6.5 points over seed-only SFT. Against textual synthesis trained with the same SFT protocol, diffusion leads by 4.2 macro points, and textual synthesis falls below the unaugmented seeds (68.3% versus 71.7%). The diversity is structural: under the signature metric of Luo et al. (2026), 42.9% of the diffusion constraint signatures are unseen in the seeds, against 2.9% for paraphrase. Per-benchmark breakdowns are in Table 3, the 2×2 grid is in Table 5, and the signature comparison is in Appendices D and F.

Protocol components. Claim (B) is that the lexicographic protocol outperforms scalar rewards. Table 3 shows that both parts of the protocol are necessary. Without weakest-layer pools, the diffusion corpus scores 62.2% (B2), below textual synthesis at 68.3% (A2), so the pools, rather than the corpus alone, carry the SFT gain. Dropping the feasibility pool (B3, 66.7%) or replacing the gate with a scalar score (B4, 69.8%) leaves the full pipeline below stopping after SFT (B1, 72.5%). Scalar pairs mostly contrast a gold program with infeasible or crashing code, so the update re-teaches feasibility; the split is in Table 6. Gated pairs are built where gold structure is known: the rejected response matches the reference on executability, structure, and feasibility, so the pair contrasts optimality. The +5.7 point gain of the preference stage therefore comes from the content of its pairs, whereas scalar ranking performs worse (B4, 69.8%).

5 Conclusion

We presented OR-Diffusion, a framework that synthesizes structurally novel modeling tasks via discrete diffusion over formal graph specifications and trains LLMs under a strict hierarchy: executability, structural fidelity, feasibility, and optimality. Weakest-layer supervised fine-tuning instills the first three competences, while feasibility-gated preference optimization refines the fourth. The two components reinforce each other: structural synthesis supplies prompts that elicit the feasible-but-suboptimal code the gated stage refines. Respecting the natural priority among failure modes is what renders synthetic data useful, since only a layered verdict can identify which synthesized instances provide signal the policy has not already internalized.

AI USE STATEMENT

Generative AI appears in this work in two roles. As a component of the method, large language models annotate seed graphs with application types and turn verified graphs into problem statements and reference solutions (Section 3.2). These uses are part of the method, and every resulting artifact enters the corpus only after solver-based verification. As a writing aid, we used generative AI tools to polish the manuscript text and LaTeX. All reported numbers come from the solvers and the training pipeline in Section 4. We have reviewed all AI-assisted writing, and we take responsibility for the final content of this work, including text, claims, and any artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies natural-language optimization modeling on publicly available benchmarks (NL4Opt, MAMO, NLP4LP, ComplexOR, and IndustryOR). It does not involve human subjects, private user data, or the release of sensitive datasets. We do not foresee significant dual-use or fairness risks beyond those already associated with general-purpose code-generating language models. Upon acceptance, we will release code and data to support transparent evaluation.

REPRODUCIBILITY STATEMENT

We provide complete experimental settings in Section 4 and Appendix C, including the diffusion data-generation pipeline, the HVE layers, weakest-layer supervised fine-tuning, the feasibility-gated preference procedure (Algorithm 1), baselines, and hyperparameters. The HVE order and the two training stages are defined in Section 3. Appendix B records the benchmark splits, and Appendix G lists the full prompt templates. Upon acceptance, we will release source code, data, configuration files, and the scripts used to produce the reported tables.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, Herman Brunborg, Shayan Talaei, Connor Lawless, and Madeleine Udell. OptiMUS-0.3: Using large language models to model and solve optimization problems at scale. *arXiv preprint arXiv:2407.19633*, 2024a.

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. *arXiv preprint arXiv:2402.10172*, 2024b.

Andreas Antoniou and Wu-Sheng Lu. *Practical Optimization: Algorithms and Engineering Applications*. Springer, 2007.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 1864–1886. PMLR, 2025. URL https://proceedings.mlr.press/v267/astorga25a.html.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed RL: Grounding large language models for authentic optimization modeling. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/98555b92e5dedbdad5921049281995bc-Abstract-Conference.html.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Zhenan Fan, Xinglu Wang, Oleksandr Yakovenko, Abdullah Ali Sivas, Owen Ren, Yong Zhang, and Zirui Zhou. Smart initial basis selection for linear programs. In *International Conference on Machine Learning*, pp. 9650–9664. PMLR, 2023.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2019.

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.

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

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. LLMOPT: Learning to define and solve general optimization problems from scratch. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=9OMvtboTJg.

Lingkai Kong, Yuanqi Du, Wenhao Mu, Kirill Neklyudov, Valentin De Bortoli, Dongxia Wu, Haorui Wang, Aaron M. Ferber, Yi-An Ma, Carla P. Gomes, and Chao Zhang. Diffusion models as constrained samplers for optimization with unknown constraints. In *Proceedings of the 28th International Conference on Artificial Intelligence and Statistics*, volume 258 of *Proceedings of Machine Learning Research*, pp. 4582–4590. PMLR, 2025. URL https://proceedings.mlr.press/v258/kong25b.html.

Sirui Li, Janardhan Kulkarni, Ishai Menache, Cathy Wu, and Beibin Li. Towards foundation models for mixed integer linear programming. *arXiv preprint arXiv:2410.08288*, 2024.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

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*, volume 267 of *Proceedings of Machine Learning Research*, pp. 40769–40802. PMLR, 2025. URL https://proceedings.mlr.press/v267/lu25o.html.

Yidong Luo, Chenguang Wang, Dong Li, and Tianshu Yu. Are your generated instances truly useful? GenBench-MILP: A benchmark suite for MILP instance generation, 2026. URL https://arxiv.org/abs/2505.24779.

OpenAI. Chatgpt (GPT-4o version) [large language model], 2026. URL https://chat.openai.com. Accessed: 2026-04-29.

Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, 2023.

Rafael Rafailov, Joey Hejna, Ryan Park, and Chelsea Finn. From r to q^*: Your language model is secretly a Q-function. In *Conference on Language Modeling*, 2024.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pp. 189–203. PMLR, 2022. URL https://proceedings.mlr.press/v220/ramamonjison23a.html.

Ajay Singh. An overview of the optimization modelling applications. *Journal of Hydrology*, 466-467: 167–182, 2012. ISSN 0022-1694. doi: https://doi.org/10.1016/j.jhydrol.2012.08.004. URL https://www.sciencedirect.com/science/article/pii/S0022169412006683.

Zhiqing Sun and Yiming Yang. DIFUSCO: Graph-based diffusion solvers for combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/0ba520d93c3df592c83a611961314c98-Abstract-Conference.html.

Clement Vignac, Igor Krawczuk, Antoine Siraudin, Bohan Wang, Volkan Cevher, and Pascal Frossard. Digress: Discrete denoising diffusion for graph generation. *arXiv preprint arXiv:2209.14734*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

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 *International Conference on Learning Representations*, 2024.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. OptiBench meets ReSocratic: Measure and improve LLMs for optimization modeling. In *International Conference on Learning Representations*, 2025b.

Yansen Zhang, Qingcan Kang, Yujie Chen, Yufei Wang, Xiongwei Han, Tao Zhong, Mingxuan Yuan, and Chen Ma. SAC-Opt: Semantic anchors for iterative correction in optimization modeling, 2026. URL https://arxiv.org/abs/2510.05115.

A Formal Setting and Graph Representation

This appendix states in full the formalism summarized in Section 3.1.

Mixed-integer linear programs. Given a problem description $q \in \mathcal{Q}$, where $\mathcal{Q}$ is the space of natural-language descriptions, the task is to produce a correct optimization model together with its solver-executable implementation. We restrict attention to MILPs, written as

$$\begin{aligned} \min / \max \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \bowtie \mathbf{b}, \\ & \mathbf{l} \le \mathbf{x} \le \mathbf{u}, \\ & x_i \in \mathbb{Z} \text{ for } i \in \mathcal{I}, \quad x_j \in \mathbb{R} \text{ for } j \in \mathcal{R}, \end{aligned}$$

where $\mathbf{x} = (x_1, \ldots, x_n)$ is the decision vector, $\mathbf{c}$ the objective coefficient vector, $\mathbf{A}$ the constraint matrix, $\mathbf{b}$ the right-hand-side vector, and $\bowtie \in \{\le, =, \ge\}^m$ the vector of constraint senses. Variable bounds are $\mathbf{l}$ and $\mathbf{u}$, and $\mathcal{I}$, $\mathcal{R}$ index the integer and continuous variables. We denote the algebraic formulation by $\mathcal{P} = (\min / \max, \mathbf{c}, \mathbf{A}, \mathbf{b}, \bowtie, \mathbf{l}, \mathbf{u}, \mathcal{I}, \mathcal{R})$ and its solver-executable program by $\mathcal{V}(\mathcal{P})$.

Attributed bipartite graph. We represent each MILP as $\mathcal{G} = (V, E, \mathcal{A})$ following the variable–constraint graph of Gasse et al. (2019) and Fan et al. (2023). *Nodes*: $V = V_{\text{con}} \cup V_{\text{var}}$, where $V_{\text{con}} = \{s_1, \ldots, s_m, s_{\text{obj}}\}$ contains one node per constraint plus a dedicated objective node, and $V_{\text{var}} = \{x_1, \ldots, x_n\}$ one node per decision variable. *Edges*: $E \subseteq V_{\text{con}} \times V_{\text{var}}$, with an edge (s_i, x_j) present when x_j appears in constraint s_i or in the objective with a nonzero coefficient; the edge weight stores that coefficient. *Attributes*: a constraint node stores its type, right-hand side, and relational sense; a variable node stores its lower bound, upper bound, and domain. A graph is *valid* when it has exactly one objective node, complete variable metadata, and no isolated components. From any valid $\mathcal{G}$ we deterministically recover $\mathcal{P}$ and $\mathcal{V}(\mathcal{P})$.

Note that coefficients and right-hand sides are not part of the discrete topology that the diffusion model manipulates: the model samples node types, edge types, and incidences, and numerical values are instantiated afterwards by the witness-point procedure of Section 3.2.

Extracting seed graphs from code. The seed population of Section 3.2 is obtained by parsing the solver-ready Python of the OR-Instruct corpus. We read `addVar`/`addVars` declarations to recover variable nodes and their domains, read `addConstr` calls to recover constraint nodes, and build the edge matrix by tracing which variables appear in each constraint's expression. Two-dimensional variable families such as $x_{i,j}$ are preserved as a grid rather than flattened into independent variables, so the extracted graph retains the original model structure. Variables are typed as continuous, binary, or integer, and constraints are uniformly typed. The result is $\mathcal{D}_0 = \{(q_i, \mathcal{G}_i, c_i)\}_{i=1}^{N_0}$, pairing each seed graph with its natural-language description q_i and its executable program $c_i = \mathcal{V}(\mathcal{P}_i)$.

B Benchmark Details

A large number of datasets and dataset-construction methods have been introduced to improve the capability of LLMs in solving operations-research (OR) problems, and these datasets have become increasingly complex and realistic (AhmadiTeshnizi et al., 2024b). Xiao et al. (2025) provide a recent and comprehensive survey of benchmark datasets for natural-language-described optimization problems. Their study highlights several recurring issues in existing benchmarks, such as logical inconsistencies, ambiguous problem statements, and incorrect ground-truth answers, and further rectifies part of these errors. Unless otherwise noted, we follow their cleaned evaluation split. Table 4 summarizes the instance counts used in our experiments.

NL4Opt. The NL4Opt dataset, introduced by Ramamonjison et al. (2022) for the NeurIPS 2022 NL4Opt competition, is a benchmark containing approximately 1,100 annotated linear programming problems. It is designed to study the mapping from natural-language descriptions to formal optimization models. The original release comprises 713 training instances, 99 validation instances, and 289 test instances. Due to noise and errors in the original dataset, we follow Xiao et al. (2025) and use their cleaned version instead. Specifically, our experiments are conducted on 214 cleaned instances.

MAMO. The MAMO dataset was introduced by Huang et al. (2024) as a benchmark for evaluating large language models on mathematical modeling tasks. It focuses on LP and MILP problems, with an emphasis on the formulation process rather than the solving step itself. The original dataset is divided into two subsets: EasyLP with 652 instances and ComplexLP with 211 instances. In our work, rather than using the original release directly, we adopt the cleaned version provided by Xiao et al. (2025). After data cleaning, we use 545 easy instances and 111 complex instances in our experiments.

NLP4LP. The NLP4LP benchmark was introduced by AhmadiTeshnizi et al. (2024b) for evaluating the ability of large language models to translate natural-language operations-research problems into solver-ready code and mathematical formulations. It consists of 269 human-authored LP and MILP problems covering classical operations-research domains such as scheduling, knapsack allocation, and production planning. In our work, rather than using the original release directly, we adopt the cleaned version provided by Xiao et al. (2025). After data cleaning, we use 178 instances in our experiments.

ComplexOR. The ComplexOR dataset was introduced by Xiao et al. (2024) as a benchmark for evaluating large language models on challenging operations-research modeling tasks. It consists of complex optimization problems drawn from advanced OR case studies and difficult problem classes, such as multi-item lot-sizing, intricate scheduling, and supply-chain design. These instances are primarily large-scale or conceptually complex mixed-integer linear programming problems that require multi-step reasoning. In our work, we use 18 instances from this dataset in our experiments.

IndustryOR. The IndustryOR dataset was introduced by Huang et al. (2025) as a benchmark for evaluating large language models on real-world operations-research problems. It consists of 100 optimization instances drawn from practical domains such as manufacturing, logistics, finance, and energy. The dataset covers multiple problem categories, including linear, integer, mixed-integer, nonlinear, and other types. In our work, rather than using the original release directly, we adopt the cleaned version provided by Xiao et al. (2025). After data cleaning, we use 42 instances in our experiments.

Evaluation protocol. On the six public sets above (counting MAMO EasyLP and ComplexLP separately), the primary metric is Opt@1: relative error $\leq 10^{-4}$ against the reference objective $f^\star$. Exec@1 (V_1) and Feas@1 (V_3: the candidate's solver status matches the reference status) are reported alongside it. Structural fidelity (V_2) compares the structure of a candidate program with that of the reference program. Public benchmarks release a problem statement and a reference objective, not this gold program, so V_2 is undefined on them. The internal set $\mathcal{D}_{\text{int}}$ (n=325) is held out from $\mathcal{D}_{\text{SFT}}$ with no graph or statement overlap, and each instance includes the gold program, so the full profile (p_1, p_2, p_3, p_4) and the weakest-layer distribution are reported there. Training still uses V_2 to route structural mismatches and to admit preference pairs.

Table 4: Public benchmark sizes. Cleaned counts follow Xiao et al. (2025), except ComplexOR, which we use as released. NL4Opt originally has ∼1,100 annotated instances (713/99/289 train/val/test); we evaluate on the cleaned test split.

Benchmark	Original	Cleaned (used)	Source
NL4Opt	289 (test)	214	Ramamonjison et al. (2022)
MAMO EasyLP	652	545	Huang et al. (2024)
MAMO ComplexLP	211	111	Huang et al. (2024)
NLP4LP	269	178	AhmadiTeshnizi et al. (2024b)
ComplexOR	18	18	Xiao et al. (2024)
IndustryOR	100	42	Huang et al. (2025)

C Additional Implementation Details

Diffusion generator. DiGress-style discrete graph diffusion on variable–constraint bipartite graphs extracted from OR-Instruct. Conditioning is the 20-dimensional application-type vector y. Each denoising step applies bipartite projection; core protection is used only when training on augmented

seed copies. Coefficients and right-hand sides are *not* in the graph: they are instantiated after sampling via a witness point, then checked by Gurobi.

HVE. V_1: compiles and runs. V_2: the program has the structure of $\mathcal{P}$ up to renaming and constraint reordering (variables and their types, constraint incidence and senses, objective direction and support); coefficients, right-hand sides, and bounds are not compared. Public benchmarks do not release that gold program, so V_2 is scored on $\mathcal{D}_{\text{int}}$; training still uses it to route structural mismatches and to admit preference pairs. V_3: the candidate's solver status matches the reference status, $\mathrm{st}(T) = \mathrm{st}(\mathcal{P})$, over OPTIMAL, INFEASIBLE, and UNBOUNDED. V_4: when $\mathrm{st}(\mathcal{P})$ is OPTIMAL, the relative gap $g(T) = |f(T) - f^\star|/|f^\star|$ (denominator 1 when $f^\star = 0$) gives $V_4 = 1$ if $g(T) \le 10^{-4}$ and $V_4 = \max(0, 1 - g(T))$ otherwise (Section 3.3); when the reference is INFEASIBLE or UNBOUNDED, reproducing the status is itself the correct answer, so $V_4 := V_3$ and no objective or solver value is compared. Every reference in $\mathcal{D}_{\text{SFT}}$, in $\mathcal{D}_{\text{int}}$, and in the six public benchmarks solves to optimality, so on those splits V_3 means that the candidate is feasible and bounded; Appendix E evaluates the status branch on a synthesized slice whose references are provably infeasible. Rewrite applies the same tolerance only to feasible candidates; public Opt@1 and internal p_4 use it as well.

Training. Backbone Qwen3-8B. SFT: learning rate 2×10^{-5}, 3 epochs, inverse-size pool weights, floor 0.1 on S1 and S2. DPO: β=0.1, K=8, hard-negative $T_b = \arg\min_{T \in \mathcal{F}_q} V_4(T)$ with gap threshold κ, chosen side fixed to off-policy T_g. One gated DPO round on pairs built from $\mathcal{D}_{\text{SFT}}$ (no HVE-directed generator loop). Hardware: 8 A100 GPUs.

Preference-pair construction. Algorithm 1 states one round of the feasibility-gated preference stage of Section 3.4.2. Every written pair is an L4 contrast: the rejected response is executable, structurally faithful, and feasible but suboptimal, and the chosen side is the verified gold. A prompt contributes no pair when all K samples are already optimal, when none passes the first three layers, or when the best available gap falls below κ.

Algorithm 1 Feasibility-Gated Optimality-Refinement DPO (one round)

Require: policy π_θ, reference π_{ref}, dataset $\mathcal{D}_{\text{SFT}}$, samples K, margin κ

$\mathcal{D}_{\text{DPO}} \leftarrow \varnothing$
for each instance $(q, \mathcal{P}, T_g) \in \mathcal{D}_{\text{SFT}}$ **do**
  sample $\{T^{(1)}, \ldots, T^{(K)}\} \sim \pi_\theta(\cdot \mid q)$; score each with HVE
  $\mathcal{F}_q \leftarrow \{\, T : V_1{=}V_2{=}V_3{=}1,\ V_4 < 1 \,\}$ ▷ feasibility- and fidelity-gated
  if $\mathcal{F}_q \neq \varnothing$ **and** $\max_{T \in \mathcal{F}_q}(1 - V_4(T)) > \kappa$ **then**
    $T_b \leftarrow \arg\min_{T \in \mathcal{F}_q} V_4(T)$; $\mathcal{D}_{\text{DPO}} \leftarrow \mathcal{D}_{\text{DPO}} \cup \{(q, T_g, T_b)\}$
  else
    skip prompt q this round
  end if
end for
update π_θ on $\mathcal{D}_{\text{DPO}}$ with objective (5)

D Additional Experimental Results

This appendix expands the diagnostics referenced in Section 4. Figure 2 and Tables 6–10 support the three experimental claims: (A) diffusion changes topology rather than wording; (B) feasibility-gated DPO isolates optimality, whereas a scalar mix of V_3 and V_4 does not; (C) residual errors migrate from executability toward structural mismatch, while DPO raises optimality without reducing feasibility.

The corpus-by-protocol 2×2. Table 5 lays out the four cells discussed in Section 4. The two seed cells are the seed-SFT control of Table 1 and row A1 of Table 3, and the two diffusion cells are row B1 and the full system. Gated DPO adds 5.7 points on the diffusion corpus and 2.4 points on the seeds, and the full system gains 6.5 points over seed-only SFT. Gated DPO can form pairs only from feasible-but-suboptimal rollouts, so the larger gain on the diffusion corpus is consistent with that corpus eliciting more of them.

Table 5: Matched 2×2 over corpus and protocol (macro Opt@1, %) at fixed backbone, corpus size N, budget, and evaluation sets. Reading across rows gives the value of the preference stage (+2.4 points on seeds, +5.7 points on diffusion). The cells are, respectively, the seed-SFT control of Table 1, row A1 and row B1 of Table 3, and the full system.

Corpus	Weakest-layer SFT	+ gated DPO
Seed (OR-Instruct)	71.7%	74.1%
Diffusion	72.5%	**78.2%**

Weakest-layer diagnosis. Figure 2 complements Table 2 on the internal set $\mathcal{D}_{\text{int}}$ (n=325), the only split with gold structure and therefore a full HVE profile. The left panel is the lexicographic mass $P(\ell{=}i)$ at Base, SFT, and gated DPO; the right panel is the four marginal rates $p_i = \Pr(V_i{=}1)$. Public benchmarks cannot support this plot: they lack gold $\mathcal{P}$, so V_2 is undefined and the weakest layer cannot be localized beyond $V_1/V_3/V_4$. The layer is still used in training, to route structural mismatches and to admit preference pairs; $\mathcal{D}_{\text{int}}$ is where that signal is measured.

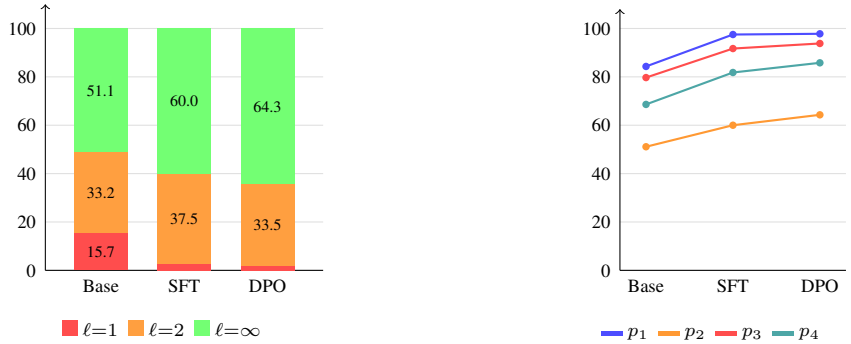


Figure 2: Weakest-layer diagnosis on $\mathcal{D}_{\text{int}}$ (n=325). Left: lexicographic mass (%). Right: marginal $p_i = \Pr(V_i{=}1)$.

The left panel shows the intended division of labor. Base Qwen3-8B fails at executability on 15.7% of instances ($\ell{=}1$; 51/325) and at structural fidelity on 33.2% ($\ell{=}2$; 108/325), with 51.1% already perfect ($\ell{=}\infty$; 166/325). Weakest-layer SFT reduces $\ell{=}1$ to 2.5% (8/325) while $\ell{=}2$ remains the dominant residual (37.5%; 122/325). Gated DPO then raises $\ell{=}\infty$ from 60.0% to 64.3% (from 195 to 209), with the executability-failure rate at 2.2% (7/325). Residual error after SFT on $\mathcal{D}_{\text{int}}$ is a structural-fidelity problem.

The right panel reports the marginal rates, a different summary from the weakest-layer mass. SFT lifts the tuple (p_1, p_2, p_3, p_4) from (84.3%, 51.1%, 79.7%, 68.6%) to (97.5%, 60.0%, 91.7%, 81.8%); gated DPO further to (97.8%, 64.3%, 93.8%, 85.8%). Throughout, p_3 and p_4 exceed p_2: many programs that fail structural matching are still solver-feasible and even optimal. DPO improves p_4 while p_3 rises from 91.7% to 93.8%, matching claim (C) that the preference stage raises optimality without reducing feasibility.

Scalar vs. gated preference pairs. Table 6 records pair composition for ablation B4 versus gated DPO on the same epoch-3 SFT rollouts. B4 ranks candidates by the scalar $s{=}0.3V_3{+}0.7V_4$ and therefore admits infeasible code onto the rejected side; gated DPO uses the admission set $\mathcal{F}_q$ of Section 3.4.2 and writes only L4 contrasts. Both methods are scored on the same prompts and $K{=}8$ samples, so the difference is the pairing rule, not the rollout pool.

Table 6: Pair composition on the same epoch-3 SFT rollouts. Scalar DPO ranks by $s{=}0.3V_3{+}0.7V_4$ (449 pairs); gated DPO uses the feasibility admission gate (378 L4 pairs). Without the gate, most pairs are gold vs. infeasible code rather than optimality contrasts.

Preference-pair statistic	Scalar DPO (B4)	Gated DPO
Chosen infeasible, rejected feasible (%)	0.0	0.0
Rejected infeasible ($V_3{=}0$) (%)	74.8	0.0
Both sides feasible (%)	25.2	100.0

Because $V_4{=}0$ whenever $V_3{=}0$, scalar ranking never promotes an infeasible *chosen* response: both columns in the first row are zero. The pathology is on the rejected side. Under B4, 74.8% of pairs contrast gold against an infeasible or crashing sample, so the DPO gradient is mostly a feasibility penalty rather than an optimality contrast; only 25.2% of the 449 pairs have a feasible rejected side, and B4 does not check their structure. The gate writes fewer pairs (378) but every one of them is an L4 contrast: the rejected response is executable, structurally matched, and feasible, and fails only V_4. This is why B4's public Opt@1 (69.8%) sits below SFT-only (72.5%): most of its preference updates re-teach V_3 instead of refining V_4.

Synthesis funnel and DPO admission gate. Table 7 reports one generation round from diffusion samples to rewrite-verified gold. It is *not* the aggregated SFT mix used in Tables 1–2 ($N{=}3{,}461$); that corpus pools later rounds. Table 8 then diagnoses the feasibility admission gate on $\mathcal{D}_{\text{SFT}}$ under the SFT epoch-3 policy ($K{=}8$, $\kappa{=}0.1$).

Table 7: One generation round: funnel from diffusion samples to rewrite-verified gold. Of 4,448 feasible graphs, 3,000 are sent to Reverse under a type quota. The SFT corpus in Tables 1–2 has size $N{=}3{,}461$.

Stage	Count	Retention (%)
Diffusion samples	6,000	100.0
Structural checks	4,448	74.1
Witness-point instantiation	4,448	74.1
Gurobi feasible / bounded	4,448	74.1
Reverse (q)	3,000	50.0
Rewrite (T_g, incl. retry)	1,910	31.8
Assembled gold	1,910	31.8

Gold labels are the verified feasible subset, not every sampled graph. Structural checks drop isolated or single-partition graphs (from $6{,}000$ to $4{,}448$; 74.1% retained). Witness-point instantiation and Gurobi then remove no further mass: coefficients and right-hand sides are chosen so that a positive witness x^w is feasible by construction (Section 3.2), and the solver merely confirms that construction. Of those $4{,}448$ feasible graphs, $3{,}000$ are sent to Reverse under an application-type quota (50.0% of the original $6{,}000$). Rewrite is the statement-to-code filter. The objective is compared with $f^\star$ only when the generated program is feasible; infeasible programs are discarded without that comparison, and a feasible program is kept only if its objective matches $f^\star$ within 10^{-4}, with up to two retries. This yields $1{,}910$ assembled gold pairs (31.8%).

Table 8: Feasibility-gate diagnosis on $\mathcal{D}_{\text{SFT}}$ under the SFT epoch-3 policy ($K{=}8$, $\kappa{=}0.1$). Empty $\mathcal{F}_q$ is almost entirely already-optimal rollouts, not rollouts that fail V_1–V_3. All written pairs are L4.

Gate statistic (SFT e3, $K{=}8$)	Value
Training prompts	3,461
On-policy already $V_4{=}1$	91.4%
Prompts with $\mathcal{F}_q \neq \varnothing$	11.0%
Skipped: already optimal	87.7%
Skipped: no sample passes V_1–V_3	1.2%
Skipped: gap $<\kappa$	0.1%
L4 pairs written	378
Infeasible chosen	0.0%
Mean $\lvert\mathcal{F}_q\rvert/K$ among nonempty	0.26
Mean $V_4(T_b)$	0.08

The gate writes pairs only where a sample that passes V_1–V_3 is still suboptimal. On $3{,}461$ training prompts, 91.4% of on-policy rollouts already satisfy $V_4{=}1$; empty $\mathcal{F}_q$ is almost entirely this already-optimal mass (87.7% of prompts skipped), with 1.2% skipped because no sample passes V_1–V_3 and 0.1% for a gap below κ. A nonempty eligible set appears on 11.0% of prompts; after dropping the 0.1% whose gap is below κ, the gate writes 378 L4 pairs, none with an infeasible chosen side. Among nonempty $\mathcal{F}_q$, the mean occupancy is $0.26K$ (about two of eight samples) and the hard negative is far from optimal: mean $V_4(T_b){=}0.08$, i.e., a mean relative gap above 90%. DPO therefore trains on residual V_4 errors among feasible code, the minimal contrast of Section 3.4.2.

Structural diversity and text vs. structure. Table 9 compares structural novelty against OR-Instruct seeds. Table 10 separates wording from topology, which is the distinction from ReSocratic-style sequential text reverse-synthesis (Section 1): diffusion is allowed to keep the OR narrative register while moving the constraint skeleton.

Table 9: Structural diversity against OR-Instruct seeds. Struct. sim. is $100(1 - \overline{\mathrm{KS}})$ over the 11 GenBench-MILP bipartite features, 50 draws of 212-instance subsets.

Corpus vs. OR-Instruct seeds	Struct. sim. ↓	Unseen signature (%) ↑
Diffusion (ours)	71.68 ± 0.72	42.9
Paraphrase / scenario swap	95.23 ± 0.83	2.9
Coefficient perturbation	92.25 ± 0.64	4.6

We report structural similarity from the 11 bipartite-graph features of GenBench-MILP (Luo et al., 2026), aggregated as $100\big(1 - \overline{\mathrm{KS}}\big)$ (lower is more diverse) of each corpus against OR-Instruct seeds, plus the fraction of constraint-combination signatures—variable-type multiset and constraint-degree sequence—unseen in the seed set. Protocol: generate a large pool, draw 212-instance subsets, repeat 50 times, report mean±std. Diffusion sits at 71.68 ± 0.72 similarity with 42.9% unseen signatures. Paraphrase / scenario swap and coefficient perturbation remain near the seed manifold (95.23 and 92.25 similarity; 2.9% and 4.6% unseen). Textual controls therefore change how a problem is described or parameterized, while diffusion moves the constraint skeleton. This is the data-side content of claim (A), and it is consistent with textual synthesis adding no gain over the unaugmented seeds (A2, 68.3% versus 71.7% in Table 3).

Table 10: Text vs. structure. Text sim. is all-mpnet-base-v2 cosine: mean nearest-neighbour of each Reverse statement to OR-Instruct seeds, or paired cosine for A2 paraphrase. Graph sim. is 0.5 signature Jaccard $+0.5$ Weisfeiler–Lehman histogram cosine: mean pairwise vs. seeds for diffusion, and paired similarity for the Table 9 paraphrase control (a variable bound rewritten as a degree-1 constraint with probability 0.05; $141/2980$ graphs change). Diffusion stays in the OR narrative register while moving the topology; paraphrase does the reverse.

Pair	Text sim. ↑	Graph sim. ↓
(seed q, Reverse q) / (seed graph, diffusion graph)	0.72	0.25
(original q, paraphrased q)	0.90	0.99

The first row pairs each Reverse statement with its nearest OR-Instruct seed in text space, and each diffusion graph with the seed graphs in signature/WL space. Text similarity remains moderate (0.72): Reverse is instructed to write fluent OR narratives with few-shot examples matched by application type y, so the language stays in-register. Graph similarity is low (0.25): the denoised topology is not a paraphrase of a seed skeleton. The A2 control does the opposite. Paired cosine between original and paraphrased statements is 0.90, while graph similarity is 0.99. Diffusion therefore supplies structural diversity under a stable narrative register; extra paraphrases of the same gold code do not.

E Infeasible-Reference Slice

Section 3.3 defines V_3 by solver-status agreement, $\mathrm{st}(T) = \mathrm{st}(\mathcal{P})$, rather than by the existence of a feasible point, and sets $V_4 := V_3$ when the reference is infeasible. No public benchmark exercises that branch: all six public sets, and every reference in $\mathcal{D}_{\text{SFT}}$ and $\mathcal{D}_{\text{int}}$, solve to optimality. Yet deciding *whether a model is infeasible at all* is among the most common tasks in applied optimization, and a scalar optimality metric cannot express it. We therefore synthesize a slice on which the correct answer *is* an infeasible status.

Construction. The witness-point argument of Section 3.2 is reversible. Given a verified feasible instance, we append two opposing rows over a variable subset S, namely $\sum_{j \in S} x_j \le C$ and $\sum_{j \in S} x_j \ge D$ with $D > C$. Taking multipliers equal to 1 on these two rows and 0 elsewhere gives $\mathbf{A}^\top \mathbf{y} = \mathbf{0}$ and $\mathbf{b}^\top \mathbf{y} = C - D < 0$, which is a Farkas certificate: the system is empty regardless of the remaining rows, and emptiness of the relaxation implies infeasibility of the MILP. Infeasibility is thus provable before any solver call, exactly as the witness point makes feasibility provable. We use this construction only for evaluation; no infeasible instance enters $\mathcal{D}_{\text{SFT}}$. The base instances are

drawn from $\mathcal{D}_{\text{int}}$ itself—the held-out set of Section 4, separated from $\mathcal{D}_{\text{SFT}}$ by graph-isomorphism and statement-overlap checks—so no checkpoint has seen the underlying structure, and the appended rows are the only content that is new at evaluation time.

Three difficulty levels. The slice holds n=180 instances, 60 per level, stratified over application types y. *Type I* places the two opposing rows directly. *Type II* spreads the contradiction over many rows—per-customer demand lower bounds against per-plant capacity upper bounds, infeasible exactly when $\sum_i d_i > \sum_j c_j$—so that the Farkas multipliers are supported on the whole constraint block. *Type III* is integrality-only: the relaxation is feasible but the integer program is not (e.g. $2x_1+2x_2 = 3$ over $\mathbb{Z}_+$), which carries no Farkas certificate and is the case that a "has a feasible point" reading of V_3 cannot express at all. Reverse is required to distribute the conflicting quantities across the narrative, so that the contradiction is visible only after the model is built rather than from the wording; we discard statements that fail a manual check of this property.

Metrics. We report the status distribution rather than a single rate. V_1^{build} is the fraction of responses that construct a model and return from `optimize()`; the status is then read from the model object, not from standard output. This differs from V_1 elsewhere in the paper because correct code on an infeasible instance frequently raises an exception when it prints an objective that does not exist, which the strict reading would score as a crash. *Status agreement* is $\Pr(\text{st}(T) = \text{INFEASIBLE})$, i.e. V_3 on this slice. *False-feasible* is $\Pr(\text{st}(T) = \text{OPTIMAL})$, the fraction of responses that report an objective value for a problem that has none. Finally, *incidental* agreement is $\Pr(V_2{=}0,\ \text{st}(T) = \text{INFEASIBLE})$: the right status from a model with the wrong structure.

Table 11: Infeasible-reference slice (n=180; %). Status agreement is $\Pr(\text{st}(T){=}\text{INFEASIBLE})$; false-feasible is $\Pr(\text{st}(T){=}\text{OPTIMAL})$. "Incidental" is status agreement attained with V_2=0, i.e. the right verdict from the wrong model. No infeasible reference appears anywhere in training.

Model	V_1^{build}	p_2	Status agr. (V_3)	False-feasible	Incidental
Base Qwen3-8B	99.4	55.6	77.2	22.2	21.7
Weakest-layer SFT	100.0	73.9	86.1	13.9	12.2
SFT + gated DPO	100.0	88.9	96.1	3.9	7.2
DeepSeek-V3 (zero-shot)	99.4	88.3	93.3	6.1	5.0

Reading. Status agreement rises from 77.2% to 96.1% and false-feasible answers fall from 22.2% to 3.9%, although no infeasible reference appears in $\mathcal{D}_{\text{SFT}}$. Structural fidelity rises in step, $55.6\% \rightarrow 73.9\% \rightarrow 88.9\%$. At every checkpoint, each response that matched the reference structure also reached the infeasible status, so status agreement equals p_2 plus the incidental share. The incidental share falls from 21.7 to 7.2 points, so the gain comes from structurally faithful models. The same checkpoint reaches 96.1% status agreement and 88.9% structural fidelity, compared with 93.3% and 88.3% for DeepSeek-V3.

By difficulty level. Table 12 splits status agreement over the three construction levels. The ordering the construction predicts is the ordering observed at all three of our checkpoints: Type I, whose contradiction sits in two adjacent rows, is easiest, and Type III, which carries no Farkas certificate, is hardest. The base model is at 98.3% on Type I; its Type II and Type III rates are 68.3% and 65.0%. The gains by level are +1.7, +30.0, and +25.0.

Of the 7 false-feasible responses left at the final checkpoint, 6 are Type III and one is Type II, with Type I solved at 60/60. Type III is the level whose relaxation is feasible and on which only integrality fails, so the solver's status is the evidence V_3 records. A reading of V_3 that asks only whether a feasible point exists cannot score this level.

Implications for evaluation. Opt@1 does not score this slice, because a problem with no optimum has no objective to compare. Defining V_3 by status agreement makes the infeasible answer, including Type III, expressible. The gains above track structural fidelity, a layer the training recipe does optimize: as V_2 improves, V_3 improves on a status branch that the training references never include.

Table 12: Status agreement $\Pr(\mathrm{st}(T){=}\textsc{infeasible})$ by construction level (%). Type I places two opposing rows adjacently; Type II spreads the contradiction over a whole constraint block; Type III is integrality-only and carries no Farkas certificate. The last row reproduces the status-agreement column of Table 11.

Level	n	Base Qwen3-8B	Weakest-layer SFT	SFT + gated DPO	DeepSeek-V3
Type I (direct pair)	60	98.3	95.0	**100.0**	95.0
Type II (spread)	60	68.3	88.3	**98.3**	98.3
Type III (integrality)	60	65.0	75.0	**90.0**	86.7
All	180	77.2	86.1	**96.1**	93.3

F Held-out Constraint Signatures and Difficulty Slices

Table 13 tests whether the Opt@1 gains in Table 1 come from new constraint combinations rather than from more text about familiar skeletons. The first column restricts $\mathcal{D}_{\text{int}}$ to instances whose constraint-combination signature (variable-type multiset and constraint-degree sequence) is absent from OR-Instruct seeds. The remaining columns report public Opt@1 on two harder slices: mixed-integer instances, and instances whose constraint count is in the upper tail of the public sets. Seed SFT and textual synthesis (A2) are the same-backbone controls of Tables 1 and 3; OR-Diffusion is the full pipeline. All three rows use Qwen3-8B and the same N.

If claim (A) is right, OR-Diffusion should lead clearly on held-out signatures: seed and paraphrase models have never seen those skeletons, whereas 42.9% of diffusion signatures are unseen in the seeds (Table 9). The MILP and high-constraint columns are coarser difficulty checks on the public sets.

The prediction holds. Against seed SFT, the held-out gap (+28.7) is the largest of the three columns (+9.9 on the MILP slice and +5.3 on the high-constraint slice). Against paraphrase, the held-out and MILP gaps are +20.0 and +21.2. OR-Diffusion leads every column.

Table 13: Held-out constraint signatures on $\mathcal{D}_{\text{int}}$, and public Opt@1 on MILP vs. high-constraint slices (%).

Method	Held-out signature Opt@1	MILP slice	High-constraint slice
Seed SFT	50.3	65.9	52.9
Textual synthesis	59.0	54.6	50.7
OR-Diffusion	**79.0**	**75.8**	**58.2**

G Prompt Templates

This section records the full LLM prompt flow used to realize a verified graph as an SFT/DPO pair (q, T_g), and the modeling instruction used at train and test time. Reverse (LP $\rightarrow$ natural language) and Rewrite (natural language $\rightarrow$ code) are called with DeepSeek-V4-Flash (DeepSeek-AI, 2026); few-shot examples are drawn from OR-Instruct and matched by application type y. Placeholders in angle brackets are filled at runtime. At inference, OR-Diffusion is given only q: no HVE scores, graph tokens, or preference labels are revealed.

We use the following order. (i) DeepSeek-V4-Flash labels each OR-Instruct seed with one of 20 application types, yielding the conditioning vector y for diffusion and the few-shot retrieval key. (ii) Reverse turns a solver-verified LP into a fluent word problem q, with type-matched reference stories used as style examples only. (iii) Rewrite sees only q and must emit executable Gurobi Python. The objective is compared with $f^\star$ only when that code is feasible; infeasible programs are discarded without that comparison, and a feasible program is kept only if the values match within 10^{-4}, with up to two retries that include the previous error. (iv) Passed instances are assembled under a single modeling instruction, which is also the prompt used for SFT, DPO rollouts, and evaluation.

G.1 Application-Type Classification Prompt

Each OR-Instruct seed is classified into exactly one of 20 application types. The resulting one-hot vector y conditions the diffusion model and selects Reverse few-shot examples. The annotator returns JSON only.

Prompt for Application-Type Classification

```
You are an expert in Operations Research (OR) and mathematical
    programming.
Classify each optimization problem into EXACTLY ONE of the 20
    categories below.
Pick the single best-fitting category based on the core modeling
    structure, not surface industry keywords.

Categories:
1. Resource Allocation: allocate limited budget/resources among
    options.
2. Production Planning: production quantities under material/machine
    capacity.
3. Scheduling: assign tasks/jobs/crews to time slots or machines.
4. Blending: mix ingredients under recipe or quality constraints.
5. Workforce Management: hire/staff employees with work-pattern
    rules.
6. Selection: choose a discrete subset under budget or logic
    constraints.
7. Sizing: determine sizes or production levels of variants under
    shared capacity.
8. Routing: vehicle/path routes visiting locations (TSP/VRP-style).
9. Inventory Management: multi-period production/inventory with
    holding costs.
10. Packaging / Packing: pack items into bins/knapsacks under
    capacity.
11. Pricing: set prices to maximize revenue or profit.
12. Balancing: balance workloads, flows, or component production.
13. Portfolio Optimization: allocate capital across assets with
    risk/return constraints.
14. Network Flow: ship commodities over a network with supplies,
    demands, and capacities.
15. Covering: open/select facilities so that all demand points are
    covered.
16. Layout: allocate space or arrange facilities under area and
    adjacency constraints.
17. Warehouse Selection: decide which warehouses to open and how to
    serve customers.
18. Assignment: assign agents to tasks/machines/jobs.
19. Cutting Stock: cut stock material into ordered pieces to
    minimize waste.
20. Transportation: ship goods from sources to sinks with
    supply/demand and unit costs.

Rules:
- Return ONLY a JSON object, no markdown fences, no extra text.
- Schema: {"category_id": <1-20>, "category": "<exact category
    name>", "confidence": <0-1>, "reason": "<one short sentence>"}
- "category" must be copied exactly from the list above.
- If two categories seem close, choose the more specific structural
    match
  (e.g. classic source-sink shipping table -> Transportation;
    multi-hop capacitated network -> Network Flow;
  open warehouses then ship -> Warehouse Selection; cover all sites
    within distance -> Covering).
```

```
Classify the following operations research problem into one of the
    20 categories.

Problem:
<QUESTION>
```

G.2 Reverse Prompt (From LP to Natural Language)

Reverse is instructed to embed every numerical value from the LP in a realistic OR narrative and *not* to transcribe constraints as spoken equations. Few-shot stories are matched by y and used for style only; their numbers must not be copied if they contradict the LP. Optional semantic notes describe auxiliary rows (deviation, share, peak, setup) that are not ordinary incidence constraints.

Prompt for Reverse Generation

```
As an Operations Research Expert, turn the given LP into a realistic
    OR-Instruct-style word problem.

The solver must still be able to recover the model from the story,
    but the story must NOT read like an LP recited in English.

Write like original OR-Instruct:
- Open with a concrete scenario (company, factory, warehouse,
    hospital, farm, airline, supermarket, ...) and named entities.
- Put numbers in tables, capacities, budgets, demands, yields, or
    short factual sentences --- not pre-assembled linear
    combinations.
- Constraints should be implicit in the narrative when possible
    ("meet demand", "each job assigned to exactly one worker",
    "cannot exceed tank capacity").
- A human reading the story should still have to decide variables,
    equalities vs inequalities, and how the objective is composed.
- Include every resource / limit / bound / type that exists in the
    LP. Do not invent extra resources, extra products, or extra
    constraints.
- All numeric values from the LP (objective coefficients, constraint
    coefficients, RHS, bounds) must appear somewhere so a modeler
    can recover them, but NEVER as a spoken formula like `8.2 A +
    9.07 B + 3.45 C <= 93.83`.
- Variable types (continuous / integer / binary) and bounds must
    remain recoverable from the prose or tables.
- If the LP marks variables General (integer): write countable
    language --- "how many", "number of batches/servings/units",
    "whole units only". NEVER say "fractional amounts",
    "continuous", or "can take fractions" for integer variables.
- If Binary: write yes/no, select, open/close, assign-or-not.
- Prefer a markdown or ASCII table for coefficients, capacities, and
    demands.
- Prefer 800-1500 characters; do not exceed 2000.
- Avoid meta language ("this optimization problem", "decision
    variables", "objective function", "LP", "constraint C0").
- Do NOT dump LP file syntax or bullet lists of equations.
- End with a direct question asking for the optimal decision or
    value.
- If reference examples are provided, match their narrative style
    (scenario -> tables/facts -> question), not their numbers.
- Follow the SEMANTIC NOTES for auxiliary quantities; never ask the
    reader to choose them.

Preferred problem type / scenario family: <APP_TYPE>
```

```
The LP has exactly <N_CON> constraints under 'Subject To'. Your
    story must cover all of them (same count).

<SEMANTIC_NOTES>

Input LP Data (hidden spec for you; do not quote it back):
<LP_DATA>

Reference Examples (style only; never copy numbers that contradict
    the LP):
<FEWSHOT_STORIES>

## Required Output format:
WORD_PROBLEM:
<the full word-problem statement only --- no LP dump, no meta
    commentary, no spoken equations>
```

G.3 Rewrite Prompt (From Natural Language to Code)

Rewrite is given only the word problem q. It must emit a complete runnable Gurobi script with named variables (never flattened x_0, x_1). Six few-shot (problem, code) pairs from OR-Instruct are prepended. The generated code is executed. The objective is compared with $f^\star$ only when the program is feasible, and a feasible program is retained only if the printed objective matches $f^\star$ within 10^{-4}.

Prompt for Rewrite Generation

```
You are an operations research expert. Write a complete runnable
    Python script using gurobipy that models and solves the word
    problem.

Requirements:
- Use named variables, lists, or dicts (e.g. buses, gold_miles,
    x[i,j]). NEVER flatten decisions to x0, x1, x2.
- DEFAULT vtype is GRB.INTEGER for countable decisions (how many,
    servings, batches, units, people, items, boxes). Use
    GRB.CONTINUOUS only if the story explicitly allows fractions /
    rates / continuous amounts. Use GRB.BINARY for yes/no or select.
- Default addVars is continuous --- always set vtype explicitly.
- A list of products A,B,C is a 1-index model. Do NOT introduce
    2-index x[i,j] unless the story has two interacting entity
    families (jobs x machines, plants x customers).
- Index nested Python lists with a[i][j], never a[i,j]. Build dict
    keys before quicksum; do not reference x[i,j] unless that key
    was created.
- Prefer addVars / quicksum / loops when entities repeat. Do not add
    constraints the story does not state.
- Use == when the story implies an exact requirement (meet demand
    exactly, assign each job once, flow conservation, a stated
    ratio).
- Only <=, >= and == may be applied to model variables. Python's <
    and > raise TypeError on a Var. "more than N" on an integer
    decision is >= N + 1.
- "A and B differ by at most D" is TWO linear constraints:
    addConstr(a - b <= D) and addConstr(b - a <= D). NEVER write
    abs(a - b) inside a comparison.
- "X is at most p% of the total of X, Y, Z" is addConstr(x <= 0.p *
    (x + y + z)). Never divide by an expression containing a
    variable.
```

```
- "the largest / busiest / peak of several totals" needs one
    auxiliary continuous variable z with addConstr(z >= total_i) for
    every group, and z in the objective. Never call Python max() or
    min() on model variables.
- A fixed setup / opening charge needs a binary y, a link
    addConstr(total <= M * y) with a constant M large enough to
    cover the total, and the charge added to the objective.
- Print the objective as: print('Objective:', model.ObjVal)
- Output ONLY Python code. No explanation, no markdown fences.

Examples of the desired coding style:

### Example 1
Problem:
<EXAMPLE_QUESTION_1>
Code:
<EXAMPLE_GUROBI_CODE_1>

### Example i
Problem:
<EXAMPLE_QUESTION_i>
Code:
<EXAMPLE_GUROBI_CODE_i>

### Now solve this problem
Problem:
<QUESTION>
Code:
```

G.4 Rewrite Retry Prompt

If the first Rewrite sample fails to execute, is infeasible, or misses $f^\star$, the same prompt is reused with the following block inserted before the target problem (at most two retries; temperature is raised on later attempts). Persistent failures are discarded.

Prompt Block for Rewrite Retry

```
### Previous attempt was incorrect --- fix it
Required objective value: <GOLD_OBJECTIVE>
Your previous objective: <PREV_OBJECTIVE>
Error / log: <ERROR_LOG>

Typical fixes: GRB.INTEGER instead of CONTINUOUS for countable
    units; list index a[i][j] not a[i,j]; do not invent extra
    constraints; include every demand/capacity/equality the story
    states.
```

G.5 Modeling Prompt (SFT, DPO, and Inference)

SFT pairs, DPO preference pairs, and test-time generation share one instruction. The model must write Gurobi Python from the question alone.

Baseline Prompt Template for Optimization Modeling

```
Below is an operations research question. Write Python code using
    'gurobipy' to solve it.
```

```
<QUESTION>
```