FrontierOR: Benchmarking LLMs' Capacity for Efficient Algorithm Design in Large-Scale Optimization

Minwei Kong [1 *] **Chonghe Jiang** [1 2 *] **Ao Qu** [1 2 †] **Wenbin Ouyang** [2] **Zhaoming Zeng** [3] **Xiaotong Guo** [4 ‡] **Zhekai Li** [2] **Junyi Li** [1] **Yi Fan** [5] **Xinshou Zheng** [6] **Xi Jing** [6] **Yikai Zhang** [6] **Zhiwei Liang** [7] **Seonghoo Kim** [8] **Runqing Yang** [6] **Zijian Zhou** [9] **Sirui Li** [10 ‡] **Han Zheng** [2] **Wangyang Ying** [11] **Ou Zheng** [11] **Chonghuan Wang** [12] **Jinglong Zhao** [6] **Hanzhang Qin** [9] **Cathy Wu** [2] **Paul Pu Liang** [1 2] **Jinhua Zhao** [1 2 §] **Hai Wang** [1 13 §]

Abstract

Large language models (LLMs) are increasingly used for optimization modeling and solver-code generation, yet practical operations research and optimization problems often require a harder capability: designing scalable algorithms that exploit problem structure and outperform direct formulation-and-solve baselines. Existing benchmarks are limited to small or simplified examples far below real-world scale and complexity. We introduce **FrontierOR**, the first literature-grounded benchmark targeting LLM-generated algorithm efficiency on realistic large-scale optimization problems. FrontierOR includes a methodologically diverse set of 180 tasks, each derived from one paper published in top-tier operations research venues, with standardized instances and a hidden, expert-verified evaluation suite. We evaluate seven LLMs spanning frontier, cost-effective, and open-source models both in one-shot and test-time evolution settings. The results reveal that frontier models still struggle to move from executable formulations to efficient optimization algorithms: the strongest one-shot model matches Gurobi's solution quality within a 1% gap at a lower runtime on only 31% of Full-set instances, and even strong coding agents with test-time evolution achieve only 50% on the selected Hard-set instances. FrontierOR establishes a practical evaluation platform for LLM-based optimization algorithm design, which enables future LLMs and agents to be systematically tested on whether they can move beyond correct formulation toward a feasible, high-quality, and efficient algorithm. Code and data are publicly released at https://github.com/Minw913/FrontierOR.

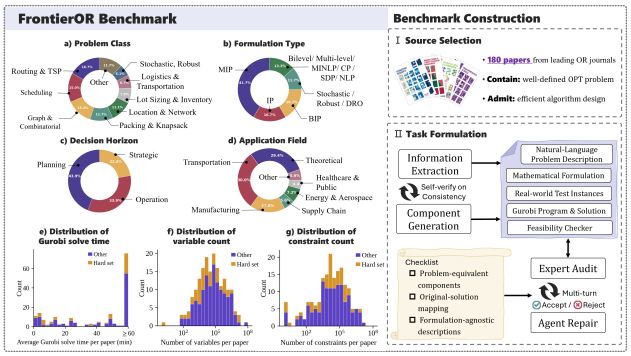


Figure 1. Overview of the FrontierOR benchmark. FrontierOR spans diverse problem domains, formulation types, and application fields. Each optimization problem involves 10^2 to 10^7 decision variables and constraints, with Gurobi failing to reach optimality on 46% of large-scale instances within a one-hour time budget. We construct the benchmark by collecting problems from leading OR journals, and ensure data quality through multi-round expert review.

1. Introduction

Recent progress in large language models (LLMs) has stimulated growing interest in using LLMs for operations research (OR). Most existing work studies LLMs as modeling assistants, evaluating whether they can translate natural-language decision problems into mathematical optimization formulations or solver-ready programs (Ramamonjison et al., 2023; AhmadiTeshnizi et al., 2024; Yang et al., 2024; Michailidis et al., 2025; Kong et al., 2025; Huang et al., 2025a; Zhou et al., 2025; Xiao et al., 2023; Jiang et al., 2024). While problem modeling is an important capability, operations

*Equal contribution [1]Singapore-MIT Alliance for Research and Technology [2]Massachusetts Institute of Technology [3]Northeastern University [4]Uber [5]Shanghai Jiao Tong University [6]Boston University [7]Emory University [8]Northwestern University [9]National University of Singapore [10]Microsoft [11]Zhiling Research [12]University of Texas at Dallas [13]Singapore Management University. Correspondence to: Jinhua Zhao <jinhua@mit.edu>, Hai Wang <haiwang@smu.edu.sg>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026. Copyright 2026 by the author(s).

research in practice is not only about formulating the correct model, but also designing algorithms that can solve large problems reliably and efficiently. The practical value of an OR algorithm depends on whether it can exploit problem structure and scale under realistic computational constraints. Therefore, this leaves a central capability largely under-measured:

Q1: Can LLMs read a real-world decision problem description and implement an efficient algorithm?

Answering this question requires a systematic evaluation framework with *realistic* problem description, *large-scale* problem coverage, *scalable* problem sizes, and *rigorous* evaluation suites. However, existing benchmarks (Reinelt, 1991; Yang et al., 2024; Sun et al., 2026) either lack natural-language problem descriptions, rely on overly simplified descriptions, or contain only a limited number of cases. This raises a complementary question about the data foundation needed for such an evaluation:

Q2: Can we construct a large-scale benchmark evaluating LLMs for efficient optimization algorithm generation with realistic descriptions, scalable instances, and rigorous evaluators?

To this end, we introduce **FrontierOR**, a benchmark for evaluating optimization algorithm efficiency on realistic large-scale OR problems. Unlike previous works that primarily evaluate formulation accuracy on textbook-scale modeling tasks, FrontierOR contains 180 tasks derived from top-tier OR literature, with instances spanning up to $\sim 10^7$ decision variables and $\sim 10^7$ constraints. The tasks are selected because they contain exploitable algorithmic structure—for example, decomposability, temporal or network coupling, sparsity, routing and assignment structure, or scenario structure—so solving them effectively requires more than constructing a monolithic solver formulation. FrontierOR therefore evaluates generated programs not only by feasibility, but also by solution quality and runtime competitiveness against an expert-verified Gurobi baseline, directly targeting algorithm efficiency rather than formulation accuracy alone. To further ensure benchmark quality, after information extraction from the source papers, each task is annotated and verified by 15 OR experts through a three-week, multi-turn review process with substantial human-in-the-loop validation.

For each task, the model is given only a natural-language problem description together with the input and output formats for test instances. It does not receive the mathematical formulation, reference solver implementation, or any hints about the intended algorithmic approach. The hidden evaluation suite contains three components: a representative mathematical formulation, an expert-verified Gurobi implementation used as the reference solver, and a task-specific feasibility checker.[1]

Evaluation proceeds in two stages. First, we assess *correctness* by running the generated program on benchmark instances and checking whether its reported solution satisfies all hard constraints using the standalone feasibility checker. Infeasible outputs receive no objective credit, even if the reported objective value appears favorable. Second, for feasible solutions, we assess *algorithmic performance* along two dimensions: solution quality, measured by the objective gap relative to the Gurobi reference solution, and computational efficiency, measured by the running time relative to the Gurobi baseline under the same resource budget. Our main quality–time efficiency metric counts an instance as successful only when the generated algorithm returns a feasible solution whose objective is within 1% of the Gurobi reference and does so no slower than Gurobi.

We evaluate seven LLMs under two protocols. In the *one-shot* protocol, the model generates a complete program for each task from the problem description alone. In the *self-evolve* protocol, an agent iteratively proposes and revises candidate programs using feedback from a development set, while the final score is computed on held-out test instances. Our experiments show that current LLMs remain far from reliable algorithm designers: the strongest one-shot model achieves quality–time efficiency on fewer than 40% of task instances, and self-evolving agents improve this rate to 50% on selected hard tasks. The generated algorithms also exhibit systematic differences across LLMs. Weaker models often default to monolithic solver calls, whereas stronger frontier models more frequently produce decomposition, local-search, or hybrid algorithms. These differences lead to distinct feasibility, quality, runtime, and failure-mode profiles, revealing a substantial gap between current LLM capabilities and the algorithmic sophistication required in real-world operations research.

In summary, we introduce a benchmark that evaluates LLMs on large-scale optimization algorithm design, shifting the focus from modeling correctness alone to feasible, scalable, and efficient algorithm generation. Our main contributions are listed as follows:

1. We construct a broad, literature-grounded benchmark, FrontierOR, which consists of 180 real-world OR tasks spanning diverse industrial problem settings, each with natural-language specifications, large-scale instances, math formulations, verified Gurobi baselines, and standalone feasibility checkers.
2. We construct a systematic evaluation pipeline for FrontierOR and conduct extensive experiments to assess the capabilities of seven state-of-the-art LLMs as well as

[1] An optimization problem may admit multiple valid formulations. For each task, we select one representative formulation and implement its corresponding Gurobi solver code as the baseline.

three representative self-evolving inference-time scaling methods. We perform a comprehensive analysis using multi-dimensional metrics, including feasibility, solution quality, and quality–time efficiency.

2. Related Work

LLMs for optimization modeling and solver implementation. A growing body of work evaluates LLMs for optimization modeling and whether they can translate natural-language decision problems into mathematical programs or solver-executable code. Representative benchmarks and systems include NL4OPT, OptiMUS, OptiBench, MAMO, ORLM, OptMATH, CP-Bench, AlphaOPT, LEAN-LLM-OPT, and StepORLM (Ramamonjison et al., 2023; AhmadiTeshnizi et al., 2024; Yang et al., 2024; Huang et al., 2025b;a; Lu et al., 2025; Michailidis et al., 2025; Liang et al., 2026; Zhou et al., 2025), which mainly test formulation correctness and solver implementation. In contrast, FrontierOR evaluates whether LLMs can design efficient *algorithms* when monolithic formulation-and-solve pipelines are too slow or memory-intensive for large-scale instances of practical problems.

Combinatorial optimization benchmark suites. Benchmark suites, which enable reproducible solver comparisons and algorithm testing beyond hand-picked instances, have long been central to empirical research in combinatorial optimization. Classical resources such as OR-Library (Beasley, 1990), TSPLIB (Reinelt, 1991), CVRPLIB (Uchoa et al., 2017), QAPLIB (Burkard et al., 1997), and MIPLIB (Gleixner et al., 2021) remain indispensable for evaluating exact and heuristic algorithms. However, they scale mainly by diversifying instances within a few classical problem classes, not by broadening problem variety.

Recent LLM-based benchmarks such as CO-Bench (Sun et al., 2026) and HeuriGym (Chen et al., 2025) have begun to study LLM algorithmic efficiency, but they remain concentrated on classical optimization cases and relatively small task collections. This leaves open whether LLMs can design scalable algorithms across diverse, large-scale, real-world OR settings. A more detailed comparison with representative OR and LLM-for-optimization benchmarks is summarized in Table 1.

Table 1. Comparison with representative OR and LLM-for-optimization benchmarks.

Benchmark	End-to-End	#Problems	Literature-Grounded
OR-Library (Beasley, 1990)		19	
MIPLIB 2017 (Gleixner et al., 2021)		N/A	
CO-Bench (Sun et al., 2026)		36	
HeuriGym (Chen et al., 2025)	✓	9	
GraphArena (Tang et al., 2024)	✓	6	
FrontierCO (Feng et al., 2025)		8	
NLCO (Jiang et al., 2026)	✓	43	
FrontierOR (Ours)	✓	**180**	✓

Note. End-to-End indicates evaluation from a natural-language problem description to an executable solution. Literature-grounded indicates that benchmark problems are extracted or adapted from papers with a real problem context.

3. FrontierOR Benchmark

This section describes the construction pipeline of FrontierOR. Figure 1 summarizes two main components: dataset collection and task formulation. The dataset covers diverse problem classes, application domains, and instance scales. The task-formulation pipeline converts selected OR papers into algorithm-design tasks through information extraction, component generation, and human validation, producing standardized instances, Gurobi references, and feasibility checkers. Additional construction details are provided in Appendix A.

3.1. Dataset Description

Source selection criteria. We curate tasks from leading OR journals, including *Operations Research*, *Management Science*, *Transportation Science*, *INFORMS Journal on Computing*, *European Journal of Operational Research*, as well as more than 20 additional venues. As shown in Figure 1, each selected paper studies a well-defined optimization problem, provides sufficient information for task construction, and motivates efficient algorithm design beyond directly applying a generic solver. These criteria ensure that every task is grounded in a real optimization setting in which scalable algorithm design is practically important.

Dataset statistics. FrontierOR contains 180 literature-grounded tasks from 1992–2025, including 69 papers published in 2020 or later. Figure 1 demonstrates that the collected benchmark tasks span five optimization paradigms (e.g., MIP, MINLP, and DRO), nine problem classes (e.g., routing, scheduling, and location), and a wide range of application fields (e.g., transportation, energy, supply chain, and healthcare). Problem instances are large and computationally hard: a median problem has roughly $40{,}000$ decision variables and $18{,}000$ constraints on its smallest large instance (the other four sizes scale up further), and a state-of-the-art commercial solver fails to reach optimality on 46% of large instances within a one-hour budget. Each task is also annotated as *planning*, *strategic*, or *operational* to align algorithmic performance with real deployment constraints. Additional statistics are provided in Appendix C.

3.2. Task Formulation

We then describe how FrontierOR tasks are constructed from source papers through a pipeline of automated component generation followed by expert quality assurance.

Component generation. As shown in Figure 1, each

selected paper is converted into a suite of task components through information extraction and component generation. We extract the natural-language problem description and mathematical formulation, then use the formulation as the primary reference for constructing a standardized test-instance format, a Gurobi implementation with reference solutions, and a standalone feasibility checker, see examples in Appendix E.1.

Quality assurance. We apply two levels of quality assurance. First, automatic verification cross-runs each feasibility checker against all Gurobi reference solutions; any inconsistency triggers a repair loop over the checker, Gurobi code, data parsing, numerical tolerance, and extracted formulation. Second, 15 OR experts audit the final component suite with the source paper through multi-turn review. The audit checks formulation fidelity, description completeness, Gurobi-code alignment, and whether the checker verifies the original problem constraints rather than solver-specific reformulation artifacts. After iterative repair and re-verification, all retained tasks have Gurobi solutions that pass their checkers and LLM-facing descriptions that are complete enough for executable algorithm design. The full checklist and common error categories are reported in Appendix B.

After the above construction and validation steps, each task contains two parts. The *LLM-facing input* includes only a prose problem description and standardized instances. The description is synthesized from the source paper, formal model, and input/output schemas, but avoids mathematical notation and modeling terminology, so the model must infer the latent optimization structure from operational language. The *hidden evaluator* includes the mathematical formulation, Gurobi implementation, reference solutions, and feasibility checker. Submitted algorithms are first checked for output schema, variable domains, and hard constraints. Infeasible outputs receive no objective credit; feasible outputs are compared with Gurobi references using the accuracy and efficiency metrics defined in Section 4.2.

Full/ Hard partition. We curate a *Hard* subset of 50 tasks from FrontierOR to evaluate algorithm performance on genuinely difficult OR problems. Tasks are selected using three criteria: *(1) Problem class with combinatorial blow-up.* Classical NP-hard families that resist monolithic MIP at realistic scales (e.g., lot-sizing, graph optimization, and sequencing scheduling). *(2) Instance structural difficulty.* Instances with above-median scale (variables, integer variables, constraints) and strong coupling structures such as multi-stage time windows, scenario uncertainty, and tight capacities. *(3) Empirical solver saturation.* Tasks where Gurobi fails to reach optimality within a 1-hour budget or terminates with a non-zero MIP gap. A task enters Hard when either (1) or (2) provides structural support and (3) confirms solver saturation.

4. Experimental Setup

4.1. Computational Environment

To ensure that algorithms are compared on algorithmic merit rather than implementation-level parallelism, all reported experiments are executed on a single CPU core of an AMD EPYC 9554 64-core processor. We adopt a docker-based execution environment that enforces strict, reproducible resource constraints: Each run is allocated a single CPU core within a fixed container image shipping Python 3.13 and Gurobi 12 as the sole optimization solver, together with standard scientific libraries (e.g., NumPy and SciPy) for data manipulation. Network access is disabled to prevent generated code from reaching external resources during evaluation. This containerized setup ensures reproducibility.

4.2. Evaluation Metrics

We report four primary metrics to assess the solution quality and computational efficiency of the LLM-generated algorithms that perform on these optimization tasks. **Execution rate**: the proportion of tasks for which the LLM's first-generated program executes successfully without runtime errors. **Feasibility**: the proportion of large instances across all tasks on which the generated program returns a feasible solution within the time budget. **Solution quality**: the proportion of large instances across all tasks on which the generated program's feasible objective is within 1% of, or better than, the Gurobi reference. **Quality–time efficiency (QTE)**: the proportion of large instances across all tasks on which the generated program matches Gurobi on *both* dimensions—returning a feasible solution with objective within 1% of the Gurobi reference *and* finishing within Gurobi's wall-clock runtime.

4.3. Benchmarked Models and Evaluation Protocols

Evaluation focus: algorithm efficiency beyond formulation accuracy. FrontierOR evaluates generated programs as *algorithms*, not as symbolic formulations. The hidden formulation and Gurobi implementation serve only as references for feasibility checking, objective comparison, and runtime comparison; the model is never asked to reproduce them. Simply producing a correct monolithic formulation is insufficient: the program must also be computationally competitive on large instances. At the same time, non-exact methods such as decomposition, local search, metaheuristics, or hybrid algorithms are rewarded whenever they return feasible, high-quality solutions faster than the baseline. This protocol directly targets LLMs' capacity for scalable algorithm design rather than formulation accuracy alone.

Base models. We evaluate 7 LLMs spanning 7 vendors: three frontier models with state-of-the-art coding capability—*GPT-5.3-Codex* (OpenAI), *Claude*

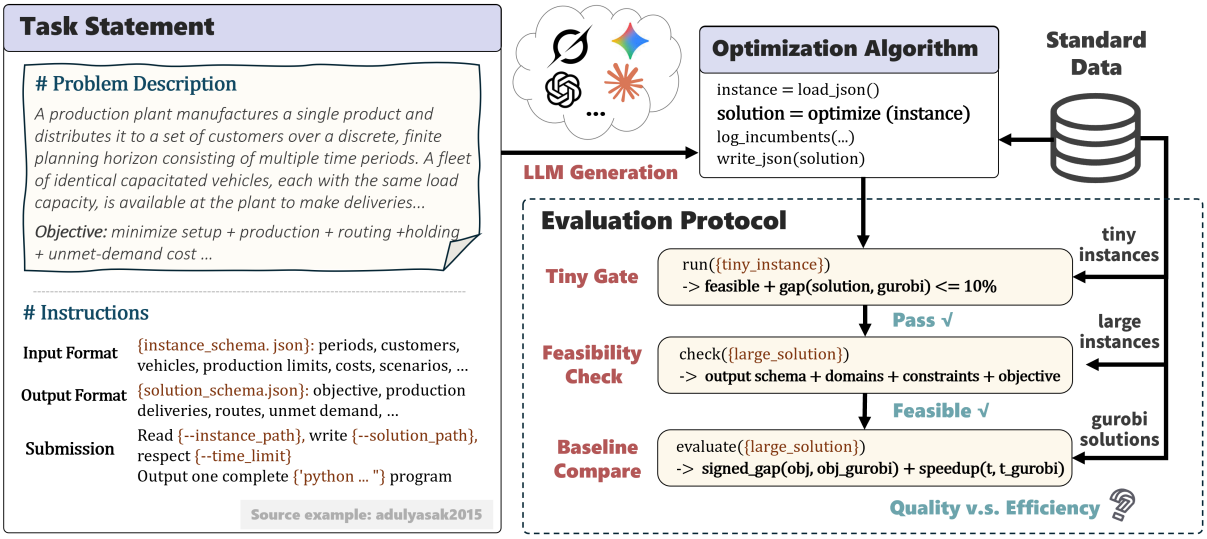


Figure 2. Evaluation Protocol of a FrontierOR benchmark task. We feed the problem description and instructions into an LLM to generate an end-to-end optimization program. The program is first sanity-checked on a tiny instance; if it passes, it is evaluated on large instance sets for feasibility and compared with a Gurobi reference in terms of solution quality and computational efficiency.

Opus 4.6 (Anthropic), and *Gemini 3.1 Pro Preview* (Google DeepMind)—and four cost-effective or open-source alternatives—*DeepSeek-R1* (DeepSeek), a reasoning-specialized model; *Grok-4.20* (xAI); *Qwen3-Coder-Plus* (Alibaba), a code-generation-specialized model; and *LLaMA-4-Maverick* (Meta), and other open-sourced models. In all settings, the model sees only the natural-language problem description and the instance/solution schemas without any formulation or algorithm hints.

One-shot generation. This protocol in Figure 2 is designed to evaluate the model's direct capability to infer the underlying optimization structure, choose a modeling approach and solution method, and generate and implement executable code. For each task, the model generates a complete optimization program from scratch, which allows a few self-debug iterations on execution errors. The sanity check on tiny instance fails the gate if the program times out, returns an infeasible solution, or exceeds an optimality gap of 10%. We introduce this tiny-instance gate to avoid unnecessary computational overhead from evaluating unqualified programs on large instances.

Self-evolving frameworks. This protocol measures the quality–time frontier achieved by LLM-designed algorithms as the test-time budget for self-evolution increases, with particular focus on whether iterative LLM-driven refinement can discover high-quality solutions more efficiently than well-crafted solver programs. We consider several representative agentic frameworks: (1) *AlphaEvolve* (Novikov et al., 2025): The LLM acts as a mutation operator within a MAP-Elites evolutionary algorithm over a program database; we adopt OpenEvolve (Sharma, 2025) as the open-source replication of this paradigm. (2) *EoH* (Liu et al., 2024): The LLM jointly evolves code and the natural-language thoughts behind it; its prompt operators explicitly encourage the agent to explore diverse solution approaches rather than locally perturbing existing code. (3) *CORAL* (Qu et al., 2026): In contrast to the single-agent exploration above, the system replaces fixed evolutionary search with autonomous multi-agents that explore, reflect, and collaborate through persistent shared memory.

Implementation details. We align all self-evolving frameworks under a unified interface: Agents can access the problem description and evaluation feedback, while the evaluator and Gurobi references remain hidden. For each task, large instances are split into a development subset for search feedback and a held-out test subset for final evaluation. To ensure fair comparison, all frameworks are limited to 30 proposed candidates per task and initialized from the same one-shot seed program.

In all self-evolving frameworks, candidates are scored by a cascaded fitness function. A tiny-instance gate first filters out low-quality programs using the same criteria as one-shot evaluation. Surviving candidates are then evaluated on the development subset under a piecewise rule: Infeasible or missing solutions receive 0; if the optimality gap exceeds 1%, only solution quality is rewarded; once the gap falls within 1%, the score includes a speed bonus based on relative runtime savings against Gurobi. The final fitness is the mean score across development instances. This design prioritizes solution quality early in evolution and shifts toward

efficiency only after acceptable quality is achieved.

5. Experimental Results

Our experiments are organized around two research questions:

1) One-shot algorithm generation. How well do current LLMs perform when directly generating optimization algorithms, measured across executability, feasibility, solution quality, and computational efficiency? What algorithmic types (e.g., calling solver, writing inexact heuristic) or failure modes do they tend to produce, and how do these modes affect scalability and competitiveness against the Gurobi baseline?

2) Test-time self-evolution. To what extent can test-time self-evolution improve LLM-generated algorithms across the same performance dimensions? How do algorithms evolve over iterations, and what improvement dynamics emerge during the evolution process?

5.1. Algorithm Performance Under the One-Shot Setting

We compare seven base models on FrontierOR *Full* ($n = 180$) and *Hard* ($n = 50$) using four metrics in Table 2, with the Hard subset consisting of tasks that are more computationally demanding.

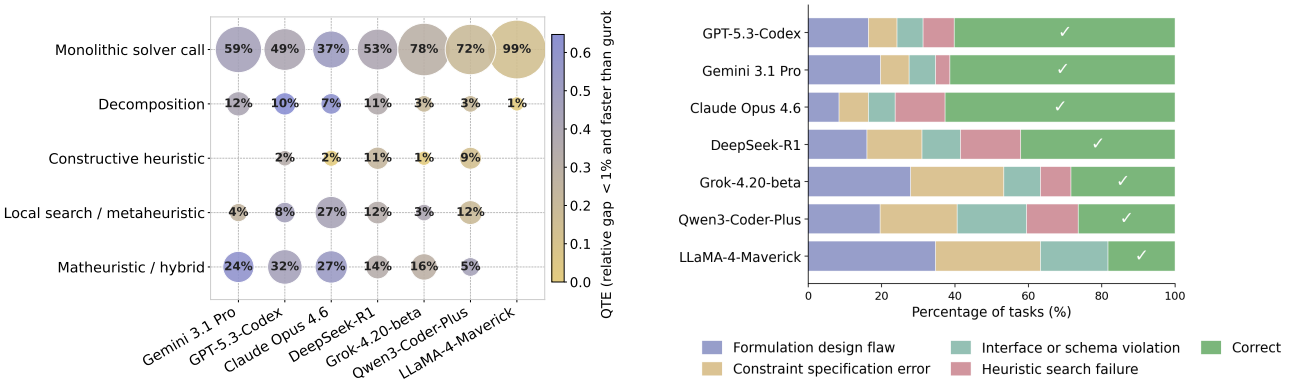
Performance across models. Table 2 shows three main patterns. (1) *Frontier models clearly outperform cost-effective models on both Full and Hard.* On Full, their feasibility scores cluster at 0.60–0.62, compared with 0.18–0.42 for cost-effective models; on Hard, the frontier band shifts down to 0.49–0.64 while cost-effective models drop to 0.13–0.37—the gap between the two tiers persists at both scales. (2) *Execution is no longer the primary bottleneck for the strongest models.* *GPT-5.3-Codex* reaches an execution rate of 0.98 and *Gemini 3.1 Pro* and *Claude Opus 4.6* 0.93 on Full, yet their feasibility and solution-quality scores remain substantially lower. This indicates that the central difficulty lies not in producing runnable code, but in generating algorithms that remain valid and effective on large instances. (3) *The Hard subset exposes sharp differences in algorithmic robustness.* On Full the three frontier models are tightly bunched (feasibility 0.60–0.62, QTE 0.25–0.31); on Hard the band widens (feasibility 0.49–0.64, solution quality 0.30–0.44, QTE 0.18–0.32) and the leaders re-separate. *Gemini 3.1 Pro* takes top Hard feasibility (0.64) and ties *Claude Opus 4.6* on solution quality (0.44), while *Opus* retains the highest QTE on both subsets (0.31 on Full, 0.32 on Hard), against 0.22 for the next frontier model. *GPT-5.3-Codex* keeps execution near saturation (0.98) but its Hard feasibility, solution quality, and QTE all drop the furthest among frontier models (0.49, 0.30, 0.18), suggesting its programs run but rarely combine validity with Gurobi-level quality and speed—the joint condition that QTE rewards. We provide continuous measures of solution quality and quality-time efficiency, which further quantify the magnitude by which LLM-generated solutions outperform or underperform the Gurobi reference. See Appendix D for details.

Algorithmic pattern across models. Figure 3a classifies each generated algorithm code into five algorithm families and reports both the frequency of each family and its average performance. The algorithm-family breakdown shows several consistent patterns. (1) *Model choices differ substantially.* Models differ not only in their final performance, but also in the types of algorithms they generate. For example, *LLaMA-4-Maverick* almost always relies on monolithic solver calls (99%), while *Claude Opus 4.6* produces a much more balanced mix: 37% monolithic solver calls, 27% local search or metaheuristics, and 27% matheuristic or hybrid methods. (2) *Frontier models use more non-monolithic algorithms.* *Gemini 3.1 Pro*, *GPT-5.3-Codex*, and *Claude Opus 4.6* more frequently generate decomposition, local-search, and matheuristic-style algorithms, whereas weaker models rely more heavily on direct solver calls, especially *Qwen3-Coder-Plus* (72%) and *LLaMA-4-Maverick* (99%). (3) *Hybrid and search-based methods are often more competitive.* Among frontier models, stronger performance is often associated with matheuristic, hybrid, local-search, or metaheuristic approaches, which suggests that combining construction, improvement, relaxation, and solver-based subroutines, is more effective than simply invoking a generic solver. (4) *Algorithm diversity helps explain QTE differences.* *Claude Opus 4.6* has the most balanced non-monolithic distribution and the best QTE on both subsets in Table 2 (0.31 on Full, 0.32 on Hard), which suggests that part of its advantage comes from choosing algorithmic templates with better quality–runtime tradeoffs.

Failure modes in generated solutions. Figure 3b decomposes one-shot outcomes into correct solutions and four failure modes. We analyze the results and give the insights. (1) *Correctness varies sharply across models.* Frontier models generate much more correct solutions than cost-effective models: *GPT-5.3-Codex*, *Gemini 3.1 Pro*, and *Claude Opus 4.6* solve a substantial fraction of tasks correctly, whereas *LLaMA-4-Maverick* generates correct solutions only for a small set of tasks. (2) *Formulation errors are the dominant failure mode.* For most models, formulation design flaw is the single largest failure category (the clear exception being *Claude Opus 4.6*, discussed below), which indicates that many failures arise before downstream implementation: the model selects an incorrect optimization structure, objective, relaxation, or algorithmic abstraction. (3) *Weaker models fail earlier in the pipeline.* Cost-effective models, especially *Grok-4.20*, *Qwen3-Coder-Plus*, and *LLaMA-4-Maverick*,

Table 2. One-shot performance on FrontierOR. Metric definitions follow Section 4.2.

Model	FrontierOR Full ($n = 180$)				FrontierOR Hard ($n = 50$)			
	Exec. rate	Feasibility	Sol. quality	QTE	Exec. rate	Feasibility	Sol. quality	QTE
Frontier models								
Claude Opus 4.6	0.93	**0.62**	0.48	**0.31**	0.94	0.60	0.44	**0.32**
GPT-5.3-Codex	**0.98**	0.60	0.48	0.26	0.98	0.49	0.30	0.18
Gemini 3.1 Pro	0.93	0.61	**0.52**	0.25	**1.00**	**0.64**	**0.44**	0.22
Cost-effective and open-source models								
DeepSeek-R1	0.74	0.42	0.31	0.17	0.82	0.37	0.20	0.11
Grok-4.20	0.74	0.28	0.22	0.13	0.76	0.20	0.14	0.06
Qwen3-Coder-Plus	0.60	0.26	0.20	0.09	0.52	0.21	0.12	0.07
LLaMA-4-Maverick	0.47	0.18	0.13	0.06	0.52	0.13	0.07	0.02



(a) Solution method frequency and performance per LLM. Bubble size and percentage encode the share of tasks in each solution family, while color shows quality–time efficiency relative to Gurobi.

(b) Failure-mode composition per LLM. Segments show the percentage of tasks classified as a specific failure mode plus correct solutions.

Figure 3. Algorithm pattern distribution and failure mode across models.

show larger percentages of constraint-specification errors and interface/schema violations. These errors reflect difficulties in translating the natural-language task into a valid optimization algorithm, not only in solving it efficiently. (4) *Stronger models shift the bottleneck from formulation to heuristic design.* Compared with other frontier models, *Claude Opus 4.6* has fewer formulation-design failures but more heuristic-search failures, suggesting that it often interprets tasks correctly, while its remaining errors are more often due to insufficient search, refinement, or improvement on large instances. Overall, the failure distribution shifts with model strength: weaker models often fail at formulation, constraint, or schema stages, while stronger models more often fail at the algorithmic-performance stage. This also explains why execution rate alone is insufficient for evaluating LLM-generated optimization algorithms.

5.2. Performance Improvement With Self-Evolution

Direct one-shot generation, even with the strongest frontier models, leaves many hard tasks either unsolved or far below the Gurobi reference (Table 2). In contrast, self-evolving agents can iteratively refine candidate programs using internal feedback, which enables broader exploration of efficient strategies. Due to compute budget constraints, we select the top 40% challenging tasks in the Hard set based on the weakest one-shot performance in feasibility, solution quality, and runtime. We use *GPT-5.3-Codex* as the shared backbone and initialize each framework with the same one-shot program, ensuring that differences arise from search strategy rather than model or initialization.

Efficiency improvement beyond one-shot generation. All three self-evolving frameworks significantly outperform one-shot generation across every metric (Table 3). The sharp rise in solution feasibility shows that during self-evolution the agent can effectively use the returned evaluation feed-

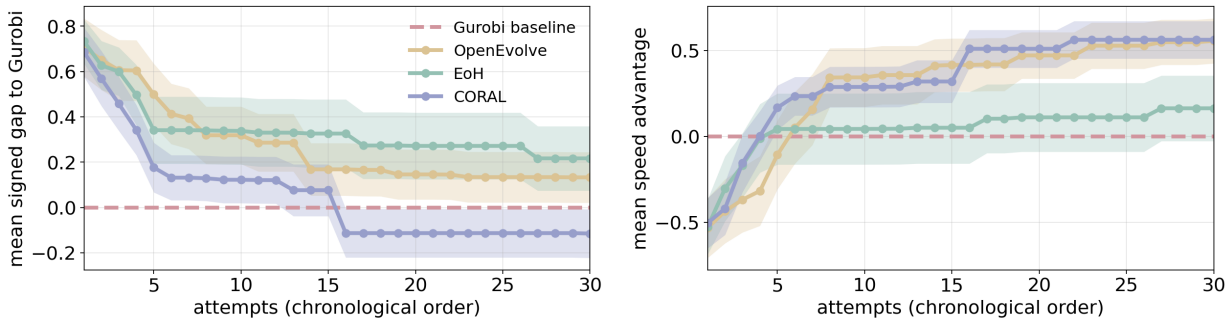


Figure 4. Self-evolution trajectories on the hard tasks. *Left:* best-so-far signed gap to Gurobi (lower is better), where signed gap measures the relative objective difference between a candidate solution and the reference while preserving which one is better. *Right:* best-so-far speed advantage $(\tau_g - t_{\text{solve}})/\tau_g$ (higher is better). Each curve averages across the selected 20 tasks (± 1 SEM).

Table 3. Test-set performance across three self-evolving frameworks.

Method	Exec.	Feas.	Sol. q.	QTE
One-shot	0.80	0.45	0.18	0.15
EoH	0.78	0.72	0.43	0.33
OpenEvolve	**1.00**	0.92	0.61	0.49
CORAL	**1.00**	**1.00**	**0.67**	**0.50**

back to mitigate the scalability bottlenecks. Under the same 30-attempt budget, *CORAL* attains the best solution quality and runtime efficiency, and outperforms the runner-up *OpenEvolve* by 6% and 1%, respectively. *EoH* delivers only limited improvement compared to other evolution systems, which suggests that algorithm-level crossover and mutation guided purely by prompt tends to be unstable, with frequent regressions even on candidates that look promising on the development sets.

Quality–efficiency trajectory dynamics across frameworks. The trajectories in Figure 4 reveal both the similarities and differences in how each framework's solution strategy evolves. All three frameworks rapidly close the gap to Gurobi within the first ~ 5 attempts, before entering a slower regime that depends on harder algorithmic restructuring. The two dimensions also evolve at different rates: in every framework, the mean speed advantage crosses the Gurobi baseline within roughly five attempts, whereas the relative solution gap to Gurobi is harder to break through—only *CORAL* consistently crosses it from around attempt 16 onwards. Among the three frameworks, *EoH* exhibits the largest variance and *CORAL* the smallest, which indicates that a flexible, autonomous self-exploration strategy yields more stable performance for algorithm optimization. We further conduct in-depth case studies on several representative problems to compare the distinct improvement pathways adopted by the three frameworks, and analyze the underlying mechanisms that give rise to these differences; see Appendix E.2 for details.

6. Conclusion

We present **FrontierOR**, a benchmark for evaluating LLMs' ability to design efficient algorithms for realistic large-scale operations research problems. FrontierOR shifts the focus from optimization modeling and solver-code generation to practical algorithm engineering: models must infer problem structure from natural-language specifications, generate executable algorithms, and compete with human-verified Gurobi baselines on both solution quality and computational efficiency.

Our evaluation shows that current LLMs remain far from reliable, scalable algorithm designers. Frontier models often generate executable code, yet many solutions violate feasibility, lose objective quality, or fail to improve over direct Gurobi solving; even the strongest one-shot model reaches only moderate quality–time efficiency, especially on the Hard subset. Self-evolving agent frameworks improve substantially by refining candidate programs with evaluation feedback, highlighting test-time search as a promising direction for LLM-based optimization. Models also diverge in algorithmic strategy and failure modes: stronger models favor hybrid, local-search, or matheuristic designs over a monolithic solver call, and failures stratify by model strength: weaker models fail early on formulation, constraints, or schema; stronger models fail later on search quality at scale.

FrontierOR is necessarily limited by the scope of the OR literature we draw from and the current set of evaluated LLMs. Our benchmark emphasizes tasks with available paper-grounded formulations and reproducible instances, which may underrepresent domains where data or solver implementations are not publicly recoverable. Still, we believe FrontierOR will support future work on LLM and agent systems that move beyond solver-calling toward practical, efficient, and reliable optimization algorithm design at real-world scale.

Impact Statement

This paper introduces a benchmark for evaluating LLM-based algorithm design in large-scale optimization. The intended impact is to support more reliable assessment of AI systems for mathematical modeling, scientific computation, and operations-research problem solving. Potential risks include over-reliance on generated algorithms in high-stakes decision settings; such outputs should be validated by domain experts and checked for feasibility, robustness, and compliance before deployment.

References

AhmadiTeshnizi, A., Gao, W., and Udell, M. Optimus: Scalable optimization modeling with (mi) lp solvers and large language models. *arXiv preprint arXiv:2402.10172*, 2024.

Beasley, J. E. Or-library: distributing test problems by electronic mail. *Journal of the operational research society*, 41(11):1069–1072, 1990.

Burkard, R. E., Karisch, S. E., and Rendl, F. Qaplib–a quadratic assignment problem library. *Journal of Global optimization*, 10(4):391–403, 1997.

Chen, H., Wang, Y., Cai, Y., Hu, H., Li, J., Huang, S., Deng, C., Liang, R., Kong, S., Ren, H., et al. Heurigym: An agentic benchmark for llm-crafted heuristics in combinatorial optimization. *arXiv preprint arXiv:2506.07972*, 2025.

Feng, S., Sun, W., Li, S., Talwalkar, A., and Yang, Y. A comprehensive evaluation of contemporary ml-based solvers for combinatorial optimization. *arXiv preprint arXiv:2505.16952*, 2025.

Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P., Jarck, K., Koch, T., Linderoth, J., et al. Miplib 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3): 443–490, 2021.

Huang, C., Tang, Z., Hu, S., Jiang, R., Zheng, X., Ge, D., Wang, B., and Wang, Z. Orlm: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 73(6):2986–3009, 2025a.

Huang, X., Shen, Q., Hu, Y., Gao, A., and Wang, B. Llms for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 2678–2710, 2025b.

Jiang, C., Shu, X., Qian, H., Lu, X., Zhou, J., Zhou, A., and Yu, Y. Llmopt: Learning to define and solve general optimization problems from scratch. *arXiv preprint arXiv:2410.13213*, 2024.

Jiang, X., Chen, J., Zhang, C., Gao, J., Hu, C., Zhang, C., Wu, Y., and Zhang, Y. Reasoning in a combinatorial and constrained world: Benchmarking llms on natural-language combinatorial optimization. *arXiv preprint arXiv:2602.02188*, 2026.

Kong, M., Qu, A., Guo, X., Ouyang, W., Jiang, C., Zheng, H., Ma, Y., Zhuang, D., Tang, Y., Li, J., et al. Alphaopt: Formulating optimization programs with self-improving llm experience library. *arXiv preprint arXiv:2510.18428*, 2025.

Liang, K., Lu, Y., Mao, J., Sun, S., Yang, C., Zeng, C., Jin, X., Qin, H., Zhu, R., and Teo, C.-P. Large-scale optimization model auto-formulation: Harnessing llm flexibility via structured workflow. *arXiv preprint arXiv:2601.09635*, 2026.

Liu, F., Tong, X., Yuan, M., Lin, X., Luo, F., Wang, Z., Lu, Z., and Zhang, Q. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*, 2024.

Lu, H., Xie, Z., Wu, Y., Ren, C., Chen, Y., and Wen, Z. Optmath: A scalable bidirectional data synthesis framework for optimization modeling. *arXiv preprint arXiv:2502.11102*, 2025.

Michailidis, K., Tsouros, D., and Guns, T. Cp-bench: Evaluating large language models for constraint modelling. *arXiv preprint arXiv:2506.06052*, 2025.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J., Mehrabian, A., et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Qu, A., Zheng, H., Zhou, Z., Yan, Y., Tang, Y., Ong, S. Y., Hong, F., Zhou, K., Jiang, C., Kong, M., et al. Coral: Towards autonomous multi-agent evolution for open-ended discovery. *arXiv preprint arXiv:2604.01658*, 2026.

Ramamonjison, R., Yu, T., Li, R., Li, H., Carenini, G., Ghaddar, B., He, S., Mostajabdaveh, M., Banitalebi-Dehkordi, A., Zhou, Z., et al. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In *NeurIPS 2022 competition track*, pp. 189–203. PMLR, 2023.

Reinelt, G. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

Sharma, A. Openevolve: an open-source evolutionary coding agent. *GitHub repository. Accessed*, pp. 08–10, 2025.

Sun, W., Feng, S., Li, S., and Yang, Y. Co-bench: Benchmarking language model agents in algorithm search for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 33126–33134, 2026.

Tang, J., Zhang, Q., Li, Y., Chen, N., and Li, J. Grapharena: Evaluating and exploring large language models on graph computation. *arXiv preprint arXiv:2407.00379*, 2024.

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., and Subramanian, A. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858, 2017.

Xiao, Z., Zhang, D., Wu, Y., Xu, L., Wang, Y. J., Han, X., Fu, X., Zhong, T., Zeng, J., Song, M., et al. Chain-of-experts: When llms meet complex operations research problems. In *The twelfth international conference on learning representations*, 2023.

Yang, Z., Wang, Y., Huang, Y., Guo, Z., Shi, W., Han, X., Feng, L., Song, L., Liang, X., and Tang, J. Optibench meets resocratic: Measure and improve llms for optimization modeling. *arXiv preprint arXiv:2407.09887*, 2024.

Zhou, C., Xu, T., Lin, J., and Ge, D. Steporlm: A self-evolving framework with generative process supervision for operations research language models. *arXiv preprint arXiv:2509.22558*, 2025.

Appendix

A. Detailed Benchmark Construction Protocol

A.1. Source Selection and Reproduction Pipeline

For each selected paper, the optimization problem, test data, and reference solution methods are extracted and reproduced by Claude Code (Opus 4.6) through a three-stage pipeline:

1. **Formulation extraction.** The original mathematical formulation of the optimization problem—sets, parameters, variables, objective, and constraints—is identified from the paper and transcribed in LaTeX as the single authoritative reference for construction and verification. When the paper presents multiple alternative formulations, we select the formulation that is most central to the computational study. When the paper explicitly gives a solver-ready reformulation, we also record it as an implementation reference.
2. **Instance specification extraction.** The configurations of all test-instance sets used in the paper's computational experiments are extracted into a machine-readable JSON specification. When instances rely on external and accessible benchmarks, their configurations are retrieved and cross-checked to reproduce the scale and structure used in the source paper.
3. **Solver implementation.** Based on the solver-ready formulation, we implement a Gurobi program that returns solutions expressed in the original decision variables. If the solver uses auxiliary reformulation variables internally, the output is projected back to the variables of the original formulation. When the paper omits a solver-ready formulation, we derive one from the mathematical model and document any necessary assumptions.

A.2. Natural-Language Problem Description

The problem description is the primary input presented to the LLMs under evaluation. It conveys all information necessary to design a correct solution method, without revealing mathematical formulas or algorithmic hints. Each description is synthesized from three sources: (i) the original paper, which provides the application context and operational semantics; (ii) the original mathematical formulation, which canonically defines the modeling rules; and (iii) the instance and solution schemas, which anchor the input and output formats.

The description follows the following style constraints:

- The text is continuous natural-language prose. No LaTeX formulas, numbered lists, section headers, or structural markup are included in the LLM-facing specification.
- Mathematical relationships are verbalized rather than written symbolically; for example, we write *each machine serves at most one product per period* rather than $\sum_i x_{ijt} \leq 1$.
- Optimization-modeling jargon such as *decision variable*, *constraint*, or *objective function* is replaced by operational language, e.g., *the company must decide...* or *the goal is to minimize....*
- Common-sense mathematical properties, such as non-negativity of physical quantities or integrality of yes/no decisions, are omitted unless they constitute essential modeling requirements that cannot be inferred from the business context.

A.3. Test Instances and Ground Truth

Each task includes two categories of test instances. **Tiny instances** are minimal-scale instances, typically solvable by Gurobi in seconds, that serve as a lightweight functional gate: They verify that an LLM-generated program executes, parses input and output correctly, and can produce at least one feasible solution before consuming computational budget on large instances. **Real-world large instances** are the primary scoring targets. Their scale is determined by expert reasoning grounded in the operational context and by the experimental configurations of the source paper. Where possible, we diversify structural characteristics at comparable nominal scales—for example, spatial layouts, graph topologies, or capacity-to-load ratios—because such changes can alter Gurobi runtime by orders of magnitude even when the MIP size is similar.

The Gurobi baseline is obtained by running the human-verified solver implementation of the mathematical formulation within the benchmark time budget. It yields a certified optimal objective when Gurobi proves optimality and the best feasible

objective otherwise. The output includes the full solution vector, objective value, solver status, and solve time. A standalone Python feasibility checker verifies every hard constraint, variable domain, and variable bound against a candidate solution and reports any violation and its magnitude. To prevent LLMs from reporting an objective inconsistent with their submitted solution, every checker also re-computes the objective from the reported decision variables and uses that re-computed value.

B. Quality-Assurance Details

Automated cross-verification. The feasibility checker is run against every Gurobi solution on every instance. Any reported violation triggers a diagnosis-and-repair loop. Observed issues fall into two broad categories: (i) *checker-side errors*, such as overly tight numerical tolerances, misimplemented constraints, or mismatched parsing logic between the checker and the solver; and (ii) *solver-side errors*, such as omitted constraints, incorrect objective terms, or ill-scaled formulations in the Gurobi code. Solver-side fixes require re-solving the affected instances. After iterative repair, all retained Gurobi solutions pass the feasibility check.

Expert review and re-verification. We engaged 15 OR experts who, over a three-week period, independently audited the complete component suite of each paper—mathematical formulation, problem description, Gurobi code, and feasibility checker implementation—against the original publication. The review followed a structured checklist that covered:

- **Mathematical formulation fidelity:** whether the extracted model is correct and complete, with all sets, parameters, variables, objective terms, and constraints faithfully extracted without omissions or redundancies.
- **Problem-description quality:** whether every element of the mathematical model has a corresponding natural-language description that is complete, unambiguous, and does not admit multiple reasonable interpretations.
- **Gurobi-code alignment:** whether the solver implementation is strictly consistent with the mathematical formulation or with an explicit reformulation used for solver tractability, and whether unavoidable defaults such as penalty coefficients or big-M values are reasonable.
- **Feasibility-checker scope:** whether the checker verifies exactly the hard constraints, variable domains, and variable bounds of the mathematical model, while excluding soft constraints, penalties, and heuristic rules.

The main errors flagged by experts concerned symbol and indexing errors in the mathematical formulation, incomplete or ambiguous natural-language descriptions, missing constraints or unauthorized simplifications in the solver code, and vacuous or incomplete feasibility checks. After several rounds of repair and re-verification, we obtained a clean final benchmark version.

C. Additional Dataset Statistics

This appendix complements the overview in Section 3.1 with detailed statistics on the benchmark's publication venues, temporal distribution (year of publication), and algorithmic diversity (the families of solution methods proposed by the source papers), shown in Table 4.

Table 4. Publication venues and algorithmic approaches of collected papers.

Venue	%	Algorithmic Approach	%
INFORMS Journal on Computing	33	Branch-and-Price / Column Generation	29
European Journal of Operational Research	17	Branch-and-Cut	18
Transportation Science	13	Metaheuristics / Matheuristics	13
Operations Research	13	Benders Decomposition	12
Computers & Operations Research	9	Tight Formulation / Other Exact	10
Management Science	3	Branch-and-Bound (Mixed-Integer / Semidefinite)	10
Mathematical Programming / Computation	1	Dynamic Programming / Decision Diagrams	4
Other	12	Other Decomposition (Lagrangian, ADMM, etc.)	3

Temporal distribution. The collection skews recent: 33 papers (18%) predate 2010, 19 papers (11%) span 2010–2014, 59 papers (33%) fall in 2015–2019, and 69 papers (38%) were published in 2020 or later. This distribution reflects the rapid growth of algorithm engineering for optimization in response to powerful commercial solvers such as Gurobi and CPLEX, which serve as baselines in many collected papers.

Algorithmic diversity. Branch-and-price / column generation (29%) and branch-and-cut (18%) are the two most prevalent paradigms. Decomposition methods—Benders decomposition (12%), Lagrangian relaxation (2%), and ADMM / stochastic decomposition (<1%)—together account for 15% of the collection. Metaheuristics and matheuristics (13%) and dynamic-programming or decision-diagram methods (4%) add further diversity in algorithm-design style.

D. Continuous Measures of Solution Quality and Quality-Time Efficiency

The metric *solution quality* and *QTE* reported in Table 2 are thresholded binary pass rates: a (paper, instance) case counts as a pass only if the generated program matches the Gurobi reference within the 1% gap window (and for *QTE* also within Gurobi's solve-time budget). These 0/1 indicators are easy to aggregate but discard the magnitude information per task. This section instead adopts continuous counterparts of *solution quality* and *QTE* to supply the insufficient measure. For each (paper, instance) pair, we compare a subject program X against a reference program Y which may be the Gurobi reference, or a program generated by a different LLM.

To compute the continuous solution quality Δq, we fix a common time budget T and compare the objectives the two programs reach within it. Let $o_X(T), o_Y(T)$ be their best objective values at wall-clock time T and $\delta \in \{+1, -1\}$ the optimization direction ($+1$ for minimization, -1 for maximization). Since quality gaps are typically within a few percent, we use a $\max$-normalized signed difference, bounded in $[-1, 1]$ at any scale:

$$\Delta q \;=\; \delta \cdot \frac{o_Y(T) - o_X(T)}{\max(|o_X(T)|,\, |o_Y(T)|)} \;\in\; [-1, 1]. \tag{1}$$

To compute the continuous *QTE*, we instead fix a common quality target Q and compare the time each program needs to reach it. Let $t_X(Q), t_Y(Q)$ be the wall-clock times at which X and Y first attain an objective at least as good as Q. Since solve times can span several orders of magnitude, we apply a squashing transform ϕ to the time ratio $r = t_Y(Q)/t_X(Q)$ where $\log$ places the ratio on a multiplicative scale and $\arctan$ bounds it to $(-1, 1)$:

$$\Delta t \;=\; \phi\!\left(\frac{t_Y(Q)}{t_X(Q)}\right) \;\in\; (-1, 1), \qquad \phi(r) = \tfrac{2}{\pi}\arctan(\log r). \tag{2}$$

D.1. Overall Comparison Against the Gurobi Reference

We evaluate the continuous magnitude of solution quality and efficiency on every large-instance case when the generated program returns a feasible solution, as shown in Table 5. Here, Δq is computed by comparing each program's *final* objective $o_X(T)$ against Gurobi reference's final objective o_g within the common time budget 1 hour; Δt is computed by comparing the runtime $t_X(Q)$ that the LLM-generated program needs to reach the shared quality target Q (an objective within 1% of o_g) against Gurobi's solve time τ_g, excluding large-instance cases in which X never reaches Q.

We identify three patterns hidden by the binary Sol.,quality and QTE columns of Table 2. (1) *Within the one-hour budget, frontier models are essentially tied with Gurobi on quality.* The three frontier models, *Claude Opus 4.6*, *GPT-5.3-Codex*, and *Gemini 3.1 Pro*, all achieve $\overline{\Delta q}$ within ± 0.025 of zero (0.001, 0.014, and 0.021, respectively), corresponding to an average objective within a few percent of o_g. The binary Sol. quality column reports these three models at 0.48–0.52, indicating that roughly half of feasible cases clear the strict 1% gap while the remaining cases miss the bar by only a small margin. (2) *LLM-generated programs are substantially faster than Gurobi reference when reaching* 1% *quality.* $\overline{\Delta t}$ ranges from 0.41 to 0.66 across all seven models, and stays in 0.58–0.66 for the four leading models, values that under Eq. (2) correspond to roughly 3×–5× wall-clock speedups to reach the same 1% quality bar. (3) *The performance of cost-effective models can sometimes match or even surpass frontier models.* On Full set, *DeepSeek-R1* matches *GPT-5.3-Codex* exactly on $\overline{\Delta q}$ (both at 0.014) and reaches a comparable level on $\overline{\Delta t}$ (0.603 vs. 0.645); on Hard set, *Grok-4.20* is the only model with a positive $\overline{\Delta q}$ (+0.014), and *DeepSeek-R1* surpasses most frontier models on $\overline{\Delta t}$ (0.637). However, these results do not imply that cost-effective models are competitive overall, since each model's mean is computed over its own feasible cases, which differ in both size and difficulty. Removing this coverage confound requires fixing a common task set for each comparison, which we examine further in the pairwise analysis in Section D.2.

Table 5. Continuous *solution quality* and *QTE* relative to the Gurobi reference, averaged over each method's feasible (paper, instance) pairs. Higher is better in both columns.

Model / Method	**FrontierOR Full** ($n = 180$)		**FrontierOR Hard** ($n = 50$)	
	$\overline{\Delta q}$	$\overline{\Delta t}$	$\overline{\Delta q}$	$\overline{\Delta t}$
Frontier models				
Claude Opus 4.6	0.001	0.627	−0.016	0.584
GPT-5.3-Codex	0.014	0.645	−0.030	0.639
Gemini 3.1 Pro	0.021	0.609	−0.036	0.633
Cost-effective models				
DeepSeek-R1	0.014	0.603	−0.042	0.637
Grok-4.20	0.005	0.658	0.014	0.612
Qwen3-Coder-Plus	−0.017	0.414	−0.056	0.426
LLaMA-4-Maverick	−0.011	0.458	−0.109	0.367

D.2. Pairwise Comparison Between Models

In Table 5 each model's solution quality and efficiency are averaged over its *own* feasible set, we further compare models *pairwise* on their shared tasks. For each pair (A, B) we restrict to their *shared feasible set* S_{AB} where both models produce a feasible solution. On each case $i \in S_{AB}$ take the subject X to be model A and the reference Y to be model B. For Δq the shared time budget is 1 hour. and we compare the objectives they reach. For Δt the shared quality target is the *worse* of the two models' final objectives and we compare the time each takes to reach it. Averaging the per-case scores over S_{AB} gives Δq_{AB} and Δt_{AB} and a positive value means A outperforms B.

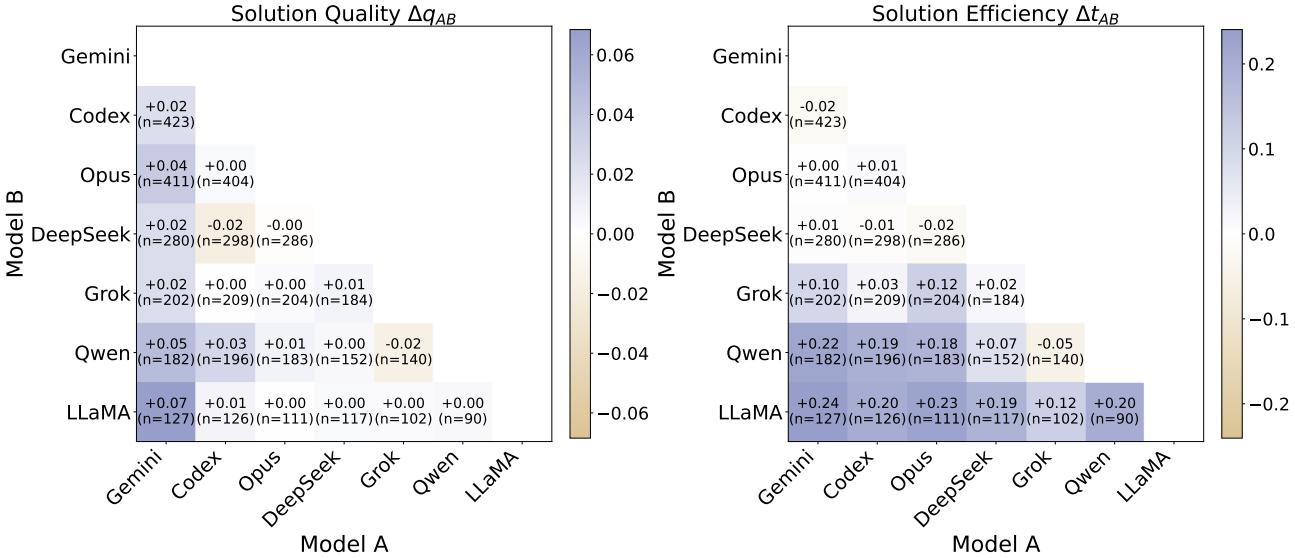


Figure 5. Pairwise comparison on shared feasible large instances of tasks. Each case at Model A column, Model B row reports the average performance difference over S_{AB}; positive (purple) indicates that model A (column) outperforms model B (row).

We demonstrate our new insights based on Figure 5. (1) *Conditional on both models solving the same task, solution-quality gaps are small, with* $|\overline{\Delta q}| < 0.05$ *for every pair except* Gemini 3.1 Pro *versus* LLaMA-4-Maverick. The larger aggregate differences in Table 2 are therefore driven more by *coverage*, namely which tasks each model can solve, than by per-task quality gaps on jointly solved tasks. (2) *Runtime efficiency separates the models more sharply than solution quality, and frontier models reach the shared quality target faster than cost-effective ones.* All three frontier columns sit at $\overline{\Delta t} \in [+0.18, +0.24]$ against the *Qwen3-Coder-Plus* and *LLaMA-4-Maverick* rows, indicating that when both models eventually match objectives, the frontier model gets there several-fold faster. Within the frontier tier itself the three leading

models are effectively tied on speed ($|\overline{\Delta t}| \leq 0.03$ on each pairwise cell), so the QTE re-ordering in Table 2 is driven by task coverage rather than by per-task speed gaps within the top tier. (3) *Cost-effective models are not uniformly worse once they solve a task. DeepSeek-R1* matches or slightly outperforms *GPT-5.3-Codex* and *Claude Opus 4.6* in both quality and speed on their shared tasks ($\overline{\Delta q} = -0.019$, $\overline{\Delta t} = -0.009$ against GPT-5.3-Codex; near-zero against Claude), so its lower aggregate performance mainly reflects lower task coverage. *LLaMA-4-Maverick* is the clear exception: every other model reaches the shared quality target 0.12–0.24 faster on jointly solved tasks, indicating weaknesses in both coverage and per-task efficiency.

E. Task Examples and Case Studies

E.1. One-Shot Generation Success and Failure Cases

We select a few representative tasks from FrontierOR and each task is a suite of components comprising a natural-language *Problem Description*, a *Mathematical Formulation* faithful to the source paper, a large-scale *Test Instance* configured following the paper, a Gurobi-based *Solver Code* with its *Reference Solution* as the comparison baseline, and a standalone *Feasibility Checker*. On these task examples, we further illustrate both the success and failure patterns of optimization programs generated by different LLMs, as well as the evolution trajectories of three selected self-evolve frameworks refining an initial program generated by GPT-5.3-Codex.

Task 1 overview. This example is from Bront, Méndez-Díaz, and Vulcano (2009), "A Column Generation Algorithm for Choice-Based Network Revenue Management," *Operations Research*. The task is airline offer-set selection under a multinomial-logit choice model: the airline decides, for every possible offer set, how many periods of the booking horizon to display it. The formulation is the CDLP master LP—one continuous variable per offer set, hence exponentially many primal variables that require column generation. The large instance includes 168 products, 12 legs, and 84 market segments over a 2000-period horizon, leading to a flat LP with $2^{168} - 1$ columns, making only column generation tractable. The Gurobi reference runs the column-generation loop (master LP, dual extraction, greedy pricing with an exact mixed-integer fallback) to the proven optimum. The feasibility checker validates per-leg capacity, horizon, and non-negativity.

Task 1 (*bront2009*): Choice-Based Network Revenue Management (CDLP)

Problem Description. *An airline operates a network of flight legs with fixed seat capacities. Products are itinerary-and-fare-class combinations, each consuming one seat per leg on its route.... Customers arrive over a discrete booking horizon and belong to market segments with multinomial-logit preferences.... The airline chooses, for every possible offer set, the number of periods to display it, so as to maximize total expected revenue subject to per-leg capacity and the horizon length....*

Mathematical Formulation

$2^n - 1$ primal variables, one per nonempty S:

$$\max \sum_{S \subseteq N} \lambda\, R(S)\, t(S)$$

$$\text{s.t.} \sum_{S} \lambda\, Q_i(S)\, t(S) \le c_i,\ \forall i$$

$$\sum_{S} t(S) \le T, \quad t(S) \ge 0$$

where $R(S) = \sum_{j \in S} r_j P_j(S)$, $P_j(S) = \sum_l p_l v_{lj} / (\sum_{h \in C_l \cap S} v_{lh} + v_{l0})$, and $Q_i(S)$ is leg-i consumption. At most $m+1$ offer sets are active at the optimum, so column generation augments an active set until no offer set has positive reduced cost.

Feasibility Checker

```
def check_feasibility(instance, solution):
    tol = 1e-5
    violations = []
    # each column = (offer set S, time t(S))
    cols = extract_columns_and_times(solution)
    # C1: per-leg expected capacity usage
    #     sum_S lambda * Q_i(S) * t(S) <= c_i
    cap = compute_capacity_usage(cols)
    for i, c_i in enumerate(c):
        if cap[i] - c_i > tol:
            violations.append((1, i))       # leg i
     over capacity
    # C2: horizon   sum_S t(S) <= T
    if sum(t for _, t in cols) - T > tol:
        violations.append(2)
    # C3: non-negativity   t(S) >= 0
    if any(t < -tol for _, t in cols):
        violations.append(3)
    return {"feasible": not violations,
            "violations": violations}
```

Test Instance

```
"network": { "num_legs": 12 },
"products":  [ 168 items ],
"segments":  [ 84  items ],
"booking_horizon": { "T": 2000 },
"lambda": 1.0,
"instance_id": "bront2009/large_11"
```

Solver Code

```
cols = [init_offer_set(prob)]
while time_left():
    m = gp.Model("CDLP_master")      # master LP
    t = m.addVars(len(cols), lb=0.0)
    cap = [m.addConstr(gp.quicksum(
            lam*Q[i][s]*t[s] for s in t) <= c[i])
           for i in legs]
    hor = m.addConstr(gp.quicksum(t[s] for s in t)
     <= T)
    m.setObjective(gp.quicksum(
        lam*R[s]*t[s] for s in t), GRB.MAXIMIZE)
    m.optimize()
    pi, sigma = [c.Pi for c in cap], hor.Pi
    S, rc = greedy_heuristic(pi, sigma)
    if rc <= 0: S, rc = exact_mip_subproblem(pi,
     sigma)
    if rc <= 0: break
    cols.append(frozenset(S))
```

Reference Solution

```
"objective_value": 243985.56,
"active_columns": [
  { "offer_set":  [1,2,3,5,...,166],
    "time_allocated": 279.50 },
  ... 12 more rows ... ]
```

Among the LLM-generated algorithms for Task 1, Case 1 shows that **failures arise when heuristic pricing misses valid columns and leaves the master LP infeasible, while successes use exact pricing that enumerates all improving columns and keeps the master feasible.** Both models build the same CDLP master LP and add columns via a pricing subproblem; the divergence lies in how pricing terminates. *GPT-5.3-Codex* runs a 2-opt local search and exits as soon as it reports no positive reduced cost. The resulting active set is too thin on large instances, and the LP allocation violates leg capacity in around 40% of them. *Claude-Opus-4.6* instead enumerates all subsets per segment with de-duplication (local-search fallback beyond budget), terminating only at an optimality certificate and matching Gurobi with a near-zero average gap.

Case 1: heuristic-only pricing (Failure) vs. exact pricing (Success)

(a) GPT-5.3-Codex – 40% infeasible solutions

```
def pricing_heuristic(pi, sigma):      # <-(a)
    w = [fare[j] - pi @ A[:,j] for j in prods]
    seeds = topk_positive(w) + segment_sets(w)
    return best_local_search(seeds, w)  # 2-opt
     only
m = gp.Model("CDLP_master")             # master LP
t = m.addVars(num_cols, lb=0.0)
cols = [single_product_sets, all_products]
while time_left():
    m.optimize()
    pi, sigma = duals(legs, horizon)
    S, rc = pricing_heuristic(pi, sigma)
    if not S or rc <= 0: break          # <-(b)
    add_column(S)
return active_columns(m)                # <-(c)
```

(b) Claude-Opus-4.6 – 100% feasible, avg gap ≈ 0

```
def solve_pricing(pi, sigma):
    if num_products small:                  # <-(a)
        return enumerate_subsets(pi, sigma)  #
     exact
    return segment_local_search(pi, sigma)   #
     <-(b)
def add_column(S):                          # <-(c)
    if S and S not in seen:                # de-dup
        seen.add(S); columns.append(S)
while time_left():
    pi, sigma = solve_master_lp(columns)  # <-(d)
     Gurobi RMP
    S = solve_pricing(pi, sigma)
    if reduced_cost(S, pi, sigma) <= 0: break
    add_column(S)
```

[!] Failure root cause

Pricing is heuristic-only (a); the loop exits as soon as the heuristic reports no improving column (b), but it misses valid columns. On large_41/51 the master's active set is too thin, so the reported allocation (c) overflows leg capacity.

[+] Why it works

Pricing is exact by subset enumeration on segment-decomposed subproblems (a), with local search only as the large-scale fallback (b); every column (c) fed into the Gurobi master (d) is valid, so the master stays feasible and matches the reference.

Task 2 overview. This example is from Schwerdfeger and Walter (2016), "A Fast and Effective Subset-Sum Based Improvement Procedure for Workload Balancing on Identical Parallel Machines," *Computers & Operations Research*. The task is workload balancing across identical parallel machines (P || NSSWD): jobs with integer processing times are assigned to machines so as to minimize the normalized sum of squared workload deviations. Because the average completion time is a fixed instance constant, the problem reduces to a binary quadratic program (NP-hard by reduction from PARTITION). The large instance assigns 280 jobs with processing times in $[1, 300]$ to 14 machines, giving 14×280 binary variables whose quadratic objective makes the monolithic MIQP hard to close at scale. The Gurobi reference is a direct MIQP—binary x_{ij}, auxiliary C_i, the assignment constraint, and a quadratic objective—returning the certified optimum; the feasibility checker recomputes each C_i from the assignment and verifies the assignment cardinality and binary domain.

Task 2 (*feizollahi2016*): Workload Balancing on Identical Parallel Machines (P || NSSWD)

Problem Description. *A facility schedules independent jobs across at least two identical parallel machines, with more jobs than machines. Each job has a known positive integer processing time and goes to exactly one machine; a machine's completion time is the total processing time it receives.... The goal is to assign every job so as to minimize the normalized sum of squared workload deviations from the average completion time....*

Mathematical Formulation

Binary $x_{ij} = 1$ iff job j is on machine i, auxiliary C_i:

$$\min \sum_{i=1}^{m} C_i^2$$
$$\text{s.t.} \sum_{j=1}^{n} p_j\, x_{ij} = C_i,\ \forall i$$
$$\sum_{i=1}^{m} x_{ij} = 1,\ \forall j$$
$$x_{ij} \in \{0, 1\}.$$

A valid lower bound is $L(m, \mu)$ from the rounded-μ allocation; optimality is certified by $C_{\max} - C_{\min} \leq 1$.

Feasibility Checker

```
def check_feasibility(instance, solution):
    tol = 1e-5
    assign = solution["assignment"]
    C      = solution["machine_completion_times"]
    # C1: workload    sum_j p[j]*x_ij == C_i
    for i, jobs in assign.items():
        if abs(sum(p[j] for j in jobs)
                - C[i]) > tol:
            violations.append((1, i))
    # C2: each job assigned to exactly one machine
    cnt = [0]*n
    for jobs in assign.values():
        for j in jobs: cnt[j] += 1
    if any(c != 1 for c in cnt):
        violations.append(2)
    # C3: binary domain (implicit from format)
    return {"feasible": not violations}
```

Test Instance

```
"num_machines": 14,
"num_jobs":     280,
"processing_time_range": [1, 300],
"average_machine_completion_time": ~3010,
"processing_times":
  [300, 298, 296, 293, 293, 293,
   292, 290, ... 280 values ...]
```

Solver Code

```
model = gp.Model("nsswd")
x = model.addVars(m, n, vtype=GRB.BINARY)
C = model.addVars(m, lb=0.0)
for i in range(m):
    model.addConstr(
        gp.quicksum(p[j]*x[i,j] for j in range(n))
        == C[i])
for j in range(n):
    model.addConstr(
        gp.quicksum(x[i,j] for i in range(m)) == 1)
model.setObjective(
    gp.quicksum(C[i]*C[i] for i in range(m)),
    GRB.MINIMIZE)
```

Reference Solution

```
"objective_value": 5.37e-4,
"machine_completion_times":
  { "0": 2858, "1": 2858, "2": 2858,
    "3": 2857, ..., "13": 2857 },
"assignment":
  { "0": [0, 11, 47, 84,  96, 117, ...],
    "1": [17, 18, 19, 54, 80, 148, ...],
    ... 12 more machines ... }
```

Among the LLM-generated algorithms for Task 2, Case 2 shows that **the failure unconditionally invokes the solver on large-scale MIQPs, while the success first secures a feasible solution via a heuristic and uses exact solving only as an optional refinement.** Both models construct the same quadratic model, while GPT-5.3-Codex's failure lies in its solution being algorithmically correct but failing to account for problem scale in the solving strategy, causing the solver to time out on large-scale instances and return a poor incumbent with a large optimality gap. In contrast, Gemini-3.1-Pro demonstrates a more mature design: it establishes a quality lower bound via a heuristic, and invokes the exact solver only when resource conditions permit (i.e., the problem size is small enough and sufficient time remains); it further accelerates convergence via symmetry-breaking constraints, thereby matching the reference solution quality on all large-scale instances while finishing faster.

Case 2: unguarded MIQP (Failure) vs. guarded warm-started MIQP (Success)

(a) GPT-5.3-Codex – avg relative gap ≈310%

```
asg, loads = lpt_initial(p, m)
asg = local_search_relocate(asg, loads)
# MIQP, built unconditionally          # <-(a)
m = gp.Model("balance")
x = m.addVars(n, M, GRB.BINARY); C = m.addVars(M)
m.setObjective(sum(C[i]*C[i]), GRB.MINIMIZE)
warm_start(x, asg); sym_break(x)
m.optimize()                           # <-(b)
    stalls
```

[!] Failure root cause

The MIQP is built and solved regardless of problem size (a). On the large instances the solver barely converges and returns a poor incumbent (b).

(b) Gemini-3.1-Pro – avg gap ≈0, ∼534s faster

```
asg = local_search(lpt_initial(p, m))     # <-(a)
if n*m <= 2e6 and time_left > 4:          # <-(b)
    guard
    build_miqp(); warm_start(x, asg)      # <-(c)
    add_sym_break_block()                 # <-(d)
    model.optimize(callback)
return best_of(miqp, asg)
```

[+] Why it works

A near-optimal LPT+local-search warm-start (a) is computed first; the MIQP runs only behind a size/time guard (b), warm-started (c) and symmetry-broken (d), converging with a certificate and beating Gurobi on wall time.

Task 3 overview. This example is from Carvalho (1999), "Exact Solution of Bin-Packing Problems Using Column Generation and Branch-and-Bound," *Annals of Operations Research*. The task is one-dimensional bin packing / cutting stock: pack integer-sized items into the fewest identical capacity-W bins. The paper uses an arc-flow / cutting-pattern decomposition solved by column generation; the Gurobi reference builds the arc-flow model and minimizes the bin (feedback-arc) flow. The large instance packs about 25,000 items of size 250–500 into bins of capacity 1000 (optimum 8334 bins), so the arc-flow model spans thousands of vertices and integer flow variables. The feasibility checker verifies every bin respects capacity, the full demand is packed, and the reported bin count is a valid non-negative integer.

Task 3 (*carvalho1999*): One-Dimensional Bin Packing / Cutting Stock

Problem Description. *A facility has unlimited identical bins of integer capacity W. The input gives each item size and the demand (number of copies) per size; every size is a positive integer at most W.... Each item goes to exactly one bin and the total size in any bin cannot exceed W; all demand must be packed.... The goal is to use as few bins as possible....*

Mathematical Formulation

Arc-flow on $V = \{0, \dots, W\}$; integer flow x_{ij} on item/loss arcs, z = bins:

$$\min z$$

$$\text{s.t.} \sum_{(i,j)} x_{ij} - \sum_{(j,k)} x_{jk} = \begin{cases} -z, & j=0 \\ 0, & 0<j<W \\ z, & j=W \end{cases}$$

$$\sum_{(k,k+w_d)} x_{k,k+w_d} \geq b_d, \ \forall d$$

$$x_{ij} \in \mathbb{Z}_{\geq 0}, \ z \geq 0.$$

Item arcs span a size w_d; loss arcs $(k, k{+}1)$ absorb waste. The feedback flow z from W to 0 counts used bins.

Feasibility Checker

```
def check_feasibility(instance, solution):
    W = instance["bin_capacity"]
    bins = solution["bin_assignments"]
    # C1: each bin within capacity
    for b in bins:
        if sum(b) > W:
            violations.append((1, b))
    # C2: full demand packed (multiset match)
    if Counter(x for b in bins for x in b) \
       != demand_counter:
        violations.append(2)
    # C3: z = num_bins non-negative integer
    return {"feasible": not violations}
```

Test Instance

```
"instance_class": "triplet",
"bin_capacity": 1000,
"num_items":   25002,
"num_bins_optimal": 8334,
"item_size_range": [250, 500],
"items": [480, 259, 261, 421,
          ... 25002 values ... ]
```

Solver Code

```
m = gp.Model("arcflow")
x = add_arc_vars(m, item_arcs + loss_arcs)
z = m.addVar(vtype=GRB.INTEGER)    # bins
for v in nodes:                    # flow
     conservation
    m.addConstr(inflow(v) - outflow(v)
                == rhs(v, z))
for d, (w_d, b_d) in enumerate(sizes):    # demand
    m.addConstr(gp.quicksum(
        x[k, k+w_d] for k in starts(w_d)) >= b_d)
m.setObjective(z, GRB.MINIMIZE)
m.optimize()
```

Reference Solution

```
"objective_value": 8334,
"num_bins": 8334,
"status": "optimal",
"bin_assignments":
  [ [480, 259, 261], [421, 312, 267],
    ... 8332 more bins ... ]
```

Among the LLM-generated algorithms for Task 3, Case 3 shows that **failure stems from MIP re-optimizing too narrow a slice of the heuristic to reach the optimum, while success stems from a full MIP anchored by a lower bound and warm-started from the heuristic, which closes the optimality gap.** Both models start from a fast packing heuristic; the divergence lies in how the solver is used. *GPT-5.3-Codex* only re-optimizes a small slice of the bins with a restricted MIP, so its reported bin count stays above the acceptance threshold on the large instances and no solution is accepted. *Gemini-3.1-Pro* instead computes an L_2 lower bound and warm-starts a full set-partition MIP from the heuristic packing, which lets the solver prune early, reach the reference solution quality on the large instances, and finish faster.

Case 3: local MIP polish (Failure) vs. bound-guided MIP (Success)

(a) GPT-5.3-Codex – gap > tiny-instance threshold

```
bins = best_fit_packing(items, W)          # <-(a)
bins = improve_by_bin_elimination(bins)    # <-(b)
# mip_polish on a small slice only         # <-(c)
m = gp.Model("bin_packing")
... limited bins / time budget ...
m.optimize()
return reported_bins                       # <-(d)
```

(b) Gemini-3.1-Pro – avg gap ≈4%, ∼368s faster

```
L2 = compute_L2(sizes, counts, W)          # <-(a) bound
bins = fast_bfd_flat(items, W)             # <-(b) BFD
m = gp.Model("BPP")                        # <-(c)
    set-part.
for j: y[j].Start = 1                      # warm-start
for i,j: x[i,j].Start = bfd_counts         # <-(d)
m.optimize(callback)                       # prune at L2
```

[!] Failure root cause

A best-fit packing (a) plus bin-elimination moves (b) is refined by a MIP restricted to a small slice (c). The polish is too local to reach the optimum, so the reported bin count (d) stays above threshold even on a tin y instance.

[+] Why it works

An L_2 lower bound (a) plus a BFD heuristic (b) warm-start a full set-partition MIP (c,d); the bound lets Gurobi prune early, closing the gap to ~ 0.04 and finishing ~ 368 s faster than the reference.

E.2. Test-Time Self-Evolution Improvement

To complement the aggregate trajectory analysis in Section 5.2, we trace, per paper, how each framework's candidate program evolves across the 30-attempt budget. We organise the case studies along three axes that distinguish the frameworks' improvement pathways:

- **Depth** (iterative refinement) – the agent perturbs an existing program (parameter retuning, constraint reformulation, neighbourhood widening) while preserving its overall structure, characteristic of *OpenEvolve*'s MAP-Elites archive that compounds small in-family edits over the program database.
- **Breadth** (algorithmic exploration) – the agent abandons the current program family and proposes a substantively new formulation or solution method (e.g., switching from a monolithic MIP to a Lagrangian decomposition), characteristic of *EoH*'s explore-style prompt operators (E1/E2) that jointly evolve code and the natural-language thoughts behind it.
- **Migration** (cross-agent reuse) – the agent imports and adapts a program developed by a sibling agent through the shared memory, characteristic of *CORAL*'s multi-agent collaboration.

OpenEvolve on *bront2009* (airline offer-set selection): incremental refinement of column-generation pricing. The seed program implements column generation with a simple flip-only local-search pricing subroutine. Over thirty MAP-Elites iterations, OpenEvolve preserves the column-generation skeleton but accumulates several structural improvements to the pricing inner loop: it replaces full objective recomputation with incremental delta evaluation, separates reduced-cost computation from the heuristic search, spawns multiple initial offer sets per iteration to broaden the pricing search, and adds an enumeration fallback for sufficiently small subproblems. The dev-set score climbs from zero at the seed to a strong combined score by the final iteration. The trajectory exemplifies the *Depth* pathway: each accepted child is a small, in-family edit, and MAP-Elites' diversity-preserving archive keeps both the original and the new pricing variants alive, allowing the LLM to compose them into a faster but algorithmically familiar program.

EoH on *feizollahi2016* (sum-of-squared-loads scheduling): jumping to a different solution-method family. The seed solves the load-balancing problem with an LPT initial assignment followed by a relocate-only local search. Within five generations, EoH surfaces a structurally distinct improvement: the best program retains a randomized greedy construction and an enriched local search, and additionally uses a mixed-integer quadratic refinement step that warm-starts from the heuristic solution and converts the residual fine-tuning into an exact subproblem. All four large instances pass under this hybrid while the seed program passes only the easier two. The jump from a pure heuristic to a heuristic–exact hybrid is precisely what EoH's exploration-style operators are designed to elicit: by jointly evolving thoughts and code, the prompt operators encourage the agent to abandon its locally-best variant and propose a fundamentally different family of methods rather than merely retuning parameters.

CORAL on *carvalho1999* (one-dimensional bin packing / cutting stock): cross-agent assembly of a pattern-decomposition algorithm. The seed is a best-fit packing heuristic with a small monolithic MIP polish. CORAL's selected program after the multi-agent run assembles three components produced by different agent threads: a best-fit-decreasing bucket heuristic, an exhaustive cutting-pattern enumeration (the decomposition core of cutting-stock), and a

small exact assignment MIP for the residual items. The combination passes the dev set cleanly. The artifact illustrates the migration pathway: the BFD heuristic, the pattern generator, and the exact assignment MIP originate from separate agent threads in CORAL's persistent shared memory, and the final program is only producible because each agent could read and adapt the others' partial solutions. A single-agent search rarely accumulates this many distinct algorithmic ingredients within a comparable budget.

Framework-specific improvement pathways. The three case studies surface a consistent pattern that aligns with each framework's design. *OpenEvolve* stays close to its seed family and compounds small constant-factor gains, characteristic of MAP-Elites' archive-driven Depth refinement. *EoH*'s prompt operators, by jointly evolving thoughts and code, periodically jump to structurally different program families—the source of EoH's late-attempt accelerations on problems where local refinement has saturated. *CORAL*'s persistent shared memory is the only mechanism that lets a chronologically stuck agent inherit a sibling's discovery; this Migration step explains CORAL's ability to assemble compound algorithms (e.g., BFD heuristic + pattern enumeration + small exact MIP) that single-agent evolution does not reach within the same budget.