OptiHive: Ensemble Selection for Learning-Based Optimization via Statistical Modeling

Maxime Bouscary **Saurabh Amin**
Massachusetts Institute of Technology, Cambridge, MA, USA
{mbscry,amins}@mit.edu

Abstract

LLM-based solvers have emerged as a promising means of automating problem modeling and solving. However, they remain unreliable and often depend on iterative repair loops that result in significant latency. We introduce OptiHive, a framework that enhances any solver-generation pipeline to produce higher-quality solvers from natural-language descriptions of optimization problems. OptiHive uses a single batched generation to produce diverse components (solvers, problem instances, and validation tests) and filters out erroneous components to ensure fully interpretable outputs. Accounting for the imperfection of the generated components, we employ a statistical model to infer their true performance, enabling principled uncertainty quantification and solver selection. On tasks ranging from traditional optimization problems to challenging variants of the Multi-Depot Vehicle Routing Problem, OptiHive significantly outperforms baselines, increasing the optimality rate from 5% to 92% on the most complex problems.

1 Introduction

Learning-based solvers, including solvers generated by Large Language Models (LLMs) and end-to-end neural solvers, have demonstrated remarkable capabilities across diverse domains. However, their application to complex optimization tasks remains hindered by unreliable solutions and self-evaluation. Such solvers often exhibit two failure modes: (i) hard errors (syntax or runtime failures) that render solvers unusable, and (ii) soft errors (incorrect algorithms or suboptimal solutions) that cannot be detected deterministically.

Existing learning-based optimization pipelines rely on costly validation. While they can address syntactic failures, they struggle to assess solution quality, often suffer from cyclical errors, and incur high latency. Test-based approaches that prompt LLMs to generate input-output pairs or simple verification functions can improve assessment in simple settings, but rarely yield valid tests for complex problems, where ground truth is unavailable without solving the problem itself.

We present OptiHive, a two-stage framework that separates interpretability from quality estimation. In Stage 1, solvers, instances, and tests are generated simultaneously and filtered via an MILP to remove non-interpretable outputs. In Stage 2, a latent-class model jointly infers instance feasibility, solver quality, and test reliability, enabling principled solver selection.

By estimating solver performance statistically rather than relying on self-critique, OptiHive departs from “generate-then-fix" pipelines. With minimal computational overhead from filtering, inference, and selection, OptiHive effectively serves two purposes: a low-latency, high-performance LLM-based pipeline for solver generation, or a wrapper around an existing solver generation framework to greatly improve performance through rigorous statistical inference. While the framework relies on the generation step to produce at least one correct solver, the stochasticity in the generation process enables OptiHive to uncover correct solvers even when deterministic generation fails.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: DiffCoALG: Differentiable Learning of Combinatorial Algorithms.

In summary, our work makes the following contributions:

1. **Minimal and consistent latency via single-batch generation and parallelization**. OptiHive produces solvers, instances, and tests once and in parallel, eliminating iterative self-correction loops. Combined with fully parallel cross-evaluation of solutions and tests, this design yields high-quality solvers with minimal latency.
2. **Statistical solver selection**. Unlike prior work relying on LLMs' poor self-critique abilities [1, 2, 3], we treat all components as noisy and employ rigorous statistical methods to estimate the true performance of solvers.
3. **Numerical experiments on two classes of complex optimization problems**. We demonstrate that OptiHive reliably identifies high-quality solvers and substantially outperforms baselines on complex variants of the Multi-Depot Vehicle Routing Problem and the Weighted Set Cover Problem.

2 Related Work

Learning-based solvers have demonstrated remarkable abilities in problem solving across a wide range of domains [4, 5, 6, 7, 8], making them increasingly relevant tools for tackling complex computational tasks. Our work lies at the intersection of two streams of research within this field: Learning-based solvers for optimization problems and LLM-generated test functions.

Learning-Based Solvers for Optimization Problems. Chain-of-Thought prompting (CoT) [9] and Chain-of-Experts [10] improve reasoning by eliciting intermediate reasoning steps. Iterative refinement approaches, including OptiMUS [11], OptimAI [12], Optimization by PROmpting [13], Self-Guiding Exploration [14], and Hercules [15], iteratively generate, evaluate, and repair solvers, which incur high latency and are prone to repetitive iterations that fail to converge. Parallel to these methods, LLMOPT [16], and LLaMoCo [17] enhance problem formulation and solver generation via instruction-tuning, but also rely on self-correction loops. Instance-level selection of neural solvers in [18] requires training a separate selection model. In contrast, OptiHive avoids such refinement procedures: it generates solvers, instances, and tests in one batch and efficiently selects the best solver, achieving low latency and high performance.

LLM-Generated Test Functions. LLMs are known to be biased and poor at self-critique [1, 2, 3, 19], motivating external test generation. Most work focuses on unit tests [20, 21, 22, 23, 24], which reduce manual effort but are limited to simple input-output checks. Other studies [25, 26, 27] propose complete test functions, but are typically restricted to a series of assert-based checks over fixed inputs. In contrast, we generate reusable test functions that verify problem-specific invariants, such as constraint feasibility and objective value consistency. The decoupling of tests from specific inputs (i.e. problem instances) makes our framework thrifty, as each test can be reused to validate diverse solver-instance pairs.

3 Methodology

Figure 1 illustrates OptiHive's two stages. The full procedure is described as an algorithm in Appendix A, and we now detail each stage.

3.1 Generation of Valid Components

3.1.1 Components Generation.

Given a problem description with input-output specifications, we use a single LLM call to generate:

1. Candidate Solvers $\bar{S}$: each solver $s \in \bar{S}$ takes an instance i and returns either `infeasible` if no solution exists, or the best solution found with status `optimal` or `time_limit`.
2. Problem Instances $\bar{I}$: generated with varied seeds and prompt phrasing, explicitly requesting feasible, infeasible, or random instances.
3. Validity Tests $\bar{T}$: each test $t \in \bar{T}$ takes an instance-solution pair (i, x) and evaluates whether solution x is feasible for instance i and its true objective value matches the reported value.

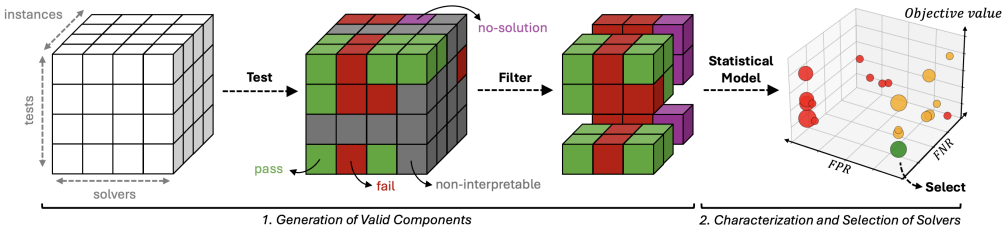


Figure 1: OptiHive produces optimization solvers through a two-stage process. In the first stage, it produces candidate solvers, problem instances, and tests, then filters out components to retain only fully interpretable solver-instance-test triples (represented as cubes). In the second stage, it applies latent class analysis [28] to estimate the performance of each solver and selects the most promising candidate. Red, yellow, and green bubbles denote solvers that return infeasible, feasible but suboptimal, and optimal solutions, respectively.

All prompts are provided in Appendix B. Components are independent, enabling parallel generation of $\bar{S}, \bar{I}, \bar{T}$ in one batch. No further LLM calls are needed during subsequent steps, which is key to the low latency of our framework.

3.1.2 Testing.

LLM-generated components may be syntactically or semantically invalid. For each $(s, i) \in \bar{S} \times \bar{I}$, we call solver s on instance i: the pair is *interpretable* if it compiles, execution raises no error, and the report contains a valid *status* field. A triple (s, i, t) is *interpretable* if test t compiles and either (s, i) yields a report with `infeasible` status, or (s, i) yields a solution and running test t on it does not raise an error during execution, and returns a boolean.

3.1.3 Filtering.

Instead of integrating a costly (and often unreliable) self-correction loop for each component to ensure compilability, executability, and evaluability, we filter out a subset of the components to retain only interpretable triples (s, i, t).

To retain a maximum number of components while filtering out all the non-interpretable triples, we formulate the following MILP:

$$\begin{aligned} \max_{w} \quad & \sum_{j \in \bar{S} \cup \bar{I} \cup \bar{T}} w_j & & (1) \\ \text{s.t.} \quad & w_s + w_i + w_t \leq 2, & \forall (s,i,t) \in \mathcal{U} \\ & w_j \in \{0,1\}, & \forall j \in \bar{S} \cup \bar{I} \cup \bar{T} \end{aligned}$$

where w_j indicates whether component j is kept, and $\mathcal{U} = \{(s,i,t) \in \bar{S} \times \bar{I} \times \bar{T} : (s,i,t) \text{ is not interpretable}\}$. After solving (1), we obtain the optimal selections $w^\star$ and define $S \triangleq \{s \in \bar{S} : w_s^\star = 1\}$, $I \triangleq \{i \in \bar{I} : w_i^\star = 1\}$, and $T \triangleq \{t \in \bar{T} : w_t^\star = 1\}$. By construction, every triple in $S \times I \times T$ is interpretable. This MILP is tractable even with hundreds of solvers, instances, and tests, as the number of variables grows linearly with the number of components, and $\mathcal{U}$ exhibits a highly structured pattern since failure of a solver, instance, or test often induces multiple non-interpretable triples involving that component.

3.2 Solver Characterization and Selection

3.2.1 Characterization.

Although every triple (s, i, t) is now interpretable, we only observe solver reports and, when a report contains a solution, whether that solution passes a suite of imperfect tests. Notably, the true feasibility of instances and the validity of reported solutions are unknown, as both solvers and tests are generated by an LLM and thus cannot be assumed to be perfectly trustworthy. By treating these unobserved variables as latent, we use a latent-class model to jointly estimate the membership of instances and solutions, the accuracy of solvers, and the reliability of tests.

We introduce the following families of variables. Observed variables are $r_{s,i} \in \{0,1\}$, indicating whether solver s reports a solution on instance i, and $r_{s,i,t} \in \{0,1\}$, indicating whether that solution passes test t. Latent variables are $f_i \in \{0,1\}$, indicating whether instance i admits a feasible solution, and $f_{s,i} \in \{0,1\}$, indicating whether the reported solution of (s,i) is truly feasible.

Our model assumes that (i) each instance is feasible with probability λ, (ii) solvers have false positive (resp. negative) rates α_s (resp. β_s) and rate of feasible reported solution γ_s, and (iii) the aggregated test outcomes $C_{s,i} = \sum_{t \in T} r_{s,i,t}$ follow Beta-Binomial distributions when conditioned on feasibility. Namely:

$$C_{s,i} \mid f_{s,i} = 0 \sim \text{BetaBinomial}\left(|T|, a_0, b_0\right), \text{ and } C_{s,i} \mid f_{s,i} = 1 \sim \text{BetaBinomial}\left(|T|, a_1, b_1\right)$$

Let $\theta = (\lambda, \{\alpha_s, \beta_s, \gamma_s\}_{s \in S}, a_0, b_0, a_1, b_1)$ denote the set of all parameters, and $\mathbf{R}$ (resp. $\mathbf{F}$) be the set of observed (resp. latent) variables. We use the expectation-maximization (EM) algorithm to find a set of parameters $\theta^\star$ locally maximizing the observed data likelihood function by iterating over the following update

$$\theta_{k+1} = \arg\max_{\theta} \mathbb{E}_{\mathbf{F} \sim \mathbb{P}(\cdot \mid \mathbf{R}, \theta_k)} \left[\ln \mathbb{P}\left(\mathbf{R}, \mathbf{F} \mid \theta_k\right)\right] \tag{2}$$

until convergence. The distribution of the latent variables $\{f_i\}_{i \in I}$, $\{f_{s,i}\}_{(s,i) \in S \times I}$ can then be estimated from $\theta^\star$. See Appendix C for details on the EM algorithm.

3.2.2 Selection.

For all reports containing a solution, let $z_{s,i}$ be the objective value reported by solver s on instance i, and $Z_s = \mathbb{E}_{i \sim \mathbb{P}(\cdot \mid r_{s,i}=1, f_{s,i}=1)}\left[z_{s,i}\right]$ be the conditional expected objective over feasible solutions reported by solver s. Since solvers may differ in their ability to detect infeasible instances or return high-quality solutions on feasible ones, we define a scalarized objective function that summarizes overall solver quality in a single score:

$$g(\theta^\star, s) \triangleq \lambda(1-\beta_s)\gamma_s Z_s + \lambda\beta_s P_{\text{miss}} + \left((1-\lambda)\alpha_s + \lambda(1-\beta_s)(1-\gamma_s)\right) P_{\text{fail}} \tag{3}$$

Here, P_{miss} penalizes reporting no solution on a feasible instance and P_{fail} penalizes reporting an infeasible solution. We set $P_{\text{miss}} = P_{\text{fail}} = 10 Z_{\max}$, where $Z_{\max}$ is the maximum absolute objective value reported across all solver-instance pairs. This ensures that both under-reporting and over-reporting solvers are severely penalized. The final solver is selected as $s^\star = \arg\min_{s \in S} g(\theta^\star, s)$.

4 Experimental Results

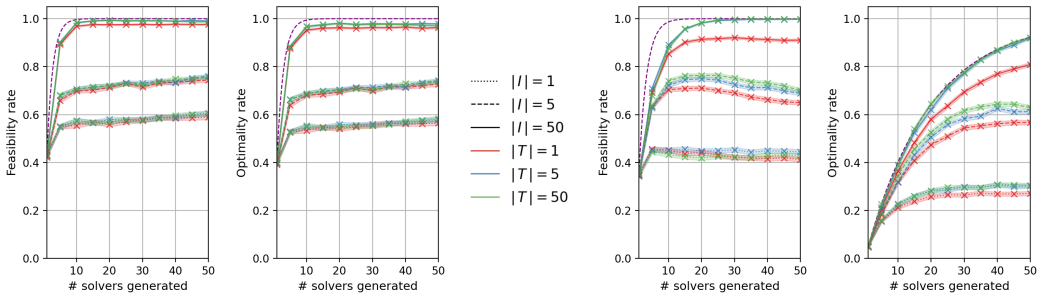
Previous benchmarks such as NLP4LP [29] and ComplexOR [10] include problems that recent LLMs can now solve reliably, yet perfect scores remain unattainable due to ambiguity in problem statements (e.g., describing integer-valued quantities but asking to formulate an LP). As failures may reflect prompt ambiguity rather than solver quality, these datasets offer limited insight into solver performance. To address this, we design variants of the Multi-Depot Vehicle Routing Problem (MDVRP) and Weighted Set Cover Problem (WSCP) with controlled complexity, enabling meaningful performance evaluation.

4.1 Experimental Setup

We compare solvers selected by OptiHive with those produced directly by the same LLM to isolate the marginal improvement from OptiHive's selection mechanism. Rather than benchmarking against other existing LLM-based optimization pipelines, our goal is to demonstrate how OptiHive can enhance any such pipeline by extracting high-quality solvers from a pool of candidates.

For each problem, we sample random tractable instances and compute ground-truth solutions using a reference solver. A candidate is *feasible* if all its reported solutions satisfy the problem-specific constraints, and *optimal* if reported objectives also match the ground truth within tolerance.

We generate 100 solvers, 100 instances, and 100 tests per problem, pre-compute all solver-instance and solver-instance-test outputs, and evaluate by repeatedly sampling components with replacement. Each run applies filtering, characterization, and selection, repeated 10,000 times with random seeds. We use OpenAI models (`gpt-4.1-nano`, `gpt-4.1-mini`, `o3`) with default temperature 0.7. Runs are parallelized on an AMD EPYC 9734 with the EM algorithm limited to 100 iterations. Each completes in under one second, negligible relative to typical LLM-based code-synthesis pipelines.



(a) DCMDVRP: Feasibility and optimality

(b) MDVRP+OBS: Feasibility and optimality

Figure 2: Feasibility and optimality rates on DCMDVRP and MDVRP+OBS across varying numbers of generated solvers, instances, and tests. The purple curve shows the optimality rate under perfect selection i.e. the probability that at least one of the generated solvers is optimal.

4.2 Multi-Depots Vehicle Routing Problems

We study two variants of MDVRP [30]:

- Distance-Constrained Multi-Depot Vehicle Routing Problem (DCMDVRP), where each vehicle has an upper bound on the total distance traveled by each vehicle.
- Multi-Depot Vehicle Routing Problem with Obstacles (MDVRP+OBS), where line-segment obstacles are present and alter the feasible routing space.

The former is a straightforward extension of the standard MDVRP, while the latter requires a non-trivial code (visibility graph construction, obstacle-aware shortest paths, MILP solving, and route reconstruction), making it substantially harder. We generate components with `gpt-4.1-mini` for DCMDVRP and `o3` for MDVRP+OBS, since `o3` consistently produces optimal solvers on DCMDVRP and `gpt-4.1-mini` never produced optimal solutions on MDVRP+OBS.

Figure 2 reports the optimality rate of OptiHive as the number of candidate solvers, instances, and tests increases. OptiHive consistently outperforms the single-solver baseline: with 50 components of each type, feasibility/optimality improves from 43%/40% to 98.7%/97.0% on DCMDVRP and from 35%/5% to 99.9%/92.1% on MDVRP+OBS.

Performance depends strongly on component diversity. More instances provide richer signals for the latent-class model, helping distinguish optimal solvers from near-optimal ones that fail on corner cases. Instance diversity is thus key to recovering optimal solvers, in particular when most candidates cluster around incorrect solutions. The number of solvers is also critical: in DCMDVRP, performance plateaus once a few optimal solvers are sampled (2a), but in MDVRP+OBS, large solver pools markedly increase the chance of including an optimal candidate and thus has a much greater impact on overall performance (2b). Since evaluating correctness is generally easier than solving the problem itself, performance saturates quickly as the number of tests increases. While test diversity remains useful to cover rare failure modes, improvements are smaller than those from adding solvers or instances.

4.3 Weighted Set Cover Problem

The WSCP [31, 32] can be illustrated through a practical scenario involving emitters and clients. Each emitter is characterized by a location, radius, and activation cost. An emitter is said to cover a client if the client lies within its coverage range. The objective is to select a minimum-cost subset of emitters so every client is covered by at least one active emitter. We consider three variants:

- K-robust WSCP: Coverage must remain after any K adversarial emitter failures. While this may appear as a complex combinatorial requirement, this variant is a standard WSCP in disguise where each client must be within range of at least $K + 1$ selected emitters.
- Probabilistic WSCP: Emitter i fails independently with probability p_i, and each client j requires coverage with probability no less than π_j. The non-linear constraint $\mathbb{P}\left(\text{client } j \text{ covered}\right) \triangleq 1 - \prod_{\{i \in S_j : x_i = 1\}} p_i \geq \pi_j$ becomes linear after taking the logarithm of both sides, yielding a tractable MILP formulation.

- Time-dependent WSCP: Clients move at constant speed along straight paths over a fixed horizon. Solving it involves computing time intervals where each client is within range of an emitter (via a quadratic equation), identifying critical subintervals with changing coverage sets, solving a static WSCP, and merging selected subintervals into contiguous activation schedules. These compounded complexities make the time-dependent variant the hardest to solve.

Ablation study. We evaluate the impact of solver, instance, and test quality by replacing one component type at a time with generations from a smaller LLM to isolate the impact of each component's quality on overall performance. The baseline method generates a single solver and returns it. For OptiHive, we sample 50 elements of each component type, run the EM algorithm, and return the solver that minimizes the scalarized objective in (3) with default penalties. Table 1 reports the results, with the reference setting using `gpt-4.1-mini` to generate all component types, and other settings using `gpt-4.1-nano` to generate one component type while keeping the other two generated by `gpt-4.1-mini`.

	K-robust				Probabilistic				Time-dependent			
	Baseline		OptiHive		Baseline		OptiHive		Baseline		OptiHive	
	Opt.	Feas.	Opt.	Feas.	Opt.	Feas.	Opt.	Feas.	Opt.	Feas.	Opt.	Feas.
Reference	98%	98%	100%	100%	73%	75%	100%	100%	3%	12%	64.1%	74.5%
`nano` solvers	42%	44%	83.1%	83.1%	33%	36%	89.9%	89.9%	0%	2%	0%	0.01%
`nano` instances	98%	98%	99.3%	99.3%	73%	75%	100%	100%	3%	12%	23.5%	37.7%
`nano` tests	98%	98%	100%	100%	73%	75%	100%	100%	3%	12%	19.7%	24.1%

Table 1: Ablation study on variants of the Weighted Set Cover Problem.

Across all variants, OptiHive markedly improves optimality and feasibility rates over the baseline, even with degraded component quality, showing that the latent-class model extracts useful signal from noisy ensembles. In the K-robust and probabilistic cases, weaker instances or tests have little effect. This is consistent with the fact that the complexity of these variants stems from conceptual depth rather than substantial coding effort, and supports our hypothesis that tests are *generally* substantially easier to write correctly than the solvers themselves.

In contrast, the time-dependent variant shows a marked performance drop when the quality of either instances or tests is weakened. Generating a balanced mix of feasible and infeasible instances is harder here, and validity tests themselves are also non-trivial to produce. This challenges the assumption that testing is significantly easier than solving. Still, noisy tests from the smaller model provide a meaningful signal, enabling OptiHive to still outperform the baseline and illustrating its ability to extract value from very noisy components.

Finally, OptiHive can successfully identify optimal solvers even when they are rare. On the time-dependent WSCP, OptiHive raises optimality from 3% to 64.1%, while with $N_S = 50$ solvers perfect selection would achieve 78.2%. Conditioned on sampling at least one optimal solver, OptiHive selects it 81.6% of the time in the reference setting, and 30.9% (resp. 25.3%) with degraded instances (resp. tests).

5 Conclusion

We introduced OptiHive, a two-stage framework that (i) generates solvers, instances, and tests in parallel while filtering out unusable components, and (ii) applies a latent-class model to infer solver quality and enable informed solver selection. By generating diverse components and avoiding self-correction loops, OptiHive delivers high-performance solutions with minimal latency, or enhances existing solver-generation pipelines by acting as a wrapper.

Experiments show substantial performance improvement: optimality rises from 5% to 92% on the hardest problems, while simple problems approach near-perfect optimality rates. Our ablation studies highlight the importance of high-quality instances and tests for distinguishing optimal solvers. Yet, OptiHive still improves performance when components come from smaller models.

Future work will explore heterogeneous solver sources and multi-stage generation to build richer ensembles and tackle even more challenging optimization problems.

References

[1] Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. On the self-verification limitations of large language models on reasoning and planning tasks. *arXiv preprint arXiv:2402.08115*, 2024.

[2] Justin Chih-Yao Chen, Archiki Prasad, Swarnadeep Saha, Elias Stengel-Eskin, and Mohit Bansal. Magicore: Multi-agent, iterative, coarse-to-fine refinement for reasoning. *arXiv preprint arXiv:2409.12147*, 2024.

[3] Jiwon Moon, Yerin Hwang, Dongryeol Lee, Taegwan Kang, Yongil Kim, and Kyomin Jung. Don't judge code by its cover: Exploring biases in llm judges for code evaluation. *arXiv preprint arXiv:2505.16222*, 2025.

[4] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[9] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[10] 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 *The twelfth international conference on learning representations*, 2023.

[11] 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*, 2024.

[12] Raghav Thind, Youran Sun, Ling Liang, and Haizhao Yang. Optimai: Optimization from natural language using llm-powered ai agents. *arXiv preprint arXiv:2504.16918*, 2025.

[13] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

[14] Zangir Iklassov, Yali Du, Farkhad Akimov, and Martin Takac. Self-guiding exploration for combinatorial problems. *Advances in Neural Information Processing Systems*, 37:130569–130601, 2024.

[15] Xuan Wu, Di Wang, Chunguo Wu, Lijie Wen, Chunyan Miao, Yubin Xiao, and You Zhou. Efficient heuristics generation for solving combinatorial optimization problems using large language models. *arXiv preprint arXiv:2505.12627*, 2025.

[16] 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. *arXiv preprint arXiv:2410.13213*, 2024.

[17] Zeyuan Ma, Hongshu Guo, Jiacheng Chen, Guojun Peng, Zhiguang Cao, Yining Ma, and Yue-Jiao Gong. Llamoco: Instruction tuning of large language models for optimization code generation. *arXiv preprint arXiv:2403.01131*, 2024.

[18] Chengrui Gao, Haopu Shang, Ke Xue, and Chao Qian. Neural solver selection for combinatorial optimization. *arXiv preprint arXiv:2410.09693*, 2024.

[19] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.

[20] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[21] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[22] Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. Learning to generate unit tests for automated debugging. *arXiv preprint arXiv:2502.01619*, 2025.

[23] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576, 2024.

[24] Zi Lin, Sheng Shen, Jingbo Shang, Jason Weston, and Yixin Nie. Learning to solve and verify: A self-play framework for code and test generation. *arXiv preprint arXiv:2502.14948*, 2025.

[25] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

[26] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2023.

[27] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J Hellendoorn. Cat-lm training language models on aligned code and tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 409–420. IEEE, 2023.

[28] Alexander Philip Dawid and Allan M Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 28(1):20–28, 1979.

[29] Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Optimization modeling using mip solvers and large language models. *arXiv preprint arXiv:2310.06116*, 2023.

[30] Paolo Toth and Daniele Vigo. *The vehicle routing problem*. SIAM, 2002.

[31] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[32] Alberto Caprara, Paolo Toth, and Matteo Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98(1):353–371, 2000.

A Full Algorithm

Algorithm 1 OptiHive

Require: Sample sizes N_S, N_I, N_T
// Generation step
Perform single batch query to the LLM to obtain:
  solvers $\bar{S}$ of size N_S
  instances $\bar{I}$ of size N_I
  tests $\bar{T}$ of size N_T
// Testing step
for all $(s, i) \in \bar{S} \times \bar{I}$ **do**
  Execute solver s on instance i to obtain report $r_{s,i}$
  if $r_{s,i} = 1$ **then**
    for all $t \in \bar{T}$ **do**
      Execute test t on solution of (s, i) to obtain $r_{s,i,t}$
    end for
  end if
end for
// Filtering step
Solve (1) to obtain $w^\star$
$S \leftarrow \{s \in \bar{S} : w_s^\star = 1\}$
$I \leftarrow \{i \in \bar{I} : w_i^\star = 1\}$
$T = \{t \in \bar{T} : w_t^\star = 1\}$
$\mathbf{R} \leftarrow \{r_{s,i,t} : (s, i, t) \in S \times I \times T\}$
// Characterization step
Initialize $\theta_0 = (\lambda, \{\alpha_s, \beta_s, \gamma_s\}_{s \in S}, a_0, b_0, a_1, b_1)$
repeat
  Compute θ_{k+1} from (2) with θ_k and $\mathbf{R}$
until convergence to $\theta^\star$ or iteration limit
// Selection step
$s^\star \leftarrow \arg\min_{s \in S} g(\theta^\star, s)$
return $s^\star$

B Prompt Templates

B.1 Solvers

Solver Generation

```
You are a code-generation agent expert in Python and Gurobi.

[Problem Specifications]
Here is the problem description: {problem_description}
Here is the template for the problem input: {input_template}
The output of the function must follow the template: {output_template}

[Instructions]
Your task is to implement a function solve with a unique argument data as
input and returning a solution to the problem.
Write the complete, executable, and well-indented code of the solve
function, including necessary imports.
Status codes are: OPTIMAL for a proven best feasible solution,
INFEASIBLE when no feasible solution is found.
Use a TimeLimit of 5 seconds for the optimization. Do not include
example usage.
```

B.2 Instances

We encourage diversity of instances via two mechanisms. First, we rotate diversity directives among six options:

1. If possible, the data should be an infeasible instance for the above problem.
2. The data should be a clearly feasible instance for the above problem in that it admits a simple feasible solution.
3. The data should result in optimal solutions to the above problem having tight constraints.
4. The data should be randomized.
5. The data should be randomized with hyperparameters that will make the instance likely feasible.
6. The data should be randomized with hyperparameters that will make the instance likely infeasible.

Second, we sample and provide a random sequence of 100 digits to limit the similarity of numerical values across generated instances. To avoid lengthy LLM outputs, we ask the LLM to provide a function that outputs an instance of the considered problem, rather than directly outputting the instance.

Instance Generation

```
You are a code-generation agent expert in Python.

[Problem Specifications]
Consider the following problem: {problem_description}
Here is the template for the problem input: {input_template}

[Instructions]
Your task is to implement a function generate_input with no argument and
returning a input following the input template.
{diversity_directives}
Write the complete, executable and well indented code of the
generate_input function, including necessary imports.
Take inspiration from the following: {seed}
```

B.3 Tests

Test Generation

```
You are a code-generation agent expert in Python.

[Problem Specifications]
Here is the problem description: {problem_description}
Here is the input template: {input_template}
Here is the solution template: {output_template}

[Instructions]
For every concrete instance data that follows the input template, there
is a corresponding solution object that follows the solution template.
Your task is to implement a function test(data, solution) -> bool that
returns True if and only if all of the following hold:
  1. The solution is feasible (it satisfies every problem constraint).
  2. The reported objective value matches the cost you compute (within a
     small numerical tolerance).
  3. All solution fields are internally coherent.
Write the complete, executable, and well-indented Python code
implementing the test function, including necessary imports.
```

```
Do not include example usage.
```

C Expectation Maximization Model Specifications

Observed variables:

$$r_{s,i} = \begin{cases} 1 & \text{if solver } s \text{ reports a solution on instance } i \\ 0 & \text{otherwise.} \end{cases}, \quad \forall (s,i) \in S \times I$$

$$r_{s,i,t} = \begin{cases} 1 & \text{solution } (s,i) \text{ passes test } t \\ 0 & \text{otherwise.} \end{cases}, \quad \forall (s,i,t) \in S \times I \times T \text{ s.t. } r_{s,i} = 1$$

For all $(s,i) \in S \times I$ such that $r_{s,i} = 1$, we define $C_{s,i} = \sum_{t \in T} r_{s,i,t}$.

Latent variables:

$$f_i = \begin{cases} 1 & \text{if instance } i \text{ admits a feasible solution} \\ 0 & \text{otherwise.} \end{cases}, \quad \forall i \in I$$

$$f_{s,i} = \begin{cases} 1 & \text{solution } (s,i) \text{ is feasible} \\ 0 & \text{otherwise.} \end{cases}, \quad \forall (s,i) \in S \times I \text{ s.t. } r_{s,i} = 1$$

Parameters:

$$\theta = (\lambda, (\alpha_s)_{s \in S}, (\beta_s)_{s \in S}, (\gamma_s)_{s \in S}, a_0, b_0, a_1, b_1)$$

where

$\lambda = \mathbb{P}(f_i = 1)$	(feasiblity rate of instances)
$\alpha_s = \mathbb{P}(r_{s,i} = 1 \mid f_i = 0)$	(type I error of solver s)
$\beta_s = \mathbb{P}(r_{s,i} = 0 \mid f_i = 1)$	(type II error of solver s)
$\gamma_s = \mathbb{P}(f_{s,i} = 1 \mid f_i = 1, r_{s,i} = 1)$	(feasibility rate of solutions reported by s on feasible instances)
a_0, b_0	Parameters of the Beta–Binomial for infeasible instances
a_1, b_1	Parameters of the Beta–Binomial for feasible instances

C.1 E-step

We first compute the intermediate quantities involved in the conditional expectations of f_i and $f_{s,i}$. Let $\mathbf{R}$ denote all observed variable and $\mathbf{R}_s = \{r_{s,i} : i \in I\} \cup \{r_{s,i,t} : (i,t) \in I \times T, r_{s,i} = 1\}$ denote the observed variables related to solver s. Let $\nu(k \mid n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}$ be the probability mass function of the Beta-Binomial distribution, where B is the beta function. We define for all $(s,i) \in S \times I$:

$$A_{s,i}^{(0)} \triangleq \nu(C_{s,i} \mid T, a_0, b_0)$$
$$A_{s,i}^{(1)} \triangleq \nu(C_{s,i} \mid T, a_1, b_1)$$

and, for all $i \in I$:

$$\begin{aligned} B_i^{(0)} &\triangleq \mathbb{P}(\mathbf{R} \mid f_i = 0, \theta) \\ &= \prod_{s \in S} \left[\alpha_s \mathbb{P}(\mathbf{R}_s \mid f_i = 0, \theta)\right]^{r_{s,i}} (1 - \alpha_s)^{(1 - r_{s,i})} \\ &= \prod_{s \in S} \left[\alpha_s A_{s,i}^{(0)}\right]^{r_{s,i}} (1 - \alpha_s)^{(1 - r_{s,i})} \end{aligned}$$

$$\begin{aligned} B_i^{(1)} &\triangleq \mathbb{P}(\mathbf{R} \mid f_i = 1, \theta) \\ &= \prod_{s \in S} \left[(1 - \beta_s)\left(\gamma_s \mathbb{P}(\mathbf{R}_s \mid r_{s,i} = 1, f_{s,i} = 1, \theta) + (1 - \gamma_s)\mathbb{P}(\mathbf{R}_s \mid r_{s,i} = 1, f_{s,i} = 0, \theta)\right)\right]^{r_{s,i}} \beta_s^{(1 - r_{s,i})} \\ &= \prod_{s \in S} \left[(1 - \beta_s)\left(\gamma_s A_{s,i}^{(1)} + (1 - \gamma_s) A_{s,i}^{(0)}\right)\right]^{r_{s,i}} \beta_s^{(1 - r_{s,i})} \end{aligned}$$

We then proceed to compute the conditional expectation of f_i and $f_{s,i}$ for a given θ. For all $i \in I$:

$$\begin{aligned}
\mathbb{E}\left[f_i \mid \mathbf{R}, \theta\right] &= \mathbb{P}\left(f_i = 1 \mid \mathbf{R}, \theta\right) \\
&= \frac{\mathbb{P}\left(\mathbf{R} \mid f_i = 1, \theta\right) \mathbb{P}\left(f_i = 1 \mid \theta\right)}{\mathbb{P}\left(\mathbf{R} \mid \theta\right)} \\
&= \frac{\mathbb{P}\left(\mathbf{R} \mid f_i = 1, \theta\right) \mathbb{P}\left(f_i = 1 \mid \theta\right)}{\mathbb{P}\left(\mathbf{R} \mid f_i = 1, \theta\right) \mathbb{P}\left(f_i = 1\right) + \mathbb{P}\left(\mathbf{R} \mid f_i = 0, \theta\right) \mathbb{P}\left(f_i = 0\right)} \\
&= \frac{\lambda B_i^{(1)}}{\lambda B_i^{(1)} + (1-\lambda) B_i^{(0)}}
\end{aligned}$$

For all $(s, i) \in S \times I$ such that $r_{s,i} = 1$:

$$\begin{aligned}
&\mathbb{P}\left(f_{s,i} = 1 \mid f_i = 1, \mathbf{R}, \theta\right) \\
&= \frac{\mathbb{P}\left(\mathbf{R} \mid f_{s,i} = 1, f_i = 1, \theta\right) \mathbb{P}\left(f_{s,i} = 1 \mid f_i = 1, \theta\right)}{\mathbb{P}\left(\mathbf{R} \mid f_i = 1, \theta\right)} \\
&= \frac{\mathbb{P}\left(\mathbf{R} \mid f_{s,i} = 1, f_i = 1, \theta\right) \mathbb{P}\left(f_{s,i} = 1 \mid f_i = 1, \theta\right)}{\mathbb{P}\left(\mathbf{R} \mid f_{s,i} = 1, f_i = 1, \theta\right) \mathbb{P}\left(f_{s,i} = 1 \mid f_i = 1\right) + \mathbb{P}\left(\mathbf{R} \mid f_{s,i} = 0, f_i = 1, \theta\right) \mathbb{P}\left(f_i = 0 \mid f_i = 1\right)} \\
&= \frac{\gamma_s A_{s,i}^{(1)}}{\gamma_s A_{s,i}^{(1)} + (1-\gamma_s) A_{s,i}^{(0)}}
\end{aligned}$$

Since $\mathbb{P}\left(f_{s,i} = 1 \mid f_i = 0\right) = 0$, we obtain:

$$\begin{aligned}
\mathbb{E}\left[f_{s,i} \mid \mathbf{R}, \theta\right] &= \mathbb{P}\left(f_{s,i} = 1 \mid f_i = 1, \mathbf{R}, \theta\right) \mathbb{P}\left(f_i = 1 \mid \mathbf{R}, \theta\right) + \mathbb{P}\left(f_{s,i} = 1 \mid f_i = 0, \mathbf{R}, \theta\right) \mathbb{P}\left(f_i = 0 \mid \mathbf{R}, \theta\right) \\
&= \frac{\gamma_s A_{s,i}^{(1)}}{\gamma_s A_{s,i}^{(1)} + (1-\gamma_s) A_{s,i}^{(0)}} \frac{\lambda B_i^{(1)}}{\lambda B_i^{(1)} + (1-\lambda) B_i^{(0)}}
\end{aligned}$$

C.2 M-step

The log-likelihood is given by:

$$\begin{aligned}
&\ln \mathbb{P}\left(\mathbf{R}, \mathbf{F} \mid \theta\right) \\
&= \sum_{i \in I} \left[\ln \mathbb{P}\left(f_i\right) + \sum_{s \in S} \ln \mathbb{P}\left(r_{s,i} \mid f_i\right) + \sum_{\substack{s \in S \\ r_{s,i}=1}} \left[\ln \mathbb{P}\left(f_{s,i} \mid f_i\right) + \sum_{t \in T} \ln \mathbb{P}\left(r_{s,i,t} \mid f_{s,i}\right) \right] \right] \\
&= \sum_{i \in I} \left[f_i \ln(\lambda) + (1 - f_i) \ln(1-\lambda) \right] \\
&\quad + \sum_{i \in I} \sum_{s \in S} \left[r_{s,i} \left(f_i \ln(1-\beta_s) + (1-f_i) \ln(\alpha_s) \right) + (1 - r_{s,i}) \left(f_i \ln(\beta_s) + (1-f_i) \ln(1-\alpha_s) \right) \right] \\
&\quad + \sum_{i \in I} \sum_{\substack{s \in S \\ r_{s,i}=1}} f_i \left[f_{s,i} \ln(\gamma_s) + (1 - f_{s,i}) \ln(1-\gamma_s) \right] \\
&\quad + \sum_{i \in I} \sum_{\substack{s \in S \\ r_{s,i}=1}} \left[f_{s,i} \ln \nu(C_{s,i} \mid T, a_1, b_1) + (1 - f_{s,i}) \ln \nu(C_{s,i} \mid T, a_0, b_0) \right]
\end{aligned}$$

Let $\hat{f}_i \triangleq \mathbb{E}\left[f_i \mid \mathbf{R}, \theta\right]$ and $\hat{f}_{s,i} \triangleq \mathbb{E}\left[f_{s,i} \mid \mathbf{R}, \theta\right] = \mathbb{E}\left[f_i f_{s,i} \mid \mathbf{R}, \theta\right]$. Taking the conditional expectation and differentiating with respect to λ, we have:

$$\frac{\partial}{\partial \lambda} \mathbb{E}\left[\ln \mathbb{P}\left(\mathbf{R}, \mathbf{F} \mid \theta\right) \mid \mathbf{R}, \theta\right] = \sum_{i \in I} \left(\frac{\hat{f}_i}{\lambda} - \frac{1 - \hat{f}_i}{1 - \lambda} \right) \tag{4}$$

Equating (4) to 0, we obtain: $\lambda = \frac{\sum_{i \in I} \hat{f}_i}{|I|}$. With analogous reasoning, we obtain:

$$\alpha_s = \frac{\sum_{i \in I} (1 - \hat{f}_i) r_{s,i}}{\sum_{i \in I} (1 - \hat{f}_i)}$$
$$\beta_s = \frac{\sum_{i \in I} \hat{f}_i (1 - r_{s,i})}{\sum_{i \in I} \hat{f}_i}$$
$$\gamma_s = \frac{\sum_{i \in I} r_{s,i} \hat{f}_{s,i}}{\sum_{i \in I} r_{s,i} \hat{f}_i}$$

We use the method of moments to update the parameters a_0, b_0, a_1, and b_1 using a Beta prior with $(\bar{\alpha}, \bar{\beta}) = (20, 1)$ for the true positive rate. Define:

$$p_0 = \frac{\sum_{s,i} \left(1 - \hat{f}_{s,i}\right) \frac{C_{s,i}}{T}}{\sum_{s,i} \left(1 - \hat{f}_{s,i}\right)}$$
$$p_1 = \frac{\sum_{s,i} \hat{f}_{s,i} \frac{C_{s,i}}{T} + \bar{\alpha} - 1}{\sum_{s,i} \hat{f}_{s,i} + \bar{\alpha} + \bar{\beta} - 2}$$

and

$$\mu_0 = \frac{\sum_{s,i} \left(1 - \hat{f}_{s,i}\right) C_{s,i}}{T \cdot \sum_{s,i} \left(1 - \hat{f}_{s,i}\right)}$$
$$\sigma_0^2 = \frac{\sum_{s,i} \left(1 - \hat{f}_{s,i}\right) \left(C_{s,i}/T - \mu_0\right)^2}{\sum_{s,i} \left(1 - \hat{f}_{s,i}\right)}$$
$$\mu_1 = \frac{\sum_{s,i} \hat{f}_{s,i} C_{s,i}}{T \cdot \sum_{s,i} \hat{f}_{s,i}}$$
$$\sigma_1^2 = \frac{\sum_{s,i} \hat{f}_{s,i} \left(C_{s,i}/T - \mu_1\right)^2}{\sum_{s,i} \hat{f}_{s,i}}$$

Then for $k \in \{0, 1\}$:

$$\rho_k = \frac{\frac{T\sigma_k^2}{\mu_k(1-\mu_k)} - 1}{T - 1}$$

Finally, retrieve a_k and b_k from p_k and ρ_k for $k \in \{0, 1\}$ as:

$$a_k = p_k \left(\frac{1}{\rho_k} - 1\right)$$
$$b_k = (1 - p_k) \left(\frac{1}{\rho_k} - 1\right)$$