EVOLVING EVALUATION GATES FOR EFFICIENT LLM-BASED MILP BRANCHING POLICY SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Efficient branching policies can substantially accelerate Mixed Integer Linear Programming (MILP) solvers, but discovering such policies remains computationally expensive. Recent LLM-based frameworks automate branching-policy discovery by generating executable policy programs and optimizing their parameters with solver feedback. However, candidate evaluation still requires many costly MILP solves, especially during zeroth-order parameter optimization. We propose a computation-aware Stage-2 evaluation gate that operates inside the Bayesian optimization loop: before each proposed parameter trial is evaluated by the solver, the gate decides whether the trial should be evaluated or skipped with conservative feedback. We further maintain a curated and archive-evolved gate-policy pool, allowing different quality–cost trade-offs across candidate programs. Experiments on standard MILP benchmarks show that trial-level gating can reduce Stage-2 training solves while largely preserving final policy quality. Our results suggest that LLM-guided branching-policy discovery can be made more efficient by optimizing not only the discovered policies, but also the evaluation process used to discover them.

1 INTRODUCTION

Mixed Integer Linear Programming (MILP) is a central modeling language for combinatorial optimization. Modern MILP solvers rely on branch-and-bound and branch-and-cut, where the branching policy selects which fractional variable to branch on at each search node. This decision can substantially affect search-tree size and solving time. Classical rules such as strong branching and pseudo-cost branching encode hand-designed trade-offs between decision quality and computational overhead (Achterberg, 2009; Bestuzheva et al., 2021), while learning-based methods seek to automate this choice from data (Khalil et al., 2016; Gasse et al., 2019; Gupta et al., 2020; Scavuzzo et al., 2022).

Large language models offer another route: generating executable algorithmic components and improving them through task feedback. LLM4Branch applies this idea to MILP branching by asking an LLM to propose Python branching-policy programs and tuning their numerical parameters with solver feedback (Hou et al., 2026). This avoids expert demonstrations and produces inspectable policies, but shifts substantial cost into candidate evaluation. After a candidate passes a lightweight Stage-1 filter, Stage-2 Bayesian optimization repeatedly evaluates parameter trials by solving training MILP instances. Across many generated candidates, these repeated solver calls become a major bottleneck.

We ask whether LLM-guided branching-policy search can be made more compute-efficient by optimizing the evaluation process itself. We introduce a Stage-2 trial-level gate that examines lightweight BO-state signals before a parameter trial is evaluated. Promising or uncertain trials are evaluated normally; low-value trials are skipped and receive conservative feedback so that the BO ask–tell loop remains well defined. The gate is local: it does not reject entire LLM-generated programs, change the branching-policy search space, or modify held-out evaluation.

Across four MILP benchmark families, trial-level gating reduces Stage-2 training-solve usage while largely preserving final policy quality. Direct gate saving rates are 11.82% on `cauctions`, 18.92% on `setcover`, 5.19% on `facilities`, and 5.59% on `indset`. Held-out results show that the trade-off is benchmark-dependent: `setcover` remains nearly neutral, `facilities` improves

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

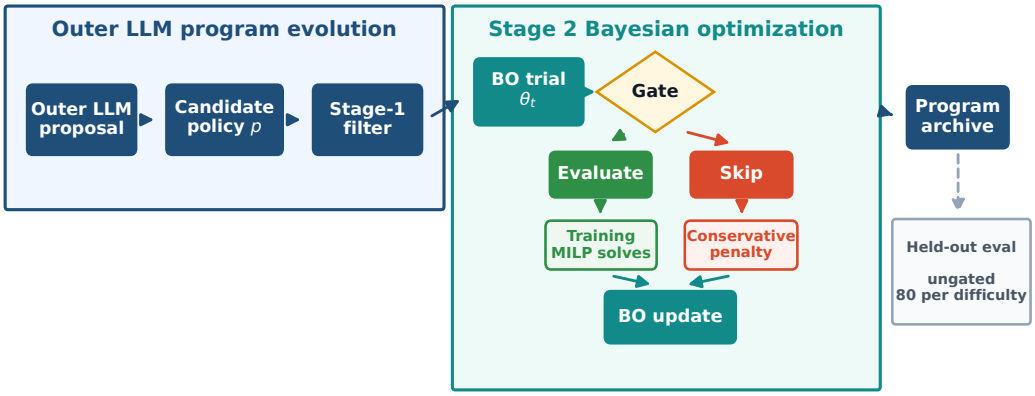

Figure 1: Overview of the proposed Stage-2 evaluation gate. The outer LLM program evolution loop remains unchanged, while the gate acts inside Stage-2 Bayesian optimization to decide whether each proposed parameter trial should consume training MILP solves. Held-out evaluation is ungated.

slightly, `cauctions` degrades on hard instances, and `indset` improves held-out runtime on all tested difficulty levels.

Our contributions are:

- identifying Stage-2 parameter optimization as a major evaluation-cost bottleneck in LLM-based MILP branching-policy discovery;
- proposing a trial-level gate that skips selected Stage-2 BO trials and returns conservative feedback without changing held-out evaluation;
- introducing a curated and archive-evolved gate-policy pool separated from the outer program search;
- providing matched experiments and held-out evaluation across standard MILP benchmarks, with both direct and observed compute accounting.

2 Related Work

Branching policies for MILP. Variable branching is a central decision in branch-and-bound and branch-and-cut MILP solvers. Classical rules such as strong branching and pseudo-cost branching trade off decision quality and online cost, and modern solvers such as SCIP combine branching with many other components (Achterberg, 2009; Bestuzheva et al., 2021). Learning-based branching methods include supervised ranking approaches that imitate strong branching (Khalil et al., 2016), graph neural policies over variable-constraint representations (Gasse et al., 2019), hybrid CPU-oriented branching models (Gupta et al., 2020), and reinforcement-learning formulations based on tree MDPs (Scavuzzo et al., 2022). More broadly, learning has been used to support MILP and combinatorial optimization through solver-control environments, neural primal heuristics, and surveys of learning-augmented optimization (Prouvost et al., 2020; Nair et al., 2020; Lodi & Zarpellon, 2017; Bengio et al., 2021). Our work is complementary: we do not propose a new branching-policy architecture, but reduce the cost of evaluating candidate policies during LLM-based search.

LLM-based program search. Recent systems use learned agents or LLMs as program generators in generate–evaluate–select loops. AlphaDev discovers low-level sorting routines with reinforcement learning (Mankowitz et al., 2023), while FunSearch and AlphaEvolve use LLM-guided program evolution for mathematical and algorithmic discovery (Romera-Paredes et al., 2024; Novikov et al., 2025). LLM4Branch applies this paradigm to MILP branching-policy discovery by generating

executable branching programs and optimizing their parameters with solver feedback (Hou et al., 2026). This avoids expert demonstrations and optimizes closer to solver-level performance, but makes candidate evaluation expensive. Our method keeps the LLM4Branch-style outer search intact and inserts a gate only inside Stage-2 parameter optimization.

Efficient evaluation in black-box optimization. Bayesian optimization and sequential model-based algorithm configuration are widely used for expensive black-box objectives (Snoek et al., 2012; Hutter et al., 2011). Hyperband and BOHB reduce search cost through adaptive resource allocation and early stopping (Li et al., 2018; Falkner et al., 2018). Our setting differs because the expensive evaluations are MILP solver calls nested inside the evaluation of an LLM-generated branching program. Moreover, skipped trials must still provide feedback to the BO ask–tell loop. We therefore use conservative penalty feedback rather than simply dropping skipped trials.

3 Background

3.1 MILP Branching

A mixed integer linear program can be written as

$$\min_x \; c^\top x \quad \text{s.t.} \quad Ax \le b, \quad x_j \in \mathbb{Z}\; \forall j \in \mathcal{I}. \tag{1}$$

Modern MILP solvers typically handle such problems with branch-and-bound or branch-and-cut. At each search node, the solver solves a relaxation. If the relaxed solution violates integrality, a branching rule selects one fractional variable and creates child subproblems.

Let $\mathcal{C}(s)$ be the set of candidate branching variables at solver state s. A branching policy scores candidates as

$$\pi_\theta(s, j), \quad j \in \mathcal{C}(s), \tag{2}$$

and the solver branches on the highest-scoring variable. This decision affects the search-tree size, the number of relaxations solved, and wall-clock time. Classical rules such as strong branching and pseudo-cost branching represent different trade-offs between decision quality and online cost.

3.2 LLM4Branch-Style Policy Search

LLM4Branch formulates branching-policy design as LLM-driven program search (Hou et al., 2026). Instead of training a black-box neural policy, the framework asks an LLM to generate executable Python programs that score candidate branching variables. The generated program contains both a program skeleton and a small parameter vector. The parameter vector is then optimized with direct solver feedback.

In the setting studied here, candidate evaluation has two stages. Stage 1 is a lightweight validity and quality filter. It checks whether the generated program can execute and whether its initial behavior is reasonable on a small set of instances. Stage 2 performs zeroth-order parameter optimization for candidates that pass Stage 1. Our method keeps the outer LLM program search unchanged and modifies only this Stage-2 evaluation.

3.3 Stage-2 Evaluation Cost

Let p denote a generated branching-policy program and θ its parameter vector. Stage-2 evaluation searches for a good θ by minimizing an empirical solver-cost objective over training instances:

$$f_p(\theta) = \frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} \ell(p_\theta, \tau), \tag{3}$$

where ℓ measures solving cost on the MILP instance generated from seed τ. In our experiments, this inner optimization is implemented with Bayesian optimization. At each BO trial, the evaluator proposes a parameter vector, runs the solver on training instances, and reports the resulting objective back to the optimizer.

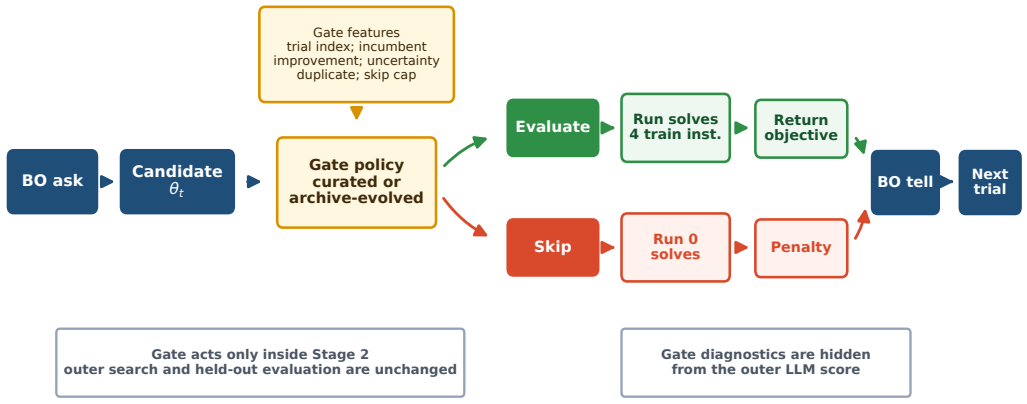

Figure 2: Trial-level gating inside Stage-2 Bayesian optimization. The gate observes lightweight BO-state features before a parameter trial is evaluated. It either runs the normal training MILP solves and returns the measured objective, or skips the trial and returns a conservative penalty to the BO loop.

If every BO trial is fully evaluated, the Stage-2 training-solve count for one candidate scales as

$$n_{\text{calls}} \cdot n_{\text{train}}. \tag{4}$$

Across many LLM-generated candidate programs, this becomes a major computational bottleneck. The key opportunity is that some parameter trials may be unlikely to improve the current incumbent, while others remain uncertain and should still be evaluated. This motivates a trial-level gate that decides whether a proposed Stage-2 parameter trial is worth full solver evaluation.

4 Method

4.1 Overview

We propose a trial-level evaluation gate for LLM-based MILP branching-policy search. The outer search loop remains the same as in the LLM4Branch-style pipeline: an LLM generates candidate branching-policy programs, each candidate is evaluated by a solver-based evaluator, and the resulting metrics update the program archive. Our modification is local to the evaluator. It acts only inside Stage-2 Bayesian optimization, where a candidate program's numerical parameters are tuned over training MILP instances.

For a candidate program p, Stage-2 optimization proposes a sequence of parameter vectors $\theta_1, \ldots, \theta_T$. In the no-gate baseline, every θ_t is evaluated by running p_{θ_t} on all Stage-2 training instances. In our gated evaluator, a gate policy observes lightweight information about the current BO state and chooses

$$a_t = g(\theta_t, \mathcal{D}_{t-1}, h_t) \in \{\text{evaluate}, \text{skip}\}, \tag{5}$$

where $\mathcal{D}_{t-1}$ is the history of previous Stage-2 observations and h_t contains auxiliary features such as trial index, incumbent quality, recent improvement, and uncertainty indicators.

The gate does not reject an entire LLM-generated program. It only decides whether one proposed parameter trial inside Stage 2 should consume training MILP solves. Held-out evaluation is always ungated.

4.2 Gate Decisions and Skipped-Trial Feedback

The gate uses only information available before the current trial is evaluated: the normalized BO trial index, the number of previous evaluated and skipped trials, the current best objective, recent

improvement, distance from previously evaluated parameter vectors, and simple uncertainty or audit indicators. Each policy enforces a minimum number of real evaluations so that BO receives enough early observations.

If the gate chooses EVALUATE, the trial receives the normal solver objective $y_t = f_p(\theta_t)$. If it chooses SKIP, the evaluator avoids the training MILP solves but still returns feedback to the BO loop. For a minimization-style Stage-2 objective, the skipped trial receives a conservative penalty:

$$\tilde{y}_t = b_{t-1} + \lambda_t, \tag{6}$$

where b_{t-1} is the best observed objective before trial t and $\lambda_t > 0$ is a policy-dependent margin. This keeps the BO ask–tell loop well-defined while avoiding the overly strong claim that the skipped trial is a catastrophic failure.

4.3 GATE-POLICY POOL AND ARCHIVE EVOLUTION

A single fixed gate may not work equally well for all candidate programs. Some programs need more Stage-2 exploration, while others repeatedly propose weak parameter trials. We therefore maintain a curated pool of gate policies with different quality–cost trade-offs, including quality-first, balanced-audit, early-stop-guarded, exploration-preserving, and hybrid styles. These policy families are summarized in Table 7 in Appendix B.

In the full GATEPOOL variant, policy selection is archive-evolved. The gate archive records which policies produced favorable quality–cost outcomes on previous candidate evaluations. Conceptually, a gate policy receives reward

$$R(g; p) = Q(m_p) + \beta C(S_p) - \eta P(g, p), \tag{7}$$

where m_p is the candidate-level metric, S_p is the number of training solves saved, Q rewards candidate quality, C rewards compute savings, and P penalizes unsafe behavior such as excessive skipping. The archive is separate from the outer program database: gate metadata are logged for analysis, but the outer LLM search receives only the ordinary candidate-program metrics.

4.4 COMPUTE ACCOUNTING

We count direct compute savings in training MILP solves avoided during Stage-2 optimization. If a benchmark uses n_{train} training instances per BO trial, then each skipped trial saves approximately n_{train} solver calls:

$$S_{\text{direct}} = \sum_{p} n_{\text{train}}(p) \sum_{t=1}^{T_p} \mathbb{I}[a_{p,t} = \text{skip}]. \tag{8}$$

We also report a baseline-relative observed reduction,

$$S_{\text{observed}} = N_{\text{baseline}} - N_{\text{gated}}, \tag{9}$$

where N_{baseline} and N_{gated} are the total observed Stage-2 training-solve counts under matched no-gate and gated configurations. Direct savings measure what the gate itself skipped. Observed reduction measures the total run-level difference, which also depends on how many candidates pass Stage 1 and enter Stage 2.

5 EXPERIMENTS

5.1 SETUP

We evaluate whether a lightweight Stage-2 gate can reduce the cost of LLM-guided branching-policy search while preserving the quality of the selected programs. The comparison uses four MILP benchmark families from the LLM4Branch evaluation pipeline: combinatorial auctions (`cauctions`), set covering (`setcover`), capacitated facility location (`facilities`), and maximum independent set (`indset`).

For each benchmark, the no-gate baseline and the gated variant use matched outer search settings. The baseline evaluates every Stage-2 Bayesian-optimization trial with full training MILP solves.

Table 1: Evolution-time results. "Dir./obs." reports direct gate saving rate and observed baseline-relative reduction.

Benchmark	Method	n	Score	Best iter	Nodes	Dir./obs.
cauctions	Baseline	5	4.4392 ± 0.0611	155.2	119.65	0.00 / 0.00
cauctions	GATEPOOL	5	4.4331 ± 0.0718	107.2	120.45	11.82 / 11.27
setcover	Baseline	3	4.3465 ± 0.0835	143.7	131.54	0.00 / 0.00
setcover	GATEPOOL	3	4.3077 ± 0.0335	123.7	136.54	18.92 / 21.56
facilities	Baseline	3	3.3705 ± 0.1198	65.0	357.88	0.00 / 0.00
facilities	GATEPOOL	3	3.2984 ± 0.0078	89.3	383.58	5.19 / 32.54
indset	Baseline	1	1.6231	82.0	2457.88	0.00 / 0.00
indset	GATEPOOL	1	1.5223	56.0	2791.00	5.59 / -7.49^{*}

Table 2: Held-out solving-time change of gated programs relative to no-gate baselines.

Benchmark	Easy	Medium	Hard
cauctions	-0.44%	-3.77%	+14.53%
setcover	-0.57%	+1.49%	-0.62%
facilities	-0.29%	-0.20%	-2.30%
indset	-3.00%	-10.45%	-17.72%

GATEPOOL uses the same outer LLM search and the same final evaluation protocol, but inserts the curated gate pool inside Stage-2 optimization. The gate is disabled during held-out evaluation.

We report two savings notions. Direct gate savings count only BO trials explicitly skipped by the gate:

$$\text{direct_saved} = n_{\text{train}} \cdot \#\{\text{skipped BO trials}\}.$$

Observed reduction compares the total Stage-2 training-solve usage of the gated run with the matched no-gate baseline:

$$\text{observed_rate} = \frac{\text{solves}_{\text{baseline}} - \text{solves}_{\text{gated}}}{\text{solves}_{\text{baseline}}}.$$

The second quantity also reflects how many candidate programs enter Stage 2, so it can differ from direct gate savings.

5.2 EVOLUTION-TIME RESULTS

Table 1 summarizes the main evolution-time comparison. Values are means across completed runs when multiple runs are available. Higher score is better and lower validation nodes are better.

The gate reduces direct Stage-2 training-solve usage in all four benchmark families. For `indset`, the selected gated run skips trials directly but enters Stage 2 more often than the baseline, giving a negative observed reduction; its held-out runtime nevertheless improves on all three difficulty levels.

5.3 HELD-OUT RESULTS

We evaluate the selected best programs on 80 held-out instances per difficulty level. The final held-out evaluation is ungated. Table 2 reports the gated runtime change relative to the matched no-gate baseline; negative values indicate faster solving.

The held-out results are benchmark-dependent. `setcover` is nearly neutral, `facilities` improves slightly, and `cauctions` degrades on hard instances despite easy and medium improvements. The `indset` gated program improves held-out runtime on all three difficulty levels, but we treat this as single-run evidence.

5.4 FROZEN INITIAL GATE POOL ABLATION

We ran one matched `cauctions` ablation with the initial eight-policy gate pool frozen throughout evolution. This isolates whether the pool itself already contains useful filters, separate from archive evolution.

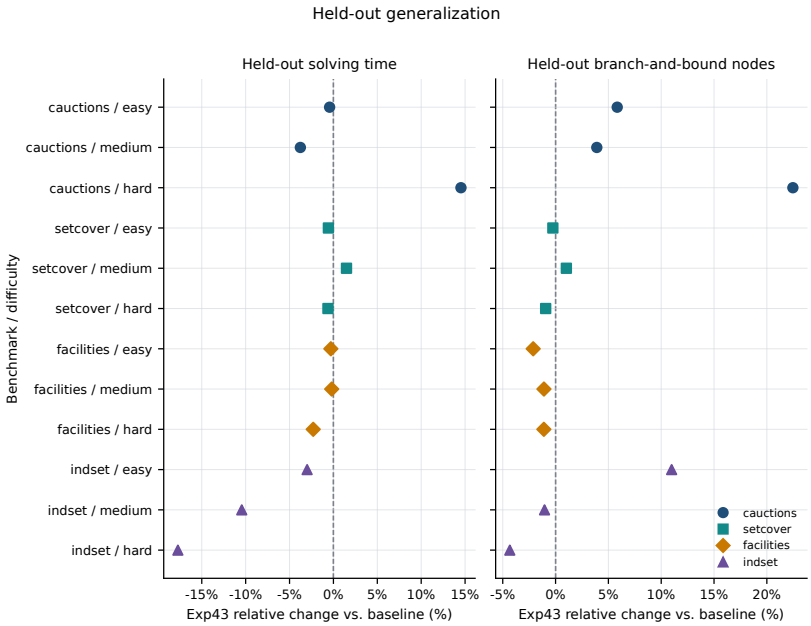

Figure 3: Held-out generalization of selected best programs. Points left of zero indicate faster solving than the matched no-gate baseline.

Table 3: Frozen Initial Gate Pool ablation on `cauctions`. Held-out times are geometric means over 80 instances.

Cond.	n	Score	Dir.	Obs.	Easy	Med.	Hard
Baseline	5	4.4392	0.00	0.00	1.328	19.079	205.909
GatePool	5	4.4331	11.82	11.27	1.322	18.360	235.829
Frozen	1	4.4296	9.66	8.34	1.399	16.649	177.250

The frozen pool obtains comparable evolution-time quality and nontrivial direct savings in this single run. Its held-out time changes relative to the baseline are $+5.40\%$, -12.73%, and -13.92% on easy, medium, and hard instances, respectively. We use this as a mechanism check rather than a stability claim.

6 Analysis

6.1 Quality–Cost Trade-Off

The main question is whether Stage-2 gating reduces evaluation cost without materially weakening the final branching policy. Figure 4 plots the change in evolution-time score against observed Stage-2 training-solve reduction. The desired region is the upper-right: higher policy quality and fewer training solves.

The replicated `cauctions`, `setcover`, and `facilities` runs all reduce total Stage-2 training solves while preserving broadly comparable evolution-time quality. The `indset` case illustrates why quality and cost must be read together: its selected gated program improves held-out runtime, but the run enters Stage 2 more often and therefore uses more total Stage-2 training solves.

6.2 Savings Decomposition

Figure 5 separates direct gate skips from the remaining run-level effect. Direct savings are the BO trials explicitly skipped by the gate. The BO-volume contribution captures changes in how many generated candidate programs reach Stage 2.

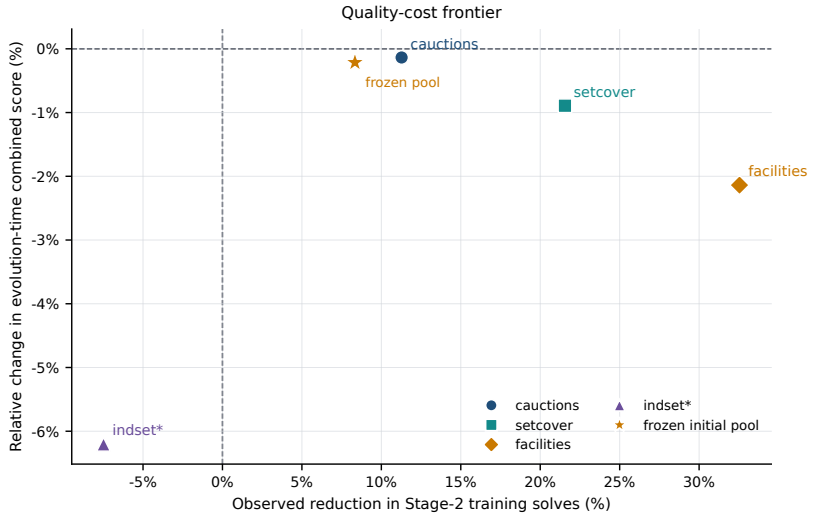

Figure 4: Quality–cost trade-off across benchmarks. The starred `indset` point has negative observed training-solve reduction but improves held-out runtime on all difficulty levels.

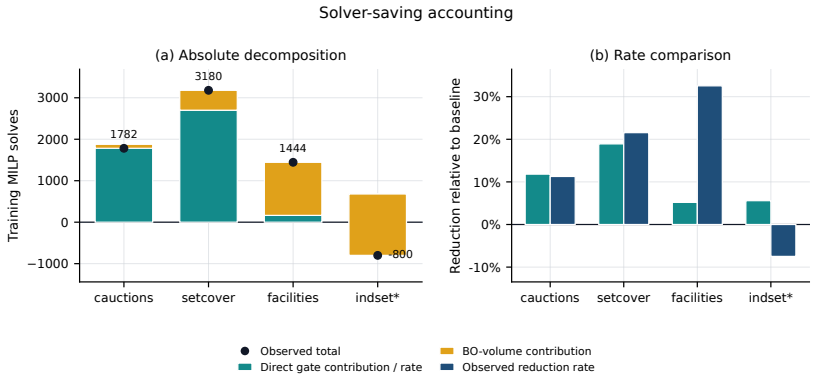

Figure 5: Stage-2 solver-saving accounting. Direct gate contribution counts skipped BO trials, while BO-volume contribution captures the remaining difference in total Stage-2 entries.

This decomposition prevents over-crediting the gate. On `cauctions`, direct and observed savings are close, so skipped BO trials explain most of the run-level reduction. On `setcover`, observed reduction is larger than direct saving because fewer candidates enter Stage 2. On `facilities`, the large observed reduction is mostly due to this candidate-count effect. On `indset`, direct skips are offset by more Stage-2 entries; the held-out result therefore matters for interpreting the run.

6.3 Gate Behavior and Robustness

The curated gate pool contains policies with different levels of conservatism: quality-first policies skip only clearly weak trials, early-stop policies reduce spending when recent BO trials show little improvement, and balanced-audit policies periodically force full evaluations to avoid silent collapse. The archive-evolved variant can select among these behaviors based on prior candidate evaluations.

The main robustness risk is that the gate becomes too aggressive and biases the Stage-2 optimizer away from useful parameter regions. We mitigate this by using conservative penalties for skipped trials and by keeping held-out evaluation ungated. The frozen-pool ablation shows that even a non-adaptive initial policy pool can produce useful savings, but it is only a single-run mechanism check.

LLM-generated candidate programs also introduce ordinary engineering noise: format-repair warnings, invalid rewrites, and Stage-1-only candidates. These events occur before gated Stage-2 evaluation and are not themselves evidence of gate behavior. For fair comparison, no-gate and gated runs share the same outer LLM generation and format-repair policy within each benchmark.

6.4 Summary

The evidence supports a focused conclusion: trial-level Stage-2 gating can reduce the training-solve cost of LLM-based MILP branching-policy search, but the benefit is benchmark-dependent and should be measured jointly with final policy quality. Direct gate savings quantify what the gate itself skips; observed reductions measure the full run-level outcome; held-out evaluation checks whether the selected final program remains useful when the gate is removed.

7 Limitations

The proposed gate reduces training-time evaluation cost, not final MILP solving cost. It is used only inside Stage-2 optimization during evolution and is disabled during held-out evaluation. A skipped trial receives conservative feedback rather than a true solver objective, so aggressive gating can bias BO away from useful parameter regions.

The effect is benchmark-dependent. `setcover` shows clean savings with little held-out change, `facilities` has large observed reduction partly from fewer Stage-2 entries, and `cauctions` degrades on hard held-out instances. The frozen-pool ablation is intentionally lightweight; larger ablations could isolate archive evolution, penalty strength, audit frequency, and minimum-evaluation budgets.

8 Conclusion

We proposed a trial-level Stage-2 evaluation gate for LLM-based MILP branching-policy search. The gate operates inside Bayesian parameter optimization, skipping selected training MILP solves while returning conservative feedback to the optimizer. It leaves the outer LLM program search and held-out evaluation protocol unchanged.

Experiments show that this lightweight mechanism can reduce Stage-2 training-solve usage while preserving competitive final policy quality in several settings. The results also show why careful accounting is necessary: direct gate savings and total observed reductions can differ when the number of Stage-2 candidates changes. More broadly, our findings suggest that for LLM-driven algorithm discovery, the evaluator itself is an important target of optimization.

Reproducibility Statement

We provide the experimental configurations, run identifiers, held-out evaluation protocols, and solver-call accounting used in our study. The appendix summarizes the benchmark settings, evolution configurations, gate-policy details, and additional per-run results needed to reproduce the main comparisons.

AI Use Statement

We used AI assistance to aid and polish writing. All experimental design choices, results, interpretations, and final manuscript content were reviewed and validated by the authors.

References

Tobias Achterberg. Scip: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009. doi: 10.1007/s12532-008-0001-1.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021. doi: 10.1016/j.ejor.2020.07.063.

Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Fredrik Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP optimization suite 8.0, 2021. URL https://arxiv.org/abs/2112.08872.

Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1437–1446, 2018.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Prateek Gupta, Maxime Gasse, Elias B. Khalil, Pawan Mudigonda, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch. In *Advances in Neural Information Processing Systems*, volume 33, pp. 18087–18097, 2020.

Zhinan Hou, Xingchen Li, Yankai Zhang, Tianxun Li, and Keyou You. Llm4branch: Large language model for discovering efficient branching policies of integer programs, 2026. URL https://arxiv.org/abs/2605.10401.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, pp. 507–523. Springer, 2011.

Elias B. Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

Andrea Lodi and Giulia Zarpellon. On learning and branching: A survey. *TOP*, 25(2):207–236, 2017. doi: 10.1007/s11750-017-0451-6.

Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Koppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618: 257–263, 2023. doi: 10.1038/s41586-023-06004-9.

Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. Solving mixed integer programs using neural networks, 2020. URL https://arxiv.org/abs/2012.13349.

Alexander Novikov, Ngan Vu, Marvin Eisenberger, Emilien Dupont, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In *Learning Meets Combinatorial Algorithms at NeurIPS 2020*, 2020. URL https://openreview.net/forum?id=IVc9hqgibyB.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, et al. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Lara Scavuzzo, Feng Yang Chen, Didier Chételat, Maxime Gasse, Andrea Lodi, Neil Yorke-Smith, and Karen Aardal. Learning to branch with tree MDPs. In *Advances in Neural Information Processing Systems*, volume 35, pp. 18514–18526, 2022.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, volume 25, 2012.

A Reproducibility Details

A.1 Benchmark Settings

Table 4 lists the benchmark instance scales used in the held-out evaluation. These settings follow the instance generator used in the LLM4Branch evaluation pipeline.

Table 4: Benchmark instance scales.

Benchmark	Easy	Medium	Hard
`setcover`	rows $= 500$, cols $= 1000$	rows $= 1000$, cols $= 1000$	rows $= 2000$, cols $= 1000$
`cauctions`	items $= 100$, bids $= 500$	items $= 200$, bids $= 1000$	items $= 300$, bids $= 1500$
`facilities`	customers $= 100$, facilities $= 100$	customers $= 200$, facilities $= 100$	customers $= 400$, facilities $= 100$
`indset`	nodes $= 750$, affinity $= 4$	nodes $= 1000$, affinity $= 4$	nodes $= 1500$, affinity $= 4$

A.2 Evolution Configurations

Table 5 summarizes the matched evolution configurations. Within each benchmark, the no-gate baseline and gated variant share the same outer iteration count, LLM configuration, Stage-2 budget, and timeout settings. The gate is the only intentional evaluator difference.

Table 5: Matched evolution configurations. Train and valid denote evaluator instance counts. Timeouts are Stage-1/Stage-2 wall-clock limits in seconds.

Benchmark	Iter.	n_{calls}	n_{initial}	Train	Valid	Timeouts
`cauctions`	200	24	6	4	8	600 / 2000
`setcover`	200	24	6	4	8	600 / 2000
`facilities`	200	16	2	4	8	1000 / 3000
`indset`	200	16	2	4	8	1000 / 2000

All comparisons use one outer evaluation worker. The maximum evaluator workers are 4 for `cauctions`/`setcover` and 8 for `facilities`/`indset`. For `indset`, the underlying SCIP time limits are 180 seconds for both training and validation solves.

A.3 Held-Out Protocol

For each completed evolution run, we extract the saved `best/best_program.py`. We then evaluate all selected best programs on the same held-out test seeds, together with the SCIP `relpscost` reference. Each benchmark and difficulty level uses 80 held-out instances.

Table 6: Post-evolution held-out protocol.

Benchmark	Programs evaluated	Levels	Instances per level
`cauctions`	5 baseline + 5 gated + `relpscost`	easy/medium/hard	80
`setcover`	3 baseline + 3 gated + `relpscost`	easy/medium/hard	80
`facilities`	3 baseline + 3 gated + `relpscost`	easy/medium/hard	80
`indset`	1 baseline + 1 gated + `relpscost`	easy/medium/hard	80

B Additional Method Details

B.1 Gate-Policy Families

Table 7 summarizes the gate-policy families used by the curated pool.

Table 7: Gate-policy families used by the curated pool.

Policy family	Main behavior	Intended role
Quality-first	Skip only clearly weak trials	Preserve final policy quality
Balanced audit	Mix skipping with periodic forced evaluation	Stable savings with safeguards
Early-stop guarded	Reduce spending in stagnant BO regions	Increase training-solve savings
Explore-preserving	Evaluate uncertain or diverse trials	Avoid premature gate collapse
Hybrid	Combine quality, uncertainty, and audit checks	Adapt across candidate types

B.2 Stage-2 Gating Algorithm

Algorithm 1 Stage-2 Bayesian optimization with trial-level evaluation gating

Input: candidate program p, BO budget T, training instances $\mathcal{T}$, gate-policy pool $\mathcal{G}$
Output: optimized parameters $\theta^\star$ and candidate metrics

1. Select gate policy $g \in \mathcal{G}$.
2. Initialize BO history $\mathcal{D}_0 \leftarrow \emptyset$.
3. For $t = 1, \ldots, T$:
   (a) Propose parameters $\theta_t \leftarrow \mathrm{BO.ask}(\mathcal{D}_{t-1})$.
   (b) Compute lightweight gate features from θ_t and $\mathcal{D}_{t-1}$.
   (c) Choose action $a_t \leftarrow g(\theta_t, \mathcal{D}_{t-1})$.
   (d) If $a_t = \mathrm{evaluate}$, run p_{θ_t} on $\mathcal{T}$ and observe y_t.
   (e) If $a_t = \mathrm{skip}$, avoid solver calls and set $y_t \leftarrow \tilde{y}_t$.
   (f) Update $\mathcal{D}_t \leftarrow \mathcal{D}_{t-1} \cup \{(\theta_t, y_t)\}$.
4. Select $\theta^\star$ from the best fully evaluated trial.
5. Return candidate metrics under the standard evaluator.
6. Update the gate archive with quality and saved-solve statistics.

C Additional Results

C.1 Full Held-Out Metrics

Table 8 reports the full held-out time, node, and gap values corresponding to Table 2. Each time and node entry is a geometric mean over 80 held-out instances.

C.2 Additional Diagnostic Figures

Table 8: Full held-out results. Each cell reports baseline/gated values.

Benchmark	Level	Programs	Time gmean (s)	Nodes gmean	Gap avg	Time change
cauctions	easy	5 / 5	1.328 / 1.322	93.57 / 99.03	0.0000 / 0.0000	-0.44%
cauctions	medium	5 / 5	19.079 / 18.360	1934.74 / 2010.23	0.0000 / 0.0000	-3.77%
cauctions	hard	5 / 5	205.909 / 235.829	24561.75 / 30079.35	0.0000 / 0.0000	+14.53%
setcover	easy	3 / 3	4.129 / 4.105	185.87 / 185.39	0.0000 / 0.0000	-0.57%
setcover	medium	3 / 3	53.324 / 54.119	3283.73 / 3317.29	0.0000 / 0.0000	+1.49%
setcover	hard	3 / 3	1515.965 / 1506.509	67227.06 / 66597.33	0.0182 / 0.0187	-0.62%
facilities	easy	3 / 3	31.339 / 31.247	269.81 / 264.09	0.0000 / 0.0000	-0.29%
facilities	medium	3 / 3	129.449 / 129.194	404.86 / 400.41	0.0000 / 0.0000	-0.20%
facilities	hard	3 / 3	605.408 / 591.479	466.33 / 461.17	0.000015 / 0.000013	-2.30%
indset	easy	1 / 1	26.918 / 26.112	767.51 / 851.83	0.0000 / 0.0000	-3.00%
indset	medium	1 / 1	212.567 / 190.364	8807.87 / 8715.87	0.0015 / 0.0024	-10.45%
indset	hard	1 / 1	3044.181 / 2504.671	60549.89 / 57926.53	0.0176 / 0.0180	-17.72%

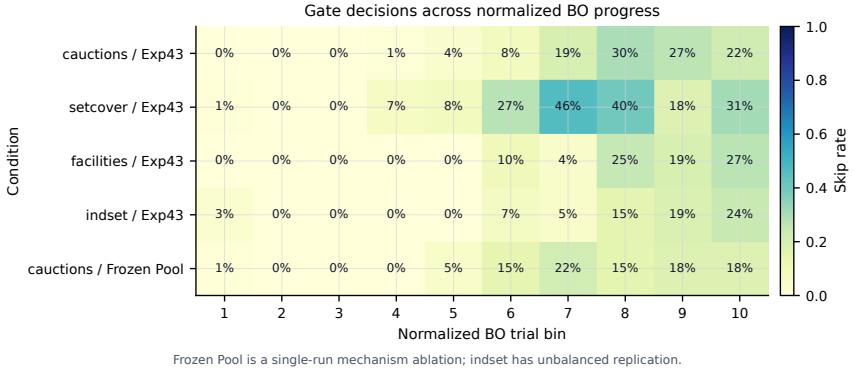

Figure 6: Gate skip behavior over BO trials. Darker cells indicate skipped trials.

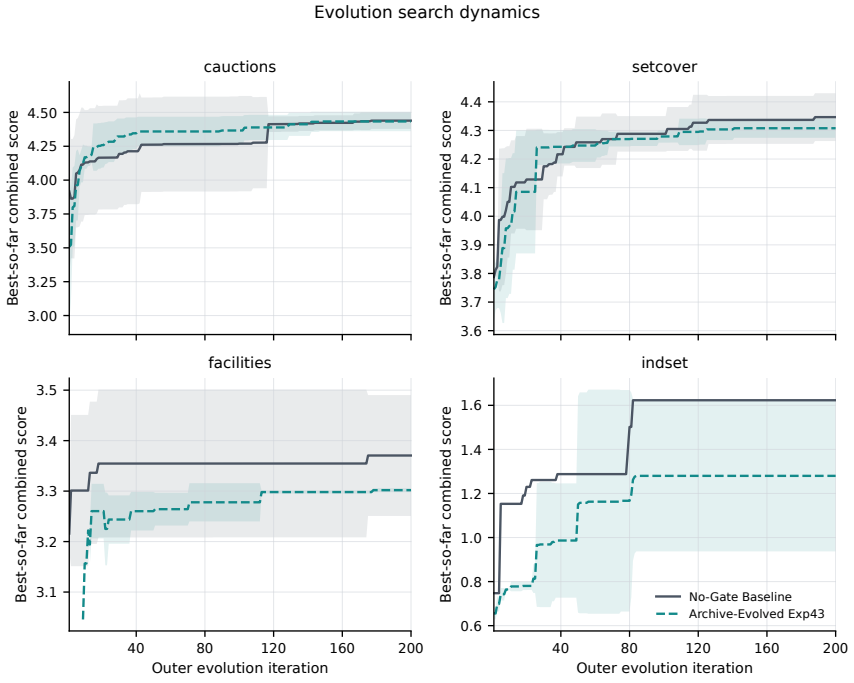

Figure 7: Best-so-far evolution curves over outer iterations.