

From Natural Language to Solver-Ready Power System Optimization: An LLM-Assisted, Validation-in-the-Loop Framework

Yunkai Hu

*Department of Electrical and
Computer Engineering
University of Southern California
Los Angeles, CA
yunkaihu@usc.edu*

Tianqiao Zhao

*Department of Electrical
Engineering
University of Texas at Arlington
Arlington, TX
tianqiao.zhao@uta.edu*

Meng Yue

*Interdisciplinary Science
Department
Brookhaven National Laboratory
Upton, NY
yuemeng@bnl.gov*

Abstract—This paper introduces a novel Large Language Models (LLMs)-assisted agent that automatically converts natural-language descriptions of power system optimization scenarios into compact, solver-ready formulations and generates corresponding solutions. In contrast to approaches that rely solely on LLM to produce solutions directly, the proposed method focuses on discovering a mathematically compatible formulation that can be efficiently solved by off-the-shelf optimization solvers. Directly using LLMs to produce solutions often leads to infeasible or suboptimal results, as these models lack the numerical precision and constraint-handling capabilities of established optimization solvers. The pipeline integrates a domain-aware prompt and schema with an LLM, enforces feasibility through systematic validation and iterative repair, and returns both solver-ready models and user-facing results. Using the unit commitment problem as a representative case study, the agent produces optimal or near-optimal schedules along with the associated objective costs. Results demonstrate that coupling the solver with task-specific validation significantly enhances solution reliability. This work shows that combining AI with established optimization frameworks bridges high-level problem descriptions and executable mathematical models, enabling more efficient decision-making in energy systems.

Index Terms—Power system optimization, large language models, natural language processing, power system operational decisions

I. INTRODUCTION

Classical optimization remains the backbone of power-system decision making. Unit commitment (UC), economic dispatch, and optimal power flow are routinely expressed as mixed-integer and nonlinear programs and solved with mature engines that provide feasibility enforcement, dual information, and—where applicable—optimality certificates [1]. Decades of work on decomposition, cutting planes, Lagrangian relaxations, and stochastic/robust variants have pushed these formulations to industrial scale while preserving auditability and model transparency [2]. The principal drawback is the front end: translating evolving operational policies into mathematically precise, solver-compatible models is labor-intensive and error-prone, which slows iteration and limits accessibility for non-experts.

AI/ML approaches have sought to reduce this burden or accelerate solve times [3]. Supervised surrogates approximate AC-OPF mappings or screen active constraint sets [4], [5]; reinforcement learning (RL) has been explored for scheduling and corrective actions [6]; and “learning to optimize” techniques provide warm starts, learned cuts, or heuristics embedded in the solver loop [7], [8]. These methods can deliver speedups or good average-case performance within the training distribution, but they often trade certainty for speed: constraint violations can slip through, guarantees are scarce, and behavior may degrade under distribution shift. Even when constraints are enforced via projection or penalty terms, certification and interpretability remain challenging, which limits adoption in mission-critical operations.

More recently, Large Language Models (LLMs) have been used to “solve” grid operations problems directly from text by reasoning step-by-step and outputting candidate schedules or dispatch vectors [9]–[12]. While appealingly simple, this direct mode collides with the numerical precision and strict feasibility demands of power-system optimization. Directly using LLMs to produce solutions often leads to infeasible or suboptimal results, as these models lack the numerical precision and constraint-handling capabilities of established optimization solvers. As problem scale grows and constraints tighten—e.g., minimum up/down times, ramping, spinning reserves, and network limits—the outputs become inconsistent and non-reproducible, and there is no principled way to certify optimality or even feasibility.

Our work takes a different path by using the LLM not as a solver but as a *formulation generator* [13]: an agent built on LLMs that converts natural-language scenario descriptions into compact, solver-ready mathematical programs and then delegates computation to off-the-shelf solvers. The pipeline is built around domain-aware prompting and a structured schema so that the LLM emits typed variables, objective terms, and constraints with explicit data bindings rather than free-form text. A validation-and-repair layer detects omissions and inconsistencies—such as missing reserve coupling, incorrect ramp logic, or commitment integrality—and engages the LLM

to patch the model before any optimization is attempted. By deferring numerical work to mature solvers, the agent preserves feasibility and leverages optimality certificates while still delivering the usability of natural-language interaction. In addition, we contribute a *solution-enhancement* layer that accelerates branch-and-cut without altering problem semantics: a novel GNN-based policy proposes branching priorities for commitment binaries [14], and a lightweight LLM configures cut separators per instance within a guarded, solver-parameter interface [15]. Both mechanisms are optional, preserve correctness, and complement the translation layer—together yielding reliable schedules and costs faster, in a manner that is auditable, extensible, and readily generalizable beyond UC.

II. PROBLEM FORMULATION

We pose the task as translating a natural-language scenario S into a solver-ready mixed-integer program $\mathcal{M}(S) = \{\text{sets, parameters, variables, objective, constraints}\}$. From S , our agent extracts entities (e.g., generators I , horizon T), attributes (capacities, costs, ramps, minimum up/down times), system requirements (demand, reserves), and policies (must-run, mutual exclusivity). An LLM produces a typed schema that binds these elements to a canonical template; a validator enforces completeness and logical consistency and, if needed, triggers iterative repair before passing $\mathcal{M}(S)$ to a MILP solver.

Taking unit commitment (UC) as an example, the model determines each unit’s on/off status and dispatch over T to meet demand at minimum cost [16]. UC combines binary commitment variables with continuous dispatch, accounting for variable, start-up, and no-load costs. Constraints include power balance with reserves, capacity limits, minimum up/down times, ramp limits, network deliverability, fuel or emissions limits, and coupling to renewables or storage. Although problem size grows with $|I| \times |T|$, modern MILP solvers handle moderate instances efficiently with robust encodings.

ML/AI accelerators [17]—such as surrogates, RL schedulers, and learning-to-branch/cut—can reduce runtime but often lack feasibility or optimality guarantees, degrade under distribution shift, and require case-specific tuning [14]. LLM-based code generation improves accessibility but is brittle for UC: prompts often miss or mis-specify constraints [18], while direct LLM solution generation is non-deterministic, scales poorly, and offers no guarantees. We address these issues with a domain-specialized, LLM-assisted translator that maps S to a validated, solver-compatible UC MILP with schema checks and iterative repair, yielding optimal or near-optimal schedules—or precise diagnostics when inputs conflict—while preserving solver rigor.

III. LLM-ASSISTED OPTIMIZATION AGENT OVERVIEW

The agent transforms a natural-language specification into a solver-ready model and a validated solution. As illustrated in Fig. 1, a utility operator submits a scenario; an LLM-driven parser performs schema-aware parameter synthesis (entities, attributes, policies) to produce a typed model skeleton; the agent instantiates a canonical MILP formulation and hands it

to the solver for execution; and the resulting schedules, costs, and diagnostics are returned through user-facing visualizations. Throughout, a validation layer enforces unit/type/shape checks and tests feasibility against the specification; on any error or violation, the agent enters a diagnosis-and-repair loop that provides targeted feedback to the LLM, amends parameters or constraints, and re-solves until success or a small iteration budget is reached.

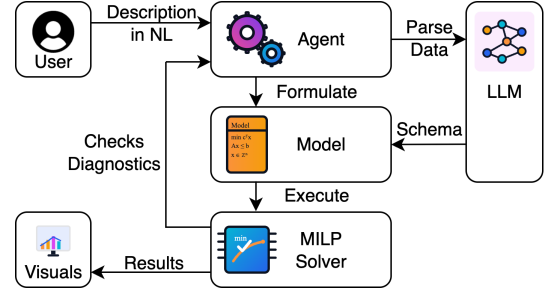


Fig. 1: High-level Framework of LLM-assisted Agent.

Fig. 2 outlines the end-to-end architecture: a domain expert supplies a scenario in plain text; an LLM-driven parser synthesizes a typed parameter schema; automatic checks verify sizes, units, and completeness, triggering an iterative repair loop when needed; a formulation module maps the schema to a solver-ready MILP; an optional guidance stage configures solver tactics; a commercial solver computes a solution; and a diagnostics loop—invoked only on failure or detected violations—pinpoints whether issues arise from code or data, repairs them, and re-solves. Successful runs produce optimized schedules and metrics, which are rendered as human-readable reports and visualizations.

A. Natural Language Processing and Parameter Synthesis

The workflow begins with a free-form scenario supplied by the user—often semi-structured as a generator table followed by operational notes and a demand profile. A light preprocessing pass standardizes units and resolves obvious inconsistencies, after which an LLM-driven parser performs schema-aware extraction of entities (generators and time horizon), attributes (capacities, costs, ramps, minimum up/down times), system requirements (load and reserves), and policy statements (must-run windows, mutual exclusivity). Rather than emitting ad hoc text, the LLM is instructed to return a typed parameter object with explicit cardinalities and array shapes. A validator then enforces type, range, and shape invariants and checks basic consistency (e.g., $P_i^{\min} \leq P_i^{\max}$, feasible ramp rates). If a check fails, targeted diagnostics are fed back to the LLM and the parameters are repaired before proceeding. By the end of this stage, the system holds the original description and a unit-harmonized, horizon-aligned schema ready for modeling.

B. MILP Model Formulation

Given the validated schema, the agent compiles a solver-ready model by prompting the LLM with a canonical UC

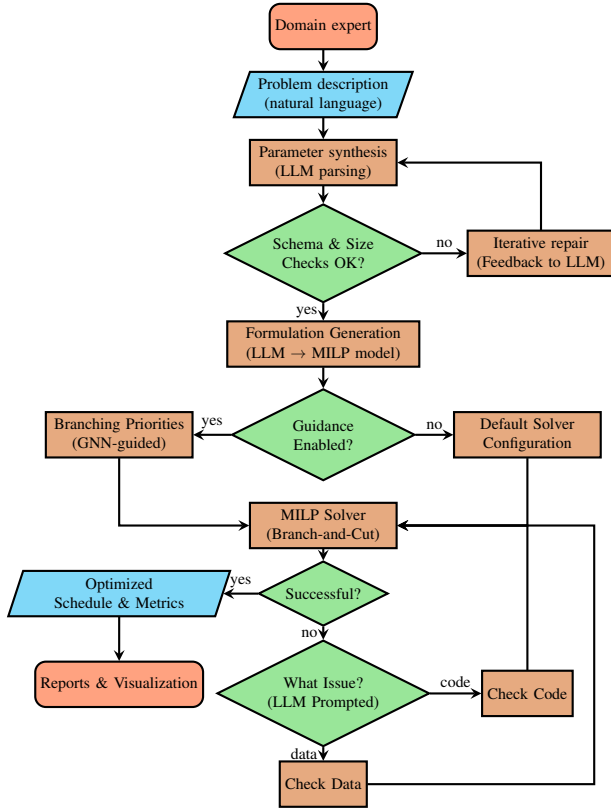


Fig. 2: End-to-end Pipeline.

template, the extracted data, and a few illustrative exemplars. The LLM first produces a compact, pseudo-mathematical specification that maps typed fields to variables, objective terms, and constraint families, and then renders executable code. In our implementation, code generation targets the Gurobi Python API to enable immediate execution and rapid error surfacing. Domain guidance is embedded directly in the prompt—such as enforcing capacity coupling $p \leq P_{\max}u$, respecting minimum up/down time encodings, and applying ramp trajectories at start-up/shut-down—so that common omissions are avoided without sacrificing generality.

C. Optimization and Solution Retrieval

The generated model is passed to the MILP solver together with the synthesized parameters. If the solver returns an optimal or feasible solution, the agent validates the schedules against the user’s specification to catch any residual mismatches, then assembles a concise report with per-unit commitment and dispatch, start-up/shut-down events, total cost, and reserve performance for delivery via a web dashboard. If the solver reports infeasibility or an error, or if post-solve validation flags a violation, the agent enters a diagnosis-and-repair loop: structured solver feedback (e.g., infeasible reserve hours, contradictory exclusivity rules, ramp violations around transitions) is summarized for the LLM, which proposes targeted edits to parameters or constraints; the instance is recompiled and re-solved. This loop is capped at

a small iteration budget (five in our experiments) to prevent drift while providing enough room for corrective refinement.

IV. SOLUTION ENHANCEMENT

To reduce time-to-solution without altering problem semantics, the agent can augment the baseline branch-and-cut process with learned guidance that operates strictly through solver interfaces. Because the mathematical model is unchanged, feasibility and optimality guarantees remain intact. Two complementary options are supported: a graph neural network (GNN) that proposes instance-specific branching priorities for commitment binaries, and a lightweight LLM policy that configures cut separators prior to the solve. Both are optional and can be toggled per run.

A. GNN-Based Branching Policy

Branching choices largely determine search-tree growth. Building on the learning-to-branch paradigm [14], we represent the root-node MILP as a bipartite graph over variables and constraints with features drawn from the LP relaxation and problem structure, and use a message-passing GNN to score binary variables. The model is trained offline on representative UC instances with rank-based objectives derived from strong-branching proxies or relative node-count reductions on calibration trees. At inference, scores are mapped to integer priorities (e.g., Gurobi BranchPriority), biasing the search toward UC-salient structure while preserving completeness and optimality. Empirically, this reduces explored nodes and wall-clock time on UC benchmarks.

B. LLM-Based Separator Configuration

Cut management is the second major lever in branch-and-cut. Following recent LLM-guided approaches [15], the agent summarizes inexpensive pre-solve diagnostics—variable/constraint counts, sparsity and row/column statistics, proportions of constraint families, and root-LP gap—and requests a configuration over a constrained, whitelisted action space (enabling standard separators and setting pass counts/aggressiveness). A guard layer validates types and ranges and reverts to conservative defaults on any anomaly. The configuration is applied once, with an optional single re-evaluation if early progress stalls. This policy complements learned branching and is particularly helpful when the initial relaxation is weak.

C. Prompt Template

Figure 3 illustrates the lightly structured prompt used for compilation. The template names typed fields for parameters, declares the variables to be created, and invokes canonical constraint and objective fragments, allowing the LLM to assemble a complete model deterministically from the schema. In effect, the prompt serves as a contract between natural-language intent and executable artifacts: parameters remain immutable data, variables are decision quantities, constraints encode feasibility and logic, and the objective aggregates cost terms. This structure both streamlines code generation and simplifies downstream validation.

Prompt Template:

We operate New England’s regional grid over 24 one-hour periods.
 Goal: minimize total production cost while meeting the hourly demand.
 Generator data
 G001: min-up 48 h, min-down 48 h,
 variable cost \$13.54/MWh

 Hourly demand (MW):
 20198,18476,17959,16238,15112,14861,14948,15871,
 15633...

(a)

Parameters: MinUpTime = data["MinUpTime"]
 Variables: powerOutput = model.addVars(...
 Constraints: model.addConstr
 (powerOutput[g, t] >= Pmin[g])
 Objectives: model.setObjective
 (...powerOutput[g, t] * VariableCost[g]...

(b)

Fig. 3: Prompt Template.

V. CASE STUDY

In this study, we take GPT-4 as the backbone agent API [19] and procedurally generate UC instances in a MILP-Evolve style [20], treating UC as a parametric family rather than hand-crafted cases. Each instance is a thermal UC MILP with 50–100 units partitioned into base-load, mid-merit, and peakers to induce heterogeneous costs, ramps, and start-up penalties. Two horizons are considered: a 7-day hourly horizon with weekday/weekend structure, and a 1-month daily horizon with demand and operations aggregated to daily resolution. Demand profiles combine smooth seasonal/diurnal components with stochastic perturbations and are scaled to target reserve margins. In addition, we generate a small-scale benchmark set consisting of much smaller UC instances, with only 3, 10, 30, or 60 generator over 24-hour (one day) hourly horizon. These smaller cases help evaluate our approach’s performance across varying problem scales. All datasets are split by disjoint random seeds into a runtime cohort (to compare GNN-guided versus default solver speed) and an accuracy cohort (to compare objective values between expert formulations and our system).

A. Performance Metrics and Evaluation

Each instance in the runtime cohort is solved twice—once with default solver settings and once with our GNN-guided branching (optionally with variable fixation)—and wall-clock time is recorded. We report mean and median runtime, empirical runtime distributions, and the share of instances on which guidance is faster to quantify consistency; timeouts, when present, are counted at the cap and reported separately.

Accuracy is evaluated by comparing the *optimality gap* (MIPGap) reported by the solver at termination under identical stopping criteria (same time limit and tolerances) from the expert-written formulation and for our system-generated

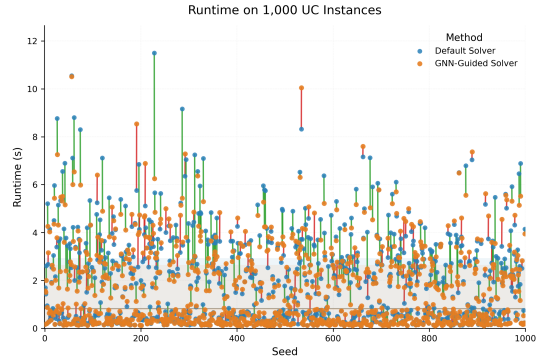


Fig. 4: Comparison Results: the Improved Runtime.

formulation. For the main benchmarks, we use 300 representative instances; for the small-scale benchmarks, we use 100 instances. We adopt the standard definition for minimal quality gap $= \frac{|z_P - z_D|}{z_P}$, where z_P is the incumbent objective bound and z_D is the best dual bound at termination. All solutions produced by our system are verified feasible with respect to the original MILP constraints, and infeasible runs (if any) are excluded from gap statistics and reported separately as validation failures.

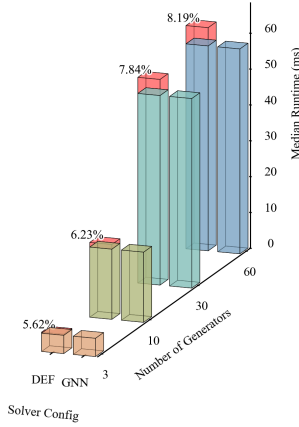
B. Results: Solver Runtime Performance

We quantify speedup using the per-instance ratio $\tau = t_{\text{guided}}/t_{\text{default}}$ (values < 1 indicate improvement). On the 1,000-instance benchmark (Fig. 4), the default solver averages ≈ 1.79 s per instance, whereas our GNN guidance reduces this to ≈ 1.64 s, a $1.09\times$ ($\sim 9\%$) gain with a similarly improved median. The histogram of τ concentrates below 1.0, with a light tail above 1.0 driven by very easy cases where scoring overhead can offset benefits.

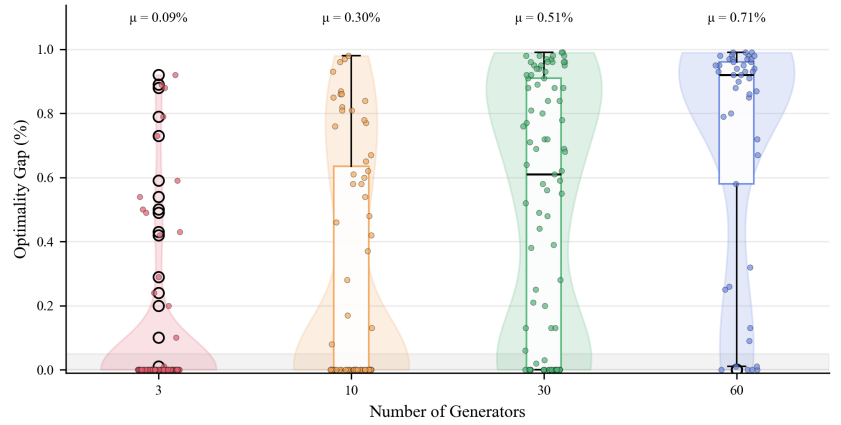
We observe the same pattern on 24-hour UC benchmarks spanning 3–60 generators (5,000 instances; Fig. 5a). Speedups persist across all size bins and become more pronounced as instance size grows, where search dominates overhead; for the smallest bins, improvements are modest and occasionally neutral. These results indicate that learned branching priorities can accelerate MILP solving in the energy domain without modifying the underlying engine. Even a $\sim 10\%$ reduction in wall-clock time is meaningful for large-scale or time-sensitive operations, and leaves room for further gains via tuned models or adaptive refresh policies.

C. Results: Solver Quality and Accuracy

We assess solution quality using the solver-reported optimality gap (MIPGap) under identical tolerances and time limits. Fig. 5b reports the small-scale set with four configurations (3, 10, 30, and 60 generators; ~ 100 cases each over 24 hours). Across all four sizes, gaps cluster near zero with compact interquartile ranges (IQR) and only a few outliers. A similar pattern is observed on the 300-instance quality cohort for the 50-100 generator benchmarks, indicating consistent solution quality across scales.



(a) Median runtime by generator count



(b) Optimality-gap distributions

Fig. 5: Performance on Small-Scale UC Benchmarks (3/10/30/60 generators).

VI. CONCLUSION

We presented an end-to-end system that translates natural-language UC descriptions into solver-ready MILP models with schema-aware parameter synthesis, constraint templates, and iterative repair. Coupled with solver-in-the-loop validation, the approach bridges domain intent and formal optimization, achieving 100% success on our test scenarios and clearly outperforming a single-pass LLM baseline in formulation fidelity. To accelerate computation, we add GNN-guided branching that prioritizes critical decisions. Together, LLM-driven modeling and learned solver guidance are complementary—LLMs provide flexibility and domain alignment, while the MILP solver and learned policies preserve rigor and efficiency—making AI-assisted UC practical for operational settings.

Future work will extend coverage beyond thermal UC to multi-resource scheduling (hydro, storage, and variable renewables), improve first-pass accuracy via domain-tuned prompting, and explore tighter LLM–solver coupling for per-instance configuration.

REFERENCES

- [1] A. J. Wood, B. F. Wollenberg, and G. B. Sheblé, *Power Generation, Operation, and Control*, 3rd ed. Wiley, 2013.
- [2] Y. Yang and L. Wu, “Machine learning approaches to the unit commitment problem: Current trends, emerging challenges, and new strategies,” *The Electricity Journal*, vol. 34, no. 1, p. 106889, 2021.
- [3] A. Perera, P. Wickramasinghe, V. M. Nik, and J.-L. Scartezini, “Machine learning methods to assist energy system optimization,” *Applied energy*, vol. 243, pp. 191–205, 2019.
- [4] N. Guha, Z. Wang, M. Wytock, and A. Majumdar, “Machine learning for ac optimal power flow,” *arXiv preprint arXiv:1910.08842*, 2019.
- [5] X. Pan, M. Chen, T. Zhao, and S. H. Low, “Deepopf: A feasibility-optimized deep neural network approach for ac optimal power flow problems,” *arXiv preprint arXiv:2007.01002*, 2020.
- [6] B. Huang, T. Zhao, M. Yue, and J. Wang, “Bi-level adaptive storage expansion strategy for microgrids using deep reinforcement learning,” *IEEE Transactions on Smart Grid*, vol. 15, no. 2, pp. 1362–1375, 2024.
- [7] J. Qin and N. Yu, “Solve large-scale unit commitment problems by physics-informed graph learning,” *IEEE Transactions on Power Systems*, 2025.
- [8] L. Yang, P. Li, S. Chen, and H. Zheng, “Stable variable fixation for accelerated unit commitment via graph neural network and linear programming hybrid learning,” *Applied Sciences*, vol. 15, no. 8, p. 4498, 2025.
- [9] Y. Cheng, H. Zhao, X. Zhou, J. Zhao, Y. Cao, C. Yang, and X. Cai, “A large language model for advanced power dispatch,” *Scientific Reports*, vol. 15, no. 1, p. 8925, 2025.
- [10] C. Huang, S. Li, R. Liu, H. Wang, and Y. Chen, “Large foundation models for power systems,” in *2024 IEEE Power & Energy Society General Meeting (PESGM)*. IEEE, 2024, pp. 1–5.
- [11] X. Zhou, H. Zhao, Y. Cheng, Y. Cao, G. Liang, G. Liu, W. Liu, Y. Xu, and J. Zhao, “Elecbench: a power dispatch evaluation benchmark for large language models,” *arXiv preprint arXiv:2407.05365*, 2024.
- [12] L. Zhang, W. Ji, Y. Zhao, D. Huang, and B. Qian, “Review on application and challenges of large language models in power grids,” *IEEE Access*, 2025.
- [13] A. AhmadiTeshnizi, W. Gao, H. Brunborg, S. Taleai, C. Lawless, and M. Udell, “Optimus-0.3: Using large language models to model and solve optimization problems at scale,” *arXiv preprint arXiv:2407.19633*, 2024.
- [14] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi, “Exact combinatorial optimization with graph convolutional neural networks,” *Advances in neural information processing systems*, vol. 32, 2019.
- [15] C. Lawless, Y. Li, A. Wikum, M. Udell, and E. Vitercik, “Llms for cold-start cutting plane separator configuration,” in *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2025, pp. 51–69.
- [16] M. Carrión and J. M. Arroyo, “A computationally efficient mixed-integer linear formulation for the thermal unit commitment problem,” *IEEE Transactions on Power Systems*, vol. 21, no. 3, pp. 1371–1378, 2006.
- [17] Y. Bengio, A. Lodi, and A. Prouvost, “Machine learning for combinatorial optimization: A methodological tour d’horizon,” *arXiv preprint arXiv:1811.06128*, 2021.
- [18] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [19] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [20] S. Li, J. Kulkarni, I. Menache, C. Wu, and B. Li, “Towards foundation models for mixed integer linear programming,” *arXiv preprint arXiv:2410.08288*, 2024.