OptiMUS: Scalable Optimization Modeling with (MI)LP Solvers and Large Language Models

Ali AhmadiTeshnizi [1] **Wenzhi Gao** [2] **Madeleine Udell** [1 2]

Abstract

Optimization problems are pervasive in sectors from manufacturing and distribution to healthcare. However, most such problems are still solved heuristically by hand rather than optimally by state-of-the-art solvers because the expertise required to formulate and solve these problems limits the widespread adoption of optimization tools and techniques. This paper introduces OptiMUS, a Large Language Model (LLM)-based agent designed to formulate and solve (mixed integer) linear programming problems from their natural language descriptions. OptiMUS can develop mathematical models, write and debug solver code, evaluate the generated solutions, and improve its model and code based on these evaluations. OptiMUS utilizes a modular structure to process problems, allowing it to handle problems with long descriptions and complex data without long prompts. Experiments demonstrate that OptiMUS outperforms existing state-of-the-art methods on easy datasets by more than 20% and on hard datasets (including a new dataset, NLP4LP, released with this paper that features long and complex problems) by more than 30%. The implementation and the datasets are available at https://github.com/teshnizi/OptiMUS.

1. Introduction

Optimization problems are common in many fields such as operations, economics, engineering, and computer science. Important applications of optimization include reducing energy costs in smart grids, improving supply chains, and increasing profits in algorithmic trading (Singh, 2012; Antoniou & Lu, 2007). Major advances in optimization algorithms over the last several decades have led to reliable and efficient optimization methods for a wide variety of structured optimization problems, including linear programming (LP) and mixed-integer linear programming (MILP), among many others. Unfortunately, optimization modeling, transforming a business problem into a mathematical optimization problem, still requires expert knowledge. According to a recent survey, 81% of Gurobi's commercial solver users have advanced degrees, with 49 % of them holding a degree in operations research (Gurobi Optimization, 2023). This expertise gap prevents many organizations from using optimization, even when it could significantly improve their operations. Examples include inventory management in supermarkets, patient operations in hospitals, transportation policies in small municipalities, energy management in local solar farms, and operations in small businesses or NGOs (Saghafian et al., 2015; Aastrup & Kotzab, 2010; Yao et al., 2020; Shakoor et al., 2016). Automating optimization modeling would allow sectors that cannot afford to have access to optimization experts to improve efficiency using optimization techniques. Large language models (LLMs) offer a promising way to make optimization more accessible. LLMs have demonstrated the ability to understand, generate, and interpret natural language for many tasks. In the optimization domain, LLMs can make it easier to formulate problems and obtain solutions, making expert-level optimization more accessible (Ramamonjison et al, 2023) However, several challenges remain before LLMs can reliably model real-life optimization problems:

- **Ambiguous Terms:** An optimization problem can be described in many ways. For example, a user might use different terms (e.g. vehicle vs. car vs. truck vs. carrier), notations (e.g. *price* and *capacity* vs. p and c vs. x and y), or omit common-sense assumptions (e.g. capacity of a vehicle is non-negative, number of employees is an integer, etc.). Moreover, defining the right variables can be a challenge. For instance, information flow through a network requires a different set of variables than phys-

[1]Department of Management Science and Engineering, Stanford University, CA, USA [2]Institute for Computational and Mathematical Engineering, Stanford University, CA, USA. Correspondence to: Ali AhmadiTeshnizi <teshnizi@stanford.edu>.

Proceedings of the 41^{st} International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

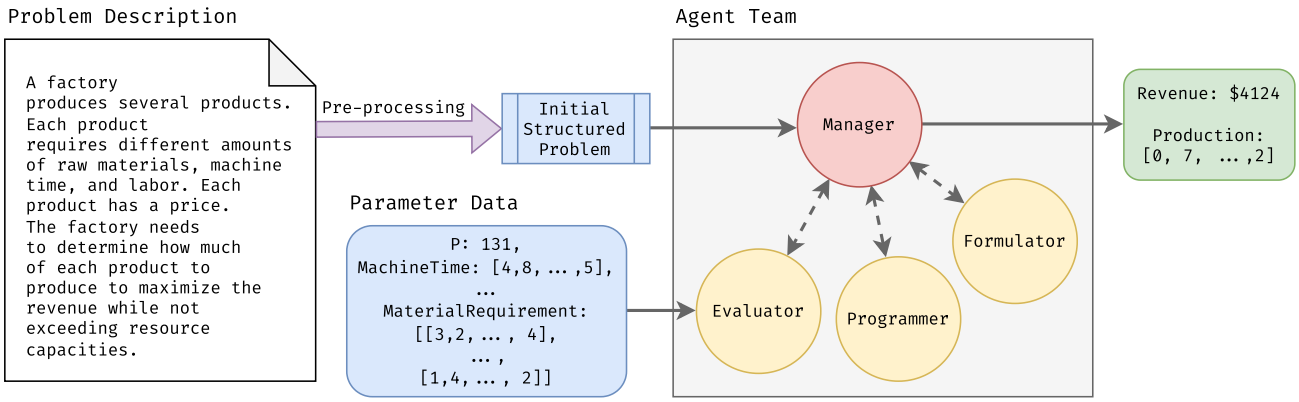

Figure 1. OptiMUS uses a structured sequence of LLM agents to effectively model and solve optimization problems. First, the natural language representation of the problem is preprocessed into a structured problem. Then, a team of agents iteratively augments the structured problem with a connection graph, mathematical formulations of each clause, and code for each clause. The agents continue work until the problem is solved. (Dashed lines represent communications that can occur multiple times.)

ical goods, as the quantity of information need not be conserved.

- **Long Problem Descriptions:** LLMs have a limited context size. However, real-world problems can be long and complex: for example, the energy system problem in (Holzer et al., 2023) has a 60-page documentation. Even for long-context models, performance decreases substantially as the input context grows (Liu et al., 2023). Consequently, LLMs tend to make more mistakes as the length of the problem description increases and perform poorly on complex problems.
- **Large Problem Data:** The specification of an optimization problem often involves large amounts of data, such as customer attributes or sales of goods. Previous approaches to optimization modeling using LLMs, which pass numerical data to the LLM directly, are thus restricted to the simplest of toy problems. Once the size of the problem data grows beyond a certain limit, passing the whole data to the LLM is not feasible anymore.
- **Unreliable Outputs:** The solutions provided by LLMs are not always reliable. The generated code may be incorrect or even not executable. It is especially challenging to verify the solution when the code runs, but the output is incorrect. For instance, if the code runs and claims that the problem is unbounded, perhaps a constraint has been accidentally omitted from the formulation.

Contributions. This paper develops a novel perspective on optimization modeling that addresses each of these limitations and makes the following contributions:

- Existing datasets for natural language optimization modeling are too easy to capture the challenge of long problem descriptions and large problem data. This work introduces NLP4LP, an open source dataset of 67 complex optimization problems. Table 1 compares NLP4LP to existing datasets and Section 4.1 describes NLP4LP.
- We develop a modular, LLM-based agent to model and solve optimization problems, which we call OptiMUS. OptiMUS beats the previous state-of-the-art methods on existing datasets by over 20% and on our more challenging dataset by 30%. OptiMUS employs a novel connection graph that allows it to process each constraint and objective independently. Using this connection graph, and separating data from the problem description, OptiMUS can solve problems with long descriptions and large data files without excessively long prompts.

Structure of the Paper This paper is organized as follows: Section 2 discusses the background and related work; Section 3 describes the details of our LLM-based optimization agent; Section 4 discusses the datasets and presents the experiments and analysis; Section 5 concludes the paper with future directions and implications. The appendix includes prompts, details on the experiments' setup, and further analysis.

2. Background and Related Work

Optimization problems are mathematically defined by an objective function and a set of constraints. For example, an

MILP can be written mathematically as

$$\begin{aligned} \underset{\{x_j\}}{\text{minimize}} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \ (\leq, =, \geq)\ b_i, i = 1, \dots m \\ & l_j \leq x_j \leq u_j, j = 1, \dots, n \\ & x_j \in \mathbb{Z}, j \in \mathcal{I} \end{aligned}$$

An optimization workflow consists of **1)** formulating an optimization problem in mathematical form by identifying its objective and constraints, and then **2)** solving the realization of problem from real data, generally using code that calls an optimization solver.

Progress in LLMs. Recent progress in Natural Language Processing (NLP) has led to the development of large language models (LLMs) useful for tasks such as answering questions, summarizing text, translating languages, and coding (OpenAI, 2023; Touvron et al., 2023; Chowdhery et al., 2022; Wei et al., 2023; Gao et al., 2023; Borgeaud et al., 2022). Connections to other software tools extend the reach and accuracy of LLMs, as demonstrated by plugins for code writing and execution (Paranjape et al., 2023; Wei et al., 2023). (Yang et al., 2023) use LLMs to directly generate solutions to optimization problems without calling traditional solvers through prompt optimization to improve performance. The approach is limited to small problems since the performance of LLMs degrades as the input context grows, even for explicitly long-context models (Liu et al., 2023).

Chatbots for Optimization. In a recent paper, Chen et al. (2023) developed a chatbot to help users detect and fix infeasible optimization problems expressed in `Pyomo` code and servers as an AI assistant rather than as a solver. Li et al. (2023) designed a chatbot to answer natural-language queries about an optimization model. Alibaba Cloud (2022) also developed a chatbot to facilitate optimization modeling, but there is no public paper or documentation available on it.

Constraint Learning and Automated Modeling. The detection of constraints from structured representations (Akgun et al., 2011) or natural language descriptions (Kiziltan et al., 2016) has been a significant research focus in constraint programming. To enhance accuracy, constraint learning typically begins with a structured system of prespecified constraint types and syntax (Beldiceanu & Simonis, 2012; 2016; Bessiere et al., 2017; De Raedt et al., 2018). These system-based approaches are stable, explainable, and perform well when the problem aligns with the corpus. However, they are less flexible when addressing new problems, making large language models (LLMs) a valuable complement in this context.

Benchmark-driven Optimization Modeling. More closely related to our approach, (Ramamonjison et al, 2023) introduced a dataset of 1101 natural language representations of LP problems. They proposed a two-stage mapping from the natural-language representation to the problem formulation using an intermediate representation. (Ramamonjison et al, 2022) designed a system to simplify and improve the modeling experience for operations research, but did not offer an end-to-end solution. (Xiao et al., 2024) presented a multi-agent cooperative framework to automatically model and program complex operation research (OR) problems, and evaluated it on NL4Opt and another more complex dataset they curate.

Traditional MILP solvers generally benchmark against the `MIPLIB` benchmark, which offers a diverse collection of MILP problems in standard form. Unfortunately, most of these problems are not associated with a natural-language description, and so cannot be used to study optimization modeling as we do in this paper.

3. Methodology

This section details the design of OptiMUS. See Figure 1 for an illustration. The problem presented in Figure 2 serves as a running example. OptiMUS starts with a natural language description of the optimization problem. The problem is first preprocessed to extract the parameters, constraints, objective function, and background information. Then OptiMUS uses a multi-agent framework to process and solve the structured problem. Appendix E includes all prompts used in OptiMUS. For brevity, we use the word *clause* to refer to a constraint or objective.

3.1. Structured Problem

The OptiMUS preprocessor converts a natural language description of the problem into a *structured problem* (Figure 2) with the following components:

- **Parameters**: A list of parameters of the optimization problem. Each parameter has three components: **1)** symbol, **2)** shape, and **3)** text definition. OptiMUS can choose symbols, infer the shape, and define the parameters if they are not explicitly included in the problem statement. Importantly, numerical data that may be included in the problem statement is omitted from the parameters and stored for later use. This ensures that the parameters are short and easy to include in future prompts.
- **Clauses**: A list of the *clauses* (objective and constraints) of the optimization problem. The preprocessor initializes each clause with its natural language description. Later

Table 1. A comparison on different aspects of complexity for various datasets. The unit for description length is characters

Dataset	Description Length	Instances (#MILP)	Multi-dimensional Parameters
NL4Opt	518.0 ± 110.7	1101 (0)	×
ComplexOR	497.1 ± 247.5	37 (12)	✓
NLP4LP (Ours)	908.9 ± 504.6	67 (13)	✓

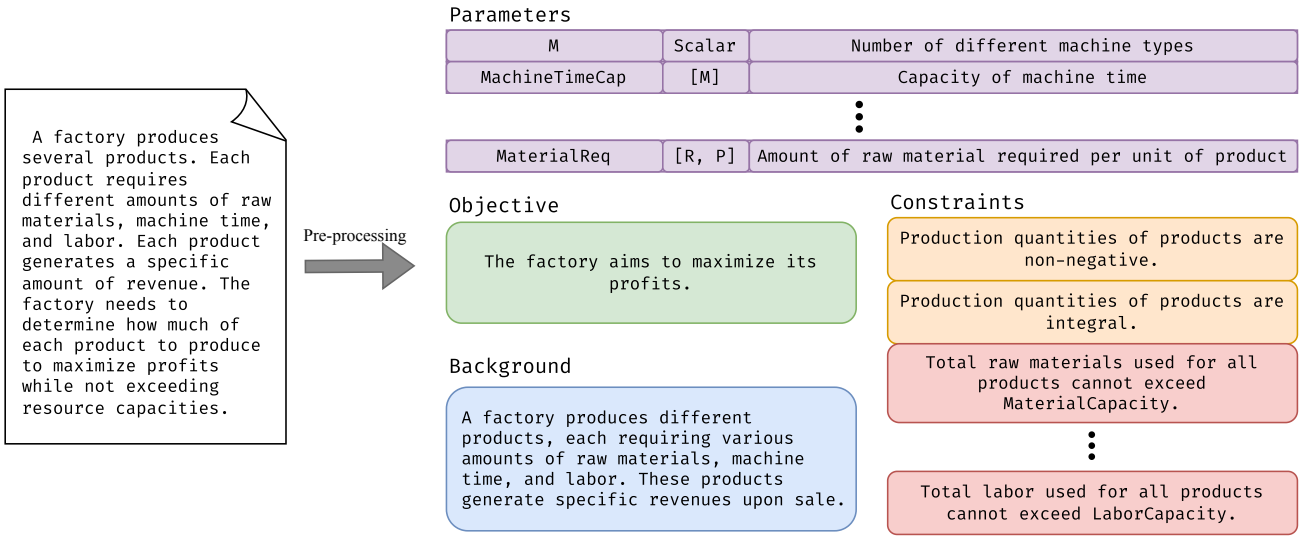

Figure 2. OptiMUS preprocesses natural language representations of a problem into a modular state. The components of the modular state are: **1)** parameters and their shape, **2)** objective, **3)** background and context, and **4)** implicit and explicit constraints.

these clauses will be augmented with LaTeX formulations and code as well.

- **Background**: A short string explaining the real-world context of the problem. This string is included in every prompt to improve common sense reasoning.

The preprocessing uses three prompts: the first prompt extracts the parameters, the second segments the problem into objective and constraints, and the third eliminates redundant (e.g., two restatements of the constraint that production quantity is nonnegative), unnecessary (such as facts about the problem parameters, e.g., that price is nonnegative), and incorrect constraints (e.g., production quantity must exactly equal demand). The second step can also be a challenge: for example, in the factory example shown in Figure 2, the production amount for each product should be a positive value, but this is not stated explicitly.

3.2. Agents

After preprocessing, OptiMUS defines problem variables, formulates and codes each clause. To ensure consistency of the formulations, OptiMUS constructs and maintains a connection graph to record which variables and parameters appear in each constraint. This connection graph is key to performance and scalability of OptiMUS, as it allows the LLM to focus only on the relevant context for each prompt, generating more stable results. The list of variables and the LaTeX formulations and code are initially empty; when all clauses are formulated, programmed, and validated, the process is complete.

Algorithm 1 Workflow of OptiMUS

Input: Natural language description of problem $\mathcal{P}$
$P^{(0)} \leftarrow \text{PREPROCESS}(\mathcal{P})$
Initialize msg $\leftarrow$ ""
Initialize conversation $\leftarrow$ []
for $t = 1, \ldots$ **do**
  AGENT, task $\leftarrow \text{MANAGER}(\text{conversation})$
  $P^{(t+1)}$, msg $\leftarrow \text{AGENT}(P^{(t)}, \text{task})$
  conversation += msg
  if msg = Done **then** break
end

Manager. Inspired by (Wu et al., 2023), OptiMUS uses a manager agent to coordinate the work of formulation, programming, and evaluation, recognizing that these steps

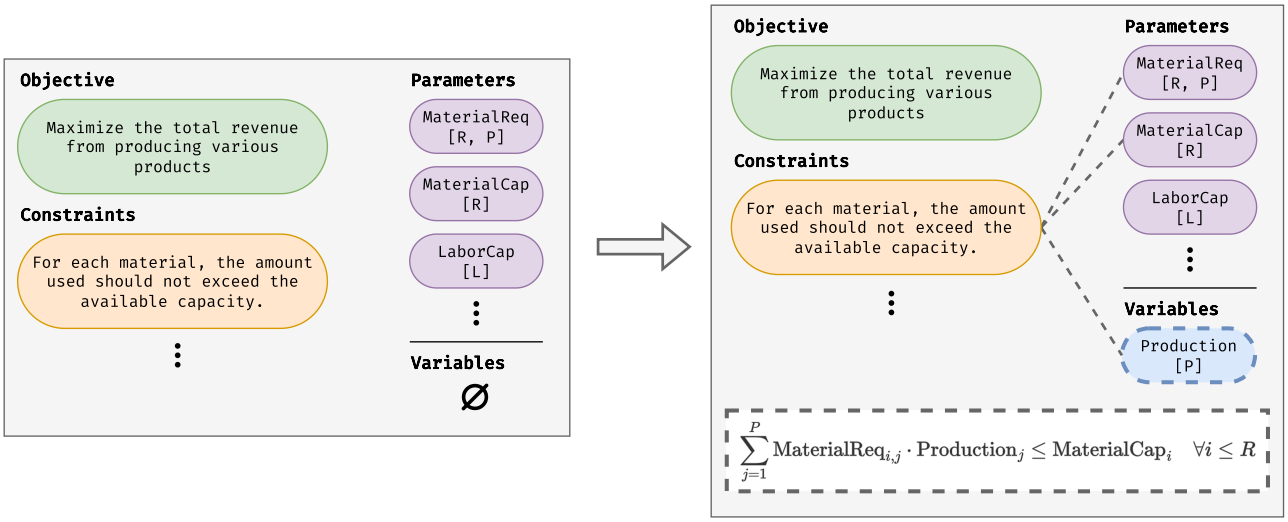

Figure 3. The formulation process for a single constraint. The formulation agent identifies any parameters and variables appearing in the constraint, including new variables that it may need to define. It defines new variables as needed, updates the connection graph which records which constraints use which parameters and which variables, and annotates the constraint with a LaTeX formulation. (dashed lines represent new connections and variables)

may need to be repeated to ensure consistency and correctness (see Algorithm 1). At each step, the manager looks at the conversation so far and chooses the next agent (formulator, programmer, or evaluator) to process the problem. The manager also generates and assigns a *task* to the chosen agent, for example:

Review and fix the formulation of the objective.

Formulator. The formulator agent must:

1. Write and correct mathematical formulations for variables and clauses.
2. Define new variables and auxiliary constraints.
3. Update the links in the connection graph.

If the assigned task is to formulate new clauses, the formulator iterates over the clauses that have not yet been formulated and generates new formulations for them. During this process, it will also define auxiliary constraints and new variables when necessary. Moreover, it decides which parameters and variables are related to the clause (see Figure 3). This information is used to update the connection graph. On the other hand, if the task is to fix incorrect formulations reported by the evaluator or the programmer, the agent iterates through the clauses marked as incorrect, fixes their formulations, and updates the connection graph. OptiMUS also has an extra modeling layer that captures special model structures (e.g., special-ordered-set and indicator variables) and we leave a more detailed discussion to the Appendix B.

Programmer. The responsibility of the programmer agent is to write and debug the solver code. When the programmer is called by the manager, it first reads the task. If the task is to program new clauses, the agent iterates over the clauses that have not yet been coded and generates code from their formulations. If the task is to fix incorrect formulations reported by the evaluator, the agent iterates through the clauses marked as bogus and fixes their codes.

In our experiments, the programmer uses Python as the programming language and `Gurobi` as the solver. OptiMUS can target other solvers and programming languages as long as they are supported by the LLM.

Evaluator. The evaluator agent's responsibility is to execute the generated code on the data and to identify any errors that occur during the execution. If evaluator faces a runtime error, it flags the variable or clause with the bogus code and responds to the manager with appropriate explanation of the error. The information will later be used by the other agents to fix the formulation and debug the code.

3.3. The connection graph

Recall from Section 3.2 that OptiMUS maintains a connection graph over constraints, objectives, parameters, and variables. OptiMUS uses this graph to retrieve the relevant context for each prompt so prompts remain short. This graph is used also to generate and debug code and to correct wrong formulations. Figure 4 provides an example.

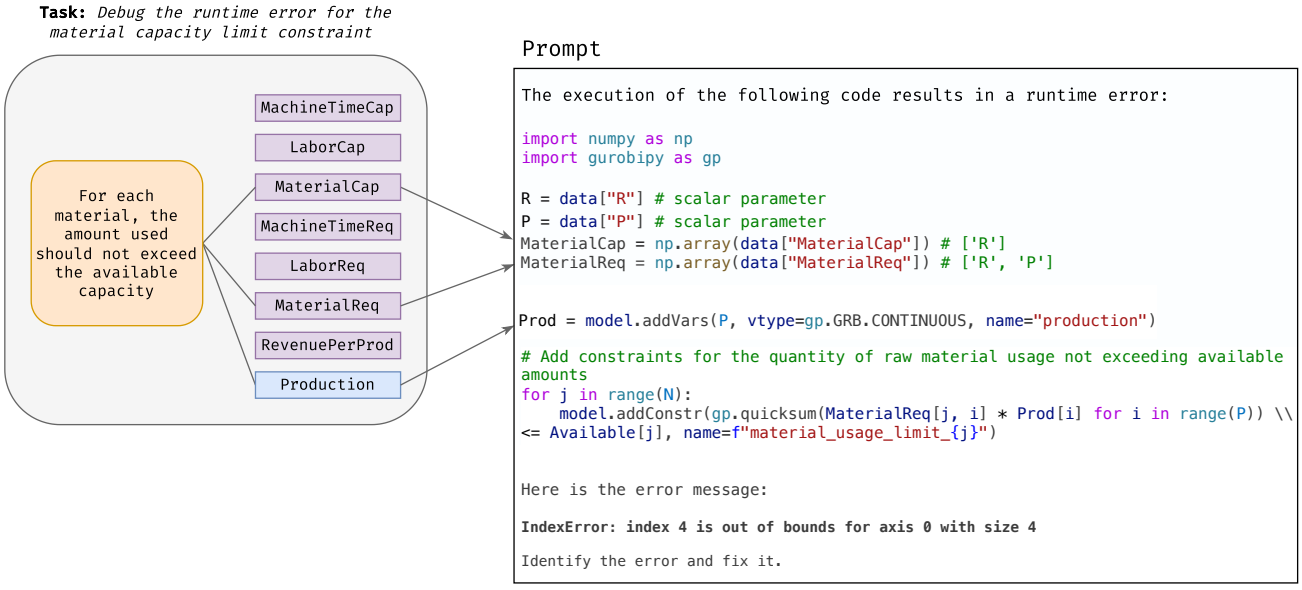

Figure 4. OptiMUS uses the connection graph to extract and use only the relevant context in each prompt. In this example, the programmer agent fetches the context via the connection graph to debug a bogus constraint code. Without the graph, the LLM would have needed to process the whole code, including the code for the other parameters, variables, constraints, and the objective.

4. Experiments

In this section, we conduct a comprehensive evaluation of OptiMUS. We begin by detailing the datasets used in our experiments and showcase the superior performance of OptiMUS across these datasets, highlighting its strengths. An ablation study demonstrates the impact of different system components on our results, and a sensitivity analysis probes the internal dynamics of OptiMUS. We conclude this section by identifying failure cases and potential areas for further improvement.

4.1. Dataset

NL4OPT. This dataset is a collection of 1101 easy linear programming problems proposed as part of the NL4OPT competition (Ramamonjison et al, 2023). The dataset contains a natural language description of each problem, along with an annotated *intermediate representation* that lists parameters, variables, and clauses.

ComplexOR. ComplexOR is a collection of 37 complex operations research problems in a variety of application domains (Xiao et al., 2024). At the time of writing this paper, the publicly available version of this dataset is incomplete. We gathered 21 problems from the ComplexOR dataset to use in our experiments by

NLP4LP. As shown in Table 1, existing datasets for natural language optimization modeling lack problems with long descriptions. Real-world problems often are much longer, see e.g. (Holzer et al., 2023). To address this issue, we create NLP4LP (Natural Language Processing for Linear Programming), a benchmark consisting of 54 LP and 13 MILP problems (67 instances in total). NLP4LP problems are drawn from textbooks and lecture notes on optimization (Bertsimas & Tsitsiklis, 1997b; Williams, 2013; Nace, 2020), including facility location, network flow, scheduling, portfolio management, and energy optimization problems. These resources were created before 2021, so it is possible parts of these books have been used to train LLMs. However, none of these textbooks includes code. Moreover, our results show that LLMs still find it challenging to formulate and solve these problems. For each instance, NLP4LP includes the description, a sample parameter data file, and the optimal value, obtained either from the textbook solution manual or by solving the instance by hand. Together, NLP4LP and ComplexOR offer a variety of challenging optimization problems with different lengths, facilitating the research on automated optimization modeling.

4.2. Overall Performance

To evaluate the overall performance of OptiMUS, we compare it with standard prompting, Reflexion, and Chain-of -Experts (CoE) (Shinn et al., 2023; Xiao et al., 2024) (More details in D). Reflexion is the highest-performing general-purpose framework and CoE is the state-of-the-art method for natural-language optimization modeling. Three main metrics have been used in the literature: accuracy, compilation error (CE) rate, and runtime error (RE) rate. However, a method can generate a totally irrelevant short code

that runs, or fix runtime and complication errors by completely removing relevant sections of the code. Hence, we only compare the models' accuracy. Accuracy is defined as the number of instances correctly solved (An instance is considered as correctly solved only if the code runs successfully and the optimal value is correct. Optimal values are obtained from the dataset or by solving the problems manually). Results are presented in Table 2. OptiMUS outperforms all other methods in all datasets by a large margin. This remarkable performance improvement highlights the importance of modularity and structure compared to a single prompt to solve complex problems using LLMs.

The next experiments clarify which features of OptiMUS contribute to its good performance.

4.3. Ablation Study

Table 3 shows the impact of debugging and of the choice of LLM on the performance of OptiMUS. One interesting observation is the significant performance drop that occurs when smaller LLMs are used instead of GPT-4. The first reason is that the OptiMUS prompts are on average longer than the other methods and involve more complicated reasoning. Smaller LLMs are worse at reasoning (Wang et al., 2023; OpenAI, 2023). The second reason is the novel and modular structure of OptiMUS's prompts. Prompts used in the other methods mostly adhere to a questions answering format that is abundant in the public domain (e.g. posting the whole bogus code snippet and asking for the correct version is common on StackOverflow, or writing the whole problem description and then the complete formulation is common in optimization textbooks). However, in OptiMUS, the prompts are more complex and not common in human-human interactions. Smaller LLMs have limited generalization and reasoning abilities and, therefore, show poor performance on such prompts (OpenAI, 2023). Fine-tuning smaller models on these novel prompt templates might improve their performance and reduce the cost of running a system like OptiMUS.

We also evaluated a version of OptiMUS which uses GPT-3.5 for the manager and GPT-4 for the other agents. We can see that in NL4OPT the difference in performance is small. The reason is that most instances of NL4OPT are solved with a simple chain of formulation-programming-evaluation. However, in ComplexOR and NLP4LP where more complicated interactions between agents are required, the manager's importance becomes more visible. Moreover, we did experiments in which the debugging feature of the programmer agent was disabled. Similarly to the manager, we see that debugging is more important in more complicated datasets.

Table 2. Performance of OptiMUS and the baselines using GPT4.

	NL4OPT	ComplexOR	NLP4LP
Standard	47.3%	9.5%	35.8%
Reflexion	53%	19.1%	46.3%
CoE	64.2%	38.1%	53.1%
OptiMUS (Ours)	**78.8%**	**66.7%**	**72.0%**

Table 3. Ablation study of OptiMUS.

	NL4OPT	ComplexOR	NLP4LP
OptiMUS (GPT-4)	78.7%	66.7%	71.6 %
w/o debugging	72.3%	57.1%	58.2%
w/ GPT-3.5 Mngr	74.9%	52.4%	53.7%
w/ GPT-3.5	28.6%	9.5%	14.4%
w/ Mixtral-8x7B	6.6%	0.0%	3.0%

4.4. Sensitivity Analysis

Figure 5 shows how the maximum number of times the manager is allowed to select agents affects the accuracy. For NL4OPT, most problems are solved by selecting each of the formulator, programmer, and evaluator agents only once. However, for ComplexOR and NLP4LP, OptiMUS often makes mistakes at the beginning and iteratively fixes them by selecting the other agents multiple times.

Section 4.5 shows the number of times each agent is selected per instance. As expected, the average selection frequency is higher in ComplexOR and NLP4LP. Moreover, programmer and evaluator agents are selected more often than the formulator. This bias is reasonable:

- Coding errors are more common. LLMs often generate code with trivial bugs that are easy to fix. In OptiMUS, the programmer agent fixes such bugs.
- Coding errors are easier to identify and fix. In contrast, identifying bugs in the formulation require deeper reasoning and is harder. Hence the manager in OptiMUS is prompted to prioritize fixing the code before considering errors in the formulation. The formulator is only selected for debugging if the programmer claims that the code is correct.

Hence in our experiments, we observe the programmer is selected more often than the formulator.

Table 4 shows the average prompt length of OptiMUS and CoE for different data sets. Observe that the prompt length for OptiMUS barely changes across datasets, while the prompt length for CoE increases on more challenging datasets. The reason is the modular approach, which allows OptiMUS to extract and process only the relevant context for each LLM call. Unlike non-modular methods, Opti-

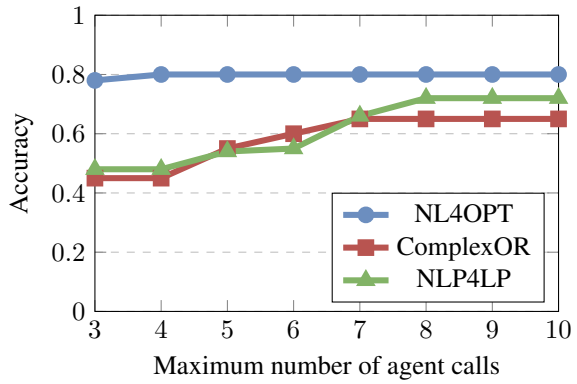

Figure 5. OptiMUS can solve more problems on difficult datasets (ComplexOR, NLP4OPT) when more agent calls are allowed, demonstrating the importance of self-improvement.

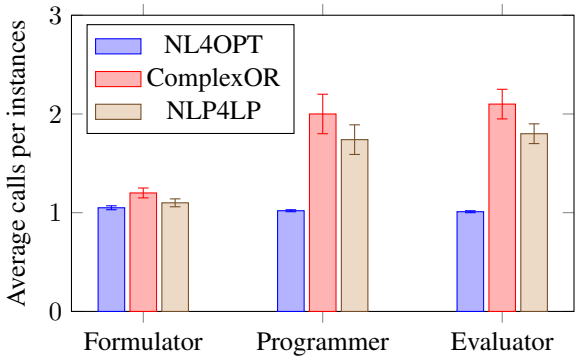

Figure 6. Average number of calls to each agent among solved problems. OptiMUS only requires one call per agent on the simple problems of NL4OPT. On the more complex datasets, it relies more heavily on the programmer to fix errors identified by the evaluator, but rarely improves by fixing formulation errors.

MUS can scale to larger and longer problems.

4.5. Failure Cases

To understand its strengths and weaknesses, we analyze the most common reasons why OptiMUS fails (Table 6). We categorize failure cases into the following groups:

- Missing or wrong constraints: OptiMUS generates wrong constraint in the preprocessing step (e.g., price ≥ 0 where price is a parameter), or fails to extract all of the constraints from the description.
- Incorrect model: OptiMUS tackles the problem with an incorrect mathematical model (e.g., defining binary variables for visiting cities instead of links in TSP).
- Coding error: OptiMUS does not generate error-free code even after debugging. Coding errors often occur when the LLM is confused by the language used (e.g., in the "prod" problem in ComplexOR, the description explicitly refers to "parameters" and "variables").

We normalize the failure rates to sum to 1.0. Incorrect modeling is more common on datasets with more complicated problems, while on the easier dataset NLP4OPT, the model is less likely to be wrong.

Understanding and interpreting the problems is also challenging for LLMs, resulting in formulations with missing constraints and wrong constraints. Fine-tuning might improve the performance of LLMs on this task, and is an important direction for future research.

5. Conclusion

How can LLMs collaborate and divide work in order to achieve complex goals? This paper interrogates this question in the domain of optimization and showcases the importance of modular structure. We develop OptiMUS, a modular LLM-based agent designed to formulate and solve optimization problems from natural language descriptions. Our research serves as a proof-of-concept, illustrating the potential for automating various stages of the optimization process by combining LLMs with traditional solvers. To showcase the performance of OptiMUS, we released NLP4LP, a dataset of long and challenging optimization problems to demonstrate the efficacy of the techniques implemented within OptiMUS. OptiMUS achieves SOTA performance across all existing datasets, and scales to problems with large amounts of data and long descriptions.

Real-world optimization problems are often complex and multifaceted. Developing LLM-based solutions for these problems requires domain-specific considerations, including integrating existing optimization techniques to leverage problem structure. We are at the early stages of this research, but anticipate significant developments that will enable these systems to address more complex, industrial-level problems. It is interesting to notice that the challenge of using AI for an applied domain is much larger in safety-critical domains such as self-driving, which demand extremely high accuracy, than in domains where AI can function as an assistant and where answers are easy to check, as in theorem-proving or optimization. Here, AI systems with moderate accuracy can still usefully augment human effort.

Future directions. Smaller LLMs are faster and cheaper, but our experiments indicate that they perform poorly in optimization modeling out-of-the-box. Identifying which prompts might benefit from fine-tuned small models and which require large (and expensive) LLM calls is an important topic for future research. Furthermore, we believe that integrating user feedback into the process can improve the performance of agents on natural-language optimiza-

Table 4. CoE requires longer prompts on difficult datasets, while OptiMUS barely increases its prompt length.

	NL4OPT	ComplexOR	NLP4LP
CoE	2003 ± 456	3288 ± 780	3825 ± 1002
OptiMUS	2838 ± 822	3241 ± 1194	3146 ± 1145

Table 5. When OptiMUS fails, why?

Mistake	NL4OPT	ComplexOR	NLP4LP
Incorrect modeling	43.0%	62.5%	53.8%
Missing constraints	36.0%	12.6%	15.4%
Coding errors	21.0%	24.9%	30.8%

tion modeling. Studying interactions between such agents and their users is an exciting avenue. Another important direction is to automatically select the best solver based on a comprehensive evaluation of both accuracy and runtime requirements. Finding the optimal formulation based on user preferences, runtime, problem size, etc. is also another interesting area to explore. Additionally, it would be interesting to see how the modular LLM structure presented here can be enhanced using reinforcement learning to teach the manager how to choose the next agent.

Table 6. Runnable and solved instances

State	NL4OPT	ComplexOR	NLP4LP
Runnable	85.6%	76.2 %	75.0%
Solved	78.8%	66.7 %	72.0%

Acknowledgements

The authors gratefully acknowledge support from the Office of Naval Research (ONR) Award N000142212825 and the Alfred P. Sloan Foundation. We also appreciate the constructive feedback from the reviewers and the area chair.

Impact Statement

This paper presents work whose goal is to advance the field of optimization modeling. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Aastrup, J. and Kotzab, H. Forty years of out-of-stock research–and shelves are still empty. *The International Review of Retail, Distribution and Consumer Research*, 20(1):147–164, 2010.

Akgun, O., Miguel, I., Jefferson, C., Frisch, A., and Hnich, B. Extensible automated constraint modelling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, pp. 4–11, 2011.

Alibaba Cloud. Alibaba cloud mindopt copilot, 2022. URL https://opt.alibabacloud.com/chat.

Antoniou, A. and Lu, W.-S. *Practical optimization: algorithms and engineering applications*, volume 19. Springer, 2007.

Beale, E. and Forrest, J. J. Global optimization using special ordered sets. *Mathematical Programming*, 10:52–69, 1976.

Beldiceanu, N. and Simonis, H. A model seeker: Extracting global constraint models from positive examples. In *International Conference on Principles and Practice of Constraint Programming*, pp. 141–157. Springer, 2012.

Beldiceanu, N. and Simonis, H. Modelseeker: Extracting global constraint models from positive examples. *Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach*, pp. 77–95, 2016.

Bertsimas, D. and Tsitsiklis, J. N. *Introduction to linear optimization*, volume 6. Athena scientific Belmont, MA, 1997a.

Bertsimas, D. and Tsitsiklis, J. N. *Introduction to linear optimization*, volume 6. Athena scientific Belmont, MA, 1997b.

Bessiere, C., Koriche, F., Lazaar, N., and O'Sullivan, B. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., Van Den Driessche, G. B., Lespiau, J.-B., Damoc, B., Clark, A., et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pp. 2206–2240. PMLR, 2022.

Chen, H., Constante-Flores, G. E., and Li, C. Diagnosing infeasible optimization problems using large language models. *arXiv preprint arXiv:2308.12923*, 2023.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim,

H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways, 2022.

De Raedt, L., Passerini, A., and Teso, S. Learning constraints from examples. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Gamrath, G., Berthold, T., Heinz, S., and Winkler, M. *Structure-based primal heuristics for mixed integer programming*. Springer, 2016.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023.

Gurobi Optimization. 2023 state of mathematical optimization report, 2023. URL https://www.gurobi.com/resources/report-state-of-mathematical-optimization-2023/.

Holzer, J., Coffrin, C., DeMarco, C., Duthu, R., Elbert, S., Eldridge, B., Elgindy, T., Greene, S., Guo, N., Hale, E., Lesieutre, B., Mak, T., McMillan, C., Mittelmann, H., Oh, H., O'Neill, R., Overbye, T., Palmintier, B., Safdarian, F., Tbaileh, A., Hentenryck, P. V., Veeramany, A., and Wert, J. Grid optimization competition challenge 3 problem formulation. https://gocompetition.energy.gov/sites/default/files/Challenge3_Problem_Formulation_20230126.pdf, 2023. Accessed: Access Date.

Kiziltan, Z., Lippi, M., Torroni, P., et al. Constraint detection in natural language problem descriptions. In *IJCAI*, volume 2016, pp. 744–750. International Joint Conferences on Artificial Intelligence, 2016.

Li, B., Mellou, K., Zhang, B., Pathuri, J., and Menache, I. Large language models for supply chain optimization. *arXiv preprint arXiv:2307.03875*, 2023.

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. Lost in the middle: How language models use long contexts, 2023.

Nace, D. Lecture notes in linear programming modeling, 2020. URL https://www.hds.utc.fr/~dnace/dokuwiki/_media/fr/lp-modelling_upt_p2021.pdf.

OpenAI. Gpt-4 technical report, 2023.

Paranjape, B., Lundberg, S., Singh, S., Hajishirzi, H., Zettlemoyer, L., and Ribeiro, M. T. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

Ramamonjison et al, . Augmenting operations research with auto-formulation of optimization models from problem descriptions. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 29–62, Abu Dhabi, UAE, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-industry.4.

Ramamonjison et al, . Nl4opt competition: Formulating optimization problems based on their natural language descriptions, 2023. URL https://arxiv.org/abs/2303.08233.

Saghafian, S., Austin, G., and Traub, S. J. Operations research/management contributions to emergency department patient flow optimization: Review and research prospects. *IIE Transactions on Healthcare Systems Engineering*, 5(2):101–123, 2015.

Shakoor, R., Hassan, M. Y., Raheem, A., and Wu, Y.-K. Wake effect modeling: A review of wind farm layout optimization using jensen' s model. *Renewable and Sustainable Energy Reviews*, 58:1048–1059, 2016.

Shinn, N., Cassano, F., Berman, E., Gopinath, A., k Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning, 2023.

Singh, A. An overview of the optimization modelling applications. *Journal of Hydrology*, 466:167–182, 2012.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Wang, Y., Ivison, H., Dasigi, P., Hessel, J., Khot, T., Chandu, K. R., Wadden, D., MacMillan, K., Smith, N. A., Beltagy, I., et al. How far can camels go? exploring the state of instruction tuning on open resources. *arXiv preprint arXiv:2306.04751*, 2023.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models, 2023.

Williams, H. P. *Model building in mathematical programming*. John Wiley & Sons, 2013.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., and Wang, C. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

Xiao, Z., Zhang, D., Wu, Y., Xu, L., Wang, Y. J., Han, X., Fu, X., Zhong, T., Zeng, J., Song, M., and Chen, G. Chain-of-experts: When LLMs meet complex operations research problems. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=HobyL1B9CZ.

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers, 2023.

Yao, E., Liu, T., Lu, T., and Yang, Y. Optimization of electric vehicle scheduling with multiple vehicle types in public transport. *Sustainable Cities and Society*, 52: 101862, 2020.

A. System Design and Software Engineering

In this section, we include more details on software implementation of the pipeline.

A.1. Preprocessing

The preprocessor in OptiMUS extracts data parameters and constraints from the natural language problem description. Preprocessor is driven by six prompts that extract a) problem type b) problem background c) parameters.

Here is an example of the prompts we used to identify the problem type:

```
You are an expert mathematical modeler and an optimization professor at a
top university. Here is the description of an optimization problem:

-----
{{description}}
-----

Your task is to identify the type of optimization problem this is, and put
it in a json file in this format:

-----
{{"type": "type of optimization problem"}}
-----

You should replace "type of optimization problem" with the type of
optimization problem you think this is. Options are: 1. linear programming
2. mixed-integer programming 3. others

You will be awarded one million dollars if you get it right
```

A.2. Debugging

We maintain a for loop and a global code snippet variable, and every time we concatenate a new block of Python code (for a constraint) from programmer to the code snippet. Then we execute the code using Python's subprocesses library. Once there is an error, we know that the error is thrown by the most recent code block (constraint) and mark the constraint as "erroneous" in the state. Then the state is passed back to the programmer/formulator, where the LLM is given the error message and the context of that constraint, and is asked to debug the constraint accordingly.

A.3. Connection Graph

In OptiMUS, a connection graph is maintained so that when we need to work on a particular constraint(context), agents focus on only the relevant variables and parameters. This is important when the model gets large and there are many variables/parameters.

The connection graph in OptiMUS is tripartite with three layers: 1) Parameters 2) Constraints, and 3) Variables. When creating a new constraint, OptiMUS first adds new variable nodes if we need to define new variable. Then OptiMUS asks the formulator to also identify the set of relevant variables and parameters for that constraint. Then, an edge is added between this new constraint and the relevant variables/parameters. When OptiMUS needs to retrieve a context (for example when manager decides to debug a constraint), we identify only the relevant nodes in the connection graph and format them into a prompt. Here is an example:

```
for parameter in state["parameters"]:
    if parameter["symbol"] in constraint_context["related_parameters"]:
        prep_code += parameter["code"] + "\n"

for variable in state["variables"]:
    if variable["symbol"] in constraint_context["related_variables"]:
        prep_code += variable["code"] + "\n"

prompt = debugging_refined_template_target.format(
                                        constraint=constraint,
                                        prep_code=prep_code,
                                        error_line=error_line,
                                        error_message=error_message)
```

B. Optimization Techniques

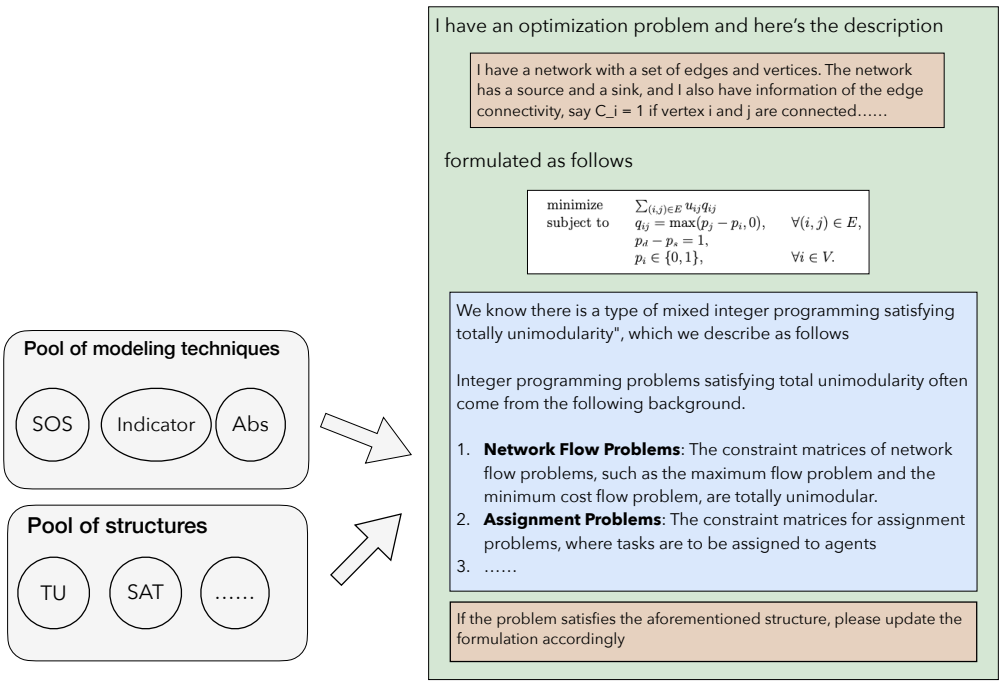

Figure 7. OptiMUS iterates through a pool of advanced optimization techniques

Optimization solvers exploit problem-specific structure to improve performance when solving MILPs (Gamrath et al., 2016) and often provide a customized interface for these special structures. Using the interface not only reduces the complexity of (and potential for errors in) auxiliary variables or constraints, but also informs the solver about the existence of structure that can be exploited to solve the problem faster. Moreover, the solver's performance can suffer when these structures are not signaled in the model. For example, a bad choice of big-M coefficient when reformulating an indicator variable can reduce the strength of the linear relaxation. Typical examples of structure include Special Ordered Set (SOS) (Beale & Forrest, 1976), indicator variables, and general constraints (Bertsimas & Tsitsiklis, 1997a).

Although state-of-the-art optimization solvers can detect some problem structures automatically, it works better to specify structure during problem formulation. Hence the formulator is prompted to leverage advanced optimization techniques and structures, including **1)** Special Ordered Set. **2)** Indicator variable. **3)** General constraints. **4)** SAT and constraint programming problem. **5)** Totally unimodular problem detection.

OptiMUS iterates through a sequence of "cheatsheet" prompts (Figure 7), each corresponding to one of these structures.

Within each prompt, the LLM is provided with the description of the structure, explained by an example illustrating how the structure should be exploited. The LLM is asked to decide whether the structure can be applied to the existing formulation. Upon identifying the appropriate structure, the formulation is adjusted to utilize the customized solver interface when available.

C. Limitations and Weaknesses

C.1. Sensitivity of LLMs to and parameter variable names

Based on the training data, LLMs can be biased towards interpreted certain terms in certain ways that are not necessarily correct. Consider the following optimization problems:

A global fashion brand sells articles of clothing in several markets. We have an estimate of how price changes affect sales for each article, assuming constant price elasticity. Given the sales forecast per article for the next twelve months and past elasticities, the goal is to choose new prices for each article to maximize expected revenue.

A manufacturing company produces different items using different raw materials. We have some amount of each raw material available, and we know how much of each raw material is needed to produce 1 unit of each item. The price for each item in the market is known. The goal is to choose how many of each item to produce to maximize the revenue.

In the first problem, prices should be defined as variables, while in the second problem prices are parameters. If the first problem appears in an LLM's training corpus, it will be biased to identify price as a variable in the second problem at inference time. We did not perform large-scale experiments to evaluate the performance of LLMs on such cases, particularly because of lack of access to a relevant dataset. However, our experiments indicate that even small models such as LLama3-8b can identify the differences and avoid confusion if prompted correctly.

C.2. Hidden Errors

A common issues with LLMs is high confidence hallucination. In the context of optimization modeling, this can be problematic when in the modeling and coding steps, especially if the mistakes do not result in explicit errors (for instance redundant incorrect constraints, or missing constraints). This is because of the nature of the task we are tackling, and can happen even if the problem is solved by human experts. We believe that this such systems can be used as products if the error rate goes below a certain threshold. To reduce the error rate, we can adopt methods that experts commonly use. This includes 1) automatically generating dummy data with feasible solutions and evaluating the model on it, 2) generating plots for the solution and looking at the trends (using VLMs), and 3) adding reasoning steps to double-check the correctness of the initial LLM outputs. Moreover, we also believe that involving humans in the loop at the right time can be really helpful. We have observed promising results in all of these areas, but discussing them in detail is out of the scope of this paper.

D. Applying Existing Methods to Our Dataset

In standard prompting the problem description and input data are passed into the LLM and it is asked to write solver code. The code is then executed once to get the output. In Reflexion, the output of the LLM is updated multiple times based on a feedback mechanism. The feedback is obtained from the compilation and runtime errors, allowing for iterative refinement of the previous steps until the agent is satisfied with the answer. Chain of experts uses a system of agents with a manager to solve the problem.

E. Prompts

E.1. Manager Prompt

```
You're a manager in a team of optimization experts. The goal of the team is to solve an optimization
problem. Your task is to choose the next expert to work on the problem based on the current situation.
- The user has already given us the problem description, the objective function, and the parameters. Only
call the user proxy if there is a problem or something ambiguous or missing.

Here's the list of agents in your team:
{agents}

And here's the history of the conversation so far:
{history}

Considering the history, if you think the problem is solved, type DONE. Otherwise, generate a json file
with the following format:
{{
   "agent_name": "Name of the agent you want to call next",
   "task": "The task you want the agent to carry out"
}}

to identify the next agent to work on the problem, and also the task it has to carry out.
- If there is a runtime error, ask the prorammer agent to fix it.
- Only generate the json file, and don't generate any other text.
- If the latest message in history says that the code is fixed, ask the evaluator agent to evaluate the
code!
```

E.2. Formulation Generation Prompt

```
You are an expert mathematical formulator and an optimization professor at a top university. Your task is to model
{clausType} of the problem in the standard LP or MILP form.

Here is a {clausType} we need you to model:
{targetDescription}

Here is some context on the problem:
{background}

Here is the list of available variables:
{variables}

And finally, here is list of input parameters:
{parameters}

First, take a deep breath and explain how we should define the {clausType}. Feel free to define new variables if you
think it is necessary. Then, generate a json file accordingly with the following format (STICK TO THIS FORMAT!):

{{
  "{clausType}": {{
    "description": "The description of the {clausType}",
    "formulation": "The LaTeX mathematical expression representing the formulation of the {clausType}"
  }},
  "auxiliary_constraints": [
      {{
          "description": "The description of the auxiliary constraint",
          "formulation": "The LaTeX mathematical expression representing the formulation of the auxiliary constraint"
      }}
  ]
  "new_variables": [
      {{
          "definition": "The definition of the variable",
          "symbol": "The symbol for the variable",
          "shape": [ "symbol1", "symbol2", ... ]
      }}
  ],
 }}

- Your formulation should be in LaTeX mathematical format (do not include the $ symbols).
- Note that I'm going to use python json.loads() function to parse the json file, so please make sure the format is
correct (don't add ',' before enclosing '}}' or ']' characters.
- Generate the complete json file and don't omit anything.
- Use '```json' and '```' to enclose the json file.
- Important: You can not define new parameters. You can only define new variables.Use CamelCase and full words for
new variable symbols, and do not include indices in the symbol (e.g. ItemsSold instead of itemsSold or items_sold or
ItemsSold_i)
- Use \\textup{{}} when writing variable and parameter names. For example (\\sum_{{i=1}}^{{N}}
\\textup{{ItemsSold}}_{{i}} instead of \\sum_{{i=1}}^{{N}} ItemsSold_{{i}})
- Use \\quad for spaces.
- Use empty list ([]) if no new variables are defined.
- Always use non-strict inequalities (e.g. \\leq instead of <), even if the constraint is strict.
- Define auxiliary constraints when necessary. Set it to an empty list ([]) if no auxiliary constraints are needed.
If new auxiliary constraints need new variables, add them to the "new_variables" list too.


Take a deep breath and solve the problem step by step.
```

E.3. Formulation Fixing Prompt

```
You are a mathematical formulator working with a team of optimization experts. The objective is to tackle a complex
optimization problem, and your role is to fix a previously modelled {target}.

Recall that the {target} you modelled was
{constraint}

and your formulation you provided was
{formulation}

The error message is
{error}

Here are the variables you have so far defined:
{variables}

Here are the parameters of the problem
{parameters}

Your task is carefully inspect the old {target} and fix it when you find it actually wrong.
After fixing it modify the formulation. Please return the fixed JSON string for the formulation.

The current JSON is
{json}

Take a deep breath and solve the problem step by step.
```

E.4. Clause Coding Prompt

```
You're an expert programmer in a team of optimization experts. The goal of the team is to solve an
optimization problem. Your responsibility is to write {solver} code for different {target}s of the problem.
Here's a {target} we need you to write the code for, along with the list of related variables and
parameters:

{context}

- Assume the parameters and variables are defined, and gurobipy is imported as gp. Now generate a code
accordingly and enclose it between "=====" lines.
- Only generate the code and the ===== lines, and don't generate any other text.
- If the {target} requires changing a variable's integrality, generate the code for changing the variable's
integrality rather than defining the variable again.
- If there is no code needed, just generate the comment line (using # ) enclosed in ===== lines explaining
why.
- Variables should become before parameters when defining inequality {target}s in gurobipy (because of the
gurobi parsing order syntax)

Here's an example:
**input**:
{{
   "description": "in month m, it is possible to store up to storageSize_{{m}} tons of each raw oil for use
later.",
   "formulation": "\(storage_{{i,m}} \leq storageSize, \quad \\forall i, m\)",
   "related_variables": [{{
           "symbol": "storage_{{i,m}}",
           "definition": "quantity of oil i stored in month m",
           "shape": [
               "I",
               "M"
           ]
       }}],
   "related_parameters": [{{
           "symbol": "storageSize_{{m}}",
           "definition": "storage size available in month m",
           "shape": [
               "M"
           ]
       }}]
}}

***output***:
=====
# Add storage capacity constraints
for i in range(I):
   for m in range(M):
       model.addConstr(storage[i, m] <= storageSize[m], name="storage_capacity")
=====

Take a deep breath and approach this task methodically, step by step.
```

E.5. Variable Coding Prompt

```
You're an expert programmer in a team of optimization experts. The goal of the team is to solve an
optimization problem. Your responsibility is to write {solver} code for defining variables of the problem.
Here's a variable we need you to write the code for defining:
{variable}

Assume the parameters are defined. Now generate a code accordingly and enclose it between "=====" lines.
Only generate the code, and don't generate any other text. Here's an example:

**input**:

{{
   "definition": "Quantity of oil i bought in month m",
   "symbol": "buy_{{i,m}}",
   "shape": ["I","M"]
}}

***output***:

=====
buy = model.addVars(I, M, vtype=gp.GRB.CONTINUOUS, name="buy")
=====

- Note that the indices in the symbol (what comes after _) are not a part of the variable name in code.
- Use model.addVar instead of model.addVars if the variable is a scalar.


Take a deep breath and solve the problem.
```

E.6. Debugging Prompt

```
You're an expert programmer in a team of optimization experts. The goal of the team is to solve an
optimization problem. Your responsibility is to debug the code for the problem.

When running the following code snippet, an error happened:
{context_code}
{error_line}

and here is the error message:
{error_message}

We know that the code for importing packages and defining parameters and variables is correct, and the
error is because of the this last part which is for modeling the {target}:
{error_line}

First reason about the source of the error. Then, if the code is correct and the problem is likely to be in
the formulation, generate a json in this format (the reason is why you think the problem is in the
formulation):

{{
   "status": "correct",
   "reason": "A string explaining why you think the problem is in the formulation"
}}

otherwise, fix the last part code and generate a json file with the following format:

{{
   "status": "fixed",
   "fixed_code": "A sting representing the fixed {target} modeling code to be replaced with the last part
code"
}}

- Note that the fixed code should be the fixed version of the last part code, not the whole code snippet.
Only fix the part that is for modeling the {target}.
- Do not generate any text after the json file.
- Variables should become before parameters when defining inequality constraints in gurobipy (because of
the gurobi parsing order syntax)
- The parameter shapes are parameters definitions are correct.

Take a deep breath and solve the problem step by step.
```