Proceedings of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 165–174
DOI: 10.15439/2025F5420
ISSN 2300-5963 ACSIS, Vol. 43

Toward Conversational Decision Support Systems: Integrating LLMs in the Operations Research Methodology

Mariusz Kaleta
0000-0002-2225-8956
Warsaw University of Technology
ul. Nowowiejska 15/19, 00-665 Warsaw, Poland
Email: mariusz.kaleta@pw.edu.pl

***Abstract*—This paper introduces the concept of Conversational Decision Support Systems (C-DSS)—a novel, agent-based framework that leverages Large Language Models (LLMs) to enhance the Operations Research (OR) methodology. We focus on the modeling and coding stages of decision support systems, where language-based interaction is crucial. The paper evaluates the effectiveness of LLMs in generating mathematical models and AMPL code for a curated set of 20 LP/MILP artifacts. Four architectural setups are analyzed: a monolithic LLM agent (M/C), its enhancement with a code verifier (M/C+V), agent-based decomposition with RAG-enhanced coding (M+C^R+V), and full specialization with RAG-enhanced modeling and coding (M^R+C^R+V). Experimental results on two benchmark problems reveal that the targeted retrieval-augmented generation technique (RAG) significantly improves performance for complex modeling patterns such as piecewise functions, indicator constraints, and nested logic. We also propose a broader vision of C-DSS as a multi-agent ecosystem—including agents for visualization, explanation, verification, and orchestration—suggesting a path toward more explainable, adaptable, and intelligent decision support systems.**

I. Introduction

A. Backgrounds

OPERATIONS Research (OR) is grounded in analytical methods applied to improve decision-making. OR projects are often characterized by high complexity and a substantial risk of failure. Similar to software projects, they require the development and deployment of software; however, they also introduce a number of additional challenges. These include the need for a deeper understanding of the problem domain, a thorough comprehension of the decision-makers' goals and intentions, the construction of mathematical models, the selection or development of computational algorithms, and uncertainty regarding the quality of the resulting solutions. To address these challenges, OR methodology typically relies on a structured and systematic approach [1]. The individual steps of this methodology are illustrated in Figure 1.

The methodology begins with the real-world system that is the object of the decision-making process. The first step is to identify the decision problem. Next, the scope of the implementation is defined—this does not necessarily have to fully cover the previously identified area. The subsequent steps

Fig. 1. Operations research methodology enhanced with communication and potential languages used at various stages of the methodology

involve constructing a mathematical model and developing a solution method. The resulting solution is then tested using real data. If these tests are successful, the optimization results can be incorporated into actual decision-making processes, leading to full implementation and real-world impact—thus completing the methodology cycle. In practice, however, this process is rarely linear or strictly cyclical; backtracking or skipping steps may occur at various stages.

Effective communication is critical to the success of an OR project, and both its form and language vary across different phases of the methodology. Figure 1 enriches the classical OR methodology with examples of the communication modes used at each stage. In the early phases, interactions among domain experts dominate—for example, requirement elicitation and project scoping are typically conducted through expert meetings. During the modeling and method development

phases, mathematical and algorithmic languages are primarily used. Implementation requires programming languages, while solution validation often involves further expert consultations, sometimes supported by statistical language. Finally, the results must be communicated to end users via a human-machine interface, such as a dedicated GUI.

Large Language Models (LLMs) have emerged as transformative tools for effective communication across a wide range of domains. By leveraging vast amounts of linguistic data and advanced machine learning techniques, LLMs are capable of understanding context, generating coherent text, and engaging in meaningful dialogue. These capabilities enable them to assist with tasks such as summarizing information, translating languages, drafting content, and facilitating real-time interaction between users. Their ability to perform logical reasoning and apply mathematics to solve logical problems is rapidly advancing. In the domain of programming, they have become significant tools for automating source code generation and testing.

These capabilities make LLMs promising candidates for supporting the OR methodology at various stages. Their potential can be utilized in system discovery and requirement elicitation. They are naturally suited to support modeling and coding. Moreover, areas such as test automation, results explanation and visualization, or GUI generation are also promising fields where LLMs may play a significant role.

Recognizing these possibilities leads to the concept of Conversational Decision Support Systems (C-DSS)—a system in which an LLM serves as a proxy across multiple stages of Decision Support System (DSS) development and operation. The C-DSS concept touches on various aspects of building and operating decision support systems and opens up broad opportunities for future research.

In this paper, we introduce the C-DSS concept. However, due to the subject's complexity and scope, we focus specifically on the modeling and programming stages, and investigate how the effectiveness of language models can be enhanced at these steps. This work is a contribution to a broader vision—one that may significantly advance the field of decision support systems.

B. State of the art

Efforts to automate the process of solving optimization problems began gaining traction in 2022. One of the earliest contributions came from Ramamonjison et al. [2], who proposed a system that automatically formulates mathematical models from natural language descriptions. To evaluate their approach, they introduced the Linear Programming Word Problems (LPWP) dataset.

Building on this work, the same authors organized the NL4Opt Competition [3], which aimed to identify the most effective techniques for automated mathematical model formulation. The competition was based on the LPWP dataset—hereafter referred to as NL4OPT. In their conclusions, the organizers noted that solutions utilizing Large Language Models (LLMs) outperformed all other approaches.

Chen et al. introduced a tool named OptiChat that employs LLMs in conjunction with multiple agents, solvers (e.g., Gurobi, Mosek), and the algebraic modeling language Pyomo to support mathematical modeling [4]. Its core functionality includes detecting issues in model formulation and offering corrective suggestions to the user. At around the same time, Yang et al. proposed OPRO [5], a solution that bypasses traditional solvers by leveraging LLMs directly for optimization tasks. OPRO iteratively refines its solutions based on previous outputs and newly generated meta-feedback. However, this approach has so far only been tested on relatively simple problems like the traveling salesman problem and linear regression, leaving its applicability to more complex scenarios an open question.

Li et al. extended the research to Mixed-Integer Linear Programming (MILP) problems [6]. Their work expands the NL4OPT dataset and focuses on training LLMs to produce correct mathematical formulations, including not just quantitative constraints (e.g., resources, demand), but also logical constraints, which were largely neglected in earlier research. The issue of the simplicity of problems in data sets has been also addressed by Ziyang et al. [7]. The authors proposed the ComplexOR dataset, which features more challenging optimization tasks. Alongside the dataset, the authors introduced a multi-agent system implementing the idea of Chain-of-Experts framework. Although, they report performance improvement, there is no analysis on which problem aspects are hard and most problematic for LLMs.

Beyond the simplicity of tasks, the lack of real-world relevance was another concern. This gap was addressed by the creators of CoPilot [8], who focused on practical business scenarios. Their system guides users through understanding the problem, formulating the model, and solving the optimization task—once again, leveraging LLMs as the core engine.

One of the first comprehensive solutions attempting to address the entire modeling pipeline is OptiMUS [9]. This system not only formulates the optimization model but also generates executable solver code to find and return optimal solutions. Alongside the system, the authors introduced a new dataset—NLP4LP—containing both LP and MILP tasks to support their experiments. However, the limitation of OptiMUS was the need for users to prepare problem descriptions in a specific format. This issue has been addressed in a follow-up study [10], where the authors enhanced OptiMUS to automatically convert natural language inputs into the required structured format.

A significant challenge across all of the above efforts is the variability in both solver performance and LLM behavior. This issue is the focus of LM4OPT [11], a project that seeks to standardize evaluation procedures for such systems. Additionally, the researchers investigate the benefits of fine-tuning LLMs, including compact models like Llama-7B, to improve efficiency.

C. Identified gaps and our contribution

As the literature review indicates, this research area is relatively new, and current studies focus primarily on the use of LLMs for modeling and coding optimization problems, often overlooking the broader perspective of the OR methodology. Even within this narrow scope, there is currently no established knowledge regarding optimal architectures or methods for integrating LLMs into the OR process. With the exception of the limited work by Li et al. [6], there is a lack of analysis and identification which modeling artifacts pose the greatest challenges for language models. Such insights could help address specific issues and improve the effectiveness of automated modeling and coding efforts. Additionally, existing work mostly centers around the use of general-purpose programming languages, which rely on libraries for solving optimization problems. In particular, there is a notable absence of research evaluating the effectiveness of modeling using dedicated declarative languages, such as AMPL (A Mathematical Programming Language) [12]. Most of the research focuses on imperative languages, mainly Python, while optimization problems are natively defined in a declarative way.

Our contribution can be summarized as follows:

- We provide a broader context for mathematical modeling and coding by introducing the concept of Conversational Decision Support Systems (C-DSS) and taking a more comprehensive view of the OR methodology.
- We define a catalog of common artifacts in LP and MILP problems, along with associated sets of textual optimization problem descriptions.
- For the identified artifacts, we analyze the performance of LLMs within various agent-based architectures.
- Unlike previous studies, we employ LLMs to encode problems using the AMPL mathematical modeling language.
- We briefly demonstrate that LLM capabilities are also relevant at other stages of the OR methodology, opening opportunities for further research within the OR community.

II. Methods

A. Modeling artifacts

Based on practices in mathematical modeling, we have identified 20 typical modeling artifacts, summarized in Table I. The first five pertain to linear programming (LP) models and include: a simple LP model (artifact 1), minimization of the maximum value (artifact 2), a weighted sum of absolute values with weights a and b depending on the sign of the argument (absolute value in the objective, artifact 3), the absolute value in a constraint (artifact 4), and a piecewise linear approximation of a convex function $f_i(x)$ (artifact 5). By the basic LP model (artifact 1), we mean a situation where all variables directly represent the problem described in natural language, i.e., there is no need for intermediate variables.

The second group of artifacts concerns mixed-integer linear programming (MILP) problems and includes standard logical constraints typical for binary variables. These constraints typically involve the selection of at most/at least/exactly N elements (artifacts 6–8) and a range of conditional relationships (artifacts 9–14), in which binary variables may or must take certain values if other binary variables satisfy a given Boolean condition.

TABLE I
Modeling artifacts

LP	1	simple LP	no implicit variables
	2	min max	$\min x$
	3	absolute value in objective	$\min(ax^+ + bx^-)$
	4	absolute value in constraint	$\min(ax^+), x^- \leq X^-$
	5	easy linearization	$\min x, x \geq f_i(x)$
MILP (logic constr.)	6	at most N of x, y, z,...	$x + y + z + \cdots \leq N$
	7	at least N of x, y, z,...	$x + y + z + \cdots \geq N$
	8	exactly N of x, y, z,...	$x + y + z + \cdots = N$
	9	if x then y	$x \geq y$
	10	if x then not y	$x + y \leq 1$
	11	if x then y and z	$y \geq x, z \geq x$
	12	if y or z then x	$x \geq y, x \geq z$
	13	if y and z then x	$x \geq \frac{y+z}{2}$
	14	if M or more from x, y, z then v	$v \geq \frac{x+y+\ldots M+1}{N-M+1}$
MILP	15	min min	$\min \min f_i(x)$
	16	disjunction	$x \in \langle a; b\rangle \cup \langle b; d\rangle$
	17	min. activity level	$x = 0$ or $x \in \langle a, b\rangle$
	18	indicator constraint	$x \to C, x \in \{0, 1\}$
	19	concave linearization	$\min f(x), f(x)$ concave
	20	non-monotonic linearization	$\min f(x), f(x)$ non-mon.

The third group of artifacts involves more complex MILP models, which usually include non-convexities or discontinuities. This group comprises minimization of the minimum value (artifact 15), disjoint variable domains (artifact 16), minimum activity level (artifact 17), indicator constraint (activation of a linear constraint C based on a binary variable x, artifact 18), linearization of a concave function (artifact 19), and linearization of a general non-monotonic function (artifact 20).

B. Datasets

To evaluate which artifacts a large language model handles better or worse, we prepared two base problems: problem #64 from the NL4Opt dataset (the pharmaceutical paste problem, with minor linguistic adjustments) and our own original problem (the sewage discharging problem), which has never been published online. Each of the base problems represents artifact 1 — a simple LP problem. Subsequently, the content of each base problem was modified to sequentially introduce artifacts 2 through 20.

Although the dataset may seem small, it is important to note that we currently lack both the capability to automatically generate problem statements in a way that ensures the inclusion of specific artifacts, and methods for automatically verifying the correctness of models proposed by LLMs for given problem descriptions. Modifying problem statements to explicitly induce a given artifact is non-trivial and was carried out manually. Similarly, each solution in the form of a mathematical model obtained during the study was manually verified by an expert in operations research. Given the 20

artifacts, four agent configurations discussed later in the paper, and five runs for each problem and configuration, this results in 800 models whose correctness had to be manually verified.

The content of both problems is presented in Appendix A.

C. Environment settings

The conducted experiments cover four distinct language-agent environments, differing in the number of agents involved and their assigned roles. Each experimental setup processes a set of 40 test tasks, composed of two base problems, each instantiated in 20 variants corresponding to the predefined artifacts. The goal of each environment is to generate a mathematical model in textual form as well as a corresponding AMPL model.

The mathematical model (in text form) includes the definition and description of sets, parameters, and variables, along with the objective function and constraints, formulated in the style typical for LP/MILP problems. For the AMPL model, the system is allowed to produce either a single combined file containing both the model and data, or separate files—‘.mod‘ for the model and ‘.dat‘ for the problem data.

Since the experimental output consists of two distinct results—a textual mathematical model and AMPL source code—errors may occur independently in each. Therefore, we report separate success rates for modeling and coding.

Modeling success is defined as the correct formulation of the optimization problem in textual form, manually verified by an operations research (OR) expert. Any identified flaw in the model leads to its classification as incorrect. The ratio of correct mathematical models is reported in the result tables under the column labeled "Model."

Coding success is defined as the correct encoding of the previously generated mathematical model into AMPL syntax. Correctness was verified both manually by an OR expert and by executing the model in the AMPL environment. A model is considered correctly implemented if it runs without errors and matches the expected logic, regardless of whether the original formulation was valid. Hence, a syntactically correct AMPL implementation of an incorrect model still counts as a success in the coding phase. The ratio of correct AMPL models is reported in the column labeled "Code."

Overall success requires both a valid mathematical model and its correct AMPL implementation, i.e., success in both the modeling and coding stages. The ratio of fully correct and functional solutions is reported in the column labeled "Success."

All experiments were conducted using the GPT-4o model. For each combination of base problem, artifact, and agent configuration, the generation and evaluation process was repeated five times.

III. Results

A. Monolith architecture (M/C)

In the monolith architecture, we use an LLM as both modeler and coder (M/C) in a single call. The architecture is illustrated in Figure 2. The LLM acts as a single agent responsible for both modeling and coding. As input, it receives a word problem inserted into the following zero-shot prompt with Chain-of-Thought (CoT) reasoning enabled by default in GPT-4o:

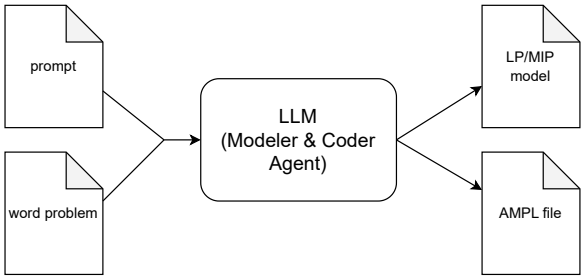

Fig. 2. Monolith architecture (M/C)

> *Formulate a linear programming model for the following problem.*
> ...
> *Provide the model as an AMPL file.*

Results are presented in Table II. Simple LP problems—artifacts 1 to 3—show a high success rate, reaching nearly 100% (only one coding failure occurred). These problems are considered easy for language models in both modeling and coding. The presence of an absolute value in a constraint (artifact 4) reduces the modeling success rate to 60%, though the coding success rate remains relatively high (80%). Artifact 5, which requires a linearization of a piecewise convex function, posed a significant challenge to the LLM and resulted in a very low modeling success rate (10%).

In the second group of artifacts, we observe high modeling success rates for most logical constraints, alongside surprisingly low coding success rates. Interestingly, artifact 13 was particularly challenging; the LLM often interpreted it as artifact 12. We believe this may be due to a linguistic ambiguity, as many people tend to confuse "and" with "or" in everyday language. Artifact 14 also resulted in a relatively low modeling success rate.

In the third group of artifacts, only the minimum activity level (artifact 17) yielded a high success rate. Other artifacts significantly reduced the LLM's ability to model correctly, with three artifacts never resulting in a valid model. However, this group showed a comparatively higher coding success rate than the second group.

Since the obtained results were not satisfactory, we considered several possible directions that may lead to improvements:

1) Fine tuning
2) Special prompting technique, including:
   - Tree-of-thoughts, Graph-of-thoughts
   - Generated knowledge – first get more knowledge about the problem, then model

TABLE II
RESULTS FOR M/C ARCHITECTURE

		Variant	Model	Code	Success
LP	1	simple LP	100%	100%	100%
	2	min max	100%	100%	100%
	3	abs. val. in obj.	100%	90%	90%
	4	abs. val. in constr.	60%	80%	40%
	5	piece-wise fun.	10%	70%	10%
MILP (logic)	6	at most N	100%	50%	50%
	7	at least N	100%	50%	50%
	8	exactly N	80%	40%	40%
	9	$x \Longrightarrow y$	90%	40%	40%
	10	$x \Longrightarrow \neg y$	100%	60%	60%
	11	$x \Longrightarrow y \wedge z$	100%	40%	40%
	12	$y \vee z \Longrightarrow x$	100%	50%	50%
	13	$y \wedge z \Longrightarrow x$	20%	40%	10%
	14	if $\geq M$ from $y \Longrightarrow y, z, \ldots$	50%	40%	30%
MILP	15	min min	0%	100%	0%
	16	disjun.	50%	90%	50%
	17	min. activ. level	100%	90%	90%
	18	indic. constr.	20%	80%	20%
	19	concave linear.	0%	50%	0%
	20	non-monoton. linear.	0%	50%	0%

- Cognitive verifier
- Few-shot – providing examples
- Persona/audiance pattern
- Reflection
- Chain-of-Experts
- ...

3) Agent-based decomposition (mixture-of-experts)
4) Retrieval Augmented Generation (RAG)

For further research, we exclude fine-tuning, as it requires additional examples that are difficult to generate in this domain and may be computationally expensive. We also do not include any special prompting techniques, as it remains unclear which approaches could reliably improve modeling and coding quality. Our observations suggest that for different artifacts, either modeling or coding may be the more challenging task. Following this insight, we believe that decomposition into specialized agents—potentially equipped with additional context and expert knowledge and implemented as RAG—is the most promising solution. The next three environment settings follow this concept.

B. Enhancing M/C with Code Verifier agent

In this setting, we build on the observation that the coding success rate is relatively low, particularly for the second group of artifacts, and that most failures are due to simple syntax issues. We introduced a new component—the verifier agent—responsible for validating the AMPL source file by executing it in the AMPL environment and reporting any errors encountered. The resulting architecture is illustrated in Figure 3.

If an error occurs, the verifier sends the following prompt, including the error message, back to the Modeler/Coder (M/C) agent:

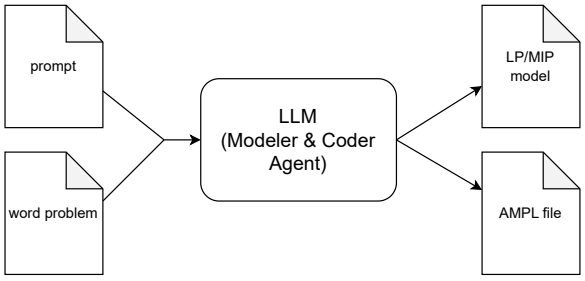

Fig. 3. M/C architecture enhanced with verifier agent (M/C+V)

TABLE III
RESULTS FOR M/C+V ARCHITECTURE

		Variant	Model	Code	Success
LP	1	simple LP	100%	100%	100%
	2	min max	100%	100%	100%
	3	abs. val. in obj.	100%	**100%**	**100%**
	4	abs. val. in constr.	60%	**90%**	**50%**
	5	piece-wise fun.	10%	**80%**	10%
MILP (logic)	6	at most N	100%	**60%**	**60%**
	7	at least N	100%	**50%**	**50%**
	8	exactly N	80%	**70%**	50%
	9	$x \Longrightarrow y$	90%	**50%**	**50%**
	10	$x \Longrightarrow \neg y$	100%	60%	60%
	11	$x \Longrightarrow y \wedge z$	100%	**50%**	**50%**
	12	$y \vee z \Longrightarrow x$	100%	**60%**	**60%**
	13	$y \wedge z \Longrightarrow x$	20%	**50%**	10%
	14	if $\geq M$ from $y \Longrightarrow y, z, \ldots$	50%	**60%**	**40%**
MILP	15	min min	0%	100%	0%
	16	disjun.	50%	**100%**	50%
	17	min. activ. level	100%	**100%**	100%
	18	indic. constr.	20%	**90%**	20%
	19	concave linear.	0%	**90%**	0%
	20	non-monoton. linear.	0%	50%	0%

AMPL reported the following issues:

```
...
test.mod, line 2 (offset 54):
 A is not defined
context: set CONTAINERS := A >>> B <<< C D;
No variables declared.

...
```

Fix all errors.

The M/C agent may respond by correcting the identified issues and regenerating the AMPL source code, which is then returned to the verifier. We allow only a single round of back-propagation. Although further iterations are technically possible, we observed that they do not improve success rates and may lead to repetitive error loops.

Results for the M/C+V architecture are presented in Table III. Changes compared to the M/C architecture (Table II) are shown in bold. We observe improvements in the coding success rate for nearly all artifacts where it was previously below 100%. The typical improvement is around 10 percentage points, but it reaches as much as 40 percentage points for artifact 19.

TABLE IV
RESULTS FOR M+C^R+V ARCHITECTURE

		Variant	Coder	Coder + RAG
LP	1	simple LP	100%	100%
	2	min max	100%	100%
	3	abs. val. in obj.	100%	100%
	4	abs. val. in constr.	80%	80%
	5	piece-wise fun.	60%	60%
MILP (logic)	6	at most N	60%	60%
	7	at least N	20%	**60%**
	8	exactly N	80%	80%
	9	$x \Longrightarrow y$	40%	**80%**
	10	$x \Longrightarrow \neg y$	40%	**60%**
	11	$x \Longrightarrow y \wedge z$	0%	**20%**
	12	$y \vee z \Longrightarrow x$	20%	**60%**
	13	$y \wedge z \Longrightarrow x$	0%	**40%**
	14	if $\geq M$ from $y \Longrightarrow y, z, \ldots$	20%	**60%**
MILP	15	min min	100%	100%
	16	disjun.	100%	100%
	17	min. activ. level	100%	100%
	18	indic. constr.	100%	100%
	19	concave linear.	80%	80%
	20	non-monoton. linear.	80%	**100%**

TABLE V
SHARE OF CODING ERRORS IN M/C AND M+C^R+V ARCHITECTURES

	CODER	CODER + RAG
set definition syntax	54.6%	38.1%
many objectives	4.6%	0.0%
two-side constr.	9.0%	2.4%
multiply constr. in one "s.t."	4.6%	4.8%
double definitions	4.6%	0.0%
syntax	22.7%	54.8%

TABLE VI
IMPROVEMENTS FOR M^R+C^R+V ARCHITECTURE VERSUS M/C ARCHITECTURE

		Variant	Model M/C	Model RAG
MILP LP	5	piece-wise fun.	10%	**100%**
MILP	13	$y \wedge z \Longrightarrow x$	20%	**40%**
Hard MILPs	15	min min	0%	**50%**
	18	indic. constr.	20%	**80%**
	19	concave linear.	0%	**70%**
	20	non-monoton. linear.	0%	**80%**

C. Separation of Modeler and Coder (M+C^R+V)

To further improve coding quality, we decomposed the monolithic M/C architecture into separate modeling and coding agents. This allowed us to augment the coder with Retrieval-Augmented Generation (RAG). Initially, we used chapters 1–9 from the original AMPL book [12], with each chapter provided as a separate PDF file. However, we observed no significant improvement.

Subsequently, we created a concise custom RAG resource focused on recurring coding issues. The RAG content is included in Appendix B. This short but targeted resource significantly improved outcomes—particularly for the second problem, which had previously suffered from poor coding quality. Table IV presents coding success rates for the second problem under the monolithic architecture (column "Coder") and the M+C^R+V architecture (column "Coder + RAG"). Changes are marked in bold.

As shown, coding was particularly weak in the second group of artifacts. The addition of a compact RAG significantly improved the coding success rate—in some cases, tripling it (e.g., artifacts 7 and 14).

Table V presents the distribution of error types reported by the AMPL environment. Several types of errors occur frequently. In the M/C architecture, more than half of the errors relate to syntax issues in set definitions. Nine percent concern the use of two-sided constraints, while similar proportions pertain to multiple constraints under a single 'subject to' statement, double definitions, and multiple objective functions. The remaining errors are categorized as general syntax issues.

When RAG is introduced, the distribution shifts. The frequency of specific known errors drops—sometimes to zero. As a result, the share of general syntax errors increases. However, the total number of errors decreases, indicating that the targeted RAG resource is effective. It successfully reduces both the frequency and proportion of the common, previously observed issues.

D. Enhancing Modeler agent with RAG: M^R+C^R+V

Encouraged by the positive impact of a relatively small RAG set for the Coder, we identified common modeling issues and developed a dedicated RAG for the Modeler agent as well. The resulting RAG is a two-page document that provides guidelines for addressing the most frequent challenges faced by the Modeler agent. Appendix C presents its contents, divided into three sections: piecewise linear function modeling, the minimum of minima problem, and indicator constraints.

Table VI shows the improvements achieved over the monolithic M/C model for a selection of the most challenging artifacts covered by the new RAG. The observed gains are significant, and the effectiveness of the targeted RAG should be considered high.

Table VII presents the complete results for the M^R+C^R+V architecture, which includes decomposition into specialized agents, each enhanced with its own RAG. Although the results are significantly improved compared to the original monolithic M/C architecture, some limitations remain. Notably, coding artifacts from the second group still show signs of weakness. Additionally, modeling performance remains suboptimal for artifacts 13–16. Among the simpler problems, artifact 4—featuring absolute values in constraints—underperforms relative to others in this group.

IV. THE CONCEPT OF C-DSS

Our experience with decomposing the architecture into specialized agents leads us to propose a general framework for Conversational Decision Support Systems (C-DSS), as illustrated in Figure 4. The agents marked in bold represent the components studied in this work: the Modeler, Coder,

TABLE VII
RESULTS FOR M^R+C^R+V ARCHITECTURE

		Variant	Model	Code	Success
LP	1	simple LP	100%	100%	100%
	2	min max	100%	100%	100%
	3	abs. val. in obj.	100%	100%	100%
	4	abs. val. in constr.	60%	90%	60%
	5	piece-wise fun.	100%	80%	80%
MILP (logic)	6	at most N	100%	60%	60%
	7	at least N	100%	50%	50%
	8	exactly N	80%	70%	50%
	9	$x \Longrightarrow y$	90%	50%	50%
	10	$x \Longrightarrow \neg y$	100%	60%	60%
	11	$x \Longrightarrow y \wedge z$	100%	50%	50%
	12	$y \vee z \Longrightarrow x$	100%	60%	60%
	13	$y \wedge z \Longrightarrow x$	40%	50%	40%
	14	if $\geq M$ from $y \Longrightarrow y, z, \ldots$	50%	60%	40%
MILP	15	min min	50%	100%	50%
	16	disjun.	50%	100%	50%
	17	min. activ. level	100%	100%	100%
	18	indic. constr.	80%	90%	80%
	19	concave linear.	70%	90%	60%
	20	non-monoton. linear.	80%	50%	80%

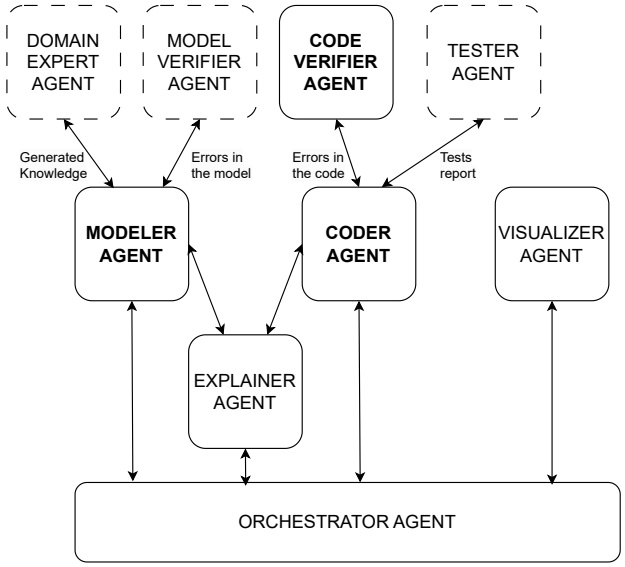

Fig. 4. The concept of Conversational Decision Support Systems

and Code Verifier. However, other elements of a classical Decision Support System (DSS) also involve communication and can be implemented within a language-agent-based architecture.

One example is the DSS user interface, whose role can be taken over by a *Visualizer agent*, dynamically responding to user requests for result visualization. Figure 5 shows a sample response from the Visualizer agent for base problem one, given the following prompt:

> *Visualize the results for the Management Board*

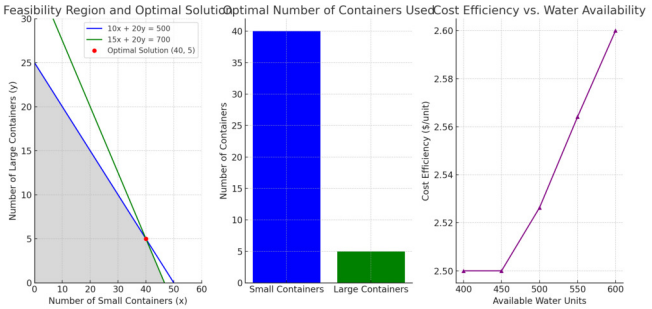

Fig. 5. Visualization of results proposed by the Visualizer agent (overlapping text left as it was generated by the agent)

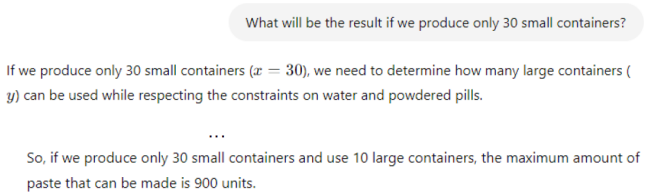

Fig. 6. Exemplary session 1 with Explainer agent

The agent proposed a visualization of the feasible solution set, which may not be suitable for a Management Board. It also presented the solution using a simple bar chart. Interestingly, the agent suggested performing an efficiency analysis with respect to water availability, thereby independently initiating a non-trivial sensitivity analysis.

Although the resulting output is not perfect and contains some visual flaws (e.g., overlapping text), it is important to note that the agent used was neither trained nor equipped with RAG, and the conversation was brief and based on a simple prompt. In this context, the results are promising and illustrate the potential for further development of such interfaces.

Another example is the *Explainer agent*, specialized in interpreting and explaining the obtained results to the user. In particular, it can conduct sensitivity or what-if analyses, and thus collaborates closely with the Modeler and Coder agents.

Figures 6 and 7 present two short sessions with the Explainer agent for base problem one. In the first session, the user seeks reassurance that the solution proposed by the C-DSS is valid and asks about the impact of reducing the number of small containers produced. The agent responds with a detailed explanation; Figure 6 shows only the opening sentence and final conclusion.

In the second session, the user wants to understand why increasing the production of small containers is not feasible. As before, the agent provides a longer explanation, and Figure 7 shows the first and final statements. In both cases, the Explainer agent offers a fairly detailed and accurate analysis of the user's proposed changes.

Assuming the Modeler is a general-purpose agent, it is reasonable to enable communication with a domain-specific agent that provides expert knowledge in a given field. While

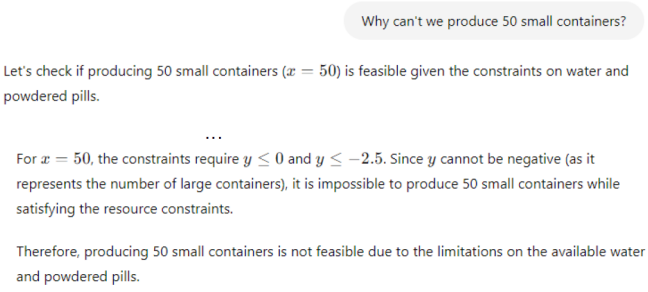

Fig. 7. Exemplary session 2 with Explainer agent

the implementation details remain open, introducing a Critic agent—responsible for verifying and validating the developed mathematical model (i.e., a Model Verifier)—appears well justified. This agent could replace or support the function of a human expert, which proved to be a bottleneck in our experiments.

In the area of code development, the current Code Verifier only checks for syntactic correctness by testing whether the code runs in a given environment. However, a dedicated *Tester agent* should also be envisioned—capable of validating the encoded model using predefined unit tests or scenario checks.

Finally, the entire agent ecosystem requires workflow management, a task that would fall to an *Orchestrator agent*. This agent would be responsible for dynamically arranging tasks, coordinating agent communication, and preventing loops or deadlocks in the system.

V. Summary

This paper introduces the concept of Conversational Decision Support Systems (C-DSS), which leverage Large Language Models (LLMs) to enhance Operations Research (OR) methodology, particularly during the modeling and coding stages. We propose a novel framework in which LLMs serve as proxies across various phases of decision support system development, with a focus on mathematical modeling and AMPL code generation.

By identifying 20 typical modeling artifacts for Linear Programming (LP) and Mixed-Integer Linear Programming (MILP), we evaluate LLM performance across four agent-based architectures: Monolith (M/C), M/C with Verifier (M/C+V), Modeler and Coder with Retrieval-Augmented Generation (M+C^R+V), and both Modeler and Coder enhanced with RAG (M^R+C^R+V). Using two base problems—one from the NL4Opt dataset and a custom-designed sewage discharging problem—we observe that simpler LP artifacts yield near-perfect success rates. In contrast, complex MILP artifacts involving logical constraints or non-convex functions remain significantly more challenging.

Our experiments reveal patterns in LLM behavior, suggesting that targeted solutions to specific artifacts offer great potential for improvement. We demonstrate that even a very short RAG for the Coder agent and a slightly longer, yet focused, RAG for the Modeler agent can significantly improve performance—particularly for piecewise functions, minimum-of-minima formulations, and indicator constraints.

Fig. 8. Potential role of LLM in OR methodology

We also envision a more comprehensive C-DSS architecture, incorporating agents such as the Modeler, Coder, Verifier, Visualizer, Explainer, and Orchestrator. This multi-agent design offers a unified approach to integrating LLMs into OR and opens new avenues for research in automating decision support systems. A Decision Support System is a product developed within the broader framework of the OR methodology, as referenced in Figure 1. As shown, the various steps in the methodology rely on different forms of language to facilitate communication among stakeholders—making it inherently well-suited for enhancement through LLMs. Therefore, we propose considering a revised OR methodology in which language models serve as core tools at each stage. Figure 8 illustrates this vision.

Despite rapid advances in large language models, there are still significant limitations. Our results show that for some quite typical cases, LLMs are not a reliable tool for modelling and coding. Since the real problems are typically more complex and may include more sophisticated aspects, such as complex nonlinearity, dynamics, or uncertainty, it must be admitted that the C-DSS concept is in its early stages of development. It requires a significant improvement for real applications. However, we believe that our proposed C-DSS concept not only advances the design of automated decision systems but also lays the foundation for further research into LLM-enhanced OR methodologies.

APPENDIX

A. Word problems used in the research

Base problem 1 (#64 from NL4Opt data set)

> There are two specialized containers, a small and large one, that are used to make a pharmaceutical paste. The small container requires 10 units of water and 15 units of the powdered pill to make 20 units of the paste. The large container requires 20 units of water and 20 units of the powdered pill to make 30 units of the paste. The pharmacy has 500 units of water and 700 units of powdered pills available. How many of each container should be used to maximize the amount of paste that can be made?

Base problem 2 (the sewage discharging problem)

> The two cities discharge sewage into two treatment plants with a capacity of 5,000 tons and 3,000 tons per day, respectively. The purified water is pumped further into the river. Each city can divide sewage in any proportion between both treatment plants. The daily operating cost of treatment plant 1 is USD 3.5/ton. The daily operating cost of treatment plant 2 is USD 6/ton. The operating cost of the clean water pumping station at treatment plant 1 is USD 2.5/ton, and at treatment plant 2 - USD 2.2/ton. Assuming that cities must discharge at least 4,000 and 3,500 tons of sewage per day, plan the system's operation to minimize the daily cost of its operation.

B. Content of Coder's RAG

> This GPT acts as an expert in linear programming and AMPL (A Mathematical Programming Language). It assists users in creating AMPL files for given linear programming problems by understanding their requirements and translating them into correct and optimized AMPL code. It ensures the AMPL files are syntactically accurate and efficient, adhering to best practices in linear programming and AMPL usage. It will guide users through the process of defining sets, parameters, variables, objective functions, and constraints in their linear programming models.
> Whenever you are asked to create an AMPL file, follow these rules:
> 1) Start with the declarations of sets followed by declarations of parameters and variables.
> 2) When declaring a set of parameters, do not assign any value. Add the section "data;" at the end of the AMPL file and put all assignments there.
> 3) Do not use sharp inequalities. Always apply "lower than or equal" or "greater than or equal".
> 4) Do not use double inequality constraints. Break the double inequalities into two separate constraints.
> 5) There can be only one objective in the AMPL file.
> 6) 'var' definitions should not include assignment from other variables directly.
> 7) Under the keyword "subject to", only one constraint can be defined. If more constraints are needed, each constraint definition should start with "subject to".

C. Content of Modeler's RAG

> **Piecewise functions**
> Remember that when dealing with easy piecewise function cases, there is NO NEED to use auxiliary binary variables. It is enough to add constraints for each segment of the piecewise function. For instance, if the problem is minimizing the convex piecewise function, we can add auxiliary continuous variable x and a constraint $x \geq f_i(x)$, where $f_i(x)$ is an i-th segment of the piecewise function. However, if you add binary variables indicating the segment of the function, these variables must be constrained by the original variable. If the i-th linear segment is denoted by $f_i(x)$ and it is valid in the range from x_{i-1} to x_i, then we need to set auxiliary variable v_i to 1 whenever x is in the range from x_{i-1} to x_i. We can model that as follows:
>
> $$x \leq x_i v_i + M(1 - v_i)$$
> $$x \geq x_{i-1} v_i - M(1 - v_i)$$
>
> For instance, let's consider the function consisting of two piecewise segments, $f_1(x)$ and $f_2(x)$. For x from 5 to 10, there is function $f_1(x)$, and for x in a range from 10 to 18, there is function $f_2(x)$. We introduce auxiliary variables v_1 and v_2 such that
>
> $$v_1 + v_2 = 1$$
>
> To constraint variable x by variables v_1 and v_2, we define inequalities as follows
>
> $$x \leq 10v_1 + M(1 - v_1)$$
> $$x \geq 5v_1 \breve{} M(1 - v_1)$$
> $$x \leq 18v_2 + M(1 - v_2)$$
> $$x \geq 10v_2 \breve{} M(1 - v_2)$$
>
> To calculate the value of the piecewise function, we introduce the auxiliary continuous variable z, and we define the following constraints:
>
> $$z \geq f_1(x) \breve{} M(1 - v1)$$
> $$z \leq f_1(x) + M(1 - v1)$$
> $$z \geq f_2(x) \breve{} M(1 - v2)$$
> $$z \leq f_2(x) + M(1 - v2)$$
>
> In the final version of the model, $f_1(x)$ and $f_2(x)$ must be substituted with linear formulas.

Min of min problem (minimization of the minimal value)
If the problem is to minimize the minimum of some values, we need to add auxiliary binary variables that indicate which value is the lowest. Let us consider variables x and y and the problem of minimizing the minimal value out of variables x and y. Then, we must introduce a binary variable v that is 0 if x is lower or equal to y, and v is 1 if y is lower than or equal to x. In the case of $x = y$, the value of v can be any. To model that, we need to add the following constraints:

$$x \leq y + Mv$$

$$y \leq x + M(1 - v)$$

Having variable v that indicates whether x or y is the minimal value we must introduce auxiliary continuous variable z that will be equal x if v is 0 or to y if v is 1. It is an indicator constraint that can be modeled as follows:

$$z <= x + Mv$$

$$z >= x\breve{}Mv$$

$$z <= y + M(1 - v)$$

$$z >= y - M(1 - v)$$

Remember that z must be constrained from above and below. Since z is the minimal value of x and y, it can be directly minimized in the objective:

$$\min z$$

WARNING! Remember, YOU CAN NOT SIMPLY constraint z in this way:

$$z \leq x$$

$$z \leq y$$

and minimize z. This is MISTAKE! Variable z will always be 0. You must use an auxiliary binary variable. Watch out for this case!

Indicator constraints
An indicator constraint is a constraint that is controlled by a binary variable. If the constraint is "lower than", and the auxiliary variable that controls this constraint is z, then we can add Mz to the right-hand side of the constraint, where M is a big number. If the original constraint is as follows:

$$\sum_i a_i x_i \leq c$$

then it can be controlled with a binary variable y in the following way:

$$\sum_i a_i x_i \leq c + Mz$$

Indicator constraint can be used to model conditional constraint in which one variable depends on the value of another variable. For instance, if x must be greater or equal to A when y is greater or equal to B, we can add the indicator constraint for variable y as follows

$$y \geq B - M(1 - z)$$

where z is an auxiliary binary variable. So if z gets 1 then the constraint becomes active. Then z must be set to 1 if x is greater or equal A, which can be modeled as follows:

$$z \geq (x - A)/M$$

Remember that in the general case of conditional constraint, we need to use the concept of the indicator constraint, so an auxiliary variable is needed.

REFERENCES

[1] F. S. Hillier, *Introduction to operations research*, 9th ed. Boston [etc.]: McGraw-Hill, 2010. ISBN 9780071267670

[2] R. Ramamonjison, H. Li, T. Yu, S. He, V. Rengan, A. Banitalebi-dehkordi, Z. Zhou, and Y. Zhang, "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*, Y. Li and A. Lazaridou, Eds. Abu Dhabi, UAE: Association for Computational Linguistics, Dec. 2022. doi: 10.18653/v1/2022.emnlp-industry.4 pp. 29–62.

[3] R. Ramamonjison, T. Yu, R. Li, H. Li, G. Carenini, B. Ghaddar, S. He, M. Mostajabdaveh, A. Banitalebi-Dehkordi, Z. Zhou, and Y. Zhang, "Nl4opt competition: Formulating optimization problems based on their natural language descriptions," in *Proceedings of the NeurIPS 2022 Competitions Track*, ser. Proceedings of Machine Learning Research, M. Ciccone, G. Stolovitzky, and J. Albrecht, Eds., vol. 220. PMLR, 28 Nov–09 Dec 2022. doi: 10.48550/arXiv.2303.08233 pp. 189–203. [Online]. Available: https://proceedings.mlr.press/v220/ramamonjison23a.html

[4] H. Chen, G. E. Constante-Flores, and C. Li, "Diagnosing infeasible optimization problems using large language models," 2023.

[5] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," 2024.

[6] Q. Li, L. Zhang, and V. Mak-Hau, "Synthesizing mixed-integer linear programming models from natural language descriptions," 2023.

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

[8] S. Wasserkrug, L. Boussioux, D. den Hertog, F. Mirzazadeh, I. Birbil, J. Kurtz, and D. Maragno, "From large language models and optimization to decision optimization copilot: A research manifesto," 2024.

[9] A. AhmadiTeshnizi, W. Gao, and M. Udell, "Optimus: Optimization modeling using mip solvers and large language models," 2023.

[10] ——, "Optimus: Scalable optimization modeling with (mi)lp solvers and large language models," 2024.

[11] T. Ahmed and S. Choudhury, "Lm4opt: Unveiling the potential of large language models in formulating mathematical optimization problems," *INFOR: Information Systems and Operational Research*, vol. 62, no. 4, pp. 559–572, 2024. doi: 10.1080/03155986.2024.2388452. [Online]. Available: https://doi.org/10.1080/03155986.2024.2388452

[12] R. Fourer, D. Gay, and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, ser. Scientific Press series. Thomson/Brooks/Cole, 2003. ISBN 9780534388096. [Online]. Available: https://books.google.com.hk/books?id=Ij8ZAQAAIAAJ