ORMind: A Cognitive-Inspired End-to-End Reasoning Framework for Operations Research

Zhiyuan Wang[1†*], **Bokui Chen**[1,5†], **Yinya Huang**[3], **Qingxing Cao**[4‡], **Ming He**[2‡], **Jianping Fan**[2], **Xiaodan Liang**[4,5]

[1]Tsinghua Shenzhen International Graduate School, Tsinghua University, [2]AI Lab of Lenovo Research, [3]ETH Zurich, [4]Sun Yat-sen University, [5]Peng Cheng Laboratory

{wang-zy22, chenbk}@tsinghua.edu.cn, yinya.huang@hotmail.com heming01@foxmail.com ,caoqx@mail2.sysu.edu.cn, jfan1@lenovo.com, xdliang328@gmail.com

Abstract

Operations research (OR) is widely deployed to solve critical decision-making problems with complex objectives and constraints, impacting manufacturing, logistics, finance, and healthcare outcomes. While Large Language Models (LLMs) have shown promising results in various domains, their practical application in industry-relevant operations research (OR) problems presents significant challenges and opportunities. Preliminary industrial applications of LLMs for operations research face two critical deployment challenges: 1) Self-correction focuses on code syntax rather than mathematical accuracy, causing costly errors; 2) Complex expert selection creates unpredictable workflows that reduce transparency and increase maintenance costs, making them impractical for time-sensitive business applications. To address these business limitations, we introduce ORMind, a cognitive-inspired framework that enhances optimization through counterfactual reasoning. Our approach emulates human cognition—implementing an end-to-end workflow that systematically transforms requirements into mathematical models and executable solver code. It is currently being tested internally in Lenovo's AI Assistant, with plans to enhance optimization capabilities for both business and consumer customers. Experiments demonstrate that ORMind outperforms existing methods, achieving a 9.5% improvement on the NL4Opt dataset and a 14.6% improvement on the ComplexOR dataset.

1 Introduction

Operations research (OR) is critical for business decision-making, helping companies optimize resources, reduce costs, and improve operational efficiency across manufacturing, logistics, and supply chain management. However, previous approaches usually require specialized expertise to translate real-world problems into mathematical optimization problems, hindering their application potential in broader domains. Industry practitioners consistently report that optimization projects face a 30-40% failure rate due to the disconnect between business requirements and mathematical formulation.

Recent advancements in LLMs have enabled the solving of OR problems. Such automation procedures can avoid inconsistent math performance of LLMs (Ahn et al., 2024; Imani et al., 2023; Yu et al., 2024a) and leverage LLMs' ability and knowledge to extract implicit variables and constraints from real-world problems.

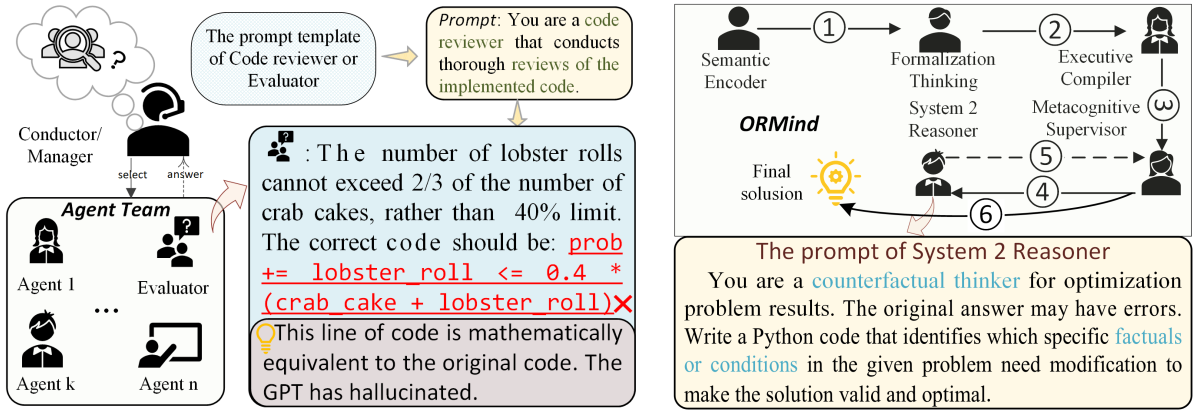
However, as Figure 1a illustrates, existing approaches(Xiao et al., 2024; Wang et al., 2024; AhmadiTeshnizi et al., 2024) to operations research automation face critical deployment challenges. Their complex agent orchestration creates excessive cognitive load through numerous API calls, overwhelming analysts with irrelevant information while significantly increasing costs. These unpredictable expert selection processes reduce solution transparency and create substantial overhead, fundamentally misaligning with human reasoning capabilities.

Inspired by cognitive science and how the brain solves problems, ORMind implements a business-oriented framework based on dual-process theory, combining intuitive analysis with deliberate reasoning. Our specialized modules mirror analyst workflows, from rapid comprehension to deep mathematical thinking. Unlike existing multi-agent frameworks that rely on unpredictable agent selection and complex orchestration, ORMind's innovation lies in its structured, predictable workflow that drastically reduces API calls while maintaining solution quality. ORMind framework is shown in Figure 1b.

We evaluate ORMind on standard benchmark

[*]Work done as an intern at AI Lab of Lenovo Research.
[†]Equal contributions.
[‡]Corresponding authors.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track), pages 104–131
July 28-30, 2025



(a) (b)

Figure 1: Current frameworks rely on complex agent orchestration with unpredictable execution paths, dramatically increasing API calls and computation time. Their focus on code syntax rather than mathematical accuracy results in costly errors that can propagate through business operations undetected. This excessive coordination overhead makes these systems impractical for time-sensitive business applications. Compared to traditional methods, ORMind employs a streamlined end-to-end workflow with counterfactual reasoning, significantly enhancing solution reliability.

datasets and complex OR problems in industrial scenarios, creating more trustworthy AI systems for business applications. Our contributions include:

- An industry-focused framework that streamlines optimization workflows.
- A counterfactual reasoning methodology for business-critical constraint validation.
- A workflow that improves solution trustworthiness and clarity, reducing implementation risks.

2 Related Work

Operations Research Solving with LLMs. Operations research problem solving (Ramamonjison et al., 2022; AhmadiTeshnizi et al., 2024; Xiao et al., 2024) contains multiple and diverse applied mathematical problems that require a model to perform complex understanding and reasoning. A traditional line of approaches (Ramamonjison et al., 2022) decomposes the OR solving into two separate tasks, first solving the NER task to recognize the optimization problem entities (He et al., 2022), then generating a precise meaning representation of the optimization formulation (Gangwar, 2022). Another line of work (Tang et al., 2024; Yang et al., 2024) leverages LLMs to synthesize abundant and diverse operations research problems, which later empowers the LLMs with such synthetic data. Such approaches may suffer guaranteed data quality and, at the same time, can be costly.

LLM-based Multi-Agent Workflow Recent research has demonstrated the potential of collaborative problem-solving through autonomous cooperation among AI agents (Li et al., 2023; Wang et al., 2024; Hong et al., 2024a). Compared with existing multi-agent collaboration approaches, ORMind's primary innovation lies in its counterfactual strategy and memory pool coordination mechanism, which aligns more closely with actual business decision-making logic and transparency requirements. This enables the system to exhibit unique advantages in industrial NLP problem scenarios.

LLM-based Reasoning Frameworks. Recent advancements in LLMs have introduced various innovative frameworks to enhance their complex reasoning capabilities. For example, for solving mathematical problems in such as textbooks and contests (Cobbe et al., 2021; Hendrycks et al., 2021; Lightman et al., 2023; Zheng et al., 2022), current research efforts (Gou et al., 2024; Zhu et al., 2023a; Yu et al., 2024b; Hao et al., 2024) have explored using LLMs via employing various structures to enhance reasoning fidelity.

However, these single-agent reasoning methods demonstrate notable shortcomings when dealing with intricate Operations Research (OR) problems. This is because they struggle to address the combined challenges of implicit constraints and factual hallucination on knowledge-intensive tasks.

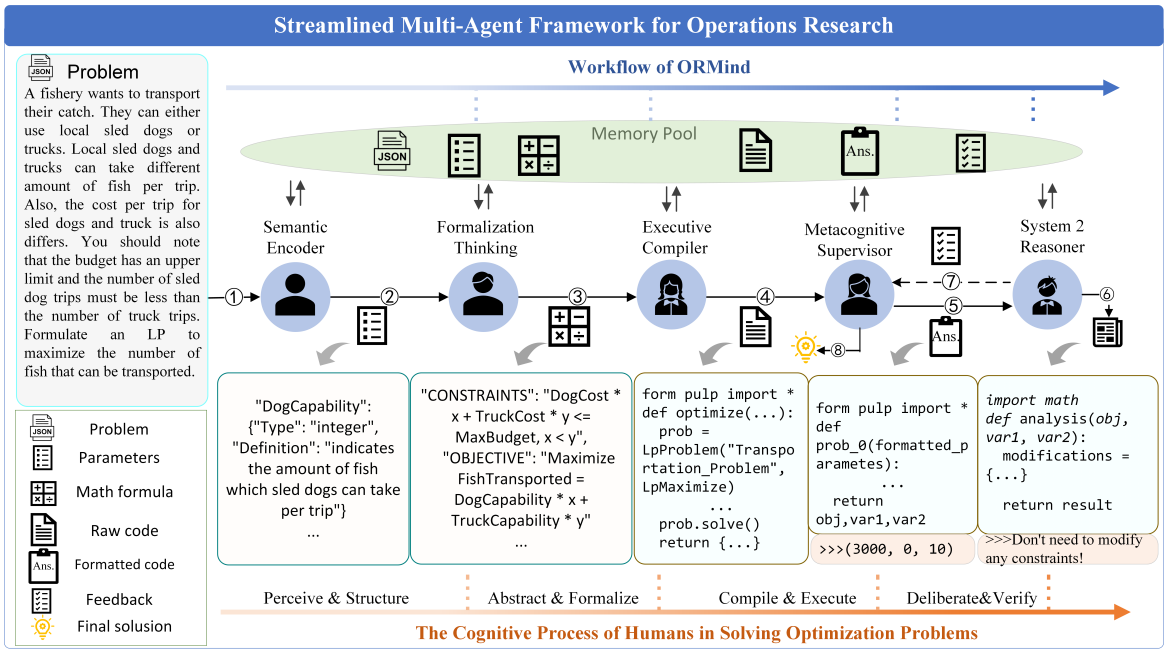


Figure 2: Our approach is grounded in established cognitive science theories, particularly dual-process framework(Kahneman, 2011) and tripartite model of cognition(Stanovich, 2009). The Semantic Encoder and Formalization Thinking modules correspond to Type 1 (intuitive) processing, while the System 2 Reasoner implements Type 2 (analytical) processing. The Metacognitive Supervisor embodies the reflective mind, monitoring and coordinating between these systems.

3 Methodology

3.1 Problem Formulation

Optimization problems are typically expressed in mathematical terms, consisting of an objective function to be minimized or maximized, subject to a set of constraints. For instance, a Integer Linear Program can be formulated mathematically as:

$$\text{minimize} \quad \sum_{j=1}^{n} c_j x_j \tag{1}$$

$$\text{subject to} \sum_{j=1}^{n} a_{ij} x_j \leq b_i, i = 1, \ldots, m \tag{2}$$

$$l_j \leq x_j \leq u_j, \quad j = 1, \ldots, n \tag{3}$$

$$x_j \in \mathbb{Z}, \quad j \in I \tag{4}$$

3.2 Architecture Overview

As illustrated in Figure 2, when humans solve optimization problems, their cognitive process aligns with our framework. The brain first performs semantic encoding, rapidly identifying key variables from complex descriptions. It then uses formalization thinking, methodically constructing mathematical relationships between variables and constraints. Next, executive compiler translate these abstract models into actionable solution.

With problem input D and agent sequence $\mathbb{A} = \{A_{\phi_1}, A_{\phi_2}, \ldots, A_{\phi_{N_a}}\}$, where N_a represents total agents and ϕ_k denotes agent-specific configurations, each component builds upon previous outputs stored in memory pool P.

The transformation operation for agent k follows:

$$O_k = A_{\phi_k}(D, P_{k-1})$$

where D represents business requirements input and P contains all previously processed outputs. Each agent's contribution O_k incrementally enhances the solution repository:

$$P_k = P_{k-1} \cup \{O_k\}$$

This collaborative memory architecture enables robust business optimization by leveraging specialized expertise while maintaining a comprehensive solution context—critical for enterprise deployments where reliability and solution quality directly impact operational outcomes.

3.3 Brief Introduction of Components

3.3.1 Semantic Encoder

The Semantic Encoder transforms unstructured text into structured knowledge representations, reducing the working memory load. It recognizes and

Algorithm 1 Workflow of ORMind

Require: Pre-processed problems set $\mathbb{D}$=$\{D_1, D_2, \ldots, D_{N_T}\}$, maximum number of problems N_T, Memory Pool accessible to all modules

Ensure: Optimized solutions $S_1^*, S_2^*, \ldots, S_{N_T}^*$

for $t = 1$ to N_T **do**
  $\Theta_t \leftarrow \text{SemanticEncoder}(D_t)$
  $M_t \leftarrow \text{Formalization}(D_t, \Theta_t)$
  $C_t \leftarrow \text{ExecutiveCompiler}(M_t)$
  $F_t \leftarrow \text{Supervisor}_\text{f}(D_t, \Theta_t, M_t, C_t)$
  $S_t \leftarrow F_t$ ▷ Run the code
  if S_t indicates any error **then**
    $R_t \leftarrow \text{Reasoner}(S_t, F_t)$
    $F_t' \leftarrow \text{Supervisor}(D_t, \Theta_t, M_t, C_t, R_t)$ ▷ Revise the code based on errors
    $S_t \leftarrow F_t'$ ▷ Run the code
  end if
  $R_t \leftarrow \text{Reasoner}(S_t, D_t)$
  if R_t indicates discrepancies with fact **then**
    $F_t' \leftarrow \text{Supervisor}(D_t, \Theta_t, M_t, C_t, R_t)$
  else
    $S_t^* \leftarrow F_t'$ ▷ Get solution
  end if
end for
return $S_1^*, S_2^*, \ldots, S_{N_T}^*$

categorizes parameters as either scalars or vectors and determines the type of each parameter (e.g., integer, float, boolean, categorical). The output is a parameter set $\Theta = \{\theta_1, \theta_2, ..., \theta_{N_p}\}$, where each θ represents a parameter with its associated information. This process mirrors the human cognitive ability of selective attention and pattern recognition, where experts rapidly identify and categorize relevant information from complex scenarios.

3.3.2 Formalization Thinking

The Formalization Thinking executes deep analytical thinking to construct mathematical models and constraint conditions. The critical steps in this agent involve defining variables, formulating constraints, and constructing the objective function. This component emulates the human brain's abstract reasoning capabilities, where domain experts mentally translate real-world situations into symbolic representations through conceptual abstraction and relationship mapping.

3.3.3 Executive Compiler

The Executive Compiler transforms abstract models into executable code snippets S, similar to the operationalization process of brain executive functions. This transformation reflects the cognitive process of implementation planning, where the human brain converts abstract intentions into concrete action sequences with precise operational details.

3.3.4 System 2 Reasoner

System 2 reasoner provides oversight, while deliberate verification employs counterfactual reasoning to test solutions by asking "what if" questions. While conventional approaches verify solutions by checking constraints directly, ORMind asks "what constraints need to modify would make this solution optimal?" - essentially learning from hypothetical scenarios to identify potential flaws. This approach mirrors human experts who often validate complex solutions by considering what would need to change for an alternative answer to be correct, enabling more robust error detection than direct verification alone. The approach also involves Syntax Error Analysis. In cases where code execution fails due to syntax errors, the specialist pinpoints the problematic line and communicates the probable cause to the Metacognitive Supervisor for swift resolution.

A core innovation in ORMind is the use of counterfactual reasoning for error identification and solution refinement. Assume that the optimization problem can be described by a structural causal model (SCM) with variables X, Y, and C, where:

$$Y = f_Y(X, U), \tag{5}$$

$$C = f_C(X, Y, U), \tag{6}$$

and U denotes latent (exogenous) variables. In our framework, X represents decision variables (e.g., production quantities), Y represents the objective function value (e.g., total cost or profit), and C encapsulates the business constraints.

Inspired by dual-process theories in cognitive science, ORMind divides the reasoning into an intuitive (System 1) phase and a deliberate, analytical (System 2) phase.

For example, given a solution $S_t = \{obj = 150, var_1 = 30, var_2 = 20\}$, the System 2 Reasoner might reason:

$$c_1(S_t) : 2var_1 + 3var_2 \leq 100$$
$$c_2(S_t) : var_1 + var_2 \leq 35$$

Using Python tools to assist its reasoning, the agent might determine:

$$R_t = \begin{cases} \text{“Modify to: } 2var_1 + 3var_2 \leq 130\text{”} & \text{for } c_1 \\ \text{“Modify to: } var_1 + var_2 \leq 50\text{”} & \text{for } c_2 \end{cases}$$

This approach allows the agent to think through which conditions should be altered to make the given result valid, mimicking the cognitive process of a human expert.

3.3.5 Metacognitive Supervisor

The Metacognitive Supervisor mirrors human metacognition—enabling self-awareness of solution quality, strategic oversight, and adaptive decision-making when errors are detected. It monitors the entire solution generation process, making high-level decision adjustments:

$$F_t = \text{Supervisor}_{\text{forward}}(D_t, \Theta_t, M_t, C_t)$$

When constraint violations are detected in production scenarios:

$$F'_t = \text{Supervisor}_{\text{backward}}(S_t, R_t)$$

where R_t contains business-critical constraint failure details. The Supervisor uses this intelligence to prioritize adjustments for maximum operational impact.

Once all business constraints are satisfied:

$$S^*_t = \text{Run}(F'_t)$$

This production-ready state S^*_t represents a deployment-vetted solution meeting all business requirements and optimization targets.

4 Enterprise Application

Lenovo is piloting this innovative approach within its AI Assistant system. The assistant leverages customer computing requirements and budget constraints to formulate mathematical models that optimize the performance-to-cost ratio. Beyond product configuration, Lenovo's AI Assistant extends this optimization capability throughout the customer journey: it streamlines pre-sale product recommendations to shorten decision cycles, automatically applies maximum discounts during purchases to optimize the ordering process, and efficiently handles post-sale services.

At the same time, ORMind is undergoing internal evaluation to enhance product configurations across 292 product categories comprising more than 8,000 potential SKUs (with approximately 2,000+ active SKUs available for recommendation due to business rules requiring in-stock and direct sales items). During testing, the system handled an average of 3,000+ customer inquiries per day, maintaining configuration time below 6 seconds and achieving task completion rates exceeding 80%. Internal assessment tracked additional metrics: intent recognition accuracy reached 85%+, recommendation adoption rate (CTR) was 18%+, and average customer satisfaction score was 4.2 out of 5. Business analysts found the system's transparent reasoning aligned with their own, enabling quick validation and intervention.

5 Experiments

5.1 Datasets

To compare our method, we utilized two datasets:

1. **NL4Opt**: This dataset, collected from the NL4Opt competition[1] at NeurIPS 2022, contains 1101 elementary-level linear programming (LP) problems. It is divided into 713 training samples, 99 validation samples, and 289 test samples.

2. **ComplexOR**: This dataset contains 37 actual industrial optimization problems with the complex constraints and business requirements that characterize real-world applications. Each problem mirrors complex decision-making challenges under various business conditions.

5.2 Experiment Setup and Metrics

We used GPT-3.5-turbo (OpenAI, 2022) as our default large language model, with a temperature of 0. Our experimental framework is built upon LangChain[2], an open-source library designed to facilitate the development of applications powered by language models. We extend the implementation of ORMind to other backbones, including GPT-4o-mini and GPT-4 (OpenAI, 2023).

Our evaluation employs metrics that assess both the correctness and executability of solutions against practical requirements:

Success Rate (SR): The success rate in solving problems.

Model Formulation Failure Rate (MFFR): The percentage of optimization problems where the system fails to formulate a valid mathematical model due to constraint interpretation errors.

[1] https://nl4opt.github.io/

[2] https://www.langchain.com/

Method	NL4Opt			ComplexOR		
	SR↑	MFFR↓	IEFR↓	SR↑	MFFR↓	IEFR↓
tag-BART (Gangwar, 2022)	47.9%	-	-	0%	-	-
OptiMUS (AhmadiTeshnizi et al., 2024)	28.6%	4.0%	11.9%	9.5%	7.9%	15.0%
Chain-of-Thought (Wei et al., 2022)	45.8%	20.5%	9.4%	0.5%	35.3%	8.6%
Progressive Hint (Zheng et al., 2023)	42.1%	19.4%	10.3%	2.2%	35.1%	13.5%
Tree-of-Thought (Yao et al., 2024)	47.3%	17.4%	9.7%	4.9%	31.4%	7.6%
Graph-of-Thought (Besta et al., 2024)	48.0%	16.9%	9.1%	4.3%	32.4%	8.1%
ReAct (Yao et al., 2023)	48.5%	15.5%	11.2%	14.6%	31.9%	10.8%
Reflexion (Shinn et al., 2023)	50.7%	7.3%	9.0%	13.5%	12.9%	10.1%
Solo Performance (Wang et al., 2024)	46.8%	17.9%	13.6%	7.0%	46.5%	13.5%
Chain-of-Experts (Xiao et al., 2024)	58.9%	3.8%	7.7%	25.9%	7.6%	6.4%
ORMind	68.8%	0.4%	2.0%	40.5%	5.4%	21.6%

Table 1: Comparison with baselines on Nl4Opt and ComplexOR.

Method	NL4Opt			ComplexOR		
	SR↑	MFFR↓	IEFR↓	SR↑	MFFR↓	IEFR↓
ORMind (Full)	68.8%	0.4%	2.0%	40.5%	5.4%	21.6%
w/ Conductor	63.2%	0.4 %	1.4%	40.5 %	2.7%	16.2%
w/ Terminology Interpreter	64.9%	0.4%	2.4%	29.7%	5.4%	29.7%
w/ Code Reviewer	33.0%	0.4%	6.6%	32.4%	0.0%	35.1%
w/o Semantic Encoder	58.0%	1.0%	6.9%	32.4%	5.4%	24.3%
w/o Formalization Thinking	65.6%	1.4%	7.2%	35.1%	2.7%	32.4%
w/o Counterfactual Analysis	59.4%	2.8%	11.1%	32.4%	10.8%	24.3%
w/o Syntax Error Analysis	62.2%	1.0%	8.3%	35.1%	5.4%	29.7%
w/o All modules	42.4%	18.1%	13.2%	0.5%	36.8%	8.6%

Table 2: Ablation Study of ORMind.

Implementation Execution Failure Rate (IEFR): The percentage of optimization models that fail during solver execution due to technical incompatibilities or resource limitations.

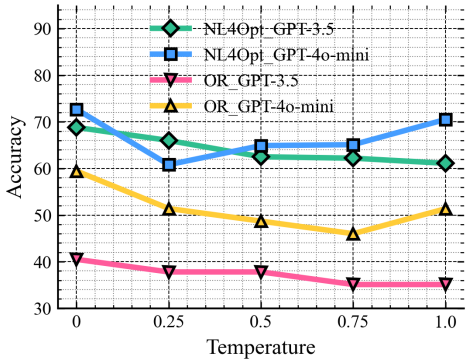


Figure 3: Temperature analysis on NL4Opt and ComplexOR

5.3 Baseline Comparison

In contrast, ORMind's more structured, human-inspired workflow provides a clearer and more effective problem-solving strategy, highlighting its advantages in tackling complex operational research challenges. We benchmark against traditional optimization solutions, including Tag-BART (Gangwar, 2022), and standard LLM frameworks: Chain-of-Thought (Wei et al., 2022), Progressive Hint (Zheng et al., 2023), Tree-of-Thought (Yao et al., 2024), Graph-of-Thought (Besta et al., 2024), ReAct (Yao et al., 2023), Reflexion (Shinn et al., 2023), Solo Performance Prompting (Wang et al., 2024), CoE (Xiao et al., 2024) and OptiMUS (AhmadiTeshnizi et al., 2024).

5.4 Performance Evaluation

Our evaluation reveals critical limitations in existing approaches. Tag-BART (Gangwar, 2022) completely failed on ComplexOR's complex scenarios, while Reflexion (Shinn et al., 2023) showed moderate error-handling capabilities. However, when tackling the more intricate ComplexOR problems, ReAct's performance (Yao et al., 2023) slightly surpassed Reflexion, likely due to its advantage in accessing external knowledge bases, underscoring the importance of external data in handling complex scenarios. The results for OptiMUS are cited from their original paper. They suffer significant performance degradation when tested on GPT-3.5 due to counterintuitive workflow structures that deviate from established problem-solving methodologies(AhmadiTeshnizi et al., 2024). In practice, we found that the sequence in which agents are invoked in these frameworks often appeared counterintuitive and failed to reflect the natural problem-solving process of human experts.

The performance disparity between NL4Opt and ComplexOR datasets highlights a key finding: ORMind excels at accurately formulating mathematical models (achieving near-zero MFFR on NL4Opt), while implementation challenges emerge in more complex industrial scenarios (higher IEFR on ComplexOR). This pattern suggests that future improvements should focus on enhancing the robustness of code generation for complex constraint structures rather than model formulation accuracy.

5.5 Ablation Study

5.5.1 Parameter Sensitivity Analysis

As shown in Figure 3, we evaluated the effect of temperature on GPT-3.5 and GPT-4o-mini models. Lower temperature values led to better performance across both models, suggesting that more deterministic expert responses are beneficial.

Method	GPT-4	
	NL4Opt	ComplexOR
Standard	47.3%	4.9%
Reflexion	53.0%	16.8%
Chain-of-Experts	64.2%	31.4%
OptiMUS	78.8%	66.7%
ORMind	79.9%	62.2%

Table 3: Robustness of ORMind under Different Large Language Models.

5.5.2 Impact of Various Components.

Table 2 quantifies each component's contribution to ORMind's performance across industry-relevant datasets. Ablation studies show that removing Semantic Encoder or Formalization Thinking significantly reduces solution quality, highlighting their importance for enterprise problem structuring. The System 2 Reasoner proves essential for production systems, with its partial function removal causing 6-9% performance degradation.

Adding a Conductor for agent selection increased operational complexity without improving performance, as our streamlined approach proved more cost-efficient. Introducing a Terminology Interpreter decreased performance by 3-5%, suggesting additional interpretation layers create unnecessary overhead. Similarly, Code Reviewer caused hallucinations in large language models, incorrectly modifying appropriately functioning code.

5.5.3 Method Robustness

Table 3 demonstrates ORMind's reliability with GPT-4 as the foundation model. The consistent performance enhancement across metrics confirms that ORMind's architecture effectively leverages advanced LLMs, delivering superior optimization solutions for business operations.

5.5.4 Operational Efficiency

Method	NL4Opt	ComplexOR
CoE	2003 ± 456	3288 ± 780
OptiMUS	2838 ± 822	3241 ± 1194
ORMind	2676 ± 518	3336 ± 997
w/o Reasoner	1539 ± 228	2390 ± 500

Table 4: Comparison of prompt lengths across different datasets for other methods.

ORMind maintains optimal token efficiency across enterprise-scale datasets, reducing computational overhead by streamlining earlier processing stages. Ablation study demonstrates that our system exhibits significant robustness, transparency, and engineering efficiency in industrial scenarios.

6 Conclusion

This paper introduces ORMind, a cognitive-inspired end-to-end reasoning framework, which is being piloted within Lenovo's AI Assistant as part of internal evaluations to enhance optimization capabilities for business. Future work will validate the framework on larger enterprise datasets and refine module coordination to build a stronger theoretical foundation and practical benchmarks for industrial decision systems.

Acknowledgement

This work was supported by the Scientific Research Innovation Capability Support Project for Young Faculty No.ZYGXQNJSKYCXNLZCXM-I28.

Ethics Statement

In developing and deploying the ORMind framework, we have recognized that addressing ethical challenges is crucial for generating fair, transparent, and sustainable outcomes. One of the primary concerns is data bias. To mitigate this risk, we implement rigorous data cleaning and curation processes. Model robustness is another ethical challenge that we address in ORMind. Given the complexity of the multi-agent framework and the heavy reliance on large language models, we recognize that unexpected inputs or adversarial scenarios may lead to instability. As a risk mitigation measure, we have developed a robust error-detection mechanism to catch anomalies and iteratively correct errors.

Limitations

Our model's performance is highly dependent on the input data quality, and even with our robust data cleaning protocols, there is still a risk that residual biases may affect outcomes. Further work is needed to develop automated workflows that periodically audit and adjust data sources, thus reducing this risk over the long term. In terms of robustness, while our multi-agent iterative process allows for continuous refinement, the inherent brittleness of large language models under adversarial conditions poses a challenge. Future improvements will focus on integrating adversarial testing, uncertainty quantification, and more sophisticated error-correction protocols to enhance overall stability. Moreover, the orchestration of multiple agents demands significant computational and memory resources, which may not be feasible in every deployment scenario. To address this issue, we plan to explore model compression, caching techniques, and scalable infrastructure solutions that can dynamically allocate resources based on the current load.

References

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. 2024. Optimus: Scalable optimization modeling with (mi) lp solvers and large language models. In *Forty-first International Conference on Machine Learning*.

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. *arXiv preprint arXiv:2402.00157*.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690.

Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2024. Chateval: Towards better llm-based evaluators through multi-agent debate. In *The Twelfth International Conference on Learning Representations*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168.

Nickvash Kani Neeraj Gangwar. 2022. Tagged input and decode all-at-once strategy.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Minlie Huang, Nan Duan, Weizhu Chen, et al. 2024. Tora: A tool-integrated reasoning agent for mathematical problem solving. In *The Twelfth International Conference on Learning Representations*.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2024. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *Advances in neural information processing systems*, 36.

JiangLong He, Mamatha N, Shiv Vignesh, Deepak Kumar, and Akshay Uppal. 2022. Linear programming word problems formulation using ensemblecrf ner labeler and t5 text generator with data augmentations. *Preprint*, arXiv:2212.14657.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024a. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024b. Metagpt: Meta programming for a multi-agent collaborative framework. In *ICLR*.

Shima Imani, Liang Du, and Harsh Shrivastava. 2023. Mathprompter: Mathematical reasoning using large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 37–42.

Daniel Kahneman. 2011. *Thinking, fast and slow*. macmillan.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

OpenAI. 2022. Introducing chatgpt.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. 2022. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pages 189–203. PMLR.

Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. 2023. Sayplan: Grounding large language models using 3d scene graphs for scalable robot task planning. In *Conference on Robot Learning*, pages 23–72. PMLR.

Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2(5):9.

Keith E Stanovich. 2009. Distinguishing the reflective, algorithmic, and autonomous minds: Is it time for a tri-process theory. *In two minds: Dual processes and beyond*, pages 55–88.

Zhengyang Tang, Chenyu Huang, Xin Zheng, Shixi Hu, Zizhuo Wang, Dongdong Ge, and Benyou Wang. 2024. Orlm: Training large language models for optimization modeling. *arXiv preprint arXiv:2405.17743*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. 2024. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 257–279.

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

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, et al. 2024. Chain-of-experts: When llms meet complex operations research problems. In *The Twelfth International Conference on Learning Representations*.

Zhicheng Yang, Yinya Huang, Wei Shi, Liang Feng, Linqi Song, Yiwei Wang, Xiaodan Liang, and Jing Tang. 2024. Benchmarking llms for optimization modeling and enhancing reasoning via reverse socratic synthesis. *Preprint*, arXiv:2407.09887.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Longhui Yu, Weisen Jiang, Han Shi, YU Jincheng, Zhengying Liu, Yu Zhang, James Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2024a. Metamath: Bootstrap your own mathematical questions for large language models. In *The Twelfth International Conference on Learning Representations*.

Longhui Yu, Weisen Jiang, Han Shi, YU Jincheng, Zhengying Liu, Yu Zhang, James Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2024b. Metamath: Bootstrap your own mathematical questions for large language models. In *The Twelfth International Conference on Learning Representations*.

Chuanyang Zheng, Zhengying Liu, Enze Xie, Zhenguo Li, and Yu Li. 2023. Progressive-hint prompting improves reasoning in large language models. *arXiv preprint arXiv:2304.09797*.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Xinyu Zhu, Junjie Wang, Lin Zhang, Yuxiang Zhang, Yongfeng Huang, Ruyi Gan, Jiaxing Zhang, and Yujiu Yang. 2023a. Solving math word problems via cooperative reasoning induced language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4471–4485.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. 2023b. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*.

Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, et al. 2023. Mindstorms in natural language-based societies of mind. *arXiv preprint arXiv:2305.17066*.

A NL4Opt Case Study

In this section, we provide a detailed case study to illustrate the workings of our proposed ORMind framework. We take a specific problem from the **NL4Opt** dataset and walk through the entire process from problem input to final solution. The case study demonstrates how each agent in the ORMind framework contributes to the final output.

A.1 Problem Description

A pharmacy has 3000 mg of morphine to make painkillers and sleeping pills. Each painkiller pill requires 10 mg of morphine and three units of digestive medicine. Each sleeping pill requires 6 mg of morphine and five units of digestive medicine. The pharmacy needs to make at least 50 painkiller pills. Since sleeping pills are more popular, at least 70% of the pills should be sleeping pills. How many of each should the pharmacy make to minimize the total amount of digestive medicine needed?

A.2 Workflow Execution

Semantic Encoder

```
# Function Signature
output = self.forward_chain.invoke(
    {"problem_example": problem_example,
     "comment_text": comment_pool.get_current_comment_text()}
).content

# Output
{
"painkillers": {"Type": "integer", "Definition": "the number of painkiller pills"},
"sleeping_pills": {"Type": "integer", "Definition": "the number of sleeping pills"}
}
```

Formalization Thinking

```
# Function Signature
output = self.forward_chain.invoke(
{"problem_description": problem_description,
"comments_text": comments_text}
).content

# Output
{
    "VARIABLES": "painkillers: integer, sleeping_pills: integer",
    "CONSTRAINTS": "painkillers >= 50, sleeping_pills >= 0.7 * (painkillers + sleeping_pills), 10*painkillers + 6*sleeping_pills <= 3000",
    "OBJECTIVE": "Minimize 3*painkillers + 5*sleeping_pills"
}
```

Executive Compiler

```
# Function Signature
```

```
output = self.forward_chain.invoke({
"problem_description":problem_description,
"comments_text":comments_text}
).content

# Output
import pulp

def optimize_pharmacy():
    # Create a LP minimization problem
    prob = pulp.LpProblem("Pharmacy Optimization", pulp.LpMinimize)

    # Define decision variables
    painkillers = pulp.LpVariable("painkillers", lowBound=50, cat='Integer')
    sleeping_pills = pulp.LpVariable("sleeping_pills", lowBound=0,
        ↪ cat='Integer')

    # Add constraints
    problem += 10*painkillers_var + 6*sleeping_pills_var <= 3000
    problem += 3*painkillers_var + 5*sleeping_pills_var >=
        ↪ 0.7*(painkillers_var + sleeping_pills_var)

    # Set objective function
    prob += 3*painkillers + 5*sleeping_pills

    # Solve the problem
    prob.solve()

    # Return the optimal values of painkillers and sleeping pills
    return {
        "painkillers": int(painkillers.varValue),
        "sleeping_pills": int(sleeping_pills.varValue)
    }
```

Metacognitive Supervisor forward

```
# Function Signature
answer = self.forward_chain.invoke({
    "comment_text": comment_text,
    "code_example":problem["code_example"],
    "attention":attention
}).content

# Output
from pulp import LpProblem, LpVariable, LpMinimize, lpSum

def prob_123(painkillers, sleeping_pills):
    """
    Args:
        painkillers: an integer, representing the number of painkiller pills
        sleeping_pills: an integer, representing the number of sleeping pills
    Returns:
        amount_of_digestive_medicine: an integer, representing the total
            ↪ amount of digestive medicine needed
    """
    # Create a LP minimization problem
    problem = LpProblem("Pharmacy Optimization Problem", LpMinimize)

    # Define decision variables
    painkillers_var = LpVariable("painkillers", lowBound=50, cat='Integer')
    sleeping_pills_var = LpVariable("sleeping_pills", lowBound=0,
        ↪ cat='Integer')
```

```
    # Add objective function
    problem += 3*painkillers_var + 5*sleeping_pills_var

    # Add constraints
    problem += 10*painkillers_var + 6*sleeping_pills_var <= 3000
    problem += 3*painkillers_var + 5*sleeping_pills_var >=
        ↪ 0.7*(painkillers_var + sleeping_pills_var)

    # Solve the problem
    problem.solve()

    return (problem.objective.value(), int(painkillers_var.varValue),
        ↪ int(sleeping_pills_var.varValue))

    # Running Result:
    (150.0, 50, 0)
```

System 2 Reasoner

```
# Function Signature
answer = self.forward_chain.invoke({
    "problem_description": problem['description'],
    "code_example": code_example,
    "input_content": input_content
}).content

# Output
import math

def counterfactual_solution_analysis(obj, var1, var2):
    """
    Analyze what changes would be necessary for the given solution to be
        ↪ valid and optimal.
    The function variable names must remain obj, var1 and var2. Do not alter
        ↪ these names.
    Args:
        obj: The objective value
        var1: Value of variable 1
        var2: Value of variable 2

    Returns:
        dict: Contains suggested modifications for each constraint and
            ↪ overall assessment
    """
    epsilon = 1e-2
    modifications = {
        "Modification1": {
            "check": lambda: var1 >= 0-epsilon,
            "message": "Adjust constraint to allow var1 to be
                ↪ {:.2f}".format(var1)
        },
        "Modification2": {
            "check": lambda: var2 >= 0-epsilon,
            "message": "Adjust constraint to allow var2 to be
                ↪ {:.2f}".format(var2)
        },
        "Modification3": {
            "check": lambda: 10 * var1 + 6 * var2 <= 3000+epsilon,
            "message": "Modify resource constraint to allow 10*var1 + 6*var2
                ↪ to be {:.2f}".format(10*var1 + 6*var2)
        },
```

```
        "Modification4": {
            "check": lambda: var1 >= 50,
            "message": "Adjust constraint to allow var1 to be at least 50"
        },
        "Modification5": {
            "check": lambda: var2 >= 0.7 * (var1 + var2),
            "message": "Adjust constraint to ensure at least 70% of pills are
                ↪ sleeping pills"
        },
        "Modification6": {
            "check": lambda: math.isclose(var1, round(var1)) and
                ↪ math.isclose(var2, round(var2)),
            "message": "Remove integer constraint on variables"
        },
        "Modification7": {
            "check": lambda: math.isclose(obj, round(obj)),
            "message": "Remove integer constraint on objective"
        }
    }

    results = {}
    all_valid = True

    for name, modification in modifications.items():
        needed = not modification["check"]()
        results[name] = {
            "modification_needed": needed,
            "suggestion": modification["message"] if needed else None
        }
        if needed:
            all_valid = False

    results["solution_valid_without_changes"] = all_valid

    return results

    # Running Result:
    Adjust constraint to ensure at least 70% of pills are sleeping pills
```

Metacognitive Supervisor backward

```
# Function Signature
output = self.backward_chain.invoke({
"problem_description": problem["description"],
    "previous_code": self.previous_code,
    "feedback": feedback_pool.get_closet_comment_text(),
    "attention": self.attention
}).content

# Output
from pulp import LpProblem, LpVariable, LpMinimize, lpSum

def prob_123(painkillers, sleeping_pills):
    """
    Args:
        painkillers: an integer, representing the number of painkiller pills
        sleeping_pills: an integer, representing the number of sleeping pills
    Returns:
        amount_of_digestive_medicine: an integer, representing the total
            ↪ amount of digestive medicine needed
    """
    # Create a LP minimization problem
```

```
    problem = LpProblem("Pharmacy Optimization Problem", LpMinimize)

    # Define decision variables
    painkillers_var = LpVariable("painkillers", lowBound=50, cat='Integer')
    sleeping_pills_var = LpVariable("sleeping_pills", lowBound=0,
        ↪ cat='Integer')

    # Add objective function
    problem += 3*painkillers_var + 5*sleeping_pills_var

    # Add constraints
    problem += 10*painkillers_var + 6*sleeping_pills_var <= 3000
    problem += 3*painkillers_var + 5*sleeping_pills_var >=
        ↪ 0.7*(painkillers_var + sleeping_pills_var)

    # Adjust constraint to ensure at least 70% of pills are sleeping pills
    problem += sleeping_pills_var >= 0.7*(painkillers_var +
        ↪ sleeping_pills_var)

    # Solve the problem
    problem.solve()

    return (problem.objective.value(), int(painkillers_var.varValue),
        ↪ int(sleeping_pills_var.varValue))


    # Running Result:
(735.0, 50, 117)
```

A.3 Discussion of Results

In this case study, we explored how each agent in the ORMind framework contributed to solving the optimization problem of minimizing the total amount of digestive medicine needed to produce painkillers and sleeping pills at a pharmacy.

Initially, the Semantic Encoder correctly identified key variables, such as the number of painkillers and sleeping pills, as integers. The Formalization Thinking then successfully formulated the problem by defining the constraints and the objective function. Specifically, the constraints ensured that at least 50 painkiller pills must be produced and that at least 70% of the pills should be sleeping pills, while the objective was to minimize the use of digestive medicine.

The Programming Expert translated this mathematical model into Python code using the ‘pulp‘ library, ensuring the formulated constraints were implemented correctly. Upon initial solution generation, the Metacognitive Supervisor evaluated the code and returned a solution where only 50 painkiller pills were produced, with no sleeping pills, resulting in a minimal amount of digestive medicine used. However, this solution did not satisfy the 70% requirement for sleeping pills.

The System 2 Reasoner identified this issue through counterfactual analysis and suggested adjusting the constraint to enforce the 70% sleeping pill requirement. After incorporating this feedback, the Metacognitive Supervisor revised the model, leading to a new solution in which 50 painkiller pills and 117 sleeping pills were produced, minimizing the digestive medicine to 735 units.

This iterative process highlights the strength of the ORMind framework in refining solutions through multiple expert agents, each focusing on specific aspects of the problem. By leveraging the System 2 Reasoner’s counterfactual reasoning, the framework was able to correct an oversight in the initial solution, ensuring compliance with all constraints and optimizing the objective function more effectively. This case study demonstrates the framework’s capability to generate solutions and iteratively improve them, thereby achieving a robust and optimal outcome.

B ComplexOR Case Study

In this section, we provide a detailed case study to illustrate the workings of our proposed ORMind framework. We take a specific problem from the **ComplexOR** dataset and walk through the entire process from problem input to the final solution. The case study demonstrates how each agent in the ORMind framework contributes to the final output.

B.1 Problem Description

Consider a production problem. Given a set of products param{ProductNum}. Each product p is produced at a rate of param{ProductionRate_p} tons per hour. There are param{AvailableHours} hours available in a week. The profit per ton for each product p is param{ProfitPerTon_p}. There is a lower limit param{MinimumSale_p} and an upper limit param{MaximumSale_p} on the tons of each product p sold in a week. The problem aims to maximize the total profit from selling all products. It is constrained that the total of hours used by all products may not exceed the hours available. How do we decide the tons of each product p to be produced?

B.2 Workflow Execution

Semantic Encoder

```
output = self.forward_chain.invoke(
    {"problem_example": problem_example,
     "comment_text": comment_pool.get_current_comment_text()}
).content

# Output
{
    "ProductNum": {"Type": "Integer", "Definition": "The number of products"},
    "ProductionRate": {"Type": "Array of floats, shape: ProductNum", "Definition": "The production rate of each product in tons per hour"},
    "ProfitPerTon": {"Type": "Array of floats, shape: ProductNum", "Definition": "The profit per ton for each product"},
    "MinimumSale": {"Type": "Array of floats, shape: ProductNum", "Definition": "The lower limit on the tons of each product sold in a week"},
    "MaximumSale": {"Type": "Array of floats, shape: ProductNum", "Definition": "The upper limit on the tons of each product sold in a week"},
    "AvailableHours": {"Type": "Integer", "Definition": "The total available hours in a week for production"}
}
```

Formalization Thinking

```
output = self.forward_chain.invoke(
{"problem_description": problem_description,
"comments_text": comments_text}
).content

# Output
{
    "VARIABLES": "x_p: Production quantity in tons for product p, where p ranges from 1 to ProductNum",
```

```
    "CONSTRAINTS": "1. x_p >= 0 for all p (Non-negativity constraint)\n2. Sum(p=1 to ProductNum) (x_p / ProductionRate_p) <= AvailableHours (Total production time constraint)\n3. MinimumSale_p <= x_p <= MaximumSale_p for all p (Sales constraints)",
    "OBJECTIVE": "Maximize Sum(p=1 to ProductNum) (ProfitPerTon_p * x_p) (Total profit)"
}
```

Executive Compiler

```
    \begin{lstlisting}
output = self.forward_chain.invoke({
"problem_description":problem_description,
"comments_text":comments_text}
).content

# Output
import pulp

def solve_production_problem(ProductNum, ProductionRate, ProfitPerTon, MinimumSale, MaximumSale, AvailableHours):
    # Create a linear programming problem to maximize profit
    problem = pulp.LpProblem("Maximize_Profit", pulp.LpMaximize)

    # Define decision variables
    x = pulp.LpVariable.dicts("Production_Quantity", range(ProductNum), lowBound=0)

    # Objective Function: Maximize total profit
    profit_contributions = [ProfitPerTon[i] * x[i] for i in range(ProductNum)]
    problem += pulp.lpSum(profit_contributions), "Total_Profit"

    # Constraint: Total production time must not exceed available hours
    time_contributions = [(x[i] / ProductionRate[i]) for i in range(ProductNum)]
    problem += pulp.lpSum(time_contributions) <= AvailableHours, "Total_Production_Time"

    # Constraints: Sales constraints for each product
    for i in range(ProductNum):
        problem += x[i] >= MinimumSale[i], f"Min_Sale_{i}"
        problem += x[i] <= MaximumSale[i], f"Max_Sale_{i}"

    # Solve the problem
    problem.solve()

    # Check if a valid solution has been found
    if pulp.LpStatus[problem.status] == 'Optimal':
        # Extract the solution
        production_quantities = {f'Product_{i+1}': x[i].varValue for i in range(ProductNum)}
        total_profit = pulp.value(problem.objective)
        return production_quantities, total_profit
    else:
        return None, None
```

Metacognitive Supervisor(forward)

```
answer = self.forward_chain.invoke({
    "comment_text": comment_text,
    "code_example":problem["code_example"],
    "attention":attention
}).content

# Output
from pulp import *
import numpy as np

def solve(data):
    ProductNum = data["ProductNum"]
    ProductionRate = np.array(data["ProductionRate"])
    ProfitPerTon = np.array(data["ProfitPerTon"])
    MinimumSale = np.array(data["MinimumSale"])
    MaximumSale = np.array(data["MaximumSale"])
    AvailableHours = data["AvailableHours"]

    prob = LpProblem("Steel3_Problem", LpMaximize)
    tons = LpVariable.dicts("Tons", range(ProductNum), lowBound=0,
        ↪ cat='Continuous')

    # Objective Function: Maximize total profit
    prob += lpSum([ProfitPerTon[i] * tons[i] for i in range(ProductNum)])

    # Constraint: Total production time must not exceed available hours
    prob += lpSum([tons[i] / ProductionRate[i] for i in range(ProductNum)])
        ↪ <= AvailableHours

    # Constraints: Sales constraints for each product
    for i in range(ProductNum):
        prob += tons[i] >= MinimumSale[i]
        prob += tons[i] <= MaximumSale[i]

    prob.solve()

    if prob.status == LpStatusOptimal:
        # Extract the optimized variables as values
        optimized_vars = {
            "tons": {i: tons[i].varValue for i in range(ProductNum)}
        }
        return {
            "status": "Optimal",
            "objective_value": value(prob.objective),
            "optimized_vars": optimized_vars
        }
    else:
        return {"status": LpStatus[prob.status]}

    # Running Result:
TypeError: unsupported operand type(s) for /: 'LpVariable' and 'int'
```

System 2 Reasoner

```
answer = self.forward_chain.invoke({
    "problem_description": problem['description'],
    "code_example": code_example,
    "input_content": input_content
```

```
}).content

# Output
1. The error is caused because the PuLP library does not support division of an `LpVariable` by an integer directly in the constraint formulation.
2. The specific problematic code section is: `prob += lpSum([tons[i] / ProductionRate[i] for i in range(ProductNum)]) <= AvailableHours`.
```

Metacognitive Supervisor(backward)

```
output = self.backward_chain.invoke({
"problem_description": problem["description"],
    "previous_code": self.previous_code,
    "feedback": feedback_pool.get_closet_comment_text(),
    "attention": self.attention
}).content

# Output
from pulp import *

def solve(data):
    ProductNum = data["ProductNum"]
    ProductionRate = data["ProductionRate"]
    ProfitPerTon = data["ProfitPerTon"]
    MinimumSale = data["MinimumSale"]
    MaximumSale = data["MaximumSale"]
    AvailableHours = data["AvailableHours"]

    prob = LpProblem("Steel3_Problem", LpMaximize)
    tons = LpVariable.dicts("Tons", range(ProductNum), lowBound=0, cat='Continuous')

    # Objective Function: Maximize total profit
    prob += lpSum([ProfitPerTon[i] * tons[i] for i in range(ProductNum)])

    # Constraint: Total production time must not exceed available hours
    # Correcting the division issue by multiplying with ProductionRate[i] instead of dividing
    prob += lpSum([tons[i] * (1 / ProductionRate[i]) for i in range(ProductNum)]) <= AvailableHours

    # Constraints: Sales constraints for each product
    for i in range(ProductNum):
        prob += tons[i] >= MinimumSale[i]
        prob += tons[i] <= MaximumSale[i]

    prob.solve()

    if prob.status == LpStatusOptimal:
        # Extract the optimized variables as values
        optimized_vars = {
            "tons": {i: tons[i].varValue for i in range(ProductNum)}
        }
        return {
            "status": "Optimal",
            "objective_value": value(prob.objective),
            "optimized_vars": optimized_vars
        }
    else:
        return {"status": LpStatus[prob.status]}

```

```
49
50      # Running Result:
51  {'status': 'Optimal', 'objective_value': 194828.5706, 'optimized_vars':
        ↪ {'tons': {0: 6000.0, 1: 500.0, 2: 1028.5714}}}
```

B.3 Discussion of Results

The process began with the Semantic Encoder correctly identifying the key variables, such as the number of products, production rates, profits per ton, and sales constraints. These parameters were crucial in formulating the problem accurately.

Next, the Formalization Thinking constructed the mathematical model by defining the decision variables and the constraints. The objective function was set to maximize the total profit. At the same time, the constraints ensured that the total production time did not exceed the available hours and that the production quantities stayed within the specified sales limits.

The Programming Expert then translated this model into Python code using the pulp library. This initial code successfully captured the essence of the problem but encountered a technical issue: the division of LpVariable objects by integers within the constraints, which the pulp library does not directly support.

The System 2 Reasoner identified this issue and provided specific feedback, pinpointing the problematic code and the nature of the error. This feedback was crucial in guiding the Metacognitive Supervisor's subsequent code revision.

The Metacognitive Supervisor corrected the division issue by multiplying instead of dividing the variables within the constraint formulation. This adjustment ensured that the constraints were correctly implemented and allowed the model to be solved without errors.

Finally, the revised model was solved, yielding an optimal solution where the production quantities and total profit were maximized while adhering to all constraints. The solution indicated optimal production quantities for each product and a corresponding total profit, demonstrating the effectiveness of the ORMind framework.

C Prompt Templates for Agents

Below, we list the prompt templates used for each agent in the ORMind framework. These templates are crucial for guiding the LLMs in performing their respective tasks.

Semantic Encoder

```
1
2   # Prompt Template:
3   Please review the following example and extract the parameters along with
        ↪ their concise definitions:
4   {problem_example}
5   The comment from your colleague is:
6   {comment_text}
7   Your output should be in JSON format as follows:
8   {{
9       "Parameter1": {{"Type": "The parameter's data type or shape",
            ↪ "Definition": "A brief definition of the parameter"}},
10      "Parameter2": {{"Type": "The parameter's data type or shape",
            ↪ "Definition": "A brief definition of the parameter"}},
11      ...
12  }}
13  Provide only the requested JSON output without any additional information.
```

Formalization Thinking

```
# Prompt Template:
Now the origin problem is as follows:
{problem_description}
You can use the parameters information from your colleague:
{comments_text}
The order of given parameters is random. You should clarify the meaning of each parameter to choose proper parameter to construct constraint.
Give your Mathematical model of this problem.
Your output format should be a JSON like this:
{{
    "VARIABLES": "A concise description about variables and its shape or type",
    "CONSTRAINTS": "A mathematical Formula about constraints",
    "OBJECTIVE": "A mathematical Formula about objective"
}}
Don't give any other information.
```

Executive Compiler

```
# Prompt Template:
You are presented with a specific problem and tasked with developing an efficient Python program to solve it.
The original problem is as follows:
{problem_description}
Your colleague has constructed a mathematical model for reference:
{comments_text}
Please note that this model may contain errors and is used as a reference.
You can analyze the problem step by step and provide your own code.
Requirements:
1. Use the PuLP library for implementation.
2. Provide a function that solves the problem.
3. Do not include code usage examples or specific variable values.
4. Focus on creating a general, reusable solution.
```

System 2 Reasoner

```
# Prompt Template:
Analyze the following optimization problem:
{problem_description}

Task: Write a Python function that identifies which specific constraints or conditions in the given problem are not satisfied. This condition will need modification to achieve a valid and optimal solution.

Function specifications:
- Input arguments and their types: {input_content}
- Adhere to the given data types.
- Reference this code structure: {code_example}
- Import the necessary libraries.
```

```

Notes:
The code example is only for reference in terms of format and structure. Generate code specifically for the given problem, not based on any examples.
All specific constraints should be determined based on the problem description provided.
Make sure to include checks for all constraints mentioned in the problem description. Don't give any Example usages.
```

Metacognitive Supervisor(backward)

```

# Prompt Template:
   FORWARD_TASK: Your colleague Executive Compiler has given his answer:
{comment_text}
This answer has not been formatted. You need to format the code as the example.
The final code must has the same input args and function name as the code example:
{code_example}
You also need to return the optimized variables.
Important: Your final code should strictly use same input args, function name and return style of the code example exactly.
{attention}
Don't forget to import the library. Don't give any example usage.

   BACKWARD_TASK: In your previous answer may have errors, you receive feedback about the error.
The feedback is generated by counterfactual reasoning,
which means that the feedback does not represent actual changes that need to be made to the problem conditions.
the feedback highlights where your code may have misinterpreted the original conditions.
{feedback}

For example, If you receive a message like 'Remove integer constraint on variables',
it means that your previous answer is correct only when the integer constraint is removed.
This strongly suggests that your earlier solution likely overlooked the integer constraint.
You need to add the constraint.
If you receive a message like 'Modify resource constraint to allow...',
it means that your previous answer is correct only when this constraint is modified.
This strongly suggests that your earlier solution likely has error in this constraint.
You need to doublecheck your previous code corresponding to the feedback and fix the error.

Carefully review the feedback and give the final code as the format of your previous code.
{attention}

The original problem description remains unchanged:
{problem_description}

Your previous code for analyzing the solution was:
{previous_code}

```

```
Your task is to carefully review the original problem description and the counterfactual feedback.

Remember:
Provide your corrected code in the same format as your previous code.
Do not give any example or explanation.
If the feedback is not existed in the description, you may directly use the original code.
Use "from PuLP import *" to import the library as the example.
```

Conductor

```

# Prompt Template:
Now, you are presented with an operational optimization-related problem:
{problem_description}
In this multi-expert system, there are many agent_team, each of whom is responsible for solving part of the problem.
Your task is to CHOOSE THE NEXT EXPERT TO CONSULT.
The names of the agent_team and their capabilities are listed below:
{experts_info}
Experts that have already been commented include:
{commented_experts}
Please select an expert to consult from the remaining expert names {remaining_experts}.
Please note that the maximum number of asked agent_team is {max_collaborate_nums}, and you can ask {remaining_collaborate_nums} more times.
You should output the name of expert directly. The next expert is:'''
```

Terminology Interpreter

```

# Prompt Template:
As a domain knowledge terminology interpreter, your role is to provide additional information and insights related to the problem domain.
Here are some relevant background knowledge about this problem: {knowledge}.

You can contribute by sharing your expertise, explaining relevant concepts, and offering suggestions to improve the problem understanding and formulation.
Please provide your input based on the given problem description:
{problem_description}

Your output format should be a JSON like this (choose at most 3 hardest terminology. Please provide your output, ensuring there is no additional text or formatting markers like ```json. The output should be in plain JSON format, directly parsable by json.loads(output).):
[
  {{
    "terminology": "...",
    "interpretation": "..."
  }}
]
```

Code Reviewer

```

# Prompt Template:
As a Code Reviewer, your responsibility is to conduct thorough reviews of implemented code related to optimization problems.
You will identify possible errors, inefficiencies, or areas for improvement in the code, ensuring that it adheres to best practices and delivers optimal results. Now, here is the problem:
{problem_description}.

You are supposed to refer to the codes given by your colleagues from other aspects: {comments_text}
```

D Code Example

The following are code examples used by the ORMind framework for the Counterfactual Analysis.

NL4Opt Code Example for Counterfactual Analysis

```python
import math

def counterfactual_solution_analysis(obj, var1, var2):
    """
    Analyze what changes would be necessary for the given solution to be valid and optimal.
    The function variable names must remain obj, var1 and var2. Do not alter these names.
    Args:
        obj: The objective value
        var1: Value of variable 1
        var2: Value of variable 2

    Returns:
        dict: Contains suggested modifications for each constraint and overall assessment
    """
    epsilon = 1e-2
    modifications = {
        "Modification1": {
            "check": lambda: var1 >= 0-epsilon,
            "message": "Adjust constraint to allow var1 to be {:.2f}".format(var1)
        },
        "Modification2": {
            "check": lambda: var2 >= 0-epsilon,
            "message": "Adjust constraint to allow var2 to be {:.2f}".format(var2)
        },
        "Modification3": {
            "check": lambda: 2 * var1 + 3 * var2 <= 100+epsilon,
            "message": "Modify resource constraint to allow 2*var1 + 3*var2 to be {:.2f}".format(2*var1 + 3*var2)
        },
        "Modification4": {
            "check": lambda: var1 + var2 <= 35+epsilon,
            "message": "Adjust daily production limit to allow var1 + var2 to be {:.2f}".format(var1 + var2)
        },
        "Modification5": {
```

```
            "check": lambda: math.isclose(var1, round(var1)) and
                ↪ math.isclose(var2, round(var2)),
            "message": "Remove integer constraint on variables"
        },
        "Modification6": {
            "check": lambda: math.isclose(obj, round(obj)),
            "message": "Remove integer constraint on objective"
        }
    }

    results = {}
    all_valid = True

    for name, modification in modifications.items():
        needed = not modification["check"]()
        results[name] = {
            "modification_needed": needed,
            "suggestion": modification["message"] if needed else None
        }
        if needed:
            all_valid = False

    results["solution_valid_without_changes"] = all_valid

    return results
```

ComplexOR Code Example for Counterfactual Analysis

```
import numpy as np

def counterfactual_solution_analysis(alloys_used, data):
    """
    Analyze what changes would be necessary for the given solution to be
        ↪ valid and optimal.

    Returns:
        dict: Contains suggested modifications for each constraint and
            ↪ overall assessment
    """
    AlloysOnMarket = data["AlloysOnMarket"]
    RequiredElements = data["RequiredElements"]
    CompositionDataPercentage = np.array(data["CompositionDataPercentage"])
    DesiredBlendPercentage = np.array(data["DesiredBlendPercentage"])
    AlloyPrice = np.array(data["AlloyPrice"])

    alloys_used_array = np.array([alloys_used[a] for a in
        ↪ range(AlloysOnMarket)])

    modifications = {
        "Modification1": {
            "check": lambda: all(alloys_used_array >= 0),
            "message": "Adjust non-negativity constraint to allow negative
                ↪ quantities: {}".format(alloys_used_array)
        },
        "Modification2": {
            "check": lambda: all(np.dot(CompositionDataPercentage,
                ↪ alloys_used_array) >= DesiredBlendPercentage *
                ↪ np.sum(alloys_used_array)),
            "message": "Modify desired blend percentages to:
                ↪ {}".format(np.dot(CompositionDataPercentage,
                ↪ alloys_used_array) / np.sum(alloys_used_array))
        },
```

```
        "Modification3": {
            "check": lambda: all(alloys_used_array <= 1),
            "message": "Increase market availability to allow quantities:
                ↪ {}".format(alloys_used_array)
        }
    }

    results = {}
    all_valid = True

    for name, modification in modifications.items():
        needed = not modification["check"]()
        results[name] = {
            "modification_needed": needed,
            "suggestion": modification["message"] if needed else None
        }
        if needed:
            all_valid = False

    results["solution_valid_without_changes"] = all_valid

    return results
```

E Hardware and Software Configurations

E.1 Software

The software environment used in the experiments includes: - **Operating System**: Windows11 - **Python**: 3.10 - **LangChain**: 0.2.7 - **LangChain-Community**: 0.2.7 - **NumPy**: 1.23.5 - **Tqdm**: 4.62.3 - **PuLP**: 2.8.0 - **OpenAI API Key**: Required for accessing OpenAI's models

F Data Format Example

Formatted NL4OPT data in JSON format

```


"description":A fishery wants to transport their catch. They can either use
    ↪ local sled dogs or trucks. Local sled dogs and trucks can take
    ↪ different amount of fish per trip. Also, the cost per trip for sled
    ↪ dogs and truck is also differs. You should note that the budget has
    ↪ an upper limit and the number of sled dog trips must be less than
    ↪ the number of truck trips. Formulate an LP to maximize the number of
    ↪ fish that can be transported.
[
    {
        "input": {
            "DogCapability": 100,
            "TruckCapability": 300,
            "DogCost": 50,
            "TruckCost": 100,
            "MaxBudget": 1000
        },
        "output": [
            3000
        ]
    }
]
```

Formatted ComplexOR data in JSON format


```
{
    "description": "The Aircraft Assignment Problem is a mathematical
        ↪ programming model that aims to assign \\param{TotalAircraft}
        ↪ aircraft to \\param{TotalRoutes} routes in order to minimize the
        ↪ total cost while satisfying availability and demand constraints.
        ↪ The availability for each aircraft i is \\param{Availability_i}
        ↪ and it represents the maximum number of routes that the aircraft
        ↪ can be assigned to. The demand for each route j is
        ↪ \\param{Demand_j} and it denotes the number of aircraft required
        ↪ to fulfill the passenger or cargo needs of the route. The
        ↪ capability of each aircraft i for each route j is given by
        ↪ \\param{Capacity_{i,j}} and it defines whether the aircraft can
        ↪ service the route, considering factors such as range, size, and
        ↪ suitability. Finally, \\param{Cost_{i,j}} represents the cost of
        ↪ assigning aircraft i to route j, which includes operational,
        ↪ fuel, and potential opportunity costs.",
    "parameters": [
        {
            "symbol": "TotalAircraft",
            "definition": "The total number of aircraft available for
                ↪ assignment",
            "shape": []
        },
        {
            "symbol": "TotalRoutes",
            "definition": "The total number of routes that require aircraft
                ↪ assignment",
            "shape": []
        },
        {
            "symbol": "Availability",
            "definition": "The availability of each aircraft, indicating the
                ↪ maximum number of routes it can be assigned to",
            "shape": [
                "TotalAircraft"
            ]
        },
        {
            "symbol": "Demand",
            "definition": "The demand for each route, indicating the number of
                ↪ aircraft required",
            "shape": [
                "TotalRoutes"
            ]
        },
        {
            "symbol": "Capacity",
            "definition": "The capacity matrix defining the suitability and
                ↪ capability of each aircraft for each route",
            "shape": [
                "TotalAircraft",
                "TotalRoutes"
            ]
        },
        {
            "symbol": "Costs",
            "definition": "The cost matrix representing the cost of assigning
                ↪ each aircraft to each route",
            "shape": [
                "TotalAircraft",
                "TotalRoutes"
            ]
        }
```

```
    ]
}



[
{
    "TotalAircraft": 5,
    "TotalRoutes": 5,
    "Availability": [10, 19, 25, 15, 0],
    "Demand": [250, 120, 180, 90, 600],
    "Capacity": [
        [16, 15, 28, 23, 81],
        [0, 10, 14, 15, 57],
        [0, 5, 0, 7, 29],
        [9, 11, 22, 17, 55],
        [1, 1, 1, 1, 1]
    ],
    "Costs": [
        [17, 5, 18, 17, 7],
        [15, 20, 9, 5, 18],
        [15, 13, 8, 5, 19],
        [13, 14, 6, 16, 8],
        [13, 14, 14, 10, 7]
    ]
},
    "output": [
        "Infeasible"
        ]
    }
]
```

G Agent-Memory Pool Interaction in ORMind

The Memory Pool in *ORMind* functions as a centralized repository that supports the collaboration and coordination of agents during the reasoning process. It stores and provides access to shared data, ensuring consistency and efficiency in solving complex operations research (OR) problems.

Agents interact with the Memory Pool primarily through retrieval and update. Before performing a task, an agent retrieves relevant information from the Memory Pool, such as the current problem state, previously identified variables and constraints, and intermediate results from earlier reasoning steps. This ensures that all agents operate with access to the most up-to-date context, avoiding redundant computations and inconsistencies.

Once an agent completes a task, it updates the Memory Pool with its results. These updates include newly discovered variables, constraints, other task-specific outputs, and annotations summarizing the reasoning process. Every update is tagged with metadata, such as the agent's identifier and a timestamp, to maintain traceability and facilitate debugging.

The Memory Pool also plays a critical role in the iterative refinement process. As new information becomes available, earlier results can be revisited and improved by subsequent agents, allowing for modular and adaptive problem-solving. This centralized structure ensures that the system's collective progress is reflected in a single shared repository, enabling efficient and coherent reasoning across all agents.

The Memory Pool enhances the ORMind framework's ability to tackle complex OR problems by providing a shared, structured, and continuously updated context. It promotes collaboration, reduces redundancy, and ensures that agents work synchronized and context-awarely.

H Comparison with Other Planning with Feedback Methods

While our methodology adopts a multi-expert framework, it distinguishes itself through two unique features: human problem-solving process and counterfactual reasoning. These features enable a more structured and iterative problem-solving process compared to other approaches.

The table 5 highlights the differences between our approach and other methods regarding key functionalities such as multi-agent frameworks, industry-focused processes, external knowledge access, and feedback refinement.

Method	Multi-agents	Industry-Focused	External Knowledge	Feedback Refinement
ReAct(Yao et al., 2023)	✗	✗	✓	✗
Voyager(Wang et al., 2023)	✗	✗	✓	✓
Ghost(Zhu et al., 2023b)	✗	✗	✓	✓
SayPlan(Rana et al., 2023)	✗	✗	✓	✓
MetaGPT(Hong et al., 2024b)	✓	✗	✓	✗
NLSOM(Zhuge et al., 2023)	✓	✗	✓	✗
SSP(Wang et al., 2024)	✓	✗	✗	✗
ChatEval(Chan et al., 2024)	✓	✗	✗	✗
ORMind	✓	✓	✗	✓

Table 5: Comparison of ORMind with existing planning and feedback-based methods.

I Long-term Research Value and Future Directions

This work establishes a foundation for advanced reasoning systems in operations research with implications far beyond the current implementation. Below, we analyze the key long-term research values and potential future directions:

I.1 Counterfactual Reasoning as a Fundamental AI Capability

The counterfactual reasoning approach introduced in ORMind represents a fundamental advancement in how AI systems can validate and refine solutions. By reasoning about what constraints would need to change for a given solution to be valid, our approach begins to bridge the gap between correlation and causation in AI reasoning systems. This opens avenues for more sophisticated causal reasoning frameworks that can identify patterns and underlying causal mechanisms. Beyond operations research, this methodology could fundamentally transform how AI systems approach problem validation and solution refinement across domains ranging from scientific discovery to medical diagnosis. The ability to perform "what-if" analyses on potential solutions provides a form of self-verification that increases solution reliability without requiring explicit programming of all edge cases, a crucial advancement for mission-critical enterprise applications.

I.2 Cognitive Architectures for Complex Decision Making

ORMind's cognitively-inspired framework mirrors human expert reasoning processes and offers a blueprint for next-generation business intelligence systems. The sequential decomposition of complex problems into stages of understanding, formulation, and refinement provides a generalizable architecture that could be applied to various reasoning tasks beyond optimization. This represents a significant shift from current approaches that often rely on monolithic models or rigid predefined workflows. Future research could explore how such cognitive architectures can dynamically adapt their reasoning strategies based on problem characteristics, incorporate domain-specific knowledge while preserving flexible reasoning, and create natural interaction points for human-AI collaboration. The emergence of such cognitively-aligned systems could fundamentally transform how organizations approach complex decision-making, enabling more intuitive, explainable, and effective enterprise AI solutions.