ORFlow: Unveiling Structural Topology Effects in Multi-Agent Optimization Problems Solving

Anonymous ACL submission

Abstract

Multi-agent systems powered by large language models (LLMs) demonstrate strong potential for automating operations research (OR). While various frameworks propose distinct workflow topologies to translate natural language into executable solver code, current systems tightly couple their architectures with specific engineering choices, obscuring which topological structures actually drive performance. To address this gap, we propose ORFlow, a framework that reduces prompt-design confounding through standardized prompting, enumerates workflow topologies, and attributes architectural effects. Specifically, ORFlow introduces: (1) a consensus-based prompt standardization method that derives neutral templates via cross-system semantic similarity; (2) a topology design space based on Directed Acyclic Graph (DAG) that models the unidirectional OR pipeline, generating 12 distinct architectures via three graph-theoretic operations; and (3) an attribution method using cooperative game theory (Shapley values) to quantify each operation's marginal contribution. Controlled experiments across four benchmarks and three LLMs systematically identify the topological structures that enhance OR solving. Furthermore, Shapley attribution offers a quantitative explanation for these performance variations, uncovering how inter-phase dependencies shape overall system performance. Code is available at https://anonymous.4open.science/r/ORFlow-C2BF.

1 Introduction

Operations research (OR) optimization is widely applied across domains such as supply chain management (Lambert and Cooper, 2000), scheduling (Pinedo and Hadavi, 1992), and logistics planning (Toth and Vigo, 2014), yet solving these problems demands extensive mathematical modeling and programming expertise. Large language models (LLMs) exhibit strong reasoning (Wei et al., 2023; Ahn et al., 2024) and code generation (Chen et al., 2021; Huynh and Lin, 2025) capabilities, motivating their application to OR automation. Following the NL4Opt benchmark for translating problem descriptions into mathematical formulations (Ramamonjison et al., 2023), multi-agent systems like OptiMUS (AhmadiTeshnizi et al., 2023), Chain-of-Experts (Xiao et al., 2024), ORLM (Huang et al., 2025), and ORMind (Wang et al., 2025) now automate the full pipeline from natural language to executable solver code. While sharing a semantic pipeline of component extraction, formulation, and programming, these systems employ distinct workflow topologies, ranging from single-agent end-to-end solvers to multi-stage serial pipelines with varying granularity. Topology shapes inter-agent information flow and task decomposition, introducing architectural factors that prompt refinement alone may not fully address.

However, despite the pivotal role of topology, existing literature lacks a principled understanding of topology effects under controlled prompting, leaving three critical gaps. First, **topology–prompt confounding**: current frameworks couple novel architectures with specific prompts, making it difficult to attribute performance gains to topology alone. Second, **unsystematic topology design**: systems propose custom architectures without a formal framework to systematically enumerate and compare topological variants. Third, **absence of quantitative attribution**: no mechanism exists to quantify the marginal contribution of each structural operation—leaving the superior performance of the optimal architecture structurally unexplained.

To address these gaps, we propose ORFlow, a systematic framework for reducing prompt-design confounding, enumerating workflow topologies, and attributing topology effects in multi-agent OR problem solving. Our key contributions are as fol-

lows:

- **Block-Level Prompt Standardization.** To mitigate the confounding effects of prompt engineering in multi-agent evaluations, we introduce a consensus-based standardization method. Specifically, we use semantic aggregation to convert high-consensus semantics into standard prompt templates, and reserve low-consensus elements as optional skills documents for supplementary experiments. Through this design, ORFlow establishes an objective baseline that facilitates a fairer and more reliable comparison of underlying architectural structures.

- **Workflow-Level Architecture Design Space.** We formalize the multi-agent OR solving pipeline as a Directed Acyclic Graph (DAG). Applying three fundamental graph-theoretic operations (edge contraction, vertex splitting, and component-wise splitting), we provide a complete enumeration within the predefined topology operations, yielding 12 distinct architectures. Furthermore, we leverage cooperative game theory, utilizing Shapley values to rigorously quantify the marginal contribution of each topological transformation.

- **Quantitative Assessment.** Extensive experiments across four OR benchmarks and three LLMs reveal how decomposition granularity affects solving capabilities. In addition to evaluation accuracy, our Shapley-based attribution provides a quantitative, post-hoc diagnostic view. The results suggest that weakly coupled phases may benefit from decomposition, whereas formulation-like strongly coupled phases are less suitable for independent decomposition due to continuity requirements, offering practitioners concrete criteria for architectural design.

2 Related Work

2.1 LLMs for Operations Research

The application of large language models to operations research has grown rapidly, with LLM-based systems automating the full pipeline from natural language problem descriptions to executable solver code. Existing approaches can be broadly categorized into two lines. *Prompting-based approaches* employ multi-agent workflows and prompt strategies to guide LLMs through OR solving without modifying model weights. Representative systems include OptiMUS (AhmadiTeshnizi et al., 2023, 2024b,a), Chain-of-Experts (Xiao et al., 2024), OR-LLM-Agent (Zhang and Luo, 2025), ORMind (Wang et al., 2025), OptiTrust (Lima et al., 2025). *Learning-based approaches* improve LLM capabilities for OR through fine-tuning or reinforcement learning, including ORLM (Huang et al., 2025), LLMOPT (Jiang et al., 2025), MURKA (Xie et al., 2026), OR-R1 (Ding et al., 2026), and AutoOR (Motwani et al., 2026). While each system employs a distinct multi-agent topology, the intrinsic impact of topology on reasoning performance remains under-explored. To address this limitation, we quantify topology effects under standardized prompting, providing practitioners with evidence-based design principles.

2.2 Multi-Agent Collaboration Topology

Research on LLM-based multi-agent systems highlights the significant impact of topology on task performance. Existing role-based frameworks including CAMEL (Li et al., 2023b), AgentVerse (Chen et al., 2024), AutoGen (Wu et al., 2023), MetaGPT (Hong et al., 2024), ChatDev (Qian et al., 2024) conventionally model architectures as Agent Communication Graphs (ACGs) tailored for open-ended dialogues and iterative consensus. However, unlike open-ended tasks, automatic OR problem-solving requires strict deductive execution orders driven by unidirectional semantic transformations from natural language to mathematical formulations and executable code. Thus, we propose modeling multi-agent OR workflows as Directed Acyclic Graphs (DAGs) of task dependencies, which provides a principled basis for comparing architectural decomposition and parallelization.

3 Methodology

3.1 Block-Level Design: Consensus-Based Prompt Standardization

To objectively evaluate topology, we introduce a principled method for constructing standardized prompts. Given that topology and prompt design closely depend on each other, they cannot be entirely isolated. To eliminate *artificial* prompt bias—ensuring performance differences reflect structural effects rather than prompt engineering—we derive each agent's prompt from a unified neutral source: the consensus across authoritative literature. The detailed processing procedure is

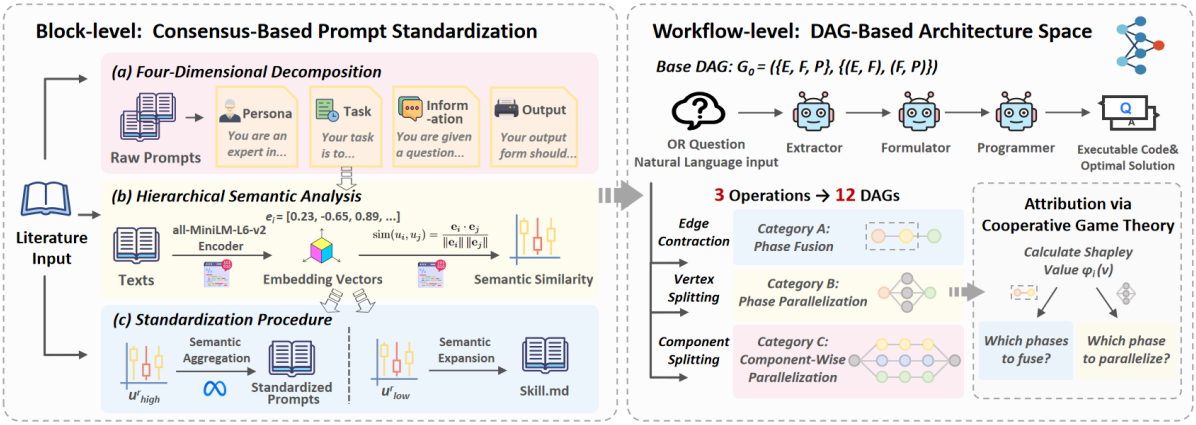

Figure 1: The overall pipeline of the ORFlow framework. At the block-level, it standardizes prompts via semantic consensus, aggregating high-similarity semantic components into prompt templates and expanding divergent ones into skills documents. At the workflow-level, it models the OR pipeline as a DAG, applies graph operations to generate 12 architectures, and quantifies structural contributions via Shapley attribution.

provided in Appendix C.

Prompt Collection and Four-Dimensional Decomposition. We reviewed multi-agent systems from 30 published LLMs-for-OR papers (2023–2026), categorizing agents by function and collecting their prompts from GitHub repositories, papers' main texts, and appendices. These encompass the vast majority of all agent roles required across our 12 architectures; for the remaining roles lacking direct literature—primarily sub-task agents—we manually designed their prompts by adapting those of the relative main-task agents. Following established taxonomies (Wang et al., 2024; Xi et al., 2023), we decomposed each prompt into four dimensions: (1) *Persona*, defining the agent's identity, professional background, and expertise; (2) *Task Instruction*, specifying the concrete actions and reasoning pathways; (3) *Information Interaction*, detailing the input data received by the agent; and (4) *Output Constraints*, dictating the required content and formatting specifications of the response.

Hierarchical Semantic Analysis. We adopt a coarse-to-fine procedure to identify cross-system consensus at the appropriate semantic granularity. Let $\mathcal{P}_d^r = \{p_{d,1}^r, \ldots, p_{d,N}^r\}$ denote the prompt segments collected from N published systems for agent role r in dimension $d \in \mathcal{D}$. All segments are encoded using Sentence-BERT (Reimers and Gurevych, 2019) (all-MiniLM-L6-v2), and pairwise cosine similarity is computed as:

$$\mathrm{sim}(u_i, u_j) = \frac{\mathbf{e}_i \cdot \mathbf{e}_j}{\|\mathbf{e}_i\| \, \|\mathbf{e}_j\|} \tag{1}$$

where $\mathbf{e}_i = \mathrm{SBERT}(u_i)$ denotes the sentence embedding of text segment u_i.

Step 1: Dimension similarity. For each role r and dimension d, we pair all N collected segments and compute the average over all $\binom{N}{2}$ pairwise similarities:

$$\bar{s}_d^r = \frac{1}{\binom{N}{2}} \sum_{1 \le i < j \le N} \mathrm{sim}(p_{d,i}^r, p_{d,j}^r) \tag{2}$$

Each $\bar{s}_d^r$ quantifies the degree of semantic agreement on dimension d across the N systems. While capturing initial community consensus, this measurement may conflate heterogeneous subcomponents within a single dimension, as certain semantic aspects exhibit strong agreement while others diverge. When $\bar{s}_d^r$ is moderate yet underlying pairwise similarities exhibit high variance, a finer-grained analysis is required.

Step 2: Sub-unit decomposition. To resolve this heterogeneity, domain experts manually categorize each dimension into 2–3 semantic subclasses based on OR problem-solving task semantics (see Appendix C.2 for the full taxonomy). Let $\{U_{d,1}^r, \ldots, U_{d,M_d}^r\}$ denote the resulting sub-units for role r in dimension d. We compute $\bar{s}(U_{d,k}^r)$ analogously to Eq. 2 within each sub-unit. At this finer granularity, the similarity distribution tends to exhibit clearer polarization, facilitating a natural separation between high-consensus and low-consensus sub-units.

Standardization Procedure. Let $\mathcal{U}_{\text{high}}^r$ and $\mathcal{U}_{\text{low}}^r$ denote the high-similarity and low-similarity sub-unit groups identified from the bimodal distribution.

We construct the standardized prompt as:

$$\text{Prompt}^r = \text{Concat}_{d\in\mathcal{D}}\Big(\text{Agg}\big(\mathcal{U}^r_{\text{high},d}\big)\Big) \quad (3)$$

where $\text{Agg}(\cdot)$ denotes LLM-based semantic aggregation (Llama-3-8B-Instruct (Grattafiori et al., 2024), temperature=0 to maximize adherence to the original semantics and eliminate stochastic variation) that merges the consensus sub-units within each dimension into a single coherent text, and $\text{Concat}_{d\in\mathcal{D}}(\cdot)$ denotes ordered concatenation of the aggregated dimension-level segments into the final prompt. Sub-units in $\mathcal{U}^r_{\text{low}}$, which capture divergent system-specific strategies, are used as reference material: we adapt and expand their content following a standardized skill document format, producing a tailored *skill module* (SKILL.md) for each agent role. These skill modules serve as optional supplementary instructions that can be loaded at inference time; we later compare performance with and without skills to assess the impact of prompt richness on different topologies.

3.2 Workflow-Level Design: DAG-Based Architecture Space

3.2.1 DAG Formulation for OR Problem Solving

An OR optimization problem can be formally represented as:

$$\min_{\boldsymbol{x}\in\mathcal{X}\subseteq\mathbb{R}^D} f(\boldsymbol{x}), \quad \text{s.t.} \quad G(\boldsymbol{x}) \le \boldsymbol{c} \quad (4)$$

where $\boldsymbol{x} = (x_1, x_2, \ldots, x_D)^\top$ is the D-dimensional decision variable, $\mathcal{X} \subseteq \mathbb{R}^D$ is the feasible domain, $f : \mathcal{X} \to \mathbb{R}$ is the objective function to be minimized, and $G(\boldsymbol{x}) : \mathbb{R}^D \to \mathbb{R}^m$ represents the vector-valued constraint function with upper bounds $\boldsymbol{c} = (c_1, c_2, \ldots, c_m)^\top$.

Given a natural language description D of such a problem, the goal of automated OR solving is to produce executable solver code that computes the optimal solution. This requires a sequence of semantic transformations:

$$D \xrightarrow{\mathcal{E}} \mathcal{K} \xrightarrow{\mathcal{F}} \mathcal{M} \xrightarrow{\mathcal{P}} \textit{Code} \quad (5)$$

where $\mathcal{K}$ denotes the structured natural-language representation, $\mathcal{M}$ denotes the mathematical formulation, and *Code* denotes executable solver code. This process can be decomposed into three phases:

- **Extraction** ($\mathcal{E} : D \to \mathcal{K}$): Identify parameters, decision variables, constraints, and objective from D, producing a structured representation $\mathcal{K} = \{K_V, K_C, K_O, K_{\text{param}}\}$.
- **Formulation** ($\mathcal{F} : \mathcal{K} \to \mathcal{M}$): Translate the extracted components into a rigorous mathematical program $\mathcal{M} = \{M_V, M_C, M_O\}$ with formal variable domains, constraint expressions, and an optimization objective.
- **Programming** ($\mathcal{P} : \mathcal{M} \to \textit{Code}$): Convert the mathematical formulation into executable solver code (e.g., Gurobi Python).

This process exhibits two structural properties that align naturally with Directed Acyclic Graphs (DAGs) : (1) information flows unidirectionally—each phase depends on its predecessor's output but never cycles back; and (2) within each phase, semantically independent components (variables, constraints, objectives) can be processed in parallel. We therefore formalize the pipeline as a DAG $G = (\mathcal{N}, \mathcal{A})$, where nodes $\mathcal{N}$ represent atomic subtasks and arcs $\mathcal{A}$ encode information dependencies. The fundamental three-phase pipeline defines the **Base DAG**:

$$G_0 = \big(\{\mathcal{E}, \mathcal{F}, \mathcal{P}\},\ \{(\mathcal{E}, \mathcal{F}),\ (\mathcal{F}, \mathcal{P})\}\big) \quad (6)$$

which serves as the starting point for all subsequent topological transformations.

3.2.2 Multi-Agent Architecture Taxonomy

Starting from the Base DAG G_0, we apply three graph-theoretic operations to derive 12 variant DAGs, each representing a distinct workflow topology (Table 1).

Category A: Phase Fusion. Phase fusion merges consecutive phases into a single LLM call, reducing pipeline depth and eliminating inter-phase communication overhead at the cost of increased single-agent complexity. This corresponds to **edge contraction** in graph theory (Bondy and Murty, 2008):

$$\text{Contract}(n_i, n_j) = n_i \oplus n_j, \quad (n_i, n_j) \in \mathcal{A} \quad (7)$$

Applying fusion to different edges of G_0 yields three derived DAGs (A1–A3) with reduced node count, enabling us to study the tradeoff between phase separation and phase fusion.

Category B: Phase Parallelization. Within each phase, the three semantic components V, C, O exhibit relative independence—variable extraction does not depend on constraint extraction, and objective identification proceeds independently of both. Phase parallelization exploits this decomposability

Table 1: The 12 architectures and their coalition mappings for Shapley attribution. $\oplus$: phase fusion; subscript $\|$: parallelization into sub-task agents; $\mathcal{E}_{\text{param}}$: shared parameter extraction node. Coalition column shows the operation set S in each Shapley game.

Category	**ID**	**Topology**	**Coalition** S
Base DAG	Base	$\mathcal{E} \to \mathcal{F} \to \mathcal{P}$	$\emptyset$ / $\emptyset$
A: Phase Fusion	A1	$\mathcal{E} \oplus \mathcal{F} \oplus \mathcal{P}$	$\{s_1, s_2\}$
	A2	$(\mathcal{E} \oplus \mathcal{F}) \to \mathcal{P}$	$\{s_1\}$
	A3	$\mathcal{E} \to (\mathcal{F} \oplus \mathcal{P})$	$\{s_2\}$
B: Phase Parallelization	B1	$\mathcal{E}_{\|} \to \mathcal{F} \to \mathcal{P}$	$\{p_{\mathcal{E}}\}$
	B2	$\mathcal{E} \to \mathcal{F}_{\|} \to \mathcal{P}$	$\{p_{\mathcal{F}}\}$
	B3	$\mathcal{E} \to \mathcal{F} \to \mathcal{P}_{\|}$	$\{p_{\mathcal{P}}\}$
	B4	$\mathcal{E}_{\|} \to \mathcal{F}_{\|} \to \mathcal{P}$	$\{p_{\mathcal{E}}, p_{\mathcal{F}}\}$
	B5	$\mathcal{E}_{\|} \to \mathcal{F} \to \mathcal{P}_{\|}$	$\{p_{\mathcal{E}}, p_{\mathcal{P}}\}$
	B6	$\mathcal{E} \to \mathcal{F}_{\|} \to \mathcal{P}_{\|}$	$\{p_{\mathcal{F}}, p_{\mathcal{P}}\}$
	B7	$\mathcal{E}_{\|} \to \mathcal{F}_{\|} \to \mathcal{P}_{\|}$	$\{p_{\mathcal{E}}, p_{\mathcal{F}}, p_{\mathcal{P}}\}$
C: Comp. Parallelization	C1	$\mathcal{E}_{\text{param}} \to (\mathcal{E} \to \mathcal{F} \to \mathcal{P})_{\|}$	—

by replacing a single node with a fork-join subgraph, formally a **vertex splitting** operation:

$$\text{Split}(n_i) = \text{Fork}(n_i^V,\, n_i^C,\, n_i^O) \to \text{Merge}_i \quad (8)$$

where three parallel sub-nodes each process one semantic component, and a merge node integrates their outputs. Merger details are introduced in Appendix A. Note that for the extraction phase, a parameter sub-task agent $n_{\mathcal{E}}^{\text{param}}$ runs in parallel alongside $V/C/O$, as parameters are shared inputs required by all downstream formulation and programming agents. Applying parallelization to different subsets of $\{\mathcal{E}, \mathcal{F}, \mathcal{P}\}$ yields seven derived DAGs (B1–B7) with increased width, revealing which phases benefit from decomposition.

Category C: Component-wise Parallelization. Unlike phase parallelization which decomposes within a single phase, component-wise parallelization operates along the component dimension across all phases. It first extracts shared parameters, then decomposes the remaining process into three fully independent tracks—one per semantic component (V, C, O). Each track executes its own complete $\mathcal{E} \to \mathcal{F} \to \mathcal{P}$ pipeline independently, and a code merger assembles the three fragments into the final program:

$$\begin{aligned} \text{CSplit}(G_0) :\ & \mathcal{E}_{\text{param}} \to \big(\mathcal{E}_k \to \mathcal{F}_k \to \mathcal{P}_k\big) \\ & \to \text{Merge}_{\text{code}}, \quad k \in \{V, C, O\} \end{aligned} \quad (9)$$

This operation produces a DAG with three independent parallel paths, investigating whether maintaining per-component coherence across all phases compensates for the loss of cross-component information sharing.

Within the predefined operation set, we enumerate all valid combinations. Moreover, these 12 architectures form two complete coalition sets, providing the necessary conditions for Shapley value attribution in the next section.

3.3 Attribution via Cooperative Game Theory

To quantify each topological operation's marginal contribution, we adopt the Shapley value (Shapley, 1988). We define a cooperative game (N, v) where players N are topological operations, each coalition $S \subseteq N$ maps to a concrete architecture, and the characteristic function $v(S)$ returns its accuracy. The Shapley value attributes the total performance gain by averaging each player's marginal contribution across all joining orders:

$$\begin{aligned} \phi_i = \sum_{S \subseteq N \setminus \{i\}} & \frac{|S|!\,(|N| - |S| - 1)!}{|N|!} \\ & \times \big[v(S \cup \{i\}) - v(S)\big] \end{aligned} \quad (10)$$

We conduct two Shapley analyses:

Phase Fusion Attribution. This game quantifies the effect of merging adjacent phases, revealing whether reducing pipeline depth and inter-agent communication overhead compensates for increased single-agent complexity. Players are two fusion operations: s_1 ($\mathcal{E} \oplus \mathcal{F}$) and s_2 ($\mathcal{F} \oplus \mathcal{P}$), yielding $2^2 = 4$ coalitions (Base and A1–A3 in Table 1).

Parallelization Attribution. This game quantifies the value of intra-phase parallelization, determining whether decomposing a phase into component-level sub-task agents improves accuracy through focused reasoning. Players are three split decisions: $p_{\mathcal{E}}$ ($\mathcal{E}_{\|}$), $p_{\mathcal{F}}$ ($\mathcal{F}_{\|}$), and $p_{\mathcal{P}}$ ($\mathcal{P}_{\|}$), yielding $2^3 = 8$ coalitions (Base and B1–B7 in Table 1).

4 Experiments

4.1 Experimental Setup

Datasets. We evaluate on four OR benchmarks spanning different difficulty levels: **NL4Opt** (Ramamonjison et al., 2023), **NLP4LP** (AhmadiTeshnizi et al., 2024b), **IndustryOR** (Huang et al., 2025), and **ComplexOR** (Xiao et al., 2024). These datasets span a wide spectrum of natural language

optimization problems, ranging from fundamental linear programming scenarios to complex industrial operations research tasks. Detailed descriptions of these datasets are provided in Appendix B.

Models and Configuration. We select three LLMs from different vendors with comparable capability tiers to ensure cross-family generalizability: DeepSeek-V3.2(DeepSeek-AI et al., 2025), GPT-5.4(OpenAI, 2026), and Qwen3.6-27B(Yang et al., 2025a). Temperature is set to 0.2 to ensure reproducible comparisons across architectures while retaining mild stochasticity that benefits multi-path code generation.

Metrics. We execute the LLM-generated solver code and compare its optimal objective value against the ground-truth answer to calculate the accuracy. Each experiment is repeated 5 times and averaged to mitigate stochastic variance. We additionally report average token and time usage per problem to characterize the cost-accuracy tradeoff of each architecture.

4.2 Prompt Standardization Results

Table 2 reports dimension-level SBERT cosine similarities across six representative agent roles. Most values exhibit a moderate mean (0.2–0.6) that potentially masks high intra-dimension variance, motivating a finer semantic sub-unit decomposition. Following this segmentation, a clear bimodal distribution emerges (Figure 2): 4 of the 11 sub-units achieve pairwise similarity above 0.6, while the remaining 7 fall below 0.3. Notably, each dimension contributes exactly one high-consensus sub-unit: Professional Identity, Primary Directive, Agent Information, and Output Definition. These universally capture *what* the agent should do, whereas the low-consensus sub-units encode *how* to execute it. Consequently, we semantically aggregate the high-consensus sub-units into standardized prompt templates, while retaining the low-consensus ones as optional skill modules for the topology–skill analysis. The complete prompt templates and skills documents are provided in Appendix D and Appendix E, respectively.

4.3 Architecture Accuracy Comparison

4.3.1 Main Results

Table 3 reports accuracy for all 12 architectures across four datasets and three LLMs.

Table 2: Dimension-level SBERT cosine similarity across agent roles. Moderate averages with high cross-dimension variance motivate sub-unit decomposition.

Agent Role	Persona	Task	Info	Output
Extractor($\mathcal{E}$)	.505	.484	.391	.236
Formulator($\mathcal{F}$)	.632	.477	.240	.361
Programmer($\mathcal{P}$)	.512	.385	.354	.292
Ex.$\oplus$For.($\mathcal{E}\oplus\mathcal{F}$)	.268	.417	.164	.327
For.$\oplus$Pro.($\mathcal{F}\oplus\mathcal{P}$)	—	.494	.314	.391
Ex.$\oplus$For.$\oplus$Pro. ($\mathcal{E}\oplus\mathcal{F}\oplus\mathcal{P}$)	.191	.593	.472	.115

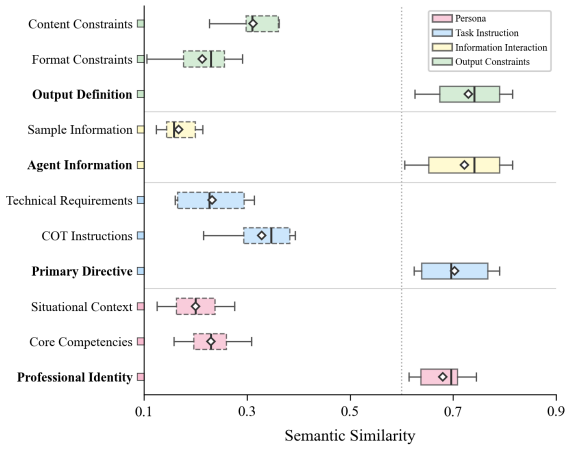

Figure 2: Sub-unit level semantic similarity distribution across four prompt dimensions. Each dimension exhibits one high-consensus sub-unit (>0.6) representing core task semantics, while remaining sub-units show low agreement (<0.3), confirming the bimodal partition.

Finding 1: Decomposition granularity yields divergent effects across problem complexity. On complex datasets (IndustryOR, ComplexOR), finer-grained decomposition (Category B) often outperforms fused architectures (Category A). On average, B5 exceeds Base by 5.27 points on IndustryOR and 10.4 points on ComplexOR. However, on simpler datasets (NL4Opt, NLP4LP), additional decomposition provides diminishing or negative returns—A3 achieves the highest single-model accuracy (90.95%) on NL4Opt, suggesting that well-structured problems may not benefit from task fragmentation.

Finding 2: Parallelization shows polarized effects. Among Category B architectures, B5 ($\mathcal{E}_{\|} \rightarrow \mathcal{F} \rightarrow \mathcal{P}_{\|}$) achieves the highest average accuracy (69.00%), while B2 ($\mathcal{E} \rightarrow \mathcal{F}_{\|} \rightarrow \mathcal{P}$) performs worst (62.38%)—below even Base (64.66%). This suggests that the choice of *which* stage to par-

Table 3: Architecture performance across LLMs and datasets (% accuracy). Highest and lowest results per column are highlighted in red and green, respectively.

Architecture	NL4Opt			NLP4LP			IndustryOR			ComplexOR			Avg
	DS	GPT	Qwen	DS	GPT	Qwen	DS	GPT	Qwen	DS	GPT	Qwen	
Base DAG													
Base	84.4	81.7	84.2	79.8	80.6	83.3	33.4	47.8	36.4	52.2	61.1	51.1	64.7
Category A: Phase Fusion													
A1	87.0	89.3	86.9	82.9	82.6	85.1	29.6	46.2	34.8	52.2	58.9	44.4	65.0
A2	85.4	84.8	84.2	80.4	79.3	84.7	33.4	46.8	35.8	53.3	57.8	50.0	64.7
A3	87.4	91.0	87.4	82.2	82.2	86.0	30.4	51.6	36.2	53.3	62.2	44.4	66.2
Category B: Phase Parallelization													
B1	84.4	84.4	84.4	81.8	81.8	82.2	36.4	54.4	37.4	55.6	63.3	55.6	66.8
B2	81.9	80.5	80.1	76.9	76.5	80.2	32.4	45.6	36.8	50.0	62.2	45.6	62.4
B3	88.3	83.0	85.6	79.3	81.0	82.6	31.2	54.0	38.8	55.6	66.7	44.4	65.9
B4	87.8	84.4	86.1	79.3	81.7	83.5	33.2	48.0	38.0	60.0	67.8	56.7	67.2
B5	84.4	83.9	83.9	78.5	82.2	86.1	38.8	57.6	37.0	67.8	70.0	57.8	69.0
B6	81.7	82.0	85.2	76.0	80.6	86.0	35.8	53.2	36.2	57.8	68.9	61.1	67.0
B7	85.2	84.0	88.3	76.9	81.9	86.1	36.6	54.2	38.2	63.3	61.1	56.7	67.7
Category C: Component-wise Parallelization													
C1	83.9	81.5	87.4	74.9	77.9	82.6	30.2	51.6	33.4	58.9	61.1	44.4	64.0

Table 4: Shapley value attribution of topological operations (percentage points). Positive values indicate beneficial operations.

Analysis	Operation	ϕ_i
Phase Fusion	s_1 ($\mathcal{E} \oplus \mathcal{F}$)	-0.61
	s_2 ($\mathcal{F} \oplus \mathcal{P}$)	$+0.94$
Parallelization	$p_{\mathcal{E}}$ ($\mathcal{E}_{\parallel}$)	$+2.26$
	$p_{\mathcal{F}}$ ($\mathcal{F}_{\parallel}$)	-0.93
	$p_{\mathcal{P}}$ ($\mathcal{P}_{\parallel}$)	$+1.71$

allelize matters more than whether to parallelize.

Finding 3: Intra-phase parallelism outperforms component-wise parallelism. B7 ($\mathcal{E}_{\parallel} \rightarrow \mathcal{F}_{\parallel} \rightarrow \mathcal{P}_{\parallel}$, 67.70%) outperforms C1 (63.99%), despite both decomposing all three stages by semantic component. The key factor is information flow: B7 merges component outputs after each phase, so downstream agents receive the full context from all components; while C1 isolates each track end-to-end, preventing cross-component information exchange until the final code merge.

4.3.2 Shapley Value Attribution

We apply the cooperative game framework using the average accuracies from Table 3, with the results summarized in Table 4.

Finding 4: Formulation-programming fusion improves accuracy. The $\mathcal{F}$–$\mathcal{P}$ fusion yields $\phi_{s_2} = +0.94$: formulation and programming share substantial mathematical context, so keeping them fused (A3) preserves notational continuity. The $\mathcal{E}$–$\mathcal{F}$ fusion yields $\phi_{s_1} = -0.61$: merging extraction into formulation increases single-agent complexity without proportional benefit, as the two stages require distinct reasoning modes.

Finding 5: Task decomposition benefits weakly coupled phases but impairs strongly coupled phases. Extraction parallelization contributes most ($\phi_{p_{\mathcal{E}}} = +2.26$): variables, constraints, and objectives appear in separate textual regions and can be extracted independently. Programming parallelization ranks second ($\phi_{p_{\mathcal{P}}} = +1.71$). Formulation parallelization receives a negative Shapley value ($\phi_{p_{\mathcal{F}}} = -0.93$): the mathematical modeling stage exhibits strong internal coupling, where isolating components into parallel sub-agents can disrupt the shared notational context essential for a coherent program.

4.4 Cost-Efficiency Tradeoffs

Figure 3 summarizes the accuracy–token Pareto frontiers.

Token consumption increases with topology complexity and problem difficulty. Fused architectures (A1–A3) consume 2–3× fewer tokens than parallel topologies (B5–B7, C1), whereas parallel branches incur *context duplication* by repeatedly receiving the shared problem description and upstream outputs. On simple datasets, A1 achieves near-Pareto-optimal accuracy at the lowest cost; on complex datasets, B5 lies on the strongest observed frontier. These results suggest *difficulty-adaptive*

Table 5: Effect of task-specific skills on architecture performance (DeepSeek). Results are reported as percentages rounded to one decimal place; Δ_{abs} shows absolute accuracy change.

Architecture	NL4Opt			NLP4LP			IndustryOR			ComplexOR			Avg
	w/o skills	w/ skills	Δ_{abs}	w/o skills	w/ skills	Δ_{abs}	w/o skills	w/ skills	Δ_{abs}	w/o skills	w/ skills	Δ_{abs}	(w/skills)
Base DAG													
Base	84.4	85.4	1.1	79.8	81.4	1.7	33.4	40.6	7.2	52.2	59.7	7.5	66.8
Category A: Phase Fusion													
A1	87.0	86.1	-0.9	82.9	82.6	-0.2	29.6	38.0	8.4	52.2	57.8	5.6	67.0
A2	85.4	85.0	-0.4	80.4	83.1	2.7	33.4	41.4	8.0	53.3	57.8	4.5	66.0
A3	87.4	86.9	-0.5	82.2	81.4	-0.8	30.4	37.4	7.0	53.3	57.8	4.4	65.9
Category B: Phase Parallelization													
B1	84.4	86.5	2.2	81.8	79.3	-2.5	36.4	37.8	1.4	55.6	56.7	1.1	65.1
B2	81.9	82.7	0.8	76.9	78.5	1.7	32.4	34.2	1.8	50.0	53.3	3.3	62.2
B3	88.3	86.5	-1.7	79.3	81.4	2.1	31.2	33.2	2.0	55.6	55.6	0.0	64.2
B4	87.8	87.9	0.1	79.3	78.1	-1.2	33.2	35.8	2.6	60.0	62.2	2.2	66.0
B5	84.4	84.8	0.4	78.5	78.9	0.4	38.8	39.0	0.2	67.8	68.9	1.1	67.9
B6	81.7	84.4	2.7	76.0	78.5	2.5	35.8	36.0	0.2	57.8	61.1	3.3	65.0
B7	85.2	83.6	-1.6	76.9	78.1	1.2	36.6	37.2	0.6	63.3	64.4	1.1	65.8
Category C: Component-wise Parallelization													
C1	83.9	84.4	0.4	74.9	77.7	2.8	30.2	32.4	2.2	58.9	60.0	1.1	63.6

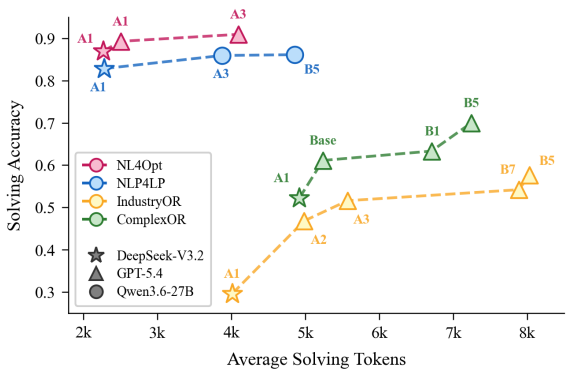

Figure 3: Pareto Frontiers of Accuracy vs. Token Cost

architecture selection: minimal pipelines for well-structured problems and fine-grained decomposition for complex industrial ones. Detailed results are provided in Appendix F.

4.5 Effect of Agent-Specific Skills

Table 5 assesses skill modules using DeepSeek-V3.2, which exhibits the highest sensitivity to topological variations(Table 3). This heightened sensitivity allows for a more clear observation of skill-topology interactions.

Finding 6: Topology advantages persist under prompt enrichment via agent-specific skills. Adding skills preserves the relative performance ordering among architectures—B5 remains the best and B2 the worst regardless of skill configuration. This demonstrates that the architectural advantages are robust to prompt enrichment.

Finding 7: Skills do not uniformly improve performance. On complex datasets, skills consistently help, but on simple datasets several architectures lose 0.2–2.5 points. Moreover, B5 shows near-zero skill sensitivity (+0.20 to +1.11), whereas Base benefits most (+7.50), suggesting diminishing returns from prompt enrichment under effective topology.

5 Conclusion

Given the wide variety of multi-agent architectures for OR optimization, there is no systematic understanding of which topological choices drive performance. ORFlow addresses this gap by standardizing prompts through cross-system semantic consensus, formalizing topology design as graph operations over a base DAG, and using Shapley values to quantify the contribution of each operation. Controlled experiments identify effective structures for OR solving and show that performance differences are associated with inter-phase coupling strength, providing practical criteria for multi-agent workflow design.

6 Limitations

While this study suggests that weakly coupled phases may benefit from decomposition, our empirical findings also reveal that architectural configurations exhibit distinct, dataset-specific preferences. Employing a static topology across diverse operations research scenarios may therefore constrain overall generalizability. Future research could ex-

plore adaptive frameworks capable of dynamically selecting the optimal multi-agent topology tailored to the intrinsic characteristics of the given problem instance.

Furthermore, ORFlow provides a systematic framework for quantifying structural topology effects across distinct multi-agent optimization pipelines. Nevertheless, the current evaluation is still restricted to unidirectional, single-pass architectures modeled as DAGs, which may not fully capture the dynamic interactions, such as iterative debugging and solver-driven feedback loops. Future work should extend the architectural design space to incorporate cyclic dependency graphs, so as to rigorously evaluate and benchmark how error-correction mechanisms interact with different base architectures.

Ethics Statement

There are no ethical concerns raised by our work as the data used in this study is public without sensitive information.

Acknowledgements

This work was supported by xx.

References

Ali AhmadiTeshnizi, Wenzhi Gao, Herman Brunborg, Shayan Talaei, Connor Lawless, and Madeleine Udell. 2024a. Optimus-0.3: Using large language models to model and solve optimization problems at scale. *arXiv preprint arXiv:2407.19633*.

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. 2023. Optimus: Optimization modeling using mip solvers and large language models. *arXiv preprint arXiv:2310.06116*.

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. 2024b. Optimus: Scalable optimization modeling with (mi) lp solvers and large language models. *arXiv preprint arXiv:2402.10172*.

Tasnim Ahmed and Salimur Choudhury. 2024. Lm4opt: Unveiling the potential of large language models in formulating mathematical optimization problems. *INFOR: Information Systems and Operational Research*, 62(4):559–572.

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 225–237.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela Van Der Schaar. 2024. Autoformulation of mathematical optimization models using llms. *arXiv preprint arXiv:2411.01679*.

Dimitris Bertsimas and Georgios Margaritis. 2024. Robust and adaptive optimization under a large language model lens. *arXiv preprint arXiv:2501.00568*.

John Adrian Bondy and Uppaluri Siva Ramachandra Murty. 2008. *Graph theory*. Springer Publishing Company, Incorporated.

Hao Chen, Gonzalo Esteban Constante-Flores, Krishna Sri Ipsit Mantri, Sai Madhukiran Kompalli, Akshdeep Singh Ahluwalia, and Can Li. 2025. Optichat: Bridging optimization models and practitioners with large language models. *INFORMS Journal on Data Science*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2024. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Zezhen Ding, Zhen Tan, Jiheng Zhang, and Tianlong Chen. 2026. Or-r1: Automating modeling and solving of operations research optimization problem via test-time reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 228–236.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang

Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. *Preprint*, arXiv:2308.00352.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. 2025. Orlm: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 73(6):2986–3009.

Nam Huynh and Beiyu Lin. 2025. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv preprint arXiv:2503.01245*.

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. 2025. Llmopt: Learning to define and solve general optimization problems from scratch. *Preprint*, arXiv:2410.13213.

Douglas M Lambert and Martha C Cooper. 2000. Issues in supply chain management. *Industrial marketing management*, 29(1):65–83.

Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. 2023a. Large language models for supply chain optimization. *arXiv preprint arXiv:2307.03875*.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023b. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008.

Junxuan Li, Ryan Wickman, Sahil Bhatnagar, Raj Kumar Maity, and Arko Mukherjee. 2025. Abstract operations research modeling using natural language inputs. *Information*, 16(2):128.

Qingyang Li, Lele Zhang, and Vicky Mak-Hau. 2023c. Synthesizing mixed-integer linear programming models from natural language descriptions. *arXiv preprint arXiv:2311.15271*.

Xin Li, Qizhi Chu, Yubin Chen, Yang Liu, Yaoqi Liu, Zekai Yu, Weize Chen, Chen Qian, Chuan Shi, and Cheng Yang. 2024. Graphteam: Facilitating large language model-based graph analysis via multi-agent collaboration. *arXiv preprint arXiv:2410.18032*.

Kuo Liang, Yuhang Lu, Jianming Mao, Shuyi Sun, Chunwei Yang, Congcong Zeng, Xiao Jin, Hanzhang Qin, Ruihao Zhu, and Chung-Piaw Teo. 2026. Llm for large-scale optimization model auto-formulation: A lightweight few-shot learning approach. *arXiv preprint arXiv:2601.09635*.

Vinicius Lima, Dzung T Phan, Jayant Kalagnanam, Dhaval Patel, and Nianjun Zhou. 2025. Toward a trustworthy optimization modeling agent via verifiable synthetic data generation. *arXiv preprint arXiv:2508.03117*.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. 2025. Optmath: A scalable bidirectional data synthesis framework for optimization modeling. *arXiv preprint arXiv:2502.11102*.

Zeyuan Ma, Yue-Jiao Gong, Hongshu Guo, Jiacheng Chen, Yining Ma, Zhiguang Cao, and Jun Zhang. 2026. Llamoco: Instruction tuning of large language models for optimization code generation. *IEEE Transactions on Evolutionary Computation*.

Mahdi Mostajabdaveh, Timothy T Yu, Rindranirina Ramamonjison, Giuseppe Carenini, Zirui Zhou, and Yong Zhang. 2024. Optimization modeling and verification from problem specifications using a multi-agent multi-stage llm framework. *INFOR: Information Systems and Operational Research*, 62(4):599–617.

Sumeet Ramesh Motwani, Chuan Du, Aleksander Petrov, Christopher Davis, Philip Torr, Antonio Papania-Davis, and Weishi Yan. 2026. Autoor: Scalably post-training llms to autoformalize operations research problems. *arXiv preprint arXiv:2604.16804*.

OpenAI. 2026. Introducing GPT-5.4. `https://openai.com/index/introducing-gpt-5-4/`. Accessed on 2026-05-25.

Michael Pinedo and Khosrow Hadavi. 1992. Scheduling: theory, algorithms and systems development. In *Operations research proceedings 1991: Papers of the 20th annual meeting/vorträge der 20. Jahrestagung*, pages 35–42. Springer.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Chatdev: Communicative agents for software development. *Preprint*, arXiv:2307.07924.

Rindranirina Ramamonjison, Timothy T. Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. 2023. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. *Preprint*, arXiv:2303.08233.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pages 3982–3992.

Lloyd S. Shapley. 1988. A value for n-person games.

Amirreza Talebi. 2025. Large language model-based automatic formulation for stochastic optimization models. *arXiv preprint arXiv:2508.17200*.

Zhengyang Tang, Zihan Ye, Chenyu Huang, Xuhan Huang, Chengpeng Li, Sihang Li, Guanhua Chen, Ming Yan, Zizhuo Wang, Hongyuan Zha, Dayiheng Liu, and Benyou Wang. 2025. Calm before the storm: Unlocking native reasoning for optimization modeling. *Preprint*, arXiv:2510.04204.

Paolo Toth and Daniele Vigo. 2014. *Vehicle routing: problems, methods, and applications*. SIAM.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6).

Zhiyuan Wang, Bokui Chen, Yinya Huang, Qingxing Cao, Ming He, Jianping Fan, and Xiaodan Liang. 2025. Ormind: A cognitive-inspired end-to-end reasoning framework for operations research. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pages 104–131.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. *Preprint*, arXiv:2308.08155.

Yang Wu, Yifan Zhang, Yurong Wu, Yuran Wang, Junkai Zhang, and Jian Cheng. 2025. Step-opt: Boosting optimization modeling in llms through iterative data synthesis and structured validation. *arXiv preprint arXiv:2506.17637*.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, and 10 others. 2023. The rise and potential of large language model based agents: A survey. *Preprint*, arXiv:2309.07864.

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

Wantong Xie, Yi-Xiang Hu, Jieyang Xu, Feng Wu, and Xiangyang Li. 2026. Murka: Multi-reward reinforcement learning with knowledge alignment for optimization tasks. *Advances in Neural Information Processing Systems*, 38:34878–34905.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pages 12028–12068.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Shi Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. 2025b. Optibench meets resocratic: Measure and improve llms for optimization modeling. In *International Conference on Learning Representations*, volume 2025, pages 24726–24759.

Zike Yuan, Ming Liu, Hui Wang, and Bing Qin. 2025. Ma-gts: A multi-agent framework for solving complex graph problems in real-world applications. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 19297–19315.

Bowen Zhang and Pengcheng Luo. 2025. Or-llm-agent: Automating modeling and solving of operations research optimization problem with reasoning large language model. *arXiv e-prints*, pages arXiv–2503.

Guoyun Zhang. 2025. Large language model enabled mathematical modeling. *arXiv preprint arXiv:2510.19895*.

Jihai Zhang, Wei Wang, Siyan Guo, Li Wang, Fangquan Lin, Cheng Yang, and Wotao Yin. 2024. Solving general natural-language-description optimization problems with large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pages 483–490.

Ni Zhang, Zhiguang Cao, Jianan Zhou, Cong Zhang, and Yew-Soon Ong. 2025. An agentic framework with llms for solving complex vehicle routing problems. *arXiv preprint arXiv:2510.16701*.

A Agents Team

Table 6 summarizes all agent roles employed across our 12 architectures. Agents are grouped into four categories: **3 main-task agents** that each handle a full pipeline phase, **3 fused agents** that collapse multiple phases into a single LLM call, **10 sub-task agents** that operate on a specific component dimension within parallel branches, and **3 merger agents** that reassemble specialist outputs.

Table 6: Agent roles in the multi-agent OR pipeline. All agents are LLM-powered except Extraction Merger and Formulation Merger, which are noted as *deterministic*.

Category	Agent	Description
Main-task Agents	Extractor	Identifies parameters, variables, constraints, and objective
	Formulator	Converts extracted components into LaTeX
	Programmer	Generates executable Python code
Fused Agents	Extractor ⊕ Formulator	Jointly extracts and formulates
	Formulator ⊕ Programmer	Jointly formulates and programs
	Extractor ⊕ Formulator ⊕ Programmer	Single agent performing three phases jointly
Sub-task Agents	{Param, Var, Constr, Obj} Extractor	Extracts specific components from Natural Language
	{Var, Constr, Obj} Formulator	Formulates specific components in LaTeX
	{Var, Constr, Obj} Programmer	Generates code for specific components
Merger Agents	Extraction Merger	*Deterministic*; concatenates extracted components by field without modification
	Formulation Merger	*Deterministic*; assembles LaTeX formulation sections without modification
	Code Merger	*LLM-based*; integrates fragments into one coherent executable program, resolving dependencies

B Benchmark Datasets

We evaluate our framework on four real-world optimization datasets of increasing difficulty:

NL4Opt (Ramamonjison et al., 2023) provides a standard-difficulty formulation benchmark of 230 linear programming word problems, with specific domains (e.g., production, transportation) exclusively reserved for testing.

NLP4LP (AhmadiTeshnizi et al., 2024b) features 242 linear programming word problems featuring extended and linguistically diverse narratives.

IndustryOR (Huang et al., 2025) mirrors practical industrial complexity through 100 testing instances across 13 sectors, encompassing linear and mixed-integer programming (MIP) structured into three difficulty levels.

ComplexOR (Xiao et al., 2024) serves as a highly demanding benchmark, comprising 18 highly challenging instances co-curated with domain specialists from advanced scenarios like supply chain logistics.

C Details of Consensus-Based Prompt Standardization

This appendix details how raw prompts are classified, transformed into standardized prompt templates and optional `SKILL.md` modules, and validated through dimension-wise similarity heatmaps.

C.1 Literature Survey and Agent Function Classification

Table 7 surveys the agent role decomposition across representative LLM-based OR systems in the literature. For each system, we identify which sub-task agents are explicitly instantiated across three core phases—extraction, formulation, and compilation—as well as auxiliary reviewer and conductor roles. The presence of a dedicated sub-task agent (e.g., a constraint extractor) versus a monolithic phase agent reflects the system's topology granularity. We extract and analyze the prompts from all surveyed systems to inform our consensus-based prompt standardization, ensuring that our controlled experiments use representative, system-neutral templates rather than prompts biased toward any single framework.

C.2 Prompt Classification Rules

This subsection defines the classification rules used to convert heterogeneous raw prompts into comparable semantic units. Each prompt fragment is first assigned to one of four dimensions: *Persona*, *Task Instruction*, *Information Interaction*, and *Output Constraints*. Within each dimension, the fragment is further assigned to one of the sub-units in Table 8. These rules provide the annotation basis for the subsequent similarity computation.

Table 7: Agent role decomposition across LLM-based OR systems. ✓: dedicated task agent present; ✗: not applicable or absent; shaded ✓ : monolithic phase agent (no sub-task decomposition).

System	Extractor				Formulator	Compiler			Reviewer			Conductor
	Var	Constr	Obj	Param	–	Obj	Constr	Var	Code	Formul.	Extract.	–
OptiMUS_1 (AhmadiTeshnizi et al., 2023)	✗	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗	✗
OptiMUS_2 (AhmadiTeshnizi et al., 2024b)	✗	✗	✗	✗	✓		✓	✓	✓	✓	✗	✓
OptiMUS_3 (AhmadiTeshnizi et al., 2024a)	✓	✓	✓	✓	✓		✓		✓	✗	✓	✗
(Li et al., 2023c)	✓		✓		✓	✗	✗	✗	✗	✗	✗	✗
OptiGuide (Li et al., 2023a)	✗	✗	✗	✗	✗		✓		✓	✗	✗	✗
Chain-of-Experts (Xiao et al., 2024)	✓	✓	✓	✓	✓		✓		✓	✗	✗	✓
OPRO (Yang et al., 2024)					✓				✗	✗	✗	✗
LM4Opt (Ahmed and Choudhury, 2024)		✓		✗	✗	✗	✗	✗	✗	✗	✗	✗
NL2OPT (Mostajabdaveh et al., 2024)		✓		✓	✓		✓		✓	✓	✗	✗
(Bertsimas and Margaritis, 2024)			✓				✓		✗	✗	✗	✗
LLaMoCo (Ma et al., 2026)	✗	✗	✗	✗	✗		✓		✗	✗	✗	✗
OptLLM (Zhang et al., 2024)		✓			✓		✓		✓	✓	✗	✗
NL2OR (Li et al., 2025)			✓			✗	✗	✗	✗	✗	✗	✗
OptiTrust (Lima et al., 2025)		✓			✓		✓		✓	✓	✓	✗
Step-Opt (Wu et al., 2025)					✓				✗	✗	✗	✗
OR-LLM-Agent (Zhang and Luo, 2025)	✗	✗	✗	✗	✓		✓		✓	✓	✗	✗
OptiChat (Chen et al., 2025)	✗	✗	✗	✗	✗		✓		✓	✗	✗	✓
LEAN-LLM-OPT (Liang et al., 2026)					✓				✗	✗	✗	✗
ORMind (Wang et al., 2025)		✓			✓		✓		✓	✗	✗	✓
Autoformulator (Astorga et al., 2024)	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
OptiBench (Yang et al., 2025b)	✗	✗	✗	✗	✗		✓		✗	✗	✗	✗
LLMOpt (Jiang et al., 2025)			✓				✓			✓	✗	✗
(Zhang, 2025)	✗	✗	✗	✗	✓		✓		✗	✗	✗	✗
(Talebi, 2025)		✓					✓			✓	✗	✗
(Li et al., 2024)		✓			✗		✓		✓	✗	✗	✗
MA-GTS (Yuan et al., 2025)		✓			✗		✓		✓	✗	✗	✗
AFL (Zhang et al., 2025)			✓				✓		✓	✗	✗	✗
OptMATH (Lu et al., 2025)					✓				✗	✗	✗	✗
ORLM (Huang et al., 2025)					✓				✗	✗	✗	✗
(Tang et al., 2025)					✓					✓	✗	✗

After applying the similarity computation described in Section 3.1, the bolded sub-units in Table 8 form the high-similarity group. These high-consensus semantics are semantically aggregated into fixed prompt templates. The remaining sub-units form the low-similarity group and are semantically expanded into optional `SKILL.md` modules. The following case study makes this classification-and-transformation procedure concrete.

C.3 Case Study: The Formulator ⊕ Programmer Agent

We use the Formulator ⊕ Programmer Agent as a demonstration example. This fused role receives extracted OR components and jointly produces a mathematical formulation and executable solver code. The case study shows how the classification rules in Table 8 are applied to raw prompt fragments, how high-similarity fragments are semantically aggregated into the base prompt, and

Table 8: Semantic sub-unit taxonomy for consensus-based prompt standardization.

Dimension	Sub-unit	Description
Persona	**Professional Identity**	Role identity and domain position.
	Core Competencies	Expertise required by the task.
	Situational Context	Operating context and agent relation.
Task Instruction	**Primary Directive**	Main objective that the agent should complete.
	COT Instructions	Reasoning or stepwise guidance.
	Technical Requirements	Modeling, coding, or solver rules.
Information Interaction	**Agent Information**	Inputs provided during execution.
	Sample Information	Examples or reference cases.
Output Constraints	**Output Definition**	Required semantic content of the response.
	Format Constraints	Required structure or syntax.
	Content Constraints	Restrictions on assumptions or irrelevant text.

Table 9: Raw prompt decomposition for the Formulator ⊕ Programmer Agent. Each cell reports the prompt fragment assigned to the corresponding semantic sub-unit. Empty cells indicate that the source prompt does not explicitly specify that sub-unit.

Dimension	Sub-unit	System 1	System 2	System 3
Persona	**Professional Identity**	–	–	–
	Core Competencies	–	–	–
	Situational Context	–	–	–
Task Instruction	**Primary Directive**	**Use Python code to solve the OR question with Gurobi.**	**Construct a mathematical model and write complete Python code using Gurobi.**	**Code the complete model in Python using Gurobi.**
	COT Instructions	First establish the formulation step by step, then generate Python code using `pyscipopt`.	–	Clearly define the objective function and code all relevant constraints.
	Technical Requirements	Follow the examples to solve the given question.	–	Explicitly include the recourse function.
Information Interaction	**Agent Information**	–	**Based on the optimization problem description provided by the user.**	**Here is the new optimization problem description.**
	Sample Information	Two formulation and Python code examples are provided.	–	A code template is provided.
Output Constraints	**Output Definition**	**The output must contain formulation and Python sections.**	**Include model construction, variable definitions, constraints, objective, solving, and result output.**	–
	Format Constraints	Use two sections: `formulation` and `python`.	Output Python code using Gurobi in a fenced code block without explanations.	You should strictly follow the template format.
	Content Constraints	–	The code should be complete and reliable.	–

how low-similarity fragments are converted into a skill document.

(a) Semantic Decomposition of Raw Prompts. Table 9 gives a schematic decomposition of paraphrased prompt fragments collected from representative LLM-based OR systems. Each fragment is first assigned to one of the four dimensions and is then mapped to the sub-units in Table 8. The example illustrates how a long instruction is converted into comparable semantic units before similarity computation.

(b) High-Similarity Semantic Aggregation into a Prompt Template. The high-consensus fragments highlighted in Table 9 are provided to Llama-3-8B-Instruct with temperature set to zero. The following interaction box illustrates this process: the aggregation agent receives the high-similarity fragments and a compact semantic aggregation instruction, then returns a four-dimensional standardized prompt template. The complete prompt template for this agent is reported in Appendix D.

Semantic Aggregation

SYSTEM MESSAGE

- You are an advanced AI agent specialized in qualitative research and prompt engineering.
- Your goal is to execute a semantic aggregation task with high precision.
- You must maintain variable format consistency, filter noise strictly, and output only the standardized template in the requested format.

USER MESSAGE

Mission Context. Build a standardized prompt template for the Formulator ⊕ Programmer Agent in an OR agent system.
Input Data. {High-similarity sub-units from Table 9: role label; Task Instruction–Primary Directive; Information Interaction–Agent Information; Output Constraints–Output Definition}.
Execution Instructions.

1. **Consensus Extraction**: identify the core semantic requirement shared by most excerpts.
2. **Noise Filtering**: discard solver-specific examples or unique details that appear in only one excerpt, unless critical for the general task.
3. **Variable Standardization**: replace placeholders such as `[DATA]`, `<input>`, or "text" with Python f-string format, such as `{input_text}`.
4. **Rewrite**: synthesize the consensus into a professional and precise four-dimensional template.

AGENT OUTPUT

Standardized prompt template:

[Task Instruction]
Construct a mathematical model based on the provided optimization problem and write complete Python code using Gurobi to solve this operations research question.

[Information Interaction]
Here is the extracted four components: {extracted_components}

[Output Constraints]
The output must contain two sections: a 'formulation' section with defined variables, objective function, and constraints, and a 'python' section providing Python code using Gurobi. The code must include model construction, variable definitions, constraint additions, objective function setting, optimization, and result output (status, variables, and objective value), formatted as python{code} without explanations.

(c) Low-Similarity Semantic Expansion into `SKILL.md`. We preserve the low-similarity sub-units to capture useful, system-specific execution strategies. We define the unbolded cells in Table 9 as *skill seeds*, which are omitted from the neutral base prompt but expanded into an optional `SKILL.md` module. This expansion utilizes a fixed slot-filling template to ensure that diverse execution heuristics are integrated as structured, loadable skills rather than arbitrary prompt extensions. Full details of the expanded skills are reported in Appendix E.

Semantic Expansion

INPUT SEEDS

{Unbolded low-similarity sub-units from Table 9: COT Instructions; Technical Requirements; Sample Information; Format Constraints; Content Constraints}.

EXPANSION MAP

Source sub-unit	Expanded `SKILL.md` section
Base agent setting	**[Metadata]**, **[Input]**, and **[Output]**; these define when the skill is triggered and what interface it preserves.
COT Instructions + Technical Requirements	**[Advanced Cognitive Directives]**; stepwise reasoning, formulation audits, recourse handling, solver-family discipline, and code-generation checks are rewritten as executable guidance.
Format Constraints + Content Constraints	**[Strict Format Constraints and Output Definition]**; output style, code-block discipline, callable function rules, dependency limits, and evaluator compatibility are made explicit.
Sample Information	**[Expected Output Schema]**; examples are abstracted into a reusable response skeleton rather than copied verbatim.

C.4 Dimension-wise Similarity Heatmaps

We also report the aggregate similarity patterns across all three main-task agents and three fused agents. Based on the classification rules in Table 8, we compute semantic similarity within each of the four prompt dimensions for each role. Figures 4 and 5 present the resulting dimension-wise heatmaps. Each row corresponds to one agent role, and the four columns correspond to *Persona*, *Task Instruction*, *Information Interaction*, and *Output Constraints*. These heatmaps serve as supplementary evidence for the case study by showing which dimensions contain stable cross-system consensus and which dimensions exhibit larger variation.

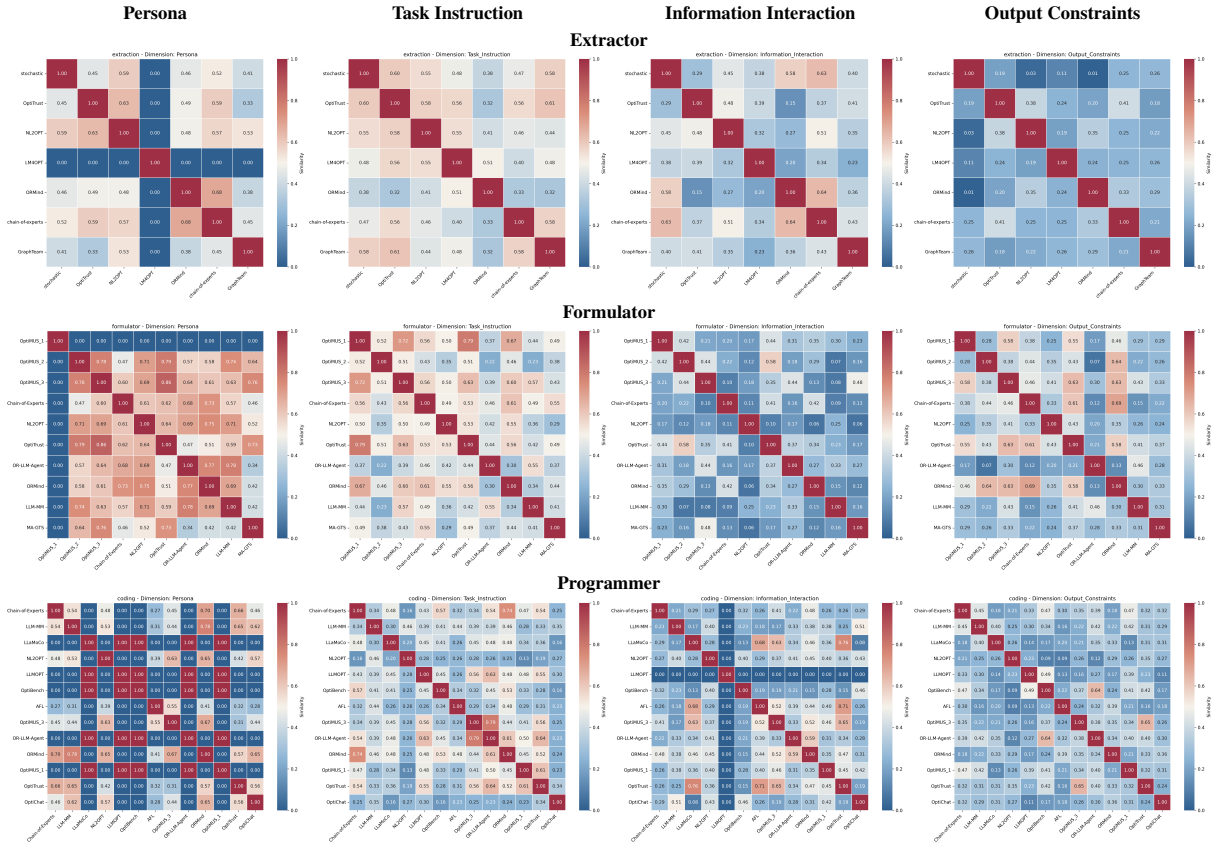

Figure 4: Dimension-wise prompt similarity heatmaps for the three single-phase agent roles. Each row shows one role from left to right across Persona, Task Instruction, Information Interaction, and Output Constraints.

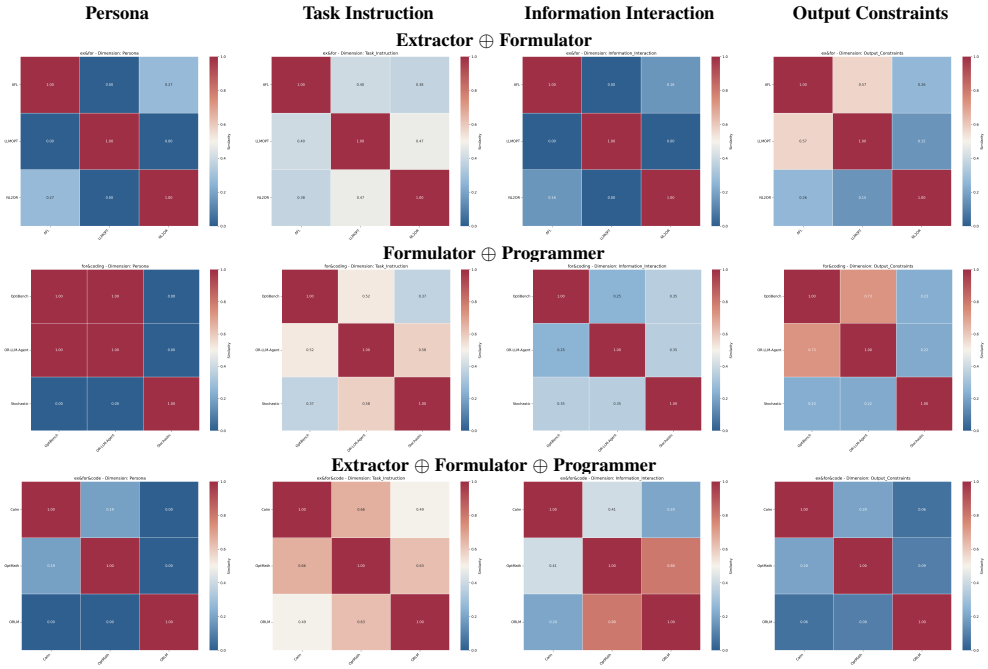

Figure 5: Dimension-wise prompt similarity heatmaps for the three fused agent roles. Each row shows one role from left to right across Persona, Task Instruction, Information Interaction, and Output Constraints.

D Agents Prompt Templates

This appendix reports the standardized prompt templates for the three main-task agents and the three fused agents used in our architecture space. Each template is constructed by semantically aggregating high-similarity prompt sub-units and then concatenating the resulting four dimensions: *Persona*, *Task Instruction*, *Information Interaction*, and *Output Constraints*.

Extractor Prompt

[Persona]
You are an expert in Operations Research, specializing in analyzing problem statements to identify and extract relevant information for building optimization models.

[Information Interaction]
You are given a problem description in natural language. Please review the new problem description: {problem_description}

[Task Instruction]
Your task is to identify and extract the following components from the problem description: parameters, decision variables, constraints, and objective.

[Output Constraints]
Your response must be in JSON format. Provide only the requested JSON output without any additional information.

Formulator Prompt

[Persona]
You are an expert in operations research and mathematical optimization, specialized in formulating and analyzing optimization models based on natural language descriptions.

[Information Interaction]
Here is the extracted four components: {extracted_components}

[Task Instruction]
Your task is to formulate a comprehensive mathematical model for the given optimization problem. You should analyze the problem's characteristics and clarify the meaning of each parameter.

[Output Constraints]
Your output format should be a JSON file containing a concise description of variables and their shapes or types, along with mathematical formulas for the constraints and the objective.Your formulation should be in LaTeX mathematical format.

Programmer Prompt

[Persona]
You are an expert in operations research and optimization, specializing in Python programming to solve complex problems.

[Information Interaction]
Here is the mathematical formulation in LaTeX for the optimization problem: {latex_formulation}

[Task Instruction]
Generate a Python program to solve the given mathematical model, ensuring the code is efficient, complete, and outputs the optimal solution and objective function value.

[Output Constraints]
Generate Python code using the Gurobi solver to implement the mathematical model, adhering to PEP 8 style guidelines and providing the code directly in the specified format.

Extractor ⊕ Formulator Prompt

[Persona]
You are an assistant working to generate a python code file for an optimization process which specifies the following parameters of the process.

[Information Interaction]
You will be provided with the operations research problem instance description: {problem_description}

[Task Instruction]
Extract the following components from the text: parameters, decision variables, constraints, and objective. Then, model the corresponding optimization problem.

[Output Constraints]
Output your formulation in JSON file exactly in this order: 1. constraints, 2. parameters, 3. variables, 4. objective.

Formulator ⊕ Programmer Prompt

[Task Instruction]
Construct a mathematical model based on the provided optimization problem and write complete Python code using Gurobi to solve this operations research question.

[Information Interaction]
Here is the extracted four components: {extracted_components}

[Output Constraints]
The output must contain two sections: a 'formulation' section with defined variables, objective function, and constraints, and a 'python' section providing Python code using Gurobi. The code must include model construction, variable definitions, constraint additions, objective function setting, optimization, and result output (status, variables, and objective value), formatted as python{code} without explanations.

Extractor ⊕ Formulator ⊕ Programmer Prompt

[Task Instruction]
Please analyze the given mathematical problem to systematically extract its key components: parameters, decision variables, constraints, and the objective function. Based on these components, formulate a precise mathematical model. Then, implement the solution in Python using the 'gurobipy' library, ensuring the code appropriately addresses the optimization problem and providing clear, detailed explanations for the modeling process.

[Information Interaction]
The following is the operations research problem description: {problem_description}

[Output Constraints]
Provide clear reasoning and explanations at each stage. Your final code block should print all key results needed for the solution.

E Agents Skills Documents

This appendix presents the `SKILL.md` documents for the three main-task agents and the three fused agents. Each skill document is derived from low-similarity prompt sub-units through semantic expansion, preserving system-specific execution strategies as optional modules.

Extractor Skill

[Metadata]
Name: extractor
Description: Triggered for Operations Research problems. Enforces rigorous cognitive unpacking, requiring deep logic auditing, strict adherence to given parameters without hallucination, and generating a highly constrained JSON output for modeling.

[Input]
Raw OR problem description.

[Output]
Strict JSON format containing parameters, variables, objective, and constraints.

[Strict Format Constraints and Output Definition]

1. **Naming convention**: The component name for parameters and variables must be lower case and contain no whitespaces.
2. **Separator rule**: The list of required parameters and variables must be separated with a colon without whitespaces around the key in the JSON structure.
3. **Mathematical operator constraint**: Within the constraints, the comparison operators must only be equal to, greater than or equal to, or less than or equal to. Strict inequalities ($>$ or $<$) are not allowed and must be replaced with $\geq$ or $\leq$.
4. **No conversational filler**: The response must only discuss these key elements and must not discuss anything else. Output only the valid JSON object. Do not include markdown formatting blocks if it breaks your system parser; just output the raw JSON string starting with '{' and ending with '}'.

[Expected Output Schema]

```
{
  "parameters": {
    "parameter1": {
      "Type": "The parameter data type or shape",
      "Definition": "A brief definition of the parameter exactly as found in text"
    },
    "parameter2": {
      "Type": "The parameter data type or shape",
      "Definition": "A brief definition of the parameter exactly as found in text"
    }
  },
  "variables": {
    "variable1": {
      "Type": "The variable data type or shape",
      "Definition": "A brief definition of the decision variable"
    }
  },
  "objective": "A mathematical formula or clear definition representing the goal",
  "constraints": [
    "Constraint 1 definition or formula",
    "Constraint 2 definition or formula"
  ]
}
```

Formulator Skill

[Metadata]
Name: formulator
Description: Triggered for Operations Research problems. Guides the Formulator to transform extracted components into a solvable mathematical programming model, preserve source semantics, clarify the meaning of each existing parameter, and emit a strictly constrained JSON and LaTeX formulation.

[Input]
Extracted components.

[Output]
Strict JSON format containing variable definitions, optional new variables, constraints, auxiliary constraints, and objective in LaTeX.

[Advanced Cognitive Directives]
Upon receiving the extracted components, the agent must operate in a concise but rigorous formulation mode and follow these steps:

1. **Variable and parameter check**: Review the existing variables and parameters, clarify each variable symbol, type, and required non-negativity, and do not introduce new parameters. Add new variables only when they are truly necessary.
2. **Objective construction**: Express the objective function precisely in LaTeX. If multiple objectives appear, convert them into a single-objective formulation while

preserving the original intent as much as possible.

3. **Constraint formulation**: Write all explicit constraints and any necessary implicit constraints. If the existing variables are insufficient, auxiliary variables and auxiliary constraints may be added, but the formulation must avoid redundancy and remain faithful to the source description.

[Strict Format Constraints and Output Definition]

1. **Output form**: The output must be a complete and valid JSON object, containing only the result itself with no extra commentary, and it must be directly parsable by Python `json.loads()`.
2. **Fixed top-level keys**: The JSON object must use these top-level keys: `VARIABLES`, `NEW_VARIABLES`, `CONSTRAINTS`, `AUXILIARY_CONSTRAINTS`, and `OBJECTIVE`; `OBJECTIVE` must appear last.
3. **Mathematical expression rule**: All formulas in `VARIABLES`, `CONSTRAINTS`, `AUXILIARY_CONSTRAINTS`, and `OBJECTIVE` must be written in LaTeX. Variable names and parameter names must use `\textup{}`, and `\quad` should be used when spacing is needed.
4. **Constraint rule**: Constraints may only use =, `\geq`, or `\leq`, never > or <. The objective and constraints should stay as close as possible to the original natural language description, and the output must be complete.

[Expected Output Schema]

```
{
  “VARIABLES”: [
    {
      “symbol”: “\\textup{Flow}_{i,j}”,
      “type”: “Continuous | Integer |
Binary”,
      “definition”: “A concise definition of
the decision variable”
    }
  ],
  “NEW_VARIABLES”: [],
  “CONSTRAINTS”: [
    “\\sum_{i \\in I} \\textup{Flow}_{i,j}
\\leq \\textup{Capacity}_{j}”
  ],
  “AUXILIARY_CONSTRAINTS”: [],
  “OBJECTIVE”: “\\min \\sum_{i \\in I}
\\textup{Cost}_{i,j} \\textup{Flow}_{i,j}”
}
```

Programmer Skill

[Metadata]
Name: programmer
Description: Triggered for Operations Research problems. Guides the Programmer to translate the upstream formulation into runnable Gurobi Python code, validate modeling consistency, preserve variable domains and implicit constraints, and emit clean executable code.

[Input]
Mathematical formulation in LaTeX.

[Output]
Runnable Python code using `gurobipy`.

[Advanced Cognitive Directives]
Upon receiving the formulation of an optimization problem, the agent must operate in a concise but rigorous code-generation mode and follow these steps:

1. **Model-to-code mapping**: Identify all parameters, decision variables, domains, objective terms, explicit constraints, and implicit constraints such as non-negativity, then map each element into Gurobi model components.
2. **Structured implementation**: Generate a complete Python script using `gurobipy`, including imports, reusable function definitions, model creation, variable declarations, clearly separated constraint blocks, objective setting, and optimization.
3. **Solution handling**: Make the model runnable and reusable. Handle solver status properly, print the objective value using `Optimal value:`, and return the final results in a dictionary.

[Strict Format Constraints and Output Definition]

1. **Output form**: Output only the final Python code, enclosed in a single `python` code block, with no explanation before or after the code.
2. **Required code content**: The code must include the full solvable Gurobi model: imports, model construction, parameter usage, variable definitions, constraint additions, objective setting, optimization, and result output.
3. **Structure and reusability**: Follow PEP 8 style, keep different families of constraints in separate code sections for readability, and ensure any generated function is general-purpose, reusable, and returns a dictionary.
4. **Fidelity rules**: Only implement components required by the provided specifications and formulation. Do not add extra fields, omit required components, include usage examples, or invent specific data values. Preserve existing comments if present, except that LaTeX-style `%` comments should be converted to Python # comments.
5. **Constraint and output rules**: Each constraint must contain only one comparison operator, so any double-sided constraint must be split into separate constraints. The generated code must be compilable, and it must print the objective value using the exact label `Optimal value:`.

[Example Output Shape]

```
import gurobipy as gp
from gurobipy import GRB
def solve_instance(data):
  model = gp.Model(“optimization_model”)
  # Define variables
  # Add constraints
  # Set objective
  model.optimize()
  result = {“status”: model.Status,
“objective_value”: None, “variables”: {}}
  if model.SolCount > 0:
    result[“objective_value”] = model.ObjVal
    print(f“Optimal value: {model.ObjVal}”)
  return result
```

Extractor ⊕ Formulator Skill

[Metadata]
Name: extractor_formulator_agent
Description: Triggered for Operations Research problems. Guides the combined Extractor–Formulator agent to identify the core optimization components, build a coherent mathematical model, and emit a strict JSON formulation in the required field order.

[Input]
Raw OR problem description.

[Output]
Strict JSON format containing constraints, parameters, variables, and objective.

[Advanced Cognitive Directives]
Upon receiving a complex OR problem description, the agent must operate in a concise but rigorous extraction-plus-formulation mode and follow these steps:

1. **Problem typing and comprehension**: Build a concise internal description of the problem type, understand the optimization goal, and identify the key sets, entities, or stages before defining the model.
2. **Component extraction**: Extract all required parameters, decision variables, constraints, and the objective from the text as comprehensively as possible, including both common and less obvious constraints.
3. **Constraint auditing**: Organize the constraints carefully, distinguish their roles, and prepare a short label or abbreviation plus a concise explanation for each one whenever possible.
4. **Model formulation**: Convert the extracted components into a coherent mathematical model. Define variable domains, include implicit constraints such as non-negativity when needed, and ensure that the objective and constraints faithfully match the source text.
5. **Consistency verification**: Cross-check the original description against the extracted components and the formulation so that no required element is omitted, no unsupported parameter is invented, and the final JSON is logically complete and solvable.

[Strict Format Constraints and Output Definition]

1. **Output form**: Output only one complete and valid raw JSON object, directly parsable by Python `json.loads()`, with no markdown code fences and no extra commentary.
2. **Fixed top-level order**: The JSON object must contain exactly these top-level keys in this order: `constraints`, `parameters`, `variables`, and `objective`.
3. **Content requirements**: `constraints` must be comprehensive and listed in order. Each constraint entry should include a concise label or abbreviation, a short explanation, and a mathematical formulation whenever possible. `parameters` and `variables` must define the extracted modeling components clearly.
4. **Modeling rules**: New parameters must not be invented. If auxiliary variables are truly necessary for a correct formulation, include them inside `variables` and identify them clearly. Mathematical formulas should use LaTeX when applicable, and constraints may only use $=$, $\geq$, or $\leq$, never $>$ or $<$.

[Expected Output Schema]

```
{
  “constraints”: [
    {“id”: 1, “name”: “capacity”,
“description”: “Vehicles or facilities cannot
exceed their available capacity.”,
    “formulation”: “\\sum_{i \\in I}
\\textup{load}_{i,j} \\leq
\\textup{capacity}_{j}”}
  ],
  “parameters”: {“capacity”: {“type”:
“indexed parameter”, “definition”: “Available
capacity of resource j”}},
  “variables”: {“load”: {“type”:
“continuous”, “definition”: “Amount assigned
or transported on index pair (i, j)”}},
  “objective”: {“sense”: “minimize”,
“description”: “Minimize the total operating
cost.”,
  “formulation”: “\\min \\sum_{i \\in I}
\\sum_{j \\in J} \\textup{cost}_{i,j}
\\textup{load}_{i,j}”}
}
```

Formulator ⊕ Programmer Skill

[Metadata]
Name: formulator_programmer_agent
Description: Triggered for Operations Research problems after component extraction. Guides the combined Formulator–Programmer agent to keep the base agent's natural Markdown output style while adding deeper model checks, formulation discipline, and compile-safe Python code drafting.

[Input]
Raw operations research problem description and extracted components.

[Output]
Natural Markdown model audit, mathematical formulation, one Python implementation block, and brief verification or interpretation.

[Advanced Cognitive Directives]
Upon receiving an optimization problem and extracted components, the agent must preserve the base agent's natural Markdown style while making the formulation and code more careful, complete, and robust. It must not convert the answer into a rigid machine-readable JSON protocol. Follow these steps:

1. **Problem and model audit**: Understand the problem goal, extracted components, objective sense, available data, decision entities, and intended optimization structure. Treat extracted components as strong guidance, but verify them against the original problem text.
2. **Formulation construction**: Build the mathematical model step by step. Clearly define parameters, decision variables, domains, objective, and all relevant constraints, including implicit constraints such as non-negativity when needed. Do not invent unsupported parameters, data values, dimensions, or helper assumptions.
3. **Special structure handling**: If the problem includes recourse, staged decisions, linking constraints, scenario logic, or other multi-stage structure, represent

that logic explicitly in the formulation and code rather than leaving it implicit.

4. **Code implementation**: Translate the formulation into one high-quality Python implementation block. The code should be a clear Gurobi-style optimization draft that the downstream supervisor can normalize into the final evaluator entrypoint without changing solver families.

[Strict Format Constraints and Output Definition]

1. **Natural Markdown output**: Output should follow the same general style as the no-skill base agent: clear Markdown headings, concise bullets, readable formulas, and one Python implementation block.
2. **No JSON protocol**: Do not output hard protocol labels such as `formulation` and `python`. Do not output raw JSON objects or fenced JSON blocks.
3. **One fenced code block only**: The only fenced code block in the entire answer must be the final `python` implementation block.
4. **Python code rules**: The Python block should define the public solver function requested by the task context. If no target function name is visible, define `solve`. Do not define unrelated names such as `solve_instance` or problem-specific wrappers as the only public solver.
5. **Evaluator compatibility rules**: The public solver should be directly callable by the evaluator and should return the final objective value as one numeric value by default.

[Expected Output Schema]

Step 1: Problem and Model Audit
Extracted components checked; parameters; decision variables; objective; constraints.
Step 2: Mathematical Model
Write the objective and constraints in readable mathematical form, including domains, linking constraints, non-negativity, integrality, and transformed constraints.
Step 3: Python Implementation
One `python` implementation block defining `solve`.
Step 4: Verification and Interpretation
Briefly verify that the model and code match the original problem and extracted components.

Extractor ⊕ Formulator ⊕ Programmer Skill

[Metadata]
Name: extractor_formulator_programmer_agent
Description: Triggered for Operations Research problems. Guides the full Extractor–Formulator–Programmer agent to keep the base agent's natural Markdown output style while adding deeper reasoning, formulation checks, and compile-safe Python code drafting.

[Input]
Raw operations research problem description.

[Output]
Natural Markdown problem analysis, mathematical formulation, one Python implementation block, and brief verification or interpretation.

[Advanced Cognitive Directives]
Upon receiving an optimization problem, the agent must preserve the base agent's natural Markdown style while making the reasoning more careful, complete, and robust. It must not convert the answer into a rigid machine-readable JSON protocol. Follow these steps:

1. **Problem understanding and planning**: Identify the problem type, objective direction, decision entities, relevant quantities, and all explicit or implicit constraints. Keep this analysis concise and aligned with the original problem text.
2. **Component extraction**: Extract parameters, decision variables, constraints, and the objective using Markdown bullets or compact tables. Do not use raw JSON objects or fenced JSON blocks.
3. **Mathematical formulation**: Convert the extracted components into a coherent linear or integer-linear mathematical model using ordinary Markdown and LaTeX-style expressions. Include variable domains, non-negativity, integrality, transformed constraints, and objective sense.
4. **Code implementation**: Write one high-quality Python implementation block. The code should be a clear Gurobi-style optimization draft that the downstream supervisor can normalize into the final evaluator entrypoint without changing solver families.
5. **End-to-end verification**: Briefly verify that the extraction, formulation, and code all match the source problem.

[Strict Format Constraints and Output Definition]

1. **Natural Markdown output**: Output should follow the same general style as the no-skill base agent: clear Markdown headings, concise bullets, readable formulas, and one Python implementation block.
2. **No JSON protocol**: Do not output sections named `extracted_components`, `formulation`, and `python` as hard protocol labels. Do not output raw JSON objects or fenced JSON blocks.
3. **One fenced code block only**: The only fenced code block in the entire answer must be the final `python` implementation block.
4. **Python code rules**: The Python block should define the public solver function requested by the task context. If no target function name is visible, define `solve`. Do not define unrelated names such as `solve_instance` or problem-specific wrappers as the only public solver.
5. **Evaluator compatibility rules**: The public solver should return the final objective value as one numeric value by default. It should not return assignments, dictionaries, tuples, solver or model objects, or extra metadata unless the task explicitly asks for structured output.

[Expected Output Schema]

Step 1: Problem Analysis
Parameters; decision variables; objective; constraints.
Step 2: Mathematical Model
Write the objective and constraints in readable mathematical form, including domains and transformed constraints.
Step 3: Python Implementation
One `python` implementation block defining `solve`.

Step 4: Verification and Interpretation
Briefly verify that the model and code match the original problem.

F Experimental Token and Time Usage

To evaluate computational cost and runtime efficiency, we measured the average per-problem token consumption and execution time across 12 architectures, four datasets, and three LLMs, averaging results over five independent runs(Table 10–12). Values are normalized against the Base framework within each dataset to facilitate a clearer and more pronounced performance comparison across different architectural topologies.

Token expenditure correlates positively with topological complexity. Category A (Phase Fusion) reduces token usage by merging contexts and eliminating redundant prompt transmissions. Conversely, Category B and C (Parallelization) significantly inflate costs due to *Context Duplication*, as the problem context should be repeatedly broadcast to independent sub-agents.

Execution time exhibits non-linear dynamics. Category A consistently executes faster by shortening the critical path and minimizing Time-To-First-Token(TTFT) delays. In contrast, Category B and C show high variance dictated by the trade-off between *Synchronization Overhead* and *Parallel Speedup*. For simple tasks, "Merge" node delays and straggler effects outweigh parallelization benefits, increasing latency. However, for complex problems, the substantial time saved through concurrent decoding of heavy payloads easily offsets synchronization delays, yielding faster execution than the Base framework.

Table 10: Relative average per-problem token usage and runtime of DeepSeek-V3.2. For each dataset, both token and runtime costs are normalized by the corresponding Base architecture cost.

Architecture	NL4Opt		NLP4LP		IndustryOR		ComplexOR	
	Token	**Time**	**Token**	**Time**	**Token**	**Time**	**Token**	**Time**
Base DAG								
Base	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
Category A: Phase Fusion								
A1	0.42x	0.33x	0.40x	0.35x	0.44x	0.45x	0.56x	0.41x
A2	0.55x	0.54x	0.52x	0.67x	0.74x	0.64x	0.71x	0.79x
A3	0.57x	0.46x	0.63x	0.50x	0.75x	0.35x	0.75x	0.56x
Category B: Phase Parallelization								
B1	1.22x	0.96x	1.15x	0.85x	1.25x	0.95x	1.13x	0.87x
B2	1.14x	0.86x	1.23x	0.74x	1.15x	0.63x	1.16x	0.65x
B3	1.07x	0.79x	1.19x	0.88x	1.08x	0.91x	1.26x	0.75x
B4	1.25x	0.72x	1.31x	0.78x	1.26x	0.91x	1.39x	0.58x
B5	1.11x	1.22x	1.20x	1.25x	1.19x	1.22x	1.39x	1.28x
B6	1.29x	1.17x	1.13x	1.28x	1.25x	1.27x	1.19x	1.03x
B7	1.26x	1.39x	1.22x	1.27x	1.23x	1.40x	1.31x	1.37x
Category C: Component-wise Parallelization								
C1	1.23x	0.98x	1.19x	0.93x	1.25x	1.02x	1.35x	0.96x

Table 11: Relative average per-problem token usage and runtime of GPT-5.4. For each dataset, both token and runtime costs are normalized by the corresponding Base architecture cost.

Architecture	NL4Opt		NLP4LP		IndustryOR		ComplexOR	
	Token	**Time**	**Token**	**Time**	**Token**	**Time**	**Token**	**Time**
Base DAG								
Base	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
Category A: Phase Fusion								
A1	0.57x	0.64x	0.56x	0.74x	0.47x	0.92x	0.67x	0.78x
A2	0.68x	0.73x	0.66x	0.72x	0.76x	0.61x	0.67x	0.65x
A3	0.93x	0.59x	0.80x	0.59x	0.85x	0.74x	0.81x	0.80x
Category B: Phase Parallelization								
B1	1.19x	0.97x	1.17x	0.99x	1.22x	1.18x	1.28x	0.92x
B2	1.38x	0.66x	1.18x	0.65x	1.30x	0.85x	1.34x	0.81x
B3	1.20x	1.13x	1.35x	1.15x	1.26x	1.04x	1.43x	1.07x
B4	1.23x	0.65x	1.21x	0.85x	1.17x	0.72x	1.30x	0.80x
B5	1.34x	1.08x	1.33x	1.16x	1.23x	1.06x	1.38x	1.11x
B6	1.40x	0.78x	1.38x	0.82x	1.51x	0.83x	1.68x	0.83x
B7	1.25x	1.05x	1.22x	1.07x	1.20x	0.94x	1.20x	0.96x
Category C: Component-wise Parallelization								
C1	1.15x	0.99x	1.23x	0.94x	1.17x	0.82x	1.22x	0.89x

Table 12: Relative average per-problem token usage and runtime of Qwen3.6-27B. For each dataset, both token and runtime costs are normalized by the corresponding Base architecture cost.

Architecture	**NL4Opt**		**NLP4LP**		**IndustryOR**		**ComplexOR**	
	Token	**Time**	**Token**	**Time**	**Token**	**Time**	**Token**	**Time**
Base DAG								
Base	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
Category A: Phase Fusion								
A1	0.63x	0.64x	0.62x	0.56x	0.52x	0.74x	0.55x	0.50x
A2	0.61x	0.81x	0.60x	0.80x	0.51x	0.65x	0.56x	0.57x
A3	0.76x	0.63x	0.76x	0.52x	0.79x	0.55x	0.73x	0.72x
Category B: Phase Parallelization								
B1	1.14x	0.91x	1.14x	0.89x	1.49x	0.83x	1.27x	0.70x
B2	1.20x	0.75x	1.37x	0.77x	1.34x	0.68x	1.39x	0.58x
B3	1.25x	1.12x	1.38x	1.14x	1.38x	1.03x	1.47x	0.92x
B4	1.25x	0.78x	1.35x	0.69x	1.28x	0.77x	1.18x	0.87x
B5	1.24x	1.15x	1.25x	1.20x	1.29x	1.08x	1.20x	1.01x
B6	1.54x	1.08x	1.55x	1.05x	1.85x	1.04x	1.58x	1.13x
B7	1.24x	1.03x	1.21x	1.05x	1.33x	1.17x	1.20x	1.29x
Category C: Component-wise Parallelization								
C1	1.26x	0.98x	1.23x	1.03x	1.14x	1.13x	1.16x	0.97x