GPBENCH: BENCHMARKING LLMS FOR ORCHESTRATING MULTI-STAGE GRAPH ANALYSIS WORKFLOW

Anonymous authors
Paper under double-blind review

ABSTRACT

Real-world graph analysis often requires composing multiple graph algorithms into a workflow, where graph data are processed through dependent stages to solve a single analytical problem. Manually constructing such workflows requires substantial expertise in graph algorithms and their execution dependencies. Large language models (LLMs) provide a promising approach to automating this process by translating high-level user requests into graph-analysis workflows. However, doing so requires LLMs to jointly decompose an analytical problem into concrete operations, select an appropriate algorithm for each operation, and resolve algorithm parameters and cross-stage dependencies, making workflow orchestration substantially more challenging than conventional graph-analysis tasks evaluated for LLMs. Existing benchmarks primarily focus on general planning, single-step tool use, or graph problems with a single computational objective, leaving multi-stage graph-analysis workflow orchestration insufficiently evaluated. We present **GPBench**, a benchmark evaluating three core capabilities: workflow decomposition, algorithm selection, and dependency and parameter resolution. GPBench contains 1,000 multi-stage graph-analysis problems across five representative workflow topologies. Experiments show that Graph Planner achieves 82.7% accuracy on workflow decomposition. A hierarchical graph-algorithm knowledge base improves strict Top-1 algorithm-selection accuracy from 71.7% to 82.9%, while models achieve up to 90.0% workflow-instantiation accuracy given the correct algorithms and their parameter descriptions. However, end-to-end composition causes substantial performance degradation due to error propagation across stages, indicating that improving individual stages alone is insufficient. We further introduce targeted improvements to the major error-prone components to improve end-to-end workflow orchestration. Code and data are available at `https://anonymous.4open.science/r/GPBench-F305/`.

1 INTRODUCTION

Graph analysis is widely used in domains such as social networks, financial systems, biological networks, and infrastructure (Easley et al., 2010; Gross et al., 2018), with analyses typically relying on specialized graph algorithms and computational libraries (Hagberg et al., 2007; Bonald et al., 2020; Perret et al., 2019). In real-world settings, an analytical objective often cannot be addressed by a single graph algorithm, but instead requires multiple graph operations connected through data dependencies, where intermediate results from earlier stages determine subsequent computations (Li et al., 2024a;b; Wei et al., 2026; Tan et al., 2026). Constructing such workflows manually therefore requires substantial expertise in selecting appropriate graph algorithms and determining their execution dependencies across stages. Recent advances in large language models (LLMs) in natural-language understanding, code generation, and tool use (Yao et al., 2022; Schick et al., 2023; Qin et al., 2024; Patil et al., 2024) provide a promising approach to automating this process by translating high-level natural-language questions into multi-stage graph-analysis workflows. Tool orchestration and graph-specific API invocation have also been explored (Shen et al., 2023; Song et al., 2023; Zhang, 2023). However, doing so requires more than solving an individual graph problem or invoking a single tool. As illustrated in Figure 1, the LLM must jointly identify the required analytical operations and their dependencies, select an appropriate graph algorithm for each operation,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Comparison with representative benchmarks by evaluation scope. ✓ denotes dedicated evaluation, △ partial or related coverage, and ✗ no dedicated evaluation.

Benchmark	Graph Specific	Multi-stage	Workflow Structure	Graph Alg. Selection	Dependency / Parameter	Stage-wise Diagnosis
NLGraph (Wang et al., 2023)	✓	✗	✗	✗	✗	✗
GraphInstruct (Luo et al., 2024)	✓	✗	✗	✗	✗	✗
GraphArena (Tang et al., 2025)	✓	✗	✗	✗	✗	✗
GraphOmni (Xu et al., 2026b)	✓	✗	✗	✗	✗	✗
GrAlgoBench (Zhang et al., 2026)	✓	✗	✗	✗	✗	✗
ProGraph (Li et al., 2024a)	✓	✗	✗	✗	✗	✗
TaskBench (Shen et al., 2024)	✗	✓	✓	✗	△	✓
WorFBench (Qiao et al., 2025)	✗	✓	✓	✗	✗	✗
GABench (Tan et al., 2026)	✓	✓	△	△	✗	✗
GPBench	✓	✓	✓	✓	✓	✓

and resolve algorithm parameters together with cross-stage intermediate-result dependencies. These capabilities are therefore essential for LLMs to orchestrate multi-stage graph-analysis workflows.

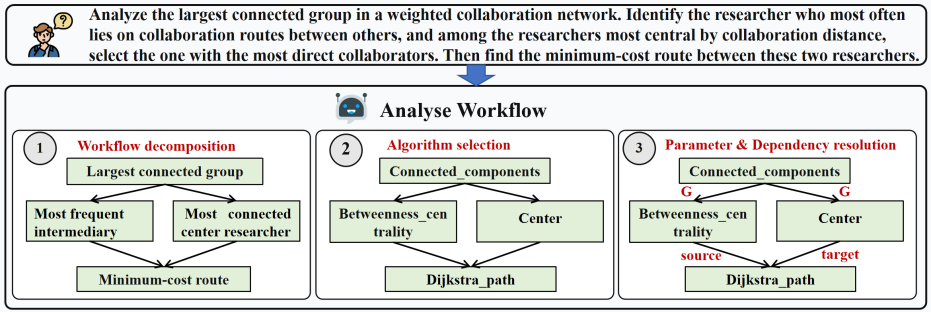

Figure 1: From graph-analysis questions to multi-algorithm workflows.

Existing benchmarks evaluate LLM capabilities in graph analysis and workflow planning from multiple perspectives. Graph-oriented benchmarks such as NLGraph (Wang et al., 2023), GraphInstruct (Luo et al., 2024), GraphArena (Tang et al., 2025), GraphOmni (Xu et al., 2026b), GrAlgoBench (Zhang et al., 2026), and ProGraph (Li et al., 2024a), together with related studies on graph representation, instruction following, pattern understanding, and generalization (Guo et al., 2023; Fatemi et al., 2024; Chen et al., 2024; Zhang et al., 2024; Dai et al., 2025; Yuan et al., 2025), focus on graph-analysis or graph-theoretic tasks in which the target computation is specified by the question. They therefore assess whether LLMs can perform a given graph computation, rather than construct a multi-stage, multi-algorithm workflow from a high-level analytical goal. GABench (Tan et al., 2026) extends evaluation to multi-step agentic graph analysis, but its metrics mainly evaluate tool-use trajectories and final task outcomes, making it difficult to identify which planning decision causes a failure. On the other hand, TaskBench (Shen et al., 2024), WorFBench (Qiao et al., 2025), and related workflow benchmarks (Xiao et al., 2024) evaluate aspects of task decomposition, tool selection, and workflow structure, but are designed for general task automation and workflow generation rather than graph-specific planning. They therefore do not evaluate graph-algorithm selection or the input/output constraints and intermediate-result dependencies among graph algorithms. Table 1 summarizes the coverage of representative benchmarks across these dimensions. Overall, existing work still lacks a graph-specific, stage-wise evaluation of multi-stage graph-analysis planning, making it difficult to precisely localize error sources within a complete workflow.

To address this gap, we introduce **GPBench**, a benchmark for evaluating LLMs on constructing multi-stage graph-analysis workflows from high-level natural-language questions. We decompose the orchestration process into three tasks. **Problem-to-Workflow Decomposition** evaluates whether a model can decompose a problem into analytical subtasks and construct their dependency DAG. **Subtask-to-Algorithm Matching** evaluates whether the model can select an applicable graph algorithm for each subtask. **Algorithm Dependency and Parameter Resolution** evaluates whether

the model can determine algorithm parameters and resolve cross-stage data dependencies. GPBench contains 1,000 graph-analysis problems, each involving 2–5 graph algorithms arranged in one of five representative workflow topologies: sequential, parallel, divergence, convergence, and diamond. For each problem, we provide a validated reference workflow specifying the subtasks and their dependencies, selected algorithms, algorithm parameters, and cross-stage data dependencies. GPBench evaluates the three tasks independently and further evaluates their end-to-end composition, enabling fine-grained analysis of errors throughout the workflow orchestration process.

Our experiments reveal stage-specific limitations in existing methods and error accumulation that reduces end-to-end accuracy. Workflow decomposition suffers from inaccurate or incomplete identification of analytical operations. To address this, we propose GraphPlanner, which guides decomposition using structured graph-analysis knowledge, improving Workflow Accuracy from 74.8% to 82.7%. Algorithm matching requires distinguishing functionally similar algorithms. We therefore propose Hierarchical RAG, which combines category routing with within-category semantic differentiation, improving Top-1 Overall Accuracy on GPT-5.5 from 71.7% to 82.9%. For parameter and dependency resolution, direct LLM inference already achieves 90.0% Overall Accuracy given reference algorithms, with residual errors concentrated in interpreting downstream parameter roles. Despite these strong stage-wise results, composing the three stages yields only 48.8% exact end-to-end accuracy. Further analysis reveals that some intermediate-result mismatches preserve the information required downstream. We propose *Intermediate-Result Alignment* (IRA) to assess whether intermediate outputs satisfy downstream input requirements, allowing predefined deterministic transformations that preserve this information. Under IRA, end-to-end accuracy reaches 53.1%, demonstrating the importance of accounting for semantic alignment across stages. These results highlight the remaining challenges in reliable multi-stage graph-analysis planning.

2 GPBENCH

GPBench evaluates whether LLMs can transform high-level natural-language graph-analysis problems into complete multi-stage graph-algorithm workflows. Given a problem q, we represent its analytical structure as a directed acyclic graph $D = (V, E)$, where $V = v_1, \ldots, v_n$ denotes atomic analytical subtasks with explicit computational objectives and E their dependencies. GPBench evaluates this process through three stages: *Problem-to-Workflow Decomposition*, *Subtask-to-Algorithm Matching*, and *Algorithm Dependency and Parameter Resolution*, with reference annotations provided for each stage.

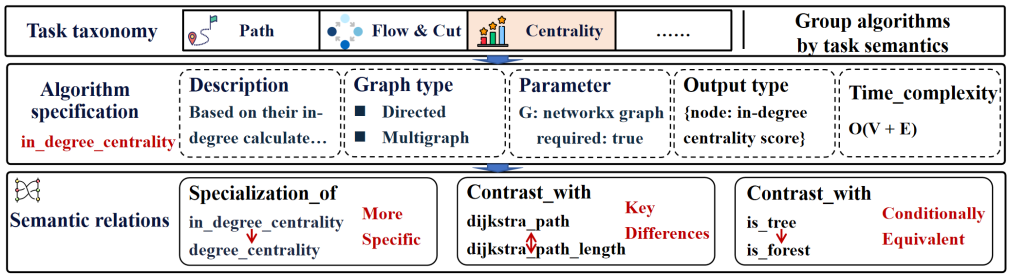

Figure 2: Structure of the hierarchical graph algorithm knowledge base.

2.1 BENCHMARK CONSTRUCTION

Hierarchical Graph Algorithm Knowledge Base. The hierarchical knowledge base progressively narrows the candidate space from task semantics to algorithm-level constraints, providing structured knowledge for reference-workflow construction and algorithm matching. We build it primarily from NetworkX documentation (Hagberg et al., 2007) and the algorithm literature referenced therein. As shown in Figure 2, it consists of three layers. The first is a *task taxonomy* that groups algorithms by their primary computational objectives, such as *Flow& Cut*, *Centrality*, and *Path*, restricting each analytical operation to relevant candidates. The second contains *algorithm specifications*, including descriptions, supported graph types, parameters, output types, and time complexity, for assessing applicability and checking cross-stage dataflow compatibility. The third

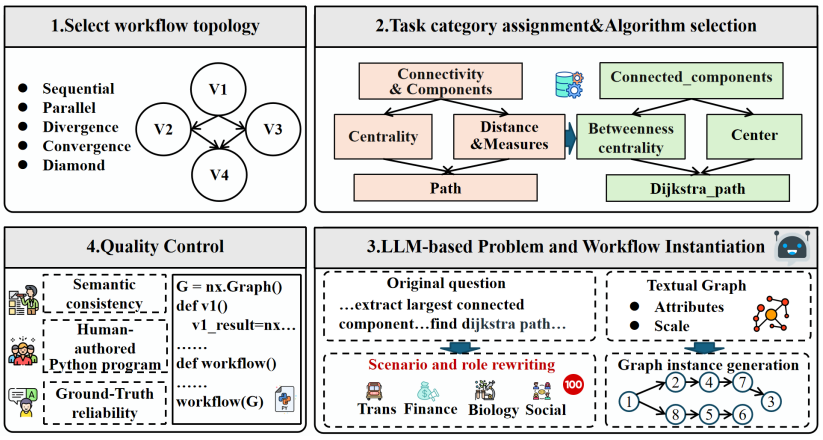

Figure 3: **GPBench construction pipeline.** We define the workflow structure, select task categories and algorithms, and generate natural-language questions and graph instances.

captures semantic relations within each category, including algorithmic hierarchy and functional distinctions, to further distinguish closely related algorithms with different applicability conditions or outputs. The knowledge base covers 143 algorithms across 16 task categories; the complete taxonomy, specification fields, and relation definitions are provided in Appendix A.

Construction Process. Based on this knowledge base, we constructed 1,000 multi-stage graph-analysis problems, comprising a total of 3,360 algorithmic subtasks. Each problem consists of 2–5 algorithmic stages and adopts one of five representative workflow topologies: sequential, parallel, divergence, convergence, and diamond. These topologies reflect common workflow patterns such as sequencing and synchronization (van der Aalst et al., 2003). The specific workflow topologies and scale distributions are shown in Table 2. As shown in Figure 3, we first determine the workflow topology type and the number of stages, then assign task categories and select specific algorithms based on the dependency constraints within the topology. Candidate workflows are checked for compatibility based on graph properties, algorithm interfaces, and data flows between stages. For natural-language realization, we follow the role-play contextualization strategy of ProGraph (Li et al., 2024a). We first use an LLM to verbalize the sampled workflow as a graph-analysis question. The question is then contextualized with a professional role and application setting while preserving the analytical objectives, graph properties, parameter values, and cross-stage dependencies. Finally, a graph instance is generated from the graph properties and parameter constraints extracted from the constructed question and the selected algorithms.

Quality Control. To ensure reliable reference annotations, we validate each benchmark instance at both the semantic and execution levels. We first manually check consistency among the natural-language question, reference workflow, and graph instance, ensuring that the question captures the required analytical operations, parameter conditions, and cross-stage dependencies, while the graph satisfies the applicability requirements of all reference algorithms. We then implement a human-authored Python reference program for each problem from the reference algorithm DAG, parameters, and cross-stage dependencies, and execute the complete workflow in topological order. An instance is retained only if all stages execute successfully, intermediate results are correctly passed to their designated downstream inputs, and the program produces a valid final output. Further details on workflow topologies, algorithm-composition constraints, graph-instance generation, and validation are provided in Appendix A.3.

2.2 Benchmark Tasks and Evaluation

Each problem is annotated with an analytical-operation DAG, algorithms, parameters, and cross-stage dependencies, which define the three GPBench tasks. As illustrated by the diamond workflow in Figure 1, Task 1 recovers the operations and dependency structure, Task 2 selects an algorithm per operation, and Task 3 resolves parameters and cross-stage dependencies.

Table 2: Distribution of workflow topologies and algorithm counts in GPBench.

Topology	2 stages	3 stages	4 stages	5 stages	Total
Sequential	100	70	30	20	220
Parallel	100	70	30	20	220
Divergence	0	100	100	20	220
Convergence	0	100	100	20	220
Diamond	0	0	100	20	120
Total	200	340	360	100	1,000

Task 1: Problem-to-Workflow Decomposition. Given a natural-language problem, the model identifies the required analytical operations and constructs their dependency DAG. Because predicted and reference nodes may differ in wording or valid decomposition granularity, an LLM-based semantic evaluator categorizes each node alignment as *exact*, *split*, *incomplete*, *merged*, or *spurious*. We report **Node-set Accuracy**, **Edge-set Accuracy**, and **Workflow Accuracy**, where Workflow Accuracy requires both the predicted nodes and dependencies to be completely correct.

Task 2: Subtask-to-Algorithm Matching. Given a ground-truth subtask, the model selects the corresponding graph algorithm from the candidate set. We report **Subtask Accuracy@**k and **Overall Accuracy@**k for $k \in \{1, 3, 5\}$. The former measures whether the reference algorithm appears among the Top-k candidates for each subtask, whereas the latter requires all subtasks within a problem to be matched correctly. For hierarchical methods, we additionally report **Category Accuracy**.

Task 3: Algorithm Dependency and Parameter Resolution. Given the natural-language problem, selected algorithms, and their parameter descriptions, the model determines static parameters and resolves cross-stage dependencies. **Static Parameter Accuracy** requires all static parameter assignments within a problem to be correct, while **Dynamic Dependency Accuracy** requires every cross-stage data binding to connect the correct upstream result to the intended downstream parameter. **Overall Accuracy** requires both components to be completely correct. Detailed task formulations and evaluation protocols are provided in Appendix B.

3 Experimental Evaluation

We evaluate models across the three stages of GPBench: workflow decomposition, subtask-to-algorithm matching, and parameter/dependency resolution. Experiments use the full 1,000-problem benchmark. We first describe the setup and then analyze each stage.

3.1 Experimental Setup

Environment. Experiments are conducted on a server with two Intel Xeon Silver 4316 CPUs, 512GB DRAM per CPU, and two NVIDIA RTX A6000 GPUs with 48GB GDDR6 memory, running Ubuntu 20.04 LTS and CUDA 12.4.

LLM backends. We use GPT-5.5 (Singh et al., 2025) and DeepSeek-V4-Flash (Xu et al., 2026a) as representative proprietary and open-weight LLM backends, respectively. The prompt templates are provided in Appendix D.

3.2 Problem-to-Workflow Decomposition

This experiment evaluates whether a model can identify the analytical operations required by a natural-language graph-analysis question and construct their dependency DAG. We further compare the Task 1 automatic evaluator with human evaluation to assess its reliability; the detailed evaluation procedure and results are provided in Appendix B.1.3.

Baselines. We compare three settings. **LLM-CoT** (Wei et al., 2022) progressively identifies the required analytical operations and their dependencies along a single reasoning path. **LLM-ToT** (Yao

Table 3: Results on problem-to-workflow decomposition.

LLM Backend	Method	Node Acc.	Edge Acc.	Workflow Acc.	Tokens
DeepSeek-V4-Flash	LLM-CoT	72.90	**95.30**	71.00	**4,804.34**
	LLM-ToT	78.70	92.10	74.20	20,715.52
	LLM-GoT	79.00	91.70	74.80	11,301.69
	Graph Planner	**85.20**	**95.30**	**82.70**	5,632.97
GPT-5.5	LLM-CoT	75.10	94.50	71.80	**2,708.72**
	LLM-ToT	55.80	87.80	51.20	4,657.13
	LLM-GoT	67.70	89.70	63.40	4,523.63
	Graph Planner	**85.00**	**96.30**	**82.70**	2,810.28

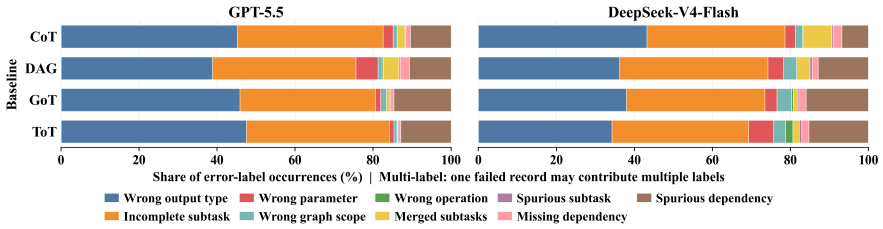

Figure 4: Distribution of Task 1 error types across methods and LLM backends. Each bar is normalized by the total number of error-label occurrences; a failed workflow may contain multiple error labels.

et al., 2023) explores multiple candidate planning branches and selects among alternative intermediate states, whereas **LLM-GoT** (Besta et al., 2024) organizes intermediate states as a graph, allowing different reasoning branches to be connected and combined during planning.

Results. As shown in Table 3, CoT achieves 71.00% and 71.80% Workflow Accuracy on DeepSeek-V4-Flash and GPT-5.5, respectively. ToT and GoT employ tree- or graph-structured reasoning, expanding and integrating multiple intermediate planning states to determine the required analytical operations and their dependencies. They achieve 74.20% and 74.80% Workflow Accuracy on DeepSeek-V4-Flash, respectively, but decrease to 51.20% and 63.40% on GPT-5.5, while incurring substantially higher token costs. These results indicate that increasing reasoning complexity does not consistently improve workflow decomposition accuracy across different backends. Figure 4 further shows that the dominant errors fall into two categories: *Incomplete Subtask*, where the required analytical operations are not fully identified, and *Wrong Output Type*, where the expected output of an operation is incorrectly specified. Based on this observation, we propose Graph Planner, which uses graph analysis operations as the basic unit for problem decomposition and sub-task planning. Specifically, GraphPlanner first identifies candidate sub-tasks and their expected inputs and outputs, then matches graph analysis operations based on computational objectives and input-output types. Based on this, it refines the task description and adjusts the decomposition granularity so that each sub-task corresponds to a specific graph analysis operation. Finally, dependencies are established based on the input-output relationships between subtasks, thereby forming a more comprehensive and semantically unambiguous analysis workflow. Graph Planner increases Node Accuracy to 85.20% on DeepSeek-V4-Flash and 85.00% on GPT-5.5, while achieving 82.70% Workflow Accuracy on both backends, outperforming the three general planning methods.

Findings. Task 1 indicates that the main challenge in workflow decomposition lies in fully and accurately identifying the analytical operations required to solve a problem and their expected outputs. Compared to expanding the number of intermediate planning states, using graph-based analytical operations as the basic planning unit and organizing subtasks around their computational goals and outputs can more effectively improve the reliability of workflow decomposition.

Table 4: Results on subtask-to-algorithm matching.

LLM Backend	Method	k	Category Acc.	Subtask Acc.	Overall Acc.	Tokens
DeepSeek-V4-Flash	Flat KB	1	–	88.57	66.40	119,432.69
		3	–	96.04	87.80	129,155.88
		5	–	98.72	96.10	135,694.59
	Hierarchical RAG	1	96.60	**92.92**	**78.90**	31,097.69
		3	96.50	**97.56**	**93.50**	46,313.62
		5	97.40	**98.93**	**97.00**	69,596.88
GPT-5.5	Flat KB	1	–	90.57	71.70	117,983.31
		3	–	97.53	92.20	119,030.50
		5	–	98.75	**96.90**	128,108.05
	Hierarchical RAG	1	97.40	**94.29**	**82.90**	26,107.23
		3	97.50	**98.57**	**95.30**	26,771.01
		5	97.80	**99.08**	**96.90**	27,405.31
G1-CoT-SFT-7B	Direct Prediction	1	–	54.79	16.60	4,120.14
		3	–	57.98	18.60	5,063.52
		5	–	65.30	28.50	5,501.00

3.3 Subtask-to-Algorithm Matching

This experiment evaluates whether an LLM can map the analytical intent of each subtask to the concrete graph algorithm that solves it.For each subtask, we identify candidate algorithms according to the input graph type and computational objective, and use the algorithm that satisfies the corresponding constraints with preferable time complexity as the ground-truth label.

Baselines. We compare three settings. **Flat KB** directly provides the LLM with all 143 algorithms in the knowledge base and their associated knowledge for candidate ranking. **Hierarchical RAG** first identifies the task category of each subtask and then ranks algorithms using only the corresponding algorithm descriptions and semantic relations. **G1-CoT-SFT-7B** (Guo et al., 2025) is a graph-reasoning model.

Results. As shown in Table 4, Flat KB achieves Top-1 Overall Accuracy of 66.40% and 71.70% on DeepSeek-V4-Flash and GPT-5.5, respectively, whereas G1-CoT-SFT-7B reaches only 16.60%. G1 is mainly trained for reasoning over concrete graph problems, while Task 2 requires distinguishing specific algorithms from high-level analytical semantics, which likely limits its transfer to this setting. As k increases, Flat KB improves substantially, reaching 96.10% and 96.90% Overall Accuracy at Top-5. This gap indicates that the correct algorithm is often recognized as a relevant candidate but remains difficult to rank first among functionally similar algorithms. Our hierarchical knowledge base addresses this issue by first routing each subtask according to its computational objective to reduce the candidate space, and then using within-category relations, such as specialization and contrast, to distinguish algorithms with similar functionality but different applicability conditions or outputs. This raises Top-1 Overall Accuracy to 78.90% and 82.90%, while reducing token usage by approximately 74.0% and 77.9%. Category routing alone achieves 96.60% and 97.40% Top-1 Accuracy. Moreover, removing within-category semantic relations decreases Overall Accuracy@1 by 6.20 and 5.90 percentage points. Table 5 shows that candidate-space reduction alone is insufficient and that modeling fine-grained semantic differences further improves algorithm matching.

Findings. Task 2 shows that effective algorithm selection benefits from a hierarchical decision structure that progressively connects high-level task semantics to specific algorithms while explicitly modeling differences among similar candidates. The hierarchical knowledge base decomposes global algorithm selection into task grounding and semantic discrimination: the former grounds high-level analytical semantics into a concrete graph-analysis task, while the latter uses semantic relations to distinguish functionally similar algorithms. This coarse-to-fine decision process provides a more effective basis for fine-grained algorithm selection.

Table 5: Ablation of semantic relations in Hierarchical RAG.

LLM Backend	Variant	Subtask Acc.			Overall Acc.		
		@1	@3	@5	@1	@3	@5
DeepSeek-V4-Flash	w/o Semantic Relations	90.95	96.19	96.46	72.70	88.00	88.70
	Full	**92.92**	**97.56**	**98.93**	**78.90**	**93.50**	**97.00**
GPT-5.5	w/o Semantic Relations	91.52	97.38	98.01	77.00	93.30	95.20
	Full	**94.29**	**98.57**	**99.08**	**82.90**	**95.30**	**96.90**

Table 6: Results on algorithm dependency and parameter resolution.

System	Static Acc.	Dynamic Acc.	Overall Acc.	Tokens
DeepSeek-V4-Flash	94.90	92.20	87.70	11248.15
GPT-5.5	**95.00**	**94.70**	**90.00**	3222.74
GraphForge	50.10	43.30	22.60	2808.64

3.4 Algorithm Dependency and Parameter Resolution

This experiment evaluates whether the model can further parse the static parameters required for algorithm execution and identify cross-stage dependencies between the outputs of upstream algorithms and the inputs of downstream algorithms, given that the specific algorithms have been determined and their corresponding parameter descriptions provided.

Baselines. We compare two approaches. **Direct LLM Extraction** predicts static parameters and cross-stage dependencies directly from the natural-language problem and the provided algorithms (Li et al., 2023; Patil et al., 2025; Shen et al., 2024; Qiao et al., 2025). **GraphForge** is our implementation of GraphTool-Instruction (Wang et al., 2025) and serves as a graph-specific baseline for parameter extraction.

Results. As shown in Table 6, the overall accuracy rates for DeepSeek-V4-Flash and GPT-5.5 are 87.70% and 90.00%, respectively, whereas GraphForge achieves only 22.60%. GraphForge primarily targets parameter extraction for explicit single-tool calls; however, Task 3 requires the joint inference of parameters and cross-stage binding relationships based on natural language queries, resulting in Static Accuracy and Dynamic Accuracy rates of only 50.10% and 43.30%, respectively. The Dynamic Accuracy of all three systems is lower than their Static Accuracy, indicating that—even when the algorithms and their parameter descriptions are provided—parsing cross-stage dependencies from natural language queries remains more challenging than determining static parameters for individual algorithms. Figure 5(a) further pinpoints the source of this gap: matching accuracy drops further when dependency matching also requires consistency in downstream input parameters. Figure 5(b) illustrates that, for graph dependencies, there is typically a direct semantic correspondence between upstream results and downstream graph inputs; in contrast, data dependencies are often described only through high-level semantics in the natural language query, requiring the model to map these semantics to specific downstream parameters, which makes binding errors more likely. Furthermore, GPT-5.5 utilizes approximately 71% fewer tokens than DeepSeek-V4-Flash, demonstrating superior token usage efficiency.

Findings. Task 3 shows that, even with the algorithms, parameter descriptions, and workflow DAG provided, concrete cross-stage bindings still require interpretation of the problem semantics. Graph dependencies typically have clearer correspondence, whereas data dependencies require finer-grained understanding of parameter roles, making them more prone to binding errors.

3.5 End-to-End Evaluation

We compose **Graph Planner**, **Hierarchical KB (Top-1)**, and **Direct LLM Extraction**, using GPT-5.5 throughout the pipeline. Each stage directly receives the predictions from the preceding stage, allowing us to evaluate how the three planning stages compose within a complete workflow.

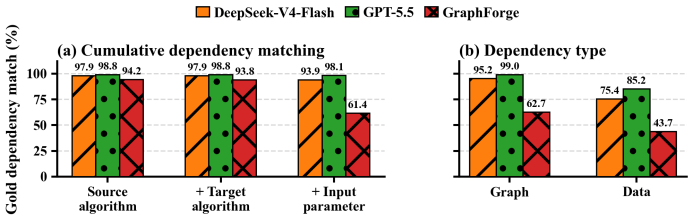

Figure 5: Cross-stage dependency diagnostics. (a) Recovery from upstream producer to downstream typed receiver. (b) Direct graph-object handoffs versus semantically inferred handoffs.

Table 7: End-to-end planning accuracy under exact matching and Intermediate-Result Alignment (IRA). Stage accuracy is conditional on preceding stages passing.

Stage	Exact Matching		IRA	
	Stage Acc.	Cumulative Acc.	Stage Acc.	Cumulative Acc.
Task 1	82.70%	82.70%	82.70%	82.70%
Task 2	72.67%	60.10%	80.17%	66.30%
Task 3	81.20%	**48.80%**	80.09%	**53.10%**

Results. As shown in Table 7, under strict reference-based evaluation, end-to-end accuracy decreases from 82.70% after Task 1 to 60.10% after Task 2, and finally to 48.80%. This shows that strong performance at individual stages does not directly translate into high end-to-end accuracy, since multiple planning decisions must be correct within the same workflow. To further analyze how local prediction differences affect subsequent stages, we examine the propagation of intermediate results across the workflow. Because intermediate computational results are passed between stages, selecting an algorithm different from the reference does not necessarily remove the information required by subsequent computations. If the predicted intermediate result preserves the required computational semantics, data type, and graph scope, the discrepancy may still support subsequent analysis.Based on this observation, we introduce **Intermediate-Result Alignment (IRA)**, which evaluates whether intermediate-result semantics satisfy downstream input requirements and allows deterministic transformations that preserve the information required by later stages. Detailed alignment criteria are provided in Appendix B.5. IRA increases the final end-to-end accuracy from 48.80% to 53.10%, showing that some predictions that differ from the reference algorithm still preserve the information required for subsequent computation.

Findings. IRA further improves end-to-end planning accuracy by identifying intermediate results that preserve downstream-required information, demonstrating the benefit of cross-stage semantic preservation. However, the remaining gap from isolated task performance indicates that multi-stage planning still requires stronger cross-stage coordination to maintain consistency among analytical objectives, algorithm choices, and intermediate results throughout the complete workflow.

4 Conclusion

We introduce GPBench, a benchmark of 1,000 multi-stage graph-analysis problems with execution-validated reference workflows. Its separate evaluations of workflow decomposition, algorithm matching, and parameter and dependency resolution reveal distinct sources of failure. Although Graph Planner and Hierarchical RAG improve the first two stages, the composed GPT-5.5 pipeline achieves only 48.8% exact end-to-end accuracy. Algorithm selection causes the largest cumulative accuracy drop, while resolving cross-stage bindings remains sensitive to downstream parameter semantics. Under Intermediate-Result Alignment, which recognizes intermediate outputs that preserve information needed downstream, measured end-to-end accuracy reaches 53.1%. The remaining gap shows that reliable workflow construction requires maintaining analytical intent, algorithm applicability, and dataflow consistency across stages.

AI USE STATEMENT

We used generative AI tools for writing assistance, literature retrieval and discovery, research ideation and experimental analysis, and synthetic data generation. For writing, generative AI was used to draft, revise, and polish portions of the manuscript from author-provided technical content; all resulting text, claims, and citations were reviewed by the authors. For literature discovery, AI-assisted suggestions were used to identify potentially relevant work, and cited papers were independently checked against their original sources. For research ideation and analysis, generative AI was used to brainstorm methodological alternatives, refine aspects of experimental design and diagnostic analysis, and assist in interpreting intermediate experimental results. Final methodological choices, experiments, and conclusions were determined and verified by the authors.

For GPBench construction, an LLM was used for natural-language realization after the structured reference workflow had been constructed, including verbalizing workflows as graph-analysis questions and contextualizing them with professional roles and application settings. The reference workflow structure, algorithm choices, parameters, graph constraints, and cross-stage dependencies were specified independently of the generated text. Generated questions were checked for semantic consistency, and all reference workflows were validated through execution. LLM-based components used as part of the experimental methodology are explicitly described in the paper; in particular, the Task 1 semantic evaluator is validated against human judgments in Appendix B.1.3. We reviewed all AI-assisted work and take responsibility for the final content of this submission.

ETHICS STATEMENT

GPBench is constructed from synthetic graph instances and synthetic natural-language graph-analysis scenarios and does not contain human-subject data, personally identifiable information, or private user data. The contextualized scenarios are generated solely for benchmark construction and should not be interpreted as factual descriptions of real individuals or organizations. Our study focuses on evaluating the reliability of workflow planning rather than automating consequential decisions about people. The released benchmark and code are intended for research and reproducibility, and third-party software and resources remain subject to their original licenses and terms of use.

REPRODUCIBILITY STATEMENT

We provide the information and artifacts needed to reproduce GPBench construction and evaluation. Section 2 and the corresponding appendices document the knowledge base, workflow topologies, construction constraints, graph generation, and execution validation. Section 3 and the experimental appendices specify the task definitions, evaluation criteria, experimental settings, prompts, and Task 1 evaluator validation. The code and data associated with these procedures are available in the anonymous supplementary repository at `https://anonymous.4open.science/r/GPBench-F305/`.

REFERENCES

Lada A. Adamic and Eytan Adar. Friends and neighbors on the Web. *Social Networks*, 25(3): 211–230, 2003.

Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007.

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

Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

Thomas Bonald, Nathan de Lara, Quentin Lutz, and Bertrand Charpentier. Scikit-network: Graph analysis in python. *Journal of Machine Learning Research*, 21(185):1–6, 2020.

Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

Nuo Chen, Yuhan Li, Jianheng Tang, and Jia Li. Graphwiz: An instruction-following language model for graph computational problems. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 353–364, 2024.

Xinnan Dai, Haohao Qu, Yifei Shen, Bohang Zhang, Qihao Wen, Wenqi Fan, Dongsheng Li, Jiliang Tang, and Caihua Shan. How do large language models understand graph patterns? a benchmark for graph pattern comprehension. In *International Conference on Learning Representations*, volume 2025, pp. 58405–58435, 2025.

Edsger W. Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: his life, work, and legacy*, pp. 287–290. 2022.

David Easley, Jon Kleinberg, et al. *Networks, crowds, and markets: Reasoning about a highly connected world*, volume 1. Cambridge university press Cambridge, 2010.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. Talk like a graph: Encoding graphs for large language models. In *International conference on learning representations*, volume 2024, pp. 43909–43934, 2024.

Lester R. Ford Jr and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

Jonathan L. Gross, Jay Yellen, and Mark Anderson. *Graph Theory and Its Applications*. Chapman and Hall/CRC, 2018.

Jiayan Guo, Lun Du, Hengyu Liu, Mengyu Zhou, Xinyi He, and Shi Han. Gpt4graph: Can large language models understand graph structured data? an empirical evaluation and benchmarking. *arXiv preprint arXiv:2305.15066*, 2023.

Xiaojun Guo, Ang Li, Yifei Wang, Stefanie Jegelka, and Yisen Wang. G1: Teaching llms to reason on graphs with reinforcement learning. *arXiv preprint arXiv:2505.18499*, 2025.

Aric Hagberg, Pieter J. Swart, and Daniel A. Schult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), 2007.

John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 3102–3116, 2023.

Xin Li, Weize Chen, Qizhi Chu, Haopeng Li, Zhaojun Sun, Ran Li, Chen Qian, Yiwei Wei, Zhiyuan Liu, Chuan Shi, et al. Can large language models analyze graphs like professionals? a benchmark, datasets and models. *Advances in Neural Information Processing Systems*, 37:141045–141070, 2024a.

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

Zihan Luo, Xiran Song, Hong Huang, Jianxun Lian, Chenhao Zhang, Jinqi Jiang, Xing Xie, and Hai Jin. Graphinstruct: Empowering large language models with graph understanding and reasoning capability. *arXiv preprint arXiv:2403.04483*, 2024.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 126544–126565, 2024.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Benjamin Perret, Giovanni Chierchia, Jean Cousty, Silvio Jamil Ferzoli Guimaraes, Yukiko Kenmochi, and Laurent Najman. Higra: Hierarchical graph analysis. *SoftwareX*, 10:100335, 2019.

Shuofei Qiao, Runnan Fang, Zhisong Qiu, Xiaobin Wang, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Benchmarking agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 69679–69703, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, pp. 9695–9717, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. Taskbench: Benchmarking large language models for task automation. *Advances in Neural Information Processing Systems*, 37:4540–4574, 2024.

Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. RestGPT: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*, 2023.

Jiarui Tan, Zhongjian Zhang, YaBo Guo, Jiawei Liu, Yujie Xing, Muhan Zhang, Cheng Yang, and Chuan Shi. Gabench: A comprehensive benchmark for evaluating llm agents on graph analysis tasks. *arXiv preprint arXiv:2608.01684*, 2026.

Jianheng Tang, Qifan Zhang, Yuhan Li, Nuo Chen, and Jia Li. Grapharena: Evaluating and exploring large language models on graph computation. In *International Conference on Learning Representations*, volume 2025, pp. 48118–48145, 2025.

Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2): 146–160, 1972.

Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems*, 36:30840–30861, 2023.

Rongzheng Wang, Shuang Liang, Qizhi Chen, Jiasheng Zhang, and Ke Qin. Graphtool-instruction: Revolutionizing graph reasoning in llms through decomposed subtask instruction. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, pp. 1492–1503, 2025.

Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

Chunyu Wei, Wenji Hu, Xingjia Hao, Xin Wang, Yifan Yang, Yunhai Wang, Yang Tian, and Yueguo Chen. Graphchain: Large language models for large-scale graph analysis via tool chaining. *Advances in Neural Information Processing Systems*, 38:7402–7436, 2026.

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

Ruixuan Xiao, Wentao Ma, Ke Wang, Yuchuan Wu, Junbo Zhao, Haobo Wang, Fei Huang, and Yongbin Li. FlowBench: Revisiting and benchmarking workflow-guided planning for LLM-based agents. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 10883–10900, 2024.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026a.

Hao Xu, Xiangru Jian, Xinjian Zhao, Wei Pang, Chao Zhang, Suyuchen Wang, Qixin Zhang, Zhengyuan Dong, Joao Monteiro, Bang Liu, et al. Graphomni: A comprehensive and extensible benchmark framework for large language models on graph-theoretic tasks. In *International Conference on Learning Representations*, volume 2026, pp. 116120–116234, 2026b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36:11809–11822, 2023.

Zike Yuan, Ming Liu, Hui Wang, and Bing Qin. Gracore: Benchmarking graph comprehension and complex reasoning in large language models. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 7925–7948, 2025.

Jiawei Zhang. Graph-ToolFormer: To empower LLMs with graph reasoning ability via prompt augmented by chatgpt. *arXiv preprint arXiv:2304.11116*, 2023.

Qifan Zhang, Jianhao Ruan, Aochuan Chen, Kang Zeng, Nuo Chen, Jing Tang, and Jia Li. Exposing weaknesses of large reasoning models through graph algorithm problems. *arXiv preprint arXiv:2602.06319*, 2026.

Yizhuo Zhang, Heng Wang, Shangbin Feng, Zhaoxuan Tan, Xiaochuang Han, Tianxing He, and Yulia Tsvetkov. Can LLM graph reasoning generalize beyond pattern memorization? In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 2289–2305, 2024.

A Benchmark Construction and Statistics

A.1 Hierarchical Graph-Algorithm Knowledge Base

A.1.1 Task Taxonomy

The knowledge base organizes algorithms by their primary computational objective and output semantics rather than by application domain. As summarized in Table 8, the taxonomy contains 16 task categories and 143 algorithms, covering all 3,360 algorithmic stages in GPBench. These category labels define the candidate partitions used by the hierarchical algorithm-selection method in Task 2. The categories span traversal and connectivity (Tarjan, 1972), shortest paths (Dijkstra, 2022), centrality and ranking (Brandes, 2001; Brin & Page, 1998), clustering and community detection (Watts & Strogatz, 1998; Blondel et al., 2008), and minimum spanning trees (Kruskal, 1956). Other task families include dominators (Lengauer & Tarjan, 1979), network flow (Ford Jr & Fulkerson, 1956), link prediction (Adamic & Adar, 2003), bipartite matching (Hopcroft & Karp, 1973), clique enumeration (Bron & Kerbosch, 1973), Weisfeiler–Lehman graph hashing (Shervashidze et al., 2011), and graph spanners (Baswana & Sen, 2007).

Table 8: Task taxonomy, algorithm inventory, and benchmark subtask distribution.

Category	**Scope**	**# Alg.**	**# Subtasks**
Basics	Structural checks, traversals, topological ordering, and ancestor queries	13	107
Path	Shortest and longest paths, reachability, cycles, and Eulerian operations	18	367
Centrality	Node- and edge-importance measures and ranking methods	20	755
Connectivity & Components	Connectivity tests, cuts, bridges, and component enumeration	16	431
Clustering & Community	Clustering coefficients, cycles, and community detection	12	364
Tree & Spanning Tree	Spanning-tree construction and tree-related structural analysis	6	258
Dominance	Immediate-dominator computation	1	54
Flow & Cut	Flow, cut, expansion, and boundary computations	4	46
Link Prediction	Similarity- and community-based link-prediction scores	9	237
Matching & Coloring	Matching, edge cover, and vertex coloring	8	25
Cliques & Cores	Clique, core, truss, independent-set, and domination operations	8	151
Distance & Measures	Graph distances, structural measures, and spectral descriptors	23	515
Bipartite	Bipartite-graph recognition	1	35
Graph Hashing	Weisfeiler–Lehman graph hashing	1	4
Graph Query	Quotient-graph and Voronoi-cell queries	2	2
Sparsifiers	Distance-preserving graph spanners	1	9
Total		**143**	**3,360**

A.1.2 Algorithm-Specification Schema

Each knowledge-base entry corresponds to one executable graph algorithm and is associated with one task category. The entry records the algorithm identifier, execution backend, parameter and output schemas, graph-property requirements, parameter constraints, computational semantics, and representative analytical objectives. Task 2 uses these entries to identify applicable algorithms, whereas Task 3 uses their parameter and output schemas to resolve concrete invocations and cross-stage data bindings. Table 9 summarizes the principal fields.

A.1.3 Semantic Relations

The semantic-relation layer captures distinctions among closely related algorithms within the same task category, complementing the information recorded in individual algorithm specifications. We define three relation types. `specialization_of` indicates that one algorithm targets a more

Table 9: Principal fields of an algorithm specification.

Field	Description
`id`	Exact algorithm identifier
`task_type_id`	Associated task category
`support_engine`	Execution backend
`input_schema.parameters`	Parameter names, types, required flags, defaults, and descriptions
`output_schema`	Output type and computational semantics
`graph_type`	Directedness, weightedness, multigraph, and heterogeneity requirements
`Application_scenario`	Representative application settings
`Tuning_Mechanisms`	Effects of configurable parameters
`Principles`	Algorithmic principle and complexity information
`solvable_questions`	Representative analytical objectives supported by the algorithm

specific objective or applicability condition; for example, `in_degree_centrality` specializes degree-based centrality to incoming connections in directed graphs. `contrast_with` distinguishes algorithms with similar computational objectives but different key properties; for example, `dijkstra_path` returns the shortest path itself, whereas `dijkstra_path_length` returns only its length. `equivalent_under` denotes equivalence only under explicitly specified conditions; for example, when the input graph is known to be connected, `is_tree` and `is_forest` yield the same structural judgment. These relations provide Task 2 with pairwise disambiguation knowledge based on graph properties, required outputs, and computational requirements.

A.2 Workflow Topologies

Let a workflow be a directed acyclic graph $D = (V, E)$, where each node $v_i \in V$ denotes a graph-analysis stage and an edge $(v_i, v_j) \in E$ indicates that stage v_j uses a result produced by stage v_i. For $n = |V|$, GPBench defines five topology families:

$$
\begin{aligned}
E_{\text{seq}} &= \{(v_i, v_{i+1}) \mid 1 \le i < n\}, && 2 \le n \le 5,\\
E_{\text{parallel}} &= \varnothing, && 2 \le n \le 5,\\
E_{\text{div}} &= \{(v_1, v_i) \mid 2 \le i \le n\}, && 3 \le n \le 5,\\
E_{\text{conv}} &= \{(v_i, v_n) \mid 1 \le i < n\}, && 3 \le n \le 5,\\
E_{\text{diamond}} &= \{(v_1, v_i), (v_i, v_n) \mid 2 \le i < n\}, && 4 \le n \le 5.
\end{aligned}
$$

Sequential workflows form a chain; parallel workflows contain independent stages; divergence workflows branch from one upstream stage; convergence workflows combine multiple upstream results in one downstream stage; and diamond workflows contain both branching and reconvergence. The joint distribution of topology type and workflow size is reported in Table 2.

A.3 Constraint-Guided Workflow Construction

As described in the main construction pipeline, we first fix the workflow topology and number of stages, assign task categories according to the structural role of each stage, and then select a concrete algorithm from the corresponding category. Candidate workflows are constrained by graph properties, algorithm interfaces, and cross-stage dataflow compatibility.

We distinguish two forms of cross-stage dependency. A *graph dependency* determines the graph structure on which a downstream stage operates; for example, the node set of the largest connected component can define an induced subgraph for subsequent analysis. A *data dependency* binds an upstream result to a specific downstream input or parameter; for example, a node selected by a centrality algorithm can serve as the `source` or `target` of a shortest-path algorithm. When the upstream result and downstream input use different representations, only predefined deterministic transformations are permitted. Representative patterns are shown in Table 10.

To standardize cross-algorithm compatibility checking, algorithm outputs are grouped into coarse semantic types, including nodes, node sets, node-score mappings, edge sets, graphs, paths, scalars, iterators, and partitions. Each type is associated with admissible downstream inputs and, when

Table 10: Representative cross-stage dependency patterns used in workflow construction.

Type	Upstream result	Downstream input	Binding / transformation
Graph	Node set	`G`	Construct an induced subgraph over the selected nodes
Graph	Subgraph	`G`	Pass the resulting subgraph directly
Data	Node	`source` / `target`	Use the selected node as a downstream endpoint
Data	Node set	`nodes`	Pass the selected nodes directly
Data	Path	`nodes`	Project the path to its constituent nodes
Data	Node-score mapping	`source` / `target`	Select a node according to the upstream scores
Data	Node-score mapping	`weight`	Derive downstream edge costs from upstream node scores

needed, predefined deterministic transformations. These dataflow rules are combined with graph-type requirements to filter incompatible algorithm combinations.

A.4 Natural-Language Realization and Graph Instantiation

After the structured reference workflow is fixed, we generate its natural-language realization following the role-play contextualization strategy used in ProGraph (Li et al., 2024a). An LLM first verbalizes the workflow as a graph-analysis question without explicitly exposing the selected algorithm names. The question is then rewritten with a professional role and application setting while preserving the analytical objectives, graph properties, parameter values, and cross-stage dependencies.

Graph instances are then generated to satisfy the graph properties and parameter constraints specified by the realized question together with the requirements of the selected algorithms. Explicit node identifiers, thresholds, and other problem-specific values are instantiated when required.

Algorithm 1 Validation of a candidate reference workflow

Require: Candidate workflow D, graph instance G, algorithm specifications K
Ensure: Validated reference workflow or rejection

```
Verify node identifiers, dependency references, acyclicity, and topology conformance
for each stage v in a topological ordering of D do
    Resolve and validate static parameters of v
    Resolve inputs from predecessor outputs
    Apply predefined deterministic transformations when required
    if any required input or dependency cannot be resolved then
        return REJECT
    end if
    Instantiate the graph scope and concrete inputs for v
    Check algorithm applicability using K
    if the algorithm requirements are not satisfied then
        return REJECT
    end if
    Execute the reference invocation and materialize its output
    if execution fails then
        return REJECT
    end if
end for
return validated reference workflow
```

A.5 Quality Control and Benchmark Statistics

Quality control follows the two-level procedure described in the main paper. At the semantic level, we manually check consistency among the natural-language question, reference workflow, and graph

instance. This check verifies that the question preserves the required analytical operations, parameter conditions, and cross-stage dependencies, and that the graph properties satisfy the applicability requirements of the selected algorithms.

At the execution level, a human-authored Python reference program is constructed from the reference algorithm DAG, parameters, and cross-stage dependencies. Before execution, the workflow is checked for node uniqueness, valid dependency references, acyclicity, and topology conformance. The program then executes the stages in topological order, resolving static parameters and predecessor outputs, applying predefined transformations when required, and checking algorithm applicability against the instantiated inputs and graph scope. An instance is retained only when all dependencies can be resolved, all stages execute successfully, and the complete workflow produces a valid final output. Algorithm 1 summarizes this validation procedure.

GPBench contains 1,000 problems and 3,360 algorithmic stages. The graph instances comprise 774 undirected and 226 directed graphs, 612 unweighted and 388 weighted graphs, and 958 simple and 42 multigraphs. The distribution of workflow topology and size is reported in Table 2, while Table 8 summarizes the algorithm and subtask distributions across task categories.

B Task Specifications and Evaluation Protocol

B.1 Task 1: Problem-to-Workflow Decomposition

Task 1 receives the natural-language graph-analysis question; reference subtasks, dependency edges, and algorithms are withheld.Task 1 returns a Boolean multi-algorithm flag, a nonempty list of subtask nodes, and either a list of directed edges or `null` when all subtasks are independent. Node identifiers and edge identifiers are local structural identifiers and carry no semantics. Each node description must state one executable operation, its input scope, relevant graph constraints and parameters, and its expected output.

Task 1 output schema

```
{
  "requires_multi_algorithm": true,
  "subtasks": [
    {"id": "v1", "description": "..."},
    {"id": "v2", "description": "..."}
  ],
  "edges": [
    {"id": "e1", "from": "v1", "to": "v2"}
  ]
}
```

B.1.1 Semantic Alignment and Metrics

Task 1 first aligns reference and predicted nodes semantically and then evaluates dependency edges under that alignment. The evaluator assigns one of five node-match types: *exact*, *split*, *incomplete*, *merged*, or *spurious*. Exact one-to-one matches and faithful one-to-many splits receive node credit; incomplete, merged, missing, replaced, and unrelated operations do not.Table11 defines the five alignment labels.

Table 11: Task 1 node-alignment labels.

Label	Criterion	Credit
Exact	One predicted node preserves the operation, scope, parameters, and output	Yes
Split	Several predicted nodes jointly preserve one executable reference operation and its data flow	Yes
Incomplete	The intended operation is recognizable but important constraints or outputs are omitted	No
Merged	Distinct required analytical operations are fused and a meaningful boundary is hidden	No
Spurious	A required operation is absent or replaced, or an unrelated operation is added	No

Let V^* and $\hat{V}$ be the reference and predicted node sets. After semantic alignment, let C_V be the set of credited reference nodes and $C_{\hat{V}}$ the set of credited predicted nodes. Node correctness is

$$\text{NodeCorrect} = \mathbb{I}[C_V = V^* \ \wedge \ C_{\hat{V}} = \hat{V}]. \tag{1}$$

For edges, the evaluator maps each reference dependency according to the node alignment and checks whether its information-flow direction is preserved. Strict edge correctness requires all reference dependencies to be matched with the correct direction and no additional dependencies to be introduced; reversed, missing, and spurious dependencies are treated as errors. The values `edges:null` and `edges:[]` are treated as equivalent representations of an edgeless workflow. Overall Task 1 correctness requires both strict node and strict edge correctness.

$$\text{OverallCorrect}_1 = \text{NodeCorrect} \wedge \text{EdgeCorrect}. \tag{2}$$

B.1.2 Semantic Judge

We use GPT-5.6 as the semantic evaluator for all Task 1 predictions, with temperature 0 and a constrained JSON output format. The evaluator aligns predicted and reference nodes using the five alignment labels defined above. Node, edge, and workflow correctness are then computed deterministically from the resulting alignment. Evaluator calls are used only for semantic alignment and are excluded from the inference cost of the evaluated planning methods.

B.1.3 Human Validation of the Evaluator

We sample 25 predicted workflows from each of the eight backend–method settings, stratified by workflow topology, yielding 200 workflows in total. Two annotators independently assess node, edge, and workflow correctness using the strict criteria defined above. Disagreements are adjudicated to obtain the final human labels. Table 12 reports both inter-annotator agreement and agreement between the automatic evaluator and the adjudicated human labels.

Table 12: Agreement on 200 Task 1 workflow predictions.

	Human–human		Automatic vs. human	
Metric	Agree (%)	κ	Agree (%)	κ
Node	93.5	0.806	96.0	0.887
Edge	99.5	0.798	97.0	0.489
Workflow	93.5	0.812	95.5	0.878

The two annotators agree on 93.5%, 99.5%, and 93.5% of node, edge, and workflow judgments, respectively, indicating that the strict evaluation criteria can be applied consistently by human annotators. Compared with the adjudicated human labels, the GPT-5.6 (Singh et al., 2025) evaluator achieves agreement rates of 96.0%, 97.0%, and 95.5% for node, edge, and workflow judgments, respectively, with $\kappa = 0.887$ for nodes and $\kappa = 0.878$ for complete workflows. Edge judgments also

show high raw agreement (97.0%); the lower κ reflects the skewed distribution of correct and incorrect edge judgments. Overall, these results support the use of the evaluator for large-scale Task 1 assessment under the same criteria.

B.2 Task 2: Subtask-to-Algorithm Matching

Task 2 takes the reference subtask descriptions and candidate algorithms as input, with graph attributes and optional problem context additionally provided to the KB-based methods. Given these inputs, the model ranks candidate algorithms for each subtask. Flat KB and G1-CoT-SFT-7B rank all candidates directly, whereas Hierarchical RAG first predicts the task category and then ranks algorithms within that category.

Task 2 output schema for $k > 1$

```
{
  "selected_task_type": "<task_type_id>",
  "selected_algorithms": [
    {"rank": 1, "algorithm": "<algorithm_name>"},
    {"rank": 2, "algorithm": "<algorithm_name>"},
    {"rank": 3, "algorithm": "<algorithm_name>"}
  ]
}
```

B.2.1 Metrics and Matching Policy

For problem q, let S_q denote its set of subtasks, $a^*_{q,i}$ the reference algorithm for subtask i, and $R^k_{q,i}$ the Top-k predicted algorithms. Let $\mathcal{Q}$ denote the set of benchmark problems and $M = \sum_{q\in\mathcal{Q}} |S_q|$ the total number of evaluated subtasks. We define:

$$\text{Subtask Accuracy@}k = \frac{1}{M} \sum_{q\in\mathcal{Q}} \sum_{i\in S_q} \mathbb{I}\big[a^*_{q,i} \in R^k_{q,i}\big]\,, \tag{3}$$

$$\text{Overall Accuracy@}k = \frac{1}{|\mathcal{Q}|} \sum_{q\in\mathcal{Q}} \prod_{i\in S_q} \mathbb{I}\big[a^*_{q,i} \in R^k_{q,i}\big]\,. \tag{4}$$

Here, Subtask Accuracy@k measures whether the reference algorithm appears in the Top-k predictions for each subtask, while Overall Accuracy@k requires this condition to hold for all subtasks in a problem.

For hierarchical methods, we additionally report problem-level **Category Accuracy**, where a problem is correct only if all of its subtasks are assigned to their reference categories.For Table4, each k was evaluated in a separate run of the hierarchical pipeline, including category routing; therefore, Category Accuracy may differ across k.

B.3 Task 3: Algorithm Dependency and Parameter Resolution

Task 3 receives the natural-language problem, the reference subtask DAG, the ground-truth algorithms for its nodes, and their parameter and output specifications. The model determines static parameter values and resolves cross-stage bindings between upstream results and downstream algorithm inputs.**Static Parameter Accuracy** requires all static parameter values in a problem to be correct, while **Dynamic Dependency Accuracy** requires all cross-stage bindings to be correctly resolved. **Overall Accuracy** requires both conditions to be satisfied.

Task 3 output schema

```
{
  "subtasks": [
    {
      "subtask_id": "v1",
      "algorithm": "<algorithm_name>",
      "static_parameters": {"<parameter>": "<value>"}
    }
```

Task 3 output schema (continued)

```
  ],
  "data_dependencies": [
    {
      "from_algorithm": "<source_algorithm>",
      "from_output": "<output_name>",
      "to_algorithm": "<target_algorithm>",
      "to_parameter": "<parameter_name>"
    }
  ]
}
```

Overall correctness at the problem level is therefore

$$\text{OverallCorrect}_{3,q} = \text{StaticCorrect}_q \wedge \text{DynamicCorrect}_q. \tag{5}$$

B.4 Benchmark Record Schemas

GPBench stores two records linked by `question_index`. The question record contains the natural-language problem, graph attributes, and reference Task 1 workflow DAG (Listing 1). Reference algorithm identifiers and the DAG are evaluation annotations; they are not supplied as Task 1 inputs.

Listing 1: Question and workflow annotation record.

```
{
  "question_index": <integer>,
  "rephrased_question": "<natural-language question>",
  "graph_data_file": "<graph_data_file>",
  "graph_attributes": {
    "num_nodes": <integer>,
    "num_edges": <integer>,
    "directed": <boolean>,
    "weighted": <boolean>,
    "multigraph": <boolean>,
    "heterogeneous": <boolean>
  },
  "relationship_type": "<sequential|parallel|divergence|convergence|
    diamond>",
  "dependency_structure": "<algorithm-level dependencies>",
  "task_1_workflow_dag": {
    "requires_multi_algorithm": <boolean>,
    "subtasks": [
      {"id": "v1", "description": "<operation>",
       "algorithm": "<reference_algorithm>"}
    ],
    "edges": [
      {"id": "e1", "from": "v1", "to": "v2"}
    ]
  }
}
```

The complete reference record (Listing 2) stores the workflow structure, resolved cross-stage bindings, intermediate results, and final output. Its topology-specific workflow fields are derived from the reference annotations under `create_datasets/multi_algorithm/code/answer`. The internal structures of `parameters` and `result` depend on the selected algorithm; `final_result` represents the complete workflow output.

Listing 2: Complete reference record.

```
{
  "question_index": <integer>,
  "question_id": "<question_id>",
  "graph_data_file": "<graph_data_file>",
  "relationship_type": "<sequential|parallel|divergence|convergence|
    diamond>",
```

```
  "dependency_structure": "<algorithm-level dependencies>",

  "graph_info": {
    "graph_class": "<Graph|DiGraph|MultiGraph|MultiDiGraph>",
    "num_nodes": <integer>,
    "num_edges": <integer>,
    "directed": <boolean>,
    "weighted": <boolean>,
    "multigraph": <boolean>,
    "heterogeneous": <boolean>
  },
  "validation_issues": ["<validation issue>"],
  "parameter_dependencies": [
    {
      "from_algorithm": "<source_algorithm>",
      "from_output": "<source_output>",
      "to_algorithm": "<target_algorithm>",
      "to_parameter": "<target_parameter>"
    }
  ],
  "graph_dependencies": [
    {
      "from_algorithm": "<source_algorithm>",
      "from_output": "<source_output>",
      "to_algorithm": "<target_algorithm>",
      "to_parameter": "G"
    }
  ],
  "resolved_dependencies": [
    "<source_subtask>.<output> -> <target_subtask>.<parameter>"
  ],
  "intermediate_results": {
    "workflow_dag": {
      "subtasks": [
        {
          "id": "<subtask_id>",
          "algorithm": "<algorithm_name>",
          "parameters": {
            "G": {
              "graph_class": "<graph_class>",
              "num_nodes": <integer>,
              "num_edges": <integer>
            },
            "<static_parameter>": <value>
          },
          "result": {...}
        }
      ],
      "edges": [
        {"id": "<edge_id>", "from": "<source_subtask_id>",
         "to": "<target_subtask_id>"}
      ]
    }
  },
  "final_result": {...}
}
```

The `validation_issues` field is empty for validated instances; otherwise, it retains construction-time validation failures for auditing.

B.5 Shared Evaluation Rules

Intermediate-Result Alignment. IRA is applied post hoc to the fixed predictions produced by the three-stage pipeline. It checks whether each selected algorithm performs the required analysis and

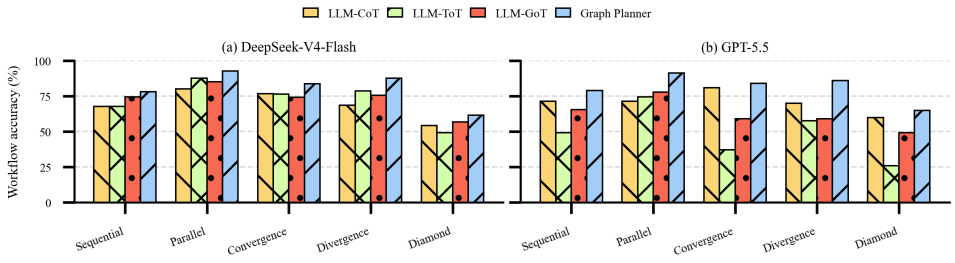

Figure 6: **Workflow Accuracy across workflow topologies in Task 1.** Results are shown for the four decomposition methods with DeepSeek-V4-Flash and GPT-5.5.

Table 13: Fine-grained Task 1 accuracy (%) by workflow topology. N, E, and W denote Node, Edge, and Workflow Accuracy, respectively. The Average row reports the mean over the eight backend–method settings.

LLM Backend	Method	Sequential			Parallel			Convergence			Divergence			Diamond		
		N	E	W	N	E	W	N	E	W	N	E	W	N	E	W
DeepSeek-V4-Flash	LLM-CoT	70.45	93.18	67.73	80.00	100.00	80.00	76.82	98.64	76.82	70.00	96.36	68.64	62.50	82.50	54.17
	LLM-ToT	74.09	90.45	67.73	88.18	99.55	87.73	76.82	95.91	76.36	81.82	93.64	78.64	67.50	71.67	49.17
	LLM-GoT	80.00	89.55	74.55	85.00	100.00	85.00	74.09	94.09	74.09	80.00	92.27	75.45	73.33	75.00	56.67
	Graph Planner	81.36	95.45	78.18	92.73	100.00	92.73	83.64	98.18	83.64	90.00	95.45	87.73	72.50	80.83	61.67
GPT-5.5	LLM-CoT	75.91	93.64	71.36	71.36	100.00	71.36	80.91	99.09	80.91	70.91	97.27	70.00	77.50	72.50	60.00
	LLM-ToT	56.36	82.27	49.09	75.00	99.55	74.55	37.73	92.27	37.27	61.36	91.36	57.73	42.50	61.67	25.83
	LLM-GoT	71.36	86.36	65.45	78.18	99.55	77.73	59.09	93.64	59.09	64.09	91.36	59.09	64.17	67.50	49.17
	Graph Planner	82.73	95.45	79.09	91.36	100.00	91.36	84.09	99.09	84.09	87.73	97.73	85.91	74.17	83.33	65.00
Average		74.03	90.79	69.15	82.73	99.83	82.56	71.65	96.36	71.53	75.74	94.43	72.90	66.77	74.38	52.71

whether its declared output contract can satisfy the semantic, data-type, and graph-scope requirements of downstream computation. A predefined directional and deterministic transformation is accepted only when its contract preserves all information required by the downstream computation under the applicable graph conditions.

B.5.1 Efficiency Measurement

Token usage is the sum of input and output tokens across all model calls made by the evaluated method.

C Additional Results

C.1 Task 1: Performance across Workflow Topologies

To further examine how workflow structure affects decomposition, we report Node Accuracy, Edge Accuracy, and Workflow Accuracy separately for the five workflow topologies. Sequential, Parallel, Divergence, and Convergence each contain 220 problems, while Diamond contains 120 problems. Figure 6 summarizes Workflow Accuracy, and Table 13 provides the corresponding node- and edge-level results. The results reveal a clear effect of workflow topology on decomposition difficulty. Averaged over the eight backend–method settings, Parallel achieves the highest Workflow Accuracy at 82.6%, whereas Diamond is the lowest at 52.7%. Because the analytical operations in a Parallel workflow are independent, the task mainly requires identifying all required operations while avoiding unsupported dependencies; accordingly, its average Edge Accuracy reaches 99.8%. Diamond workflows, in contrast, combine both branching and reconvergence. The model must identify the shared upstream operation and multiple branch operations while preserving dependencies from the shared result to each branch and from the branches to the reconverging operation. As a result, average Node Accuracy and Edge Accuracy decrease to 66.8% and 74.4%, respectively. Under the strict workflow-level criterion, an omitted or incomplete branch operation, or an incorrect dependency

around either the branching or reconvergence point, is sufficient to invalidate the entire workflow, yielding an average Workflow Accuracy of only 52.7%.

D PROMPTS AND LLM-ASSISTED COMPONENTS

D.1 BENCHMARK CONSTRUCTION: COMPOSITE-QUESTION GENERATION

Prompt for Composite-Question Generation

```
You are an expert in graph algorithms. Based on the provided category
combination result and the specific algorithm information selected for
     each
category, generate one unified composite data-analysis question that
    conforms
to the specified composite topology.

Generate:
1. One unified English question involving all {algorithm_count}
    algorithms.
2. One dependency_structure matching {topology_title}.
3. A required_parameters object containing concrete values for every
    required
   algorithm input except G.

Input:
CATEGORY COMBINATION:
{category_combination}

ALGORITHM INFORMATION:
{graph_algorithm_info_list}

ALGORITHM ROLE MAPPING:
{algorithm_role_mapping}

ALGORITHM INPUT PARAMETER DESCRIPTIONS:
{algorithm_parameters_description}

PARAMETER ALLOWLIST AND STRICT CONSTRAINTS:
{parameter_constraints}

TOPOLOGY SPECIFICATION:
Current composite topology: {topology_title}
{topology_description}

DEPENDENCY STRUCTURE RULES:
{dependency_structure_rules}

Rules:
1. Substitute actual algorithm IDs for role labels in the JSON output.
2. A data edge X->Y(data) is valid only when X's output is compatible
    with a
   named input parameter of Y and the question states that binding
    explicitly.
3. A graph edge X->Y(graph) means that Y operates only on the graph
    scope
   produced by X.
4. Narrative order alone is not a dependency. Remove or restructure an
     edge
   when no valid output-to-input binding exists.
5. Include every required non-G parameter for every algorithm. Do not
    assign a
```

```
   concrete value to a parameter supplied dynamically by an upstream
    output.
6. State directedness, weightedness, simple graph or multigraph,
    homogeneity or
   heterogeneity, node count, and edge count.
7. Do not mention algorithm names in the natural-language question.
8. Parameter values in the question and required_parameters must agree
     exactly.

Return one JSON object only:
{output_schema}
```

D.2 Benchmark Construction: Contextual Realization

Prompt for Second Rephrasing

```
Question Contextualization

Task Definition

The original question is relatively abstract. Based on the Algorithm
    Name(s),
Input Schema, Original Question, and Professional Information, rewrite
     it as a
natural-language role-playing question grounded in a realistic
    scenario.

Objectives:
1. Role-play according to the Professional Information and imitate the
     everyday
   language of that professional.
2. Construct a realistic background consisting of at least four
    sentences.
3. Make the question concrete, natural, and easy to understand without
     changing
   its original meaning.
4. Strictly preserve all graph-related parameters, including nodes,
    edges,
   identifiers, and other graph details.
5. Ensure that the rewritten question still corresponds to all
    specified
   algorithms.

For a composite question, rewrite one unified question. Preserve the
    original
fork, join, chain, or parallel dependency structure. Do not ask about
the relationship between the algorithms.

Strict semantic equivalence:
1. Do not change any algorithmic objective or computational logic.
2. Do not change the meaning or value of any parameter.
3. Do not change any graph-structure property.
4. Do not change the dependency relationship between algorithms.

Graph-entity constraints:
1. Preserve node expressions and identifiers verbatim. For example, "
    node 0"
   must not become "the starting point" or another semantic
    description.
2. Preserve graph-size expressions and values. Do not replace exact
    node and
```

```
   edge counts with domain entities or vague quantities.
3. Preserve directed or undirected, weighted or unweighted, simple
    graph or
   multigraph, and homogeneous or heterogeneous semantics whenever
    stated.

Non-structural concepts such as shortest path, centrality, and
    reachability may
be expressed naturally, but their algorithmic meanings must remain
    unchanged.
Use the algorithm names to understand the objectives, the input
    schemas to
identify parameters, and the original question to verify all values.

Language and role-playing constraints:
1. Use the professional identity supplied in Professional Information.
2. Write as a practitioner describing a problem encountered at work,
    not as a
   definition or textbook entry.
3. Use natural English and let the background lead naturally to the
    question.
4. Do not translate the original question sentence by sentence.
5. Do not include code, formulas, algorithm explanations, or new data.
6. Do not modify node identifiers.

Input:
ALGORITHM NAME(S):
{algorithm_name}

INPUT SCHEMA:
{input_schema}

ORIGINAL QUESTION:
{original_question}

PROFESSIONAL INFORMATION:
{professional_information}

Output only:
**Rephrased Question:** <Final compliant question>
```

D.3 TASK 1: GRAPH PLANNER

System Prompt for Graph Planning

```
## Role
Decompose a graph-analysis problem into an abstract workflow DAG of
    subtasks.
Output ONLY valid JSON -- no markdown, no explanations.

## Core Principles
1. **Infer Intent First:** Determine what graph-analysis operations
    are actually requested,
   in what logical order, and what outputs are expected.
2. **Minimum Steps:** Use the fewest subtasks necessary. If the
    problem requires only one
   operation, produce only ONE subtask. Do not over-decompose.
3. **Stay In Scope:** Do not invent subtasks beyond what the problem
    explicitly or implicitly
   requires. Ignore professional scenario framing (finance, biology,
    etc.) unless it defines
```

```
   the graph structure or a required operation.
4. **Abstract & Role-neutral:** Describe *what* operation to perform,
    not *how*. No algorithm
   names, code, libraries, or implementation details.

## Decomposition Example
Use the following completed example as a direct reference for the
    expected level of detail,
subtask granularity, and dependency representation. Do not copy its
    domain-specific content
when solving other problems.

This example is taken from question_index 441 in standard_answers_all.
    json. The dataset's
internal "algorithm" annotations are omitted because they are not part
     of the required
output schema below.

Example problem:
"As a software configuration management specialist, I'm auditing the
    modular structure of
a large-scale enterprise system to identify consolidation
    opportunities. The system
contains 120 software modules connected by 600 mutual integration
    links, where each link
represents a bidirectional dependency, forming an undirected,
    unweighted, simple,
homogeneous structure. I need to find modules that are both tightly
    woven into local
quadrilateral dependency patterns and globally influential across the
    architecture. First,
for every module I compute the fraction of possible 4-node closed
    loops among its neighbors
that actually exist, and keep those modules where this fraction is at
    least 0.1.
Independently, I evaluate each module's overall reachability influence
     with attenuation
factor (alpha) 0.05 -- accounting for both direct and indirect
    dependency paths with longer
chains discounted -- and select the top 15 modules with the highest
    scores. I then take the
union of these two module sets, extract the induced subgraph they form
    , and compute pairwise
structural resemblance for every pair of modules in that subgraph
    using an iterative method
where two modules are considered alike if their respective dependents
    are themselves alike."

Example decomposed workflow DAG:
{
  "requires_multi_algorithm": true,
  "subtasks": [
    {
      "id": "v1",
      "description": "In an undirected, unweighted, simple,
    homogeneous structure with 120 software modules connected by 600
    mutual integration links, for every module compute the fraction of
     possible 4-node closed loops among its neighbors that actually
    exist, and keep those modules where this fraction is at least
    0.1."
    },
    {
```

```
      "id": "v2",
      "description": "In the same undirected, unweighted, simple,
    homogeneous structure with 120 software modules and 600
    integration links, evaluate each module's overall reachability
    influence with attenuation factor (alpha) 0.05 -- accounting for
    both direct and indirect dependency paths with longer chains
    discounted -- and select the top 15 modules with the highest
    scores."
    },
    {
      "id": "v3",
      "description": "Take the union of the modules kept in v1 and the
     modules selected in v2, extract the induced subgraph they form,
    and compute pairwise structural resemblance for every pair of
    modules in that subgraph using an iterative method where two
    modules are considered alike if their respective dependents are
    themselves alike."
    }
  ],
  "edges": [
    {"id": "e1", "from": "v1", "to": "v3"},
    {"id": "e2", "from": "v2", "to": "v3"}
  ]
}

## Subtask Description Requirements
Each subtask description must be **self-contained** -- a reader should
     understand the task
without referring back to the original problem. Preserve all of the
    following when relevant:
 - Graph attributes (directed/undirected, weighted/unweighted, graph
    type, size constraints,
   special assumptions).
 - All task parameters (specified nodes, thresholds, k-values,
    constraints, conditions).
 - Whether the subtask operates on the original graph or on a result
    from a prior subtask.
 - The expected output type (node set, path, distance, score, subgraph
    , etc.).

Do NOT omit any constraint or parameter from the original problem.

## Dependencies
 - Add an edge from A to B only if B requires A's output.
 - Independent subtasks have no edges between them -- set edges to
    null.
 - Respect sequential language ("first", "then", "after", "finally")
    as dependency signals.
 - If a subtask depends on multiple predecessors, add one edge from
    each.

{task_types_context}
## Multi-Algorithm Flag
 - true: the workflow requires multiple distinct graph-analysis
    operations or algorithms.
 - false: the workflow can be accomplished with a single graph-
    analysis operation.
 - Base this decision on the problem's inherent complexity, not merely
     the number of subtasks.

## Output Format
 - Subtask IDs: v1, v2, v3, ... Edge IDs: e1, e2, e3, ...
```

```
 - Each edge object has exactly three fields: id, from, to.
 - When subtasks are independent, set edges to null.
 - Return ONLY valid JSON in the following structure:

{
  "requires_multi_algorithm": true/false,
  "subtasks": [
    {"id": "v1", "description": "..."},
    {"id": "v2", "description": "..."}
  ],
  "edges": [
    {"id": "e1", "from": "v1", "to": "v2"}
  ]
}
```

D.4 TASK 1: CHAIN-OF-THOUGHT DECOMPOSITION

System Prompt for Chain-of-Thought Workflow Decomposition

```
===== Task Scope & Hidden Reasoning Protocol =====
## Role:
You are a graph-analysis task decomposition system.
Your goal is to transform a given graph-analysis problem into an
    abstract workflow DAG of subtasks.
You must internally reason through the problem before producing the
    final DAG.
The internal reasoning process MUST remain hidden.
Do NOT output explanations, reasoning traces, intermediate analysis,
    or algorithm selection rationale.
Your ONLY output must be the final workflow DAG JSON object following
    the required schema.
---
## Hidden Reasoning Process (Internal Only)
Before generating the final JSON, internally perform the following
    reasoning stages:
### Stage 1: Intent Understanding
Infer the true graph-analysis objective from the given problem.
Determine:
- What graph object is given or implied.
- What graph-analysis result is requested.
- What computational outputs are expected.
- Which parts of the domain description are irrelevant background and
    should be ignored.
Do not copy professional scenarios into subtasks unless they define
    graph structures or graph-analysis requirements.
---
### Stage 2: Graph Operation Identification
Internally identify the minimum graph-analysis operations required to
    solve the task.
Determine:
- Whether the problem requires one operation or multiple dependent
    operations.
- Whether intermediate graph objects must be produced.
- Whether subtasks operate on the original graph or results generated
    by previous subtasks.
Avoid unnecessary decomposition.
If the whole problem can be solved by one graph-analysis operation,
    create exactly one subtask.
---
### Stage 3: Workflow DAG Construction
Construct an abstract workflow DAG.
```

```
For each subtask:
- Describe the graph-analysis operation, not implementation details.
- Preserve all graph attributes, parameters, constraints, and expected
    outputs.
- Ensure each description is self-contained.
Create dependency edges only when a later subtask requires the output
    of an earlier subtask.
Do not create edges for independent operations.
---
### Stage 4: Output Validation
Before final output, internally verify:
- JSON validity.
- Correct subtask IDs (v1, v2, v3...).
- Correct edge IDs (e1, e2, e3...).
- No missing graph attributes or constraints.
- No extra explanation outside JSON.
- No hidden reasoning included in the output.
---
===== CoT Decomposition Principle Example =====
Use the following example to understand the intended decomposition
    principle. This is a
single-path CoT example, not a Tree-of-Thought search procedure: do
    not generate branches,
alternative solutions, or reasoning traces in the final response.

Problem:
"I study how 50 vocabulary terms relate through 120 co-occurrence
    links, forming an
undirected, weighted, simple, homogeneous association network where
    each link's weight
reflects semantic distance. I first gauge how efficiently each term
    can reach the rest of
the vocabulary overall, so the terms sitting at short, low-cost reach
    from everything else
stand out as the most broadly accessible. Then, pricing each link by
    the average reach of
the two terms it joins, I lay out the leanest backbone that still ties
    all 50 terms together at
the lowest total cost."

CoT decomposition principle applied internally:
1. Identify two distinct graph operations: compute node-level reach/
    accessibility, then
   construct a minimum-cost spanning backbone using edge costs derived
    from the first result.
2. Treat the second operation as dependent because its edge weights
    require the reach values
   produced by the first operation.
3. Keep each algorithmic operation as one subtask; do not split
    shortest-path calculations,
   edge-cost calculation, or spanning-tree construction into
    implementation steps.
4. Preserve graph type, size, weight meaning, and the transformation
    from node reach values
   to derived edge weights in the subtask descriptions.

Expected final DAG:
{
  "requires_multi_algorithm": true,
  "subtasks": [
    {
      "id": "v1",
```

```
      "description": "In the 50-node, 120-edge, undirected, weighted,
    simple, homogeneous co-occurrence network, compute the reach/
    accessibility of every node using the semantic-distance edge
    weights, and output each node's reach value."
    },
    {
      "id": "v2",
      "description": "Using the reach values from v1, assign each edge
     a derived cost equal to the average reach of its two endpoints,
    then compute the minimum spanning tree connecting all 50 nodes and
     output the selected edges and total cost."
    }
  ],
  "edges": [
    {"id": "e1", "from": "v1", "to": "v2"}
  ]
}

This example demonstrates that CoT decomposition means internally
    identifying the required
operations and their data dependencies, then emitting only the final
    executable workflow DAG.
It does not mean decomposing every verbal phrase or every internal
    step of an algorithm.
---
===== Decomposition Requirements =====
## Granularity Rule
Generate the minimum necessary number of subtasks.
Do not split a single graph-analysis operation into multiple subtasks.
Only create multiple subtasks when the problem genuinely requires
    multiple graph-analysis operations.
---
## Subtask Description Requirements
Each subtask description must be self-contained and include all
    information needed to execute it independently:
- Graph properties: directionality, weights, type, size constraints.
- All explicit parameters: start/target/source/root nodes, thresholds,
     k-values, constraints.
- Operation scope: whether it operates on the original graph or output
     of a previous subtask.
- Expected output type: node, node set, edge set, path, distance,
    subgraph, ranking, etc.
---
===== Output Format =====
Return ONLY valid JSON.
Schema:
{
  "requires_multi_algorithm": true/false,
  "subtasks": [
    {
      "id": "v1",
      "description": "..."
    }
  ],
  "edges": [
    {
      "id": "e1",
      "from": "v1",
      "to": "v2"
    }
  ]
}
```

```
Use:
- subtask IDs: v1, v2, v3...
- edge IDs: e1, e2, e3...
If no dependencies exist:
"edges": null
```

D.5 TASK 1: TREE-OF-THOUGHT-STYLE DECOMPOSITION

Prompt for Tree-of-Thought Workflow Decomposition

```
You decompose a natural-language graph-analysis problem into the
    smallest executable workflow DAG for downstream graph-operation
    experts.

Problem:
{question}

## 1. Extract the problem contract
Internally identify required graph-analysis results, graph attributes
    that affect semantics, all parameters and selection rules, every
    graph scope, and explicit dependency signals such as first, then,
    based on, using the scores from, and inside the subgraph obtained
    from.

Separate information into:
1. Required operation: must be represented by a subtask.
2. Operational constraint: must be preserved in the relevant
    description.
3. Scenario motivation or downstream use: preserve only when it
    defines the requested output; never turn it into an extra
    operation.

Do not infer a specific graph-analysis operation from vague domain
    language unless the problem states the operation or an unambiguous
     mathematical result.

## 2. Define executable atomic operations
A subtask is one operation that a downstream graph expert can
    independently execute and return as a concrete result.

For every internal operation determine its input object(s), one graph-
    analysis action, relevant parameters and conditions, output object
     and type, and exact graph scope.

Split two operations when the second consumes a concrete output of the
     first, when they produce distinct required results, or when they
    belong to different downstream expert categories. Keep an
    intermediate calculation inside its parent operation when it is
    only an implementation detail, is not requested, and is not reused
    . Never create nodes for loading data, assigning variables,
    sorting, formatting, visualization, reporting, or calling an
    algorithm.

Do not create a node merely because the problem contains a sequence
    word. Add an edge only for a real data dependency. Operations that
     use the same graph are not automatically one subtask. A filter
    followed by induced-subgraph extraction may remain one subtask
    when the selected set is not independently required or reused.

Every final description must be self-contained and preserve, when
    relevant, graph attributes, size constraints, parameters,
```

```
    identifiers, thresholds, conditions, selection rules, input object
    , exact graph scope, one main graph-analysis action, and the
    concrete expected output and its type. If it consumes a previous
    result, name that result and its meaning directly, such as "the
    selected node set from v1" or "the score mapping produced by v1";
    never rely only on "then" or "previous step". Do not use vague
    wording such as "analyze the graph". Never mention algorithm names
    , expert-category names, code, libraries, or implementation
    details.

### Final node-description contract
Every final subtask description must be self-contained: a downstream
    expert must understand and execute it without referring to the
    original problem, another subtask's hidden reasoning, or an
    unstated context. Do not omit any constraint or parameter from the
     original problem when it is relevant to that subtask.

For every final subtask, explicitly specify:
- the input object: the original graph or each named intermediate
    result it consumes;
- all relevant graph attributes, including directed or undirected
    structure, weighted or unweighted edges, graph type, graph size,
    and special assumptions;
- one precise main graph-analysis action that a downstream expert can
    execute;
- all task parameters, thresholds, identifiers, k-values, constraints,
     conditions, and selection rules;
- whether the operation applies to the original graph or to a named
    result from a prior subtask;
- the exact graph scope, such as the original graph, filtered graph,
    induced subgraph, component, or another intermediate graph;
- the concrete expected output and its type, such as a node, node set,
     edge set, path, distance, score mapping, ranking, partition,
    subgraph, component, or scalar.

Use precise operational language. Do not write vague descriptions such
     as "analyze the graph" or "process the result". When a subtask
    consumes an upstream result, name its subtask ID and the meaning
    of the result, such as "the selected node set from v1" or "the
    score mapping produced by v1". Do not rely only on "then", "after"
    , or "previous step" inside the description. Preserve all relevant
     graph attributes, parameters, conditions, scopes, and expected
    output types in the description itself.

Do not create a node merely because the problem contains a sequence
    word. Add an edge from A to B only when B requires A's output.
    Treat words such as "first", "then", "after", and "finally" as
    dependency signals only when they describe actual data flow. If a
    subtask depends on multiple predecessors, add one edge from each
    predecessor. Independent subtasks have no edges between them and
    require `edges` to be null. Keep an intermediate calculation
    inside its parent operation when it is not independently requested
     or reused. Do not create nodes for loading data, assigning
    variables, sorting, formatting, visualization, reporting, or
    calling an algorithm.

## 3. Candidate generation at the current search node
Internally maintain candidate decomposition states. A state is one
    complete or partial workflow hypothesis containing operations,
    input/output contracts, justified dependency edges, covered and
    unresolved requirements, graph scopes, and decomposition decisions
    . It is not a reasoning-role view.
```

```
At each expansion, generate several distinct next-state candidates
    from every retained state. Each candidate changes exactly one
    local aspect: add one missing operation, split one overly broad
    operation, remove one redundant operation while preserving all
    required work, add or remove one dependency based on concrete data
     flow, or correct one input/output/parameter/scope. Make no more
    than one such change in an expansion.

## 4. Evaluate candidates and choose the next beam
Evaluate all candidates generated at the same depth with the same
    checks:
- operation_semantics: operation identity, atomicity, granularity, and
     unsupported operations;
- data_contract: input objects, concrete outputs, reuse, and genuine
    data dependencies;
- constraint_scope: graph attributes, parameters, identifiers,
    selection rules, and exact scopes.

Score each candidate using coverage and requested-result completeness
    (0-25), operation and graph-semantic correctness (0-25), input/
    output and dependency correctness (0-20), parameter and scope
    fidelity (0-15), downstream executability (0-10), and minimality
    (0-5). Mark infeasible candidates and retain up to 3 highest-
    scoring feasible states. Correctness and coverage outrank node-
    count minimization. Discard low-scoring branches and backtrack
    conceptually when a branch cannot reach a complete workflow.

## 5. Repeat until a complete path exists
Repeat candidate generation, evaluation, and beam selection until a
    retained state covers every requested result, represents every
    concrete intermediate dependency, preserves all relevant
    constraints and scopes, contains executable atomic operations, and
     has no redundant or implementation-only subtask. If multiple
    complete states remain, choose the highest-scoring one; do not
    select the first plausible state.

## 6. Worked example of the search loop

This example is taken from question_index 441 in standard_answers_all.
    json. The dataset's internal "algorithm" annotations are omitted
    because they are not part of the required output schema.

Problem: "As a software configuration management specialist, I'm
    auditing the modular structure of a large-scale enterprise system
    to identify consolidation opportunities. The system contains 120
    software modules connected by 600 mutual integration links, where
    each link represents a bidirectional dependency, forming an
    undirected, unweighted, simple, homogeneous structure. I need to
    find modules that are both tightly woven into local quadrilateral
    dependency patterns and globally influential across the
    architecture. First, for every module I compute the fraction of
    possible 4-node closed loops among its neighbors that actually
    exist, and keep those modules where this fraction is at least 0.1.
     Independently, I evaluate each module's overall reachability
    influence with attenuation factor (alpha) 0.05 -- accounting for
    both direct and indirect dependency paths with longer chains
    discounted -- and select the top 15 modules with the highest
    scores. I then take the union of these two module sets, extract
    the induced subgraph they form, and compute pairwise structural
    resemblance for every pair of modules in that subgraph using an
```

```
    iterative method where two modules are considered alike if their
    respective dependents are themselves alike."

Depth 0, current state S0:
- unresolved requirements: a node set selected by the local
    quadrilateral-pattern threshold, a node set selected by attenuated
     reachability influence, and pairwise structural resemblance
    inside the induced subgraph on the union of those sets;
- constraints to preserve: 120 modules, 600 mutual links, undirected,
    unweighted, simple, homogeneous structure, threshold 0.1, alpha
    0.05, top 15, union scope, and all-pairs output;
- operations: none; dependencies: none.

Depth 1, candidate generation from S0:
- Candidate A adds one broad operation that attempts to perform both
    selections and the final resemblance computation together.
- Candidate B adds one partial state containing v1 for the local
    quadrilateral-pattern values and threshold-0.1 node selection.
- Candidate C adds one partial state containing v1 for attenuated
    reachability influence with alpha 0.05 and the top-15 node
    selection.

Depth 1, same-rubric evaluation and search decision:
- A is infeasible because it conflates distinct requested outputs and
    hides the two concrete selected-set inputs required by the final
    operation.
- B and C are both valid but incomplete states. Each preserves one
    required independent selection operation and leaves the other
    requirements unresolved.
- Retain B and C in the beam; discard A.

Depth 2, one local expansion per retained state:
- From B, candidate B1 adds v2 for attenuated reachability influence
    with alpha 0.05 and top-15 selection, with no edge from v1 because
     both operations consume the original graph. Candidate B2 adds the
     same operation but also adds an unjustified v1 -> v2 edge.
- From C, candidate C1 adds v2 for the local quadrilateral-pattern
    values and threshold-0.1 selection, with no edge from v1.
    Candidate C2 adds the same operation but also adds an unjustified
    v1 -> v2 edge.
- B1 and C1 pass the dependency check and remain in the beam. B2 and
    C2 score lower and are discarded because one independent selection
     does not consume the other selection's output.

Depth 3, expand the retained partial states:
- From B1, candidate B1a adds v3 that consumes both selected node sets
    , forms their union, extracts exactly the induced subgraph, and
    returns pairwise structural resemblance for every pair in that
    scope. It adds both required edges v1 -> v3 and v2 -> v3.
    Candidate B1b adds a v3 that consumes only one selected set and is
     therefore incomplete.
- From C1, candidate C1a adds a join operation and both incoming edges
     but describes its graph scope only as "the resulting graph",
    omitting the required union-induced-subgraph contract. Candidate
    C1b forces the independent selections into a chain before the join
    .

Depth 3, same-rubric evaluation and search decision:
- B1a is the highest-scoring complete state: it covers all three
    requested results, preserves every parameter and scope, and
    represents the genuine two-input fan-in.
```

```
- B1b is incomplete, C1a loses a required scope constraint, and C1b
    contains an unsupported dependency. Discard them rather than
    combining pieces across branches.

Termination:
Select B1a as the highest-scoring complete path. This is Tree-of-
    Thought reasoning because several alternative workflow states are
    expanded and evaluated at each depth, a bounded beam is retained,
    and one complete branch is ultimately selected.

Final workflow DAG for the example:
{
  "requires_multi_algorithm": true,
  "subtasks": [
    {
      "id": "v1",
      "description": "In an undirected, unweighted, simple,
    homogeneous structure with 120 software modules connected by 600
    mutual integration links, for every module compute the fraction of
     possible 4-node closed loops among its neighbors that actually
    exist, and keep those modules where this fraction is at least
    0.1."
    },
    {
      "id": "v2",
      "description": "In the same undirected, unweighted, simple,
    homogeneous structure with 120 software modules and 600
    integration links, evaluate each module's overall reachability
    influence with attenuation factor (alpha) 0.05 -- accounting for
    both direct and indirect dependency paths with longer chains
    discounted -- and select the top 15 modules with the highest
    scores."
    },
    {
      "id": "v3",
      "description": "Take the union of the modules kept in v1 and the
     modules selected in v2, extract the induced subgraph they form,
    and compute pairwise structural resemblance for every pair of
    modules in that subgraph using an iterative method where two
    modules are considered alike if their respective dependents are
    themselves alike."
    }
  ],
  "edges": [
    {"id": "e1", "from": "v1", "to": "v3"},
    {"id": "e2", "from": "v2", "to": "v3"}
  ]
}

## 7. Project the selected ToT state
Select the highest-scoring complete state and project that state into
    its workflow DAG. Do not force the selected state to match the
    worked example or any predefined number of subtasks, number of
    edges, dependency pattern, depth, fan-in, fan-out, or value of `
    requires_multi_algorithm`. The problem and the winning state
    determine the DAG's shape.

The output contract fixes only the fields needed by the downstream
    evaluator; it does not prescribe a graph topology:
- Return one JSON object with exactly the top-level fields `
    requires_multi_algorithm`, `subtasks`, and `edges`.
```

```
- Determine `requires_multi_algorithm` from the operations actually
    required by the problem. Use true only for multiple distinct graph
    -operation capabilities; otherwise use false. Do not infer it from
     subtask count, edge presence, or sequence words.
- Include exactly as many subtasks as the selected complete state
    requires, with no arbitrary minimum beyond one and no arbitrary
    maximum. Give them consecutive IDs v1, v2, v3, ... in executable-
    description order. Each subtask object contains only `id` and a
    self-contained `description`.
- Derive every edge from concrete output consumption in the selected
    state. If no such dependencies exist, use `edges`: null. Otherwise
     include exactly the necessary edges, with consecutive IDs e1, e2,
     e3, ... . Do not add an edge merely to make the result sequential
    .
- Keep every relevant constraint in the appropriate description, but
    do not expose the search tree, rejected branches, scores,
    evaluations, or selection rationale as workflow nodes or extra
    fields.

Return only valid JSON for the dynamically selected DAG. Do not return
     markdown, explanations, comments, reasoning, metadata, or wrapper
     objects.
```

D.6 Task 1: Graph-of-Thought-Style Decomposition

Prompt for Graph-of-Thought Workflow Decomposition

```
You are a graph-analysis problem decomposition system.

Your task is to transform the given problem into one abstract workflow
     DAG of
meaningful graph-analysis subtasks. The final answer must use exactly
    the same
workflow-DAG output contract as the other benchmark methods.

## GoT execution protocol

### Stage 1: Seed a shared thought graph
Parse the problem into a contract before proposing operations. Extract
     every
requested result, including results reused later; the input graph and
    its exact
attributes, identifiers, sizes, and scope; all parameters, thresholds,
k-values, source/target and selection rules; explicit intermediate
    graph scopes;
and unresolved operation boundaries.

Create initial nodes for distinct candidate interpretations or partial
operations. Each node records its input object, one main graph-
    analysis action,
concrete output, scope, provenance, and unresolved requirements.
    Initial nodes
may be incompatible; do not force them into a tree or a single chain.

### Stage 2: Arbitrary graph expansion and transformation
Expand the shared graph one local change at a time. A new node may be
    created
from any existing node or partial subgraph, not only from the current
    leaf.
Allowed transformations are:
1. create a successor from any promising node;
```

```
2. connect a node to a compatible node from another root when a
    concrete output
   is consumed;
3. create a fan-in node that combines concrete outputs from multiple
    paths;
4. create fan-out when one concrete output is reused;
5. merge equivalent nodes while preserving unique constraints and
    outputs;
6. split a node only for distinct requested or reused outputs;
7. repair or delete an unjustified edge; and
8. create a refinement successor for a weak node or partial subgraph.

A newly generated node may consume outputs from non-sibling nodes, and
     an
aggregate node may be expanded again. The shared graph, not a path or
    branch,
is the unit of reasoning. Never add nodes for loading, sorting,
    filtering
mechanics, algorithm invocation, arithmetic, formatting, visualization
    , or
reporting unless that result is explicitly requested or reused later.

### Stage 3: Local evaluation, feedback, and graph rewriting
Evaluate individual nodes and relevant subgraphs, not only complete
    paths. Check
requested coverage, operation atomicity, input/output contracts, scope
    ,
incoming-edge compatibility, downstream usefulness, redundancy, and
    unsupported
assumptions.

Use scores to locate weak regions in the shared graph, not to select
    one isolated
winning branch. When two partial paths contain complementary
    information,
connect or aggregate them before generating the next operation. For a
    weak node,
create a repair/refinement successor that consumes the weak node and
    its
feedback, then re-evaluate affected connected regions. A node or
    aggregate may
be revisited after it has already been expanded. Permit a bounded
    internal cycle
of generate -> evaluate -> repair/merge -> re-evaluate. This
    optimization cycle
must not become a cycle in the final executable workflow.

### Stage 4: Graph-wide aggregation, convergence, and projection
Convergence requires a complete supported subgraph covering every
    requested
result, with producers for reused intermediate objects, preserved
    constraints
and scopes, atomic executable operations, and concrete information-
    flow edges.
Select this subgraph from the shared workspace, not from one pre-
    existing
branch. Merge duplicates, retain useful fan-in/fan-out, remove
    unsupported
edges, and break executable cycles. Project only executable operation
    nodes and
information-flow edges into the required workflow DAG. Hide roots,
    scores,
```

```
feedback links, contradiction links, provenance, candidate states, and
    the
internal graph trace.

### Mandatory GoT invariants
- Maintain one shared, addressable thought graph throughout reasoning.
    Do not
  generate isolated complete solution branches and select one winner.
- Permit multiple candidate nodes for the same unresolved operation.
    Evaluate,
  connect, merge, repair, or replace them locally without restarting
    the whole
  solution.
- Allow a node to consume compatible outputs from any existing nodes,
    including
  non-siblings and nodes created at different depths. Explicitly
    represent
  cross-root fan-in and reusable-output fan-out inside the thought
    graph.
- Evaluate both individual nodes and connected subgraphs. Feed local
    validation
  results back into the weak region, then re-expand that region while
    preserving
  correct nodes elsewhere in the graph.
- Internal proposal, feedback, contradiction, repair, and merge
    relations may
  form a bounded optimization cycle. The final projected workflow must
     contain
  only executable information-flow nodes and must remain acyclic.

Decomposition rules:
- Use the minimum number of meaningful graph-analysis subtasks.
- If only one graph-analysis operation is required, create exactly one
     subtask.
- Do not split implementation details, internal calculations, loading,
     sorting,
  formatting, visualization, reporting, or algorithm invocation.
- Split operations when they produce distinct requested outputs, when
    a later
  operation consumes a concrete output of an earlier operation, or
    when they are
  genuinely different graph-analysis operations.
- Preserve directionality, weightedness, graph type, node and edge
    counts,
  identifiers, thresholds, k-values, conditions, selection rules, and
    graph scope
  whenever stated.
Every final description must be self-contained and preserve, when
    relevant, graph attributes, size constraints, parameters,
    identifiers, thresholds, conditions, selection rules, input object
    , exact graph scope, one main graph-analysis action, and the
    concrete expected output and its type. If it consumes a previous
    result, name that result and its meaning directly, such as "the
    selected node set from v1" or "the score mapping produced by v1";
    never rely only on "then" or "previous step". Do not use vague
    wording such as "analyze the graph". Never mention algorithm names
    , expert-category names, code, libraries, or implementation
    details.
- Add an edge only when the destination requires the source's concrete
     output.
  Do not add edges merely because the problem uses "first", "then", or
     "finally".
```

```
- Independent subtasks must have no edges; use null for the edges
    field.
- Do not invent operations, parameters, graph properties, or
    dependencies.
- Do not mention algorithm names, code, libraries, data structures, or
     implementation
  details in subtask descriptions.

- Each final subtask must describe one executable graph-analysis
    operation.
- State the input object, one main action, relevant constraints and
    parameters,
  exact graph scope, and concrete output type.
- If it consumes an upstream result, name the producing subtask and
    result meaning.
- Keep the final description self-contained and precise; do not use
    vague wording.
- Do not expose internal calculations as nodes unless their output is
    requested or
  reused by another operation.

### Final subtask description contract
Every final subtask description must be self-contained. A downstream
    reader
must understand and execute it without referring to the original
    problem, the
hidden thought graph, another subtask's unstated reasoning, or omitted
     context.
Do not omit any constraint or parameter from the original problem when
     it is
relevant to that subtask.

For every final subtask, explicitly preserve, when relevant:
- graph attributes: directed or undirected structure, weighted or
    unweighted
  edges, graph type, graph size constraints, and special assumptions;
- every task parameter: specified nodes, source/target identifiers,
    thresholds,
  k-values, distance limits, constraints, conditions, and selection
    rules;
- the input object: the original graph or each named intermediate
    result it
  consumes;
- whether the operation applies to the original graph, a filtered
    graph, an
  induced subgraph, a component, or another exact intermediate scope;
    and
- the concrete expected output and its type, such as a node, node set,
     path,
  distance, score mapping, ranking, partition, edge set, subgraph,
    component,
  or scalar.

Use one precise main graph-analysis action per description. If a
    subtask
consumes an upstream result, name its subtask ID and the meaning of
    that result,
such as "the score mapping produced by v1" or "the selected node set
    produced
by v2". Do not rely only on "then", "after", or "previous step". Do
    not write
```

```
vague descriptions such as “analyze the graph” or “process the result”
    .

### Dependency and flag contract
Add an edge from A to B only when B requires A’s concrete output.
    Treat
“first”, “then”, “after”, and “finally” as dependency signals only
    when they
describe actual data flow. If B depends on multiple predecessors, add
    one edge
from each predecessor. Independent subtasks have no edges between them
    ; use
‘edges‘: null when all subtasks are independent.

Set ‘requires_multi_algorithm‘ to true only when the problem requires
    multiple
distinct graph-analysis operations or algorithm capabilities. Set it
    to false
when the problem can be accomplished by one coherent graph-analysis
    operation.
Base this decision on the inherent complexity of the problem, not
    merely on the
number of subtasks, the presence of an edge, or sequential wording.

Worked GoT topology example:
Use this example to understand shared-graph execution, not to copy its
     domain
facts, operation names, node counts, identifiers, scores, or output
    values.

Problem: "As a software configuration management specialist, I’m
    auditing the
modular structure of a large-scale enterprise system to identify
    consolidation
opportunities. The system contains 120 software modules connected by
    600 mutual
integration links, where each link represents a bidirectional
    dependency,
forming an undirected, unweighted, simple, homogeneous structure. I
    need to find
modules that are both tightly woven into local quadrilateral
    dependency patterns
and globally influential across the architecture. First, for every
    module I
compute the fraction of possible 4-node closed loops among its
    neighbors that
actually exist, and keep those modules where this fraction is at least
     0.1.
Independently, I evaluate each module’s overall reachability influence
     with
attenuation factor (alpha) 0.05 -- accounting for both direct and
    indirect
dependency paths with longer chains discounted -- and select the top
    15 modules
with the highest scores. I then take the union of these two module
    sets, extract
the induced subgraph they form, and compute pairwise structural
    resemblance for
every pair of modules in that subgraph using an iterative method where
     two
modules are considered alike if their respective dependents are
    themselves
```

```
alike."

Hidden shared thought graph construction (internal only; never output
    this trace):

1. Seed several partial nodes in one workspace rather than building
    complete
   competing solutions:
   - R1a proposes computing the local quadrilateral-pattern values but
     omits the
     thresholded node selection.
   - R1b contains both the per-module operation and the required
    threshold 0.1
     selection.
   - R2a proposes a general reachability-influence score but omits
    alpha and the
     requested cardinality.
   - R2b preserves alpha 0.05 and the top-15 selection.
   - J0 is a partial join candidate that recognizes that the two
    selected module
     sets must be combined, but has not yet preserved the full induced
    -subgraph
     scope and pairwise output contract.

2. Perform local evaluation and graph rewriting:
   - Compare R1a and R1b at the node level. Feed the missing-selection
     finding
     back to R1a, merge any unique valid content into R1b, and retain
    one complete
     node R1* instead of keeping two full branches.
   - Compare R2a and R2b locally. Repair the missing alpha 0.05 and
    top-15
     constraints, merge compatible content, and retain complete node
    R2*.
   - R1* and R2* remain independent executable roots because both
    consume the
     original graph and neither consumes the other's output.

3. Connect non-sibling results and continue expansion from the
    aggregate:
   - Add information-flow links R1* -> J0 and R2* -> J0. J0 is a cross
    -root
     fan-in node that can directly address both selected module sets.
   - Evaluate the connected subgraph {R1*, R2*, J0}. Feedback
    identifies that J0
     must form the union, extract exactly its induced subgraph, and
    return
     pairwise structural resemblance for every pair in that scope.
   - Create refinement successor J1 from J0 using that feedback. If
    another node
     separately duplicates the union or induced-subgraph construction,
     merge its
     useful scope constraint into J1 and remove the duplicate.
   - Re-evaluate only the affected connected region; do not regenerate
     the
     already valid R1* and R2* nodes.

4. Converge and project:
   - The shared thought graph has converged when R1*, R2*, and J1
    jointly cover
     all requested outputs and the incoming links to J1 match its
    concrete inputs.
```

```
   - Project R1*, R2*, and J1 to v1, v2, and v3. Preserve only the
    executable
     information-flow edges v1 -> v3 and v2 -> v3.
   - Hide rejected candidates R1a/R2a/J0, local scores, feedback,
    repair links,
     merge relations, and provenance. They are GoT optimization state,
     not
     workflow subtasks.

This is Graph-of-Thought reasoning because useful partial nodes
    coexist in one
shared graph, non-sibling outputs are connected, weak regions are
    repaired
locally, equivalent nodes are merged, and an aggregate is re-expanded
    after
feedback.

Final workflow DAG for the example:
{
  "requires_multi_algorithm": true,
  "subtasks": [
    {
      "id": "v1",
      "description": "In an undirected, unweighted, simple,
    homogeneous structure with 120 software modules connected by 600
    mutual integration links, for every module compute the fraction of
     possible 4-node closed loops among its neighbors that actually
    exist, and keep those modules where this fraction is at least
    0.1."
    },
    {
      "id": "v2",
      "description": "In the same undirected, unweighted, simple,
    homogeneous structure with 120 software modules and 600
    integration links, evaluate each module's overall reachability
    influence with attenuation factor (alpha) 0.05 -- accounting for
    both direct and indirect dependency paths with longer chains
    discounted -- and select the top 15 modules with the highest
    scores."
    },
    {
      "id": "v3",
      "description": "Take the union of the modules kept in v1 and the
     modules selected in v2, extract the induced subgraph they form,
    and compute pairwise structural resemblance for every pair of
    modules in that subgraph using an iterative method where two
    modules are considered alike if their respective dependents are
    themselves alike."
    }
  ],
  "edges": [
    {"id": "e1", "from": "v1", "to": "v3"},
    {"id": "e2", "from": "v2", "to": "v3"}
  ]
}

The final answer for the original problem must contain only the
    workflow DAG. Do not
return this internal state trace or any candidate-state metadata.

Do not output candidate thoughts, Thought Graphs, reasoning traces,
    scores,
```

```
aggregation decisions, validation objects, or any other extra fields.
    Return only
one valid JSON object with exactly these top-level fields:
{
  "requires_multi_algorithm": true,
  "subtasks": [
    {"id": "v1", "description": "..."}
  ],
  "edges": [
    {"id": "e1", "from": "v1", "to": "v2"}
  ]
}

Use "edges": null when all subtasks are independent. Use subtask IDs
    v1, v2,
v3, ... and edge IDs e1, e2, e3, ... . Do not output markdown fences,
explanations, chain-of-thought, or any additional fields.

Original problem:
{question}
```

D.7 TASK 1: SEMANTIC JUDGE

Prompt for Task-1 Semantic Judging

```
You are a strict evaluation expert for graph analysis workflow DAG
    generation. Your job is to judge whether a model's predicted
    workflow DAG matches the ground truth (standard answer).

Evaluate the prediction in two independent stages:

1. node_matches: align semantically equivalent standard and predicted
    subtasks.
2. dependency_judgments: evaluate standard dependency edges after
    applying the node alignment.

Use error_labels only to identify concrete errors. Do not judge or
    return the requires_multi_algorithm field. Do not output numeric
    scores, confidence, or free-form reasoning. Binary node, edge, and
     overall accuracies are computed by the evaluation program from
    your structured judgments.

TASK CONTEXT:
Question: {rephrased_question}
--- STANDARD ANSWER (Ground Truth) ---
Subtasks:
{fmt_subtasks(std_subtasks)}
Edges:
{fmt_edges(std_edges)}
--- MODEL PREDICTION ---
Subtasks:
{fmt_subtasks(pred_subtasks)}
Edges:
{fmt_edges(pred_edges)}
--- EVALUATION RULES ---
1. node_matches:
   - Match subtasks by operation identity, input graph or intermediate
     result, graph scope, parameters, constraints, and expected output
    .
   - Node IDs and wording are irrelevant when the semantics are
    equivalent.
```

```
- Use match_type "exact" for one-to-one semantic equivalence,
 including an exactly equivalent analytical output contract.
- Use match_type "split" when one standard subtask is represented
 by multiple predicted subtasks that together preserve the same
 executable workflow and the same terminal analytical output
 contract. Explicit loading, preprocessing, filtering, sorting,
 induced-subgraph construction, intermediate transformation, result
  selection, formatting, or saving may be part of a valid split
 when they support the same requested operation and preserve its
 data flow. An intermediate result may be computed and consumed
 internally, but it must not become an additional terminal output
 unless the standard answer requests it.
- Use match_type "incomplete" only when a predicted subtask is
 recognizably intended to perform the same operation as a standard
 subtask but omits, adds, or substitutes important graph attributes
 , parameters, scope, constraints, or analytical outputs.
 Incomplete is not used when the operation itself is replaced by a
 different operation or is absent entirely.
- Use match_type "merged" only when two or more different required
 graph-analysis operations are fused into one predicted subtask.
 The operations must be distinct analytical operations, not merely
 stages of one operation, data preparation, result selection,
 formatting, or saving. A merge is an error when it hides distinct
 algorithmic capabilities, independently required outputs, or a
 genuine dependency boundary.
- Use match_type "spurious" when a required graph-analysis
 operation is absent or replaced by a semantically different
 operation, or when a predicted subtask performs an operation
 unrelated to the requested workflow. A completely missing standard
  operation is represented with gold_ids containing the missing
 standard node and predicted_ids empty. An unrelated extra
 prediction is represented with gold_ids empty and its predicted ID
  in predicted_ids.
- Do not penalize a semantically faithful split merely because the
 prediction contains more nodes than the standard answer.
Strict output-contract rule (compare the prediction directly with
 the standard answer):
- Compare the output contract required by each standard subtask
 directly with the output contract declared by its matched Task_1
 prediction. Judge the semantic returned object, not surface
 wording: different wording for the same returned object is
 equivalent, while an explicit change to that object is an error.
- The output contract covers the semantic result type, value-versus
 -witness distinction, cardinality or granularity, scope, and
 explicitly returned fields/results. It is not merely the output's
 wording or serialization format.
- The predicted analytical outputs must be exactly equivalent to
 the outputs required by the standard answer. A strict superset is
 not equivalent: returning the required result plus an additional
 analytical result is an output-contract error, even if the
 required result is still present and correct.
- Therefore, when the main operation is recognizable but its output
  is omitted, expanded, narrowed, or replaced, do not use
 match_type "exact" or "split". Use match_type "incomplete" and
 include "wrong_output_type" in error_labels.
- Typical invalid expansions include: a scalar/count plus a witness
  set when only the scalar/count is required; a boolean plus
 component details when only the boolean is required; a graph-level
  aggregate plus per-node values when only the aggregate is
 required; a path length plus the full path when only the length is
  required; or a final result plus exposed intermediate scores/
 objects when only the final result is required.
```

```
- Harmless presentation details such as naming fields, ordering an
 already requested collection, or specifying JSON/table/file
 serialization do not change the analytical output contract,
 provided they add no new analytical result and do not change
 result scope or granularity.
Matching examples (use these as semantic guidance, not as literal
 templates):
- exact example 1:
  Standard v1: "In an undirected, unweighted, simple, homogeneous
 graph of 50 entities and 75 observed links, identify the 15
 entities with the strongest structural importance and output their
  ranked node list."
  Prediction u3: "For the same 50-node undirected, unweighted,
 simple, homogeneous graph, rank all entities by structural
 importance and return the top 15 nodes."
  Judgment: gold_ids=["v1"], predicted_ids=["u3"], match_type="
 exact". The graph scope, constraints, selection parameter, and
 output are equivalent.
- exact example 2:
  Standard v2: "Using the weighted reach scores produced for the 50
  vocabulary terms, assign each of the 120 links a derived cost
 from its endpoint scores and return a minimum-cost connected
 backbone over all terms."
  Prediction u2: "Consume the per-term reach values, derive one
 cost for every link from the two endpoint values, and output the
 connected spanning backbone with the lowest total derived cost."
  Judgment: gold_ids=["v2"], predicted_ids=["u2"], match_type="
 exact". The input intermediate result, edge-cost construction,
 graph-wide constraint, and output are preserved.
- split example 1:
  Standard v1: "In an undirected, unweighted, simple, homogeneous
 graph of 50 persons and 75 contact links, keep the 15 most
 structurally important persons and extract their induced subgraph
 ."
  Prediction u1: "Rank the 50 persons by structural importance and
 select the top 15 persons."
  Prediction u2: "Using the selected 15-person set from u1,
 construct and output the corresponding induced subgraph."
  Judgment: gold_ids=["v1"], predicted_ids=["u1", "u2"], match_type
 ="split". The prediction makes the selection and induced-subgraph
 construction explicit without changing the requested workflow.
- split example 2:
  Standard v1: "For a directed, weighted, simple, homogeneous
 network of 40 locations and 100 routes, compute the lowest-cost
 route structure from the specified source and return the resulting
  reachable distances and route records."
  Prediction u1: "Load the directed weighted network and retain the
  route costs for all 100 links."
  Prediction u2: "From the specified source, compute the lowest-
 cost distance to every reachable location."
  Prediction u3: "Reconstruct the corresponding route for each
 reachable location and save the distance-route records."
  Judgment: gold_ids=["v1"], predicted_ids=["u1", "u2", "u3"],
 match_type="split". Loading, distance computation, route
 reconstruction, and result serialization are accepted because they
  form one faithful executable workflow and preserve the requested
 outputs.
- incomplete example 1:
  Standard v1: "In an undirected, weighted graph of 60 nodes and
 140 links, compute the weighted reach score of every node and
 return the complete node-to-score mapping."
```

```
  Prediction u1: "Compute a reach score for the nodes in the graph
 ."
  Judgment: gold_ids=["v1"], predicted_ids=["u1"], match_type="
incomplete". The prediction is clearly attempting the same
operation but omits important graph attributes, coverage, and
output requirements.
- incomplete example 2:
  Standard v2: "Inside the induced subgraph containing the 15
selected entities from v1, measure local square-like reinforcement
 and return scores only for the five highest-ranked entities."
  Prediction u2: "Inside the induced subgraph containing the 15
selected entities from v1, measure local square-like reinforcement
 and return one score for every entity in the subgraph."
  Judgment: gold_ids=["v2"], predicted_ids=["u2"], match_type="
incomplete". The prediction recognizes the same operation and
graph scope, but does not preserve the required top-five output
restriction and returns an over-broad result.

- output-contract violation example 1:
  Standard v1: "Compute the minimum number of nodes whose removal
disconnects the graph and return that number."
  Prediction u1: "Compute the minimum number of nodes whose removal
 disconnects the graph and return both that number and one minimum
 node-cut set."
  Judgment: gold_ids=["v1"], predicted_ids=["u1"], match_type="
incomplete", and error_labels must include "wrong_output_type".
The cut set is an additional witness object, so count plus cut set
 is not output-equivalent to count alone.

- output-contract violation example 2:
  Standard v1: "Return the network's average node eccentricity."
  Prediction u1: "Return the average node eccentricity together
with the eccentricity of every node."
  Judgment: gold_ids=["v1"], predicted_ids=["u1"], match_type="
incomplete", and error_labels must include "wrong_output_type". A
graph-level aggregate plus a per-node mapping is a strict output
expansion.

- output-contract violation example 3:
  Standard v1: "Return the center nodes of the graph."
  Prediction u1: "Compute every node's eccentricity, use it to
identify the center nodes, and return only the center-node set."
  Judgment: gold_ids=["v1"], predicted_ids=["u1"], match_type="
exact". Computing eccentricities internally is compatible with the
 requested operation because the intermediate mapping is not
exposed as an additional output. If the prediction instead returns
 both the center-node set and the full eccentricity mapping, use
match_type="incomplete" and include "wrong_output_type".

- merged example 1:
  Standard v1: "In an undirected, unweighted, simple, homogeneous
graph of 50 entities and 75 contact links, rank every entity by
the change in the network's structural energy after its removal
and output the complete ranking."
  Standard v2: "In the same original graph, measure for every
entity how often it participates in locally reinforced square-like
 contact patterns and output one score per entity."
  Standard edges: null, because both analyses independently consume
 the original graph and neither uses the other's output.
  Prediction u1: "Rank every entity by the change in structural
energy after removal and measure each entity's participation in
```

```
locally reinforced square-like contact patterns, returning both
results together."
  Judgment: gold_ids=["v1", "v2"], predicted_ids=["u1"], match_type
="merged". Two different graph-analysis operations are fused into
one node even though they are independent tasks with distinct
outputs; this is not a valid split or a harmless implementation
detail.

- merged example 2:
  Standard v1: "In an undirected, weighted, simple, homogeneous
association network of 50 terms and 120 links, compute a weighted
reach score for every term and output the complete term-to-score
mapping."
  Standard v2: "In the same original network, identify groups of
terms whose connections are denser within groups than across
groups and output the resulting node grouping."
  Standard edges: null, because the grouping task does not consume
the reach-score mapping and both tasks operate independently on
the original network.
  Prediction u1: "Compute the weighted reach score of every term
and identify the dense within-group structure, returning the score
 mapping and the node grouping as one combined graph-analysis task
."
  Judgment: gold_ids=["v1", "v2"], predicted_ids=["u1"], match_type
="merged". A node-level weighted scoring operation and an
independent group-structure analysis are different graph
algorithms with separate outputs; combining them hides the
independent task boundary.

- spurious example 1:
  Standard v1: "Inside the induced subgraph containing the 15
selected entities from v1, measure local square-like reinforcement
 for every entity and output one score per entity."
  Prediction: no node performs this operation.
  Judgment: gold_ids=["v1"], predicted_ids=[], match_type="spurious
". The required graph-analysis operation is completely absent, so
there is no recognizable predicted counterpart to mark as
incomplete.

- spurious example 2:
  Standard v1: "Using the node scores from the previous step,
derive a cost for every link and construct the lowest-cost
connected backbone over all 50 nodes."
  Prediction u1: "Using the node scores, compute pairwise community
 separation for the nodes."
  Judgment: gold_ids=["v1"], predicted_ids=["u1"], match_type="
spurious". The prediction replaces the requested link-cost and
connected-backbone workflow with a semantically different graph
operation.

- Dependency judgment examples:
  1. matched with different node IDs:
     Standard subtasks: v1 selects the 15 highest-ranked entities;
v2 constructs the induced subgraph from v1's selected set.
Standard edge: ["v1", "v2"]. Prediction subtasks u1 and u2 perform
 the corresponding operations, with predicted edge ["u1", "u2"].
Output: {"gold_edge": ["v1", "v2"], "predicted_edges": [["u1", "u2
"]], "status": "matched"}.
  2. matched after a valid split:
     Standard v1 selects the 15 entities and constructs their
induced subgraph; standard v2 measures local structure inside that
 subgraph. Prediction u1 selects the entities, u2 constructs the
```

```
induced subgraph from u1, and u3 measures local structure inside
u2. Predicted edges are u1->u2 and u2->u3. Output: {"gold_edge":
["v1", "v2"], "predicted_edges": [["u1", "u2"], ["u2", "u3"]], "
status": "matched"}. A complete data-flow chain may represent one
standard dependency after a faithful split.
 3. missing dependency:
    Standard v1 produces the selected entity set and v2 constructs
 an induced subgraph from that set, so the standard edge is ["v1",
 "v2"]. The prediction contains both operations but v2 constructs
the subgraph directly from the original graph without consuming v1
's selected set. Output: {"gold_edge": ["v1", "v2"], "
predicted_edges": [], "status": "missing"} and include ["v1", "v2
"] in missing_critical_dependencies.
 4. reversed dependency:
    Standard v1 produces node scores and v2 uses those scores to
derive link costs, so the standard edge is ["v1", "v2"]. The
prediction contains the corresponding operations but outputs the
edge ["u2", "u1"]. Output: {"gold_edge": ["v1", "v2"], "
predicted_edges": [["u2", "u1"]], "status": "reversed"}.
 5. spurious dependency between independent tasks:
    Standard v1 computes structural importance on the original
graph and v2 independently measures local reinforcement on the
same original graph. The standard edges are null. If the
prediction adds ["u1", "u2"], output: {"gold_edge": [], "
predicted_edges": [["u1", "u2"]], "status": "spurious"} and
include ["u1", "u2"] in spurious_dependencies.
 6. one correct dependency plus one spurious dependency:
    Standard v1 produces node scores, v2 uses those scores to
derive link costs, and v3 independently identifies node groups
from the original graph. The only standard edge is ["v1", "v2"].
If prediction edges are ["u1", "u2"] and ["u1", "u3"], mark ["v1",
 "v2"] as matched by ["u1", "u2"] and mark ["u1", "u3"] as
spurious. Shared input graphs do not create dependencies.
 - An edge A -> B means B requires a concrete output produced by A
. Narrative order, shared input graphs, or the word "then" alone
do not justify an edge.
 - Match dependencies after applying node_matches. Edge direction
matters.
 - Use status "matched" only when the required data flow is
preserved; use "missing" when it is absent, "reversed" when its
direction is opposite, and "spurious" when an extra dependency
connects independent tasks.
 - For a valid split, map one standard edge to the complete
predicted data-flow chain and mark it matched only when every
required downstream operation receives the needed upstream output.
 - Output node_matches ordered by the earliest standard node
represented by each match object. Emit exactly one match object
for each standard node, including nodes that are incomplete or
missing. For a split, put all predicted IDs in the one object for
that standard node. For a merged case, put all affected standard
IDs in their standard order in one object; those IDs may be non-
contiguous in the global standard order. Do not repeat a standard
ID across match objects, and do not reuse a predicted ID across
match objects.
 - Output dependency_judgments in exactly the order of the
standard edges. Emit exactly one judgment for each standard edge,
including missing or reversed edges. Include the corresponding
gold edge ID when it is available. If the standard has no edges,
output an empty dependency_judgments array. Prediction-only edges
must not be inserted as fake gold edges; list them only in
spurious_dependencies.
```

```
     - unmatched_gold_nodes and unmatched_predicted_nodes are
    consistency fields. Every standard node must be represented in
    node_matches; every unmatched prediction-only node must be listed
    in unmatched_predicted_nodes. Keep these arrays empty when there
    are no such items.

4. error_labels:
   Use only labels from: incomplete_subtask, spurious_subtask,
    wrong_operation, wrong_parameter, wrong_graph_scope,
    wrong_output_type, merged_subtasks, missing_dependency,
    spurious_dependency, reversed_dependency, invalid_dag.

--- RESPONSE FORMAT ---
Return ONLY one valid JSON object. Do not include markdown, confidence
    , numeric scores, reasoning, or the requires_multi_algorithm field
    . Use empty arrays when there are no items.
{
  "node_matches": [
    {"gold_ids": ["v1"], "predicted_ids": ["v1"], "match_type": "exact
    "}
  ],
  "unmatched_gold_nodes": [],
  "unmatched_predicted_nodes": [],
  "dependency_judgments": [
    {"gold_edge_id": "e1", "gold_edge": ["v1", "v2"], "predicted_edges
    ": [["v1", "v2"]], "status": "matched"}
  ],
  "missing_critical_dependencies": [],
  "spurious_dependencies": [],
  "error_labels": []
}
```

D.8 Task 2: Flat KB Algorithm Ranking

Flat KB: Algorithm Ranking

```
You are a senior graph algorithm expert. Your task is to identify the
    most suitable graph algorithm(s) for solving the given subtask.

SUBTASK ID:
{subtask_id}

SUBTASK DESCRIPTION:
{subtask_description}

{f'CONTEXT: {context}' if context else ''}

GRAPH ATTRIBUTES:
{graph_attributes_str}

AVAILABLE ALGORITHMS:
{algorithms_str}

TASK INSTRUCTIONS:
1. Carefully understand the goal, input/output requirements, and
    implicit graph properties of the subtask.
2. Determine what type of graph problem the subtask represents, such
    as traversal, shortest path, connectivity, ranking, clustering,
    matching, flow, centrality, spanning tree, coloring, or
    optimization.
```

```
3. Treat the provided GRAPH ATTRIBUTES as authoritative constraints.
    Select only algorithms compatible with these attributes,
    especially directed, weighted, multigraph, and heterogeneous. Use
    num_nodes and num_edges to assess graph scale, density, and
    practical efficiency. Do not infer graph properties that conflict
    with the provided attributes.
4. Consider important constraints including negative weights,
    connectivity, scalability, and time/space complexity in addition
    to the provided graph attributes.
{selection_instruction}
6. Prefer algorithms that directly solve the core problem rather than
    algorithms that are only loosely related.
7. If multiple algorithms appear relevant, prioritize the one(s) with
    the best conceptual fit, correctness guarantee, and practical
    efficiency for the described subtask.
8. Provide a concise but technically sound justification.

RESPONSE FORMAT:
Please respond strictly in the following JSON format:

{response_format}

CRITICAL REQUIREMENTS - ALGORITHM NAME MATCHING:
{memory_note}
- The selected algorithm name(s) MUST match EXACTLY one/ones from the
    AVAILABLE ALGORITHMS list.
- Do NOT modify, abbreviate, paraphrase, translate, or reformat any
    algorithm name.
- Copy the algorithm name(s) directly from the AVAILABLE ALGORITHMS
    list.
- The output algorithm name(s) must be character-for-character
    identical to the corresponding name(s) in the AVAILABLE ALGORITHMS
     list.
- Do NOT output any algorithm that does not appear in the AVAILABLE
    ALGORITHMS list.
- Do NOT include extra text outside the required JSON object.
- Ensure the JSON is valid and parseable.
- Be precise, objective, and professional in your reasoning.
- If selecting multiple algorithms, rank them from most suitable to
    least suitable.
```

D.9 TASK 2: G1-COT-SFT-7B DIRECT PREDICTION

G1-CoT-SFT-7B: Direct Prediction

```
Select the best graph algorithm for the subtask below. Output ONLY the
     JSON, no explanation.

SUBTASK: {subtask_description}
{f'CONTEXT: {context}' if context else ''}

AVAILABLE ALGORITHMS:
{algorithms_str}

{memory_note}
Output ONLY valid JSON (replace each <algorithm_name> with a real
    algorithm name from the list above):
{response_format}
```

D.10 TASK 2: HIERARCHICAL KB CATEGORY ROUTING

Hierarchical KB: Category Routing

```
You are a senior graph algorithm expert. Your task is to identify the
    single most appropriate graph task type for the given subtask.

SUBTASK ID:
{subtask_id}

SUBTASK DESCRIPTION:
{subtask_description}

GRAPH ATTRIBUTES (AUTHORITATIVE):
{_format_graph_attributes(graph_attributes)}

{f'CONTEXT: {context}' if context else ''}

AVAILABLE TASK TYPES:
{task_types_str}

TASK INSTRUCTIONS:
1. Carefully understand the goal, input/output requirements, and
    implicit graph reasoning target of the subtask.
2. Compare the subtask only against the AVAILABLE TASK TYPES listed
    above.
3. Select exactly ONE task type id from the AVAILABLE TASK TYPES.
4. The selected task type id must be the best semantic match for the
    subtask.

RESPONSE FORMAT:
Please respond strictly in the following JSON format:

{"selected_task_type": "<task_type_id>"}

CRITICAL REQUIREMENTS - TASK TYPE MATCHING:
- The value of "selected_task_type" MUST exactly match one existing id
     from the AVAILABLE TASK TYPES.
- Valid task type ids are: {valid_ids_str}
- Do NOT invent any new id.
- Do NOT rewrite, paraphrase, translate, abbreviate, or modify an id.
- Do NOT output an algorithm name.
- Do NOT output a task_type label instead of the id.
- If you are uncertain, still choose the single best matching id from
    the AVAILABLE TASK TYPES list.
- Do NOT include extra text outside the required JSON object.
- Ensure the JSON is valid and parseable.
```

D.11 TASK 2: HIERARCHICAL RAG ALGORITHM RANKING

Hierarchical RAG: Algorithm Ranking

```
You are a senior graph algorithm expert. Your task is to identify the
    most suitable graph algorithm(s) for solving the given subtask.

SUBTASK ID:
{subtask_id}

SUBTASK DESCRIPTION:
{subtask_description}

GRAPH ATTRIBUTES (AUTHORITATIVE):
```

```
{graph_attributes_str}

{f'CONTEXT: {context}' if context else ''}

AVAILABLE ALGORITHMS:
{algorithms_str}
{relations_section}
TASK INSTRUCTIONS:
1. Carefully understand the goal, input/output requirements, and
    implicit graph properties of the subtask.
2. Determine what type of graph problem the subtask represents, such
    as traversal, shortest path, connectivity, ranking, clustering,
    matching, flow, centrality, spanning tree, coloring, or
    optimization.
3. Consider important constraints including graph directionality, edge
     weights, negative weights, connectivity, sparsity/density,
    scalability, and time/space complexity.
{selection_instruction}
4. Prefer algorithms that directly solve the core problem rather than
    algorithms that are only loosely related.
5. When CANDIDATE DISAMBIGUATION KNOWLEDGE is present, apply its
    decision rules only when the subtask provides the required
    evidence.
6. Treat equivalent_under as active only when every listed condition
    is explicitly stated or directly implied by the subtask and graph
    attributes.
7. If a distinction remains unresolved, do not invent facts; prefer
    the generic candidate unless the subtask explicitly commits to the
     specialized algorithm.
8. Provide a concise but technically sound justification.

RESPONSE FORMAT:
Please respond strictly in the following JSON format:

{response_format}

CRITICAL REQUIREMENTS - ALGORITHM NAME MATCHING:
{memory_note}
- The selected algorithm name(s) MUST match EXACTLY one/ones from the
    AVAILABLE ALGORITHMS list.
- Do NOT modify, abbreviate, paraphrase, translate, or reformat any
    algorithm name.
- Copy the algorithm name(s) directly from the AVAILABLE ALGORITHMS
    list.
- The output algorithm name(s) must be character-for-character
    identical to the corresponding name(s) in the AVAILABLE ALGORITHMS
     list.
- Do NOT output any algorithm that does not appear in the AVAILABLE
    ALGORITHMS list.
- Do NOT include extra text outside the required JSON object.
- Ensure the JSON is valid and parseable.
- Be precise, objective, and professional in your reasoning.
- If selecting multiple algorithms, rank them from most suitable to
    least suitable.
```

D.12 TASK 3: DIRECT LLM EXTRACTION

Direct LLM Extraction: Invocation Planning

```
You are an expert in graph algorithms and system design. Your task is
    to generate a complete algorithm invocation plan for ALL subtasks
    in a multi-algorithm workflow DAG.

ORIGINAL NATURAL LANGUAGE PROBLEM:
{original_question}

SELECTED ALGORITHMS FOR THE COMPLETE WORKFLOW:
{algorithms_text}

COMPLETE WORKFLOW DAG (JSON):
{dag_json}

{all_algo_specs if all_algo_specs else ''}

TASK INSTRUCTIONS:
Your goal is to specify the invocation parameters for ALL subtasks in
    the workflow.

CRITICAL: You must provide a result for EVERY subtask in the workflow,
     even if some have empty static_parameters.
OUTPUT CONTRACT: Return exactly one strict JSON object. Do not use
    Markdown code fences,
comments, explanatory prose, Python literals, placeholders, or
    executable expressions.
Use only JSON true, false, and null for boolean/null values.

For each subtask, provide:
1. **Static Parameters**: Translate the task's explicit execution
    semantics into concrete API parameters.
   - Use only parameter names defined in the algorithm's input schema.
   - A value does not need to appear as literal API syntax when the
    mapping is unambiguous; for example, named endpoints map to source
    /target parameters.
   - Do not include the input graph `G` or values produced by
    predecessor subtasks; those are dynamic inputs supplied by
    workflow dependencies.
   - When the current subtask explicitly requests a weighted, cost-
    based, or distance-based computation and the schema has a weight/
    distance parameter, use the stored edge-attribute key `"weight"`.
   - A weighted input graph alone does not mean every subtask should
    use weights. Keep hop-count, connection-count, and explicitly
    unweighted computations unweighted.
   - Do not copy graph summary facts into `static_parameters`, invent
    optional tuning choices, or add parameters merely because the
    schema declares a default.
   - Treat ranking size, display limits, and Top-N output requests as
    post-processing unless the schema explicitly defines a matching
    invocation parameter.
   - IMPORTANT: Even if no parameters are found, you must still
    include the subtask in your response with empty static_parameters.

2. **Data Dependencies**: Identify which outputs from predecessor
    subtasks are used as inputs to this subtask.
   - Use algorithm names (not node IDs like v1, v2) in dependency
    specifications.
   - Only add dependencies when a predecessor output is actually used
    as input.
```

```
   - If a predecessor's Boolean result only controls whether a
    downstream subtask executes, use `to_parameter: "condition"`;
    never pass that Boolean value to `G`.
   - Check the DAG edges and subtask descriptions to determine data
    flow.

Example input problem:
"As a security analyst, I study 50 test modules joined by 120
    dependency links in an undirected, unweighted, simple, homogeneous
     graph. To identify modules that could expose indirect access
    routes between parts of the test structure, I find the 10 modules
    that most often lie on shortest dependency routes between other
    modules and extract their induced subgraph. Within that selected
    subgraph, I identify the modules that are separated from the
    others by the fewest dependency steps, helping locate convenient
    access points for security checks and incident response."

Example output:
{
  "subtasks": [
    {
      "subtask_id": "v1",
      "algorithm": "betweenness_centrality",
      "static_parameters": {}
    },
    {
      "subtask_id": "v2",
      "algorithm": "closeness_centrality",
      "static_parameters": {}
    }
  ],
  "data_dependencies": [
    {
      "from_algorithm": "betweenness_centrality",
      "from_output": "induced_subgraph",
      "to_algorithm": "closeness_centrality",
      "to_parameter": "G"
    }
  ]
}

Please respond in the following JSON format:
{
  "subtasks": [
    {
      "subtask_id": "<subtask_id>",
      "algorithm": "<algorithm_name>",
      "static_parameters": {}
    }
  ],
  "data_dependencies": []
}

IMPORTANT NOTES:
- You MUST provide an entry for EVERY subtask in the DAG, even if its
    static_parameters is {}
- The number of entries in "subtasks" must match the number of
    subtasks in the input DAG
- For each subtask, use the original problem, subtask description, and
     algorithm schema together
- static_parameters must follow the schema and the explicitly
    requested execution semantics
```

```
- Never fill static_parameters with invented tuning values, graph
    summary facts, or predecessor outputs
- Use algorithm names (not node IDs) in data_dependencies
- Return the JSON object only, with no text before or after it
```

D.13 Task 3: Subtask-Level Parameter and Dependency Extraction

For each subtask, the subtask-level parameter-planning implementation makes one model call with the prompt below. The algorithm has already been selected by Task 2; this prompt extracts its explicitly stated static parameters. The implementation also asks for subtask-level data dependencies, while a separate workflow-level call extracts dependencies for the complete DAG.

All braces denote runtime substitutions. The algorithm-specification block is constructed from the selected algorithm's YAML entry. Each parameter is rendered as `- {parameter_name} ({parameter_type}): {parameter_description}{required_marker}{default_marker}`, where `required_marker` is either " [REQUIRED]" or empty and `default_marker` is either " [default: {default_value}]" or empty. The tuning-mechanism section is omitted when no tuning mechanisms are defined.

Subtask-Level Parameter and Dependency Extraction

```
You are an expert in graph algorithms and system design. Your task is
    to generate a complete algorithm invocation plan for a specific
    subtask within a multi-algorithm workflow DAG.

ORIGINAL NATURAL LANGUAGE PROBLEM:
{question}

SELECTED ALGORITHMS FOR THE COMPLETE WORKFLOW:
{algorithms_text}

COMPLETE WORKFLOW DAG (JSON):
{dag_json}

CURRENT SUBTASK ID: {subtask_id}
CURRENT SUBTASK DESCRIPTION: {description}
ALGORITHM TO USE FOR CURRENT SUBTASK: {algorithm}

ALGORITHM SPECIFICATIONS:
Algorithm: {algorithm}

INPUT SCHEMA - PARAMETERS:
{algorithm_parameter_lines}

TUNING MECHANISMS:
{tuning_mechanism_lines}

OUTPUT SCHEMA:
{algorithm_output_lines}

TASK INSTRUCTIONS:
Your goal is to specify the STATIC HYPERPARAMETERS for this algorithm
    node.

1. **Static Parameters**: Extract only concrete parameter values
    explicitly stated in the original problem or current subtask
    description.
   - Do not infer values from algorithm defaults, best practices,
    graph attributes, or required-parameter lists.
```

```
   - Do not include the input graph `G` or any value produced by
    another algorithm.
   - If a parameter is not explicitly stated, omit it from `
    static_parameters`.

3. **How to decide**: Use the algorithm specification and current
    subtask description to identify only explicitly stated parameter
    values.

   Example `rephrased_question`:
   "As a security analyst, I study 50 test modules joined by 120
    dependency links in an undirected, unweighted, simple, homogeneous
     graph. To identify modules that could expose indirect access
    routes between parts of the test structure, I find the 10 modules
    that most often lie on shortest dependency routes between other
    modules and extract their induced subgraph. Within that selected
    subgraph, I identify the modules that are separated from the
    others by the fewest dependency steps, helping locate convenient
    access points for security checks and incident response."

   Example selected algorithms:
   ["betweenness_centrality", "closeness_centrality"]

   Example complete DAG:
   {
     "requires_multi_algorithm": true,
     "subtasks": [
       {
         "id": "v1",
         "description": "In an undirected, unweighted, simple,
    homogeneous graph of 50 test modules joined by 120 dependency
    links, identify modules that could expose indirect access routes
    between parts of the test structure by finding the 10 modules that
     most often lie on shortest dependency routes between other
    modules and extracting their induced subgraph.",
         "algorithm": "betweenness_centrality"
       },
       {
         "id": "v2",
         "description": "Within the selected subgraph obtained in v1,
    identify the modules that are separated from the others by the
    fewest dependency steps, helping locate convenient access points
    for security checks and incident response.",
         "algorithm": "closeness_centrality"
       }
     ],
     "edges": [
       {"id": "e1", "from": "v1", "to": "v2"}
     ]
   }

   Correct dependency analysis for this example:
   - `betweenness_centrality` is the source algorithm.
   - Its `induced_subgraph` result is the source output.
   - `closeness_centrality` is the target algorithm because it
    operates on the selected subgraph.
   - Therefore `to_algorithm` must be `closeness_centrality`, while `
    induced_subgraph` must only appear as `from_output`.

   Example output:
   {
     "data_dependencies": [
```

```
        {
          "from_algorithm": "betweenness_centrality",
          "from_output": "induced_subgraph",
          "to_algorithm": "closeness_centrality",
          "to_parameter": "G"
        }
      ]
    }

    Additional rules:
    - A number explicitly stated in the problem, such as 10, belongs in
      `static_parameters` only when it is a parameter of the relevant
     algorithm.
    - For `weight`/`distance`, output the edge-attribute key `"weight"`
      only when that subtask explicitly requires weighted computation;
     for hop-count, connection-count, or explicitly unweighted
     computation, omit a `None` default or output `null` only when
     needed to override a non-`None` default. A weighted input graph
     alone does not imply that every subtask uses its weights.
    - If a predecessor's Boolean result only controls whether the
     current subtask executes, record it as a dependency with `
     to_parameter: "condition"`; never pass that Boolean value to `G`.
    - A DAG edge only indicates a possible workflow order. Add a
     dependency only when a predecessor output is actually used as a
     successor input.

Please respond in the following JSON format:
{
    "algorithm": "{algorithm}",
    "static_parameters": {
        "param_name_1": <concrete_value>,
        "param_name_2": <concrete_value>,
        ...
    },
    "data_dependencies": [
        {
            "from_algorithm": "<predecessor algorithm>",
            "from_output": "<predecessor output>",
            "to_algorithm": "{algorithm}",
            "to_parameter": "<current input parameter>"
        }
    ]
}

IMPORTANT NOTES:
- static_parameters must contain only values explicitly present in the
     original problem or current subtask description
- Never fill static_parameters with defaults, inferred values,
    required parameters, or predecessor outputs
- The `output` field is generated automatically from the algorithm
    specification; do not generate it
- Do not include `invocation_prompt`, `model_output`, or reasoning
    fields
```

Workflow-Level Dependency Extraction

```
You are an expert in graph algorithms and system design. Your task is
    to identify data dependencies between algorithms in a multi-
    algorithm workflow.

ORIGINAL NATURAL LANGUAGE PROBLEM:
```

```
{question}

SELECTED ALGORITHMS FOR THE WORKFLOW:
{algorithms_text}

WORKFLOW DAG (JSON):
{dag_json}

ALGORITHM OUTPUT SPECIFICATIONS:
{algo_specs_text}

TASK INSTRUCTIONS:
Analyze the workflow and identify which outputs from predecessor
    algorithms are used as inputs to successor algorithms.

For each data dependency, specify:
1. **from_algorithm**: The source algorithm name
2. **from_output**: The output field name from the source algorithm
3. **to_algorithm**: The target algorithm name
4. **to_parameter**: The input parameter name in the target algorithm

Rules:
- Use algorithm names (not node IDs like v1, v2)
- Only add dependencies when a predecessor output is actually used as
    input
- Check the DAG edges and subtask descriptions to determine data flow
- The input graph G is typically passed between algorithms

Please respond in the following JSON format:
{
  "data_dependencies": [
    {
      "from_algorithm": "<source_algorithm>",
      "from_output": "<output_name>",
      "to_algorithm": "<target_algorithm>",
      "to_parameter": "<parameter_name>"
    }
  ]
}
```