Plan-over-Graph: Towards Parallelizable LLM Agent Schedule

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have demonstrated exceptional abilities in task planning, enabling LLMs to dive and conquer complex agentic tasks. However, challenges related to parallel scheduling remain underexplored. This paper introduces a novel paradigm, *plan-over-graph*, where a real-world task is first decomposed into executable subtasks to construct an abstract task graph, then the abstract graph is leveraged to generate a plan for parallel execution that minimizes overall time cost. We design an automated and controllable pipeline to generate synthetic graphs and propose a two-stage training scheme to enhance the planning capability of complex, scalable graphs. Experimental results show that our *plan-over-graph* method significantly improves planning performance on both API-based LLMs and trainable open-sourced LLMs, naturally supporting parallel execution and demonstrating global efficiency by normalizing complex tasks as graphs. Further analysis confirms the scalability of our approach with respect to textual task descriptions and increasing graph complexity.

1 Introduction

The commendable progress in large language models (OpenAI, 2023; Templeton et al., 2024; Yang et al., 2024a) has facilitated the impressive capability of agents for complicated, interactive tasks (Yao et al., 2023b, 2022; Xi et al., 2024; Ma et al., 2024; Yang et al., 2024b). Recent studies have demonstrated that generating a plan before execution enhances agents' performance, referred to as *plan-then-execute* (Zhao et al., 2024; Hao et al., 2023; Liu et al., 2023; Zhang et al., 2025; Lin et al., 2024a). Planning integrates global knowledge, enabling overall coherence rather than just local optimality (Qiao et al., 2024; Ruan et al., 2024). Planning breaks down complex tasks into subtasks as single-step operations, which is especially crucial for tasks requiring intricate workflows and precise

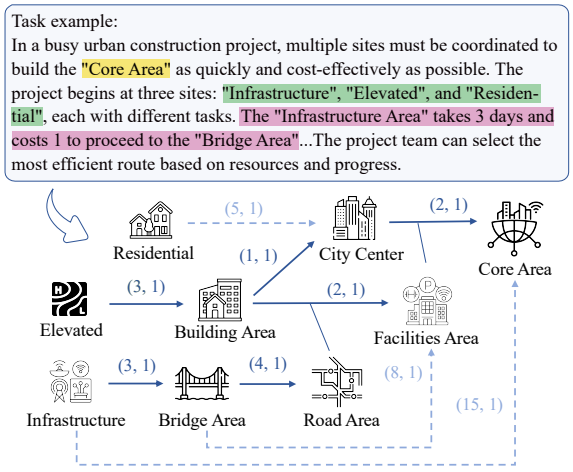

Figure 1: An example of our task: from a realistic textual query to a parallel plan. The plan is represented as a graph. Edges are available rules, and *Residential, Elevated, and Infrastructure* are the initial sources, *Core* being the target. The solid edges denote the optimal plan under the constraint of time consumption.

action interfaces, such as system control (Hong et al., 2024; Wu et al., 2024; Zhang et al., 2024a) and software engineering (Yang et al., 2024c).

Despite the inspiring progress, the parallelism of the plan remains underexplored. Multi-step agentic frameworks generally default to blocking pipelines, where each step waits for the previous ones to complete, regardless of whether it depends on their outcome (Wu et al., 2024; Gou et al., 2024). The reasoning capabilities of agents are significantly stimulated by Chain-of-Thoughts (CoT) (Wei et al., 2023), enabling them to divide and conquer a complex task. Although the reasoning structure is extended to trees and graphs (Yao et al., 2023a; Besta et al., 2024; Zhou et al., 2024), the actions for subtasks are still executed sequentially. However, these subtasks could be performed in parallel if they are independent. While recent studies (Lin et al., 2024b; Gonzalez-Pumariega et al., 2025) have explored the time efficiency of asynchronous execution, gaps remain when applied to real-world

scenarios, limited to small-scale evaluation and neglecting the uncertainty of subtasks.

Motivated by this, we dive into parallelism in planning for agentic tasks with complex task graphs, as the example in Figure 1. We propose *plan-over-graph*, a new paradigm where the agent first explores subtasks and extracts a graph, then plans on the graphic structure under global consumption constraints. To start with, we construct a dataset of complex tasks that involve parallel subtasks. Each sample is initialized with a connected directed acyclic graph, annotated with source and target nodes, along with feasible plans and the optimal solution. These graphs are further contextualized with appropriate scenarios by prompting an LLM to generate a textual description, finally forming realistic task descriptions in natural language.

We then propose a trainable scheme for the *plan-over-graph* paradigm. As graphs scale, graph comprehension becomes increasingly challenging for LLMs (Fatemi et al., 2024; Chen et al., 2024; Luo et al., 2024; Dai et al., 2024), creating a performance bottleneck. Hence, we conduct a two-stage training strategy on abstract graphs, with supervised fine-tuning for warm-up and direct preference optimization to reinforce optimal parallelism. To adapt to textual task descriptions, the LLM first extracts abstract graphs from textual queries and plans over them with the trained adapter. Our experiments show that *plan-over-graph* achieves significant performance gains on both API-based and open-sourced LLMs across comprehensive metrics, including success rate, optimal accuracy, and efficiency. We further analyze the impact of graph scalability, demonstrate how parallel execution improves time efficiency, and identify common errors in complex task planning.

Our contributions can be summarized as follows: (i) We define *plan-over-graph* for planning by enabling parallelism of sub-tasks, and construct a dataset of complex task graphs. (ii) We enhance *plan-over-graph* by training on task graphs and achieve significant improvement across LLMs. (iii) We analyze that our approach achieves planning efficiency and maintains robustness across both diverse LLMs and graph topologies.

2 Related Work

2.1 Planning for LLM-based Agents

Autonomous agents interact with environments to solve complex tasks (Yao et al., 2022; Fan et al., 2022). Planning involves developing an action sequence before execution, leveraging global knowledge of the task and environment to suggest a logically consistent trajectory (Huang et al., 2024).

Traditional planning approaches primarily rely on symbolic formalisms, e.g., Planning Domain Definition Language (PDDL) (Haslum et al., 2019) or reinforcement learning (RL) policies (He et al., 2016; Yao et al., 2020). Symbolic methods demand expert-driven domain specifications and lack error tolerance, while RL-based planners suffer from sample inefficiency, making them impractical when data collection is time-consuming or expensive. In contrast, LLM-based planning leverages natural language understanding to directly generate plans, marking a paradigm shift (Huang et al., 2024).

LLM-based planning automates the task decomposition based on global knowledge (Valmeekam et al., 2023; Wang et al., 2023; Wu et al., 2024). Searching strategies are applied to explore the optimal plan, such as depth- or breadth-first search and Monte Carlo tree search (Yao et al., 2023a; Zhou et al., 2024; Qi et al., 2024; Zhao et al., 2024). A world model or a reward model integrates global knowledge and predicts the environment states or estimates rewards (Hao et al., 2023; Qiao et al., 2024), providing granular guidance. Although multi-agent systems naturally allow concurrent execution (Yu et al., 2025; Jia et al., 2025), parallelizable planning within single-agent systems remains underexplored despite its importance.

2.2 Graph for LLM-based Agents

As realistic and intricate challenges can be formed as graphs, recent studies explore the LLMs' capabilities for reasoning with graphs. Graph-of-Thought (Besta et al., 2024; Ning et al., 2024) first proposes to transform the problem thinking into an arbitrary graph to enable the generation, aggregation, and refining of sub-tasks. Divide-Then-Aggregate (Zhu et al., 2025) transforms tree-based tool invocations into a DAG, enabling parallel subtask calls and result aggregation. PLaG (Lin et al., 2024b) combines graphs with natural language prompts for reasoning about asynchronous plans in real-life tasks, instructing the model to either reason based on a given graph or to generate a graph itself and then reason about it.

However, it is demonstrated that the capabilities of graph reasoning and understanding decrease as the scale and complexity of graphs increase. Empirical studies have observed a "comprehension

collapse" phenomenon as the graph size increases (Sui et al., 2023; Cao et al., 2024). DARG (Zhang et al., 2024b) evaluates LLMs' reasoning capability on graphs and also reports a performance decrease with increasing complexity of graphs.

3 Preliminary: Problem Statement

3.1 Formulation of Planning

Planning requires an agent to decompose a high-level task description into executable subtasks, schedule their execution under dependencies, optimize for time, cost, or multi-objective criteria, and finally achieve the goal. Formally, given a task description, the model generates a plan P by solving the planning problem defined by the tuple $\langle G, \Omega \rangle$, where G denotes the complex task and Ω denotes the global criteria.

Task Representation Any high-level task description can be represented as a Directed Acyclic Graph (DAG),

$$G = (T, E), \quad T = \{t_1, t_2, \ldots, t_n\}, \tag{1}$$

where vertices represent subtasks and edges represent precedence constraints: $t_i \prec t_j$ means t_i must precede t_j. A feasible plan P is a subgraph of G that includes a subset of subtasks and respects the precedence constraints among them, such that executing them accomplishes the original task. Let $\mathcal{F}(G)$ denote the set of all feasible plans for G.

Optimality Criterion In realistic scenarios, there are criteria for constraints like execution time. We formally define a global function $\Omega(P)$ that measures the quality of a plan, where the optimal plan minimizes Ω while achieving the task:

$$P_{\text{opt}} = \arg \min_{P \in \mathcal{F}(G)} \Omega(P). \tag{2}$$

Evaluation Metrics The measurements examine whether the predicted plan is optimal or at least feasible, and also compute the metrics of global criteria. Specifically, given each predicted plan $\hat{P}(G)$ on a test dataset D, we adopt three metrics to evaluate its helpfulness:

Optimal Rate: The proportion of optimal plans,

$$\text{OR} = \frac{1}{|D|} \sum_{G \in D} \mathbf{1}\left[\hat{P}(G) = P_{\text{opt}}(G)\right]. \tag{3}$$

Success Rate: The proportion of plans that successfully achieve the goal,

$$\text{SR} = \frac{1}{|D|} \sum_{G \in D} \mathbf{1}\left[\hat{P}(G) \in \mathcal{F}(G)\right]. \tag{4}$$

Global Criterion Value: The value of the global criterion Ω evaluated on the predicted plan, denoted as $\Omega(\hat{P}(G))$, which is a task-specific objective instantiated as time or cost depending on the setting.

3.2 Challenges of Planning

We designed experiments for a pilot study considering two critical limitations left by existing explorations: First, understanding graphs is currently a bottleneck in complex task planning for LLMs. (Lin et al., 2024b) has shown that even with explicit graph representations, there is still a huge gap in handling complex graph topologies, suggesting unresolved challenges in structural reasoning. Second, the scale of the currently considered graph is still very limited. WorFBench (Qiao et al., 2025) considered graphs where the majority of nodes lie within 2 to 10 steps. In AsyncHow (Lin et al., 2024b), most of the graph complexity $|V| + |E|$ are also between 10 and 20.

(i) We construct 100 random graphs with 10, 30, 50 nodes and ask LLM to find the shortest path as the solution. Table 1 shows the accuracy of feasible and optimal paths, where the performance decreases sharply as the number of nodes increases.

Node Count	Optimal Rate ↑	Success Rate ↑
10	29.0	79.0
30	16.0	35.0
50	6.0	10.0

Table 1: Llama-3.1 on random graphs.

(ii) To test whether conventional deterministic algorithms can solve planning when a task graph is well-defined, we first instructed GPT-4o (Hurst et al., 2024) to write Python code given the instruction and a graph. Surprisingly, the success rate was 0% over 10 samples, demonstrating LLMs' inability to implement algorithmic solutions to resolve the planning challenge, even though a code interpreter is provided as an available tool.

These findings demonstrate that the core bottleneck lies in planning on complex graph topologies with constraints, inspiring us to prioritize graph comprehension to enhance the planning task.

4 Methodology: Plan-over-Graph

Motivated by these findings, we propose the *plan-over-graph* paradigm, where the LLM is prompted to gather information and build the task graph, on which we have the model perform planning. Then, we focus on enhancing the graph comprehension for planning by a two-stage training.

Sections 4.1 and 4.2 formulate the task and introduce a data construction method to acquire control-

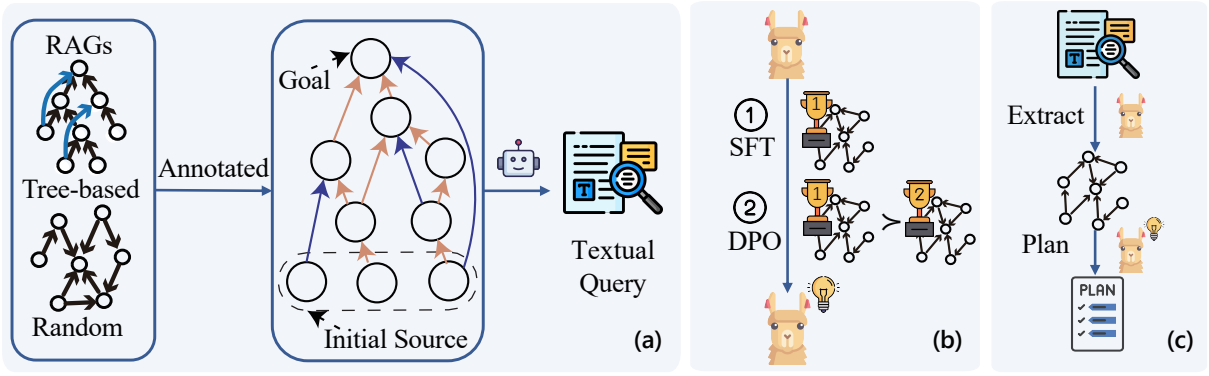

Figure 2: The overview of our framework. (a) shows the data synthesis pipeline; (b) shows our training process; (c) displays the plan-over-graph paradigm.

lable graph data automatically. Section 4.3 presents our training pipeline with this graph data to form the *plan-over-graph* paradigm for inference. Our overall framework is shown in Figure 2.

4.1 Abstract Task Formulation

First, we redefine the planning task on the graph structure. Without the loss of generality, we consider time and cost limits for Ω, the plan needs to minimize makespan or total cost.

Task Graph We define *rules*, $R = \{r_i\}$, that allow parallel execution on a graph. Each rule $r = (S, t, \tau, c)$ states that after S is satisfied, t can be executed with the required time τ and cost c. In our setting, if a rule's condition S is met, it can be performed at any time, which means that the fulfillment of the condition is persistent.

Query A query including prior knowledge and ultimate goal can be denoted as the initial node set and the target node as $q = (I, t_{\text{target}})$.

Plan The overall plan P is a set of several sub-plans, each denoted as

$$p = (S, t, D), \quad p_j \in D_i \iff t_j \in S_i, \tag{5}$$

where S and t are the preceding vertices and the subtask, determining a rule $r \in R$. D captures the dependencies between sub-plans.

Criteria Here, we define the two considered global criteria, time consumption and cost. Subtasks that are independent of each other can be executed in parallel. In other words, p_i only needs to wait for its dependency to end before starting execution. The end time of the execution of p_i can be expressed as:

$$\begin{aligned} \text{End\_time}(p_i) &= \max_{p_j \in D_i} (\text{End\_time}(p_j)) + \tau_{p_i}, \\ \Omega_{\text{time}}(P) &= \max_{p \in P} (\text{End\_time}(p)). \end{aligned} \tag{6}$$

The global cost is defined as the sum of the cost values of all subtasks, as an auxiliary criterion. Hence, P_{opt} minimizes the weighted sum, where $\epsilon \ll 1$ prioritizes time over cost,

$$\begin{aligned} \Omega_{\text{cost}}(P) &= \sum_{p \in P} c_p, \\ \Omega(P) &= \Omega_{\text{time}}(P) + \epsilon \cdot \Omega_{\text{cost}}(P). \end{aligned} \tag{7}$$

According to our graph definition, the Ω is measured by Time Efficiency (TE) and Cost Efficiency (CE) as follows:

$$\text{TE} = \frac{1}{n} \sum_{i=1}^{n} \frac{\Omega_{\text{time}}(P_i)}{\Omega_{\text{time}}(P_{opt})}, \quad \text{CE} = \frac{1}{n} \sum_{i=1}^{n} \frac{\Omega_{\text{cost}}(P_i)}{\Omega_{\text{cost}}(P_{opt})}. \tag{8}$$

Failed plans are penalized with a time and cost ratio of 4, with details in Appendix C.

4.2 Data Simulation

Following our definition above, we design an automated and scalable pipeline to generate synthetic data, which consists of the following steps:

◦ Generate a connected DAG. Two distinct graph structures are employed: (i) Random DAGs. (ii) To better conform to the hierarchical structure in reality and avoid excessive shortcuts in a random graph, we also construct a tree-based structure. We first construct a tree with a depth constraint of no more than 4, and then, a small number of ancestral and cross edges are added to introduce additional dependencies and enrich the graph structure. The edges are all directed to the root. The structural trade-offs between these two graph representations are analyzed in Section 6.1. The graph is ensured to be connected, meaning there exists at least one path from the initial vertices to the target.

◦ Define rules. For each non-head subtask t in the DAG, its predecessor vertices are randomly partitioned into groups as uniformly as possible. Then each predecessor group and t forms the source and target of one rule. Each rule is assigned a random

time value, which is sampled from a uniform distribution ranging from 1 to 50. The cost for all rules is fixed to 1, simplifying the cost structure while maintaining the focus on time optimization.

◦ Define initial and target nodes. All head vertices in the DAG are designated as the initial source vertices for the entire task. A target vertex is randomly selected from the tail vertices, ensuring that the task has a well-defined goal.

◦ Annotate optimal and feasible solutions. A dynamic programming algorithm is applied to compute the labels of each solution of the graphs. The optimal solution is the gold label.

◦ Construct query descriptions. We generate textual queries based on those graphs. We prompt an LLM to transform graphs into real-life scenario descriptions. To ensure consistency between the generated task and the original graph, we let the LLM perform the self-correction to verify that the query description matches the original graph.

4.3 Training Scheme

This section focuses on optimizing LLMs for graph comprehension and parallel planning via two-stage training: warm-up and preference optimization.

Warm-up We fine-tune an LLM on our abstract task datasets. This enables the model to solve planning tasks using graph representations of the problem space. We use the Low-Rank Adaptation (LoRA) method (Hu et al., 2022), which allows efficient adaptation of large pre-trained models by learning a small-sized adapter. Two setups are considered for training: (i) Fine-tuning with optimal data instances; (ii) Mixing the second-best solutions with the optimal solutions to enable the model to learn both optimal and feasible solutions.

Preference Optimization Following fine-tuning, direct preference optimization (DPO) (Rafailov et al., 2023) is applied to distinguish the optimal solution from feasible ones. For each sample, the second-best solution works as the rejected output, while optimal solutions are the chosen output. This step further refines the model's ability to prioritize optimal solutions over feasible ones. We also evaluate a variant trained solely with DPO, i.e., without preceding supervised fine-tuning; detailed descriptions for this setting are presented in Appendix F.

After training, we aggregate the extraction and planning steps during the inference. Given a query with a goal description, we first extract the task graph from the description, then we generate the plan on the graph with the trained adapter loaded.

5 Experiment

5.1 Dataset

Task Graph The theoretical edge of a connected graph with node count n range spans from $n-1$ to $n(n-1)/2$. Massive edges lead to excessively long input when n is large. And tree-based graph structure also cannot have too many edges. Therefore, we adopt two practical strategies to avoid this: (i) linear scaling with edges $\in [2n, 3n]$ for random graphs and $\in [n, 1.5n]$ for tree-based graphs; (ii) uniform distribution across the full edge range, which is only used on random graphs.

The training set contains 12,000 training instances, divided equally across three node scales (10, 30, 50 nodes) and random and tree-based DAG structures. It employs a uniform distribution edge configuration for 10/30-nodes graphs but restricts 50-node graphs to linear scaling. We generate 1000 input instances for each node scale and graph structure. Each input corresponds to an optimal solution and a chosen feasible solution. The testing set comprises two components: (i) baseline tests with linear edge scaling across node counts (10, 20, 30, 40, 50 nodes), each node count and graph structure containing 100 instances; (ii) edge-variation tests specifically for 10/30-nodes random graphs with uniformly distributed edges, to evaluate the model's ability to understand graphs as the number of edges changes. Due to the wide range of edge counts, we generate 1,000 instances each. The statistics of our synthetic data are shown in Table 5 in Appendix B.

Textual Query To systematically evaluate the parallel planning capacity of the model in real-world task scenarios and validate our *plan-over-graph* paradigm, we construct an evaluation dataset utilizing the DeepSeek-R1 (DeepSeek-AI et al., 2025) model. This dataset synthesizes 200 tasks derived from some real-world problem domains, where each task specification is transformed from graphs into executable workflow descriptions. The query data statistics are outlined in Appendix D.

We additionally evaluate our method on Robotouille (Gonzalez-Pumariega et al., 2025), with full experimental details in Appendix J.

5.2 Baseline

We evaluate our method against baseline models including API-based GPT-4o (Hurst et al., 2024) and Claude 3.5 Sonnet (Anthropic, 2024), and open-

source Llama-3.1-8B-Instruct (Dubey et al., 2024) and Qwen2.5-7B-Instruct (Yang et al., 2024a). Following existing work (Lin et al., 2024b), we also compare with chain-of-thought (Wei et al., 2023) and the PLaG (Lin et al., 2024b) for each setting. Detailed implementations are in Appendix E.

5.3 Main Results

Table 2 presents experimental results. Our results answer the following three key questions.

Q1: Can our training method teach the LLMs to plan on the graph? Which method yields the best performance? The results for each model are shown in the upper part of Table 2. Overall, **training largely improves the planning performance and the two-stage training achieves even better scores.** Without training, Claude demonstrates high success rates (90.0%) but relatively lower optimal rates (39.2%). GPT-4o and Llama have similar success rates (51.3% and 52.3%), but GPT-4o achieves a higher optimal rate (14.1%) than Llama's 1.8%. Qwen shows weaker performance in both success and optimal rates (13.2% and 0.5%). When applying CoT prompting to Llama, both success and optimal rates dropped to 31.8% and 0.5%, respectively. This degradation suggests that the model's intrinsic reasoning capacity may be insufficient, causing CoT to mislead it into incorrect solutions.

Only training on optimal solutions significantly improves graph understanding and planning ability, leading to a 75.7% success rate and a 61.4% optimal rate for Llama. Mixing feasible solutions further improves the performance to 86.1% and 67.5%. The best results of optimal rates (71.6%) are derived from the two-stage training, combining SFT on mixed data and DPO, maintaining high success rates (83.6%). This is because the model further tends to choose the optimal solution through DPO. We also observed consistent performance improvements on Qwen, which is inferior to Llama.

Q2: What phenomena does the expansion of the graph scale lead to? As shown in the upper part of Figure 3, when the number of nodes increases, which leads to a larger graph structure, the overall performance of all models decreases. For the time and cost ratio, these models have shown similar sensitivity to the node count. However, the cost ratio of Llama is quite obviously increased, which shows its tendency to select more subtasks on larger graphs. For success rate, GPT-4o and Llama drop sensitively. When the number of nodes reaches 30, GPT-4o even falls below Llama. However, Claude and our trained model continue to demonstrate strong capabilities with less sensitive drops. Claude still suffers between 10 and 20 points on the optimal rate, indicating a gap in the understanding of larger-scale graphs. Across all node counts, our trained Llama significantly outperforms all other models on the optimal rate.

The lower part of Figure 3 shows the results of Claude and our trained Llama on the full range of edge counts for 1000 cases with 10/30-nodes, respectively. For 10 nodes, due to the number of nodes being small, both models have demonstrated robustness to changes in the number of edges across the metrics. For 30 nodes, as the number of edges increases, the difficulty for the model to find the optimal solution increases more significantly. Therefore, the average time ratio for both is on the rise. The average cost ratio increases less significantly because as the graph becomes denser, the number of feasible solutions also increases, allowing the model to complete tasks by selecting fewer subtasks, though not in the most optimal time.

Q3: Can our plan-over-graph method improve the planning performance on textual queries?

The lower part of Table 2 presents the results of real-life queries, **showing that our plan-over-graph method consistently improves different models' performance**. Without extraction, Claude achieves a 89.5% success rate but struggles with the 14.5% optimal rate. Adding CoT to Claude raises the optimal rate from 50.0% and lifts the success rate to 97.0%, while applying PLaG further increases the optimal rate to 57.5% but brings success down to 89.0%. In contrast, Llama's weak reasoning yields only a 20.0% success rate with CoT and suffers a drop to 16.0% under PLaG. With extraction, the basic Extract+Plan setup raises Claude to 41.5% optimality and 93.5% success and gives Llama its first gains with 3.5% optimality and 38.0% success. Adding CoT on planning after extraction brings Claude to its peak—65.5% optimality and 99.5% success—demonstrating the value of our extraction module. Finally, the Llama-trained planner on extracted graphs achieves the highest optimal rate at 72.5% and a success rate of 83.0%, showing that even a weaker base model can reach top performance after training.

Compared to methods like CoT and PLaG that only benefit strong reasoners like Claude, the combination of our abstract graph extraction with CoT or our trained planner delivers consistently high optimality and success across different models.

Results on Abstract Graphs					
Model	**Optimal Rate ↑**	**Success Rate ↑**	**Feasible Rate**	**Avg Time Ratio ↓**	**Avg Cost Ratio ↓**
Claude 3.5 Sonnet	39.2	**90.0**	50.8	1.545	1.589
GPT-4o	14.1	51.3	37.2	2.657	2.889
Llama-3.1	1.8	52.3	50.5	2.616	4.512
Llama-3.1$_{\text{DPO}}$	1.0-0.8	3.4-48.9	2.4	3.898+1.282	3.954-0.558
Llama-3.1$_{\text{CoT}}$	0.5-0.8	31.8-20.5	31.3	3.175+0.556	4.190-0.322
Llama-3.1$_{\text{opt SFT}}$	61.4+59.6	75.7+23.4	14.3	1.746-0.870	1.769-2.743
Llama-3.1$_{\text{opt+feas SFT}}$	67.5+65.7	86.1+33.8	18.6	1.507-1.109	**1.498**-3.014
Llama-3.1$_{\text{opt+feas SFT + opt DPO}}$	**71.6**+69.8	83.6+31.3	12.0	**1.502**-1.114	1.520-2.992
Qwen2.5	0.5	13.2	12.7	3.612	4.072
Qwen2.5$_{\text{opt+feas SFT + opt DPO}}$	27.0+26.5	75.8+62.6	48.8	2.191-1.421	2.029-2.043
Results on Textual Query					
Method	**Optimal Rate ↑**	**Success Rate ↑**	**Feasible Rate**	**Avg Time Ratio ↓**	**Avg Cost Ratio ↓**
Claude Plan	14.5	89.5	75.0	1.904	2.302
Claude CoT Plan	50.0+35.5	97.0+7.5	47.0	1.535-0.369	1.522-0.780
Claude PLaG Plan	57.5+43.0	89.0-0.5	31.5	1.387-0.517	1.715-0.587
Claude Extract + Plan	41.5+27.0	93.5+4.0	52.0	1.514-0.390	1.689-0.613
Claude Extract + CoT Plan	65.5+51.0	**99.5**+10.0	34.0	**1.127**-0.777	**1.190**-1.112
Llama Plan	0.0	19.0	19.0	3.433	4.103
Llama CoT Plan	0.0	20.0+1.0	20.0	3.751+0.318	4.006-0.097
Llama PLaG Plan	0.0	16.0-3.0	16.0	3.546+0.113	4.109+0.006
Llama Extract + Plan	3.5+3.5	38.0+19.0	34.5	2.952-0.481	3.553-0.550
Llama Extract + CoT Plan	0.0	31.5+12.5	31.5	3.123-0.310	3.968-0.135
Llama Extract + Llama-trained Plan	**72.5**+72.5	83.0+64.0	17.0	1.540-1.893	1.526-2.577

Table 2: Main results. The upper part shows results on all baseline test sets; the lower part shows results on real-life tasks.

6 Analysis

6.1 Graph Features

This section discusses (i) our considerations regarding the graph structure; (ii) the impact of changes in the number of nodes and edges in the graph on the planning capability of the model.

Graph Structures The tree-based structure is designed to better reflect real-world parallel scenarios, offering a stronger hierarchical organization that facilitates the generation of more reasonable specific scenarios. To ensure a clear distinction between parallel and non-parallel execution, we implement the following strategies: (i) controlling the depth of the tree to manage the number of branches, and (ii) grouping nodes such that the number of nodes in each group does not exceed two-thirds of the total number of predecessor nodes. In addition, we also used an undefined random graph structure to verify the robustness of the model.

Impact of Node and Edge Counts We calculated the absolute value of correlation coefficients and slopes of normalized four metrics with changes in the number of points and edges. Overall, the impact of the edge count is smaller than the node count. For node counts, almost all metrics of the models showed strong correlation coefficients between 0.8 and 1.0. For edge variations on 10 nodes, both model shows low correlation coefficients which are less than 0.5, indicating robustness to changes in the number of edges on graphs with fewer points. However, on 30 nodes, Claude has higher correlation coefficients on all four metrics than our trained Llama, which are more than 0.7 correlation coefficients, demonstrating lower stability. Please refer to Appendix I for details.

Node Count	**Optimal**		**Llama-trained**		**Qwen-trained**	
	R	T	R	T	R	T
10	0.88	0.92	0.88	0.92	0.88	0.93
20	0.76	0.74	0.77	0.80	0.79	0.79
30	0.74	0.75	0.75	0.75	0.73	0.68
40	0.70	0.68	0.71	0.68	0.68	0.61
50	0.68	0.62	0.73	0.61	0.70	0.56

Table 3: The ratio of the parallel execution time of the plans provided by each model at the test cases to the sequential execution time. R represents the random graph structure, and T represents the tree-based structure.

6.2 Time Efficiency

Our tasks inherently support parallel execution of subtasks, yet most existing methods do not consider parallelism during planning, leading to unnecessary waiting times.

We calculate the time ratio of parallel execution to sequential execution (that is, the sum of all subtask durations) in the plans. Table 3 shows the results of optimal labels and outputs of our trained Llama and Qwen. The results demonstrate that the capability of planning parallel solutions can significantly reduce time compared to blocking sequential execution. Such efficiency is more significant as

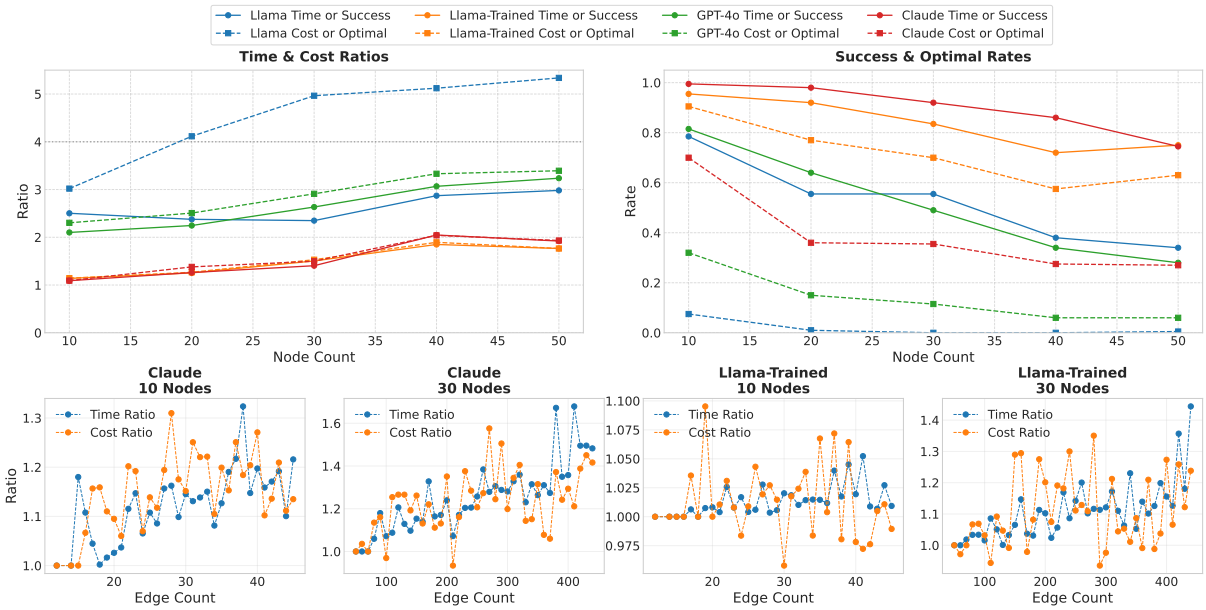

Figure 3: The upper part of this figure shows model performance across different node counts. The left plot shows the time and cost ratio change with the number of points, and the right shows the success and optimal rate. The lower part shows the average time and cost ratio across different edge counts of Claude and our trained Llama.

the graph scales. Specifically, plans from Llama and Qwen show high cost ratios, indicating that there are many redundant subtasks. When executing these plans sequentially, the inefficiency will be further amplified, leading to a low ratio of the parallel execution time to the sequential.

6.3 Wrong Case Study

Task Graph The wrong cases on abstract graphs fall into two types. (i) *Invalid subtask*, where the plan includes subtasks without corresponding transformation rules, and (ii) *Unavailable source*, where the required source for a subtask is not achieved during execution. The latter indicates either a failure to consider the source availability during planning or an incorrect handling of dependencies. Table 4 shows the proportion of two error types. After training, the source dependency has almost been resolved, but the hallucination of invalid subtasks is currently the performance bottleneck.

Model	Invalid Subtask	Unavailable Source
Claude 3.5 Sonnet	0.4	9.6
GPT-4o	4.7	44.0
Llama-3.1-8B-Instruct	17.6	30.1
Llama-3.1-8B-Instruct-Trained	11.6	4.8
Qwen2.5-7B-Instruct	60.7	26.1
Qwen2.5-7B-Instruct-Trained	19.9	4.3

Table 4: The proportion of two error causes in all test cases of the model.

Textual Query Failure to extract essential rules for subtasks will compromise the model's overall performance. However, interestingly, even with extraction errors, the model can still complete the task correctly if the subsequent planning does not encounter incorrect rules. Results of Llama show that the extraction step significantly improves the baseline success rate, and while the trained model's success rate is slightly lower than that of Claude, its optimal rate is superior. After taking a closer look, we found that only 15% matched the original graph exactly. However, we computed the F_1 scores for the model-extracted initial sources, target, and rules against the ground truth and averaged them to yield an overall similarity of 92.3%, indicating minimal impact. This supports our focus on improving the model's planning capabilities on abstract graphs.

7 Conclusion

We present plan-over-graph, a novel paradigm enhancing parallelism in LLM-based agentic planning. Our approach extracts task dependencies as structured graphs, optimizing parallel planning through graph-aware reasoning. We develop a synthetic dataset annotated with directed acyclic graphs and a two-stage training scheme, achieving significant improvements. Analysis reveals graph structure inversely affects performance and parallel time reduction. This work establishes a framework for parallel agentic systems, bridging the gap between abstract graphs and real-world applications.

Limitations

We acknowledge the limitations of this work. (i) Although we believe and verify that the ability to plan on the graph is more important than extraction, open-source models have also shown certain flaws in extraction. (ii) In reality, the model's plan can be a dynamic process that interacts with the environment, where the model can refine the previously given plan through perception. Our future work will focus on these two directions.

References

Anthropic. 2024. Claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michał Podstawski, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690.

Yukun Cao, Shuo Han, Zengyi Gao, Zezhong Ding, Xike Xie, and S. Kevin Zhou. 2024. Graphinsight: Unlocking insights in large language models for graph structure understanding. *ArXiv*, abs/2409.03258.

Runjin Chen, Tong Zhao, Ajay Kumar Jaiswal, Neil Shah, and Zhangyang Wang. 2024. LLaGA: Large language and graph assistant. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 7809–7823. PMLR.

Xinnan Dai, Haohao Qu, Yifen Shen, Bohang Zhang, Qihao Wen, Wenqi Fan, Dongsheng Li, Jiliang Tang, and Caihua Shan. 2024. How do large language models understand graph patterns? a benchmark for graph pattern comprehension. *ArXiv*, abs/2410.05298.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. 2022. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35:18343–18362.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2024. Talk like a graph: Encoding graphs for large language models. In *International Conference on Learning Representations (ICLR)*.

Gonzalo Gonzalez-Pumariega, Leong Su Yean, Neha Sunkara, and Sanjiban Choudhury. 2025. Robotouille: An asynchronous planning benchmark for LLM agents. In *The Thirteenth International Conference on Learning Representations*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8154–8173.

Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. 2019. An introduction to the planning domain definition language.

Ji He, Jianshu Chen, Xiaodong He, Jianfeng Gao, Lihong Li, Li Deng, and Mari Ostendorf. 2016. Deep reinforcement learning with a natural language action space. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL - Association for Computational Linguistics.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and 1 others. 2024. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14281–14290.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. 2024. Understanding the planning of llm agents: A survey. *CoRR*, abs/2402.02716.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Ziqi Jia, Junjie Li, Xiaoyang Qu, and Jianzong Wang. 2025. Enhancing multi-agent systems via reinforcement learning with llm-based planner and graph-based policy. *Preprint*, arXiv:2503.10049.

Bill Yuchen Lin, Yicheng Fu, Karina Yang, Faeze Brahman, Shiyu Huang, Chandra Bhagavatula, Prithviraj Ammanabrolu, Yejin Choi, and Xiang Ren. 2024a. Swiftsage: A generative agent with fast and slow thinking for complex interactive tasks. *Advances in Neural Information Processing Systems*, 36.

Fangru Lin, Emanuele La Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony G. Cohn, and Janet B. Pierrehumbert. 2024b. Graph-enhanced large language models in asynchronous plan reasoning. In *Forty-first International Conference on Machine Learning*.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. 2023. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*.

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

Xinbei Ma, Zhuosheng Zhang, and Hai Zhao. 2024. CoCo-agent: A comprehensive cognitive MLLM agent for smartphone GUI automation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 9097–9110, Bangkok, Thailand. Association for Computational Linguistics.

Xinyu Ning, Yutong Zhao, Yitong Liu, and Hongwen Yang. 2024. Dgot: Dynamic graph of thoughts for scientific abstract generation. *ArXiv*, abs/2403.17491.

OpenAI. 2023. Gpt-4 technical report. *ArXiv preprint*, abs/2303.08774.

Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. 2024. Mutual reasoning makes smaller llms stronger problem-solvers. *Preprint*, arXiv:2408.06195.

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

Shuofei Qiao, Runnan Fang, Ningyu Zhang, Yuqi Zhu, Xiang Chen, Shumin Deng, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. 2024. Agent planning with world knowledge model. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. 2024. Identifying the risks of LM agents with an LM-emulated sandbox. In *The Twelfth International Conference on Learning Representations*.

Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. 2023. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*.

Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, and 3 others. 2024. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Transformer Circuits Thread*.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. 2023. On the planning abilities of large language models - a critical investigation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. 2024. OS-copilot: Towards generalist computer agents with self-improvement. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

Zhiheng Xi, Yiwen Ding, Wenxiang Chen, Boyang Hong, Honglin Guo, Junzhe Wang, Dingwen Yang, Chenyang Liao, Xin Guo, Wei He, Songyang Gao, Lu Chen, Rui Zheng, Yicheng Zou, Tao Gui, Qi Zhang, Xipeng Qiu, Xuanjing Huang, Zuxuan

Wu, and Yu-Gang Jiang. 2024. Agentgym: Evolving large language model-based agents across diverse environments. *Preprint*, arXiv:2406.04151.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024a. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024c. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Shunyu Yao, Rohan Rao, Matthew Hausknecht, and Karthik Narasimhan. 2020. Keep calm and explore: Language models for action generation in text-based games. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

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

Junwei Yu, Yepeng Ding, and Hiroyuki Sato. 2025. Dyntaskmas: A dynamic task graph-driven framework for asynchronous and parallel llm-based multi-agent systems. *Preprint*, arXiv:2503.07675.

Shaoqing Zhang, Zhuosheng Zhang, Kehai Chen, Xinbei Ma, Muyun Yang, Tiejun Zhao, and Min Zhang. 2024a. Dynamic planning for llm-based graphical user interface automation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 1304–1320.

Zhehao Zhang, Jiaao Chen, and Diyi Yang. 2024b. DARG: Dynamic evaluation of large language models via adaptive reasoning graph. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Zhuosheng Zhang, Yao Yao, Aston Zhang, Xiangru Tang, Xinbei Ma, Zhiwei He, Yiming Wang, Mark Gerstein, Rui Wang, Gongshen Liu, and 1 others. 2025. Igniting language intelligence: The hitchhiker's guide from chain-of-thought reasoning to language agents. *ACM Computing Surveys*, 57(8):1–39.

Zirui Zhao, Wee Sun Lee, and David Hsu. 2024. Large language models as commonsense knowledge for large-scale task planning. *Advances in Neural Information Processing Systems*, 36.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning acting and planning in language models.

Dongsheng Zhu, Weixian Shi, Zhengliang Shi, Zhaochun Ren, Shuaiqiang Wang, Lingyong Yan, and Dawei Yin. 2025. Divide-then-aggregate: An efficient tool learning method via parallel tool invocation. *arXiv preprint arXiv:2501.12432*.

A Prompt Template

We show the prompt templates following examples.

Prompt template for graph planning

You are given a set of transformation rules, where each rule consists of source nodes (materials or subtasks), target nodes (resulting materials or tasks), the time required to complete the transformation, and a cost associated with the transformation. Your goal is to plan a path from the initial nodes to the target node, supporting parallel transformations, to obtain the target node in the shortest time possible, while minimizing the total cost.
Input format:
- Transformation rules: A list of dictionaries, where each dictionary represents a transformation rule and contains:
  - source: A list of source nodes (the prerequisites for the transformation).
  - target: A list of target nodes (the result of the transformation).
  - time: The time required to complete the transformation (an integer).
  - cost: The cost associated with the transformation (an integer).
- Initial nodes: A list of strings representing the available nodes at the start.
- Target node: A string representing the node that needs to be obtained.
Output format:
- Plan: A list of subtasks, where each subtask is a JSON object with the following fields:
  - name: The name of the subtask or node being completed. The default name format is "Subtask" followed by a sequence number.
  - source: A list of source nodes involved in this subtask. The sources must be products you already have or can obtain through previous steps.
  - target: The target node resulting from this subtask. Both the source and target must conform to a given rule and cannot be assumed or self-created.
  - dependencies: A list of dependencies (other subtask names) that need to be completed before this subtask can be executed. This ensures the execution order between subtasks, and the dependencies must provide the required sources for this subtask.
Important:
- The generated JSON must strictly follow the JSON format. The following rules must be strictly adhered to:
  - All keys and values must be enclosed in double quotes.
  - All elements in arrays must be separated by commas.
  - All fields in the JSON must be complete and correctly formatted, with no missing or incorrect elements.
- All planned steps must comply with a given rule.
- All substances involved must conform to the given rules.
Your task is to generate the final plan in the specified JSON format, minimizing both the completion time and total cost. Do not provide any implementation code.
Here is an example to better understand the task:
{graph_planning_example}
Now, based on the following transformation rules, initial nodes, and target node, please provide an optimal plan that allows the target node to be obtained in the shortest time with the minimal total cost, supporting parallel transformations.
Only include necessary steps that are required for the fastest completion with the least cost. Do not add any extra or redundant transformation steps.
Task:
```json
{task}
```
Your task is to generate the final plan in the specified JSON format. Do not provide any implementation code.

Prompt template for query planning with CoT

You are given a set of transformation rules, where each rule consists of source nodes (materials or subtasks), target nodes (resulting materials or tasks), the time required to complete the transformation, and a cost associated with the transformation. Your goal is to plan a path from the initial nodes to the target node, supporting parallel transformations, to obtain the target node in the shortest time possible, while minimizing the total cost.
Input format:
- Transformation rules: A list of dictionaries, where each dictionary represents a transformation rule and contains:
  - source: A list of source nodes (the prerequisites for the transformation).
  - target: A list of target nodes (the result of the transformation).
  - time: The time required to complete the transformation (an integer).
  - cost: The cost associated with the transformation (an integer).
- Initial nodes: A list of strings representing the available nodes at the start.
- Target node: A string representing the node that needs to be obtained.
Output format:
- Plan: A list of subtasks, where each subtask is a JSON object with the following fields:
  - name: The name of the subtask or node being completed. The default name format is "Subtask" followed by a sequence number.
  - source: A list of source nodes involved in this subtask. The sources must be products you already have or can obtain through previous steps.
  - target: The target node resulting from this subtask. Both the source and target must conform to a given rule and cannot be assumed or self-created.
  - dependencies: A list of dependencies (other subtask names) that need to be completed before this subtask can be executed. This ensures the execution order between subtasks, and the dependencies must provide the required sources for this subtask.
Important:
- The generated JSON must strictly follow the JSON format. The following rules must be strictly adhered to:
  - All keys and values must be enclosed in double quotes.
  - All elements in arrays must be separated by commas.
  - All fields in the JSON must be complete and correctly formatted, with no missing or incorrect elements.
- All planned steps must comply with a given rule.
- All substances involved must conform to the given rules.
Your task is to generate the final plan in the specified JSON format, minimizing both the completion time and total cost. Do not provide any implementation code.
Here is an example to better understand the task:
{graph_planning_example}
{CoT_example}
Now, based on the following transformation rules, initial nodes, and target node, please provide an optimal plan that allows the target node to be obtained in the shortest time with the minimal total cost, supporting parallel transformations.
Only include necessary steps that are required for the fastest completion with the least cost. Do not add any extra or redundant transformation steps.
Task:
```json
{task}
```
Your task is to generate the final plan in the specified JSON format. Do not provide any implementation code.

Prompt template for query planning

For the input task, please provide an optimal plan that allows the target to be obtained. Minimize the cost under the premise of the shortest time.
Projects without dependencies can be completed in parallel to improve overall efficiency.
Please provide the final solution in JSON format:
- Plan: A list of subtasks, where each subtask is a JSON object with the following fields:
  - name: The name of the subtask or node being completed. The default name format is "Subtask" followed by a sequence number.
  - source: A list of source nodes involved in this subtask. The sources must be products you already have or can obtain through previous steps.
  - target: The target node resulting from this subtask. Both the source and target must conform to a given rule and cannot be assumed or self-created.
  - dependencies: A list of dependencies (other subtask names) that need to be completed before this subtask can be executed. This ensures the execution order between subtasks, and the dependencies must provide the required sources for this subtask.

Here is an example:
Input:
{query_example}
Output:
```json
{query_example_plan}
```
Input:
{task}
Output:

Prompt template for query planning with CoT

For the input task, please provide an optimal plan that allows the target to be obtained. Minimize the cost under the premise of the shortest time.
Projects without dependencies can be completed in parallel to improve overall efficiency.
Please provide the final solution in JSON format:
- Plan: A list of subtasks, where each subtask is a JSON object with the following fields:
  - name: The name of the subtask or node being completed. The default name format is "Subtask" followed by a sequence number.
  - source: A list of source nodes involved in this subtask. The sources must be products you already have or can obtain through previous steps.
  - target: The target node resulting from this subtask. Both the source and target must conform to a given rule and cannot be assumed or self-created.
  - dependencies: A list of dependencies (other subtask names) that need to be completed before this subtask can be executed. This ensures the execution order between subtasks, and the dependencies must provide the required sources for this subtask.

Let's think step by step and then output the final solution in JSON format.
Here is an example:
Input:
{query_example}
Output:
```json
{query_example_plan}
```
{CoT_example}
Input:
{task}
Output:

Prompt template for query planning with PLaG

Assume that that infinite resources are available, which means projects without dependencies can be completed in parallel to improve overall efficiency. For the input task, please provide an optimal plan that allows the target to be obtained. Minimize the cost under the premise of the shortest time. Let's construct a graph with the nodes and edges first to represent step ordering constraints. Use the graph and dictionary to calculate the shortest possible time needed for the task. Let's think step by step and then output the final solution in JSON format.
Please provide the final solution in JSON format:
A list of subtasks, where each subtask is a JSON object with the following fields:
- name: The name of the subtask or node being completed. The default name format is "Subtask" followed by a sequence number.
- source: A list of source nodes involved in this subtask. The sources must be products you already have or can obtain through previous steps.
- target: The target node resulting from this subtask. Both the source and target must conform to a given rule and cannot be assumed or self-created.
- dependencies: A list of dependencies (other subtask names) that need to be completed before this subtask can be executed. This ensures the execution order between subtasks, and the dependencies must provide the required sources for this subtask.

Let's think step by step and then output the final solution in JSON format.
Here is an example:
Input:
{query_example}
Output:
Here is a graph representation of the task:
```json
{query_example_plan}
```
{CoT_example}
```json
{query_example_plan}
```
Input:
{task}
Output:

Prompt template for extracting graph from query

Task: Extract structured transition rules from unstructured workflow narratives. Objective: Identify all transitions between nodes in the text. For each transition, extract:
- Source nodes (prerequisites)
- Target nodes (outcomes)
- Time (duration)
- Cost (numeric resource units)

Additionally, determine the initial_source (starting node) and target (final node).
Input: A story describing a workflow process. Example phrases may include:
- "From [NodeA], proceed to [NodeB] in X days at a cost of Y units"
- "When both [NodeA] and [NodeB] are ready, [NodeC] can be completed in X days at a cost of Y units"
- Shortcuts like "directly from [NodeA] to [NodeC] in X days at a cost of Y units"

Output:
A JSON object with:
1. "rules": A list of transition rules, each containing:
- "id" (sequential integer starting from 0)
  - "source" (list of node IDs, e.g., ["N1"])
  - "target" (list of node IDs, e.g., ["N2"])
  - "time" (numeric value)
  - "cost" (numeric value)

2. "initial_source": List of starting node IDs (e.g., ["N1"])
3. "target": Final node ID (e.g., "N8")

Your Task: Convert the following story into the JSON format above. Ensure:
1. All transitions are captured, including multi-source dependencies and shortcuts
2. Node IDs (e.g., N1, N2) are preserved exactly as written
3. Time and cost values are strictly numeric
4. Follow the JSON schema precisely

Example Input Story:
{query_example}
Example Output:
```json
{query_example_plan}
```
Input Story:
{task}
Output:

Prompt template for generating query from graph

Transform this abstract task into a specific task in a real-world scenario, noting the following:
1. Tasks without dependencies can be executed in parallel.
2. Please express the instructions in complete natural language without explicitly listing the rules.
3. As long as there is one path that reaches the final goal, it is considered successful.
4. The source of a rule must be fully achieved before proceeding with the rule and obtaining its target.
5. You must strictly follow the rules I have given, make sure the rules in your story correspond one-to-one with the rules I have provided, and the sum of rules in your story must be equal to the sum of rules in the task.
6. You must explicitly mention both the time and cost associated with each rule in the story.
7. You only need to write the rules as a story, without offering any additional evaluation comments or introductory remarks.
Here is an example from another task for reference:
Input: “‘json
{query_example_plan}
“‘
OutPut:
{query_example}
Input:
“‘json
{task}
“‘
Output:

Examples

graph_planning_example:
Task:
“‘json
{ "rules": [{ "source": ["N1"], "target": ["N2"], "time": 3, "cost": 1 }, { "source": ["N6"], "target": ["N3"], "time": 4, "cost": 1 }, { "source": ["N2", "N3"], "target": ["N4"], "time": 2, "cost": 1 }, { "source": ["N4"], "target": ["N5"], "time": 1, "cost": 1 }, { "source": ["N2"], "target": ["N5"], "time": 5, cost": 1 }], "initial_source": ["N1", "N6"], "target": "N5" }
“‘
Expected output:
“‘json
[{ "name": "Subtask1", "source": ["N1"], "target": ["N2"], "dependencies": [] }, { "name": "Subtask2", "source": ["N6"], "target": ["N3"], "dependencies": [] }, { "name": "Subtask3", "source": ["N2", "N3"], "target": ["N4"], "dependencies": ["Subtask1", "Subtask2"] }, { "name": "Subtask4", "source": ["N4"], "target": ["N5"], "dependencies": ["Subtask3"] }]

query_example:
In a busy urban construction project, multiple sites must be coordinated to build the "Core Area(N9)" as quickly and cost-effectively as possible. The project begins at three sites: "Infrastructure(N1)," "Elevated(N3)," and "Residential(N7)," each with different tasks. The "Infrastructure Area(N1)" takes 3 days and costs 1 to proceed to the "Bridge Area(N2)", while the "Elevated Area(N3)" moves to the "Building Area(N4)" in 3 days and at a cost of 1. The "Bridge Area(N2)" connects to the "Road Area(N5)" in 4 days and costs 1, and can directly connect to the "Facilities Area(N6)" in 8 days at a cost of 1. The "Building Area(N4)" partners with the "Road Area(N5)" to build the "Facilities Area(N6)" in 2 days and at a cost of 1. The "Residential Area(N7)" takes 5 days and costs 1 to reach the "City Center Area(N8)", while the "Building Area(N4)" directly reaches it in 1 day and costs 1. Once the "Facilities(N6)" and "City Center(N8)" areas are ready, they combine to complete the "Core Area(N9)" in 2 days at a cost of 1. The "Infrastructure Area(N1)" has a shortcut to bypass other areas and reach the "Core Area(N9)" in 15 days at a cost of 1. The project team can select the most efficient route based on resources and progress.

query_example_plan:
{"rules": [{ ’id’: 0, "source": ["N1"], "target": ["N2"], "time": 3, "cost": 1 }, { ’id’: 1, "source": ["N3"], "target": ["N4"], "time": 3, "cost": 1 }, { ’id’: 2, "source": ["N2"], "target": ["N5"], "time": 4, "cost": 1 }, { ’id’: 3, "source": ["N4", "N5"], "target": ["N6"], "time": 2, "cost": 1 }, { ’id’: 4, "source": ["N2"], "target": ["N6"], "time": 8, "cost": 1 }, { ’id’: 5, "source": ["N7"], "target": ["N8"], "time": 5, "cost": 1 }, { ’id’: 6, "source": ["N4"], "target": ["N8"], "time": 1, "cost": 1 }, { ’id’: 7, "source": ["N6", "N8"], "target": ["N9"], "time": 2, "cost": 1 }, { ’id’: 8, "source": ["N1"], "target": ["N9"], "time": 15, "cost": 1 },], "initial_source": ["N1", "N3", "N7"], "target": "N9"}

Examples

CoT_example:
Think Step by Step:
To determine the optimal plan, we analyze dependencies and parallelization opportunities: 1. **Initial Parallelization**:
- (N3→N4, 3 days) and (N1→N2, 3 days) can start immediately and run in parallel since they have no dependencies.
- (N7→N8, 5 days) is also independent and starts concurrently.
2. **Sequential Dependencies**:
- (N2→N5, 4 days) requires (N1→N2, 3 days) to finish first. Thus, it starts after 3 days and completes at **7 days**.
- (N4+N5→N6, 2 days) depends on (N3→N4, 3 days) and (N2→N5, 7 days). It begins at **7 days** and ends at **9 days**.
- (N4→N8, 1 day) depends on (N3→N4, 3 days). It starts at **3 days** and finishes at **4 days**.
3. **Critical Path Analysis**:
- The path (N1→N2, 3 days) → (N2→N5, 4 days) → (N4+N5→N6, 2 days) takes **3 + 4 + 2 = 9 days**.
- The path (N3→N4, 3 days) → (N4→N8, 1 days) takes **3 + 1 = 4 days**.
- (N6+N8→N9, 2 days) requires both (N4+N5→N6) (9 days) and (N4→N8) (4 days). It starts at **9 days** and ends at **11 days**.
4. **Avoiding the Shortcut**:
- The alternative shortcut (N1→N9, 15 days) is slower than the critical path (11 days), so it's excluded.
Conclusion:
The longest path (critical path) determines the total time: (N1→N2, 3 days) → (N2→N5, 4 days) → (N4+N5→N6, 2 days) → (N6+N8→N9, 2 days) takes (3 + 4 + 2 + 2 = **11 days**). Parallel execution of independent tasks minimizes cost while achieving the shortest time.
‘‘‘

B Dataset Details

The statistics of our synthetic data are shown in Table 5.

C Penalty Value Selection

The penalty of 4 is chosen empirically. We first computed the average time and cost ratio among successful plans and found that the maximum is around 3 and does not exceed 4. Without penalizing failures, models with weak capabilities would appear deceptively competitive: although they have low success rates overall, they succeed on simpler cases where the ratio is naturally low, resulting in an unintuitive final metric. A penalty too large would cause failed cases to dominate and distort comparisons. A value of 4 therefore balances clarity and numerical intuitiveness.

D Textual Query Statistics

Figure 4 shows the statistics on our synthetic query data.

Set	Nodes	Graph Structure	Edge Relation	Samples
Training	10	Random	Uniform	2000
		Tree-based	Linear	2000
	30	Random	Uniform	2000
		Tree-based	Linear	2000
	50	Random	Linear	2000
		Tree-based	Linear	2000
Testing	10	Random	Linear	100
		Tree-based	Linear	100
		Random	Uniform	1000
	20	Random	Linear	100
		Tree-based	Linear	100
	30	Random	Linear	100
		Tree-based	Linear	100
		Random	Uniform	1000
	40	Random	Linear	100
		Tree-based	Linear	100
	50	Random	Linear	100
		Tree-based	Linear	100

Table 5: Detailed statistics of our training and testing dataset.

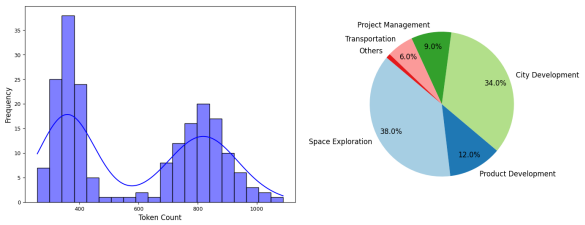

Figure 4: Statistics on our synthetic query. The bar chart on the left displays the distribution of tokens. The pie chart on the right shows the topic distribution.

E Trainning Setups

Training is performed using the LLaMa Factory (Zheng et al., 2024) framework.

In SFT data without mixup, for a graph G, the inputs and outputs are obtained as $(x, y) = (G, p_{\text{opt}})$; with mixup, for G, the inputs and outputs include both $(x, y) = (G, p_{\text{opt}})$ and (G, p_{second}). For DPO data, the input x, chosen output y_w and rejected output y_l are derived as $(G, p_{\text{opt}}, p_{\text{second}})$.

The training loss function of SFT is defined as follows:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{SFT}}} \sum_{t=1}^{|y|} \log P_\theta(y_t|x, y_{<t}). \tag{9}$$

The training loss function of DPO is defined as follows:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}_{\text{DPO}}} \log\sigma\Bigg(\beta\Bigg(\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\Bigg)\Bigg). \quad (10)$$

The detailed hyperparameter settings are outlined in Table 6.

Name	Value
cutoff len	8,192
epochs	10
batch size per device	1
gradient accumulation steps	4
learning rate	1e-6
lr scheduler type	cosine
warmup ratio	0.1
bf16	true

Table 6: Detailed training hyperparameters.

All experiments were conducted on 4 NVIDIA A100 GPUs.

F DPO-Only Training

We describe the details for directly using DPO to train the model. In addition to the original DPO dataset, we collected 1,000 examples of erroneous model plans as the rejected output, paired with their corresponding optimal solutions as the chosen output. We evaluated this model on the same testing set described in Section 5.1, with results in Table 2.

We observed that the DPO-only model performed substantially worse than both the pretrained base model and the two-stage SFT + DPO model. We attribute this drop in performance to the lack of a supervised fine-tuning phase: without SFT, the model fails to learn basic plan generation skills, struggles with JSON format and plan correctness, and DPO alone only teaches ranking between optimal and non-optimal outputs rather than imparting fundamentals of valid plan construction. Improving this setup would require collecting a richer, more diverse set of negative examples covering multiple error dimensions, e.g., formatting errors, logical consistency violations, an effort that may not surpass the simplicity and effectiveness of including an SFT stage.

G Edge variation Status

Figure 5 shows how Claude (top) and our trained Llama (bottom) perform across varying edge counts on 10-node (left) and 30-node (right) graphs, with colors indicating failure (red), feasible (green), and optimal (orange) solutions.

As edge density increases, optimal solutions decline for both models. On 10-node graphs, both perform well overall, though Llama already shows occasional failures. On 30-node graphs, the two models respond quite differently: Claude shifts from optimal to feasible as the dominant outcome and begins to show some failures, while Llama's failures become the majority. These results confirm edge count as a reliable indicator of task difficulty, and reveal that the two models degrade in meaningfully different ways as graphs grow more complex.

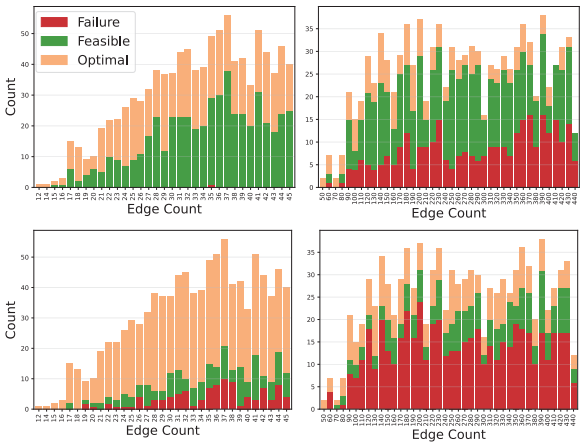

Figure 5: Claude and our trained Llama performance across different edge counts. The vertical axis represents the corresponding number of cases of each state(fail, feasible, optimal). The horizontal axis represents the number of edges segmented at certain intervals.

H Query Example

A query and corresponding plan in a real-life scenario are shown in A.

I Supplementary of the Experiments

Tables 7 and 8 show the absolute value of correlation coefficients and slopes of four normalized metrics with changes in the number of nodes and edges.

Metrics are scaled to the [0,1] range using min-max normalization:

$$y_{\text{norm}} = \frac{y - y_{\min}}{y_{\max} - y_{\min}} \quad (11)$$

where y represents raw values of: node and edge counts, success counts, optimal counts, average time ratios, and average cost ratios.

Model	Success Rate		Optimal Rate		Time Ratio		Cost Ratio	
	r	*m*	*r*	*m*	*r*	*m*	*r*	*m*
Claude 3.5 Sonnet	-0.96	-0.99	-0.82	-0.82	0.92	1.02	0.94	0.99
GPT-4o	-0.99	-1.02	-0.94	-0.97	0.99	1.09	0.98	1.1
Llama-3.1-8B-Instruct	-0.95	-0.96	-0.95	-1.01	0.79	0.92	0.93	0.97
Llama-3.1-8B-Instruct-Trained	-0.94	-1.04	-0.89	-0.86	0.94	1.04	0.93	1.0
Qwen2.5-7B-Instruct	-0.96	-0.98	-0.96	-0.95	0.96	0.97	0.66	0.66
Qwen2.5-7B-Instruct-Trained	-0.99	-0.98	-0.80	-0.82	0.81	0.76	0.96	0.98

Table 7: Correlation coefficients (r) and slopes (m) between metrics and node counts.

Model	Node Count	Success Rate		Optimal Rate		Time Ratio		Cost Ratio	
		r	*m*	*r*	*m*	*r*	*m*	*r*	*m*
Claude 3.5 Sonnet	10	0.27	0.15	-0.44	-0.28	0.50	0.26	0.43	0.23
	30	-0.74	-0.56	-0.71	-0.55	0.81	0.59	0.78	0.59
Llama-3.1-8B-Instruct-Trained	10	0.05	0.03	-0.28	-0.17	0.58	0.34	0.45	0.24
	30	-0.39	-0.29	-0.62	-0.37	0.45	0.34	0.60	0.43

Table 8: Correlations (r) and slopes (m) between edge variations and metrics with node counts.

Pearson correlation coefficients (r) between node and edge counts (X) and metrics (Y) are calculated as:

$$r_{XY} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}. \tag{12}$$

The slope (m) of the best-fit line is computed using linear regression:

$$m = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}. \tag{13}$$

All results are rounded to 2 decimal places for final reporting.

For a given plan P composed of sub plans p, the time ratio of parallel execution to sequential execution can be calculated as:

$$\text{Ratio} = \frac{\Omega_{\text{time}}(P)}{\sum_{p \in P} \tau_p}. \tag{14}$$

J Evaluation on Robotouille

We evaluate our method on the Robotouille benchmark (Gonzalez-Pumariega et al., 2025) using GPT-4o, consistent with the original benchmark setup. Our method generates a subtask graph and produces a complete plan in a single pass, with up to two retries upon failure.

As shown in Table 9, our method substantially outperforms I/O CoT (Sync: 36% vs. 14%, Async: 9% vs. 1%), which is the most comparable baseline as both generate a complete plan in a single pass. While ReAct achieves 47% on synchronous tasks through full multi-turn interaction, this comes at significantly higher inference cost. By introducing a lightweight retry mechanism, our method reaches 45% on synchronous tasks and 17% on asynchronous tasks, achieving comparable synchronous performance to ReAct while surpassing it on asynchronous tasks.

Method	Sync (%)	Async (%)
I/O	4	1
I/O CoT	14	1
ReAct	47	11
Ours	36	9
Ours + retry	45	17

Table 9: Success rate on the Robotouille benchmark. All methods use GPT-4o. Baseline results are from the original benchmark.

We use the following prompts for planning and refinement.

Prompt template for Robotouille planning - Part 1

You are an agent exploring an environment with a goal to achieve. You will propose a plan in the current state to reach the goal.
Below is a description of the environment:
You are a robot in a kitchen environment. The objects in the kitchen and your goal are described in the Observation.
The various types of objects in the kitchen include
- Station: A location in the kitchen where you can perform special actions, e.g. cooking or cutting
- Item: An object that can be picked up and potentially used in a Station
- Player: Robots, including you, that are present in the kitchen
- Container: An object that can hold other objects, e.g. a pot or a pan
- Meal: A mixture of ingredients contained within a Container
The rules of the environment are as follows:
- A Player can only operate on Stations where they are located
- A Player can only hold a single Item at a time
- An Item must be placed on a Station to perform an action on it
- A Station can only contain a single Item or stacked Items, but cannot contain multiple unstacked Items at a time
- Items can only be stacked on top of one another, not side by side
- Stacked Items should be treated as multiple separate Items, Player cannot pick up/place multiple stacked Items as a single unit
- A Container can hold multiple Items
- A Meal can be transferred between Containers

Prompt template for Robotouille planning - Part 2

The goal of this environment is to satisfy a human's request, such as 'make me a hamburger'. These goals are intentionally underspecified so common sense reasoning is required to complete them. Specifically, it is important to consider
- the minimal ingredients required to satisfy the request
- any preparation steps for the ingredients like cooking, cutting, etc.
When the goal is achieved or a time limit is reached, the environment will end.
Follow this recipe guide to learn how to make food, attention to ingredients of each recipe:
Sandwich: Stack the **needed prepared ingredients** on a **bread**, then stack another **bread** on it.
Hamburger: A **bottom bun**, stacked on **prepared ingredients**, stacked on a **top bun**.
Soup: A pot of boiling water containing prepared ingredients served in a bowl.
You must strictly follow the ingredients needed to use and the stacking order in the recipe.
Important: Always check ingredient states in the observation (e.g., "can be cut", "is cooked"). Process ingredients as needed - cut items that "can be cut" and cook items that need cooking before using them in recipes.

Prompt template for Robotouille planning - Part 3

You will receive the initial state and the goal as follows:
Observation: ...
Valid Actions: ...
where
- 'Observation' contains state information about objects in the environment and the goal.
- 'Valid Actions' is the list of actions you can take in the current state.
You must first decompose the whole task into a JSON-formatted subtasks, then propose a plan based on the subtasks.
Notice the following when decomposing the task:
- Reasoning about the task and the environment, and how to achieve the goal.
  - If there is an error feedback, you must first analyze the error and reason about how to avoid it in the future.
  - Considering all the rules and constraints, and the current state of the environment.
  - Always follow the rules of the environment when decomposing the task.
  - Considering which ingredients are exactly needed to use referring to the recipe, or to move according to the environment.
  - Break subtasks down into finer-grained steps:
    - If an ingredient needs to be moved then processed, these should be two separate subtasks.
    - If an ingredient needs to be moved then stacked, it can directly be stacked.
  - About positions:
    - Notice the position of the goal. Since stacked Items cannot be moved as a single unit, you must decide which item to place at the bottom, and whether it is in a feasible position.
    - Consider the position of each ingredient, and avoid moving items that are already in a feasible position.
    - If Item i1's target location s1 is already occupied by another Item i2:
      - If i1 should not be stacked on i2, you must first move i2 to another position before moving i1 to s1.
      - If i1 should be stacked on i2, you can directly stack i1 on i2 without moving i2.
    - Place useless ingredients to some unrelevant positions if necessary, and do not let them effect the goal.
- Represent ingredients and subtasks in JSON format:
  - "initial_sources": all the ingredients that are available at the start.
  - "goal": the final goal you aim to achieve.
  - "subtasks": define how to transform one or some ingredients into another, or move to another position.
    - "sources": the list of ingredients that can be used to create a new ingredient.
    - "target": the ingredient that is created from the sources by the transformation.
    - "time": the time it takes to perform the transformation.
    - "cost": the cost of performing the transformation, always 1.

Prompt template for Robotouille planning - Part 4

- About the names:
  - Each initial or created ingredient should be named uniquely, and annotated with its position, such as "bread1_on_table1".
  - The position suffices mean the position and the moving of the ingredients.
  - Stacked ingredients are named by concatenating the names of the ingredients, such as "onion1_on_bread1_on_table1" for onion1 stacked on bread1.
  - Processed ingredients should be named with the action performed, such as "onion1_cut_on_board1" for sliced onion1.

Finally, based on the JSON-subtasks, output the plan:
- 'Plan' is the sequence of actions you propose to take in the environment to reach the goal
  - The actions should be formatted exactly as they are in the environment description
  - Do not include any numbering or bullet points for the actions
- Always follow the rules of the environment when proposing the plan.
- Only choose subtasks that leads to the target. Do not include any subtasks that are useless.
- Firstly minimize the time of the plan, then minimize the cost.
- Avoid unnecessary actions, such as moving items around without a purpose, or moving items which have been in a correct position.
- Pay attention to the stacking order of ingredients according to the target, clarify which should be picked up in order.

Always format your response as follows:
Reasoning: ...
JSON-subtasks:
"'json { "initial_sources": [...], "goal": ..., "subtasks": [...] } "'
Plan: ...
The actions you can take in the environment are as follows:
- Move {p1} from {s1} to {s2}
- Pick up {i1/c1} from {s1} using {p1}
- Place {i1/c1} on {s1} using {p1}
- Stack {i1} on top of {i2} using {p1}
- Unstack {i1} from {i2} using {p1}
- Cook {i1} on {s1} using {p1}
- Cut {i1} on {s1} using {p1}
- Fry {i1} on {s1} using {p1}
- Fill {c1} with water from {s1} using {p1}
- Boil {c1}'s contents on {s1} using {p1}
- Add {i1} into {c1} using {p1}
- Fill {c1} with {c2}'s contents using {p1}
- Do nothing

Observation: {observation}
Valid Actions: {valid_actions}

Prompt template for Robotouille refinement

Your previous plan was not correct. Please refine your plan step by step based on the following feedback:
Error Feedback: {feedback}
Please strictly follow the steps below:
1. **Error Analysis**: - Analyze the error feedback and summarize the reason for the error. - Clearly state which rule, position, or sequence was violated.
2. **Subtasks Check**: - Check if the decomposition of subtasks (JSON-subtasks) is correct and complete. - If there are errors or omissions in the subtasks, re-decompose and output a new JSON-subtasks.
3. **Action Plan Check**: - If the subtasks are correct, check if the specific actions in the plan are legal and feasible. - If there are illegal actions, refine the plan by modifying only the problematic actions.
4. **Reasoning about refinement**: - Provide reasoning for the refinement, explaining how the new plan addresses the error feedback. - Ensure that the new plan adheres to the rules of the environment and the constraints of the task, avoid other errors.

Please output your reasoning and corrections in the following format:
Error Analysis: ... Reasoning: ... JSON-subtasks: "'json {{ "initial_sources": [...], "goal": ..., "subtasks": [...] }} "' Plan: ...
Do not include any additional explanations or symbols. Only output the above content.