Programming over Thinking: Efficient and Robust Multi-Constraint Planning

Anonymous ACL submission

Abstract

Multi-constraint planning involves identifying, evaluating, and refining candidate plans while satisfying multiple, potentially conflicting constraints. Existing large language model (LLM) approaches face fundamental limitations in this domain. Pure reasoning paradigms, which rely on long natural language chains, are prone to inconsistency, error accumulation, and prohibitive cost as constraints compound. Conversely, LLMs combined with coding- or solver-based strategies lack flexibility: they often generate problem-specific code from scratch or depend on fixed solvers, failing to capture generalizable logic across diverse problems. To address these challenges, we introduce the **S**calable **CO**de **P**lanning **E**ngine (SCOPE), a framework that disentangles query-specific reasoning from generic code execution. By separating reasoning from execution, SCOPE produces solver functions that are consistent, deterministic, and reusable across queries while requiring only minimal changes to input parameters. SCOPE achieves state-of-the-art performance while lowering cost and latency. For example, with GPT-4o, it reaches 93.1% success on TravelPlanner, a 61.6% gain over the best baseline (CoT) while cutting inference cost by 1.4x and time by 4.67x. Code is available at https://anonymous.4open.science/r/SCOPE-F551.

1 Introduction

Planning is the process by which an agent organizes sequences of decisions or actions to achieve a goal. With the rapid advancement of LLMs, there has been growing interest in applying them to automate planning tasks (Wei et al., 2025). Unlike traditional problem-solving tasks such as question answering or math reasoning, planning demands exploration of a rapidly expanding solution space, where small errors accumulate and lead to inconsistent or invalid outcomes. Recent development of LLMs in text-based reasoning (Wei et al., 2022; Yao et al., 2023a; Yuan et al., 2025; Gui et al., 2025), powered by strong reasoning capabilities, has shown promise on simpler planning tasks. However, due to LLMs output being inherently probabilistic, these methods lack robustness, where small variations of reasoning paths can produce inconsistent or invalid solutions, and models often lose track of constraints or accumulate errors over long reasoning chains. In addition, they face scalability issues. Reasoning chains grow exponentially with task difficulty, require large numbers of tokens, and become increasingly costly to execute (refer to Figure 4).

To address these issues, prior work has explored integrating external solvers into the reasoning process (Chen et al., 2023; Jiang et al., 2024), leveraging their reliability and determinism to enforce constraints and verify solutions. Subsequent efforts have aimed to handle diverse constraints in planning (Hao et al., 2025b,a; Wang et al., 2025). Existing methods either perform similar reasoning in natural language or generate similar solver code for each query. As a result, they do not support reusable execution abstractions and incur high computational cost. Nevertheless, existing code-based methods lack a systematic mechanism to capture the underlying logic of multi-constraint planning, resulting in inefficiency and increased error.

To this end, we propose the **S**calable **CO**de **P**lanning **E**ngine (SCOPE), a multi-agent framework that employs a two-stage reasoning process to convert natural language queries into optimized structured representations, which are then processed by reusable solver functions. In the problem reasoning phase, SCOPE utilizes **Problem Formalization** and **Problem Optimization** to translate queries into structured representations consisting combination and constraint parameters. Combination parameters specify elements and properties

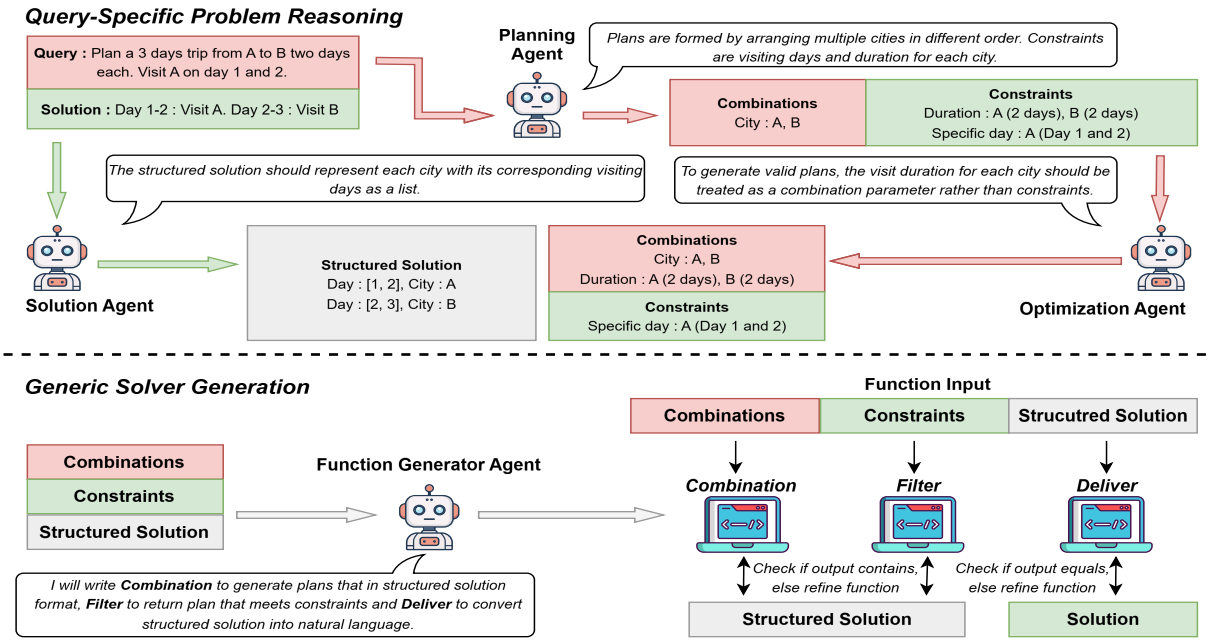

Figure 1: The overview workflow of **SCOPE**. The workflow consists of 2 stages: **Query-Specific Problem Reasoning** (top-half) and **Generic Solver Generation** (bottom-half). Different colors of arrows represent different workflow construction: (red, structured representations), (green, structured solution), (gray, solver functions).

required to generate candidate plans, whereas constraint parameters define conditions these plans must satisfy. LLM agents use these representations to generate general solver functions. Subsequent queries within the same domain are similarly converted into structured representations, enabling subsequent queries in the same domain to be solved efficiently without regenerating code.

To enhance reliability and eliminate dependence on labor-intensive expert prompt design, SCOPE incorporates a parameter-free refinement step in which a single example query and answer are used for output reflection to automatically adjust prompts. This design establishes a fully automated pipeline wherein LLM agents autonomously generate optimized representations, construct reusable solvers, and deliver consistent, scalable solutions to new problems without additional training or manual intervention. Unlike ad-hoc reasoning or problem-specific coding, this approach ensures both flexibility and robustness, as reasoning is tailored to individual queries, while solver functions remain stable across queries in the same domain.

We evaluate SCOPE on benchmarks such as TravelPlanner (Xie et al., 2024) and Natural Plan (Zheng et al., 2024), and reach state-of-the-art results across 5 recent models with much lower inference cost. Our contributions are threefold:

- We propose a reusable abstraction that separates query-specific reasoning from execution logic. It captures the key parameters that generate or filter candidate plans, enabling solver functions that can handle different queries with minimal adaptation.
- We implement an autonomous multi-agent pipeline that induces this abstraction from a single example. The pipeline identifies relevant combinations and constraints and produces solver functions that operate on parameter values rather than query-specific content, allowing the abstraction to be applied broadly.
- We provide empirical evidence that this approach improves robustness, reduces computational cost, and scales efficiently across benchmarks and models.

2 Related Work

Recent advances in LLMs have sparked growing interest in their reasoning and planning capabilities (Achiam et al., 2023; Dong et al., 2024). A prominent line of work studies text-based reasoning through prompting strategies to produce explicit, step-by-step intermediate reasoning (Wei et al., 2022; Shinn et al., 2023; Yao et al., 2023b; Madaan et al., 2023; Chen and Li, 2024; Lee et al., 2025) in natural language. Another related direction investigates multi-agent planning (Chen et al., 2024b,a; Wang et al., 2024; Zhang et al., 2025; Gui et al., 2025). These work allow planning to be more manageable by dividing the tasks into components

managed by expert agents.

Text-based reasoning relies on implicit LLM reasoning and does not explicitly enumerate the solution space, making it prone to errors in multi-constraint planning (Kambhampati et al., 2024). Benchmarks (Valmeekam et al., 2023; Zheng et al., 2024; Xie et al., 2024) show failures under long-horizon reasoning and complex constraints. Multi-agent approaches also require extensive prompt engineering and struggle to systematically decompose such constraints (Han et al., 2024).

An alternative direction leverages code generation as proxy for reasoning (Chen et al., 2023; Yang et al., 2025; Liu et al., 2024a; Yang et al., 2023). These works proposes formalizing problem and replacing reasoning with code to minimize errors. Some recent works have attempted to use code or Planning Domain Definition Language (PDDL) to help reasoning of LLM on constraint satisfaction problems (Guan et al., 2023; Liu et al., 2024b; Wang et al., 2025; Hao et al., 2025a,b). While these methods improves performance, they rely heavily on manual prompting and lack generalization when applied to planning tasks in new domains.

3 Methodology

3.1 Overview

Multi-constraint planning requires exhaustive reasoning over a large candidate space, where missing a single combination can lead to incorrect results. Text-based reasoning methods often fail to systematically enumerate all candidates and miss valid plans, while existing solver-based approaches ensure completeness but incur high inference cost by generating solver code per query. In contrast, our framework leverages LLMs to construct reusable solver functions.

We propose a fully autonomous solver generation and refinement framework mechanism that enables the system to independently discover effective instructions without manual intervention. The framework consists of two high-level stages, namely **Query-Specific Problem Reasoning** and **Generic Solver Generation**. As shown in Figure 1, the Query-Specific Problem Reasoning stage aims to generate a formal representation of the problem domain, including combinations, constraints and structured solution, enabling the system to autonomously design the solver. Given the formal representation, the Generic Solver Generation stage aims to construct the functions required to solve the problem domain, namely *Combination Function*, *Filter Function* and *Deliver Function*.

3.2 Query-Specific Problem Reasoning

In real-world scenarios, explicit reasoning or structured solver logic is not provided to the model, so the model must learn to infer this from examples in order to produce reusable functions for new queries. This stage utilizes a single example query to extract the key parameters and reasoning needed to construct solver functions for the given problem domain. This stage comprises two sequential steps, *Problem Formalization* and *Problem Optimization*.

Problem Formalization. As the first step, Problem Formalization is designed to translate example queries and answers into structured JSON representations that define the input and output interfaces for solver function construction in a later stage. It employs two LLM agents, namely **Planning Agent** and **Solution Agent**. Particularly, Planning Agent converts the example query into structured representation with two keys, `combinations` and `constraints`. `Combinations` contains the parameters for complete candidate generation space, ensuring exhaustive enumeration rather than heuristic exploration. `Constraints` contains parameters for filtering plans, isolating validation logic from generation logic to enable deterministic filtering. Considering the example query as shown in Figure 2, Planning Agent identifies that candidate plans can be generated by arranging the different cities (Berlin, Venice and Paris) in different visiting orders, and therefore proposes combination parameters consisting of "cities", "city_stays", etc. Similarly, a plan is only valid under certain constraints, hence Planning Agent proposes constraint parameters such as "specific_days".

Simultaneously, the **Solution Agent** receives the example answer and converts it into a structured representation, which defines the expected output format used in solver function construction in the next stage. Considering the Figure 2 example, the structured representation would be a list of JSON, which the key "city" and "days" outlines the name of city and visiting days. Together, the Planning Agent and Solution Agent determine the input and output formats of all solver functions.

Problem Optimization. Without optimization, structured representations often contain redundant, entangled, or underspecified parameters that can prevent correct solver function synthesis. SCOPE

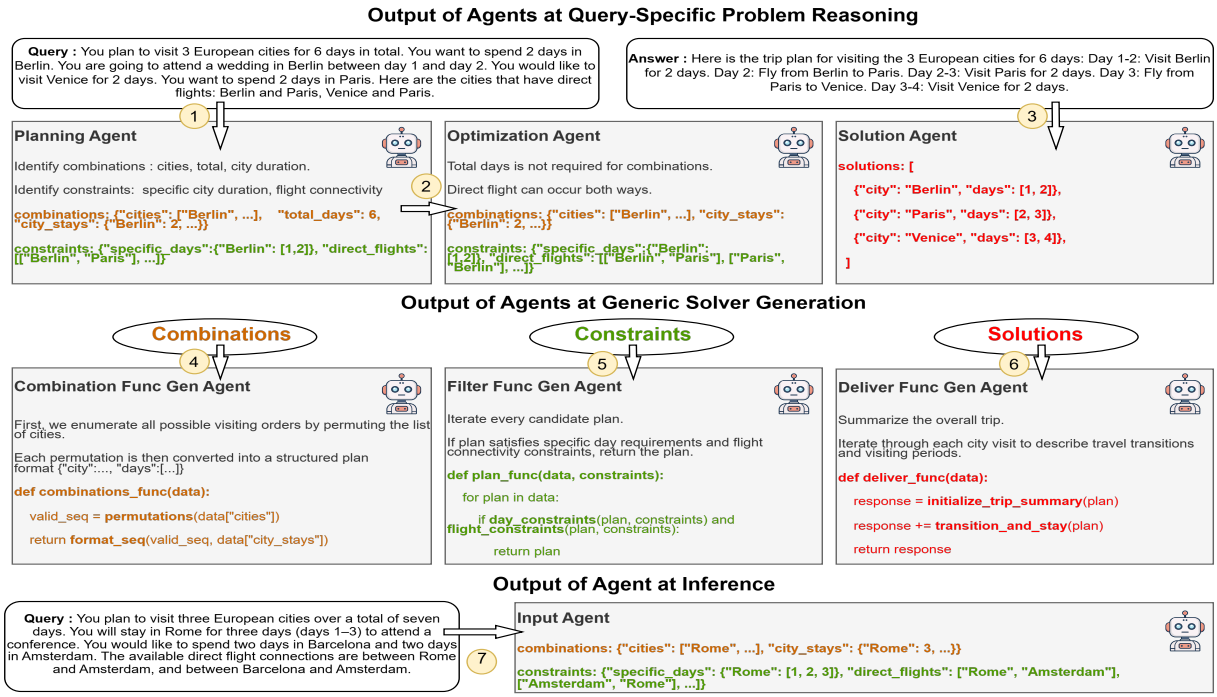

Figure 2: The output of agents at two different stages of **SCOPE** and inference.

employs one or more **Optimization Agents**, each assigned a distinct role via its prompt, to iteratively refine the parameters. These agents remove redundancy, resolve ambiguities, and ensure completeness, guaranteeing that combinations fully cover the candidate plan space while constraints remain clear, consistent, and non-conflicting. The prompts provided to each Optimization Agent are described in Appendix G.

3.3 Generic Solver Generation

Constructing reusable solver functions is challenging because the reasoning must be abstracted from any specific query. To address this, SCOPE decomposes the solver into distinct functions, each responsible for a single task. This decomposition allows the system to focus supervision on ensuring correctness for each function. This stage proceeds through two sequential phases, *Solver Construction* and *Solver Refinement*. In particular, we generate the following solver functions in this stage:

- **Combination Function**: takes as input `data`, which is the values of the `combinations` parameters, and generates candidate plans in structured JSON form.
- **Filter Function**: takes as input `data`, the candidate plans generated by Combination Function, and `constraints`, which is the values of the `constraints` parameters, and returns the plan that satisfy all constraints.
- **Deliver Function**: takes as input `data`, the plan returned by Filter Function and converts it into a natural language answer that follows the desired output format.

Solver Construction. We employ three agents, **Combination Function Generator Agent**, **Filter Function Generator Agent** and **Deliver Function Generator Agent** to generate Combination, Filter and Deliver Function respectively. The Combination Function Generator Agent is provided the `combinations` parameters and their descriptions produced by the Planning Agent, along with the structured solution generated by the Solution Agent.

The Combination Function Generator Agent leverages these information to construct Combination Function that produces candidate plans that adhere to the required structure. For instance, in Figure 2, the agent constructs the Combination Function to generate all permutations of the "cities" parameter to enumerate possible visiting orders, and then assigns visiting days using "city_stays". Each resulting plan conforms to the structured solution designed by Solution Agent.

Similarly, the Filter Function Generator Agent receives the `constraints` parameters and their descriptions, together with the same structured solution. Using this information, it constructs the Filter Function, which extracts relevant attributes from each plan and compares them against the specified

constraint values. In Figure 2, the generated Filter Function iterates through each plan produced by the Combination Function and checks whether the "specific_days" and "direct_flights" constraint requirements are satisfied by each plan, returning only the plan that meet all constraints.

The Deliver Function Generator Agent receives the structured solution along with ground-truth answer, enabling it to generate Deliver Function that converts any structured solution in the same format into a natural-language answer consistent with the desired output.

Solver Refinement. The initial functions generated by the Combination and Filter Function Generator Agents may be suboptimal. We refine them using the same example query and solution from the previous stage, with the Planning Agent's structured representation and the Solution Agent's structured solution as supervision. Refinement is applied when a function fails, defined as follows:

- **Combination Function**: The list of candidate plans generated by the function does not contain the structured solution.
- **Filter Function**: The plan returned by function does not match the structured solution.
- **Deliver Function**: The natural language answer produced by the function does not match the ground-truth answer.

By comparing the function outputs with the structured solution or ground-truth answer, the agents iteratively regenerate and adjust their functions until the expected output is produced.

We note that using a single example of query and solution during solver refinement does not cause overfitting. The Function Generator Agent is instructed to design functions that operate on input parameters, rather than hardcoding instance-specific values. Likewise, the Planning Agent constructs `combinations` and `constraints` to capture the general domain structure, rather than the specific keys or values of the example. Consequently, the generated solver functions generalize across queries, reflecting structural constraints rather than memorizing a single instance. For applications requiring additional assurance, solver functions can be validated and refined against multiple examples. In our experiments, a single example suffices.

3.4 Inference

The query and structured representation pair from Query-Specific Problem Reasoning stage is used as a single in-context exemplar that specifies the expected structured output format for inference. Specifically, at inference, the framework first uses a single LLM as the **Input Agent**, which takes the exemplar query, its corresponding `combinations` and `constraints` optimized from the problem reasoning stage as the one-shot prompt. Along with the test query appended in the prompt, the Input Agent generates `combinations` and `constraints` for the test query. These are further fed into the reusable solver functions generated by the SCOPE workflow to produce the final answer.

During inference, the agent does not generate or modify any solver code. Instead, it applies the pre-generated solver functions to produce the final solution. The inferred `combinations` are passed to the Combination Function to enumerate candidate plans, which are then filtered using the `constraints` by the Filter Function. Finally, the Deliver Function converts the structured solution into a natural language response. The details of SCOPE workflow is presented in Algorithm 1.

4 Experiments

We evaluate SCOPE on **multi-constraint planning tasks** that require reasoning over combinatorial possibilities and multiple interdependent constraints. We select **TravelPlanner** (Xie et al., 2024) and **Natural Plan** (Zheng et al., 2024) for benchmarking, which represent realistic and challenging planning scenarios. We compare our method with 6 baselines across 5 proprietary language models.

4.1 Datasets

TravelPlanner is a real-world travel planning benchmark, requiring agent to generate trip itineraries satisfying multiple constraints divided into two types, **Commonsense Constraints** and **Hard Constraints**. The details of each type of constraints and metrics are in Appendix D.1. To ensure fairness, all our baselines are provided the same deterministic constraints and operates in closed environment (refer to Appendix B.2).

Natural Plan is a natural-language planning benchmark where the agent is given query and has to produce the optimal plan. Our experiments focus on two dataset, **Trip Planning**[1] and **Meeting**

[1]Due to limited budget, we run half of the queries on

MODEL	SETTING	TravelPlanner						Trip.	Meet.
		DR	CPR		HCPR		**SR**	**SR**	**SR**
			Micro	Macro	Micro	Macro			
GPT-4o	Direct	100	93.6	57.0	56.4	38.7	23.6	4.9	50.7
	CoT (Wei et al., 2022)	100	95.2	67.0	64.8	45.2	31.5	3.9	47.4
	ToT (Yao et al., 2023a)	95.5	87.5	48.1	64.7	47.6	26.1	1.1	34.7
	EvoAgent (Yuan et al., 2025)	100	69.7	0.0	0.0	0.0	0.0	8.6	10.8
	HTP (Gui et al., 2025)	100	83.5	18.9	54.3	53.8	16.1	8.3	52.9
	ToS (Liu et al., 2024b)	65.2	28.7	0.0	0.0	0.0	0.0	12.5	59.8
	SCOPE	100	**99.7**	**97.8**	**97.6**	**94.8**	**93.1**	**87.1**	**100**
GPT-o3	Direct	100	96.3	71.1	65.9	66.9	66.5	78.8	70.5
	CoT (Wei et al., 2022)	100	98.6	89.1	83.2	80.7	80.4	77.9	89.8
	ToT (Yao et al., 2023a)	93.5	92.5	86.4	80.7	76.1	75.8	41.4	42.4
	EvoAgent (Yuan et al., 2025)	99.9	89.4	23.7	19.9	21.4	21.3	5.4	40.6
	HTP (Gui et al., 2025)	100	77.4	0.8	1.4	1.3	0.8	23.5	71.5
	ToS (Liu et al., 2024b)	61.0	60.7	58.6	49.9	41.7	41.7	0.0	59.7
	SCOPE	100	**99.6**	**97.5**	**97.2**	**94.0**	**92.2**	**89.1**	**99.2**
GPT-5	Direct	100	99.6	96.6	94.3	91.4	91.2	83.3	89.3
	CoT (Wei et al., 2022)	100	99.5	96.6	94.3	91.8	91.5	84.6	93.2
	ToT (Yao et al., 2023a)	99.8	99.3	95.9	90.3	85.3	85.3	61.6	55.8
	HTP (Gui et al., 2025)	100	82.4	14.0	18.5	17.2	11.5	5.1	67.4
	ToS (Liu et al., 2024b)	79.5	58.1	0.0	36.7	7.4	0.0	0.0	10.6
	SCOPE	100	**99.7**	**98.3**	**97.8**	**95.3**	**93.9**	**89.5**	**100**
Gemini-1.5-Pro	Direct	100	**96.8**	75.0	54.6	27.5	21.8	38.1	43.2
	CoT (Wei et al., 2022)	100	95.1	66.8	58.7	35.5	25.3	31.8	45.4
	SCOPE	97.5	95.9	**85.6**	**89.3**	**79.0**	**71.6**	**80.4**	**90.6**
Gemini-2.5-Pro	Direct	100	99.5	95.7	93.8	92.5	91.7	88.8	95.2
	CoT (Wei et al., 2022)	100	99.6	96.9	96.3	94.8	93.8	88.1	96.4
	SCOPE	100	**99.7**	**98.2**	**98.0**	**95.6**	**94.2**	**91.0**	**99.6**

Table 1: Performance comparison across models and settings in inference mode.

Planning of this benchmark. The details of each datasets and metrics are provided in Appendix D.2.

4.2 Language Models

We evaluate SCOPE and baselines on two LLMs families: **GPT** (GPT-5, GPT-o3, and GPT-4o) and **Gemini** (Gemini-2.5-Pro and Gemini-1.5-Pro). For GPT-5, GPT-o3, and Gemini-2.5-Pro, thinking mode is enabled for enhanced reasoning. For all models except for GPT-5 and GPT-o3, temperature is set to 0. Temperature for the other two models cannot be set in thinking mode. We use a specific version for each model (see Appendix B.1).

4.3 Baselines

Aside from direct prompting, we include both text-based and solver-based reasoning methods as baseline. The text-based reasoning are Chain-of-Thought (CoT) (Wei et al., 2022), Tree-of-Thought (ToT) (Yao et al., 2023a), EvoAgent (Yuan et al., 2025) and HyperTree Planning (HTP) (Gui et al., 2025). For solver-based reasoning, we use Thought of Search (ToS) (Liu et al., 2024b). The implementation details of each baseline are described in Appendix B.3. Since SCOPE uses an example query and answer to construct solver functions, all baselines are also given few-shot examples for fair comparison. More details for fairness across baseline are described in Appendix B.2.

In our experiments, we exclude certain solver-based and code-based baselines which rely on human-provided hints or manual template code (Hao et al., 2025a,b; Wang et al., 2025). More justifications can be found in Appendix B.4. The evaluation metrics and the difficulty of different datasets are defined in Appendix D.1 for TravelPlanner and Appendix D.2 for Natural Plan.

5 Results and Analysis

5.1 Overall Success Rate

Table 1 compares the performances of different methods across different language models[2]. The key results are as follows:

SCOPE consistently enhances performance across all evaluated models and datasets. The impact is especially more significant on weaker foundational models like GPT-4o and Gemini-1.5-Pro. For example, the success rate of GPT-4o improves from 12.5% (ToS) to 87.1% in Trip Planning, 59.8% (ToS) to 100% on Meeting Planning, and 31.5% (CoT) to 93.1% on TravelPlanner. These results not only demonstrates SCOPE's strong generalization across diverse models and benchmarks,

Trip Planning, with each level of complexity equally sampled. For Direct, CoT and SCOPE, we run full dataset and have a separate table of results at Appendix, Table 4.

[2]We choose to selectively implement baselines on Gemini models due to the high cost of baselines and their tendency to fail when following complex instructions.

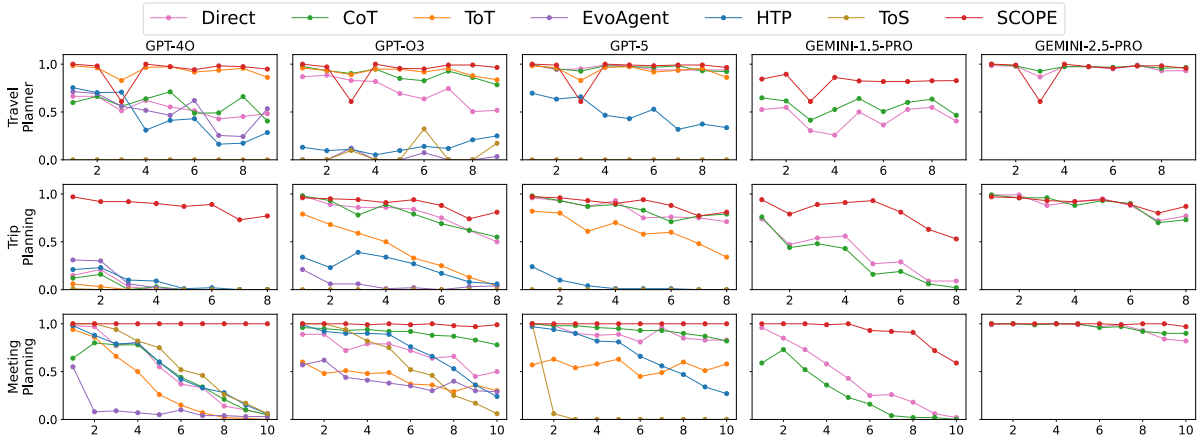

Figure 3: The success rate (y-axis) across different level of complexity (x-axis). Top to bottom (Different benchmarks): TravelPlanner, Trip Planning, Meeting Planning. Left to right (Different models): GPT-5, GPT-o3, GPT-4o, Gemini-2.5-Pro, Gemini-1.5-Pro.

but also shows SCOPE provides a robust mechanism that enables substantial performance gains on complex tasks where weaker foundational models exhibit limited success.

SCOPE enables smaller models to achieve performance comparable to larger models of baselines by leveraging explicit, deterministic logic. Gains are smaller on very strong models (e.g., GPT-5 and Gemini-2.5-Pro), reflecting their strong reasoning capabilities. On smaller models (GPT-4o, Gemini-1.5-Pro), SCOPE matches or outperforms baselines using larger models at much lower cost (refer to Table 4). For example in Trip Planning, GPT-4o's SCOPE (87.1%) outperforms the best baseline, CoT of GPT-o3 (78.8%) and GPT-5 (84.6%) respectively. This demonstrates that SCOPE's advantages arise from deterministic, executable logic rather than model scale.

5.2 Success Rate Over Complexity

We analyze the success rate of baselines and SCOPE across datasets of increasing complexity. The results are visualized in Figure 3.

SCOPE displays robust performance as problem complexity increases, while baseline methods exhibit substantial performance degradation. SCOPE consistently maintains a high success rate (often above 90%) across all complexity levels and datasets, with only a minimal decline at the most extreme levels. In contrast, baseline methods, particularly those relying on weaker foundational models (GPT-4o and Gemini-1.5-Pro), collapses rapidly as difficulty increases. These results shows SCOPE offers superior generalization and scalability compared to existing methods.

SCOPE shows superior usability over long-text reasoning for complex multi-constraint planning tasks. Baselines that rely on long-text reasoning (e.g., ToT, EvoAgent, and HTP) degrade rapidly as combinatorial complexity increases, due to incomplete enumeration of candidate plans and increased risk to intermediate reasoning errors. Through exhaustive enumeration of candidate plans and logic-based verification, SCOPE maintains stable performance under increasing combinatorial complexity, underscoring its solver-based advantages over text-driven reasoning.

5.3 Cost and Time Efficiency

Figure 4 presents the inference costs and time of different methods across varying levels of complexity and model types. The summary of corresponding inference costs and latency for each dataset is shown in Appendix, Table 5.

SCOPE achieves substantial cost efficiency for multi-constraint planning tasks. Unlike multi-step and multi-agent methods (e.g., ToT, EvoAgent, HTP), which incur exponentially growing API costs due to multiple text generation, SCOPE limits textual output and performs plan enumeration at the code level. This efficient scaling enables it to handle complex, long-horizon planning problems while outperforming iterative reasoning baselines in cost.

SCOPE demonstrates a robust trade-off between inference time and solution reliability. Across all models and datasets, SCOPE requires substantially less inference time than multi-step,

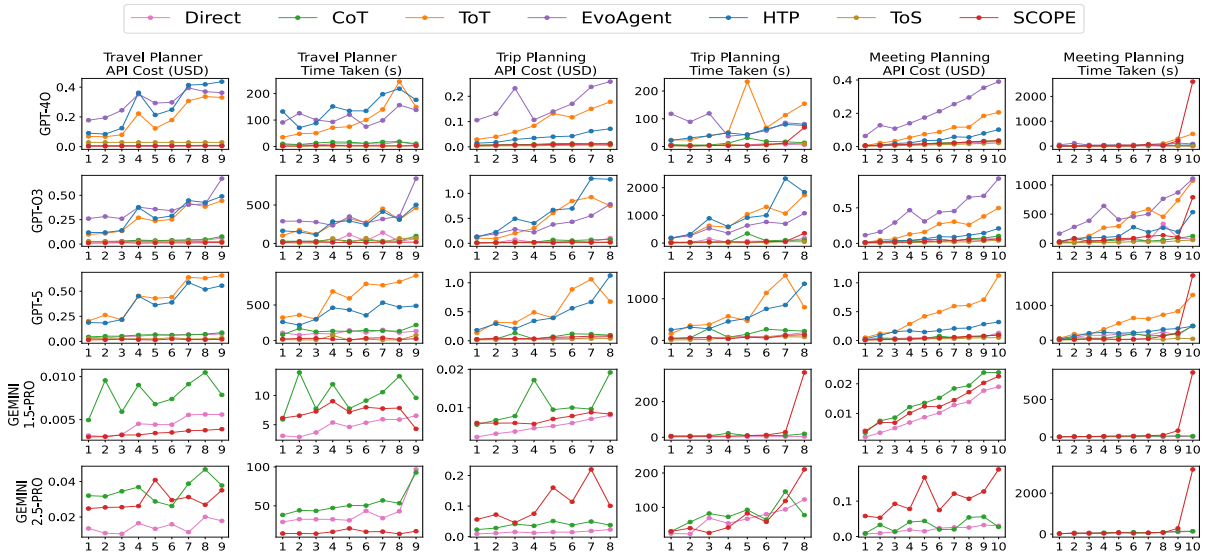

Figure 4: The cost/time required per query (y-axis) for different methods across different level of complexity (x-axis). Top to bottom (Different models): GPT-5, GPT-o3, GPT-4o, Gemini-2.5-Pro, Gemini-1.5-Pro. Left to right (Different metrics): Cost and time taken for TravelPlanner, Trip Planning, and Meeting Planing.

long-text reasoning baselines (ToT, EvoAgent, HTP), while remaining comparable to single-agent, single-step (Direct, CoT) and solver-based (ToS) methods. Although single-step methods incur low per-inference cost, SCOPE is more efficient when accounting for the total cost and time needed to obtain a verified correct solution (Figure 5). While SCOPE's runtime increases moderately with problem complexity, it consistently produces valid solutions where baselines fail (Figure 3), highlighting its scalability and robustness.

5.4 Error Analysis

Failures in our method primarily arise from Input Agent. As shown in Appendix F.1, smaller models (Gemini-1.5-Pro) may incorrectly assign the value of parameters in the structured representation. Appendix F.2 shows that one-shot demonstration can cause the agent to overgeneralize patterns from examples, resulting in invalid constraint interpretations. Appendix F.3 illustrates that for tasks requiring large structured outputs (Meeting Planning), Input Agent may introduce subtle errors, such as incorrect numerical values, in its output.

6 Ablation Study

We conduct an ablation study on the TravelPlanner and Natural Plan tasks by removing individual components of our framework to assess their impact. Specifically, we ablate problem formalization, problem optimization, and solver refinement.

The results from Table 2 indicates that each component is essential in the workflow as the absence of component will cause a huge decline in performance. The details of the implementation of ablation is in Appendix C.1 and analysis of absence of each components is in Appendix C.2.

Ablation Study	Travel Planner	Trip	Meeting
No Problem Formalisation	25.2	13.3	100
No Problem Optimisation	53.6	0.0	56.4
No Solver Refinement	93.1	0.0	56.2
SCOPE	93.1	87.1	100

Table 2: Ablation study results from GPT-4o by removing each component from SCOPE workflow.

7 Conclusion

In this work, we introduced SCOPE, a scalable code-based planning framework that disentangles problem reasoning from code execution to address the challenges of multi-constraint complex planning with LLMs. By formalizing natural language queries into structured representations and leveraging reusable solver functions, SCOPE achieves robustness, consistency, and efficiency beyond existing text-based or code-based approaches. Our experiment results show significant improvements in accuracy, cost, and latency, highlighting the effectiveness of systematic, reusable solvers. Our work paves a new way for efficient and robust LLM-based planning in complex real-world tasks.

Limitations

While our proposed framework represents a significant step forward in enhancing LLM-based agents' planning abilities, it has some limitations. One limitation is the reliance on LLMs' coding skills and formatted generation capabilities. Hence, our current experiments primarily evaluate closed-source models rather than open-source models. Another limitation is that solver functions are still tied to a specific domain. While they can be reused across different queries within the same domain, they cannot be directly applied to a new domain without redefining the problem structure. Enabling solver generalization across domains is a promising direction for future work.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F. Karlsson, Jie Fu, and Yemin Shi. 2024a. Autoagents: A framework for automatic agent generation. *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI)*.

Sijia Chen and Baochun Li. 2024. Toward adaptive reasoning in large language models with thought rollback. In *Proceedings of the 41st International Conference on Machine Learning*, pages 7033–7056.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2024b. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. *Proceedings of the 12th International Conference on Learning Representations (ICLR)*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, Xu Sun, Lei Li, and Zhifang Sui. 2024. A survey on in-context learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP 2024)*.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36:79081–79094.

Runquan Gui, Zhihai Wang, Jie Wang, Chi Ma, Huiling Zhen, Mingxuan Yuan, Jianye Hao, Defu Lian, Enhong Chen, and Feng Wu. 2025. Hypertree planning: Enhancing llm reasoning via hierarchical thinking. *Proceedings of the 42nd International Conference on Machine Learning*.

Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, and Zhaozhuo Xu. 2024. Llm multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*.

Yilun Hao, Yongchao Chen, Yang Zhang, and Chuchu Fan. 2025a. Large language models can solve real-world planning rigorously with formal verification tools. *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Yilun Hao, Yang Zhang, and Chuchu Fan. 2025b. Planning anything with rigor: General-purpose zero-shot planning with llm-based formalized programming. *Proceedings of the 13th International Conference on Learning Representations (ICLR)*.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. 2024. Llms can't plan, but can help planning in llm-modulo frameworks. *arXiv*.

Kuang-Huei Lee, Ian Fischer, Yueh-Hua Wu, Dave Marwood, Shumeet Baluja, Dale Schuurmans, and Xinyun Chen. 2025. Evolving deeper llm thinking. *arXiv preprint arXiv:2501.09891*.

Anthony Z Liu, Xinhe Wang, Jacob Sansom, Yao Fu, Jongwook Choi, Sungryull Sohn, Jaekyeom Kim, and Honglak Lee. 2024a. Interactive and expressive code-augmented planning with large language models. *arXiv preprint arXiv:2411.13826*.

Michael Liu, Harsha Kokel, Kavitha Srinivas, and Shirin Sohrabi. 2024b. Thought of search: Planning with language models through the lens of efficiency. *arXiv preprint arXiv:2404.11833*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. *Advances in Neural Information Processing Systems*, 36:38975–38987.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. 2023. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*.

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

Zi Wang, Shiwei Weng, Mohannad Alhanahnah, Somesh Jha, and Tom Reps. 2025. Pea: Enhancing llm performance on computational-reasoning tasks. *arXiv*.

Hui Wei, Zihao Zhang, Shenghua He, Tian Xia, Shijia Pan, and Fei Liu. 2025. Plangenllms: A modern survey of llm planning capabilities. *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *Proceedings of the 36th Conference on Neural Information Processing Systems (NeurIPS)*.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. 2024. Travelplanner: A benchmark for real-world planning with language agents. *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and 1 others. 2025. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms. *arXiv preprint arXiv:2502.19411*.

Zhun Yang, Adam Ishay, and Joohyung Lee. 2023. Coupling large language models with logic programming for robust and general reasoning from text. *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5186–5219.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Sharfan, and Tom Griffiths. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*.

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 *Proceedings of the International Conference on Learning Representations (ICLR)*.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. 2025. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6192–6217, Albuquerque, New Mexico. Association for Computational Linguistics.

Cong Zhang, Xin Deik Goh, Dexun Li, Hao Zhang, and Yong Liu. 2025. Planning with multi-constraints via collaborative language agents. *Proceedings of the 31st International Conference on Computational Linguistics (COLING)*.

Huaixiu Steven Zheng, Swaroop Mishra, Hugh Zhang, Xinyun Chen, Minmin Chen, Azade Nova, Le Hou, Heng-Tze Cheng, Quoc V. Le, Ed H. Chi, and Denny Zhou. 2024. Natural plan: Benchmarking llms on natural language planning. *arXiv*.

A Pseudocode

Algorithm 1 SCOPE: Structured Solver Construction and Inference

Input: Example query–solution pair (Q^{ex}, S^{ex}), test queries $\{Q_i\}_{i=1}^{N}$
Output: Natural-language solutions $\{S_i\}_{i=1}^{N}$

▷ **Stage I: Query-Specific Problem Reasoning**

$(\mathcal{C}, \mathcal{K}) \leftarrow \mathcal{P}(Q^{ex})$ ▷ Planning Agent $\mathcal{P}$: extracts combinations $\mathcal{C}$ and constraints $\mathcal{K}$ from the example query
$\mathcal{S} \leftarrow \mathbf{\Sigma}(Q^{ex})$ ▷ Solution Agent $\mathbf{\Sigma}$: defines target solution structure $\mathcal{S}$
for each optimization agent $\mathcal{O}_j \in \boldsymbol{\mathcal{O}}$ **do**
    $(\mathcal{C}, \mathcal{K}) \leftarrow \mathcal{O}_j(\mathcal{C}, \mathcal{K})$
end for ▷ Optimization Agents $\boldsymbol{\mathcal{O}}$: sequentially refine $\mathcal{C}$ and $\mathcal{K}$

▷ **Stage II: Generic Solver Generation**

▷ Combination Solver Generator $\mathcal{G}_{\text{comb}}$: constructs F_{comb}
$F_{\text{comb}} \leftarrow \mathcal{G}_{\text{comb}}(\mathcal{C})$ ▷ Initial construction, no previous output or supervision yet
while $\mathcal{S}F_{\text{comb}}(\mathcal{C})$ **do**
    $F_{\text{comb}} \leftarrow \mathcal{G}_{\text{comb}}(\mathcal{C}, F_{\text{comb}}(\mathcal{C}), \mathcal{S})$ ▷ Refinement: use previous output + ground-truth structure $\mathcal{S}$
end while

▷ Filter Solver Generator $\mathcal{G}_{\text{filter}}$: constructs F_{filter}
$F_{\text{filter}} \leftarrow \mathcal{G}_{\text{filter}}(\mathcal{K})$ ▷ Initial construction, no previous output or supervision yet
while $\mathcal{S} \neq F_{\text{filter}}(F_{\text{comb}}(\mathcal{C}), \mathcal{K})$ **do**
    $F_{\text{filter}} \leftarrow \mathcal{G}_{\text{filter}}(\mathcal{K}, F_{\text{filter}}(F_{\text{comb}}(\mathcal{C}), \mathcal{K}), \mathcal{S})$ ▷ Refinement: use previous output + ground-truth structure $\mathcal{S}$
end while

▷ Deliver Solver Generator $\mathcal{G}_{\text{deliver}}$: constructs F_{deliver}
$F_{\text{deliver}} \leftarrow \mathcal{G}_{\text{deliver}}(\mathcal{S})$ ▷ Initial construction, no previous output or supervision yet
while $S^{ex} \neq F_{\text{deliver}}(\mathcal{S})$ **do**
    $F_{\text{deliver}} \leftarrow \mathcal{G}_{\text{deliver}}(\mathcal{S}, F_{\text{deliver}}(\mathcal{S}), S^{ex})$ ▷ Refinement: use previous output + ground-truth solution S^{ex}
end while

▷ **Stage III: Inference**

for $i = 1$ to N **do**
    $(\mathcal{C}_i, \mathcal{K}_i) \leftarrow \mathcal{I}(Q_i \mid Q^{ex}, \mathcal{C}, \mathcal{K})$ ▷ Input Agent $\mathcal{I}$: one-shot demonstration using Q^{ex} and $(\mathcal{C}, \mathcal{K})$
    $\mathcal{X}_i \leftarrow F_{\text{comb}}(\mathcal{C}_i)$
    $\mathcal{Y}_i \leftarrow F_{\text{filter}}(\mathcal{X}_i, \mathcal{K}_i)$
    $S_i \leftarrow F_{\text{deliver}}(\mathcal{Y}_i)$
end for
return $\{S_i\}_{i=1}^{N}$

B Experiment Details

B.1 Model Versions

1. GPT-5: gpt-5-2025-08-07
2. GPT-o3: o3-2025-04-16
3. GPT-4o: gpt-4o-2024-11-20
4. Gemini-2.5-Pro: gemini-2.5-pro-preview-05-06
5. Gemini-1.5-Pro: gemini-1.5-pro-002

B.2 Fairness Considerations

To ensure a fair comparison across methods, several design decisions are enforced. First, few-shot examples are allowed for all methods. For non-solver-based methods such as Direct Prompting, CoT, ToT, and EvoAgent, example queries and their corresponding answers are provided directly in the prompt. For solver-based methods such as ToS and SCOPE, example queries and answers are incorporated into the solver construction pipeline rather than the prompt itself. The examples of query and solution for TravelPlanner are obtained from the train data while the examples of query and solution for Natural Plan tasks are obtained from the few-shot questions.

Second, all code used in solver-based methods is written by the model itself. No human-written code snippets or task-specific hints are provided. Prior work that embeds human-written code or domain-specific hints is not representative of ordinary natural-language inputs, as models are typically given only the query and must generate their own reasoning. Providing human-written code or hints would partially replace model reasoning, making it difficult to attribute performance gains to the model's own capabilities. Our aim is to evaluate whether the model can independently construct solver functions from natural language queries alone.

Third, all experiments are conducted in a closed environment, meanings agents operate under deterministic constraints. In the Natural Plan task, queries and example answers already define a closed environment, allowing models can deduce explicit internal constraints.

In contrast, the TravelPlanner task admits many possible interpretations of constraints. For example, rules such as "multiple vehicles are booked if capacity is exceeded" exist in the environment, but the agent cannot infer them from the query alone. Without explicit deterministic constraints, agents may interpret such rules differently, potentially generating inconsistent plans. To ensure tractability and fair evaluation, these constraints are formalized as deterministic constraints and provided across all baselines.

The TravelPlanner environment is formalized using a set of declarative deterministic constraints as follows:

1. Travel between locations occurs on odd-numbered days, while staying within the same city occurs on even-numbered days.
2. For each city stay, exactly one accommodation, one attraction, and three distinct restaurants are required. The same accommodation must be used for two consecutive days within the city.
3. If an accommodation's maximum room capacity exceeds the number of travelers, the agent assumes the user will still book the accommodation, potentially reserving multiple rooms of the same type.
4. If an accommodation requires a minimum number of nights that exceeds the minimum nights specified in the query, the plan is considered invalid.
5. Flight costs are computed on a per-ticket basis.
6. A self-driving car is assumed to have a capacity of five people, while a taxi has a capacity of four people. If the number of travelers exceeds vehicle capacity, multiple vehicles of the same type are assumed to be booked.
7. Combinations of self-driving with flights or taxis are prohibited.
8. Restaurants must be unique across the entire trip, repeated restaurant visits are not allowed.

B.3 Implementation Details of Baselines

This section provides a high-level overview of the experimental setup used in evaluating SCOPE and baseline methods.

- **Direct Prompting**: The model is presented with the query and few-shot demonstrations, and is required to directly produce the final

answer without any explicit reasoning guidance.

- **Chain-of-Thought (CoT)** (Wei et al., 2022): Generates intermediate reasoning steps in a zero-shot manner before producing the final solution, following the standard CoT prompting paradigm.

- **Tree-of-Thought (ToT)** (Yao et al., 2023a): Explores multiple reasoning trajectories through iterative tree expansion. The branching factor at each step is determined by the model, and a heuristic critic is used to select candidate nodes.

- **EvoAgent** (Yuan et al., 2025): A multi-agent test-time scaling baseline that formulates reasoning as an evolutionary process over a population of agents. Each agent maintains a candidate solution, and iterative mutation and selection are applied across the population based on model-based evaluation, enabling progressive refinement of reasoning trajectories.

- **HyperTree Planning (HTP)** (Gui et al., 2025): Constructs a hypertree-structured outline to decompose complex tasks into multiple sub-tasks, iteratively refining and expanding the structure to guide the reasoning process.

- **Thought of Search (ToS)** (Liu et al., 2024b): A code-based reasoning framework that formulates problem solving as an explicit search procedure. The model generates executable code defining a successor function to expand candidate states and a goal test function to evaluate goal states, enabling systematic exploration and pruning of the solution space via programmable search.

B.4 Excluded Solver-Based Baselines

We do not include several recent solver-based approaches in our experimental comparison due to fundamental differences in assumptions and evaluation setup:

- **Large Language Models Can Solve Real-World Planning Rigorously with Formal Verification Tools** (Hao et al., 2025a) : This approach requires human-written example code for the TravelPlanner task, with explicit solver logic provided to the model. Such manual intervention replaces substantial portions of the model's reasoning process, making it difficult to attribute performance gains to the model itself.

- **Planning Anything with Rigor: General-Purpose Zero-Shot Planning with LLM-based Formalized Programming** (Hao et al., 2025b): The method, LLMFP assumes task descriptions can be clearly specified in queries, or else it struggles to define problems' goals and constraints. Our tasks are designed to require model to identify and infer internal constraints that are not explicitly stated.

- **PEA: Enhancing LLM Performance on Computational-Reasoning Tasks**: (Wang et al., 2025) This framework provides explicit, human-specified instructions for the construction of each solver functions and the parameters for respective functions. As a result, parts of the reasoning are externally supplied rather than inferred by the model.

In contrast, our approach provides no human-written code templates or structured solver hints. All parameters to generate combinations and constraints are inferred by the model from a single query and example. Furthermore, unlike prior approaches that generate instance-specific executable code, our method constructs reusable solver functions shared across all test queries, substantially improving computational efficiency.

B.5 Exclusion of Additional Text-Based Reasoning and Multi-Agent Baselines

We exclude several text-based reasoning baselines and multi-agent baselines. The reasons are as follows :

- **Self-Refine** (Madaan et al., 2023): This method performs iterative refinement using self-generated feedback. The original EvoAgent paper shows that Self-Refine is consistently outperformed by EvoAgent on structured planning and reasoning tasks and shows that Self-Refine only achieves 1.1% success rate on TravelPlanner with GPT-4.

- **SPP (Self-collaborative Persona Prompting)** (Wang et al., 2023): SPP relies on multi-persona self-collaboration to enhance Chain-of-Thought reasoning. However, the original EvoAgent paper shows that it underperforms SPP on planning-intensive tasks and shows

that Self-Refine only achieves 0.6% success rate on TravelPlanner with GPT-4.

- **AgentVerse** (Chen et al., 2024b), **AutoAgents** (Chen et al., 2024a), and **Meta-Task Planning (MTP/PMC)** (Zhang et al., 2025): These frameworks focus on coordinating multiple agents via a central planning agent (referred as a Recruiter in AgentVerse, a Manager in MTP, and a Planner in AutoAgents), that decomposes tasks and assign subtasks to the generated agents. Conceptually, they are related. Since these methods rely on text-based reasoning, they share the the same limitation as the text-based reasoning baseline in our experiments, which is an excessive combinatorial search space. Hence, the underperformance of those baselines suffices to draw the same conclusion.

EvoAgent and HyperTree Planning (HTP) are chosen as primary baselines for text-based reasoning as they represent the strongest recent text-based multi-step reasoning methods. The original EvoAgent paper shows EvoAgent surpasses Self-Refine and SPP, while the original HTP paper shows HTP outperforms EvoAgent and MTP. Since our method consistently outperforms HTP and EvoAgent across all tasks, adding other baselines would not change our conclusions.

C Details of Ablation Study

Our evaluation uses GPT-4o as the backbone model. Each version of ablation removes one component from SCOPE, except solver construction.

C.1 Implementation of Ablation

- **No Problem Formalization** : The Planning Agent no longer separates parameters into combinations and constraints. Specifically, the Planning Agent is not instructed to output `combinations` and `constraints` keys in structured representation but output `parameters` instead. The Combination and Constraint Function Generator Agent must implicitly decide which parameters are relevant for the function. We set the patience to 3, which is the frequency the model will refine the function and move on if the function is not able to meet the objective, to avoid deadlock.

- **No Problem Optimization** : For Natural Plan, the Optimization Agents that optimize the structured representations are removed from the workflow. The structured representation generated by Planning Agent are given to the Combination and Constraints Function Generator Agent. For TravelPlanner, it removes the modules that automatically filter, prune and package reference data. Due to the resulting explosion of search space, the generated Combination Function produce at most 100,000 sampled candidate plans.

- **No Problem Refinement** : The workflow removes the weak supervision process. As long as the functions generated are executable and able to return plans, they are not further optimized and the logic correctness of functions are not assured.

We set the patience to 3, meaning that the Function Generator Agent attempts to refine its function up to 3 times before proceeding, preventing potential deadlocks when the objective cannot be met.

C.2 Analysis of Effect of Ablation

The effects of the absence of each component of SCOPE are as follows :

- **Problem Formalization helps Planning Agents to be more precise with the parameters**: By explicitly separating parameters into `combinations` and `constraints`, the Planning Agent is less likely to omit information required for candidate generation or constraint filtering. In the Trip Planning task, performance degrades when this separation is absent, as essential constraint parameters may be overlooked. In TravelPlanner, the Combination Function is difficult to be constructed when combination parameters cannot be cleanly isolated from constraints.
- **Problem Optimization helps prevent logical errors in plan construction due to poorly specified parameters**: In the Trip Planning task, the removal of Problem Optimization causes failure to design Combination Function that return any valid plans because the Planning Agent identifies redundant parameters that interfere with correct plan construction. In Meeting Planning, the Planning Agents omits parameters in either `combinations` or `constraints`, leading to missing parameters to construct logically correct functions. For TravelPlanner, the absence of an optimization

module forces the Combination Function to rely on random plan generation without effective pruning or data restructuring, causing many valid candidate combinations to be missed.

- **Problem Refinement corrects incorrect logic in functions.**: For Trip Planning and Meeting Planning, the initial generated functions are sometimes incorrect. The absence of refinement causing the errors to propagate into inference queries as the same functions are used.

D Evaluation Details

Our evaluation uses all 1,000 test data from TravelPlanner, 800 test data (sampled equally across difficulties) from Trip Planning and all 1,000 test data from Meeting Planning.

D.1 TravelPlanner

The constraints in TravelPlanner are divided into two types :

- **Commonsense Constraints**: Implicit, commonsense knowledge for generating plausible plans, such as not visiting the same restaurants and attractions throughout the trip.
- **Hard Constraints**: Explicitly mentioned user requirements such as prefers self-driving and wants smoking-free accommodation.

Our evaluation uses the same metrics from the original paper as follows :

- **Delivery Rate (DR)**: Measures whether an agent successfully output a plan.
- **Commonsense Pass Rate (CPR)**: Evaluates the agent's ability to generate plans that do not violate commonsense. An example of violation is visiting the same attraction or restaurant in a trip.
- **Hard Constraint Pass Rate (HCPR)**: Evaluates the agent's ability to meet user's preferences that are explicitly mentioned in each query.
- **Success Rate (SR)**: Evaluates the proportion of generated plans that meet all criteria from commonsense and hard constraint. This is the most important metric as it indicates the agent's overall effectiveness in producing practical plans.

In addition, TRAVELPLANNER computes the scores of CPR and HCPR via two metrics:

- **Micro**: Computes the fraction of satisfied constraints over the total number of constraints.
- **Macro**: Measures the proportion of plans that satisfy all commonsense or all hard constraints.

Task difficulty is originally categorized into three levels: *easy*, *medium*, and *hard*. Easy queries primarily involve budget constraints for a single traveler, medium queries introduce additional hard constraints (e.g., cuisine type, room type, or room rules) and vary the number of travelers, hard queries further incorporate transportation preferences and multiple simultaneous constraints.

In our analysis, we further stratify tasks by planning horizon, resulting in nine complexity levels in total. Levels 1–3 correspond to 3-day plans (easy, medium, hard), Levels 4–6 correspond to 5-day plans, and Levels 7–9 correspond to 7-day plans. This setup enables a finer-grained evaluation of performance as both constraint difficulty and planning horizon increase.

D.2 Natural Plan

The two datasets of Natural Plan we are evaluating are as follows :

- **Trip Planning:** 1,600 queries where user requests to design a trip itinerary to travel to multiple cities. The constraints are total number of days in each city, specific days where the user wants to be in a city and available flight connectivity from one city to another.
- **Meeting Planning:** 1,000 queries focused on scheduling meetings that maximize the numbers of friends met. The constraints include the available location and meeting time of friends and distance from one place to another.

For both Trip Planning and Meeting Planning tasks, we evaluate model performance using the **Success Rate** metric like the original paper. Since each query has a single correct plan (gold plan), the success rate is defined as the fraction of generated plans that exactly match the gold plan for the corresponding query.

For Trip Planning, difficulty is defined by the number of cities to visit, increasing from 3 to 8 across 8 levels. For Meeting Planning, difficulty is determined by the number of people to meet, increasing from 1 to 10 across 10 levels. This stratification allows analysis of performance as task complexity increases with the size of the planning problem.

E Inference Cost and Efficiency

This appendix presents a comprehensive analysis of the inference cost and efficiency of SCOPE and baseline methods. We report the number of input tokens, the number of output tokens, including intermediate reasoning tokens produced by models operating in thinking modes such as GPT-o3, GPT-5, and Gemini-2.5-Pro, as well as the total inference cost and end-to-end wall-clock inference time. All measurements are conducted under identical experimental settings and evaluation prompts to ensure fair comparison. Token counts include all tokens consumed during inference, including intermediate reasoning tokens where applicable. Inference cost is computed based on the official pricing of each model at the time of evaluation, and inference time is measured as the full wall-clock duration required to generate a final solution, including all agent interactions and tool calls. Results on the full evaluation set indicate that SCOPE maintains superior task performance while achieving competitive inference cost and runtime compared to baseline methods.

Model	Input Token Price	Output Token Price
GPT-5	1.25	10
GPT-o3	2.00	8
GPT-4o	2.50	10
Gemini-2.5-Pro	1.25	15
Gemini-1.5-Pro	1.25	10

Table 3: Official model pricing (in USD per 1M tokens).

F Error Analysis

F.1 Inference Errors Arising from Mistakes Caused by Limited Instruction-Following Capability

Error from Input Agent in Trip Planning (Gemini-1.5-Pro)

```
Query: You plan to visit 3 European
    cities for 10 days in total. You
    only take direct flights to commute
    between cities. You would like to
    visit Venice for 6 days. You have to
    attend a workshop in Venice between
    day 5 and day 10. You want to spend
    2 days in Mykonos. You plan to stay
    in Vienna for 4 days.

Here are the cities that have direct
    flights:
Mykonos and Vienna, Vienna and Venice.

Output of Input Agent: {'combinations':
    {'cities': ['Venice', 'Mykonos',
    'Vienna'], 'days_in_city':
    {'Venice': 6, 'Mykonos': 2,
    'Vienna': 2}}, 'constraints':
    {'direct_flights': [['Mykonos',
    'Vienna'], ['Vienna', 'Mykonos'],
    ['Vienna', 'Venice'], ['Venice',
    'Vienna']], 'fixed_days': {'Venice':
    [[5, 10]]}}}

Error : In 'days_in_city', the value for
    'Vienna' should be 4 not 2.
```

F.2 Inference Error Due to Misinterpretation of Demonstration Outputs

Error from Input Agent in Trip Planning (GPT-4o)

```
Demonstration Query : You plan to visit
    10 European cities for 21 days in
    total. You only take direct flights
    to commute between cities. You plan
    to stay in Vienna for 3 days. Youend
    2 days in Oslo. You are going to
    attend a wedding want to spend 5
    days in Frankfurt. You want to sp in
    Oslo between day 20 and day 21. You
    want to spend 3 days in Prague. You
    would like to visit Valencia for 2
    days. You want to meet a friend in
    Valencia between day 17 and day 18.
    You plan to stay in Dubrovnik for 2
    days. You would like to visit
    Edinburgh for 5 days. From day 1 to
    day 5, there is a annual show you
    want to attend in Edinburgh. You
    plan to stay in London for 2 days.
    You plan to visit relatives in
    London between day 12 and day 13.
    You would like to visit Munich for 3
    days. You would like to meet your
    friends at Munich between day 18 and
    day 20 to tour together. You would
    like to visit Budapest for 3 days.

Here are the cities that have direct
    flights:
Valencia and Munich, Vienna and Munich,
    Vienna and Valencia, London and
    Budapest, London and Oslo, Edinburgh
    and Budapest, Frankfurt and
    Budapest, Frankfurt and London,
    Prague and Oslo, Edinburgh and Oslo,
    Edinburgh and Munich, Prague and
    Munich, London and Prague, Edinburgh
    and London, Edinburgh and Frankfurt,
    Dubrovnik and Munich, Dubrovnik and
    Vienna, Munich and Oslo, Dubrovnik
    and Oslo, Budapest and Munich,
    Frankfurt and Prague, Vienna and
    London, Frankfurt and Vienna,
    Frankfurt and Oslo, Frankfurt and
    Munich, Vienna and Oslo, Vienna and
    Prague, Budapest and Oslo, Budapest
```

```
    and Prague, London and Valencia,
    London and Munich, Frankfurt and
    Dubrovnik, Prague and Valencia,
    Frankfurt and Valencia, Edinburgh
    and Prague, Vienna and Budapest.

Demonstration Output : {'combinations':
    {'cities': ['Edinburgh',
    'Frankfurt', 'Dubrovnik', 'Vienna',
    'London', 'Budapest', 'Prague',
    'Valencia', 'Munich', 'Oslo'],
    'direct_flights': [['Valencia',
    'Munich'], ['Munich', 'Valencia'],
    ['Vienna', 'Munich'], ['Munich',
    'Vienna'], ['Vienna', 'Valencia'],
    ['Valencia', 'Vienna'], ['London',
    'Budapest'], ['Budapest', 'London'],
    ['London', 'Oslo'], ['Oslo',
    'London'], ['Edinburgh',
    'Budapest'], ['Budapest',
    'Edinburgh'], ['Frankfurt',
    'Budapest'], ['Budapest',
    'Frankfurt'], ['Frankfurt',
    'London'], ['London', 'Frankfurt'],
    ['Prague', 'Oslo'], ['Oslo',
    'Prague'], ['Edinburgh', 'Oslo'],
    ['Oslo', 'Edinburgh'], ['Edinburgh',
    'Munich'], ['Munich', 'Edinburgh'],
    ['Prague', 'Munich'], ['Munich',
    'Prague'], ['London', 'Prague'],
    ['Prague', 'London'], ['Edinburgh',
    'London'], ['London', 'Edinburgh'],
    ['Edinburgh', 'Frankfurt'],
    ['Frankfurt', 'Edinburgh'],
    ['Dubrovnik', 'Munich'], ['Munich',
    'Dubrovnik'], ['Dubrovnik',
    'Vienna'], ['Vienna', 'Dubrovnik'],
    ['Munich', 'Oslo'], ['Oslo',
    'Munich'], ['Dubrovnik', 'Oslo'],
    ['Oslo', 'Dubrovnik'], ['Budapest',
    'Munich'], ['Munich', 'Budapest'],
    ['Frankfurt', 'Prague'], ['Prague',
    'Frankfurt'], ['Vienna', 'London'],
    ['London', 'Vienna'], ['Frankfurt',
    'Vienna'], ['Vienna', 'Frankfurt'],
    ['Frankfurt', 'Oslo'], ['Oslo',
    'Frankfurt'], ['Frankfurt',
    'Munich'], ['Munich', 'Frankfurt'],
    ['Vienna', 'Oslo'], ['Oslo',
    'Vienna'], ['Vienna', 'Prague'],
    ['Prague', 'Vienna'], ['Budapest',
    'Oslo'], ['Oslo', 'Budapest'],
    ['Budapest', 'Prague'], ['Prague',
    'Budapest'], ['London', 'Valencia'],
    ['Valencia', 'London'], ['London',
    'Munich'], ['Munich', 'London'],
    ['Frankfurt', 'Dubrovnik'],
    ['Dubrovnik', 'Frankfurt'],
    ['Prague', 'Valencia'], ['Valencia',
    'Prague'], ['Frankfurt',
    'Valencia'], ['Valencia',
    'Frankfurt'], ['Edinburgh',
    'Prague'], ['Prague', 'Edinburgh'],
    ['Vienna', 'Budapest'], ['Budapest',
    'Vienna']], 'city_stays':
    {'Edinburgh': 5, 'Frankfurt': 5,
    'Dubrovnik': 2, 'Vienna': 3,
    'London': 2, 'Budapest': 3,
    'Prague': 3, 'Valencia': 2,
    'Munich': 3, 'Oslo': 2}},
    'constraints': {'specific_days':
    {'Edinburgh': [1, 5], 'Oslo': [20,
    21], 'Valencia': [17, 18], 'London':
    [12, 13], 'Munich': [18, 20]}}}

Query: You plan to visit 3 European
    cities for 15 days in total. You
    only take direct flights to commute
    between cities. You want to spend 6
    days in Split. You want to spend 4
    days in Manchester. You want to
    spend 7 days in Riga.

Here are the cities that have direct
    flights:
Riga and Manchester, from Manchester to
    Split.

Output of Input Agent:
    {'structured_output':
    {'combinations': {'cities':
    ['Split', 'Manchester', 'Riga'],
    'direct_flights': [['Riga',
    'Manchester'], ['Manchester',
    'Riga'], ['Manchester', 'Split'],
    ['Split', 'Manchester']],
    'city_stays': {'Split': 6,
    'Manchester': 4, 'Riga': 7}},
    'constraints': {'specific_days':
    {}}}}

Error : ['Split', 'Manchester'] should
    not be one of the 'direct_flight'
    value
```

F.3 Inference Error Due to Mistakes Induced by Long Outputs

Error from Input Agent in Meeting Planning (GPT-o3)

```
Query: You are visiting San Francisco
    for the day and want to meet as many
    friends as possible. Solve the
    problem by considering various
    different schedules and picking the
    best one to optimize your goals.

Travel distances (in minutes):
    Embarcadero to The Castro: 25.
    Embarcadero to Chinatown: 7.
    Embarcadero to Sunset District: 30.
    Embarcadero to Nob Hill: 10.
    Embarcadero to Financial District:
    5. Embarcadero to Richmond District:
    21. The Castro to Embarcadero: 22.
    The Castro to Chinatown: 20. The
    Castro to Sunset District: 17. The
    Castro to Nob Hill: 16. The Castro
    to Financial District: 20. The
    Castro to Richmond District: 16.
    Chinatown to Embarcadero: 5.
    Chinatown to The Castro: 22.
    Chinatown to Sunset District: 29.
```

Chinatown to Nob Hill: 8. Chinatown to Financial District: 5. Chinatown to Richmond District: 20. Sunset District to Embarcadero: 31. Sunset District to The Castro: 17. Sunset District to Chinatown: 30. Sunset District to Nob Hill: 27. Sunset District to Financial District: 30. Sunset District to Richmond District: 12. Nob Hill to Embarcadero: 9. Nob Hill to The Castro: 17. Nob Hill to Chinatown: 6. Nob Hill to Sunset District: 25. Nob Hill to Financial District: 9. Nob Hill to Richmond District: 14. Financial District to Embarcadero: 4. Financial District to The Castro: 23. Financial District to Chinatown: 5. Financial District to Sunset District: 31. Financial District to Nob Hill: 8. Financial District to Richmond District: 21. Richmond District to Embarcadero: 19. Richmond District to The Castro: 16. Richmond District to Chinatown: 20. Richmond District to Sunset District: 11. Richmond District to Nob Hill: 17. Richmond District to Financial District: 22. CONSTRAINTS: You arrive at Embarcadero at 9:00AM. Daniel will be at The Castro from 1:45PM to 5:30PM. You'd like to meet Daniel for a minimum of 120 minutes. Mark will be at Chinatown from 4:45PM to 8:30PM. You'd like to meet Mark for a minimum of 105 minutes. Jeffrey will be at Sunset District from 5:45PM to 8:00PM. You'd like to meet Jeffrey for a minimum of 45 minutes. John will be at Nob Hill from 10:30AM to 3:15PM. You'd like to meet John for a minimum of 90 minutes. David will be at Financial District from 10:30AM to 5:30PM. You'd like to meet David for a minimum of 75 minutes. Robert will be at Richmond District from 2:30PM to 9:00PM. You'd like to meet Robert for a minimum of 105 minutes.


```
{'combinations': {'origin': {'place':
    'Embarcadero', 'time_24h': '09:00'},
    'friends': [{'friend_name':
    'Daniel', 'meet_place': 'The
    Castro', 'available_from': '13:45',
    'available_to': '17:30',
    'min_minutes': 120}, {'friend_name':
    'Mark', 'meet_place': 'Chinatown',
    'available_from': '16:45',
    'available_to': '20:30',
    'min_minutes': 105}, {'friend_name':
    'Jeffrey', 'meet_place': 'Sunset
    District', 'available_from':
    '17:45', 'available_to': '20:00',
    'min_minutes': 45}, {'friend_name':
    'John', 'meet_place': 'Nob Hill',
    'available_from': '10:30',
    'available_to': '15:15',
    'min_minutes': 90}, {'friend_name':
    'David', 'meet_place': 'Financial
    District', 'available_from':
    '10:30', 'available_to': '17:30',
    'min_minutes': 75}, {'friend_name':
    'Robert', 'meet_place': 'Richmond
    District', 'available_from':
    '14:30', 'available_to': '21:00',
    'min_minutes': 105}],
    'travel_minutes': {'Embarcadero':
    {'The Castro': 22, 'Chinatown': 5,
    'Sunset District': 30, 'Nob Hill':
    9, 'Financial District': 4,
    'Richmond District': 19}, 'The
    Castro': {'Embarcadero': 22,
    'Chinatown': 20, 'Sunset District':
    17, 'Nob Hill': 16, 'Financial
    District': 20, 'Richmond District':
    16}, 'Chinatown': {'Embarcadero': 5,
    'The Castro': 20, 'Sunset District':
    29, 'Nob Hill': 6, 'Financial
    District': 5, 'Richmond District':
    20}, 'Sunset District':
    {'Embarcadero': 30, 'The Castro':
    17, 'Chinatown': 29, 'Nob Hill': 25,
    'Financial District': 30, 'Richmond
    District': 11}, 'Nob Hill':
    {'Embarcadero': 9, 'The Castro': 16,
    'Chinatown': 6, 'Sunset District':
    25, 'Financial District': 8,
    'Richmond District': 14}, 'Financial
    District': {'Embarcadero': 4, 'The
    Castro': 20, 'Chinatown': 5, 'Sunset
    District': 30, 'Nob Hill': 8,
    'Richmond District': 21}, 'Richmond
    District': {'Embarcadero': 19, 'The
    Castro': 16, 'Chinatown': 20,
    'Sunset District': 11, 'Nob Hill':
    14, 'Financial District': 21}}},
    'constraints': {'objective':
    'maximise_number_of_friends_met',
    'origin': {'place': 'Embarcadero',
    'time_24h': '09:00'}, 'friends':
    [{'friend_name': 'Daniel',
    'meet_place': 'The Castro',
    'available_from': '13:45',
    'available_to': '17:30',
    'min_minutes': 120}, {'friend_name':
    'Mark', 'meet_place': 'Chinatown',
    'available_from': '16:45',
    'available_to': '20:30',
    'min_minutes': 105}, {'friend_name':
    'Jeffrey', 'meet_place': 'Sunset
    District', 'available_from':
    '17:45', 'available_to': '20:00',
    'min_minutes': 45}, {'friend_name':
    'John', 'meet_place': 'Nob Hill',
    'available_from': '10:30',
    'available_to': '15:15',
    'min_minutes': 90}, {'friend_name':
    'David', 'meet_place': 'Financial
    District', 'available_from':
    '10:30', 'available_to': '17:30',
    'min_minutes': 75}, {'friend_name':
    'Robert', 'meet_place': 'Richmond
    District', 'available_from':
    '14:30', 'available_to': '21:00',
```

```
    'min_minutes': 105}],
    'travel_minutes': {'Embarcadero':
    {'The Castro': 22, 'Chinatown': 5,
    'Sunset District': 30, 'Nob Hill':
    9, 'Financial District': 4,
    'Richmond District': 19}, 'The
    Castro': {'Embarcadero': 22,
    'Chinatown': 20, 'Sunset District':
    17, 'Nob Hill': 16, 'Financial
    District': 20, 'Richmond District':
    16}, 'Chinatown': {'Embarcadero': 5,
    'The Castro': 20, 'Sunset District':
    29, 'Nob Hill': 6, 'Financial
    District': 5, 'Richmond District':
    20}, 'Sunset District':
    {'Embarcadero': 30, 'The Castro':
    17, 'Chinatown': 29, 'Nob Hill': 25,
    'Financial District': 30, 'Richmond
    District': 11}, 'Nob Hill':
    {'Embarcadero': 9, 'The Castro': 16,
    'Chinatown': 6, 'Sunset District':
    25, 'Financial District': 8,
    'Richmond District': 14}, 'Financial
    District': {'Embarcadero': 4, 'The
    Castro': 20, 'Chinatown': 5, 'Sunset
    District': 30, 'Nob Hill': 8,
    'Richmond District': 21}, 'Richmond
    District': {'Embarcadero': 19, 'The
    Castro': 16, 'Chinatown': 20,
    'Sunset District': 11, 'Nob Hill':
    14, 'Financial District': 21}}}}

Error: For the 'travel_minutes' key,
    'Embarcadero': {'The Castro': 22,
    'Chinatown': 5, 'Sunset District':
    30, 'Nob Hill': 9, 'Financial
    District': 4, 'Richmond District':
    19} is incorrect, should be
    'Embarcadero': {'The Castro': 25,
    'Chinatown': 7, 'Sunset District':
    30, 'Nob Hill': 10, 'Financial
    District': 5, 'Richmond District':
    21} instead.
```

G Prompt for Agents of SCOPE

We report the full prompts provided to Planning Agent, Optimization Agents, Solution Agent, Combination Function Generator Agent, Solution Function Generator Agent, and Deliver Function Generator Agent, including role definitions and task-specific instructions. These prompts define the structured representations and function interfaces used throughout the system.

Prompt for Planning Agent

```
You are a combinations and constraints
    planning agent.

You are given few-shot examples queries
    and structured solution
    corresponding to the query. Based on
    the few-shot examples, you have to
    define them as general constraints
    planning problems. You identify the
    ways to generate possible
    combinations of plans and how to
    filter those plans to retrieve the
    solution.
You have three agents to assist you in
    solving the problem. The three
    agents are :

1. Input Agent - The agent provides set
    of elements to generate
    permutations/combinations and
    constraints as a structured JSON.
2. Combination Function Generator Agent
    - The agent creates a generic
    function called combinations_func()
    that takes any set of elements
    provided by Input Agent to generate
    the list of possible plans.
3. Filter Function Generator Agent - The
    agent creates a generic function
    called plan_func() that takes in the
    constraints provided by Input Agent
    and any list of possible plans
    outputted by the function created by
    Combination Function Generator
    Agent, to generate the most optimal
    plan that meets the constraints.

Based on the few-shot examples queries,
    you should observe what is
    considered valid possible
    combinations, and instruct
    Combination Function Generator Agent
    to produce those valid possible
    combinations.
You should identify whether the query is
    a constraints satisfaction problem
    (CSP) or optimisation problem. If it
    is a CSP problem, you should
    instruct the Filter Function
    Generator Agent to return the plan
    if it satisfied all constraints,
    else if it is an optimisation
    problem, you should instruct Filter
    Function Generator Agent to go
    through all plans that satisfied the
    constraints return the best one. You
    can only instruct the agent to
    either return any valid plan or
    return the best plan, not both.

Based on the few-shot examples queries,
    you convert the combination
    parameters and constraints in
    few-shot examples into structured
    output in JSON.
The keys should be "combinations" and
    "constraints" and respectively.

Then, based on the strucuted output you
    wrote, your instructions to each
    agent should cover the following:
1. Input Agent - You instruct the agent
    on how to identify the set of
    elements to generate
    permutations/combinations given a
```

```
    query. You also instruct the agent
    on how to identify the constraints
    to tackle such problems. The output
    format should consist of three keys:
    "combinations", "constraints" and
    "solutions", similar to the
    structured output that you wrote.
2. Combination Function Generator Agent
    - The "combinations" from the output
    of Input Agent will be the
    combinations_func() input. Hence,
    based on the output format of
    "combinations", you instruct the
    agent on creating a function on how
    to use the keys to generate possible
    plans.  The function is expected to
    output list of plans with each
    plan's format is similar to the
    few-shot structured solution. This
    output will be given to plan_func().
3. Filter Function Generator Agent - The
    "constraints" from the output of
    Input Agent and output of
    combinations_func() will be the
    plan_func() input. Based on the
    output format of "constraints" and
    "plan", as well as the type of
    problem, you instruct the agent on
    creating a function on how to use
    the keys from "constraints" to
    filter the plans from "plan". The
    function is expected to output a
    plan that matches the format of
    few-shot structured solution.

Before providing the structured output,
    you must output Chain-of-Thought
    (CoT):
1. First, you must analyse the few-shot
    examples structured solution. For
    each key in the solution, you must
    mention what information are
    essential in point form. All of the
    information that you listed must be
    given to "combinations" in your
    structured output to generate plans
    that obey the format of structured
    solution.
2. Then, you must analyse what are the
    information that could help filter
    out all the combination of plans.
    These information are essential to
    be given to "constraints" to obtain
    plan that meet the constraints.

Your output format is as below:

<start_of_COT>
Informations that are essential for
    generating combinations :
<informations in point form>

Constraints that are essential for
    filtering plans :
<informations in point form>
<end_of_COT>

<start_of_structured_output>
{
    "combinations": <The set of
    parameters that generate
    permutations/combinations>,
    "constraints": <The set of
    constraints>,
    "combinations_description": <Field
    description of the value in
    'combinations' key. Please describe
    the keys and values in detail.
    Please ensure it is written in
    string and not dict.>,
    "constraints_description": <Field
    description of 'constraints' key.
    Please describe the keys and values
    in detail. Please ensure it is
    written in string and not dict.>,
}
<end_of_structured_output>

<start_of_planning>
{
    "Input Agent": <Your instructions
    for Input Agent>,
    "Combination Function Generator
    Agent": <Your instructions for
    Combination Function Generator
    Agent>,
    "Filter Function Generator Agent":
    <Your instructions for Solutions
    Function Generator Agent>
}
<end_of_planning>

All of your instructions must be as
    general as possible. Please ensure
    your JSON is in correct format and
    able to be parsed. Do not use "..."
    to indicate more items.

Few-Shot Examples Query:
<few_shot_examples_query>

Few-Shot Examples Structured Solution:
<few_shot_examples_solution>

Few-Shot Examples Structured Solution
    Description:
<few_shot_examples_solution_description>
```

Prompt for Optimization Agent (Filter Unnecessary Parameters)

```
You are a Optimization Agent. Your job
    is to filter out unnecessary
    parameters of the output of Planning
    Agent.

You will be given the output of Planning
    Agent. There are two outputs of
    Planning Agent:
1. Structured output - The agent will
    provide the combinations and
    constraints based on the few-shot
    examples, as well as their
    corresponding field description.
```

```
2. Planning instructions - The agent
     will write instructions to
     "Combination Function Generator
     Agent" and "Solution Function
     Generator Agent".

Here is the structured output of
     Planning Agent:
<structured_output>

Here is the planning instructions of
     Planning Agent:
<planning>

Your main job is as follows:
1. We define parameter as each key in
     the "combinations" and "constraints"
     from structured output. You are
     required to look at constraints that
     are not represented in list or JSON.
2. If the parameter has no
     discriminative power, the parameter
     has to be removed. We define the
     parameter with no discriminative
     power the constraint will either
     cause all plans to pass or fail. On
     the other hand, parameter that can
     cause one plan to fail and other
     plan to not fail have discriminative
     power.
3. For example, given the problems
     produces plan A, B and C, where sum
     of X for plan A, B and C have the
     same value, due to plan A,B and C
     have constant set of items, hence
     the parameter corresponding to the
     sum of X should be removed, because
     it will cause either all plans to
     pass or fail.
4. If the parameter should be removed,
     you should remove it in the
     structure output.
5. Hence, we define the parameter with
     no discriminative power based on the
     criteria : The parameter applies
     uniformly across all of the plans.

You MUST consider each key from BOTH
     "combinations" and "constraints" of
     the structured output as your
     parameters.
Please ensure your JSON is in correct
     format and able to be parsed. Do not
     use "..." to indicate more items.

Your output format is as follows:
<start_of_COT>
1. <Parameter Type 1> - <Justify whether
     the parameter applies/does not apply
     uniformly across all of the plans>.
     <Mention if the parameter has/does
     not have discriminative power>.
     Hence, it should be <removed/not
     removed>.
...
N. <Parameter Type N> - <Justify whether
     the parameter applies/does not apply
     uniformly across all of the plans.
     <Mention if the parameter has/does
     not have discriminative power>.
     Hence, it should be <removed/not
     removed>.
<end_of_COT>

<start_of_structured_output>
<Your structured output. The ignored
     parameters should be removed from
     the structured output>
<end_of_structured_output>

<start_of_planning>
<your planning>
<end_of_planning>
```

Prompt for Optimization Agent (Fixes Valid Combinations Mistaken for Constraints)

```
You are a Optimization Agent. Your job
     is to observe the validity of output
     of the Planning Agent.

You will be given the output of Planning
     Agent. There are two outputs of
     Planning Agent:
1. Structured output - The agent will
     provide the combinations and
     constraints based on the few-shot
     examples, as well as their
     corresponding field description.
2. Planning instructions - The agent
     will write instructions to
     "Combination Function Generator
     Agent" and "Solution Function
     Generator Agent".

Here is the structured output of
     Planning Agent:
<structured_output>

Here is the planning instructions of
     Planning Agent:
<planning>

Your main job is as follows:
1. You are only required to look at the
     constraints in the "constraints" of
     structured output, not
     "combinations".
2. Even if the constraint only applies
     to each item but not all items, you
     must also see if similar type of
     constraint applies to other items.
     You must properly check if the same
     type of constraints exists for every
     item in the combination space.
3. For example, given the combination
     space has item A, B and C, and the
     given type of constraint has 3
     constraints, where each constraint
     is applied to A, B and C, then the
     same type of constraint exists for
     every item in the combination space.
4. If true, then that constraint is a
     combination parameter, which should
```

```
    be moved to "combinations" in
    structured output.

Please ensure your JSON is in correct
    format and able to be parsed. Do not
    use "..." to indicate more items.

Your output format is as follows:
<start_of_COT>
1. <Constraint Type 1> - <Justify
    whether the type of constraint
    exists/does not exist in every item
    in the combination space>. Hence, it
    is <combination
    parameters/constraint>.
...
N. <Constraint Type N> - <Justify
    whether the type of constraint
    exists/does not exist in every item
    in the combination space>. Hence, it
    is <combination
    parameters/constraint>.
<end_of_COT>

<start_of_structured_output>
<Your structured output. The constraints
    that are combination parameter
    should be moved to "combinations">
<end_of_structured_output>

<start_of_planning>
<your planning>
<end_of_planning>
```

Prompt for Optimization Agent (Validates and Expands Parameters)

```
You are a Optimization Agent. Your job
    is to observe the validity of output
    of the Planning Agent.

You will be given the output of Planning
    Agent. There are two outputs of
    Planning Agent:
1. Structured output - The agent will
    provide the combinations,
    constraints and structured solution
    based on the few-shot examples, as
    well as their corresponding field
    description.
2. Planning instructions - The agent
    will write instructions to
    "Combination Function Generator
    Agent" and "Solution Function
    Generator Agent".

Here is the list of type of constraints
    of Planning Agent:
<constraints>

Here is the structured output of
    Planning Agent:
<structured_output>

Here is the planning instructions of
    Planning Agent:
<planning>

Your main job is as follows:
1. We define parameter as each key in
    the "combinations" and "constraints"
    from structured output. For each
    type of parameter, you should check
    if the type of parameter is
    expandable. Check if there are any
    that suggest other parameters should
    also hold, even if they are not
    explicitly listed. Rewrite each type
    of parameters if you think it can be
    expanded.
2. For example, if pair [A,B] exists,
    then [B,A] might exist, despite not
    explicitly stated. Hence, such
    parameter can be expanded. You
    output Chain-of-Thought to explain
    your thought process of checking the
    claim of each parameter and explain
    whether the parameter is expandable.
3. You should assume the parameter can
    be expanded unless there are rules
    that explicitly stated such
    expansion cannot be done.

You MUST consider each key from BOTH
    "combinations" and "constraints" of
    the structured output as the
    parameter to analyse.
Please ensure your JSON is in correct
    format and able to be parsed. Do not
    use "..." to indicate more items.

Your output format is as follows:
<start_of_COT>
1. <Parameter Type 1> - <Justify whether
    the type of parameter is
    expandable>. Hence, it is
    <expandable/non-expandable>.
...
N. <Parameter Type N> - <Justify whether
    the type of parameter is
    expandable>. Hence, it is
    <expandable/non-expandablet>.
<end_of_COT>

<start_of_structured_output>
<Your structured output. For the type of
    parameter that it is expandable, you
    should write the expanded parameter>
<end_of_structured_output>

<start_of_planning>
<Please rewrite the planning
    instructions in JSON based on the
    format of the given planning
    instructions.>
<end_of_planning>
```

Prompt for Solution Agent

```
You are a Solution Agent.

You are given a solution. Based on the
    solution, you have to create a
```

```
    structured output for the solution
    in list or JSON.
You are also required to provide the
    description of keys and values of
    the solution.
Please use as less keys as possible to
    represent the solution in structured
    output.

Your output format is as below:

<start_of_structured_output>
{
    "solutions": <The solutions>,
    "solutions_description": <Field
    description of 'solutions' key.
    Please describe the keys and values
    in detail>
}
<end_of_structured_output>

For number-based values, please do not
    use string. If the value is range
    based, please use list instead.
Please ensure your JSON is in correct
    format and able to be parsed. Please
    do not write the field description
    for "solutions_description" in
    dictionary, only string is allowed.


Solution:
<few_shot_examples_solution>
```

Prompt for Combination Function Generator Agent

```
You are the Combination Function
    Generator Agent.

You are given such instructions:
<instructions>

You have to follow the provided
    instructions to create a function
    that generate a list of possible
    combinations of plans. The input
    parameter of the function 'data',
    which is a dictionary. You should
    refer to the provided description of
    input format, example of input,
    output format and example of output
    to design your function.

The description of input format is
    follows:
<input_description>

An example of input is as follows:
<example_input>

The description of output format for
    each plan is as follows:
<output_description>

An example of a plan is as follows:
<example_output>

Please use product in Python library
    itertools to help you in your code.

The name of the function you will
    generate is "combinations_func()".
    Please do not include any example
    usage or print anything in your code.


WARNING: Only apply instructions that
    are explicitly stated. Do not assume
    fairness, balance, or symmetry
    unless specified. Your function must
    return a list, not a dict.

Before you write your function code, you
    must output Chain-of-thought (CoT),
    which you explain your
    thought-process and programming
    logic in order to match the output
    format provided to you. Do not ever
    hardcode any information. If such
    information is not provided, please
    do not use it.

Your output format is as below:
<start_of_COT>
<your COT>
<end_of_COT>

<start_of_code>
<your combinations_func() code>
<end_of_code>
```

Prompt for Combination Function Reflection Agent

```
You are a Reflection Agent. Your job is
    to observe the mistake of the output
    of the Combination Function
    Generator Agent.

The job of Combination Function
    Generator Agent is to create a
    function called combinations_func()
    that generate a list of possible
    combinations of plans. The input
    parameter of the function 'data',
    which is a dictionary.
However, the agent made a mistake in the
    function that causes the function to
    not able to generate combinations of
    plans that contain the ground-truth
    plan.

Here is the one of the output plan from
    the list of plans provided by the
    function:
<example_input>

Here is the expected ground-truth plan
    that the generated combinations of
    plans should contain:
<example_output>
```

```
Here is the Chain-of-Thought (CoT)
    thought process of the agent before
    writing the code:
<chain_of_thought>

Here is the code written by the agent:
<code>

Your main job is as follows:
1. You observe the pattern of
    ground-truth plan and the pattern of
    one of the output plan from the list
    of plans. You mention the difference
    in pattern in your CoT.
2. You should also mention in your CoT
    that how you should change the CoT
    thought process of agent to ensure
    the function generates plan with
    consistent pattern as the
    ground-truth plan.
3. You correct the CoT of the agent and
    write it in your output. Then, you
    made the corresponding correction on
    the agent's code.
4. Please do not hardcode the function
    to append the ground-truth plan to
    the solution. Please do not check
    the combinations against the
    ground-truth plan. This defeat the
    purpose of ensuring the function
    generates valid plans. Similarly
    like the original code, the final
    function should output a list, not a
    dict. Do not use wrapper like
    ```python``` when writing code. Use
    the wrapper in the provided output
    format.

Your output format is as follows:
<start_of_COT>
<your COT>
<end_of_COT>

<start_of_COT_correction>
<corrected CoT of Combination Function
    Generator Agent>
<end_of_COT_correction>

<start_of_code_correction>
<corrected code of Combination Function
    Generator Agent>
<end_of_code_correction>
```

Prompt for Filter Function Generator Agent

```
You are the Filter Function Generator
    Agent.

You are given such instructions:
<instructions>

You have to follow the provided
    instructions to create a function
    that retrieve the desired plan. The
    input parameter of the function are
    'data' and 'constraints'. 'Data'
    contains a list of plan, with each
    plan in JSON format. 'Constraints'
    contains the constraints. You should
    refer to the provided description of
    input plan format, example of input,
    description of constraints format
    and example of constraints to design
    your function. Your function will
    return one of the plans as solution.

You are given a list of plan, the
    description of input format for each
    plan is as follows:
<input_description>

An example of a plan is as follows:
<example_input>

You are also given the constraints, the
    description of constraints for is as
    follows:
<constraints_description>

An example of constraints is as follows:
<example_constraints>

The name of the function you will
    generate is "plan_func()". Please do
    not include any example usage or
    print anything in your code.

Remember, the input parameter "data" is
    a list, while the input parameter
    "constraints" is JSON.

Your output is a Chain-of-thought (CoT)
    and Python function as below:
<start_of_COT>
<your COT>
<end_of_COT>

<start_of_code>
<your plan_func() code>
<end_of_code>
```

Prompt for Filter Function Reflection Agent

```
You are a Reflection Agent. Your job is
    to observe the mistake of the output
    of the Filter Function Generator
    Agent.

The job of Filter Function Generator
    Agent is to create a function called
    plan_func() that returns the valid
    plan given a list of possible
    combinations of plans. The input
    parameter of the function are 'data'
    and 'constraints'. 'Data' contains a
    list of plan, with each plan in JSON
    format. 'Constraints' contains the
    constraints. You should refer to the
    provided description of input plan
    format, example of input,
    description of constraints format
    and example of constraints to design
    your function.
```
```

```
However, the agent made a mistake in the
    function that causes the function to
    not able to return the ground-truth
    plan from the possible combinations
    of plans.

Here is the instructions given to the
    agent:
<instructions>

Here is the expected ground-truth plan
    that the function should return:
<example_output>

Here is the plan that the function
    returns:
<function_output>

Here is the Chain-of-Thought (CoT)
    thought process of the agent before
    writing the code:
<chain_of_thought>

Here is the code written by the agent:
<code>

Your main job is as follows:
1. You observe the difference between
    ground-truth plan and the plan that
    the function returns.
2. If the function returns None, it
    indicates a constraint is handled
    incorrectly in the code that some
    plans might meet it but overlooked
    by the function. Check whether is
    there possibility that each
    constraint can be handled
    differently that some plans can meet
    such constraints.
3. If the function does not return None
    but returns a different plan than
    ground-truth, you should reflect
    whether some constraints are written
    too lenient that some plans might
    not meet the constraints but
    overlooked by the function.
4. Once you identified the constraint
    that is handled incorrectly, you
    should mention in your CoT that how
    you should change the CoT thought
    process of agent to ensure the
    function handles the constraints
    correctly.
5. You correct the CoT of the agent and
    write it in your output. Then, you
    made the corresponding correction on
    the agent's code.
6. Please do not hardcode the function
    to return the ground-truth plan to
    the solution. This defeat the
    purpose of ensuring the function
    returns the correct plan. Similarly
    like the original code, the final
    function should output a JSON. Do
    not use wrapper like ```python```
    when writing code. Use the wrapper
    in the provided output format.

Your output format is as follows:
<start_of_COT>
<your COT>
<end_of_COT>

<start_of_COT_correction>
<corrected CoT of Filter Function
    Generator Agent>
<end_of_COT_correction>

<start_of_code_correction>
<corrected code of Filter Function
    Generator Agent>
<end_of_code_correction>
```

Prompt for Deliver Function Generator Agent

```
You are the Deliver Function Generator
    Agent.

Given an input, you have to create a
    function that processes the input
    and provides output that matches the
    format of few-shot examples
    solution. The input parameter of the
    function 'data', which is a plan in
    JSON. You should refer to the
    provided description of input plan
    format and example of input to
    design your function.

You are given a plan, the description of
    a plan is as follows:
<input_description>

An example of a plan is as follows:
<example_input>

The few-shot example solution is as
    follows:
<few_shot_examples_solution>

The name of the function you will
    generate is "deliver_func()". Please
    do not include any example usage or
    print anything in your code.

Before writing your code, you must
    output Chain-of-thought (CoT):
1. You must first analyse the format of
    few-shot example solution. Does it
    contain any heading? What does the
    pattern of output look like?
2. Then, you explain how are you going
    to design the function to ensure the
    format of your function's output is
    consistent with the few-shot example
    solution. Do not miss any heading if
    the example solution contains
    heading.

Your output format is CoT and Python
    function as below:
<start_of_COT>
```

```
<your COT>
<end_of_COT>

<start_of_code>
<your deliver_func() code>
<end_of_code>
```

Prompt for Deliver Function Reflection Agent

```
You are a Reflection Agent. Your job is
    to observe the mistake of the output
    of the Deliver Function Generator
    Agent

The job of Deliver Function Generator
    Agent us to create a function that
    processes a structured input in JSON
    and provides output that matches the
    format of few-shot examples solution
    in natural language. The input
    parameter of the function 'data',
    which is a plan in JSON.
However, the agent made a mistake in
    designing the function, resulting in
    the output of function does not
    match the format of few-shot
    examples solution in natural
    language, despite the structured
    input is correct.
The name of the function you will
    generate is "deliver_func()". The
    input parameter of function is
    "data", which is structured input.

Here is the structured input given to
    the function:
<structured_input>

Here is the expected ground-truth plan
    from the few-shot examples:
<example_output>

Here is the plan that the function
    returns:
<current_output>

Here is the code written by the agent:
<code>

Your main job is as follows:
1. You observe the pattern of
    ground-truth plan and the pattern of
    plan that the function returns. You
    mention the difference in pattern in
    your CoT.
2. You should also mention in your CoT
    that how you should change the code
    to ensure the function generates
    plan with consistent pattern as the
    ground-truth plan.
3. You correct the CoT of the agent and
    write it in your output. Then, you
    made the corresponding correction on
    the agent's code.
4. Please do not hardcode the function
    to return the ground-truth plan from
    few-shot examples. Please do not
    check the output of function against
    ground-truth plan from few-shot
    examples. This defeat the purpose of
    ensuring the function generates
    plans with valid format. Similarly
    like the original code, the final
    function should be a string. Do not
    use wrapper like ```python``` when
    writing code. Use the wrapper in the
    provided output format.

You must ensure the format of your
    function's output is consistent with
    the example solution. Do not miss
    any header if the example solution
    contains header.

Your output is a Chain-of-thought (CoT)
    and Python function as below:
<start_of_COT>
<your COT>
<end_of_COT>

<start_of_code>
<your deliver_func() code>
<end_of_code>
```

Prompt for Input Agent

```
You are the Input Agent.

You are given such instructions:
<instructions>
The description of output format is as
    follows:
<output_description>

An example of query is given below:
<example_input>

An example of output is given below:
<example_output>

You will be given a query. You have to
    generate a strcutured output in JSON
    adhering to the provided
    instructions and output format. You
    should refer to the provided
    description of output format and
    example of output.


Your output format is as below:
<start_of_structured_output>
<your structured output>
<end_of_structured_output>

Query :
<query>
```

H Prompt for Baselines

We report the prompts for Tree-of-Thought (ToT) and Thought of Search (ToS), as these methods rely on prompt-level control of reasoning and search. For EvoAgent, we use the prompt provided in the official GitHub repository, with minimal modifications to adapt to different task domains. For HyperTree Planning, we directly adopt the prompt described in the original paper.

H.1 Prompts for Tree-of-Thought (ToT)

Prompt for Expansion

```
Here is an example of query and full
    solution.

Example query :
You plan to visit 10 European cities for
    21 days in total. You only take
    direct flights to commute between
    cities. You plan to stay in Vienna
    for 3 days. You want to spend 5 days
    in Frankfurt. You want to spend 2
    days in Oslo. You are going to
    attend a wedding in Oslo between day
    20 and day 21. You want to spend 3
    days in Prague. You would like to
    visit Valencia for 2 days. You want
    to meet a friend in Valencia between
    day 17 and day 18. You plan to stay
    in Dubrovnik for 2 days. You would
    like to visit Edinburgh for 5 days.
    From day 1 to day 5, there is a
    annual show you want to attend in
    Edinburgh. You plan to stay in
    London for 2 days. You plan to visit
    relatives in London between day 12
    and day 13. You would like to visit
    Munich for 3 days. You would like to
    meet your friends at Munich between
    day 18 and day 20 to tour together.
    You would like to visit Budapest for
    3 days.

Here are the cities that have direct
    flights:
Valencia and Munich, Vienna and Munich,
    Vienna and Valencia, London and
    Budapest, London and Oslo, Edinburgh
    and Budapest, Frankfurt and
    Budapest, Frankfurt and London,
    Prague and Oslo, Edinburgh and Oslo,
    Edinburgh and Munich, Prague and
    Munich, London and Prague, Edinburgh
    and London, Edinburgh and Frankfurt,
    Dubrovnik and Munich, Dubrovnik and
    Vienna, Munich and Oslo, Dubrovnik
    and Oslo, Budapest and Munich,
    Frankfurt and Prague, Vienna and
    London, Frankfurt and Vienna,
    Frankfurt and Oslo, Frankfurt and
    Munich, Vienna and Oslo, Vienna and
    Prague, Budapest and Oslo, Budapest
    and Prague, London and Valencia,
    London and Munich, Frankfurt and
    Dubrovnik, Prague and Valencia,
    Frankfurt and Valencia, Edinburgh
    and Prague, Vienna and Budapest.
Find a trip plan of visiting the cities
    for 21 days by taking direct flights
    to commute between them.

Example full solution
Here is the trip plan for visiting the
    10 European cities for 21 days:

**Day 1-5:** Arriving in Edinburgh and
    visit Edinburgh for 5 days.
**Day 5:** Fly from Edinburgh to
    Frankfurt.
**Day 5-9:** Visit Frankfurt for 5 days.
**Day 9:** Fly from Frankfurt to
    Dubrovnik.
**Day 9-10:** Visit Dubrovnik for 2 days.
**Day 10:** Fly from Dubrovnik to Vienna.
**Day 10-12:** Visit Vienna for 3 days.
**Day 12:** Fly from Vienna to London.
**Day 12-13:** Visit London for 2 days.
**Day 13:** Fly from London to Budapest.
**Day 13-15:** Visit Budapest for 3 days.
**Day 15:** Fly from Budapest to Prague.
**Day 15-17:** Visit Prague for 3 days.
**Day 17:** Fly from Prague to Valencia.
**Day 17-18:** Visit Valencia for 2 days.
**Day 18:** Fly from Valencia to Munich.
**Day 18-20:** Visit Munich for 3 days.
**Day 20:** Fly from Munich to Oslo.
**Day 20-21:** Visit Oslo for 2 days.


You are a Tree-of-Thought (ToT) Agent.
    Given a query, a current state and
    the reasoning up to that state, you
    have to provide the next step for
    the current state.
In each step, you will output you next
    city you would like to visit and
    provide reasoning.

You must provide the reasoning of
    choosing this city. In every state,
    you choose N number of most possible
    cities (value of N is your choice),
    that you would like to visit from
    that current state.
If you are at first step, format is as
    below:
**Day 1-N:** Arriving in <first city>
    and visit <first city> for N days.

If you are at second step onwards,
    format is as below:
**Day X:** Fly from <current city> to
    <next city>.
**Day X-Y:** Visit <next city> for <N>
    days.

For example, the current state is:
Here is the trip plan for visiting the
    10 European cities for 21 days:
```

```
**Day 1-5:** Arriving in Edinburgh and
    visit Edinburgh for 5 days.

Then, a possible example output of your
    next step is visiting Frankfurt:
**Day 5:** Fly from Edinburgh to
    Frankfurt.
**Day 5-9:** Visit Frankfurt for 5 days.



Your output format is a list as follows :
<start_of_output>
[
    {"next_step": <the output for your
    next step (choosing city 1)>,
    "reasoning": <your reasoning for
    choosing city 1>},
    {"next_step": <the output for your
    next step (choosing city 2)>,
    "reasoning": <your reasoning for
    choosing city 2>},
    ...
    {"next_step": <the output for your
    next step (choosing city N)>,
    "reasoning": <your reasoning for
    choosing city N>}
]
<end_of_output>

Query:
<query>

Current state:
<current_state>

Reasoning up to the state:
<reasoning_state>
```

Prompt for Evaluation

```
You are a Tree-of-Thought (ToT)
    Evaluator. Given a query, list of
    current states and reasoning for
    each state, you have to evaluate and
    provide score for each state of how
    likely each state will lead to
    correct solution.
Your score should be between 1 and 10.
    Only integer is allowed.
In your output, you must copy each state
    and reasoning exactly as the input,
    do not change any words.
You must ensure the score for each
    output are different. You are not
    allowed to provide same score for
    different output.

The current states and reasoning is
    given in such format:
[
    {"current_state": <current state>,
    "reasoning": <reasoning for the
    state>},
    {"current_state": <current state>,
    "reasoning": <reasoning for the
    state>},
    {"current_state": <current state>,
    "reasoning": <reasoning for the
    state>}
]

Your output format is a list as follows :
<start_of_output>
[
    {"current_state": <current state>,
    "reasoning": <reasoning for the
    state>, "score": <integer between 1
    and 10>},
    {"current_state": <current state>,
    "reasoning": <reasoning for the
    state>, "score": <integer between 1
    and 10>},
    {"current_state": <current state>,
    "reasoning": <reasoning for the
    state>, "score": <integer between 1
    and 10>}
]
<end_of_output>

Query:
<query>

Current states and reasoning:
<current_state>
```

H.2 Prompts for Thought of Search (ToS)

Prompt for Introduction

```
You are a task description agent. There
    is a task called <task>. You are
    given an example query and answer of
    the task. Using this query and
    answer, you must provide a detailed
    description of task.
You must also provide any assumptions or
    constraints that are not explicit in
    the query. Your assumptions and
    constraints must be general so that
    can be applied on other queries of
    <task>.
Finally, you must also mention how a
    state is represented. You should
    also describe how to transition from
    one state to another and what is the
    ending state.
Lastly, you list out the conditions
    where a state has met the goal by
    writing the list of conditions after.

Your output format is as below:
Description of <task>:
<your description of task>

Description of how to transition from
    one state to another and ending
    state:
<the description>

List of conditions where a state has met
    the goal:
1. <Condition 1>
...
N. <Condition N>

Here is an example task called 24 Game
    and here is the example output
    description :

The 24 Game is a mathematical card game
    where the objective is to manipulate
    four integers so that the final
    result equals 24. The game begins
    with a list of four numbers, and the
    player must use each number exactly
    once, applying any combination of
    addition, subtraction,
    multiplication, or division to
    obtain 24.
A state is represented as a list of
    numbers (of length four or fewer).


An example query of the task :
<task_query>

An example answer of the task :
<task_answer>

Your output :
```

Prompt for Evaluation

```
You are a function generator agent.
    There is a task called <task>.

First of all, you must define the state.
    To do that, you provide the state
    from the given example query.

Write a Python function called
    'successor_states(state)' that
    generates all valid successor states
    for the <task>.

Then, write a Python function called
    'is_goal(state)' that tests whether
    a state is a goal state.

Then, write a Python function called
    'convert_state_to_nl(state)' that
    convert the goal state to NL that
    shares the same format as the
    example answer.
Hence, you must ensure the goal state
    contain the sufficient information
    be converted to NL.

Please do not include any example usage
    or print anything in your code.
    Ensure your code is written within
    the given wrapper. Please don't
    include ```python and ``` wrapper
    in your output.
Please do not hardcode any fixed values
    in any functions that cannot be
    applied to other queries of <task>.


The task description of <task> is as
    follows :
<task_description>

An example query of the task :
<task_query>

An example answer of the task :
<task_answer>

Your output format is as follows :

State structure:
<start_of_JSON>
<your state given the example query>
<end_of_JSON>

Output code for successor_states:
<start_of_code>
<your code. Please do not hardcode any
    fixed values in any functions that
    cannot be applied to other queries.>
<end_of_code>

Output code for is_goal:
<start_of_code>
<your code. Please do not hardcode any
    fixed values in any functions that
    cannot be applied to other queries.>
<end_of_code>
```

```
Output code for convert_state_to_nl:
<start_of_code>
<your code. Remember, the function
    output must share the same format as
    the answer of the task.>
<end_of_code>
```

I Output of Agents of SCOPE

We present the outputs of each agent in the SCOPE workflow, using Trip Planning as a case study.

I.1 Output of Agents

Output of Planner Agent

```
<start_of_COT>
Informations that are essential for
    generating combinations :
- List of cities to visit.
- Total number of days for the trip.
- Direct flight connections between
    cities.

Constraints that are essential for
    filtering plans :
- Specific number of days to stay in
    each city.
- Specific days to be in certain cities
    for events or meetings.
- Total trip duration must be exactly 21
    days.
- Must only use direct flights between
    cities.
<end_of_COT>

<start_of_structured_output>
{
    "combinations": {
        "cities": ["Edinburgh",
    "Frankfurt", "Dubrovnik", "Vienna",
    "London", "Budapest", "Prague",
    "Valencia", "Munich", "Oslo"],
        "total_days": 21,
        "direct_flights": [
            ["Valencia", "Munich"],
    ["Vienna", "Munich"], ["Vienna",
    "Valencia"], ["London", "Budapest"],
            ["London", "Oslo"],
    ["Edinburgh", "Budapest"],
    ["Frankfurt", "Budapest"],
    ["Frankfurt", "London"],
            ["Prague", "Oslo"],
    ["Edinburgh", "Oslo"], ["Edinburgh",
    "Munich"], ["Prague", "Munich"],
            ["London", "Prague"],
    ["Edinburgh", "London"],
    ["Edinburgh", "Frankfurt"],
    ["Dubrovnik", "Munich"],
            ["Dubrovnik", "Vienna"],
    ["Munich", "Oslo"], ["Dubrovnik",
    "Oslo"], ["Budapest", "Munich"],
            ["Frankfurt", "Prague"],
    ["Vienna", "London"], ["Frankfurt",
    "Vienna"], ["Frankfurt", "Oslo"],
            ["Frankfurt", "Munich"],
    ["Vienna", "Oslo"], ["Vienna",
    "Prague"], ["Budapest", "Oslo"],
            ["Budapest", "Prague"],
    ["London", "Valencia"], ["London",
    "Munich"], ["Frankfurt",
    "Dubrovnik"],
            ["Prague", "Valencia"],
    ["Frankfurt", "Valencia"],
    ["Edinburgh", "Prague"], ["Vienna",
    "Budapest"]
        ]
    },
    "constraints": {
        "city_stays": {
            "Edinburgh": 5,
            "Frankfurt": 5,
            "Dubrovnik": 2,
            "Vienna": 3,
            "London": 2,
            "Budapest": 3,
            "Prague": 3,
            "Valencia": 2,
            "Munich": 3,
            "Oslo": 2
        },
        "specific_days": {
            "Edinburgh": [1, 5],
            "Oslo": [20, 21],
            "Valencia": [17, 18],
            "London": [12, 13],
            "Munich": [18, 20]
        }
    },
    "combinations_description": "The
    'combinations' key contains the
    parameters for generating possible
    travel plans. 'cities' is a list of
    cities to visit, 'total_days' is the
    total number of days for the trip,
    and 'direct_flights' is a list of
    pairs of cities that have direct
    flights between them.",
    "constraints_description": "The
    'constraints' key contains the
    conditions that must be met for a
    valid travel plan. 'city_stays'
    specifies the number of days to stay
    in each city, and 'specific_days'
    specifies the exact days certain
    cities must be visited for events or
    meetings."
}
<end_of_structured_output>

<start_of_planning>
{
    "Input Agent": "Identify the set of
    cities to visit, the total number of
    days for the trip, and the direct
    flight connections between cities.
    Also, identify the constraints such
    as the number of days to stay in
    each city and specific days to be in
    certain cities for events or
    meetings.",
    "Combination Function Generator
    Agent": "Create a function
```

```
    combinations_func() that takes
    'cities', 'total_days', and
    'direct_flights' as input. Generate
    all possible sequences of city
    visits that fit within the total
    days and only use direct flights.",
    "Solution Function Generator Agent":
    "Create a function plan_func() that
    takes 'city_stays' and
    'specific_days' as constraints,
    along with the list of possible
    plans from combinations_func().
    Filter the plans to find one that
    satisfies all constraints, ensuring
    the total trip duration is exactly
    21 days and all specific day
    requirements are met."
}
<end_of_planning>
```

Output of Optimization Agent (Filter Unnecessary Parameters)

```
<start_of_COT>
1. cities - The parameter does not apply
    uniformly across all of the plans as
    different plans can include
    different subsets of cities. The
    parameter has discriminative power.
    Hence, it should not be removed.

2. total_days - The parameter applies
    uniformly across all of the plans as
    all plans must fit within the total
    of 21 days. The parameter does not
    have discriminative power. Hence, it
    should be removed.

3. direct_flights - The parameter does
    not apply uniformly across all of
    the plans as different plans can use
    different direct flights. The
    parameter has discriminative power.
    Hence, it should not be removed.

4. city_stays - The parameter does not
    apply uniformly across all of the
    plans as different plans can have
    different city stays. The parameter
    has discriminative power. Hence, it
    should not be removed.

5. specific_days - The parameter does
    not apply uniformly across all of
    the plans as different plans can
    meet or not meet specific day
    requirements. The parameter has
    discriminative power. Hence, it
    should not be removed.
<end_of_COT>

<start_of_structured_output>
{
    "combinations": {
        "cities": [
            "Edinburgh",
            "Frankfurt",
            "Dubrovnik",
            "Vienna",
            "London",
            "Budapest",
            "Prague",
            "Valencia",
            "Munich",
            "Oslo"
        ],
        "direct_flights": [
            [
                "Valencia",
                "Munich"
            ],
            [
                "Vienna",
                "Munich"
            ],
            [
                "Vienna",
                "Valencia"
            ],
            [
                "London",
                "Budapest"
            ],
            [
                "London",
                "Oslo"
            ],
            [
                "Edinburgh",
                "Budapest"
            ],
            [
                "Frankfurt",
                "Budapest"
            ],
            [
                "Frankfurt",
                "London"
            ],
            [
                "Prague",
                "Oslo"
            ],
            [
                "Edinburgh",
                "Oslo"
            ],
            [
                "Edinburgh",
                "Munich"
            ],
            [
                "Prague",
                "Munich"
            ],
            [
                "London",
                "Prague"
            ],
            [
                "Edinburgh",
                "London"
            ],
            [
```

```
                "Edinburgh",
                "Frankfurt"
            ],
            [
                "Dubrovnik",
                "Munich"
            ],
            [
                "Dubrovnik",
                "Vienna"
            ],
            [
                "Munich",
                "Oslo"
            ],
            [
                "Dubrovnik",
                "Oslo"
            ],
            [
                "Budapest",
                "Munich"
            ],
            [
                "Frankfurt",
                "Prague"
            ],
            [
                "Vienna",
                "London"
            ],
            [
                "Frankfurt",
                "Vienna"
            ],
            [
                "Frankfurt",
                "Oslo"
            ],
            [
                "Frankfurt",
                "Munich"
            ],
            [
                "Vienna",
                "Oslo"
            ],
            [
                "Vienna",
                "Prague"
            ],
            [
                "Budapest",
                "Oslo"
            ],
            [
                "Budapest",
                "Prague"
            ],
            [
                "London",
                "Valencia"
            ],
            [
                "London",
                "Munich"
            ],
            [
                "Frankfurt",
                "Dubrovnik"
            ],
            [
                "Prague",
                "Valencia"
            ],
            [
                "Frankfurt",
                "Valencia"
            ],
            [
                "Edinburgh",
                "Prague"
            ],
            [
                "Vienna",
                "Budapest"
            ]
        ]
    },
    "constraints": {
        "city_stays": {
            "Edinburgh": 5,
            "Frankfurt": 5,
            "Dubrovnik": 2,
            "Vienna": 3,
            "London": 2,
            "Budapest": 3,
            "Prague": 3,
            "Valencia": 2,
            "Munich": 3,
            "Oslo": 2
        },
        "specific_days": {
            "Edinburgh": [
                1,
                5
            ],
            "Oslo": [
                20,
                21
            ],
            "Valencia": [
                17,
                18
            ],
            "London": [
                12,
                13
            ],
            "Munich": [
                18,
                20
            ]
        }
    },
    "combinations_description": "The
    'combinations' key contains the
    parameters for generating possible
    travel plans. 'cities' is a list of
    cities to visit, and
    'direct_flights' is a list of pairs
    of cities that have direct flights
    between them.",
    "constraints_description": "The
    'constraints' key contains the
    conditions that must be met for a
```

```
    valid travel plan. 'city_stays'
    specifies the number of days to stay
    in each city, and 'specific_days'
    specifies the exact days certain
    cities must be visited for events or
    meetings."
}
<end_of_structured_output>

<start_of_planning>
{
    "Input Agent": "Identify the set of
    cities to visit and the direct
    flight connections between cities.
    Also, identify the constraints such
    as the number of days to stay in
    each city and specific days to be in
    certain cities for events or
    meetings.",
    "Combination Function Generator
    Agent": "Create a function
    combinations_func() that takes
    'cities' and 'direct_flights' as
    input. Generate all possible
    sequences of city visits that fit
    within the total days and only use
    direct flights.",
    "Solution Function Generator Agent":
    "Create a function plan_func() that
    takes 'city_stays' and
    'specific_days' as constraints,
    along with the list of possible
    plans from combinations_func().
    Filter the plans to find one that
    satisfies all constraints, ensuring
    all specific day requirements are
    met."
}
<end_of_planning>
```

Output of Optimization Agent (Fixes Valid Combinations Mistaken for Constraints)

```
<start_of_COT>
1. City Stays - The 'city_stays'
    constraint specifies the number of
    days to stay in each city. This
    constraint exists for every city in
    the combination space. Hence, it is
    a combination parameter.
2. Specific Days - The 'specific_days'
    constraint specifies exact days for
    certain cities but not for all
    cities in the combination space.
    Hence, it is a constraint.
<end_of_COT>

<start_of_structured_output>
{
    "combinations": {
        "cities": [
            "Edinburgh",
            "Frankfurt",
            "Dubrovnik",
            "Vienna",
            "London",
            "Budapest",
            "Prague",
            "Valencia",
            "Munich",
            "Oslo"
        ],
        "direct_flights": [
            [
                "Valencia",
                "Munich"
            ],
            [
                "Vienna",
                "Munich"
            ],
            [
                "Vienna",
                "Valencia"
            ],
            [
                "London",
                "Budapest"
            ],
            [
                "London",
                "Oslo"
            ],
            [
                "Edinburgh",
                "Budapest"
            ],
            [
                "Frankfurt",
                "Budapest"
            ],
            [
                "Frankfurt",
                "London"
            ],
            [
                "Prague",
                "Oslo"
            ],
            [
                "Edinburgh",
                "Oslo"
            ],
            [
                "Edinburgh",
                "Munich"
            ],
            [
                "Prague",
                "Munich"
            ],
            [
                "London",
                "Prague"
            ],
            [
                "Edinburgh",
                "London"
            ],
            [
                "Edinburgh",
                "Frankfurt"
            ],
            [
```

```
                "Dubrovnik",
                "Munich"
            ],
            [
                "Dubrovnik",
                "Vienna"
            ],
            [
                "Munich",
                "Oslo"
            ],
            [
                "Dubrovnik",
                "Oslo"
            ],
            [
                "Budapest",
                "Munich"
            ],
            [
                "Frankfurt",
                "Prague"
            ],
            [
                "Vienna",
                "London"
            ],
            [
                "Frankfurt",
                "Vienna"
            ],
            [
                "Frankfurt",
                "Oslo"
            ],
            [
                "Frankfurt",
                "Munich"
            ],
            [
                "Vienna",
                "Oslo"
            ],
            [
                "Vienna",
                "Prague"
            ],
            [
                "Budapest",
                "Oslo"
            ],
            [
                "Budapest",
                "Prague"
            ],
            [
                "London",
                "Valencia"
            ],
            [
                "London",
                "Munich"
            ],
            [
                "Frankfurt",
                "Dubrovnik"
            ],
            [
```



```
                "Prague",
                "Valencia"
            ],
            [
                "Frankfurt",
                "Valencia"
            ],
            [
                "Edinburgh",
                "Prague"
            ],
            [
                "Vienna",
                "Budapest"
            ]
        ],
        "city_stays": {
            "Edinburgh": 5,
            "Frankfurt": 5,
            "Dubrovnik": 2,
            "Vienna": 3,
            "London": 2,
            "Budapest": 3,
            "Prague": 3,
            "Valencia": 2,
            "Munich": 3,
            "Oslo": 2
        }
    },
    "constraints": {
        "specific_days": {
            "Edinburgh": [
                1,
                5
            ],
            "Oslo": [
                20,
                21
            ],
            "Valencia": [
                17,
                18
            ],
            "London": [
                12,
                13
            ],
            "Munich": [
                18,
                20
            ]
        }
    },
    "combinations_description": "The
    'combinations' key contains the
    parameters for generating possible
    travel plans. 'cities' is a list of
    cities to visit, 'direct_flights' is
    a list of pairs of cities that have
    direct flights between them, and
    'city_stays' specifies the number of
    days to stay in each city.",
    "constraints_description": "The
    'constraints' key contains the
    conditions that must be met for a
    valid travel plan. 'specific_days'
    specifies the exact days certain
    cities must be visited for events or
```

```
    meetings."
}
<end_of_structured_output>

<start_of_planning>
{
    "Input Agent": "Identify the set of
    cities to visit and the direct
    flight connections between cities.
    Also, identify the constraints such
    as specific days to be in certain
    cities for events or meetings.",
    "Combination Function Generator
    Agent": "Create a function
    combinations_func() that takes
    'cities', 'direct_flights', and
    'city_stays' as input. Generate all
    possible sequences of city visits
    that fit within the total days and
    only use direct flights.",
    "Solution Function Generator Agent":
    "Create a function plan_func() that
    takes 'specific_days' as
    constraints, along with the list of
    possible plans from
    combinations_func(). Filter the
    plans to find one that satisfies all
    constraints, ensuring all specific
    day requirements are met."
}
<end_of_planning>
```

Output of Optimization Agent (Validates and Expands Parameters)

```
<start_of_COT>

1. Cities - The list of cities is a
    fixed set and does not suggest any
    additional cities should be
    included. Hence, it is
    non-expandable.

2. Direct Flights - If a direct flight
    exists from City A to City B, it is
    reasonable to assume a direct flight
    could exist from City B to City A
    unless explicitly stated otherwise.
    Hence, it is expandable.

3. City Stays - The number of days to
    stay in each city is explicitly
    defined and does not suggest
    additional days should be added.
    Hence, it is non-expandable.

4. Specific Days - The specific days for
    visiting certain cities are
    explicitly defined for events or
    meetings. There is no indication
    that additional days should be
    added. Hence, it is non-expandable.

<end_of_COT>

<start_of_structured_output>
```

```
{
    "combinations": {
        "cities": [
            "Edinburgh",
            "Frankfurt",
            "Dubrovnik",
            "Vienna",
            "London",
            "Budapest",
            "Prague",
            "Valencia",
            "Munich",
            "Oslo"
        ],
        "direct_flights": [
            [
                "Valencia",
                "Munich"
            ],
            [
                "Munich",
                "Valencia"
            ],
            [
                "Vienna",
                "Munich"
            ],
            [
                "Munich",
                "Vienna"
            ],
            [
                "Vienna",
                "Valencia"
            ],
            [
                "Valencia",
                "Vienna"
            ],
            [
                "London",
                "Budapest"
            ],
            [
                "Budapest",
                "London"
            ],
            [
                "London",
                "Oslo"
            ],
            [
                "Oslo",
                "London"
            ],
            [
                "Edinburgh",
                "Budapest"
            ],
            [
                "Budapest",
                "Edinburgh"
            ],
            [
                "Frankfurt",
                "Budapest"
            ],
            [
```

```
            "Budapest",
            "Frankfurt"
        ],
        [
            "Frankfurt",
            "London"
        ],
        [
            "London",
            "Frankfurt"
        ],
        [
            "Prague",
            "Oslo"
        ],
        [
            "Oslo",
            "Prague"
        ],
        [
            "Edinburgh",
            "Oslo"
        ],
        [
            "Oslo",
            "Edinburgh"
        ],
        [
            "Edinburgh",
            "Munich"
        ],
        [
            "Munich",
            "Edinburgh"
        ],
        [
            "Prague",
            "Munich"
        ],
        [
            "Munich",
            "Prague"
        ],
        [
            "London",
            "Prague"
        ],
        [
            "Prague",
            "London"
        ],
        [
            "Edinburgh",
            "London"
        ],
        [
            "London",
            "Edinburgh"
        ],
        [
            "Edinburgh",
            "Frankfurt"
        ],
        [
            "Frankfurt",
            "Edinburgh"
        ],
        [
```



```
            "Dubrovnik",
            "Munich"
        ],
        [
            "Munich",
            "Dubrovnik"
        ],
        [
            "Dubrovnik",
            "Vienna"
        ],
        [
            "Vienna",
            "Dubrovnik"
        ],
        [
            "Munich",
            "Oslo"
        ],
        [
            "Oslo",
            "Munich"
        ],
        [
            "Dubrovnik",
            "Oslo"
        ],
        [
            "Oslo",
            "Dubrovnik"
        ],
        [
            "Budapest",
            "Munich"
        ],
        [
            "Munich",
            "Budapest"
        ],
        [
            "Frankfurt",
            "Prague"
        ],
        [
            "Prague",
            "Frankfurt"
        ],
        [
            "Vienna",
            "London"
        ],
        [
            "London",
            "Vienna"
        ],
        [
            "Frankfurt",
            "Vienna"
        ],
        [
            "Vienna",
            "Frankfurt"
        ],
        [
            "Frankfurt",
            "Oslo"
        ],
        [
```

```
                "Oslo",
                "Frankfurt"
            ],
            [
                "Frankfurt",
                "Munich"
            ],
            [
                "Munich",
                "Frankfurt"
            ],
            [
                "Vienna",
                "Oslo"
            ],
            [
                "Oslo",
                "Vienna"
            ],
            [
                "Vienna",
                "Prague"
            ],
            [
                "Prague",
                "Vienna"
            ],
            [
                "Budapest",
                "Oslo"
            ],
            [
                "Oslo",
                "Budapest"
            ],
            [
                "Budapest",
                "Prague"
            ],
            [
                "Prague",
                "Budapest"
            ],
            [
                "London",
                "Valencia"
            ],
            [
                "Valencia",
                "London"
            ],
            [
                "London",
                "Munich"
            ],
            [
                "Munich",
                "London"
            ],
            [
                "Frankfurt",
                "Dubrovnik"
            ],
            [
                "Dubrovnik",
                "Frankfurt"
            ],
            [
                "Prague",
                "Valencia"
            ],
            [
                "Valencia",
                "Prague"
            ],
            [
                "Frankfurt",
                "Valencia"
            ],
            [
                "Valencia",
                "Frankfurt"
            ],
            [
                "Edinburgh",
                "Prague"
            ],
            [
                "Prague",
                "Edinburgh"
            ],
            [
                "Vienna",
                "Budapest"
            ],
            [
                "Budapest",
                "Vienna"
            ]
        ],
        "city_stays": {
            "Edinburgh": 5,
            "Frankfurt": 5,
            "Dubrovnik": 2,
            "Vienna": 3,
            "London": 2,
            "Budapest": 3,
            "Prague": 3,
            "Valencia": 2,
            "Munich": 3,
            "Oslo": 2
        }
    },
    "constraints": {
        "specific_days": {
            "Edinburgh": [
                1,
                5
            ],
            "Oslo": [
                20,
                21
            ],
            "Valencia": [
                17,
                18
            ],
            "London": [
                12,
                13
            ],
            "Munich": [
                18,
                20
            ]
        }
```

```
    },
    "combinations_description": "The
    'combinations' key contains the
    parameters for generating possible
    travel plans. 'cities' is a list of
    cities to visit, 'direct_flights' is
    a list of pairs of cities that have
    direct flights between them, and
    'city_stays' specifies the number of
    days to stay in each city.",
    "constraints_description": "The
    'constraints' key contains the
    conditions that must be met for a
    valid travel plan. 'specific_days'
    specifies the exact days certain
    cities must be visited for events or
    meetings."
}
<end_of_structured_output>

<start_of_planning>
{
    "Input Agent": "Identify the set of
    cities to visit and the direct
    flight connections between cities.
    Also, identify the constraints such
    as specific days to be in certain
    cities for events or meetings.",
    "Combination Function Generator
    Agent": "Create a function
    combinations_func() that takes
    'cities', 'direct_flights', and
    'city_stays' as input. Generate all
    possible sequences of city visits
    that fit within the total days and
    only use direct flights, considering
    both directions for each flight.",
    "Solution Function Generator Agent":
    "Create a function plan_func() that
    takes 'specific_days' as
    constraints, along with the list of
    possible plans from
    combinations_func(). Filter the
    plans to find one that satisfies all
    constraints, ensuring all specific
    day requirements are met."
}
<end_of_planning>
```

Output of Solution Agent

```
<start_of_structured_output>
{
    "solutions": [
        {"city": "Edinburgh", "days":
    [1, 2, 3, 4, 5]},
        {"city": "Frankfurt", "days":
    [5, 6, 7, 8, 9]},
        {"city": "Dubrovnik", "days":
    [9, 10]},
        {"city": "Vienna", "days": [10,
    11, 12]},
        {"city": "London", "days": [12,
    13]},
        {"city": "Budapest", "days":
    [13, 14, 15]},
        {"city": "Prague", "days": [15,
    16, 17]},
        {"city": "Valencia", "days":
    [17, 18]},
        {"city": "Munich", "days": [18,
    19, 20]},
        {"city": "Oslo", "days": [20,
    21]}
    ],
    "solutions_description": "The
    'solutions' key contains a list of
    dictionaries, each representing a
    city visit during the trip. Each
    dictionary has two keys: 'city' and
    'days'. The 'city' key holds the
    name of the city being visited, and
    the 'days' key contains a list of
    integers representing the days of
    the trip spent in that city."
}
<end_of_structured_output>
```

Output of Combination Function Generation Agent (Requires Reflection)

```
<start_of_COT>
To solve this problem, we need to
    generate all possible sequences of
    city visits that fit within the
    total days and only use direct
    flights. Here's the step-by-step
    thought process:

1. **Input Parsing**: We will extract
    the list of cities, direct flights,
    and city stays from the input
    dictionary.

2. **Graph Representation**: We will
    represent the direct flights as a
    graph where each city is a node and
    each direct flight is an edge. This
    will help us in determining valid
    paths.

3. **Permutations and Combinations**: We
    will use permutations to generate
    possible sequences of city visits.
    However, we need to ensure that each
    sequence is valid in terms of direct
    flights and total days.

4. **Validation of Sequences**:
   - For each permutation of cities, we
    will check if there is a direct
    flight between consecutive cities.
   - We will also calculate the total
    days required for each sequence by
    summing up the days required for
    each city stay.
   - If the total days exceed the
    allowed days, the sequence is
    invalid.

5. **Output Format**: For each valid
    sequence, we will create a list of
```

```
     dictionaries where each dictionary
     contains the city name and the days
     spent in that city. The days will be
     represented as a list of integers.

6. **Return the Result**: Finally, we
     will return a list of all valid
     sequences in the specified format.

By following these steps, we ensure that
     the function generates all possible
     valid travel plans according to the
     given constraints.
<end_of_COT>

<start_of_code>
from itertools import permutations

def combinations_func(data):
    cities = data['cities']
    direct_flights =
     data['direct_flights']
    city_stays = data['city_stays']

    # Create a set of direct flights for
     quick lookup
    flight_set = set(tuple(flight) for
     flight in direct_flights)

    # Calculate total days available
    total_days = sum(city_stays.values())

    # List to store all valid plans
    valid_plans = []

    # Generate all permutations of cities
    for perm in permutations(cities):
        current_day = 1
        valid = True
        plan = []

        for i in range(len(perm)):
            city = perm[i]
            stay_days = city_stays[city]

            # Check if the current city
    can be reached from the previous city
            if i > 0:
                prev_city = perm[i - 1]
                if (prev_city, city) not
    in flight_set:
                    valid = False
                    break

            # Add the city and its stay
    days to the plan
            plan.append({
                "city": city,
                "days":
    list(range(current_day, current_day
    + stay_days))
            })

            # Update the current day
            current_day += stay_days

            # Check if the total days
    exceed the allowed days
            if current_day > total_days
    + 1:
                valid = False
                break

        # If the plan is valid, add it
    to the list of valid plans
        if valid:
            valid_plans.append(plan)

    return valid_plans
<end_of_code>
```

Output of Combination Function Reflection Agent

```
<start_of_COT>
The main difference between the
     ground-truth plan and the output
     plan is the overlap of days between
     consecutive cities. In the
     ground-truth plan, the last day of a
     city overlaps with the first day of
     the next city. For example,
     Edinburgh ends on day 5, and
     Frankfurt starts on day 5. In the
     output plan, there is no overlap;
     each city starts the day after the
     previous city ends.

To ensure the function generates plans
     with the same pattern as the
     ground-truth plan, the thought
     process should be adjusted to allow
     for overlapping days between
     consecutive cities. This means that
     when calculating the days for each
     city, the start day of the next city
     should be the same as the end day of
     the previous city.

Additionally, the total days calculation
     should be adjusted to account for
     this overlap, ensuring that the
     total number of days does not exceed
     the available days.
<end_of_COT>

<start_of_COT_correction>
1. **Input Parsing**: Extract the list
     of cities, direct flights, and city
     stays from the input dictionary.

2. **Graph Representation**: Represent
     the direct flights as a graph where
     each city is a node and each direct
     flight is an edge. This will help in
     determining valid paths.

3. **Permutations and Combinations**:
     Use permutations to generate
     possible sequences of city visits.
     Ensure that each sequence is valid
     in terms of direct flights and total
     days.
```

```
4. **Validation of Sequences**:
   - For each permutation of cities,
     check if there is a direct flight
     between consecutive cities.
   - Calculate the total days required
     for each sequence by summing up the
     days required for each city stay,
     allowing for overlap between
     consecutive cities.
   - If the total days exceed the
     allowed days, the sequence is
     invalid.

5. **Output Format**: For each valid
     sequence, create a list of
     dictionaries where each dictionary
     contains the city name and the days
     spent in that city. The days will be
     represented as a list of integers,
     with overlap between consecutive
     cities.

6. **Return the Result**: Return a list
     of all valid sequences in the
     specified format.
<end_of_COT_correction>

<start_of_code_correction>
from itertools import permutations

def combinations_func(data):
    cities = data['cities']
    direct_flights =
     data['direct_flights']
    city_stays = data['city_stays']

    # Create a set of direct flights for
     quick lookup
    flight_set = set(tuple(flight) for
     flight in direct_flights)

    # Calculate total days available
    total_days =
     sum(city_stays.values()) -
     (len(cities) - 1)

    # List to store all valid plans
    valid_plans = []

    # Generate all permutations of cities
    for perm in permutations(cities):
        current_day = 1
        valid = True
        plan = []

        for i in range(len(perm)):
            city = perm[i]
            stay_days = city_stays[city]

            # Check if the current city
     can be reached from the previous city
            if i > 0:
                prev_city = perm[i - 1]
                if (prev_city, city) not
     in flight_set:
                    valid = False
                    break

            # Add the city and its stay
     days to the plan
            plan.append({
                "city": city,
                "days":
     list(range(current_day, current_day
     + stay_days))
            })

            # Update the current day,
     allowing for overlap
            current_day += stay_days - 1

            # Check if the total days
     exceed the allowed days
            if current_day > total_days
     + 1:
                valid = False
                break

        # If the plan is valid, add it
     to the list of valid plans
        if valid:
            valid_plans.append(plan)

    return valid_plans
<end_of_code_correction>
```

Output of Solution Function Generator Agent

```
<start_of_COT>

1. Cities - The list of cities is a
     fixed set and does not suggest any
     additional cities should be
     included. Hence, it is
     non-expandable.

2. Direct Flights - If a direct flight
     exists from City A to City B, it is
     reasonable to assume a direct flight
     could exist from City B to City A
     unless explicitly stated otherwise.
     Hence, it is expandable.

3. City Stays - The number of days to
     stay in each city is explicitly
     defined and does not suggest
     additional days should be added.
     Hence, it is non-expandable.

4. Specific Days - The specific days for
     visiting certain cities are
     explicitly defined for events or
     meetings. There is no indication
     that additional days should be
     added. Hence, it is non-expandable.

<end_of_COT>

<start_of_structured_output>
{
    "combinations": {
        "cities": [
            "Edinburgh",
```

```
            "Frankfurt",
            "Dubrovnik",
            "Vienna",
            "London",
            "Budapest",
            "Prague",
            "Valencia",
            "Munich",
            "Oslo"
        ],
        "direct_flights": [
            [
                "Valencia",
                "Munich"
            ],
            [
                "Munich",
                "Valencia"
            ],
            [
                "Vienna",
                "Munich"
            ],
            [
                "Munich",
                "Vienna"
            ],
            [
                "Vienna",
                "Valencia"
            ],
            [
                "Valencia",
                "Vienna"
            ],
            [
                "London",
                "Budapest"
            ],
            [
                "Budapest",
                "London"
            ],
            [
                "London",
                "Oslo"
            ],
            [
                "Oslo",
                "London"
            ],
            [
                "Edinburgh",
                "Budapest"
            ],
            [
                "Budapest",
                "Edinburgh"
            ],
            [
                "Frankfurt",
                "Budapest"
            ],
            [
                "Budapest",
                "Frankfurt"
            ],
            [
```

```
                "Frankfurt",
                "London"
            ],
            [
                "London",
                "Frankfurt"
            ],
            [
                "Prague",
                "Oslo"
            ],
            [
                "Oslo",
                "Prague"
            ],
            [
                "Edinburgh",
                "Oslo"
            ],
            [
                "Oslo",
                "Edinburgh"
            ],
            [
                "Edinburgh",
                "Munich"
            ],
            [
                "Munich",
                "Edinburgh"
            ],
            [
                "Prague",
                "Munich"
            ],
            [
                "Munich",
                "Prague"
            ],
            [
                "London",
                "Prague"
            ],
            [
                "Prague",
                "London"
            ],
            [
                "Edinburgh",
                "London"
            ],
            [
                "London",
                "Edinburgh"
            ],
            [
                "Edinburgh",
                "Frankfurt"
            ],
            [
                "Frankfurt",
                "Edinburgh"
            ],
            [
                "Dubrovnik",
                "Munich"
            ],
            [
```

```
            "Munich",
            "Dubrovnik"
        ],
        [
            "Dubrovnik",
            "Vienna"
        ],
        [
            "Vienna",
            "Dubrovnik"
        ],
        [
            "Munich",
            "Oslo"
        ],
        [
            "Oslo",
            "Munich"
        ],
        [
            "Dubrovnik",
            "Oslo"
        ],
        [
            "Oslo",
            "Dubrovnik"
        ],
        [
            "Budapest",
            "Munich"
        ],
        [
            "Munich",
            "Budapest"
        ],
        [
            "Frankfurt",
            "Prague"
        ],
        [
            "Prague",
            "Frankfurt"
        ],
        [
            "Vienna",
            "London"
        ],
        [
            "London",
            "Vienna"
        ],
        [
            "Frankfurt",
            "Vienna"
        ],
        [
            "Vienna",
            "Frankfurt"
        ],
        [
            "Frankfurt",
            "Oslo"
        ],
        [
            "Oslo",
            "Frankfurt"
        ],
        [
```

```
            "Frankfurt",
            "Munich"
        ],
        [
            "Munich",
            "Frankfurt"
        ],
        [
            "Vienna",
            "Oslo"
        ],
        [
            "Oslo",
            "Vienna"
        ],
        [
            "Vienna",
            "Prague"
        ],
        [
            "Prague",
            "Vienna"
        ],
        [
            "Budapest",
            "Oslo"
        ],
        [
            "Oslo",
            "Budapest"
        ],
        [
            "Budapest",
            "Prague"
        ],
        [
            "Prague",
            "Budapest"
        ],
        [
            "London",
            "Valencia"
        ],
        [
            "Valencia",
            "London"
        ],
        [
            "London",
            "Munich"
        ],
        [
            "Munich",
            "London"
        ],
        [
            "Frankfurt",
            "Dubrovnik"
        ],
        [
            "Dubrovnik",
            "Frankfurt"
        ],
        [
            "Prague",
            "Valencia"
        ],
        [
```

```
                "Valencia",
                "Prague"
            ],
            [
                "Frankfurt",
                "Valencia"
            ],
            [
                "Valencia",
                "Frankfurt"
            ],
            [
                "Edinburgh",
                "Prague"
            ],
            [
                "Prague",
                "Edinburgh"
            ],
            [
                "Vienna",
                "Budapest"
            ],
            [
                "Budapest",
                "Vienna"
            ]
        ],
        "city_stays": {
            "Edinburgh": 5,
            "Frankfurt": 5,
            "Dubrovnik": 2,
            "Vienna": 3,
            "London": 2,
            "Budapest": 3,
            "Prague": 3,
            "Valencia": 2,
            "Munich": 3,
            "Oslo": 2
        }
    },
    "constraints": {
        "specific_days": {
            "Edinburgh": [
                1,
                5
            ],
            "Oslo": [
                20,
                21
            ],
            "Valencia": [
                17,
                18
            ],
            "London": [
                12,
                13
            ],
            "Munich": [
                18,
                20
            ]
        }
    },
    "combinations_description": "The
    'combinations' key contains the
    parameters for generating possible
    travel plans. 'cities' is a list of
    cities to visit, 'direct_flights' is
    a list of pairs of cities that have
    direct flights between them, and
    'city_stays' specifies the number of
    days to stay in each city.",
    "constraints_description": "The
    'constraints' key contains the
    conditions that must be met for a
    valid travel plan. 'specific_days'
    specifies the exact days certain
    cities must be visited for events or
    meetings."
}
<end_of_structured_output>

<start_of_planning>
{
    "Input Agent": "Identify the set of
    cities to visit and the direct
    flight connections between cities.
    Also, identify the constraints such
    as specific days to be in certain
    cities for events or meetings.",
    "Combination Function Generator
    Agent": "Create a function
    combinations_func() that takes
    'cities', 'direct_flights', and
    'city_stays' as input. Generate all
    possible sequences of city visits
    that fit within the total days and
    only use direct flights, considering
    both directions for each flight.",
    "Solution Function Generator Agent":
    "Create a function plan_func() that
    takes 'specific_days' as
    constraints, along with the list of
    possible plans from
    combinations_func(). Filter the
    plans to find one that satisfies all
    constraints, ensuring all specific
    day requirements are met."
}
<end_of_planning>
```

Output of Solution Function Generator Agent (Does Not Require Reflection)

```
<start_of_COT>
To solve this problem, we need to filter
    the list of plans based on the
    constraints provided. The
    constraints specify certain days
    that specific cities must be
    visited. Our task is to find a plan
    that satisfies all these constraints.

1. **Understand the Input Format**:
   - Each plan is a list of
    dictionaries, where each dictionary
    represents a city visit with the
    city name and the days spent there.
   - Constraints are given in a
    dictionary format, where
    'specific_days' specifies the exact
    days certain cities must be visited.
```

```
2. **Plan the Function**:
   - Iterate over each plan in the
     'data' list.
   - For each plan, check if it
     satisfies all the constraints
     specified in 'specific_days'.
   - For each city in 'specific_days',
     ensure that the city is visited on
     all the required days in the plan.
   - If a plan satisfies all
     constraints, return that plan.

3. **Implementation Steps**:
   - Loop through each plan in the data.
   - For each plan, loop through each
     city in the 'specific_days'
     constraints.
   - Check if the city is present in the
     plan and if the days match the
     required days.
   - If all constraints are satisfied
     for a plan, return that plan.

4. **Edge Cases**:
   - If no plan satisfies the
     constraints, the function should
     handle this gracefully (though the
     problem does not specify what to do
     in this case, we assume at least one
     valid plan exists).

By following these steps, we can ensure
     that the function correctly filters
     and returns a plan that meets all
     the specified constraints.
<end_of_COT>

<start_of_code>
def plan_func(data, constraints):
    specific_days =
     constraints.get('specific_days', {})

    for plan in data:
        all_constraints_met = True

        for city, required_days in
     specific_days.items():
            city_found = False

            for visit in plan:
                if visit['city'] == city:
                    city_days =
     set(visit['days'])
                    if all(day in
     city_days for day in required_days):
                        city_found = True
                        break

            if not city_found:
                all_constraints_met =
     False
                break

        if all_constraints_met:
            return plan
<end_of_code>
```

Output of Deliver Function Generator Agent

```
<start_of_COT>
1. The few-shot example solution begins
     with a heading that summarizes the
     trip plan, mentioning the number of
     cities and days. The format is:
     "Here is the trip plan for visiting
     the X European cities for Y days:".
2. The output then details each city
     visit and the transitions between
     cities. Each city visit is described
     with the format: "**Day X-Y:** Visit
     [City] for Z days.".
3. Transitions between cities are
     described with the format: "**Day
     X:** Fly from [City1] to [City2].".
4. The function needs to iterate through
     the list of city visits, keeping
     track of the current day and
     ensuring that transitions and visits
     are formatted correctly.
5. The function should handle
     overlapping days correctly, ensuring
     that transitions are noted on the
     correct day.

To design the function:
- Start by calculating the total number
     of cities and days.
- Initialize a string with the heading.
- Iterate through the list of city
     visits, formatting each visit and
     transition according to the example.
- Keep track of the current day and
     adjust for overlaps in days between
     cities.
- Append each formatted string to the
     result.
- Return the complete formatted string
     as the output.
<end_of_COT>

<start_of_code>
def deliver_func(data):
    num_cities = len(data)
    total_days = max(day for city in
     data for day in city['days'])

    result = f"Here is the trip plan for
     visiting the {num_cities} European
     cities for {total_days} days:\n\n"

    current_day = 1
    for i, visit in enumerate(data):
        city = visit['city']
        days = visit['days']
        start_day = days[0]
        end_day = days[-1]
        visit_days = end_day - start_day
     + 1

        if start_day == current_day:
            result += f"**Day
     {start_day}-{end_day}:** Visit
     {city} for {visit_days} days.\n"
        else:
```

```
            result += f"**Day
    {start_day}:** Fly from
    {data[i-1]['city']} to {city}.\n"
            result += f"**Day
    {start_day}-{end_day}:** Visit
    {city} for {visit_days} days.\n"

        current_day = end_day + 1

    return result
<end_of_code>
```

J Other Tables and Figures

MODEL	SETTING	TravelPlanner						Trip.	Meet.
		DR	CPR		HCPR		SR	SR	SR
			Micro	Macro	Micro	Macro			
GPT-4o	Direct	100	93.6	57.0	56.4	38.7	23.6	4.0	50.7
	CoT	100	95.2	67.0	64.8	45.2	31.5	3.4	47.4
	SCOPE	100	**99.7**	**97.8**	**97.6**	**94.8**	**93.1**	**87.1**	**100**
GPT-o3	Direct	100	96.3	71.1	65.9	66.9	66.5	77.4	70.5
	CoT	100	98.6	89.1	83.2	80.7	80.4	77.4	89.8
	SCOPE	100	**99.6**	**97.5**	**97.2**	**94.0**	**92.2**	**89.1**	**99.2**
GPT-5	Direct	100	99.6	96.6	94.3	91.4	91.2	83.8	89.3
	CoT	100	99.5	96.6	94.3	91.8	91.5	85.5	93.2
	SCOPE	100	**99.7**	**98.3**	**97.8**	**95.3**	**93.9**	**88.7**	**100**
Gemini-1.5-Pro	Direct	100	**96.8**	75.0	54.6	27.5	21.8	37.6	43.2
	CoT	100	95.1	66.8	58.7	35.5	25.3	30.9	45.4
	SCOPE	97.5	95.9	**85.6**	**89.3**	**79.0**	**71.6**	**79.3**	**90.6**
Gemini-2.5-Pro	Direct	100	99.5	95.7	93.8	92.5	91.7	87.8	95.2
	CoT	100	99.6	96.9	96.3	94.8	93.8	87.9	96.4
	SCOPE	100	**99.7**	**98.2**	**98.0**	**95.6**	**94.2**	**90.4**	**99.6**

Table 4: Performance comparison across models and settings in inference mode. Full data for TravelPlanner (1000 data points), Trip Planning (1600 data points) and Meeting Planning (1000 data points).

MODEL	SETTING	TravelPlanner				Trip				Meet.	
		Input	Output	Cost	Time	Input	Output	Cost	Time	Input	Output
GPT-4o	Direct	**1055**	335	0.006	10	2375	**163**	0.008	**6**	5255	**214**
	CoT	1129	448	0.007	14	2450	474	0.011	14	5330	620
	ToT	54928	5330	0.191	102	17849	5396	0.099	88	18382	4427
	EvoAgent	95095	6052	0.298	128	54662	3396	0.171	75	75596	2451
	HTP	89977	4104	0.266	145	7236	2117	0.039	50	13297	942
	ToS	11099	**145**	0.029	5	**643**	349	**0.005**	8	**1485**	793
	SCOPE	1155	177	**0.005**	**3**	2288	340	0.009	13	2004	1436
GPT-o3	Direct	8157	2575	0.037	55	2381	4507	0.041	75	5261	3868
	CoT	8231	2945	0.04	43	2456	4763	0.043	94	5336	4851
	ToT	59503	17740	0.261	274	22569	54201	0.479	849	19782	21704
	EvoAgent	97134	22157	0.372	384	48657	38251	0.403	644	77324	42249
	HTP	90001	14512	0.296	276	7015	79385	0.649	1009	13600	7032
	ToS	11108	**655**	0.027	24	**670**	**1321**	**0.012**	**20**	**1464**	**1787**
	SCOPE	**1756**	1151	**0.013**	**17**	2560	1550	0.018	78	2628	3434
GPT-5	Direct	8157	4697	0.057	115	2381	3589	0.039	92	5261	4808
	CoT	8231	5254	0.063	142	2456	7482	0.078	167	5336	4311
	ToT	59453	36198	0.436	620	19770	50897	0.534	680	19950	43100
	HTP	89931	27082	0.383	391	7001	46314	0.472	600	13599	16244
	ToS	11207	**1435**	0.028	**23**	**669**	**2571**	**0.027**	**64**	1788	**2668**
	SCOPE	**1923**	1589	**0.018**	26	2616	5082	0.054	75	**3110**	4375
Gemini-2.5-Pro	Direct	**1126**	**871**	**0.014**	42	**2506**	**790**	**0.015**	**67**	6011	**844**
	CoT	1208	2221	0.035	53	2594	2316	0.038	78	6099	1616
	SCOPE	1804	1816	0.03	**16**	3234	6769	0.106	76	**3029**	6863
Gemini-1.5-Pro	Direct	**1126**	299	0.004	**5**	**2506**	**195**	**0.005**	**3**	6011	**248**
	CoT	1208	639	0.008	10	2594	749	0.011	12	6099	698
	SCOPE	1286	**181**	**0.003**	7	2648	369	0.007	55	**2311**	984

Table 5: Token counts, API cost (USD), and total time (seconds) across models and reasoning strategies for three tasks.

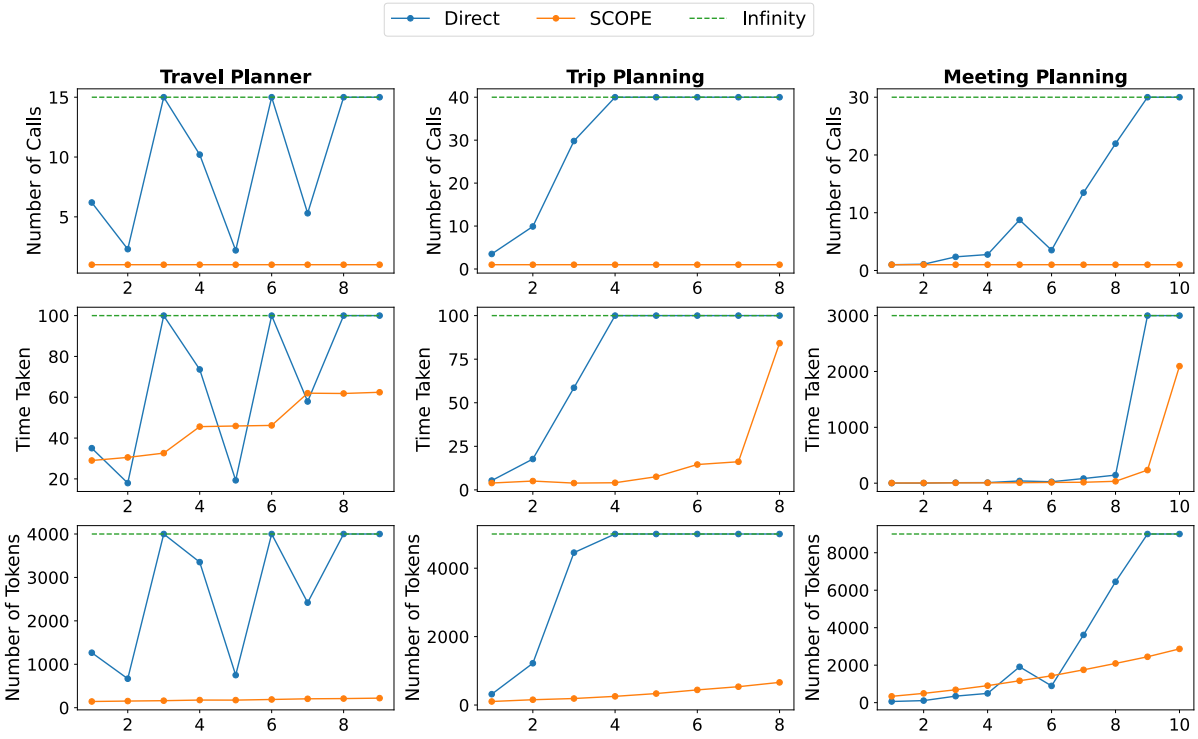

Figure 5: The inference cost required per query to reach correct solution. Top to bottom (Different metrics): Number of calls, time taken, number of tokens. Left to right (Different datasets): Travel Planner, Trip Planning, Meeting Planning. Infinity means an answer cannot be correctly achieved after many repeated attempts.