arXiv:2307.03875v2 [cs.AI] 13 Jul 2023

Large Language Models for Supply Chain Optimization

Beibin Li[1], Konstantina Mellou[1], Bo Zhang[2], Jeevan Pathuri[2], and Ishai Menache[1]

[1]Microsoft Research
[2]Microsoft Cloud Supply Chain

Abstract

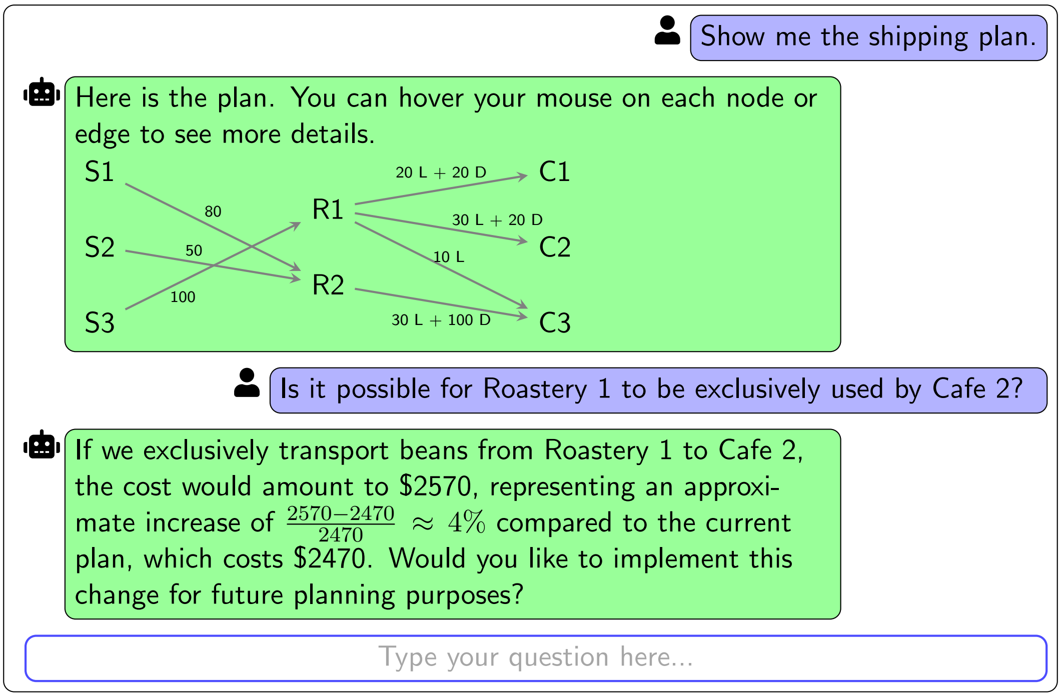
Supply chain operations traditionally involve a variety of complex decision making problems. Over the last few decades, supply chains greatly benefited from advances in computation, which allowed the transition from manual processing to automation and cost-effective optimization. Nonetheless, business operators still need to spend substantial efforts in *explaining* and interpreting the optimization outcomes to stakeholders. Motivated by the recent advances in Large Language Models (LLMs), we study how this disruptive technology can help bridge the gap between supply chain automation and human comprehension and trust thereof. We design *OptiGuide* – a framework that accepts as input queries in plain text, and outputs insights about the underlying optimization outcomes. Our framework does not forgo the state-of-the-art combinatorial optimization technology, but rather leverages it to quantitatively answer what-if scenarios (e.g., how would the cost change if we used supplier B instead of supplier A for a given demand?). Importantly, our design does not require sending proprietary data over to LLMs, which can be a privacy concern in some circumstances. We demonstrate the effectiveness of our framework on a real server placement scenario within Microsoft's cloud supply chain. Along the way, we develop a general evaluation benchmark, which can be used to evaluate the accuracy of the LLM output in other scenarios.

1 Introduction

Modern supply chains are complex, containing multiple tiers of suppliers, customers, and service providers [1]. Optimization tools have been widely utilized for decision making in such supply chains. These tools not only automate some of the decision making processes, but also result in efficiency gains and substantial cost reductions across many industries [2]. However, some of the automated processes require involving business operators, for understanding and explaining certain decisions, providing what-if analysis, and even overriding some optimization outcomes. In many cases, these operators are not equipped with the necessary background in optimization, resulting in time-consuming back-and-forth interactions with program managers, data scientists and engineers.

Large language models (LLMs) have recently emerged as a promising tool for assisting humans with a wide variety of tasks, such as writing documents, presenting work, coding and health diagnosis [3, 4, 5]. Generative multimodal LLMs, such as OpenAI's GPT-4, are being rapidly integrated within co-pilots, for answering questions and increasing productivity through simple, language based interactions with technology [6].

In this paper, we study how state-of-the-art LLMs can be applied for reasoning about supply chain optimization. Using LLMs in our context is challenging. First, the underlying optimization problems are often large scale combinatorial optimization problems, and solving



Show me the shipping plan.

Here is the plan. You can hover your mouse on each node or edge to see more details.

Is it possible for Roastery 1 to be exclusively used by Cafe 2?

If we exclusively transport beans from Roastery 1 to Cafe 2, the cost would amount to $2570, representing an approximate increase of $\frac{2570-2470}{2470} \approx 4\%$ compared to the current plan, which costs $2470. Would you like to implement this change for future planning purposes?

Figure 1: *OptiGuide* interface.

them directly is currently out of reach for LLMs [4]. Second, one needs to align the large foundation models to answer the domain-specific questions. Due to the large scale, fully training these models is not possible, and even middle-ground solutions such as fine-tuning LLMs require substantial compute and engineering investments [7]. Last but not least, any use of LLMs in business-critical operations, should have solutions when “things go wrong”, including diagnosing of and recovering from mistakes and hallucinations [8].

In view of these challenges, we design and implement *OptiGuide* – a framework that employs LLMs to interpret supply chain optimization solutions. A key idea behind *OptiGuide* is not to replace optimization technology by LLMs, but rather use optimization solvers in tandem with LLMs. In our design (see Figure 2 for system architecture), the LLM is responsible for translating the human query to “optimization code”, which is in turn used by an optimization solver to produce the necessary output; the output then passes through the LLM for producing the answer in human language (English). This architecture is used both for textual explanations and visualizations of the optimization solution, as well as for answering what-if queries. To address what-if queries, *OptiGuide* uses the LLM to appropriately modify the input to the optimization solver, and then reruns the solver under the hood to produce an answer.

To enable *OptiGuide*, we solve multiple technical challenges. First, we circumvent all forms of costly training, by applying in-context learning, namely “teaching” the LLM about the domain directly through the query’s prompt (i.e., as part of the inference). This requires careful co-design of the optimization code and the prompt with the understanding that the prompt can be space constrained. For example, we write the code in certain functional form that can be efficiently mapped to questions asked by humans. We also design a simple safeguard mechanism that confronts output mistakes.

To evaluate the ineffectiveness of *OptiGuide*, we introduce an *evaluation benchmark* that includes (i) a variety of common supply chain scenarios, and (ii) an evaluation methodology that incorporates new metrics for quantifying accuracy, generalizability within a scenario, and extrapolation capability to unseen scenarios. We test *OptiGuide* on five different scenarios and

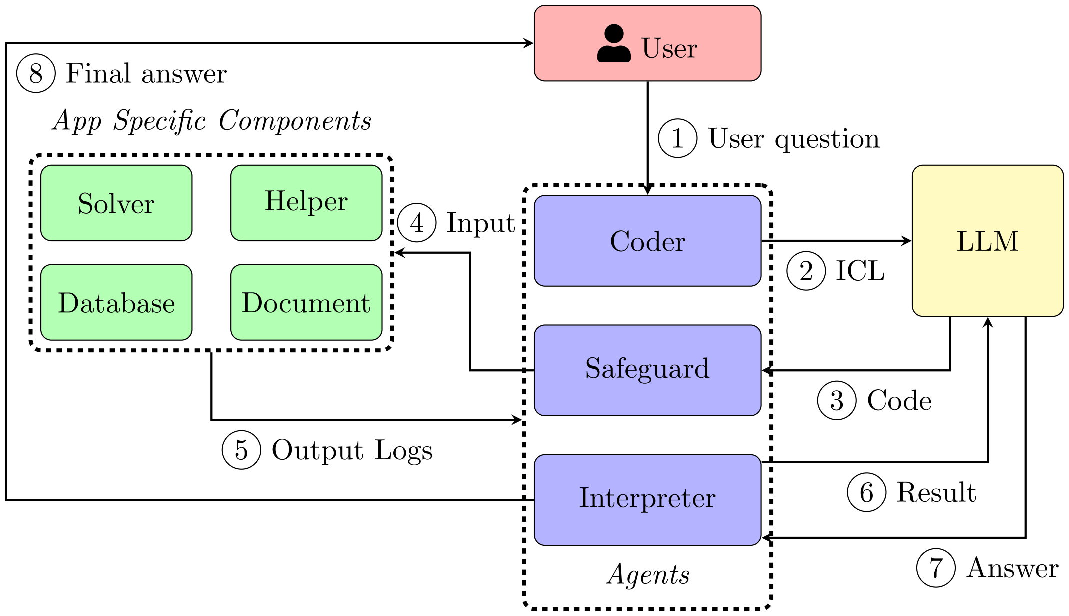


Figure 2: The *OptiGuide* framework.

obtain 93% accuracy on average using GPT-4. We view the benchmark and methodology as contributions that stand on their own, and can be used to evaluate future approaches. We are in the process of open-sourcing our benchmark. Finally, we deploy *OptiGuide* for the server deployment optimization used in Microsoft Azure's supply chain. We discuss some of the engineering challenges, and report initial promising results from our evaluation.

We believe that this paper sets important foundations, which can be used by other organizations for explaining optimization outcomes through LLMs. There are several future directions that emerge from our study, for example, using smaller models that can be trained with modest resources. As a longer-term goal, it is natural to expand the scope of LLMs beyond explainability, to facilitate *interactive* optimization (e.g., "please provide a more load-balanced solution", "please use at most two suppliers"). With the constant advances of LLM technology, it will be fascinating to examine whether LLMs can be utilized not only as translators, but also for refining and improving optimization outcomes.

The rest of the paper is organized as follows. In Section 2, we provide the necessary background on supply chain optimization and current LLM technology. In Section 3, we describe the design of *OptiGuide*. Section 4 describes our evaluation benchmark, and *OptiGuide*'s evaluation results. In Section 5, we outline our findings from *OptiGuide*'s deployment in Azure's supply chain. We discuss future perspectives in Section 6.

2 Background and Motivation

In this section, we provide brief background on decision making in supply chain operations, and elaborate on the notion of explainability. We then describe current capabilities and limitations of LLMs, and conclude with a simple dolphin choir example, which will be useful for explaining our solution approach.

2.1 Decision Making in Supply Chains

A supply chain may be defined as "an integrated network of facilities and transportation options for the supply, manufacture, storage, and distribution of materials and products" [9]. A simple supply chain may consist of a company (e.g., a service provider) and the set of its suppliers and customers [1]. However, most supply chains nowadays contain multiple tiers with suppliers of suppliers, customers of customers, and hierarchies of service providers [1]. This results in highly complex global networks where decisions must be optimized across multiple layers to satisfy customer demand while guaranteeing operational efficiency.

Decision making in supply chains spans different time-scales: starting from the design of the supply chain network (e.g., location of factories), planning (e.g., procurement of supply), and execution (e.g., transportation of goods). This leads to many types of decisions; a few examples:

- How many factories should we open, where, and with what manufacturing capacity?
- What suppliers should we use?
- How much inventory should we keep in stock and at which locations?
- How should we transport intermediate and finished goods efficiently?

The complexity of the decision-making often requires the design of optimization approaches that can incorporate a multitude of constraints and objectives, and still generate good quality solutions in plausible running times. To this end, different aspects of the supply chain (facility location, inventory planning, routing) may be optimized separately or considered jointly (e.g., inventory planning integrated with routing [10]). Common solution approaches for these optimization problems include Mixed Integer Programming based techniques and heuristics that can tackle the large scale of the problem.

2.2 Explainability

Business operators and planners involved in decision-making need to maintain a good understanding of the optimization outcomes. This allows them to not only address customer questions, but also react to unexpected events, and resolve inefficiencies and bottlenecks. However, the understanding is often challenging due to the complexity of the decision process (e.g., large scale, solution obtained by "black-box" algorithm, etc.) and lack of optimization expertise.

For concreteness, we provide below some examples of questions that operators may wish to answer.

Q1 What is the cost breakdown for each fulfilled demand?

Q2 How much excess inventory have I had per month in the past year?

Q3 What would happen if the demand at a particular location increased by 10%?

Q4 Can I reduce a factory's manufacturing capacity by 5% and still meet the demand?

Q5 Why was a particular supplier selected for a demand?

Q6 How would selecting a different transportation option affect the delivery timelines and the overall cost?

These and other questions aim at *explaining* the outcome of supply chain decisions. They include analyzing the current solution (input and output), investigating historical trends, and exploring what-if scenarios.

Obtaining insights on optimization decisions may require involving multiple professionals with different roles. Suppose that planners may wish to understand why a demand has not been fulfilled on time. They often surface the concern to the program managers, who involve domain experts, such as data scientists or the engineers that developed the optimization system. The domain experts in turn may need to write additional code and often rerun the optimization to extract the relevant insights. This overall process might be very time-consuming for all parties involved and can cause significant delays in the decision making process.

In some applications, teams maintain some custom tools that allow decision makers to reason about certain decisions. For example, application dashboards can provide visualizations or even allow enforcing some actions (e.g., fix a specific supplier for a demand). However, given the engineering overhead of maintaining the tools, they are typically limited to the most common use cases.

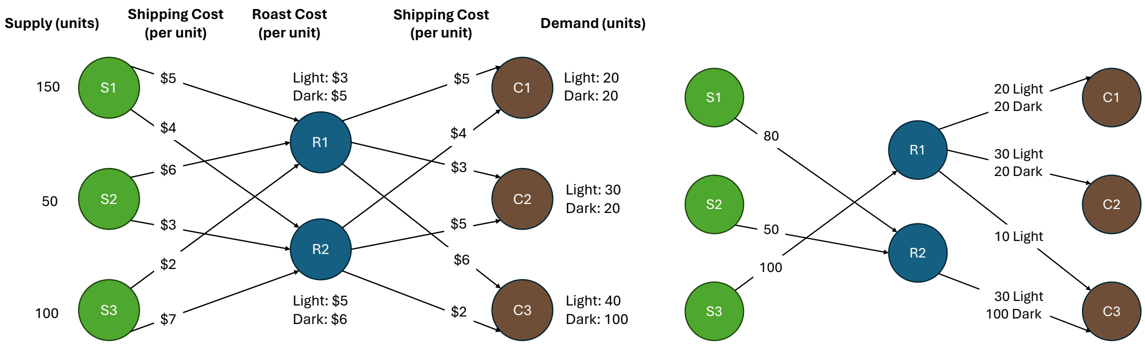
The notion of explainability is certainly not novel, and has drawn attention in both academia and industry. There have been numerous studies on explaining ML/AI [11, 12]. In the optimization context, IBM Decision Optimization [13] provides answers to a fixed set of queries that the user may choose to activate. See also [14] and references therein.

2.3 Large Language Models

Overview. A large language model (LLM) is a foundation model [15] trained on extensive text data using deep learning techniques, such as Transformer neural networks; ELMo [16], BERT [17], Turing NLG [18, 19], GPT-3 [20], GPT-4 [3], PaLM [21], PaLM-E [22], LLaMA [23], and Vicuna [24] are some examples of widely used LLMs. In the training phase, a LLM learns statistical patterns, word relationships, and contextual information from diverse sources, such as books, articles, websites, and code repositories. LLMs are used for a variety of tasks in the inference phase [4], including chatbots, translation, writing assistance, coding [25, 26, 27], planning [28], poem and story composition.

Using LLMs in applications. Multiple strategies can be employed to adapt LLMs for a specific application. The most common approaches are *fine-tuning* and *in-context learning*. Fine-tuning is a classic approach for "transfer learning" aimed at transferring knowledge from a pre-trained LLM to a model tailored for a specific application [29]. Typically, this process involves tweaking some weights of the LLM. While fine-tuning approaches can be made efficient [30, 31], they still necessitate model hosting in GPUs. This requirement can prove excessively costly for many applications. In-context learning [32] is an alternative cheaper approach, which involves incorporating a few training examples into the *prompt* (or query). The idea here is to append the prompt with domain-specific examples and have the LLM learn from these "few-shot" examples. A key advantage of this approach is that it does not require model parameter updates.

Prompt engineering. In a production setting, developers often send *prompts* (aka, queries) to the model, which can be appended with domain-specific examples for obtaining higher-quality answers. A collection of prompt management tools, such as ChatGPT Plugin [33], GPT function API call [34], LangChain [35], AutoGPT [36], and BabyAGI [37], have been designed to help engineers integrate LLMs in applications and services. The prompt size is measured in the number of *tokens*, which is proportional to the query size. LLMs can only process a limited number of tokens because of resource limitations, which is a strict constraint that developers and tools need to find workarounds for.



(a) Problem setup. (b) Optimal plan (units).

Figure 3: A simple supply chain example: coffee roasting company.

Privacy. Using domain-specific information in the prompt may involve proprietary data, which users may prefer not to reveal to LLM hosts. Even if LLM providers offer service level agreements (SLAs) for privacy, passive eavesdropping attackers might still intercept the data. Therefore, many organizations would prefer utilizing LLMs in a privacy-preserving way, namely keeping the proprietary data in-house.

Mistakes. Naturally, LLMs might provide sub-optimal outcomes, such as inaccuracies and even hallucinations [38]. There are generic tools that tackle this problem [39, 40, 41], however one may need domain specific tools for better outcomes. One example is fixing code generated by LLMs [42, 43, 44, 45].

2.4 A Simple Example

We now describe a simple supply chain example that will be useful for illustrating our approach.

The supply chain. Consider a coffee roasting company that roasts two types of coffee (light and dark roast). The company sources coffee beans from three different suppliers, it roasts them in one of its two roasting facilities, and then ships them to one of its three retail locations for selling to customers. The goal is to fulfill the demand in each retail location, while minimizing the total cost. The total cost consists of the cost of purchasing the coffee from the suppliers, the roasting cost in each facility, and the shipping cost of the end product to the retail locations. An illustration is given in Figure 3.

Model formulation. We can model this problem as a Mixed Integer Program. Let $x_{s,r}$ denote the number of units purchased from supplier s for roasting facility r, and $y^L_{r,\ell}$ and $y^D_{r,\ell}$ the amount of light and dark roast sent to retail location ℓ from roasting facility r. Each supplier s has a capacity C_s, and each retail location ℓ has demand D^L_ℓ and D^D_ℓ for light and dark roast respectively. There is a cost $c_{s,r}$ for each unit purchased from supplier s for roasting facility r, a shipping cost of $g_{r,\ell}$ for each unit sent to retail location ℓ from roasting facility r, and a roasting cost h^L_r and h^D_r per unit of light roast and dark roast respectively in facility r. The optimization problem is the following:

$$
\begin{aligned}
\text{minimize} \quad & \Big(\sum_{s,r} x_{s,r} \cdot c_{s,r} + \sum_{r,\ell} y^L_{r,\ell} \cdot h^L_r + \\
& \quad \sum_{r,\ell} y^D_{r,\ell} \cdot h^D_r + \sum_{r,\ell} (y^L_{r,\ell} + y^D_{r,\ell}) \cdot g_{r,\ell}\Big) && \text{(Objective)} \\
\text{subject to} \quad & \sum_{r} x_{s,r} \leq C_s && \forall s \quad \text{(Supplier capacity constraint)} \\
& \sum_{s} x_{s,r} = \sum_{\ell} (y^L_{r,\ell} + y^D_{r,\ell}) && \forall r \quad \text{(Conservation of flow constraint)} \\
& \sum_{r} y^L_{r,\ell} \geq D^L_\ell && \forall \ell \quad \text{(Light coffee demand constraint)} \\
& \sum_{r} y^D_{r,\ell} \geq D^D_\ell && \forall \ell \quad \text{(Dark coffee demand constraint)} \\
& x_{s,r}, y^L_{r,\ell}, y^D_{r,\ell} \in \mathbb{Z}^+ && \forall s, r, \ell \quad \text{(Integrality constraint)}
\end{aligned}
$$

Explainability. Let us now zoom into the example from Figure 3. The optimal solution is depicted in Figure 3b. We see that in the optimal plan, both roasteries produce light and dark coffee; the first roastery sources its beans from supplier 3, while the second from suppliers 1 and 2. The first two retail locations then obtain all their coffee from the first roastery, while the third retail location is supplied by both roasteries. A user may ask the following questions:

Q1 What would happen if the demand at retail location 1 increased by 10%?

Q2 What would happen if the demands at all retail locations doubled?

Q3 Why are we using supplier 3 for roasting facility 1?

Q4 Can I use roasting facility 1 only for retail location 2?

Q5 What if supplier 3 can now provide only half of the quantity?

Q6 The per-unit cost from supplier 3 to roasting facility 1 is now $5. How does that affect the total cost?

Q7 Why does Roastery 1 produce more light coffee than Roastery 2?

Q8 Why does supplier 1 ship more to Roastery 2 than Roastery 1?

Q9 Why not only use one supplier for Roastery 2?

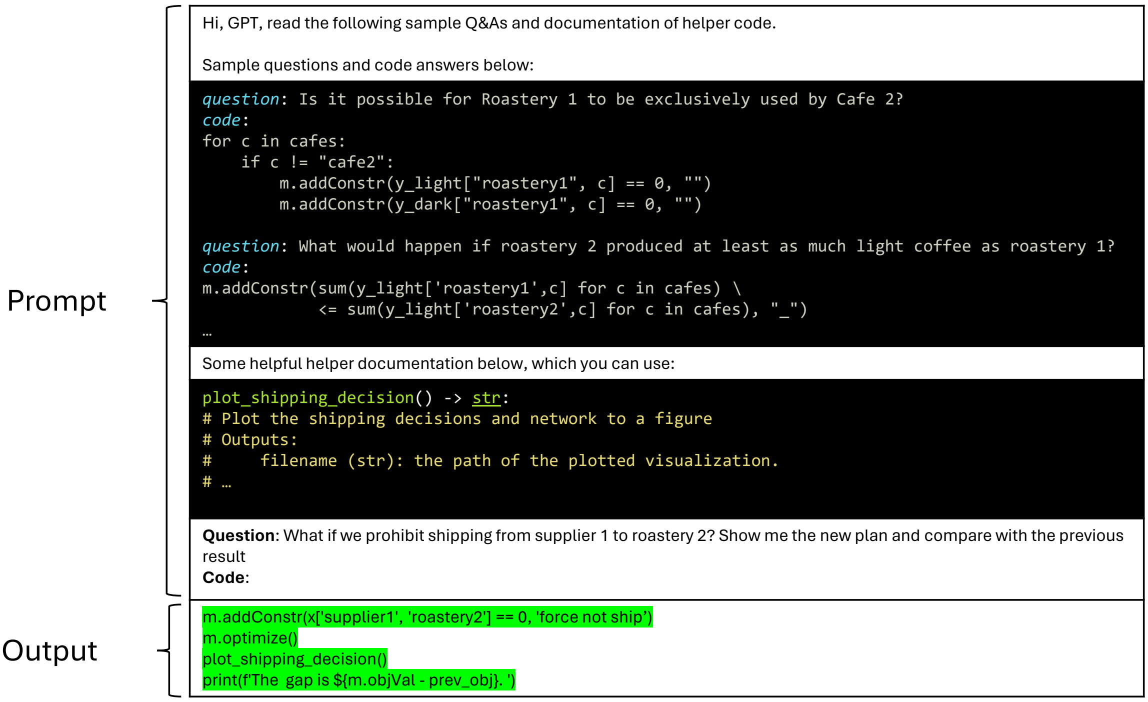
3 The LLM Framework

Large-scale supply chain management entails multiple functions, such as extensive data gathering, data processing and analysis, optimization processes and communication and enforcement of decisions across multiple stakeholders. While LLMs and supporting tools may handle part of these functions, there is a need for an end-to-end framework that will address the underlying challenges in a systematic way. In this section, we describe the design of our framework, *OptiGuide*.

3.1 System Overview

The *OptiGuide* framework, depicted in Figure 2, consists of three sets of entities: agents, LLMs, and application-specific components. When a user poses a question (①), the coder takes the question and formulates it as an in-context learning (ICL) question (②) for the LLM. The LLM then generates code (③) to answer the question. The safeguard checks the validity of the code and aborts the operation in case of a mistake; otherwise the safeguard feeds the code to an application specific component (④), such as a database engine or an optimization solver (depending on the query). The component processes the code and produces results, which are logged in a file (⑤). We note that obtaining the final result may involve multiple iterations (② to ⑤) where the query is automatically refined until the desired output is achieved. Finally, the output logs from the component are fed back into the LLM (⑥). The LLM analyzes the logs and generates a human-readable answer (⑦) that is sent back to the user (⑧). We now provide an overview of the different entities and components. More details can be found in Appendix B.

3.1.1 Agents


Prompt
Output

Hi, GPT, read the following sample Q&As and documentation of helper code.

Sample questions and code answers below:

```
question: Is it possible for Roastery 1 to be exclusively used by Cafe 2?
code:
for c in cafes:
    if c != "cafe2":
        m.addConstr(y_light["roastery1", c] == 0, "")
        m.addConstr(y_dark["roastery1", c] == 0, "")

question: What would happen if roastery 2 produced at least as much light coffee as roastery 1?
code:
m.addConstr(sum(y_light['roastery1',c] for c in cafes) \
            <= sum(y_light['roastery2',c] for c in cafes), "_")
…
```

Some helpful helper documentation below, which you can use:

```
plot_shipping_decision() -> str:
# Plot the shipping decisions and network to a figure
# Outputs:
#     filename (str): the path of the plotted visualization.
# …
```

Question: What if we prohibit shipping from supplier 1 to roastery 2? Show me the new plan and compare with the previous result
Code:

```
m.addConstr(x['supplier1', 'roastery2'] == 0, 'force not ship’)
m.optimize()
plot_shipping_decision()
print(f'The  gap is ${m.objVal - prev_obj}. ')
```

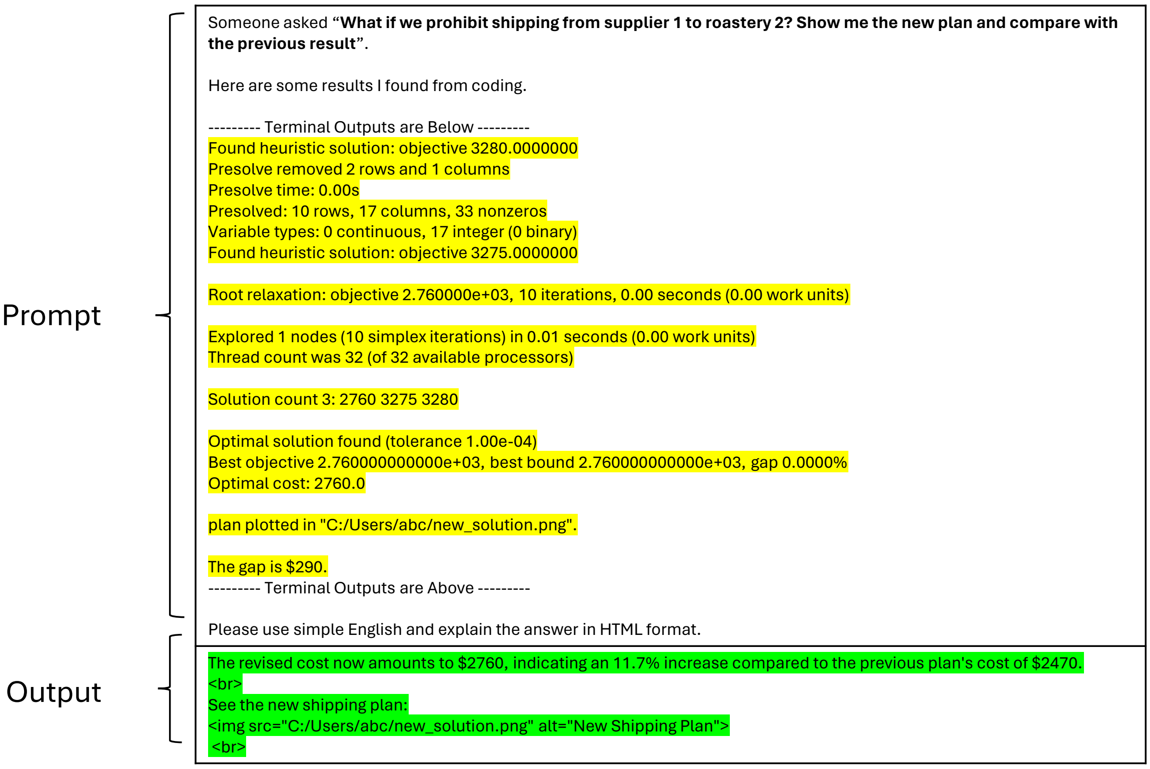
Figure 4: Coder prompt for the running example

Agents facilitate the interaction between users, the LLM, and application-specific components. The *coder* converts raw user questions into specific ICL queries. The conversion includes supplying the application context, providing ample training examples, and restructuring the user's query, as exemplified in Figure 4. The *safeguard* operates as a quality control checkpoint. It scrutinizes the code for potential discrepancies and initiates self-debugging upon encountering failures. When *OptiGuide* cannot successfully address a query, the safeguard would either initiate a new iteration with a proposed fix, or generate an error message for the user. The *interpreter* takes the output logs, tables, graphs, etc., and generates a human friendly response to the user's query.

3.1.2 Application Specific Components

Different applications may have different types of components; we provide an overview of the most common ones. *OptiGuide* is designed in a modular way, so that using *OptiGuide* for a different application requires only switching to a new set of components.

The *database* is a systematically arranged collection of data in various formats, such as CSV, SQL, JSON, Parquet, which are queried to extract answers. The *solver* can be a commercial integer programming solver, such as Gurobi. *OptiGuide* can query the solver output directly, or the output can be stored and queried from the database. If a question demands profound domain knowledge or historical context, *OptiGuide* consults *documents* to enhance the depth and relevance of the response. The *helper* is an optional component. It consists of a set of functions written by application engineers, for simplifying the code produced by LLMs. For example, a complex data analysis workflow can be simplified to a single helper function call.


Prompt
Output

Someone asked "**What if we prohibit shipping from supplier 1 to roastery 2? Show me the new plan and compare with the previous result**".

Here are some results I found from coding.

```
--------- Terminal Outputs are Below ---------
Found heuristic solution: objective 3280.0000000
Presolve removed 2 rows and 1 columns
Presolve time: 0.00s
Presolved: 10 rows, 17 columns, 33 nonzeros
Variable types: 0 continuous, 17 integer (0 binary)
Found heuristic solution: objective 3275.0000000

Root relaxation: objective 2.760000e+03, 10 iterations, 0.00 seconds (0.00 work units)

Explored 1 nodes (10 simplex iterations) in 0.01 seconds (0.00 work units)
Thread count was 32 (of 32 available processors)

Solution count 3: 2760 3275 3280

Optimal solution found (tolerance 1.00e-04)
Best objective 2.760000000000e+03, best bound 2.760000000000e+03, gap 0.0000%
Optimal cost: 2760.0

plan plotted in "C:/Users/abc/new_solution.png".

The gap is $290.
--------- Terminal Outputs are Above ---------
```

Please use simple English and explain the answer in HTML format.

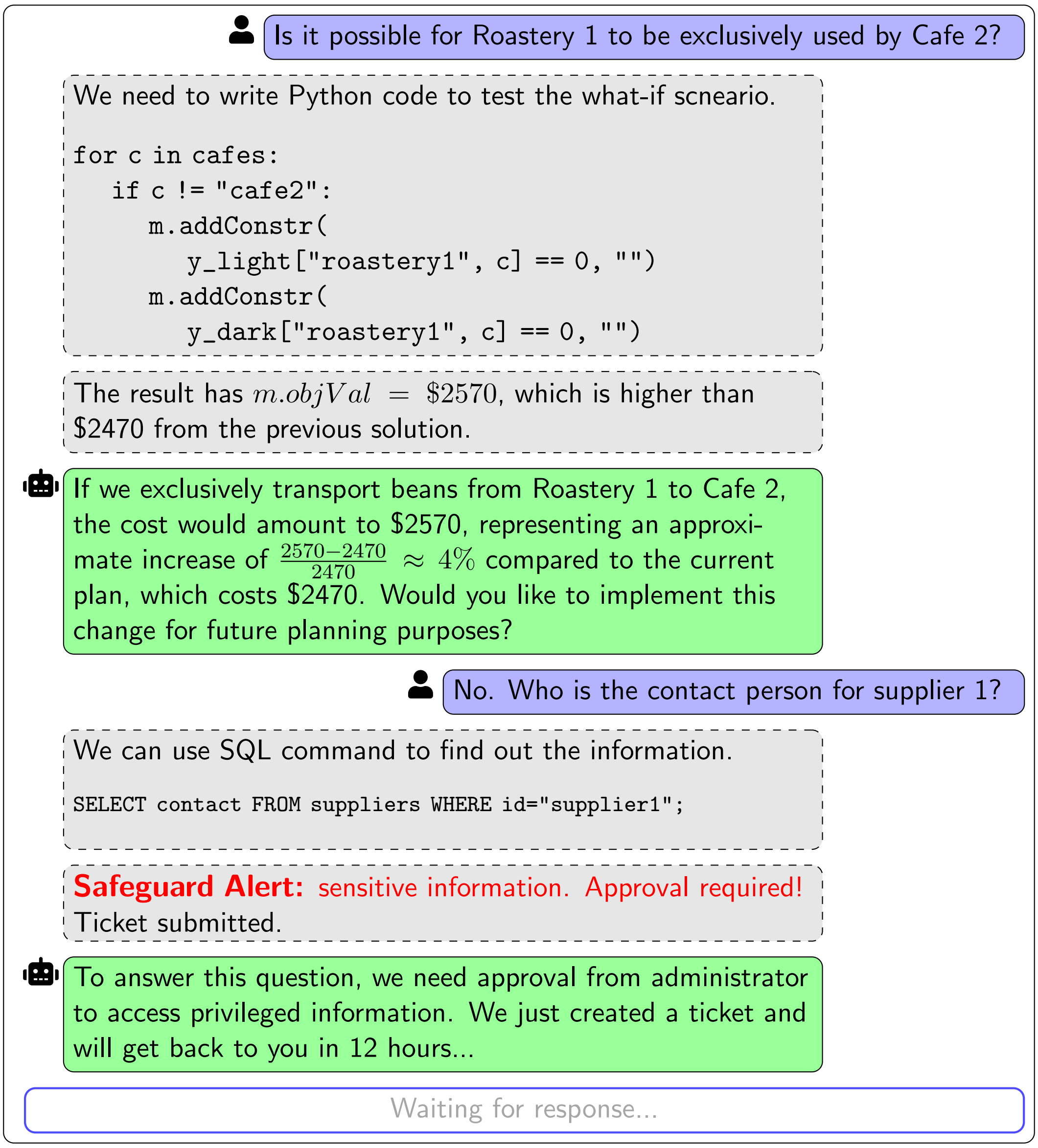
```
The revised cost now amounts to $2760, indicating an 11.7% increase compared to the previous plan's cost of $2470.
<br>
See the new shipping plan:
<img src="C:/Users/abc/new_solution.png" alt="New Shipping Plan">
<br>
```

Figure 5: Interpreter prompt for the running example

3.2 A Running Example

We illustrate *OptiGuide*'s data flow via the user question, "What if we prohibit shipping from supplier 1 to roastery 2? Show me the new plan and compare with the previous result". First, the coder converts this question into an in-context learning query for the LLM, see Figure 4 for the prompt. In addition to the question itself, the prompt contains (i) training examples, namely pairs of questions and code answers, and (ii) a documentation of the helper functions. Intuitively, (ii) supplements (i) by providing additional context into what the code does.

Subsequently, the LLM generates code that adds a new constraint (green region in Figure 4). The safeguard then extracts the code from the LLM's response, and *OptiGuide* calls the optimization solver to resolve the planning problem, yielding a result depicted in the yellow region in Figure 5. This result is then fed into the LLM by the interpreter, which produces a response. Finally, *OptiGuide* presents the response to the user alongside a visualization of the plan (green region in Figure 5) and a comparison with the original cost. Note that *OptiGuide* preserves privacy, since the domain-specific data remains in either the solver or database, and is never transferred to the LLM. Additional examples are provided in Figure 6.



Is it possible for Roastery 1 to be exclusively used by Cafe 2?

We need to write Python code to test the what-if scneario.

```
for c in cafes:
   if c != "cafe2":
      m.addConstr(
         y_light["roastery1", c] == 0, "")
      m.addConstr(
         y_dark["roastery1", c] == 0, "")
```

The result has $m.objVal = \$2570$, which is higher than $2470 from the previous solution.

If we exclusively transport beans from Roastery 1 to Cafe 2, the cost would amount to $2570, representing an approximate increase of $\frac{2570-2470}{2470} \approx 4\%$ compared to the current plan, which costs $2470. Would you like to implement this change for future planning purposes?

No. Who is the contact person for supplier 1?

We can use SQL command to find out the information.

```
SELECT contact FROM suppliers WHERE id="supplier1";
```

Safeguard Alert: sensitive information. Approval required! Ticket submitted.

To answer this question, we need approval from administrator to access privileged information. We just created a ticket and will get back to you in 12 hours...

Waiting for response...

Figure 6: An illustration of questions answered by *OptiGuide*. The gray dashed boxes represent thoughts that occur in the backend. Users can configure *OptiGuide* to display these thoughts or not.

4 Evaluation Benchmark

In this section, we develop a benchmark for evaluating the performance of our framework on a variety of supply chain optimization problems. The benchmark and the methodology around it can guide future efforts for using LLMs in supply chain optimization.

4.1 Scenarios and Data

To evaluate our framework, we selected a variety of optimization problems that capture multiple types of decisions that may be relevant in different supply chain settings. Specifically, our dataset includes a facility location scenario, a multi-commodity network flow for distribution of products, workforce assignment optimization, the traveling salesman problem, as well as the coffee distribution scenario from Section 2.4. The code for all problems is in Python and the Gurobi optimization solver [46] is used to obtain the optimal solution; Appendix C provides the code for the coffee distribution problem as an example.

Our next step is to generate a *repository* of questions and code answers for each scenario. Some of these question-answer pairs will be used as examples for in-context learning, while others for evaluating *OptiGuide*'s performance. To create a large set of questions, we write macros for each question, which results in generating *question sets* of closely related question-answer pairs. An example of a macro for a question set is the following:

```
QUESTION: What if we prohibit shipping from {{VALUE-X}} to {{VALUE-Y}}?
VALUE-X: random.choice(suppliers)
VALUE-Y: random.choice(roasteries)
GROUND-TRUTH: model.addConstr(x[{{VALUE-X}}, {{VALUE-Y}}] == 0)
```

In order to increase the diversity in the question sets, we also ask GPT to rephrase the questions while preserving their meaning. For instance, GPT might rephrase the generated question "Why would we ship beans from Supplier 1 to Roastery 2" to "What benefits are associated with the choice of shipping beans from Supplier 1 to Roastery 2?".

We note that the question sets for all problems that are used in the benchmark were created from scratch and kept in house, so that the LLMs have not observed these data as part of their training.

4.2 Evaluation Methodology

The goal of our evaluation is to assess the accuracy of LLMs in answering user questions for supply chain optimization problems. Unfortunately, existing metrics, such as pass@k which is used for analyzing coding accuracy [27, 47], are not well suited for explainability through code (intuitively, the metrics are "too forgiving"). We therefore propose a different methodology which is inspired by the unit-test approach used in software development.

Our evaluation proceeds as follows. For each scenario we run R experiments. Each experiment consists of T question sets. Each question set consists of Q test questions and answers. The LLM is asked to write the code and answer for a test question; it is given three chances to produce a response in case of an evident error (runtime or syntax). We then evaluate the correctness of the final answer. Note that we do not necessarily evaluate whether the generated code matches exactly with our ground-truth code, as there are different ways to obtain the correct response. The following example demonstrates a scenario where the generated code is quite different, but the optimization outcome would be the same.

```
1. model.addConstr(x['supplier1', 'roastery2'] == 0, 'force not ship')
2. shipping_cost_from_supplier_to_roastery[('supplier1', 'roastery2')] = 1e10
```

Accuracy. We define the accuracy metric AC as the average success rate across all scenarios, experiments and question sets. Formally,

$$AC = \frac{1}{SR} \sum_{s=1}^{S} \sum_{r=1}^{R} \frac{1}{T_s} \sum_{t=1}^{T_s} \mathbb{1}(q_t),$$

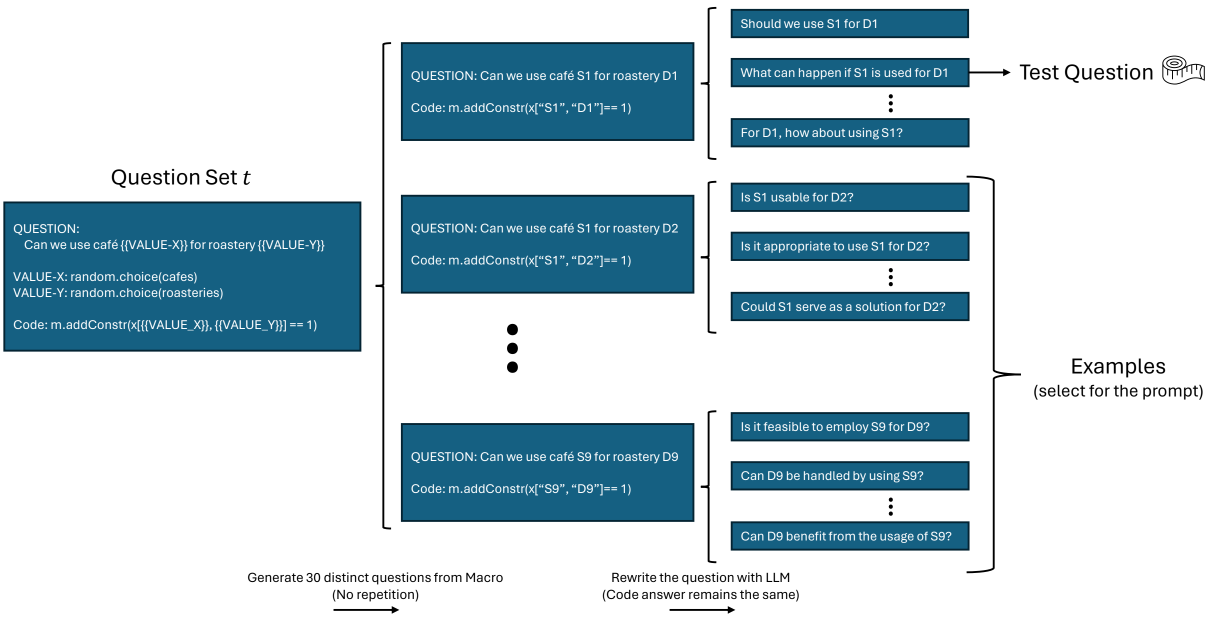

Question Set t
QUESTION:
Can we use café {{VALUE-X}} for roastery {{VALUE-Y}}
VALUE-X: random.choice(cafes)
VALUE-Y: random.choice(roasteries)
Code: m.addConstr(x[{{VALUE_X}}, {{VALUE_Y}}] == 1)
QUESTION: Can we use café S1 for roastery D1
Code: m.addConstr(x["S1", "D1"]== 1)
QUESTION: Can we use café S1 for roastery D2
Code: m.addConstr(x["S1", "D2"]== 1)
QUESTION: Can we use café S9 for roastery D9
Code: m.addConstr(x["S9", "D9"]== 1)
Should we use S1 for D1
What can happen if S1 is used for D1
Test Question
For D1, how about using S1?
Is S1 usable for D2?
Is it appropriate to use S1 for D2?
Could S1 serve as a solution for D2?
Examples
(select for the prompt)
Is it feasible to employ S9 for D9?
Can D9 be handled by using S9?
Can D9 benefit from the usage of S9?
Generate 30 distinct questions from Macro
(No repetition)
Rewrite the question with LLM
(Code answer remains the same)

Figure 7: In-distribution evaluation

where q_t is the question set, and $\mathbb{1}(q_t)$ is the indicator whether it passed successfully. The LLM passes a question set if and only if it successfully answers all questions in the question set.

In-distribution and out-of-distribution evaluation. As common practice, we evaluate our framework in both 'in-distribution' and 'out-of-distribution' [48] settings. For in-distribution evaluation (Figure 7), the test question and the examples used in the prompt are from the same question set. In contrast, for out-of-distribution evaluation (Figure 8), the example questions are extracted from different question sets.

Example selection. As the number of tokens that can be provided as input to the LLMs is limited, we explore different approaches for selecting the training examples for each query. The approaches can be evaluated both for in-distribution and out-of-distribution evaluation. One approach is *random selection*, where a fixed number of example questions is selected uniformly at random. Another approach is based on *nearest neighbors*, where we select examples that are similar to the test question; similarity is based on the text embedding [49] of the questions as determined by the model text-embedding-ada-002 [20]. We also experiment with different sizes of the example set (0, 1, 3, 5, or 10 examples).

4.3 Performance

Setup. For each scenario s, we run $R = 10$ experiments. In each experiment we evaluate $T_s \geq 10$ question sets. Each question set q_t usually contains $10 - 30$ questions and answers. We use both text-davinci-003 [20] and GPT-4 [3] for our evaluation. Performance results across different LLMs, example selection approaches, and example set sizes are summarized in Table 1.

Observations. GPT-4 consistently outperforms text-davinci-003 in both in-distribution and out-of-distribution evaluation. As expected, both models show higher accuracy on in-distribution

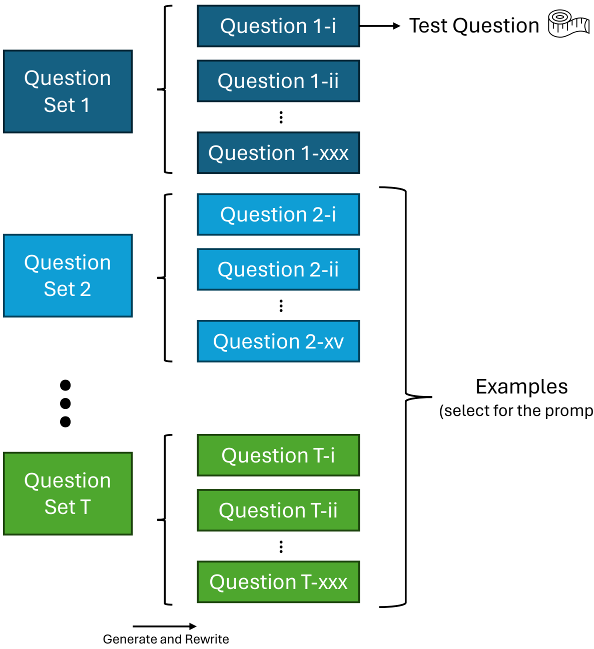


Figure 8: Out-of-distribution evaluation

Table 1: Accuracy across different LLMs, example selection approaches, and example set sizes. Each experiment was run 10 times and the average accuracy is reported.

		In-distribution		Out-of-distribution	
# Examples	Model	Random	Nearest	Random	Nearest
0	text-davinci-003	0.32			
	GPT-4	0.59			
1	text-davinci-003	0.78	0.78	0.39	0.44
	GPT-4	0.85	0.90	0.66	0.66
3	text-davinci-003	0.90	0.92	0.49	0.44
	GPT-4	0.90	0.92	0.74	0.69
5	text-davinci-003	0.93	0.93	0.52	0.48
	GPT-4	0.92	0.93	0.78	0.73
10	text-davinci-003	0.92	0.93	0.67	0.61
	GPT-4	0.93	0.93	0.84	0.80

compared to out-of-distribution evaluation. GPT-4 performs relatively much better in out-of-distribution evaluation, demonstrating its stronger reasoning and generalization capabilities; another sign for these capabilities is the 59% accuracy even without any training examples. Increasing the number of examples results in improved accuracy across the board. We also note that the gap between text-davinci-003 and GPT-4 decreases with the size of the example set.

The nearest neighbor selection approach yields slight performance improvements for in-distribution evaluation. Interestingly, when the size of the example set is greater than one, random selection outperforms nearest neighbor for out-of-distribution evaluation. One explanation here is that selecting examples based on text similarity results in overfitting, and random selection results in more diverse training examples.

5 *OptiGuide* for Azure's Supply Chain

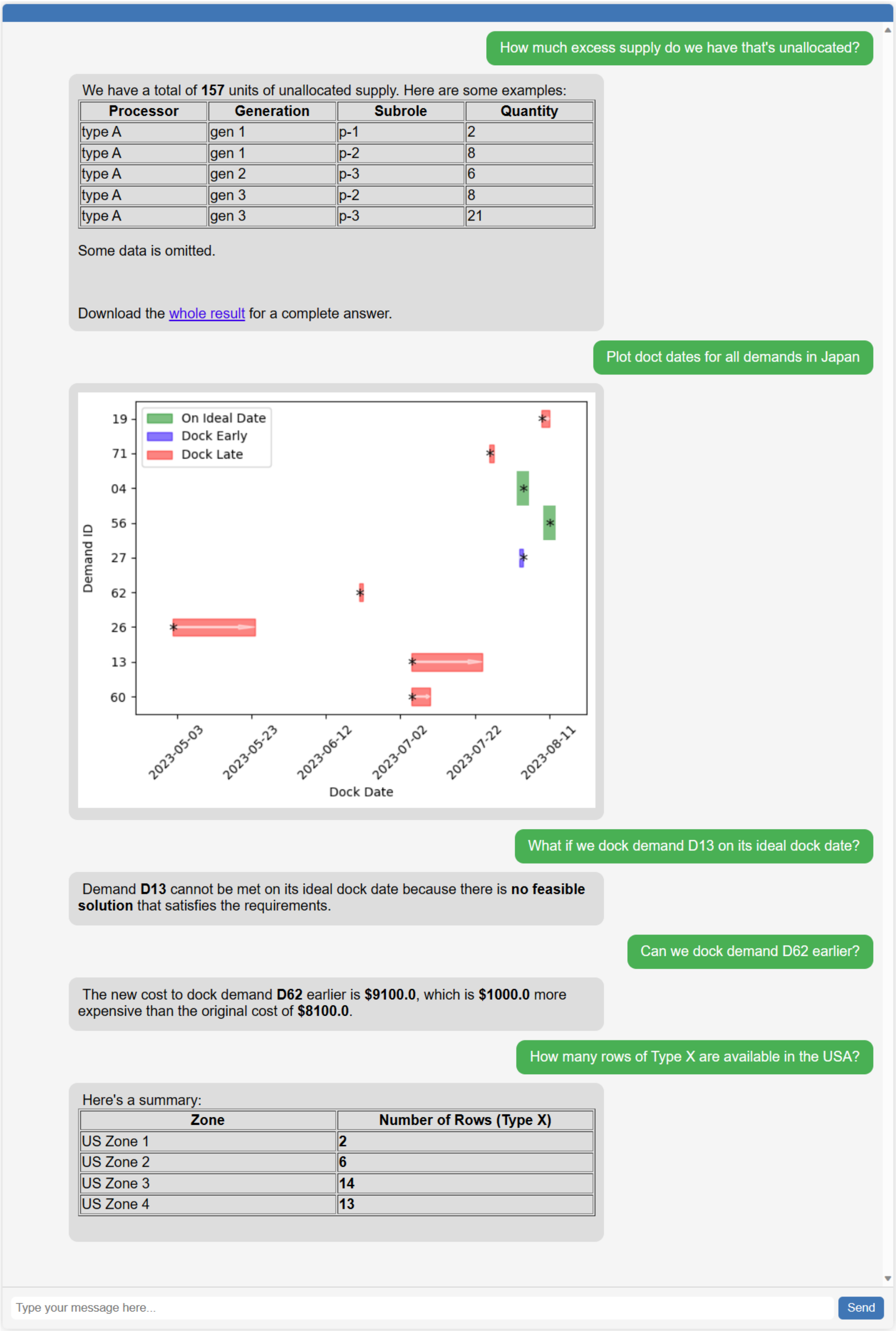
In this section, we demonstrate *OptiGuide*'s capabilities on the server fulfillment supply chain of Microsoft Azure. We start with providing the necessary details for the decisions involved in Azure's supply chain. We then outline the steps for deploying *OptiGuide* in production, and provide examples of user interactions and early feedback we obtained. We conclude this section by describing preliminary performance results.

5.1 The *Azure* Supply Chain

The rapid growth of the cloud industry requires cloud providers to continuously deploy additional capacity to keep up with the demand. This is achieved by acquiring new clusters of servers and deploying them in the data centers. The Microsoft Azure supply chain encompasses a broad array of processes including demand forecasting, strategic foresight, hardware semantic search, fulfillment planning, and document management. Due to complexity and large scale, the optimization of Azure's supply chain is assigned to different subsystems. We focus here on one such subsystem called Intelligent Fulfillment System (IFS), which deals with assigning and shipping servers from the warehouse to the data centers.

Main decisions. For each demand for cloud capacity, the main decisions consist of (i) the hardware supplier that will be used to fulfill the demand, (ii) the timeline of the deployment - in particular, the cluster's dock-date (which determines the date of shipping from the warehouse), and (iii) the cluster's deployment location in the data center (selection of a row of tiles to place the cluster on). The goal is to minimize the total cost that consists of multiple components, such as delay/idle cost of the clusters compared to their ideal dock-date and shipping costs, while respecting a multitude of constraints. Examples of constraints include capacity constraints on the suppliers and the data centers, location preferences for demands and compatibility constraints. The underlying optimization problem is formulated as a Mixed Integer Program (MIP) where the total input data size is around 500 MB. The optimal solution is obtained hourly using Gurobi. More details about the optimization problem can be found in Appendix A.

Stakeholders. The main consumers of IFS are *planners*. These are professionals that have the buisness context, so when they receive the outcome of the optimization, they can confirm that it meets business needs (or override decisions otherwise) and ensure the execution of the decisions is completed as planned. However, the increased complexity of the underlying optimization problem in combination with the global scale of decision making (hundreds of data centers) prevents immediate clarity in the reasoning behind each decision. Consequently, planners often reach out to the *engineers* (including data scientists) that develop the optimization system



How much excess supply do we have that's unallocated?

We have a total of **157** units of unallocated supply. Here are some examples:

Processor	Generation	Subrole	Quantity
type A	gen 1	p-1	2
type A	gen 1	p-2	8
type A	gen 2	p-3	6
type A	gen 3	p-2	8
type A	gen 3	p-3	21

Some data is omitted.

Download the whole result for a complete answer.

Plot doct dates for all demands in Japan

What if we dock demand D13 on its ideal dock date?

Demand **D13** cannot be met on its ideal dock date because there is **no feasible solution** that satisfies the requirements.

Can we dock demand D62 earlier?

The new cost to dock demand **D62** earlier is **$9100.0**, which is **$1000.0** more expensive than the original cost of **$8100.0**.

How many rows of Type X are available in the USA?

Here's a summary:

Zone	Number of Rows (Type X)
US Zone 1	**2**
US Zone 2	**6**
US Zone 3	**14**
US Zone 4	**13**

Figure 9: Screenshot of *OptiGuide* in Microsoft *Azure* production. We anonymized names and data by using generic values.

for obtaining additional insights. Oftentimes, planners and engineers have multiple rounds of interaction around understanding an issue or exploring what-if scenarios.

Common questions. We summarize below the main types of questions that are raised by planners:

- Q1 [Management] Does the system support a particular region, resource, or supplier?
- Q2 [Availability] Is a resource available or allocated?
- Q3 [Decisions] Why did the system make decision ‘x’ related to supplier/demand selection, time, and location?
- Q4 [Details of shipments] What are the details related to cross-geographical shipments and expected dock counts on a specific date?
- Q5 [Historical data analysis] What is the standard deviation of the supplier’s inventory in the last month?
- Q6 [Visualization] Can you visualize the dock capacity, availability, dates, or delays at a given location?

5.2 Deploying *OptiGuide* for Azure Supply Chain

Our current deployment of *OptiGuide* consists of (i) a front-end service for multiple-user interaction; (ii) an agent service, which is connected to Azure OpenAI for LLM access; (iii) multiple virtual machines (VMs) which host IFS and the application specific components to support multiple users at the same time.

We preload VMs’ memories with the input data and solver’s solutions to speedup code executions for users. The input data for the optimization problem are updated periodically (hourly), where the VMs load the updated data in a round-robin fashion so that there are always some VMs available to support users. We use GPT-4 as the LLM.

5.3 Preliminary Feedback and Results

Figure 9 provides examples of interactions between users and *OptiGuide*.

The preliminary feedback we obtained from both planners and engineers has been positive. Users expressed excitement noting the potential of *OptiGuide* to help them understand the underlying optimization logic. Users especially emphasized the benefits of supporting key what-if scenarios, which gives planners more autonomy and may substantially reduce the engineering on-call burden. For example, before *OptiGuide*, answering one what-if question would need more than three operators to coordinate the investigation and one on-call engineer to inspect the plan output.

Our preliminary evaluation indicates that *OptiGuide* can achieve more than 90% accuracy for our in-distribution evaluation. This result is consistent with the ones obtained in Section 4.

6 Concluding Remarks

We conclude this paper by discussing current limitations, and highlighting intriguing directions for future work.

6.1 Current Limitations

Users need to be specific. The user needs to ask precise questions. For instance, "Can we dock demand xc132 fifteen days earlier?" is ambiguous, because "earlier" can mean "15 days before today", "15 days before the currently planned date", or "15 days before the deadline". Consequently, the LLM might misunderstand the user and yield the wrong code.

Dependency on application-specific components. *OptiGuide* relies on proper design of application-specific components, such as the schema of the database and the helper functions. Some of these components might require non-negligible engineering efforts. While there has been progress in automating some of these components [50], there are still gaps in using them in some production settings.

Undetected mistakes. We observed cases where the LLM writes code that runs smoothly, but it may be totally wrong (e.g., due to string matching mistakes). We expect that things will improve in the future with more advances in LLMs and supporting tools.

Generalize to new questions. While the LLM performs well on seen questions, it still struggles when presented with questions that do not appear in the examples (see, e.g., Table 1). We believe that future models will have better generalizability.

Benchmark. Our current evaluation quantifies performance only for quantitative questions; for example, we exclude visualization queries from our analysis. Furthermore, the evaluation is based on a specific programming language (Python) and optimization solver (Gurobi).

6.2 Future Directions

We see our work as a cornerstone for future research in the area. One interesting direction is incorporating human feedback (e.g., from supply chain planners) which could lead to significant performance improvements [51]. Another direction that we are currently examining is using smaller models (see, e.g., [52] and references therein) for the specific tasks of supply chain optimization; using such models allows for more affordable hosting and fine-tuning of the model. In particular, we are examining whether fine-tuning can help with interpreting unseen questions. On a related note, it is of interest to consider a hybrid framework that combines the strengths of different AI models, for example combining large LMs with smaller ones. A natural longer-term goal is to go beyond explainability and facilitate *interactive* optimization, where the user directly influences the optimization outcomes; this will require designing more comprehensive safeguards, to prevent costly mistakes.

Acknowledgements

We thank Sébastien Bubeck, Yin Tat Lee, Chi Wang, Erkang Zhu, Leonardo Nunes, Srikanth Kandula, Adam Kalai, Marco Molinaro, Luke Marshall, Patricia Kovaleski, Hugo Barbalho, Tamires Santos, Runlong Zhou, Ashley Llorens, Surajit Chaudhuri, and Johannes Gehrke from Microsoft Research for useful discussions. We also thank Brian Houser, Matthew Meyer, Ryan Murphy, Russell Borja, Yu Ang Zhang, Rojesh Punnath, Naga Krothapalli, Navaneeth Echambadi, Apoorav Trehan, Jodi Larson, and Cliff Henson from the Microsoft Cloud Supply Chain for their advice and support.

References

[1] Michael H Hugos. *Essentials of supply chain management*. John Wiley & Sons, 2018.

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

[3] OpenAI. Gpt-4 technical report, 2023.

[4] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[5] Peter Lee, Sebastien Bubeck, and Joseph Petro. Benefits, limits, and risks of gpt-4 as an ai chatbot for medicine. *New England Journal of Medicine*, 388(13):1233–1239, 2023.

[6] GitHub. Github copilot: Your ai pair programmer, 2023.

[7] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[8] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. Prompt injection attack against llm-integrated applications. *arXiv preprint arXiv:2306.05499*, 2023.

[9] Daniel J Garcia and Fengqi You. Supply chain design and optimization: Challenges and opportunities. *Computers & Chemical Engineering*, 81:153–170, 2015.

[10] Pourya Pourhejazy and Oh Kyoung Kwon. The new generation of operations research methods in supply chain optimization: A review. *Sustainability*, 8(10):1033, 2016.

[11] Marina Danilevsky, Kun Qian, Ranit Aharonov, Yannis Katsis, Ban Kawas, and Prithviraj Sen. A survey of the state of explainable ai for natural language processing. *arXiv preprint arXiv:2010.00711*, 2020.

[12] Imran Ahmed, Gwanggil Jeon, and Francesco Piccialli. From artificial intelligence to explainable artificial intelligence in industry 4.0: a survey on what, how, and where. *IEEE Transactions on Industrial Informatics*, 18(8):5031–5042, 2022.

[13] Stefan Nickel, Claudius Steinhardt, Hans Schlenker, and Wolfgang Burkart. *Decision Optimization with IBM ILOG CPLEX Optimization Studio: A Hands-On Introduction to Modeling with the Optimization Programming Language (OPL)*. Springer Nature, 2022.

[14] Kristijonas Čyras, Dimitrios Letsios, Ruth Misener, and Francesca Toni. Argumentation for explainable scheduling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 2752–2759, 2019.

[15] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[16] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Corby Rosset. Turing-nlg: A 17-billion-parameter language model by microsoft. *Microsoft Blog*, 1(2), 2020.

[19] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[21] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[22] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[24] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

[25] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[26] Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, and Nachiappan Nagappan. Codecompose: A large-

scale industrial deployment of ai-assisted code authoring. *arXiv preprint arXiv:2305.12050*, 2023.

[27] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.

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

[29] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):1–40, 2016.

[30] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[31] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[32] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey for in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[33] OpenAI. ChatGPT plugins, 2023.

[34] OpenAI. Function calling and other API updates, 2023.

[35] LangChian. Introduction — langchain, 2023.

[36] Auto-GPT: An Autonomous GPT-4 Experiment, June 2023. original-date: 2023-03-16T09:21:07Z.

[37] BabyAGI. Translations: — BabyAGI, 2023.

[38] Nick McKenna, Tianyi Li, Liang Cheng, Mohammad Javad Hosseini, Mark Johnson, and Mark Steedman. Sources of hallucination by large language models on inference tasks. *arXiv preprint arXiv:2305.14552*, 2023.

[39] Potsawee Manakul, Adian Liusie, and Mark JF Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*, 2023.

[40] Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, et al. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*, 2023.

[41] Weiwei Sun, Zhengliang Shi, Shen Gao, Pengjie Ren, Maarten de Rijke, and Zhaochun Ren. Contrastive learning reduces hallucination in conversations. *arXiv preprint arXiv:2212.10400*, 2022.

[42] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. Skcoder: A sketch-based approach for automatic code generation. *arXiv preprint arXiv:2302.06144*, 2023.

[43] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*, 2023.

[44] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. *arXiv preprint arXiv:2203.05132*, 2022.

[45] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. Improving code generation by training with natural language feedback, 2023.

[46] Bob Bixby. The gurobi optimizer. *Transp. Research Part B*, 41(2):159–178, 2007.

[47] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[48] Kaiyang Zhou, Ziwei Liu, Yu Qiao, Tao Xiang, and Chen Change Loy. Domain generalization: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[49] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[50] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.

[51] Sarah Wiegreffe, Jack Hessel, Swabha Swayamdipta, Mark Riedl, and Yejin Choi. Reframing human-ai collaboration for generating free-text explanations. *arXiv preprint arXiv:2112.08674*, 2021.

[52] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

A Intelligent Fulfillment System

In this section, we present a partial formulation of the optimization in the Intelligent Fulfillment System that assigns and ships servers from the warehouse to the data centers.

A.1 Main Decisions

We introduce the following variables:

- $z_{dt} \in \{0,1\}$: equals 1 if demand d docks on day t, and 0 otherwise
- $u_{dr} \in \{0,1\}$: equals 1 if demand d docks on row r, and 0 otherwise
- $w_{ds} \in \{0,1\}$: equals 1 if d is fulfilled using supplier s, and 0 otherwise
- $y_{d,dc,t} \in \{0,1\}$: equals 1 if d docks at datacenter dc on day t, and 0 otherwise.
- $v_{d,s,t} \geq 0$: whether demand d docks on day t using supplier s or not

A.2 Constraints

This section describes some of the constraints in the formulation.

Docking day. The docking for each demand takes place on a single day.

$$\sum_t z_{dt} \leq 1 \qquad \forall d$$

Datacenter dockings. For each demand d, we dock at a datacenter dc on a specific day t only if the selected row belongs to that datacenter dc and the selected day is that particular day t.

$$\sum_{dc} y_{d,dc,t} \leq z_{dt} \qquad \forall d, t$$

$$\sum_t y_{d,dc,t} = \sum_{r \in rows(dc)} u_{dr} \qquad \forall d, dc$$

Datacenters' daily capacities. There are restrictions $restr$ on the daily amount of dockings that sets of datacenters can handle. Let R_d denote the number of racks required for demand d.

$$\sum_{d, dc \in DC(restr)} y_{d,dc,t} \cdot R_d \leq \text{DockRestrAvailCap}(restr, t) \quad \forall restr \in Restrictions, t$$

Single supplier. Each demand must be fulfilled by a single supplier. A row is selected for a demand only if a supplier has been found.

$$\sum_s w_{ds} \leq 1 \qquad \forall d$$

$$u_{dr} \leq \sum_s w_{ds} \qquad \forall d, r$$

Auxiliary supplier variables. Connecting variables v_{dst} with the rest of the variables.

$$z_{dt} = \sum_{s} v_{dst} \qquad \forall d, t$$

$$w_{ds} = \sum_{t} v_{dst} \qquad \forall d, t$$

Supply availability. We have a set of supply pools with a certain capacity (amount of available supply) evaluated at times ct. We need to make sure that the supply s we consume from each supply pool sp is available at the time t that we consume it. The time where each supply becomes available depends on its lead time.

$$\sum_{d, s \in sp, t \leq leadtime(ct, d, s)} v_{dst} \leq \text{Available\_Supply}(sp, ct) \qquad \forall sp, ct$$

Overrides. Some demand-supply combinations might be undesirable or disallowed for some reason. These can be explicitly blocked. Let B denote the set of blocked pairs.

$$w_{ds} = 0 \qquad \forall (d, s) \in B$$

A.3 Objective

Our goal is to minimize the total cost which is the aggregate of multiple components, including the cost of docking too early or too late compared to the ideal dock-date of each demand, the cost of not fulfilling demands, and the shipping cost, among others.

$$\text{DockCost} = \sum_{d,t} z_{dt} \cdot \text{Demand\_Day\_DockCost}(d, t)$$

$$\text{NoDockCost} = \sum_{d} (1 - \sum_{t} z_{dt}) \cdot \text{Unsatisfied\_Cost}(d)$$

$$\text{ShippingCost} = \sum_{d,s} w_{ds} \cdot \text{Transit\_Ship\_Cost}(d, s)$$

B Engineering Details

Figure 11, at the end of this document, presents a detailed screenshot of *OptiGuide* with *Azure* IFS, including intermediate results for illustration purposes.

B.1 Useful Tricks

SQL : Many LLMs are trained with SQL database. Hence, saving optimization input and output data into SQL could make the system easier to use and more explainable.

Logical simplification: If the prompt is not designed well, the LLM might make many simple logical mistakes (e.g., “not use” v.s. “use”, before v.s. after, etc.).

Intermediate outputs. When dealing with complex prompts, providing intermediate outputs can help keep the LLM on track. By returning intermediate results or steps, the LLM can check the consistency of its process, making it easier to debug and refine.

B.2 Failed Attempts

Chain of thought (CoT) failures. Unlike many recent studies [50] that have found that LLMs have strong CoT abilities, we found CoT is not helpful for writing complex code. This is another reason why we integrated the helper functions in the application-specific tools, which outperformed CoT. Our hypothesis is that if the LLM makes one mistake in the thinking chain, then the whole response would be wrong because correcting its own mistakes is hard.

Overuse of prompt engineering: While prompt engineering can often lead to improved results, overdoing it can sometimes lead to worse outcomes. When the prompts become too complex or too specific, the LLM might not understand them correctly or might overfit to the specific prompt structure, limiting its ability to handle a variety of questions.

C Coffee Distribution Example

C.1 Code

```
import time

from gurobipy import GRB, Model

# Example data

capacity_in_supplier = {'supplier1': 150, 'supplier2': 50, 'supplier3': 100}

shipping_cost_from_supplier_to_roastery = {
    ('supplier1', 'roastery1'): 5,
    ('supplier1', 'roastery2'): 4,
    ('supplier2', 'roastery1'): 6,
    ('supplier2', 'roastery2'): 3,
    ('supplier3', 'roastery1'): 2,
    ('supplier3', 'roastery2'): 7
}

roasting_cost_light = {'roastery1': 3, 'roastery2': 5}

roasting_cost_dark = {'roastery1': 5, 'roastery2': 6}

shipping_cost_from_roastery_to_cafe = {
    ('roastery1', 'cafe1'): 5,
    ('roastery1', 'cafe2'): 3,
    ('roastery1', 'cafe3'): 6,
    ('roastery2', 'cafe1'): 4,
    ('roastery2', 'cafe2'): 5,
    ('roastery2', 'cafe3'): 2
}

light_coffee_needed_for_cafe = {'cafe1': 20, 'cafe2': 30, 'cafe3': 40}

dark_coffee_needed_for_cafe = {'cafe1': 20, 'cafe2': 20, 'cafe3': 100}
```

```
cafes = list(set(i[1] for i in shipping_cost_from_roastery_to_cafe.keys()))
roasteries = list(
    set(i[1] for i in shipping_cost_from_supplier_to_roastery.keys()))
suppliers = list(
    set(i[0] for i in shipping_cost_from_supplier_to_roastery.keys()))

# OPTIGUIDE DATA CODE GOES HERE

# Create a new model
model = Model("coffee_distribution")

# Create variables
x = model.addVars(shipping_cost_from_supplier_to_roastery.keys(),
                  vtype=GRB.INTEGER,
                  name="x")
y_light = model.addVars(shipping_cost_from_roastery_to_cafe.keys(),
                        vtype=GRB.INTEGER,
                        name="y_light")
y_dark = model.addVars(shipping_cost_from_roastery_to_cafe.keys(),
                       vtype=GRB.INTEGER,
                       name="y_dark")

# Set objective
model.setObjective(
    sum(x[i] * shipping_cost_from_supplier_to_roastery[i]
        for i in shipping_cost_from_supplier_to_roastery.keys()) +
    sum(roasting_cost_light[r] * y_light[r, c] +
        roasting_cost_dark[r] * y_dark[r, c]
        for r, c in shipping_cost_from_roastery_to_cafe.keys()) + sum(
            (y_light[j] + y_dark[j]) * shipping_cost_from_roastery_to_cafe[j]
            for j in shipping_cost_from_roastery_to_cafe.keys()), GRB.MINIMIZE)

# Conservation of flow constraint
for r in set(i[1] for i in shipping_cost_from_supplier_to_roastery.keys()):
    model.addConstr(
        sum(x[i]
            for i in shipping_cost_from_supplier_to_roastery.keys()
            if i[1] == r) == sum(
                y_light[j] + y_dark[j]
                for j in shipping_cost_from_roastery_to_cafe.keys()
                if j[0] == r), f"flow_{r}")

# Add supply constraints
for s in set(i[0] for i in shipping_cost_from_supplier_to_roastery.keys()):
    model.addConstr(
        sum(x[i]
            for i in shipping_cost_from_supplier_to_roastery.keys()
            if i[0] == s) <= capacity_in_supplier[s], f"supply_{s}")
```

```
# Add demand constraints
for c in set(i[1] for i in shipping_cost_from_roastery_to_cafe.keys()):
    model.addConstr(
        sum(y_light[j]
            for j in shipping_cost_from_roastery_to_cafe.keys()
            if j[1] == c) >= light_coffee_needed_for_cafe[c],
        f"light_demand_{c}")
    model.addConstr(
        sum(y_dark[j]
            for j in shipping_cost_from_roastery_to_cafe.keys()
            if j[1] == c) >= dark_coffee_needed_for_cafe[c], f"dark_demand_{c}")

# Optimize model
model.optimize()
m = model

# OPTIGUIDE CONSTRAINT CODE GOES HERE

# Solve
m.update()
model.optimize()

print(time.ctime())
if m.status == GRB.OPTIMAL:
    print(f'Optimal cost: {m.objVal}')
else:
    print("Not solved to optimality. Optimization status:", m.status)
```

C.2 Question and Ground Truth Macros

```
QUESTION:
What would happen if demand at cafe {{VALUE-CAFE}} increased by {{VALUE-NUMBER}}%?
VALUE-CAFE: random.choice(cafes)
VALUE-NUMBER: random.randrange(5,30)
DATA CODE:
light_coffee_needed_for_cafe[{{VALUE-CAFE}}] =  \
    light_coffee_needed_for_cafe[{{VALUE-CAFE}}] * (1 + {{VALUE-NUMBER}}/100)
dark_coffee_needed_for_cafe[{{VALUE-CAFE}}] =  \
    dark_coffee_needed_for_cafe[{{VALUE-CAFE}}] * (1 + {{VALUE-NUMBER}}/100)
TYPE: demand-increase

QUESTION:
What if demand for light coffee at cafe {{VALUE-CAFE}} increased by {{VALUE-NUMBER}}%?
VALUE-CAFE: random.choice(cafes)
VALUE-NUMBER: random.randrange(5,30)
DATA CODE:
light_coffee_needed_for_cafe[{{VALUE-CAFE}}] = \
    light_coffee_needed_for_cafe[{{VALUE-CAFE}}] * (1 + {{VALUE-NUMBER}}/100)
```

```
TYPE: demand-increase-light

QUESTION:
What would happen if the demand at all cafes doubled?
DATA CODE:
for c in cafes:
    light_coffee_needed_for_cafe[c] = light_coffee_needed_for_cafe[c] * 2
    dark_coffee_needed_for_cafe[c] = dark_coffee_needed_for_cafe[c] * 2
TYPE: demand-increase-all

QUESTION:
Why are we using supplier {{VALUE-SUPPLIER}} for roasting facility {{VALUE-ROASTERY}}?
VALUE-SHIPPINGS: [(s, r) for (s, r), value in x.items() if value.X >= 0.999]
VALUE-IDX: random.randint(0, len({{VALUE-SHIPPINGS}}) - 1)
VALUE-SUPPLIER: {{VALUE-SHIPPINGS}}[{{VALUE-IDX}}][0]
VALUE-ROASTERY: {{VALUE-SHIPPINGS}}[{{VALUE-IDX}}][1]
CONSTRAINT CODE:
m.addConstr(x[{{VALUE-SUPPLIER}},{{VALUE-ROASTERY}}] == 0, "_")
TYPE: supply-roastery

QUESTION:
Assume cafe {{VALUE-CAFE}} can exclusively buy coffee from roasting facility
{{VALUE-ROASTERY}}, and conversely, roasting facility {{VALUE-ROASTERY}}
can only sell its coffee to cafe {{VALUE-CAFE}}. How does that affect the outcome?
VALUE-ROASTERY: random.choice(roasteries)
VALUE-CAFE: random.choice(cafes)
CONSTRAINT CODE:
for c in cafes:
    if c != {{VALUE-CAFE}}:
        m.addConstr(y_light[{{VALUE-ROASTERY}}, c] == 0, "_")
        m.addConstr(y_dark[{{VALUE-ROASTERY}}, c] == 0, "_")
for r in roasteries:
    if r != {{VALUE-ROASTERY}}:
        m.addConstr(y_light[r,{{VALUE-CAFE}}] == 0, "_")
        m.addConstr(y_dark[r,{{VALUE-CAFE}}] == 0, "_")
TYPE: exclusive-roastery-cafe

QUESTION:
What if roasting facility {{VALUE-ROASTERY}} can only be used for cafe {{VALUE-CAFE}}?
VALUE-ROASTERY: random.choice(roasteries)
VALUE-CAFE: random.choice(cafes)
CONSTRAINT CODE:
for c in cafes:
    if c != {{VALUE-CAFE}}:
        m.addConstr(y_light[{{VALUE-ROASTERY}}, c] == 0, "_")
        m.addConstr(y_dark[{{VALUE-ROASTERY}}, c] == 0, "_")
TYPE: incompatible-roastery-cafes

QUESTION:
```

```
What if supplier {{VALUE-SUPPLIER}} can now provide only half of the quantity?
VALUE-SUPPLIER: random.choice(suppliers)
DATA CODE:
capacity_in_supplier[{{VALUE-SUPPLIER}}] = capacity_in_supplier[{{VALUE-SUPPLIER}}]/2
TYPE: supplier-capacity

QUESTION:
The per-unit cost from supplier {{VALUE-SUPPLIER}} to roasting facility {{VALUE-ROASTERY}}
is now {{VALUE-NUMBER}}. How does that affect the total cost?
VALUE-SUPPLIER: random.choice(suppliers)
VALUE-ROASTERY: random.choice(roasteries)
VALUE-NUMBER: random.randrange(1,10)
DATA CODE:
shipping_cost_from_supplier_to_roastery[{{VALUE-SUPPLIER}},{{VALUE-ROASTERY}}] = \
    {{VALUE-NUMBER}}
TYPE: supplier-roastery-shipping

QUESTION:
What would happen if roastery 2 produced at least as much light coffee as roastery 1?
CONSTRAINT CODE:
m.addConstr(sum(y_light['roastery1',c] for c in cafes)
            <= sum(y_light['roastery2',c] for c in cafes), "_")
TYPE: light-quantities-roasteries

QUESTION:
What would happen if roastery 1 produced less light coffee than roastery 2?
CONSTRAINT CODE:
m.addConstr(sum(y_light['roastery1',c] for c in cafes)
            <= sum(y_light['roastery2',c] for c in cafes) - 1, "_")
TYPE: light-quantities-roasteries

QUESTION:
What will happen if supplier 1 ships more to roastery 1 than roastery 2?
CONSTRAINT CODE:
m.addConstr(x['supplier1','roastery1'] >= x['supplier1','roastery2'] + 1, "_")
TYPE: shipping-quantities-roasteries

QUESTION:
What will happen if supplier 1 ships to roastery 1 at least as much as to roastery 2?
CONSTRAINT CODE:
m.addConstr(x['supplier1','roastery1'] >= x['supplier1','roastery2'], "_")
TYPE: shipping-quantities-roasteries

QUESTION:
Why not only use a single supplier for roastery 2?
CONSTRAINT CODE:
z = m.addVars(suppliers, vtype=GRB.BINARY, name="z")
m.addConstr(sum(z[s] for s in suppliers) <= 1, "_")
for s in suppliers:
    m.addConstr(x[s,'roastery2'] <= capacity_in_supplier[s] * z[s], "_")
TYPE: single-supplier-roastery
```

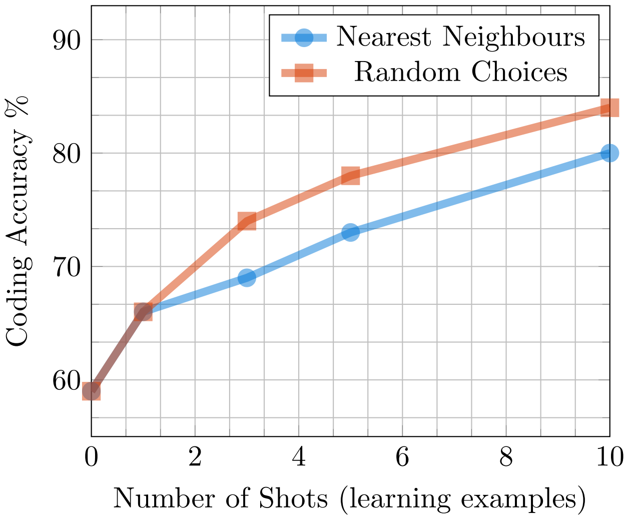
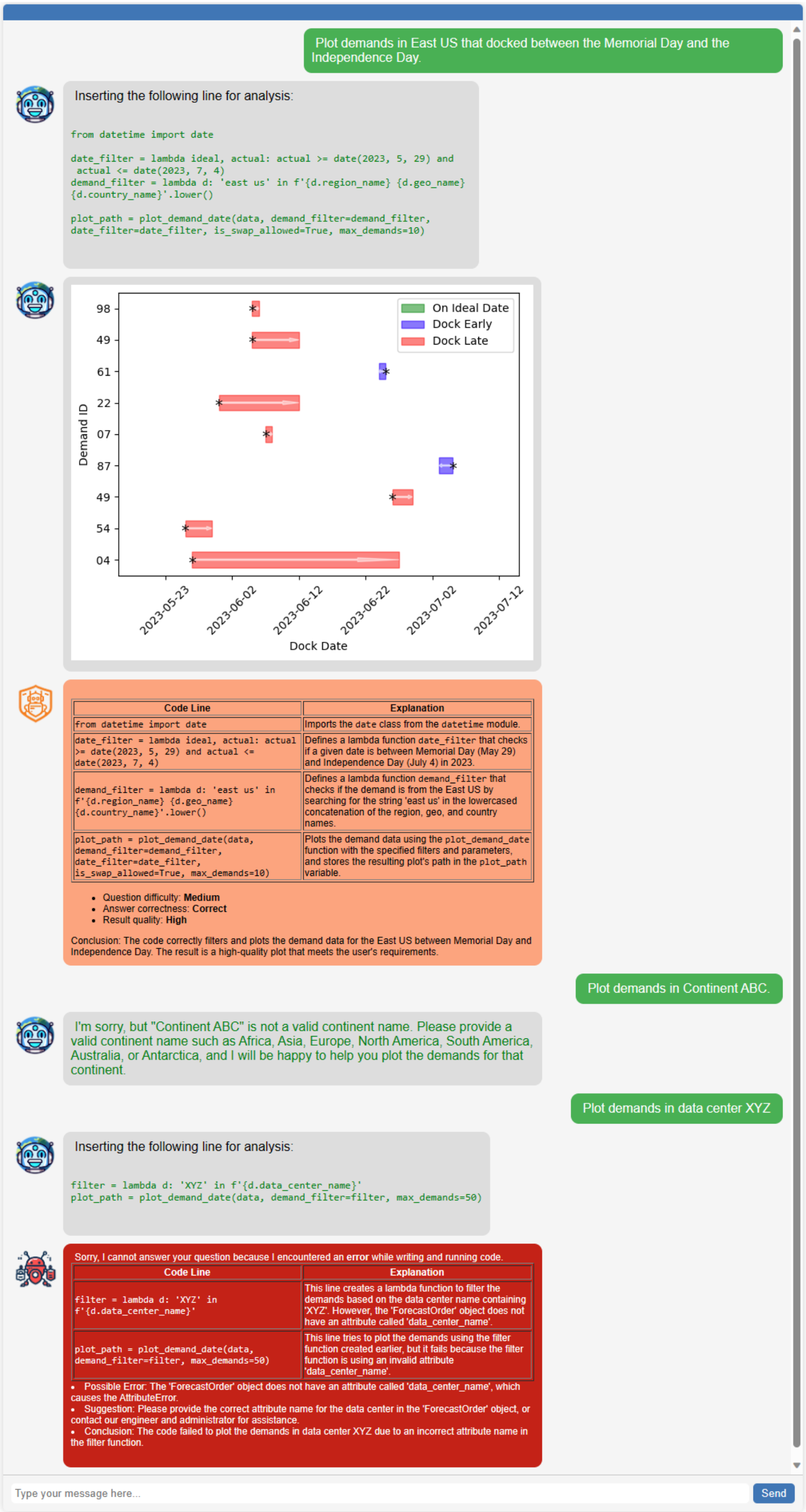


Figure 10: Out-of-distribution evaluation for GPT-4. We compare the different training example selection methods here.



Plot demands in East US that docked between the Memorial Day and the Independence Day.

Inserting the following line for analysis:

```
from datetime import date

date_filter = lambda ideal, actual: actual >= date(2023, 5, 29) and
 actual <= date(2023, 7, 4)
demand_filter = lambda d: 'east us' in f'{d.region_name} {d.geo_name}
{d.country_name}'.lower()

plot_path = plot_demand_date(data, demand_filter=demand_filter,
date_filter=date_filter, is_swap_allowed=True, max_demands=10)
```

Code Line	Explanation
`from datetime import date`	Imports the `date` class from the `datetime` module.
`date_filter = lambda ideal, actual: actual >= date(2023, 5, 29) and actual <= date(2023, 7, 4)`	Defines a lambda function `date_filter` that checks if a given date is between Memorial Day (May 29) and Independence Day (July 4) in 2023.
`demand_filter = lambda d: 'east us' in f'{d.region_name} {d.geo_name} {d.country_name}'.lower()`	Defines a lambda function `demand_filter` that checks if the demand is from the East US by searching for the string 'east us' in the lowercased concatenation of the region, geo, and country names.
`plot_path = plot_demand_date(data, demand_filter=demand_filter, date_filter=date_filter, is_swap_allowed=True, max_demands=10)`	Plots the demand data using the `plot_demand_date` function with the specified filters and parameters, and stores the resulting plot's path in the `plot_path` variable.

- Question difficulty: **Medium**
- Answer correctness: **Correct**
- Result quality: **High**

Conclusion: The code correctly filters and plots the demand data for the East US between Memorial Day and Independence Day. The result is a high-quality plot that meets the user's requirements.

Plot demands in Continent ABC.

I'm sorry, but "Continent ABC" is not a valid continent name. Please provide a valid continent name such as Africa, Asia, Europe, North America, South America, Australia, or Antarctica, and I will be happy to help you plot the demands for that continent.

Plot demands in data center XYZ

Inserting the following line for analysis:

```
filter = lambda d: 'XYZ' in f'{d.data_center_name}'
plot_path = plot_demand_date(data, demand_filter=filter, max_demands=50)
```

Sorry, I cannot answer your question because I encountered an **error** while writing and running code.

Code Line	Explanation
`filter = lambda d: 'XYZ' in f'{d.data_center_name}'`	This line creates a lambda function to filter the demands based on the data center name containing 'XYZ'. However, the 'ForecastOrder' object does not have an attribute called 'data_center_name'.
`plot_path = plot_demand_date(data, demand_filter=filter, max_demands=50)`	This line tries to plot the demands using the filter function created earlier, but it fails because the filter function is using an invalid attribute 'data_center_name'.

- Possible Error: The 'ForecastOrder' object does not have an attribute called 'data_center_name', which causes the AttributeError.
- Suggestion: Please provide the correct attribute name for the data center in the 'ForecastOrder' object, or contact our engineer and administrator for assistance.
- Conclusion: The code failed to plot the demands in data center XYZ due to an incorrect attribute name in the filter function.

Figure 11: *OptiGuide* for *Azure* IFS. Intermediate results from agents are shown in the screenshot.