A Critical Assessment of LLMs for Solving Multi-step Problems: Preliminary Results

Vincent Hsiao[1], Morgan Fine-Morris[1], Mark Roberts[2], Leslie Smith[2], Laura M. Hiatt[2]

[1]NRC Posdoc, [2]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
vincent.hsiao.ctr@us.navy.mil, morgan.f.fine-morris.ctr@us.navy.mil, mark.c.roberts20.civ@us.navy.mil,
leslie.n.smith20.civ@us.navy.mil, laura.m.hiatt.civ@us.navy.mil

Abstract

Large Language Models (LLMs) have shown impressive performance on various language processing tasks, but often struggle with complex, multi-step tasks such as travel planning. To address this challenge, extensions like LLM-modulo systems and agentic approaches have been proposed, each with its own strengths and limitations. This paper examines the unique strengths and limitations of these approaches, using the Travel Planner benchmark as a case study. We analyze the results and propose a new hybrid task planner approach to address the challenges of solving multi-step tasks with LLMs, highlighting implications for future research in this area.

1 Introduction

Large Language Models (LLMs) have emerged as a powerful tool for solving a variety of complex tasks. Currently, there is a growing interest in applying LLMs towards solving multi-step tasks such as travel planning or mission planning. However, LLMs have frequently shown disappointing results on these multi-step tasks. In benchmarks such as Travel Planner (Xie et al. 2024), even GPT-4 only achieves a 0.6% success rate for solving a user request.

To improve the performance of LLMs on multi-step tasks, extensions to LLMs have been proposed such as LLM-modulo systems (Kambhampati et al. 2024b; Gundawar et al. 2024) and other agentic approaches (Yao et al. 2022; Fourney et al. 2024). Many of these approaches couple LLM(s) with external verifiers in order to provide guarantees on their accuracy. In this paper, we explore some of the challenges encountered by these approaches. Using the Travel Planner benchmark, we find that many LLM approaches lack an ability to generate a model of the problem they are solving and are unable to do planning about the solving process. Without explicit human guidance, this deficiency reduces the likelihood that LLM can successfully complete a given multi-step task.

To address this, we propose a new approach that provides additional planning guidance to the LLM. Our approach separates the task into Planning and Acting phases that mirror recent developments in planning (e.g., (Ghallab, Nau, and Traverso 2016)(Ghallab, Nau, and Traverso to appear)[1]):

- *Planning phase*: A Hierarchical Task Network planner provides guidance, via a plan, to an LLM that describes *what* needs to be done.
- *Acting phase*: An agentic LLM operationalizes the plan by determining *how* to accomplish each action in it.

Together, these approaches limit the kinds of hallucination errors that an LLM typically produces.

We present preliminary results of our new approach showing that it more than doubles the success rate of an agentic LLM system in completing the initial steps of the Travel Planning task. We then provide a detailed analysis of the results and discuss the implications for future research in this area.

2 Related Work

LLMs for planning: There is a large debate on whether LLMs are capable of classical planning tasks. There are many papers that directly apply LLMs toward planning tasks with varied success (Silver et al. 2024). There also exist survey papers of LLM-based agents planning that categorizing recent works into Task Decomposition, Plan Selection, External Module, Reflection, and Memory (Huang et al. 2024). On the other hand, there are also many other papers that suggest that LLMs lack the capability to plan (Verma, Bhambri, and Kambhampati 2024; Kambhampati 2024).

Beyond applying LLMs directly to planning, there have also been indirect approaches that couple an LLM with a classical planner in a similar manner to the HaoTP approach (Hao et al. 2024) discussed in this paper. One approach is to have the LLM translate a given planning problem into a PDDL specification and then use a classical planner to generate a solution (Liu et al. 2023). Other than this end-to-end approach, LLMs have been applied to many domains to generate various planning representations such as temporal logic (Chen et al. 2023), task decompositions (Zhang et al. 2021), and PDDL (Guan et al. 2023).

Reasoning with LLMs: The advent of ChatGPT-o1 model changes the landscape of LLM reasoning capabilities. (Valmeekam et al. 2024) evaluate the planning capabilities of the o1-preview and o1-mini models on both planning and scheduling benchmarks. The "LLM as a Mastermind" paper (Zhang et al. 2024a) surveys the use of Large Language Models (LLMs) in strategic reasoning, a form of

[1]Preprints available at https://projects.laas.fr/planning/

reasoning requiring understanding and predicting adversary actions in multi-agent settings, highlighting scopes, applications, methodologies, and evaluation metrics.

Agentic LLMs: Agentic LLMs are a compound or hybrid system to accomplish tasks. There has been growing interest in incorporating LLMs into agentic workflows. One of the first approaches in this area was the ReAct framework (Yao et al. 2022) where agent reasoning choices are interleaved with action selection in order to interact with external environments. Another paper introduced ExpeL (Zhao et al. 2024), an agent that learns from experiences and natural language to make informed decisions without requiring parametric updates, addressing the challenges of resource-intensive fine-tuning and limited access to state-of-the-art LLMs.

LLM Tool Use: To supplement capabilities in domains where LLMs have poor performance, it is common to allow LLMs to interface with external tools. These tools enhance the LLMs with capabilities that they might otherwise not have (Wang et al. 2024) and there are numerous papers covering various aspects of this field of research (Huang et al. 2024; Li et al. 2024).

LLMs and memory: One way to enhance the performance of language models on domain-specific tasks is known as Retrieval-Augmented Generation (RAG) (Izacard et al. 2023; Lewis et al. 2020). In this approach, the LLM is combined with a non-parametric memory index (e.g. a vector index) and retrieved data is fed back into the LLM context when needed. There are various mechanisms for enhancing LLMs with memory (Zhang et al. 2024b) including the notetaking method described in this paper.

3 Multi-step Tasks

Before we discuss how to solve multi-task problems, we need to describe them generally, and travel planner specifically. A multi-step task differs from a planning problem in classical planning.

In classical planning, a planning problem consists of a description (or model) of the problem, an initial condition of the world, a set of actions that can be used to change the state of the world, and a goal condition. In contrast, in a multi-step task, we assume that the problem description (or model) is not directly given (or fully provided) to the system. Instead, portions of the problem description are located in external databases. A solution to a multi-step task includes actions taken (e.g. API requests) to acquire necessary external data.

In the case of the Travel Planner benchmark, we can formulate a travel planning problem as multi-step task that contains a planning problem. Specifically, the Travel Planner task can be broken up into two model building problems (information gathering and external tool use) followed by a classical planning problem:

1. Information Gathering: The first step of the task is to gather information such as the user request, the constraints that need to be satisfied, and what tools are necessary and how to use them.

```
{
"org": "Detroit",
"dest": "San Diego",
"days": 3,
"visiting_city_number": 1,
"date": ["2022-03-05", "2022-03-06", "2022-03-07"],
"people_number": 1,
"budget": 3000
}
```

Figure 1: Example JSON translation of an NL request

2. External Tool Use: The second step is to determine what calls to the external tools/APIs are necessary to solve a given user request. We assume that there are a set of external tools that are necessary to interact with to solve the problem (e.g. a flight search tool). The LLM could choose not to obey this step, but then it would be impossible to solve the problem.
3. Planning: After obtaining the data from external tools, the next step of the task is to combine the data into a viable travel plan.

In Xie et al. (2024), GPT4 was shown to achieve a 0.6% success rate on the benchmark when applied to the travel planner task as an end-to-end system. In Hao et al. (2024), a success rate of 93.9% (with limitations on its generalizability) was achieved when the LLM was used to solve the external tool use sub-problem and the z3 library was used to solve the planning sub-problem.

In this paper, we will investigate the challenges of designing an LLM system that can solve a multi-step task. A system designed to solve multi-step tasks should satisfy the following criteria:

1. Generalizability: The system should not require the existence of a pre-built solution.
2. End-to-end: The system should solve both the model building and planning subproblems of a multi-step task.

We will first study the system from Hao et al. (2024), which we will term HaoTP to understand what parts of a multi-step task LLMs are currently able to perform. HaoTP does not satisfy the two aforementioned criteria, so following this initial study, we will propose an agentic system designed as an end-to-end solution for multi-step tasks. For our implementations, we will employ Codestral-22b as the LLM.

4 LLMs for constraint models

To start off, we give a brief description of the approach from Hao et al. (2024) with respect to each of the three subproblems in Travel Planner.

1. Information Gathering: The first step of the approach is to take a Natural Language (NL) request and convert the request into a JavaScript Object Notation (JSON) request. An example output of this step can be seen in Fig. 1 which is the output of the system for a travel planning request for a trip from Detroit to San Diego from March 5th to Marth 7th, 2022 on a budget of $3,000.

§	Method	Process Step: Information Gathering	Tool Use	Planning	Implementation
	End-to-End ReAct	LLM	LLM	LLM	(Xie et al. 2024)
4	LLM for constraint models	LLM	Few shot LLM	SMT solver	(Hao et al. 2024)
5	Agentic LLM system	LLM	LLM via Python	LLM via Python	This paper
6	Task Planner LLM	Guided	LLM via Python	LLM via Python	”

Table 1: LLM approaches and their solutions for Travel Planner subtasks

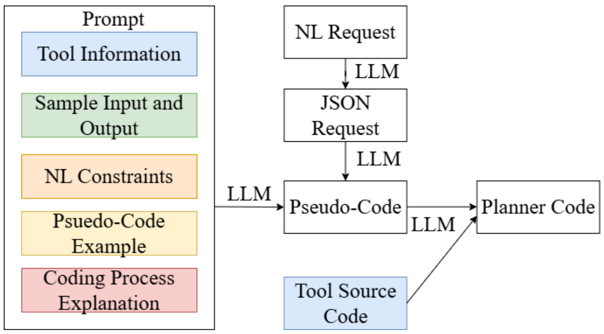

Figure 2: Forward pass LLM system for Travel Planning

2. External Tool Use: The second step of the pipeline (JSON-Steps) involves feeding the JSON request from the previous step back into the LLM and asking for ”steps” or pseudo-code. The prompt for this step includes several examples of the pseudo-code. In this case, the tool use problem is selecting the correct python statements to make calls to the tool APIs.

   The third step of the pipeline (Steps-Code) is also part of the tool use task where pseudo-code from the previous step is translated into actual python code that defines a constraint model using the z3 library. Like in the (JSON-Steps) part of the pipeline, examples of python code are provided for the LLM to use as few-shot examples.

3. Planning: After the final python code is written, the entire file is executed, which calls an SMT solver (the z3 library) that returns a final solution to the travel planning request.

The pipeline can be seen in Fig. 2 where the LLM is used to generate each of the white boxes in the flow that start from the NL request.

4.1 Generalizability

Kambhampati et al. (2024a) found that LLMs are capable of implementing functions for checking the validity of individual constraints. Thus, it may be possible to directly generate the code or the constraints for an SMT solver without relying on prior examples. To test this hypothesis, we attempt to replicate the system from Hao et al. (2024) but without the few shot examples that are provided during the tool use task. For example, instead of an explicit example of the pseudo-code we want to generate, we provide an explanation of the coding process as in Fig. 3.

Fig. 4 gives an example where this replication is successful. In the example, a flight time preference for flights that depart before 5PM is added to the original request. We can observe in the generated code that the LLM correctly adds additional constraints so that the chosen flights do not have DepTime after 1700 (5PM). In this case, the LLM is capable of adding simple constraints without any prior code examples. However, while testing this more general approach, we ran into several issues.

Lack of a world model One of the issues we noticed when we implemented this approach is that the LLM often overlooked certain constraints, particularly commonsense spatio-temporal ones that are implicit in the TravelPlanner benchmark. For example, consider a travel plan that involves traveling from city A to city B on day 1 with a flight from 10 AM to 2 PM. In this scenario, the LLM should logically infer that breakfast should take place in city A and dinner in city B. However, this type of spatio-temporal reasoning can be challenging for LLMs, and often they fail to maintain this constraint, even when explicitly instructed to do so.

A valid solution to a spatio-temporal constraint is quite complex compared to the free-form generated constraint in Fig. 4. To properly encode the constraint, it is necessary to calculate a temporal model of where the travel group is during the trip and the LLM is unable to reason that producing this model is necessary.

Higher order (meta) considerations The implementation of a constraint model can have a dramatic impact on the tractability of solving the model. In our experiments, we observed that the LLM does not take this principle into account at all. In Hao et al. (2024), the solution is constructed with backbone code such that for a travel plan with a destination of multiple cities, there is a loop that iterates through possible solution city selections (and will exit if one of these city selections results in a satisfiable problem). However, without this guidance, the LLM will choose to write a brute force constraint model where city assignments are also included as variables to solve for. The resulting model can easily end up being intractable.

Repair looping It is reasonable to assume that the code generated by an LLM does not compile or has runtime issues during the first generation pass (as is the case in most code that humans write). This could be due to a variety of reasons (such as hallucinating function names). We can ask the LLM to fix this issue using an LLM-modulo approach (Kambhampati et al. 2024a) where the code that is generated is then executed and the error message (if any) is then sent as a

```
The general outline for each section is:
1. Get data from tool if necessary.
2. Create z3 Arrays for necessary data column(s) from tool.
3. Create z3 Int variables to index the data.
4. Add constraints discussed above (there can be other constraints even if local constraint is null).
5. Calculate quantities needed for later sections.
```

Figure 3: An example of coding process instructions that would be given to an LLM for how to code a solution to the Travel Planning.

```python
# add constraint: no evening flights after 5PM
for i, flight in enumerate(flights_detroit_to_sandiego['DepTime'].values):
    s.add(If(variables['flight_index_0'] == i, Int(flight.split(':')[0]) < 17, True))
for i, flight in enumerate(flights_sandiego_to_detroit['DepTime'].values):
    s.add(If(variables['flight_index_2'] == i, Int(flight.split(':')[0]) < 17, True))
```

Figure 4: An example of free form coding. The LLM is allowed the flexibility to generate arbitrary code to write constraints for unseen requests preferences (in this case for evening flights before 5PM)

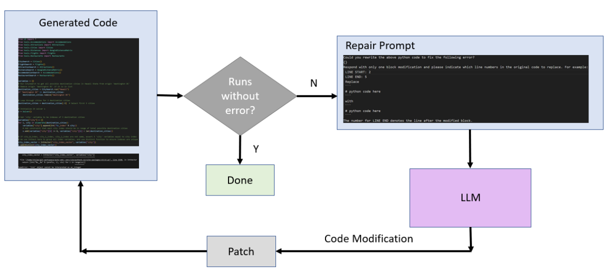

Figure 5: Example LLM-modulo approach for code-repair.

back-prompt to the LLM to fix any code generation issues (see Fig. 5).

This process can be iterated until the code can pass through the code verifier. We have observed that this LLM-modulo approach is capable of fixing small errors and can sometimes address the problem of hallucinating functions. However, this approach does not work all of the time. Occasionally, the LLM will modify an error in generated code into another error and then in the subsequent pass modifying it back into the initial error. This creates a repair loop where the generated code will alternate between two erroneous solutions.

Although one can imagine that simply extending the context of the LLM may solve this issue, there is no reason to assume that the length of the repair loop has any limit. In more complex code, it could be the case that changing code in one location breaks functionality in a different location, and an iterative repair approach could loop through the steps of this cycle endlessly.

5 Agentic LLMs

To design an end-to-end system to solve a multi-step task, we will investigate agentic LLM approaches.

In this paper, we define agentic LLMs as systems that are designed to interact with external tools (such as Wikipedia

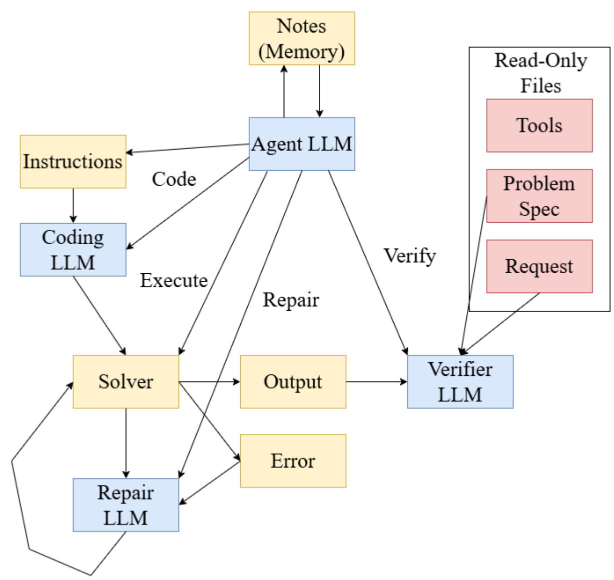

Figure 6: Simple agentic LLM approach

or other local databases) and incorporate LLMs as drivers of acting, planning, and/or evaluation; external here references anything that is external to the LLM model. For this section, we will study the capabilities and limitations of the agentic LLM system we developed that is displayed in Fig. 6. In this system, an agent LLM is responsible for giving orders to three other LLMs (coding, code repair, and verifier) in order to solve an overarching task. The agent LLM is prompted with a task specification and a list of actions and asked to choose an action. It is also allowed access to a note file which it can use to store short term memory. The goal of the agent LLM is to build a model of the problem in its short-term memory file and then use this model to instruct the other LLMs/system components to solve the problem. To the best of our knowledge, this is the first investigation into having the LLM determine by itself what to write down in a provided notes file.

For this system, we define the following actions:

1. Read: open and read any of the permissible files (red and yellow boxes in Fig. 6).
2. Write: overwrite the text in any of the write-able files (yellow boxes in Fig. 6)
3. Code: send a coding request for a secondary coding LLM to produce code according to the contents of instructions.txt.
4. Execute: execute the Python script written by the coding LLM to solver.py.
5. Repair: send a request to a code repair LLM to attempt to fix the current working Python script.
6. Verify: send a request to a verifier LLM to check if the solution provided in output.txt is valid.

At each iteration of the agentic system, the agent LLM will respond with one of the above actions which is then processed accordingly by the external environment.

5.1 Capabilities

Like with the system from (Hao et al. 2024), we provide a description of how this agentic system solves each part of the Travel Planning problem.

1. Information Gathering: The first step of the approach is to read each of the relevant files in the files provided (e.g. problem specification, tool specification, user request) and summarize their contents in the notes file.
2. External Tool Use: The second step of the approach is to use the information recorded in the notes file and write python code for the tool calls. The LLM then needs to record the output of the tool calls for the subsequent planning step.
3. Planning: The last step is to solve the planning problem. The LLM can attempt to do this by itself (i.e. like in the ReAct approch) or can write a solver that plans over the data obtained from the tool calls in the previous step.

The agentic approach is capable of synthesizing information from multiple documents. An example of this capability can be seen in Fig. 8, where the LLM reads three independent files as its first three actions and subsequently summarizes the important points of these files in its short-term memory file.

The LLM is also capable of writing well-formed instructions for a coding LLM to execute as can be seen in Fig. 7. In the example shown in the figure, the LLM reads the Python source code for the Flights tool and correctly conveys the proper arguments to use the tool to the coding LLM.

5.2 Limitations

While this is certainly a step in the right direction, the agentic LLM is not consistent in producing these results. For example, instead of choosing to record relevant facts about the problem in its short-term memory as in Fig. 8, the LLM could choose to only record a history of its goals. Although this record can be helpful for assisting the LLM in planning its next course of action, it is not helpful for solving the actual problem.

Another limitation is that the agent LLM does not always give the coding LLM enough information to work with. For example, the agent LLM could send an instruction "use the Flights tool to search for flights from Detroit to San Diego" without specifying how exactly to use the tool.

Unknown Unknowns A major recurring theme is that LLMs are unable to determine what information they lack when attempting to perform a task. In the dialogue between the agent LLM and the coding LLM, it is common that the agent LLM does not know that the coding LLM does not know how to use a given tool. Furthermore, it is even difficult to get a coding LLM to determine that it does not have enough information to provide a solution. This is true even if we directly prompt the coding LLM to reject coding requests that do not provide enough information. Instead of rejecting the instruction given by the agent LLM, it is more common for the coding LLM to hallucinate an answer to the instruction such as writing code to call erroneous functions (e.g. calling an erroneous `search_flights` method instead of the correct `run` method).

6 Task Planner LLM

We propose that a recurring theme throughout the limitations that we have discussed so far is that LLMs lack what we would term a recipe for solving multi-step problems. Beyond the direct problem of planning a given travel itinerary, we must also consider how the LLM chooses to solve a given problem.

In the hybrid approach from Hao et al. (2024), a major limitation is that the LLMs follow a strict structure for what they are allowed to generate. To support this, significant human effort is required to engineer a structural backbone which can make these two approaches impractical in many situations. In this case the LLM does not have a choice of how to solve the problem. This choice was already provided by the supporting code and prompts.

In our experiments on modifications to HaoTP, there is less of a supporting structure which allows for less human input required as well as greater solution creativity but also increases the likelihood of issues such as constraint blindness. Without knowing the exact steps to take to solve the

````
Use the Flights class to search for flights from the origin city to the destination city on the departure date.
- The Flights class provides a 'run' method that takes three arguments: origin, destination, and departure_date.
- Use the 'run' method to search for flights based on the traveler's preferences.
- For example, to search for flights from Detroit to San Diego on March 5, 2022, use the following code:
```python
flights = Flights()
results = flights.run("Detroit", "San Diego", "2022-03-05")
```
- The results will be a DataFrame containing information about the flights, such as flight number, price, departure time, arrival time, and
distance.
- If no flights are found, the results will be a message saying so.
````

Figure 7: Agentic LLM instruction writing capability

```
Action number 0: ['Read files/request.txt']
Action number 1: ['Read files/problem_specification.txt']
Action number 2: ['Read files/tool_specification.txt']
 User Preferences:
 - Solo traveller
 - Destination: Detroit to San Diego
 - Duration: 3 days (March 5th to March 7th, 2022)
 - Budget: $3,000
 - Preference: No evening flights departing after 5PM
 - Problem specification constraints:
   - Destination cities should not be repeated
   - Three restaurants per day (breakfast, lunch, dinner)
   - All specified cuisine types must be visited
   - Attraction choices should not be repeated
   - All specified attraction types must be visited
   - Accommodation minimum nights constraint
   - Budget constraint
 - Tools available:
```

Figure 8: Agentic LLM information gathering capability. The agent LLM correctly decides to view the request, problem specification, and tool specification files and then generates a summary of their contents for later use (tool notes omitted for length).

problem, the LLM can miss important constraints such as spatio-temporal consistency.

In the agentic approach, there is the least supporting structure, but in return there is the greatest chance of process derailment. The LLM does not commit to a consistent process for gathering information, storing it, and conveying it. Consequently, it is easy for such systems to get stuck or hallucinate answers that they have not yet obtained sufficient information to provide.

6.1 HTN Planning

To address the issues, we will combine the agentic LLM approach with a task planning subsystem. Specifically, we will employ a component based on Hierarchical Task Network (HTN) planning.

Formally, we define an HTN planning domain as (S, A, T, M) where S is the possible states of the world, A is a set of primitive tasks, T is a set of compound tasks, and M is a set of decomposition methods that describe how a task can be decomposed into subtasks. Specifically, a method $m \in M$ is $(t_m \in T, pre_m, sub_m)$, where t_m is the task to be decomposed, pre_m is a list of preconditions that specifies whether a method is applicable given a state s and sub_m is a (possibly ordered) list of subtasks (primitive or compound). Planning proceeds by using the methods to recursively decompose tasks into smaller and smaller subtasks until primitive tasks are reached.

Using Fig. 10 as our task network, we can look at the method for decomposing choose flights:

- Task name: the task to decompose is `choose flights`
- Precondition: the preconditions are that the origin, destination(s), and dates of the travel plan are present in the notes.
- Subtasks: `understand flights tool`, `choose departing flight`, `choose returning flight`

In our planning domain, the state of the world consists of the contents of the files in the agentic LLM system (e.g. red and orange boxes Fig. 6). The preconditions and effects consist of arbitrary natural language statements (and their conjunctions, disjunctions, negations, etc.). To determine whether a given effect is satisfied, we will use an LLM verifier LLM evaluate the state of the world (e.g. the system files) with respect to the given effect.

6.2 A task planning subsystem

The task planning subsystem will contain a repository of recipes (i.e. Hierarchical Task Networks) that describe how to solve problems. We leverage the two-pronged planning and acting approach of Ghallab, Nau, and Traverso (2016, to appear). The task planner creates a set of descriptive actions of *what* the LLM needs to do, as shown in Figure 10. At each iteration of the system, the LLM uses these descriptions to decide *how* to do the action; we use the approach from §5.

The general flow of this subsystem can be seen in Fig. 9. The agent LLM sends a verification request to the verifier LLM, which checks the completion of the current task with respect to the current state of the file system. If the check is successful, the current task is updated and the result of the verification request is sent back to the agent LLM.

For our subsystem we will use a hierarchical task network (HTN) to represent the recipes for how to solve problems. An example of such a task network for the Travel Planner problem can be seen in Fig. 10. The HTN breaks down the

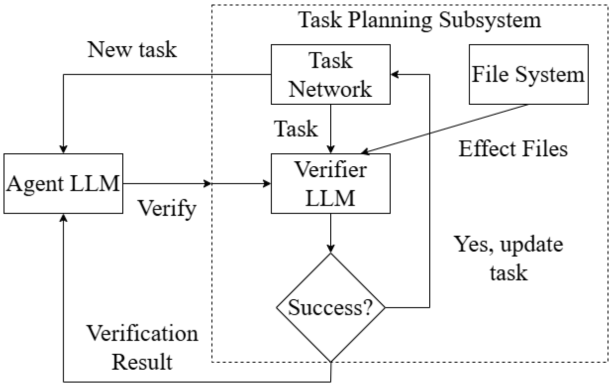

Figure 9: Task planning subsystem

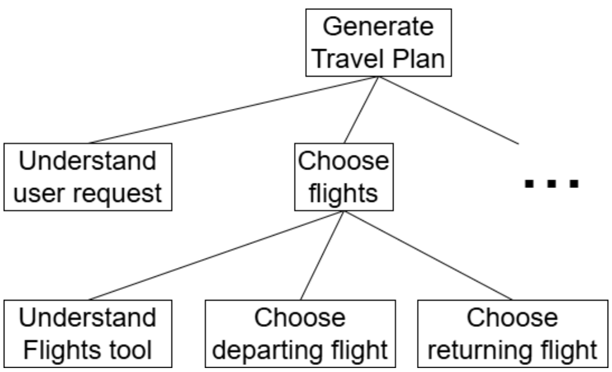

Figure 10: An example task network for the Travel Planner problem

overarching planning task into smaller subtasks for the LLM to follow.

As an example, consider the `understand user request` task that the LLM needs to accomplish at the start of generating a travel plan. To successfully complete this task, the LLM needs to read the file that has the relevant user request and take notes on any information needed to generate a travel plan. A representation of this task node in JSON format can be seen in Fig. 11.

During the verification request, the verifier LLM is asked whether the information contained in the effect files satisfy the intended effect of the task. In this example, the intended effect of the `understand user request` task is that

```
"task": {
    "name": "understand user request",
    "effect": "notes contains same details as request and contains
               origin, destination, departure date and returning date
               for the trip",
    "effect files": {
        "file1": "notes.txt",
        "file2": "request.txt"
    }
},
```

Figure 11: Example of a primitive task node

the agent LLM's short term memory file contains information that matches the user request file.

For this work, we employ an LLM as the primary verifier, but we could also use external verifiers with guarantees for certain subtasks. The primary advantage of this setup is that the task network can guide the LLM towards a consistent process when solving problems.

6.3 Preliminary Results

To study the effectiveness of this additional task network component, we perform a ablation study on a modified version of the Travel Planner task which includes the information gathering step not evaluated in past work. The general task is as follows:

- Direct Prompt Information: The LLM agent is provided information about a list of accessible files and what their overarching goal is (e.g., to generate a travel plan) but none of the specific details of how to accomplish it.
- Accessible Information: The LLM can access descriptions and the source code of various tools (e.g., FlightSearch) in the list of accessible files. We emphasize that this information is *not* provided in the prompt but only in files that the LLM can choose to read.
- Metrics: Since it is computationally intensive to generate a full travel plan using an agentic LLM architecture, we limit the maximum number of system steps (where each step is a prompt to the agent LLM for an action to take) to 20. To evaluate the methods, we check whether the LLMs have successfully booked departing and returning flights between the origin and destination(s) in the user request.

To simplify our verification, we use requests from the train set from the Travel Planner benchmark that are coupled with ground truth annotated plans. The results on this limited set can be seen in Fig. 12.

Using the task planner helps avoid a common kind of hallucinations and eliminates the need for repair loops.

For example, the LLM needs a specification of what a tool does in its short term memory before trying to write code for that tool. Adding an explicit task to do this allows the task planner to facilitate such knowledge retrieval (rather than rely on the LLM), which reduces the occurrence of function name hallucinations, such as calling an erroneous `search_flights` method instead of the correct `run` method. The reduction of these hallucinations in turn reduces the probability that the agentic LLM initializes an infinite repair loop which will prevent it from making further progress.

7 On LLMs and Code Safety

In our experiments with agentic LLMs, one concerning behavior we noticed was execution of out-of-scope code in Python. To sandbox our agent LLM, we set up the environment that processes agent commands to allow the LLM to only read and write to a pre-specified list of approved files (this list is also given to the LLM). However, we did not limit the corresponding code generation capabilities (to allow for

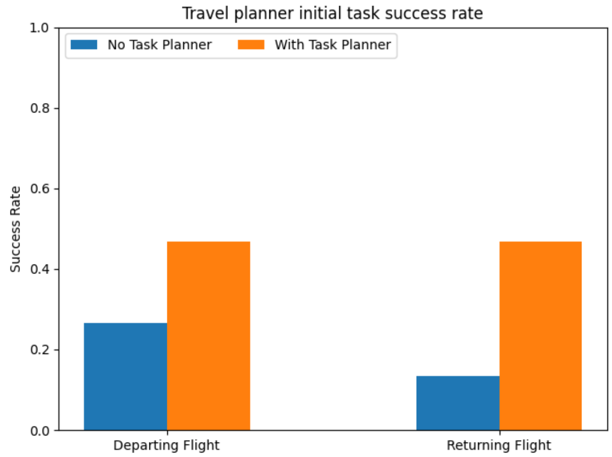

Figure 12: Success rate of LLM, with and without a task planner, on the first steps of solving a task in the travel planner benchmark.

greater problem solving creativity). This lack of limitations led to two critical unseen side effects:

- **Attempted code installation**: In one instance, our agentic LLM generated code that depended on python libraries that were not installed in the development environment it was running in. Instead of changing the code to some supported library, it attempted to programmatically install the python library within the python script being written.
- **File writing bypass**: In several instances, in order to save the results from tool calls, the agent LLM will tell the coder LLM to save the retrieved data into a local text file. This file may not necessarily be on the list of approved files.

It is important to emphasize that prompt restrictions are not a sufficient guard against unintended agent behavior. In particular, the prompt for the agent LLM specifically mentions that it only has read/write access to the pre-approved file list and the prompt for the coding LLM includes an instruction to ignore requests to write to any other file.

8 Conclusion and Future Work

In conclusion, we have discussed several distinct approaches to utilizing Large Language Models (LLMs) for various tasks. Our findings indicate that while LLMs can be successfully employed as form fillers and code editors (e.g., Hao et al. (2024)), their reliance on an existing codebase limits their applicability to other domains. The use of LLMs for free-form coding offers flexibility in handling user requests without a pre-existing codebase, but this flexibility also introduces challenges. Adjusting the structure of the response addresses some of these issues, but LLMs can still generate erroneous results.

A large number of errors when using an LLM for planning seem to be a failure to incorporate a world model into formulating problem solving process. Some problem solving paths for a given multi-step task are not sufficient or efficient computationally. To address these challenges, we propose an hybrid task planner agentic LLM system that interacts with external information gathering tools to construct a model of the task to be solved. The steps that the LLM takes are guided via planning on a Hierarchical Task Network. The task planning constrains the possible problem solving paths to be more limited thus increasing the likelihood that the LLM can successfully complete the multi-step task. By standardizing these steps and adding intermediate verification, we reduce several common errors such as hallucinating tool types or data that has not yet been seen. We plan to do more work on benchmarking the behavior of these hybrid systems on more domains and with stronger models (beyond Codestral-22b), especially on their ability to prevent error cascading that is frequently present in multi-step tasks.

The task networks in the hybrid task planner system are currently handwritten by a human. This can run into similar issues with portability as the approach from Hao et al. (2024). As mentioned in the related work section, there is a great deal of work for generating planning models such as PDDL from text descriptions using LLMs. It is conceivable that given documentation on the processes to solve multi-step problems such as travel planner, an LLM could be used to generate these task networks. To increase the effectiveness this approach, we can borrow techniques from curriculum learning where the LLM can first start off by building task networks for simple tasks before gradually building up to more complex and longer tasks (e.g., growing the task network node by node). There are also alternative approaches such as LLM-based optimization which could be used to optimize over candidate task networks.

Acknowledgements

We thank NRL for funding this research.

References

Chen, Y.; Gandhi, R.; Zhang, Y.; and Fan, C. 2023. Nl2tl: Transforming natural languages to temporal logics using large language models. *arXiv preprint arXiv:2305.07766*.

Fourney, A.; Bansal, G.; Mozannar, H.; Tan, C.; Salinas, E.; Niedtner, F.; Proebsting, G.; Bassman, G.; Gerrits, J.; Alber, J.; et al. 2024. Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks. *arXiv preprint arXiv:2411.04468*.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Ghallab, M.; Nau, D.; and Traverso, P. to appear. *Acting, Planning and Learning*. Cambridge University Press.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2023. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36: 79081–79094.

Gundawar, A.; Verma, M.; Guan, L.; Valmeekam, K.; Bhambri, S.; and Kambhampati, S. 2024. Robust Planning with LLM-Modulo Framework: Case Study in Travel Planning. *arXiv preprint arXiv:2405.20625*.

Hao, Y.; Chen, Y.; Zhang, Y.; and Fan, C. 2024. Large Language Models Can Plan Your Travels Rigorously with Formal Verification Tools.

Huang, X.; Liu, W.; Chen, X.; Wang, X.; Wang, H.; Lian, D.; Wang, Y.; Tang, R.; and Chen, E. 2024. Understanding the planning of LLM agents: A survey. *arXiv preprint arXiv:2402.02716*.

Izacard, G.; Lewis, P.; Lomeli, M.; Hosseini, L.; Petroni, F.; Schick, T.; Dwivedi-Yu, J.; Joulin, A.; Riedel, S.; and Grave, E. 2023. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research*, 24(251): 1–43.

Kambhampati, S. 2024. Can large language models reason and plan? *Annals of the New York Academy of Sciences*, 1534(1): 15–18.

Kambhampati, S.; Valmeekam, K.; Guan, L.; Stechly, K.; Verma, M.; Bhambri, S.; Saldyt, L.; and Murthy, A. 2024a. LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. *arXiv preprint arXiv:2402.01817*.

Kambhampati, S.; Valmeekam, K.; Guan, L.; Verma, M.; Stechly, K.; Bhambri, S.; Saldyt, L. P.; and Murthy, A. B. 2024b. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *Forty-first International Conference on Machine Learning*.

Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.-t.; Rocktäschel, T.; et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33: 9459–9474.

Li, X.; Wang, S.; Zeng, S.; Wu, Y.; and Yang, Y. 2024. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1): 9.

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*.

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J. B.; Kaelbling, L.; and Katz, M. 2024. Generalized planning in pddl domains with pretrained large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 20256–20264.

Valmeekam, K.; Stechly, K.; Gundawar, A.; and Kambhampati, S. 2024. Planning in Strawberry Fields: Evaluating and Improving the Planning and Scheduling Capabilities of LRM o1. *arXiv preprint arXiv:2410.02162*.

Verma, M.; Bhambri, S.; and Kambhampati, S. 2024. On the Brittle Foundations of ReAct Prompting for Agentic Large Language Models. *arXiv preprint arXiv:2405.13966*.

Wang, Z.; Cheng, Z.; Zhu, H.; Fried, D.; and Neubig, G. 2024. What are tools anyway? a survey from the language model perspective. *arXiv preprint arXiv:2403.15452*.

Xie, J.; Zhang, K.; Chen, J.; Zhu, T.; Lou, R.; Tian, Y.; Xiao, Y.; and Su, Y. 2024. TravelPlanner: A Benchmark for Real-World Planning with Language Agents.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Zhang, Y.; Jauhar, S. K.; Kiseleva, J.; White, R.; and Roth, D. 2021. Learning to decompose and organize complex tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2726–2735.

Zhang, Y.; Mao, S.; Ge, T.; Wang, X.; de Wynter, A.; Xia, Y.; Wu, W.; Song, T.; Lan, M.; and Wei, F. 2024a. LLM as a Mastermind: A Survey of Strategic Reasoning with Large Language Models. *arXiv preprint arXiv:2404.01230*.

Zhang, Z.; Bo, X.; Ma, C.; Li, R.; Chen, X.; Dai, Q.; Zhu, J.; Dong, Z.; and Wen, J.-R. 2024b. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2404.13501*.

Zhao, A.; Huang, D.; Xu, Q.; Lin, M.; Liu, Y.-J.; and Huang, G. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 19632–19642.

A Additional Figures

```
# Add constraints: restaurants must be accessible based on arrival times and city list (assume restaurants are open 24/7).
# This constraint is already satisfied as we are searching for restaurants in the city where we arrive.
# If there are additional constraints, please specify them.
```

Figure A.1: LLM constraint generation error: the top comment line is the pseudo-code input and bottom two comment lines are the LLM's response.

```
Sure, the error is because you're trying to call the `evaluate` method on an `ArithRef` object, which is not correct. Since `m.evaluate
(city_0_index).as_long()` will return an `ArithRef` object which stands for an arithmetic expression in Z3 solver, it does not have direct
methods like `evaluate`. Instead, you need to get the value by using the `eval` method of the model `m` and then get the `as_long` value from
the result.

Here is the modification you need to make:

Replace:
```
destinations_for_flights = [destinations[m.evaluate(city_0_index).as_long()], destinations[m.evaluate(city_1_index).as_long()], destinations[m.
evaluate(city_1_index).as_long()], destinations[m.evaluate(city_2_index).as_long()]]
```
with:
```
destinations_for_flights = [destinations[m.eval(city_0_index).as_long()], destinations[m.eval(city_1_index).as_long()], destinations[m.eval
(city_1_index).as_long()], destinations[m.eval(city_2_index).as_long()]]
```

LINE START: 86
LINE END: 87
```

```
Sure, the error is because you're trying to call the `eval` method on an `ArithRef` object, which doesn't have that attribute. Instead, you
should use the `as_long()` method to convert the `Int` object into a python integer.

Replace the line:

```python
destinations_for_flights = [destinations[m.eval(city_0_index).as_long()], destinations[m.eval(city_1_index).as_long()], destinations[m.eval
(city_1_index).as_long()], destinations[m.eval(city_2_index).as_long()]]
```

with:

```python
destinations_for_flights = [destinations[m.evaluate(city_0_index).as_long()], destinations[m.evaluate(city_1_index).as_long()], destinations[m.
evaluate(city_1_index).as_long()], destinations[m.evaluate(city_2_index).as_long()]]
```

Line Start: 86
Line End: 87
```

Figure A.2: LLM repair looping error. The LLM corrects the code from m.evaluate (green) to m.eval (yellow) (top) and then on the subsequent editing pass from m.eval (yellow) to m.evaluate (green) (bottom).

```
I'll need to understand the user's travel preferences, including the cities to visit, attractions, dining preferences, and accommodation
requirements. I'll also need to understand the constraints and requirements of the problem specification document.
I'll need to understand the constraints and requirements of the problem specification document, as well as the format and capabilities of the
tools provided in the tool specification document.
I'll need to understand the format and capabilities of the Flights tool before using it to find flights for the user's travel plan.
I'll need to understand the format and capabilities of the Accommodations tool before using it to find accommodations for the user's travel
plan.
I'll need to understand the format and capabilities of the Attractions tool before using it to find attractions for the user's travel plan.
I'll need to understand the format and capabilities of the Restaurants tool before using it to find restaurants for the user's travel plan.
I'll need to understand the format and capabilities of the Distances tool before using it to calculate distances for the user's travel plan.
```

Figure A.3: LLM generated goal history. Instead of generating a summary of document contents in its short term memory as in Fig. 8, the LLM could instead choose to populate its short term memory with a history of its goals.

```
You are a coding LLM tasked with following the instructions written to flies/instructions.txt that are
given by a leader LLM. The leader LLM does not know what information you need. Do not make any
assumptions on what information you need and ask for more information if you do not have it. For example,
if the leader LLM tells you to:
- Use the Flights tool to search for Flights
Then you should respond with:
***NOT ENOUGH INFORMATION***
Not enough information provided in files/instructions.txt, please specify import path, method parameters,
and method output format.
```

Figure A.4: Instructions for a coding LLM in the agentic system to reject instructions with insufficient information. We attempt to encourage the coding LLM to reject orders from the leader LLM in cases where it does not have enough information to generate correct code. However, this is not sufficient and the coding LLM can choose to follow the instructions regardless of this constraint.