

---

# Procedural Knowledge Improves Agentic LLM Workflows

---

**Vincent Hsiao**

NRC Postdoctoral Fellow, Naval Research Laboratory  
Washington, DC  
United States  
vincent.hsiao.ctr@us.navy.mil

**Mark Roberts**

Navy Center for Applied Research in AI, Naval Research Laboratory  
Washington, DC  
United States  
mark.c.roberts20.civ@us.navy.mil

**Leslie Smith**

Navy Center for Applied Research in AI, Naval Research Laboratory  
Washington, DC  
United States  
leslie.n.smith20.civ@us.navy.mil

November 12, 2025

**Abstract**

Large language models (LLMs) often struggle when performing agentic tasks without substantial tool support, prompt engineering, or fine tuning. Despite research showing that domain-dependent, procedural knowledge can dramatically increase planning efficiency, little work evaluates its potential for improving LLM performance on agentic tasks that may require implicit planning. We formalize, implement, and evaluate an agentic LLM workflow that leverages procedural knowledge in the form of a hierarchical task network (HTN). Empirical results of our implementation show that hand-coded HTNs can dramatically improve LLM performance on agentic tasks, and using HTNs can boost a 20b or 70b parameter LLM to outperform a much larger 120b parameter LLM baseline. Furthermore, LLM-created HTNs improve overall performance, though less so. The results suggest that leveraging expertise—from humans, documents, or LLMs—to curate procedural knowledge will become another important tool for improving LLM workflows.

**Keywords** Large Language Models, Agentic Systems, Task Networks

## 1 Introduction

Procedural knowledge generally describes a sequence for solving a problem and has played an important role in AI. For example, spanning nearly 50 years of automated planning research, procedural knowledge has been effective for areas such as decomposing abstract tasks into concrete actions (27; 2), control rules for search (1), learning macro operators (6), plan recognition (7), knowledge-based planning (18), and planning and acting (8), to name only a few. Procedural knowledge also plays a central role in

cognitive systems (17), where a specific module in the system manages and applies such knowledge to problem solving. Recently, procedural knowledge is used as a kind of meta-strategy to assist in solving complex problems (e.g., (3)).

While LLMs have shown impressive capability, they often perform poorly on planning problems (13), and, as we will show later, LLM performance generally degrades as task complexity increases. In complex agentic tasks, successful task completion often hinges on an implicit plan that specifies what actions to take and in what sequence to take them, a capability that current LLMs often fail to consistently complete correctly. Part of the gap is that LLMs are mostly constructed to predict the next token, while planning requires managing details that span different temporal horizons and different levels of abstraction. LLMs lack a clearly understood way to track the change that results from taking action in the world.

In contrast, this paper examines whether providing *explicit* procedural knowledge improves LLM performance on agentic tasks. To provide an intuitive example (formalized in §3), consider the task of booking a travel itinerary. Experienced travelers follow a sequence such as: book flights, book hotel, and reserve attractions or restaurants. The sequence might vary by circumstance, but its presence makes the problem much easier to solve. The sequence manages dependencies (e.g., confirm flights before booking the hotel) and each step narrows the details considered in subsequent steps. But using an abstract sequence requires a process that: 1) tracks progress on the overall procedure, 2) maintains state changes across steps, 3) takes actions to complete each step, 4) verifies each step is finished before proceeding.

When solving a travel problem like this, LLMs lack the capability to track these layers all at once, leading to inefficient or unsuccessful workflows. For example, an LLM might book a restaurant reservation at the destination before verifying flight timing, only to backtrack to address such foreseeable conflicts. By treating complex tasks as a series of independent decisions rather than attending to interconnected dependencies, LLMs struggle to leverage procedural (i.e., hierarchical, goal-oriented) knowledge.

In this paper, we address two questions: *Can we embed procedural knowledge in an LLM workflow?* and *Will procedural knowledge improve LLM performance?* We will show that the answer to both questions is Yes! To that end:

- We formalize the problem as an MDP and integrate procedural knowledge using Hierarchical Task Networks (HTNs), where abstract tasks decompose into totally ordered subtasks. An HTN ensures logical consistency and resource availability.
- We describe and implement a hybrid LLM-HTN system, ProcLLM, that uses HTNs in an agentic LLM workflow. Importantly, this workflow is compatible with any LLM.
- We evaluate the runtime, success, and number of cycles of ProcLLM on four benchmark problems, two from the literature and two synthetic problems. **For the problems and HTNs we studied, HTN knowledge always improves LLM performance, often significantly.**
- We demonstrate that smaller LLMs benefit more from this workflow and in some cases exceed the performance of much larger LLMs without HTN knowledge.
- Finally, we show that LLM-created HTNs improve performance, though the results are mixed compared to hand-coded HTNs.

Overall, these findings suggest that LLM workflows can benefit considerably from procedural knowledge, not only in overall performance but also in a reduced model size and improved response time for the same level of performance. Although our implementation uses an Agentic LLM workflow with HTN knowledge, the process is straightforward and results may eventually generalize to other LLM workflows, so we close with some possible future directions.

## 2 Related Work

**Agentic LLMs.** We define agentic LLM as a large language model framework that can act, use tools, and plan to autonomously accomplish tasks. Many problems cannot be solved in a single forward pass of an LLM, motivating workflow techniques such as prompt chaining (32), where the output of one prompt becomes the input to another. Early community systems like AutoGPT (34) showcased the potential of

chaining and memory-augmented reasoning for open-ended tasks, while frameworks such as LangChain provided modular infrastructure for building agentic pipelines around LLMs

One of the first academic approaches in this area was the ReAct framework (35), where agent reasoning choices are interleaved with action selection in order to interact with external environments. Another notable contribution was ExpeL (38), where an agent that learns from experiences and natural language to make informed decisions without requiring parametric updates.

Although there are many variations of this framework, for the purpose of this paper, we will use a framework similar to the one used by Reflexion (Fig. 2 in (23)). The Reflexion agent employs an LLM (denoted as the agent) and an environment that processes actions across multiple iterations. At each iteration, the agent receives an observation from the environment. This observation is combined with a prompt given to the LLM to output some action. The environment processes this action and then returns the observation for the next iteration.

Perhaps the closest to this paper is the concurrent work of Belcak and Molchanov (3). In their universal deep research (UDR) framework, a strategy compiler is used to take a research strategy from the user and compile it into a list of steps and corresponding outputs that a deep research tool needs to execute step by step. The structure is enforced by incorporating code comments and yield directives that specify the goal of each step and the intended outputs of the step. Our approach is more general, and we can hypothesize that there exists a task network that when applied to our system would replicate the behavior of UDR.

**LLMs and 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 (31) and there are numerous papers covering various aspects of this field of research (12; 20). Early demonstrations included Toolformer (21), which fine-tuned LLMs to invoke APIs from demonstrations, and MRKL (15), which integrated symbolic reasoning with tool-use.

Unlike past work, our system allows the LLM agent to write arbitrary code to call arbitrary APIs. This is different from works where the external tools are presented in a standardized in-context format to an LLM for usage (e.g., MCP servers). Allowing the LLM to write arbitrary code also gives more flexibility in problem solving (e.g., coding allows the LLM to embed tool calls in algorithms), which standardized tool formats do not permit.

**Multi-agent Systems.** The extension from single-agent to multi-agent frameworks has become an important line of research. In AgentOrchestra (37), an orchestrator agent creates a plan that is then executed by specialized sub-agents. Unlike our explicit task network representation, AgentOrchestra uses a simple list structure. SagaLLM (4) proposes a transactional system to handle multi-step planning in multi-agent LLM systems. Other works, such as CAMEL (19), demonstrated emergent coordination between specialized LLM agents by role-playing tasks and negotiating via natural language.

**Hybrid Systems.** Various hybrid systems have been developed for application to specific domains. For example, (11) provides a survey of LLM-based agentic recommenders that incorporate subsystems similar to planning and acting modules. AutoConcierge (36) employs LLMs to convert user requests into formalized representations to be used by a reasoner. This is also similar to the approach from (10), where user requests and API calls are translated into constraints for an SMT solver. Both approaches use LLMs to convert user data into formalized problem descriptions, which are then solved to provide a solution. This is different from our approach, which instead uses a formalized representation to assist the LLM in the problem-solving process.

Other than agentic LLM systems, there are many LLM-modulo systems that combine LLMs with external verifiers and or solvers (14). ISR-LLM (39) employs an LLM to translate planning problems in natural language into PDDL and then iteratively calls the external planner to verify LLM-generated plans. Beyond agentic LLM systems, many hybrid approaches combine LLMs with external verifiers or solvers (14). ISR-LLM (39), for instance, employs an LLM to translate planning problems in natural language into PDDL and then iteratively calls an external planner to verify and refine plans. Similarly, program synthesis approaches (5) use LLMs to generate candidate code that is executed and verified externally.

These hybrid formulations highlight a broader paradigm: using LLMs as flexible natural-language interfaces while delegating correctness and reliability to symbolic or algorithmic components.

**LLMs and Planning.** In challenging tasks, an agent may need to generate a plan before completing the task. Many researchers have investigated how LLMs can be used effectively for planning (26) (25) (14). There have been impressive gains in this area of research; for example, a recent notable example is by Verma et al. (30) which claims up to 94% planning accuracy using a fine-tuning approach paired with chain-of-thought reasoning. Despite these gains, planning remains a challenging task (24) (29) with many results showing that LLMs still have many weaknesses when applied to these problems. Furthermore, many approaches still require considerable scaffolding.

A consistent claim is that the effectiveness of LLM planning is reduced as task complexity increases. Kambhampati (13) argues that LLMs engage in universal approximate retrieval, where they rely on pattern matching rather than systematic reasoning. This suffices for generating immediate solutions or short inference chains, but can run into issues when task complexity increases, which can be seen in ACPBench (16) where the task of generating just the next action has a much higher success rate than the more complex tasks. This suggests that current LLMs are most effective at solving short tasks that do not need multi-step planning. A natural solution would be as follows: When the provided task is too complex for an LLM agent, decompose the task into subtasks for which the LLM is effective.

Unlike other works in this area, our focus is not directly on the ability of an LLM to produce a plan (i.e., a sequence of steps solving the planning task). Rather, in many tasks that agentic LLMs are applied to solve, a sort of meta-planning is a requisite process for solving complex problem instances. By decomposing a task into smaller subtasks, the meta-planning required will be less complex and the agentic system will be less impacted by weaknesses of current LLMs when encountering complex tasks.

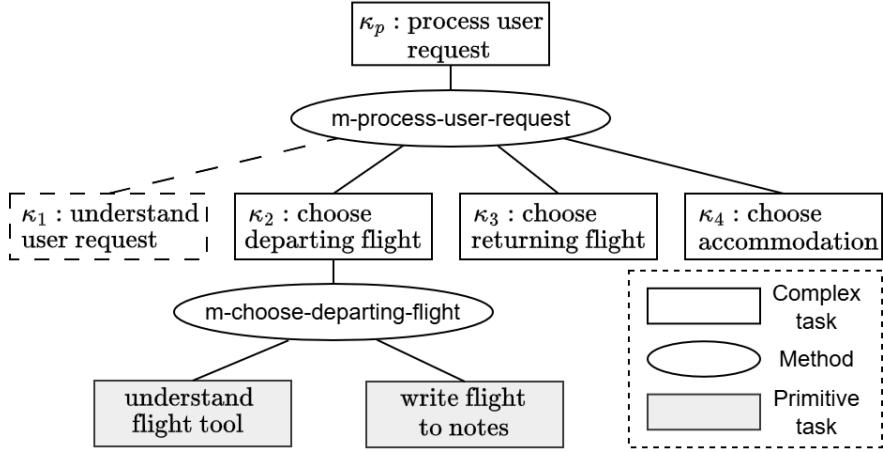
### 3 Preliminaries

We represent procedural knowledge using a Hierarchical Task Network (HTN), described next. We then formalize the problem as a Markov Decision Process (MDP) where the policy that solves the MDP is an LLM. Finally, we analyze why we expect ProcLLM to perform well.

**Hierarchical Task Networks (HTNs).** Because our focus is on how procedural knowledge improves LLM performance, our description of HTNs sacrifices exact formality for a more notional understanding. A full formal treatment can be found in (8). Let  $K$  be a set of tasks; a task  $\kappa \in K$  is a labeled name of something the system needs to do. Figure 1 (top) shows an HTN decomposition tree where  $K$  relates to the travel problem from §1. HTNs decompose complex tasks (white boxes) into primitive tasks (gray boxes) using methods (ovals). *Complex tasks* describe abstract tasks that need to be decomposed and *primitive tasks* describe concrete executable tasks. Figure 1 shows two methods. Method `m-process-user-request` decomposes  $\kappa_p$  into  $\langle \kappa_1, \dots, \kappa_4 \rangle$ , where the  $\kappa_1$  (dashed box) has passed the verify check (described below). Method `choose departing flight` decomposes  $\kappa_2$  into primitive tasks: `(understand flight tool, write flight to notes)`.

For a set of methods  $\mathcal{M}$ , a method  $m \in \mathcal{M}$  is a tuple  $(\text{head}, \text{task}, \text{pre}, \text{subtasks})$ , where the *head* is the name and any parameters of the method, *task* matches the taskname the method can decompose, *pre* is a (possibly empty) set of preconditions for applying the method in a state, and *subtasks* is a totally ordered sequence of tasks  $\langle \kappa_1, \kappa_2, \dots, \kappa_n \rangle$ . A method that matches a task is *relevant*, and HTN planners often select the first relevant method in  $\mathcal{M}$  (As described in §4, ProcLLM’s Algorithm 1, Line 24 uses `FindFirstRelevantMethod` to select methods).

**Markov Decision Process (MDP) Model .** We formalize agentic tasks as a finite-horizon deterministic Markov Decision Process (MDP), defined by the tuple  $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, r, H)$ , where  $\mathcal{S}$  is the set of states with a subset of absorbing terminal states,  $\mathcal{S}_{\text{term}} \subset \mathcal{S}$ ,  $\mathcal{A}$  is set of actions,  $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$  is a deterministic state transition function, where  $s_{i+1} = \mathcal{T}(s_i, a_i)$ ,  $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  is a reward function, and  $H$  is a finite horizon. An episode terminates either by reaching a terminal state or by exceeding the horizon (timeout).



```

1 {"m-travel-process-user-request": {
2     "task": "process user request",
3     "subtasks": {"1": "understand user request",
4                  "2": "choose a departing flight",
5                  "3": "choose a returning flight",
6                  "4": "choose an accommodation" },
7     "verify": "answer.txt contains a solution in the correct format that follows the problem specification",
8     "verify_files": {"file1": "problem_specification.txt",
9                      "file2": "answer.txt" },
10    "m-choose-departing-flight": {
11        "task": "choose a departing flight",
12        "subtasks": {"1": "understand flights tool",
13                     "2": "write flight to notes" },
14        "verify": "notes contains details about a flight number and cost for a flight from origin to destination on departure date consistent with request preferences",
15        "verify_files": {"file1": "files/notes.txt",
16                        "file2": "files/problem_specification.txt",
17                        "file3": "files/request.txt" } },
18    "m-understand-flights-tool": {
19        "name": "understand flights tool",
20        "effect": "notes contains details about flights tool methods inputs and outputs",
21        "effect_files": {"file1": "files/notes.txt",
22                         "file2": "files/tools_specification.txt" } }

```

Figure 1: Top: An in-progress HTN decomposition tree for the Travel Planner problem of §1. White boxes indicate abstract tasks, round ovals indicate methods that decompose tasks, and gray boxes indicate primitive tasks for the LLM to execute. The dashed box for  $\kappa_1$  indicates it has passed verify. Bottom: Notional methods for the same problem.

For our domains, we define a function  $\text{verify} : \mathcal{S} \rightarrow \{0, 1\}$ , that identifies terminal states (i.e.,  $\text{verify}(s) = 1$  if  $s \in \mathcal{S}_{\text{term}}$  and 0 otherwise). The reward function is then:

$$r(s, a) = \begin{cases} r_{\text{success}} & \text{if } \text{verify}(\mathcal{T}(s, a)) = 1 \quad (\text{a terminal state}) \\ r_{\text{step}} & \text{otherwise} \end{cases}$$

where  $r_{\text{success}} = 1$  is a positive reward for task completion and  $r_{\text{step}}$  is a negative reward for non-terminal steps (e.g.,  $r_{\text{step}} = -0.1$ ). Primitive tasks from the HTN are linked to  $\mathcal{S}_{\text{term}}$ , which can be verified using  $\text{verify}$ .

An agent's behavior is described by a stochastic policy  $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ , where  $\pi(s, a) = P(a_i = a | s_i = s)$  is the probability of selecting  $a$  in state  $s$ . A solution to  $\mathcal{M}$  is a policy  $\pi$  that maximizes the expected reward (in this case a policy that reaches a terminal state in the fewest steps).

**Initial Analysis.** In this work, we investigate the use of an agentic LLM system as the policy  $\pi$  that operates within this MDP framework. We show that divide-and-conquer mechanisms such as task decomposition can be used to constrain  $\pi$  to work on smaller problems, increasing the expected reward obtained by our agentic LLM.

The general MDP model allows us to hypothesize about why we expect this approach to work. We assume that there exists a task-aware assembly function  $\mathfrak{A} : \mathcal{S} \times K \rightarrow C$  and let  $C_i$  be the context produced from state  $s_i$  used as input to the LLM at each iteration.

**Task Success Rate.** A task succeeds if a terminal state  $s \in \mathcal{S}_{\text{term}}$  is reached within  $H$  steps. Let  $(a_0, \dots, a_{H-1})$  be a sequence of actions that arrives at a terminal state from an initial state  $s_0$ . Assuming deterministic transitions, the probability of taking these actions depends solely on the LLM's parameterization  $\theta$ , which governs action selection:

$$P_\theta(a_0, a_1, \dots, a_{H-1}) = \prod_{i=0}^{H-1} P_\theta(a_i | C_i)$$

Here,  $P_\theta(a_i | C_i)$  represents the LLM's conditional probability of selecting action  $a_i$  given context  $C_i$ . This factorization arises because each action depends only on the current context (Markov property).

**Task Decomposition.** A task decomposition splits a problem into a sequence of smaller subtasks,  $(\kappa_0, \dots, \kappa_{n-1})$ . Let  $P_\theta(\kappa_j)$  be the probability of successfully completing subtask  $j$ . Decomposition improves task success if:

$$P_\theta(\text{full task}) < \prod_{j=0}^{n-1} P_\theta(\kappa_j)$$

We hypothesize that we can improve the right side of this equation through contextual scaffolding. We assume that the LLM's ability to complete a subtask  $P_\theta(\kappa_j)$  increases by reducing context complexity. When the agent works on a specific subtask  $\kappa_j$ , the assembly function  $\mathfrak{A}(s_i, \kappa_j)$  generates a focused context  $C'_{i,j}$ . By constraining the problem space through reducing context complexity, the conditional probability of selecting a correct action  $a_i$  should increase:

$$P_\theta(a_i | C'_{i,j}) > P_\theta(a_i | C_i)$$

Because the probability of success for a subtask  $\kappa_j$ ,  $P_\theta(\kappa_j)$ , is the product of the individual  $P_\theta(a_i | C'_{i,j})$ , the task success rate  $P_\theta(\kappa_j)$  will also increase.

## 4 ProcLLM: Embedding Procedural knowledge into an agentic LLM

ProcLLM implements a planning and acting system (8) in which an agent iterates between HTN planning, where it updates the current (sub)task based on progress, and action execution, where it executes primitive tasks (i.e., actions) and verifies that they are completed. Relating ProcLLM to the formalism, we next describe the components of this system, as shown in Figure 2.

**Environment and states  $\mathcal{S}$ .** For the benchmarks we study in this paper, the environment corresponds to a file system with text files and python code files (Figure 2, bottom in yellow). The agent is provided

with the following text (.txt) files or python (.py) code, where the first column rwa indicates allowed read/write/append actions, respectively.

- r ProblemSpec outlines the domain-specific rules of the problem to solve and format of Answer, which is the same for all request instances. (See §C for full details.)
- r Request contains problem instance details to be solved such as the initial configuration of the problem and the goal configuration. (See §C for full details.)
- r ToolsSpec is a human readable description of Tools, which are APIs that include source code, informational text files, or structured text (e.g., csv, JSON).
- rwa Notes starts empty and will contain the agent's progress or details about the task or task history.
- rwa Solver is only provided for some problems; the environment executes it, writing to output and error.
- rwa Answer starts empty and will contain the agent's solution to the overall task.

The content of the files as well as other details are assembled into context  $C = \mathfrak{A}(e_1, e_2, \dots, e_k)$  containing one or more elements  $e_i$ . For ProcLLM this includes the action context  $C_a = \mathfrak{A}_a(s, \text{trace}, \kappa)$ , and the verify context  $C_v = \mathfrak{A}_v(s, \text{trace}, \kappa)$ .

---

**Algorithm 1** A general procedure for ProcLLM.

**Input:** Task Sequence  $K$ , Methods  $\mathcal{M}$ , horizon  $H$

---

```

1: procedure ProcLLM( $K, \mathcal{M}, H$ )
2:   UpdateTask( $K, \mathcal{M}$ )
3:   for  $i$  in  $1, \dots, H$  do
4:     if  $K == \emptyset$  then break
5:      $\kappa \leftarrow \text{head}(K)$ 
6:      $C_a \leftarrow \mathfrak{A}_a(s, \text{trace}, \kappa)$ 
7:      $a(\text{params}) \leftarrow \text{action-LLM}(C_a)$ 
8:      $\text{trace} \leftarrow "$ 
9:     if  $a = \text{verify}$  then
10:       $C_v \leftarrow \mathfrak{A}_v(s, \kappa)$ 
11:      verified, feedback  $\leftarrow \text{verify-LLM}(C_v)$ 
12:       $\text{trace} \leftarrow \text{feedback}$ 
13:      if verified then
14:        pop( $K$ )                                 $\triangleright$  remove first task  $\kappa$  from  $K$ 
15:        UpdateTask( $K, \mathcal{M}$ )
16:      else  $\triangleright$  apply r/w/a to env't; (performs  $s' \leftarrow \mathcal{T}(s, a)$ )
17:         $\text{trace} \leftarrow \text{file}$                        $\triangleright$  write file to the trace
18:        if  $a = \text{write(file, content)}$  then file.clear()
19:        file  $\leftarrow$  file + content
20:        if file = Solver then
21:          out, err  $\leftarrow \text{Exec}(\text{Solver})$ 
22:          trace  $\leftarrow \text{trace} + \text{out} + \text{err}$ 
23: procedure UpdateTask( $K, \mathcal{M}$ )
24:    $m \leftarrow \text{FindFirstRelevantMethod}(\text{head}(K), \mathcal{M})$ 
25:   if  $m$  then
26:     push( $m.\text{subtasks}, K$ )                   $\triangleright$  Prepend  $m.\text{subtasks}$ 
27:     UpdateTask( $K, \mathcal{M}$ )                       $\triangleright$  Decompose further

```

---

**Actions  $\mathcal{A}$  and transition  $\mathcal{T}$ .** The agent can take one of four actions: **verify**, **read**, **write**, or **append**.

The internal action **verify** calls a process that checks if the current task has been satisfied, returning a boolean **verified** if the task is completed as well as **feedback** which is added to the **trace**. The **verify** step is domain-specific and uses the **verify-LLM**. (A detailed example of the **verify** prompt is provided in Appendix F.)

The external actions modify the writeable files, which are subsequently assembled into the context.

**read (file):** copies file into Notes

These include: **write (file, content):** overwrites file with content

**append (file, content):** appends content to file

These external actions must be in a JSON format for the environment to execute (for a detailed example,

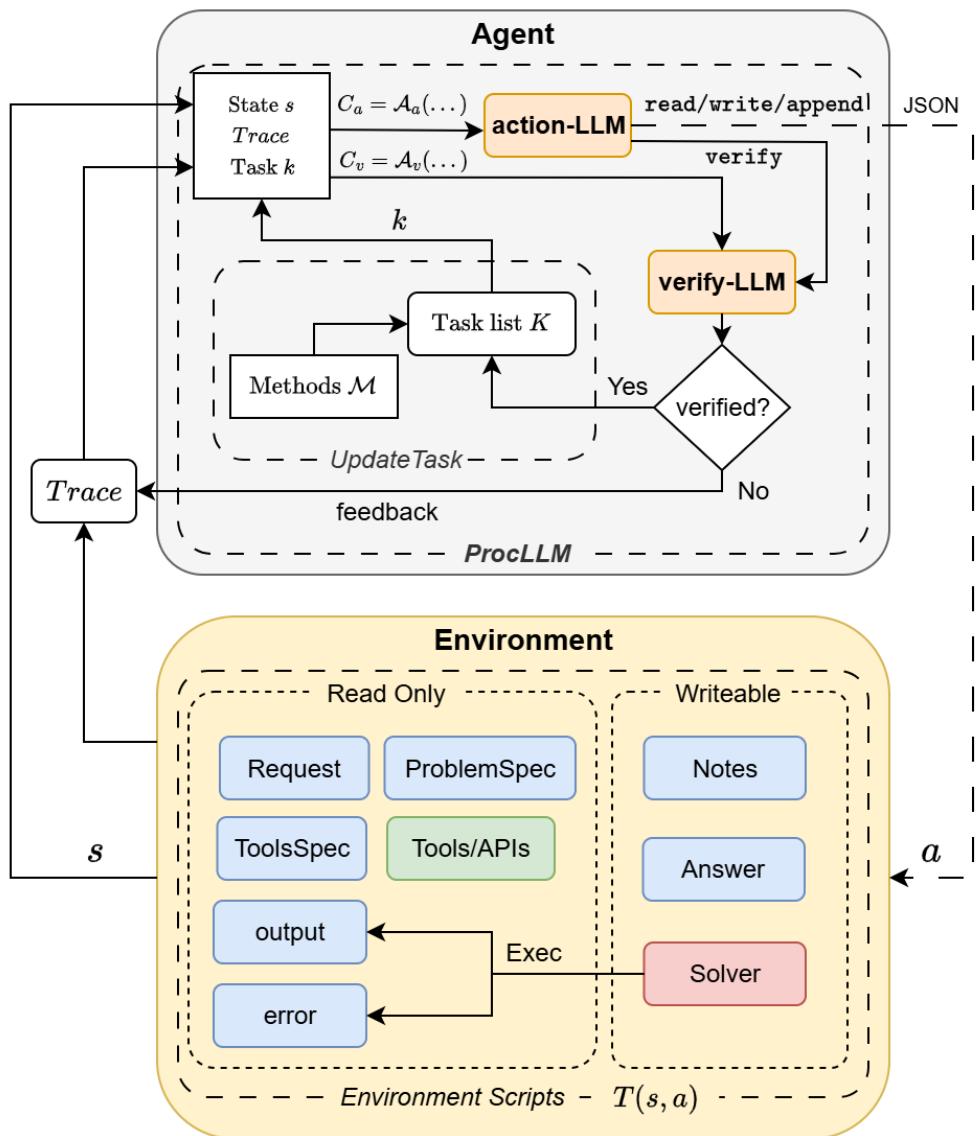


Figure 2: System overview, labeled with components of the MDP. Blue boxes denote text files, orange boxes denote LLMs, red boxes denote python files, and green boxes denote API files (scripts/databases/etc.).

see Appendix B.2). The environment implements a transition function as a set of scripts that calls `Exec(Solver)` and otherwise modifies files as indicated in the actions.

**Tasks  $K$  and Methods  $\mathcal{M}$ .** For ProcLLM, the tasks and methods are domain-dependent. Most domains will have the task `process user request`, which is the top level task that the agent will start with. Figure 1 shows some simplified methods and their associated sub/tasks; the actual methods and tasks in the system’s code differs slightly from this simple form. For the methods in this system, no preconditions are specified. In addition to the  $(\text{head}, \text{task}, \text{subtask})$ , methods in ProcLLM also include details for the `verify` step.

As an example, in Figure 1, the most abstract task `process user request` decomposes into four subtasks (Lines 3-6) and the `verify` context includes additional details (Lines 7-9). The second subtask `choose departing flight` is further decomposed by `m-choose-departing-flight` (Lines 10-17) where one of the subtasks is to understand the flight tool, as described in `m-understand-flight-tool` (Lines 18-22).

**Workflow of ProcLLM.** Figure 2 shows an overview of the system with the key components of the system. Along the top of the diagram, moving left to right, are the agentic components that produce an action. The bottom of the diagram shows the components that make up the environment.

Algorithm 1 provides pseudocode for ProcLLM. If methods  $\mathcal{M}$  are provided, Line 2 calls `UpdateTask` to decompose  $K$ . `UpdateTask` (Lines 23-27) decomposes `head( $K$ )` if there are relevant methods. If `verified` is True, `UpdateTask` will remove `head( $K$ )` and may further decompose  $K$ , where the leaves are primitive tasks that describe *what* action-LLM should do.  $K$  remains unchanged if `verified` is False,  $\mathcal{M}$  is empty, or there are no relevant methods for `head( $K$ )`.

After calling `UpdateTask`, ProcLLM then processes a sequence of tasks  $K$  up to the horizon  $H$  (Line 3) or until  $K$  is empty (Line 4). Line 6 assembles an action context  $C_a$  from the current environment state  $s$ , the execution trace of the last system execution, and the current task  $\kappa$ .  $C_a$  is processed by action-LLM (Line 7) which produces a parameterized action  $a$ , which is either an internal `verify` action or an external read/write/append action. For a `verify` action (Lines 9-15), Line 10 assembles a `verify` context  $C_v$  from the system state and current task, which is processed by the verify-LLM (Line 11).

If there remain tasks to complete, Lines 16-22 execute external actions in the environment. These external actions are JSON output that is parsed into an action  $a$ . For this paper, a set of python scripts implements the transition  $\mathcal{T}(s, a)$ , which produces a new system state  $a$  that is fed into the agent for the next cycle.

## 5 Empirical Evaluation

We evaluate ProcLLM on two benchmarks with external APIs and two with a combinatorial solution space often solved with search, summarized in the table below. **Travel Planning (TP)**, based on (33), requires booking a single-city itinerary using several tools. **Recipe Generator (RG)**, a synthetic problem, requires proposing a recipe from a list of ingredients using a tool and solver. **Blocks World (BW)**, a planning benchmark (28), requires rearranging stacks of blocks. **Unit Movement (UM)**, another synthetic benchmark, requires moving units around a graph.

	Solver?	Tools?	Problem Type	Source
TP	Y	Y	Tool Use	(33)
RG	Y	Y	Algorithmic Tool Use	This paper
BW			Planning	(28)
UM			Game	This paper

The agent is given a horizon of  $H = 100$ . Solutions are automatically verified using human-written simulators and test functions. We test three conditions: human-created  $\mathcal{M}$  (Human-TN), LLM-created  $\mathcal{M}$  (LLM-TN), or  $\mathcal{M} = \emptyset$  (No-TN). For No-TN,  $K$  has one task describing the end conditions for solving the

Table 1: Comparison: Mean success rate on subtasks in reduced Travel Planner benchmark (Verification set from (33))

Model	Flight 1	Flight 2	Hotel
Nemotron 70b No-TN	0.023	0.0	0.0
Nemotron 70b Human-TN	0.535	0.419	0.116
GPT-oss 120b No-TN	0.0	0.0	0.0
GPT-oss 120b Human-TN	0.814	0.605	<b>0.395</b>

problem instance; it functions similar to a vanilla reflexion agent (23). Experiments are performed on an AMD EPYC 7H12 64 core CPU with the LLMs being served through ollama hosted on A100 GPUs.

**Travel Planning.** This tool-use problem is based on the TravelPlanner benchmark (33), where the agent can access database APIs (FlightSearch, RestaurantSearch, etc.) to create an itinerary satisfying a user request. The original benchmark contains very difficult multi-constraint tasks where GPT-4 had a low 0.6% success rate (33). Although other works have claimed higher success rates on this benchmark, they require considerable human-written code scaffolding (10) or they use a simplified version of this benchmark (e.g., eliminating tool use in the case of (9)).

We use several simplifications so task completion is within reach of current models while keeping success non-trivial. First, we filter the validation set of (33) to obtain single destination travel requests. Second, we used a subset of the original benchmark: book a flight from origin to destination, book accommodation, and book a return flight. This subtask benchmark remains sufficiently difficult. Request is filled with a user request from the validation set of (33). ProblemSpec specifies that the agent should only fulfill parts of the request in order (in contrast to (33)).

We created 43 problems to solve for TP. Table 1 shows the success rate for completing the task *up to a given column*. For example, for Flight 2, Nemotron with Human-TN chose correct departing and returning flights in 41.9% of the requests. For both Nemotron and GPT-oss, Human-TN substantially improves task success rate over No-TN.

**Recipe Generator (RG).** In this tool use task, an agent is given a list of cooking ingredients and asked to return a dish that can be made from these ingredients. To process user requests, the agent uses two python functions: `get_ingredient_from_dish(dish)` returns the ingredients for a dish, and `get_dish_from_ingredient(ingredient)` returns a list of dishes for an ingredient. `verify` succeeds for any dish that can be made from any subset of ingredients, and the agent can write arbitrary python code calling these functions.

Using GPT-4, we generated a database of ingredients and dishes. A problem instance is constructed by randomly selecting a dish, extracting its ingredients, and augmenting this list with additional ingredients. This process ensures solvable problems that are also challenging.

Over 20 problem instances (not shown for space reasons), a task network improves results. GPT-oss-120b had a base success rate of 0% for No-TN and 25% for Human-TN. Because solutions to this task may require algorithmic solutions (e.g. filtering api call results), we applied a coding fine-tuned LLM Nemotron-70b, which averaged a success rate of 50% for No-TN and 80% for Human-TN.

**Blocks World (BW).** This problem adapts the BlocksWorld benchmark from PlanBench (28), where GPT-4 succeeded on 34.3% problems. In PlanBench, problems have 3-5 blocks, and the LLM prompt includes the rules of the task, an example problem, and a solution to the example problem.

We made several changes. ProblemSpec describes the problem and the acceptable solution format and Request contains the initial state. However, these files provide *no explicit solution example*. To assess the impact of problem complexity, we randomly choose  $b$  initial blocks that must be in a final stack of height  $h$ ; a solution is correct for stacking  $h$ , regardless of other blocks  $b - h$ .

Fig 3 reveals a downward trend in task success as complexity increases (from  $b = 3$  to  $b = 9$ ). Either task network (Human-TN or LLM-TN) greatly increases the success rate; the success rate of GPT-oss-120b Human-TN remains 70% in contrast to GPT-oss-120b No-TN of below 5%. Furthermore, the smaller GPT-oss-20b Human-TN significantly outperforms GPT-oss-120b.

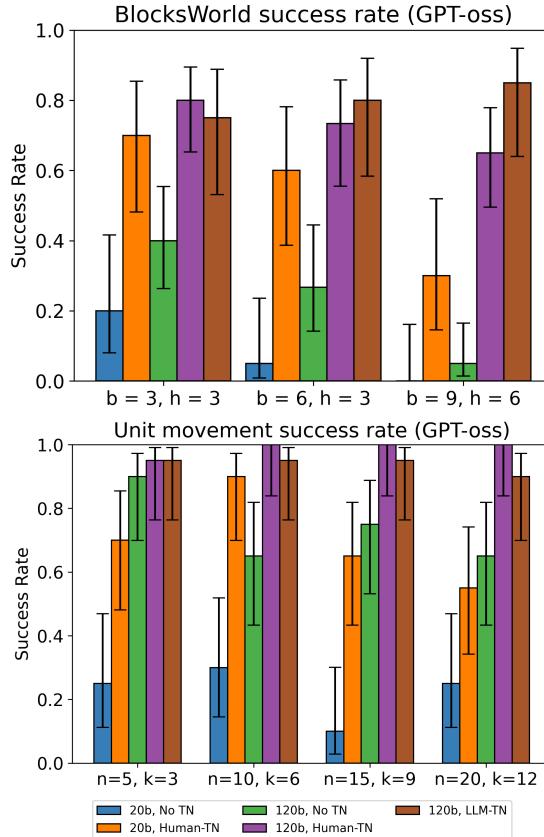


Figure 3: Success rates for BlockWorld and Unit Movement domains, evaluated across increasing problem complexity ( $b$  starting blocks and  $h$  final stack height for BW, number of units  $n$  for UM). Bars show mean success rates, with 95% Wilson confidence intervals.

**Unit Movement (UM).** In this task, for each problem instance we generate a graph  $G$  with the following properties (Figure A.1 shows an example problem). A **target node**  $v_T \in G$  is adjacent to **four neighbors**  $v^{adj}$ , each with an edge  $(v^{adj}, v_T)$ . Each neighbor has three **outer neighbors**; that is, for each node  $v_i^{adj}$ , there are at least 3 outer nodes  $v^{out}$  with an edge  $(v_i^{adj}, v^{out})$ . After generating the above node sets, we add 12 more random edges between neighboring nodes (excluding the target  $v_T$ ). For three of the sections of the graph we place  $n$  units to random nodes in each section. We randomly name each node (e.g. Seaside, Canyon, Gardens) and each unit based on their section (e.g. Bravo\_0, Alpha\_1). The goal of the task is to move units so that the target is surrounded from three neighbors with at least  $k$  units at each node.

We varied problem complexity by the number of  $n$  units per group and the number  $k$  units required for surrounding. Figure 3 reveals two insights. First, Human-TN increases the task success rate regardless of the base capability of the LLM used in the system; at  $(n = 10, k = 6)$ , the 20b model Human-TN outperforms 120b No-TN. Second, higher  $(n, k)$  values seem show a non-linear correlation with task difficulty; future work should explore whether the LLM is exploiting problem structure to solve larger problems.

## 6 Discussion

The evaluation of ProcLLM shows that Human-TN and LLM-TN improve task success rate, often significantly and even across No-TN success capability and different benchmarks. We also varied problem complexity and showed that, in BW and UM, the LLM system with a task planner seems to have a slower decrease

in task success rate than the LLM system without a task planner. This is true in both domains for GPT-oss-120b, but the results for GPT-oss-20b show less improvement; we hypothesize that smaller models such as GPT-oss-20b can run into context length issues that are not addressed by our task network approach. We further examine several more questions about these results.

**Why is No-TN doing so poorly?** Although not obvious from the results presented so far, there are some subtle differences in model behavior that we would like to discuss. To explain, consider the first few steps that an LLM may need to take to solve one of the agentic problems seen so far. In each of the problems, there is a `request.txt` and a `problem specification.txt`. The first few steps of solving an agentic task is to read each of these files and then take notes on the files into the short-term memory file (Notes). Some models (e.g., nemotron, codestral, llama) are naturally proficient in this and will generate a sequence of actions:

1. Read `request.txt`
2. Append [request details]
3. Read `problem specification.txt`
4. Append [spec details]

This sequence of actions can be generated even if the task network does not have explicit task nodes about taking notes (or even without a task network at all). However, our experiments with GPT-oss-120b revealed the system would get stuck in loops like:

1. Read `request.txt`
2. Read `problem specification.txt`
3. Read `request.txt`
4. Read `problem specification.txt`
5. etc.

This causes the system to timeout and fail as a result. We can alleviate this problem for domains such as BW by explicitly extending the final goal for the task. For example, we could change `solve` the user request into `solve` the user request and have notes on the request and problem specification. But such changes may not scale if a problem has many goals.

**Are LLM-constructed task networks better? (LLM-TN)** To study LLM-TN, we constructed a prompt that includes the problem specification, the agent prompt, and two valid task network node examples and asked Gemini 2.5 Pro to produce a task network (LLM-TN). In Figure 3, we see that task success rate of LLM-TN (brown bars) does better than No-TN while varying against Human-TN.

**Does Human-TN or LLM-TN save runtime?** Figure 4 shows runtime statistics for GPT-oss-120b in ProcLLM across the evaluation benchmarks (Figure A.2 shows full details). For Blocks World and Unit Movement, the statistics are averaged over all problem sizes.

In three of the four benchmarks, the task planner decreases the number of system iterations required for a given problem instance. The exception is Recipe Generator, where the LLM with No-TN produces quick but always incorrect outputs, unlike other benchmarks where No-TN typically fails via timeout. One explanation is that the LLM can treat the request in Recipe Generator as a query to be answered directly, bypassing the APIs altogether. This shortcircuiting behavior mirrors earlier findings ((3)) that LLMs can default to superficial strategies unless constrained externally.

Using a task network typically increases the average time per iteration while decreasing the total number of iterations. One important takeaway is that using the task network does not significantly increase the time it takes to complete a task. In some cases, such as on BlocksWorld, using GPT-oss-20b in ProcLLM will outcompete GPT-oss-120b in both time and performance metrics.

## 7 Conclusion

We have argued that procedural knowledge helps LLM workflows. We formalized, implemented, and evaluated ProcLLM that uses HTNs to improve LLM performance on complex agentic tasks. ProcLLM leveraged an HTN to decompose complex tasks into subtasks that completed by an agentic LLM. While

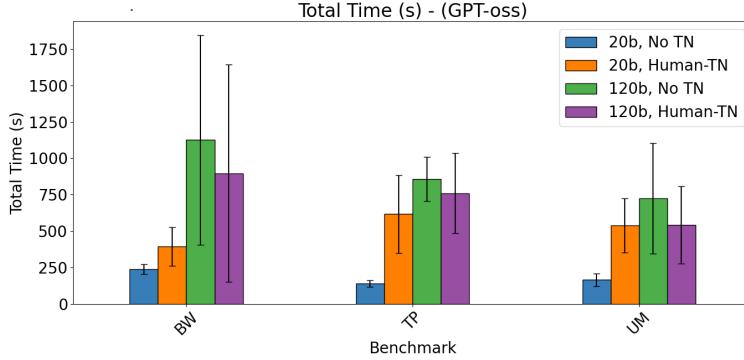


Figure 4: A comparison of average runtime statistics for GPT-oss across different benchmarks (BW - BlocksWorld, TP - Travel Planner, UM - Unit Movement). (See Fig. A.2 for full details)

ProcLLM uses a file system as the environment, this framework could be applied to employ LLM agents in more general MDP domains (e.g., games, control problems).

We demonstrated on four benchmarks that the HTN decomposition in ProcLLM significantly increased task success in difficult multi-step agentic tasks. This is effective regardless of the base LLM. This improvement uses natural language descriptions; ProcLLM does not require translation to a planning formalism (e.g., PDDL, HDDL). Using natural language allows a flexible and natural tradeoff in the level of abstraction for HTN planning. The high-level planning afforded by the task planner increases the likelihood of completing long, complex tasks and also reduces the probability of many common agentic LLM issues, such as action looping, that are exacerbated by a deficiency in implicit task planning.

LLM generated task networks sometimes do well, but they seem to generally lack the specificity of human generated networks. In future work, it would be interesting to see if LLMs can build task networks from the ground up in a curriculum learning fashion, starting with simple tasks that gradually become more complex. Also, it would be interesting to explore an equivalent approach in (3) where an LLM transforms the HTN recipe into code for speed and consistent replicability. Finally, procedures are frequently codified into documents as recipes, checklists, or standard operating procedures. Beyond learning networks through curriculum learning, we could explore methods for using RAG (e.g., (22)) or fine-tuning (e.g., (30)) to improve the distillation of task networks from domain documents

## Acknowledgments

We thank NRL for funding this research.

## References

- [1] Fahiem Bacchus and Frouduald Kabanza. 2000. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence* 116, 1 (Jan. 2000), 123–191. [https://doi.org/10.1016/S0004-3702\(99\)00071-5](https://doi.org/10.1016/S0004-3702(99)00071-5)
- [2] Fahiem Bacchus and Qiang Yang. 1994. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence* 71, 1 (1994), 43–100. [https://doi.org/10.1016/0004-3702\(94\)90062-0](https://doi.org/10.1016/0004-3702(94)90062-0)
- [3] Peter Belcak and Pavlo Molchanov. 2025. Universal Deep Research: Bring Your Own Model and Strategy. *arXiv preprint arXiv:2509.00244* (2025).
- [4] Edward Y Chang and Longling Geng. 2025. SagaLLM: Context Management, Validation, and Transaction Guarantees for Multi-Agent LLM Planning. *arXiv preprint arXiv:2503.11951* (2025).
- [5] 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [6] A. I. Coles and A. J. Smith. 2007. Marvin: A Heuristic Search Planner with Online Macro-Action Learning. *Journal of Artificial Intelligence Research* 28 (Feb. 2007), 119–156. <https://doi.org/10.1613/jair.2077>
- [7] Christopher W. Geib and Robert P. Goldman. 2009. A Probabilistic Plan Recognition Algorithm Based on Plan Tree Grammars. *Artificial Intelligence* 173, 11 (July 2009), 1101–1132. <https://doi.org/10.1016/j.artint.2009.01.003>
- [8] Malik Ghallab, Dana Nau, and Paolo Traverso. 2025. *Acting, Planning and Learning*. Cambridge University Press. author preprint <https://projects.laas.fr/planning/>.
- [9] Atharva Gundawar, Mudit Verma, Lin Guan, Karthik Valmeekam, Siddhant Bhambri, and Subbarao Kambhampati. 2024. Robust planning with llm-modulo framework: Case study in travel planning. *arXiv preprint arXiv:2405.20625* (2024).
- [10] Yilun Hao, Yongchao Chen, Yang Zhang, and Chuchu Fan. 2024. Large language models can plan your travels rigorously with formal verification tools. *CoRR* (2024).
- [11] Chengkai Huang, Junda Wu, Yu Xia, Zixu Yu, Ruhan Wang, Tong Yu, Ruiyi Zhang, Ryan A Rossi, Branislav Kveton, Dongruo Zhou, et al. 2025. Towards agentic recommender systems in the era of multimodal large language models. *arXiv preprint arXiv:2503.16734* (2025).
- [12] Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. 2024. Understanding the planning of LLM agents: A survey. *arXiv preprint arXiv:2402.02716* (2024).
- [13] Subbarao Kambhampati. 2024. Can large language models reason and plan? *Annals of the New York Academy of Sciences* 1534, 1 (2024), 15–18.
- [14] Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B Murthy. 2024. Position: LLMs can't plan, but can help planning in LLM-modulo frameworks. In *Forty-first International Conference on Machine Learning*.
- [15] Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, et al. 2022. MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv preprint arXiv:2205.00445* (2022).
- [16] Harsha Kokel, Michael Katz, Kavitha Srinivas, and Shirin Sohrabi. 2025. Acpbench: Reasoning about action, change, and planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 26559–26568.
- [17] John E. Laird, Christian Lebiere, and Paul S. Rosenbloom. 2017. A Standard Model of the Mind: Toward a Common Computational Framework across Artificial Intelligence, Cognitive Science, Neuroscience, and Robotics. *AI Magazine* 38, 4 (Dec. 2017), 13–26. <https://doi.org/10.1609/aimag.v38i4.2744>
- [18] Pat Langley and Howard E Shrobe. 2021. Hierarchical Problem Networks for Knowledge-Based Planning. (Sept. 2021), 19.

- [19] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbulin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems* 36 (2023), 51991–52008.
- [20] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. 2024. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth* 1, 1 (2024), 9.
- [21] Timo Schick, Jane Dwivedi-Yu, Roberto Dessa, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools, 2023. *arXiv preprint arXiv:2302.04761* (2023).
- [22] Kaize Shi, Xueyao Sun, Qing Li, and Guandong Xu. 2024. Compressing long context for enhancing rag with amr-based concept distillation. *arXiv preprint arXiv:2405.03085* (2024).
- [23] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652.
- [24] Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. 2025. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. *arXiv preprint arXiv:2506.06941* (2025).
- [25] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Kaelbling, and Michael Katz. 2024. Generalized planning in pddl domains with pretrained large language models. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 38. 20256–20264.
- [26] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF international conference on computer vision*. 2998–3009.
- [27] Austin Tate. 1977. Generating Project Networks. In *Proc. of IJCAI (IJCAI'77)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 888–893.
- [28] Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. *Advances in Neural Information Processing Systems* 36 (2023), 38975–38987.
- [29] Karthik Valmeekam, Kaya Stechly, Atharva Gundawar, and Subbarao Kambhampati. 2024. Planning in strawberry fields: Evaluating and improving the planning and scheduling capabilities of lrm o1. *arXiv preprint arXiv:2410.02162* (2024).
- [30] Pulkit Verma, Ngoc La, Anthony Favier, Swaroop Mishra, and Julie A. Shah. 2025. Teaching LLMs to Plan: Logical Chain-of-Thought Instruction Tuning for Symbolic Planning. *arXiv:2509.13351 [cs.AI]* <https://arxiv.org/abs/2509.13351>
- [31] Zhiruo Wang, Zhoujun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. 2024. What are tools anyway? a survey from the language model perspective. *arXiv preprint arXiv:2403.15452* (2024).
- [32] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–10.
- [33] Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. 2024. Travelplanner: A benchmark for real-world planning with language agents. *arXiv preprint arXiv:2402.01622* (2024).
- [34] Hui Yang, Sifu Yue, and Yunzhong He. 2023. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224* (2023).
- [35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).
- [36] Yankai Zeng, Abhiramon Rajasekharan, Parth Padalkar, Kinjal Basu, Joaquín Arias, and Gopal Gupta. 2024. Automated interactive domain-specific conversational agents that understand human dialogs. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 204–222.

- [37] Wentao Zhang, Ce Cui, Yilei Zhao, Rui Hu, Yang Liu, Yahui Zhou, and Bo An. 2025. Agentorchestra: A hierarchical multi-agent framework for general-purpose task solving. *arXiv preprint arXiv:2506.12508* (2025).
- [38] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 19632–19642.
- [39] Zhehua Zhou, Jiayang Song, Kunpeng Yao, Zhan Shu, and Lei Ma. 2024. Isr-llm: Iterative self-refined large language model for long-horizon sequential task planning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2081–2088.

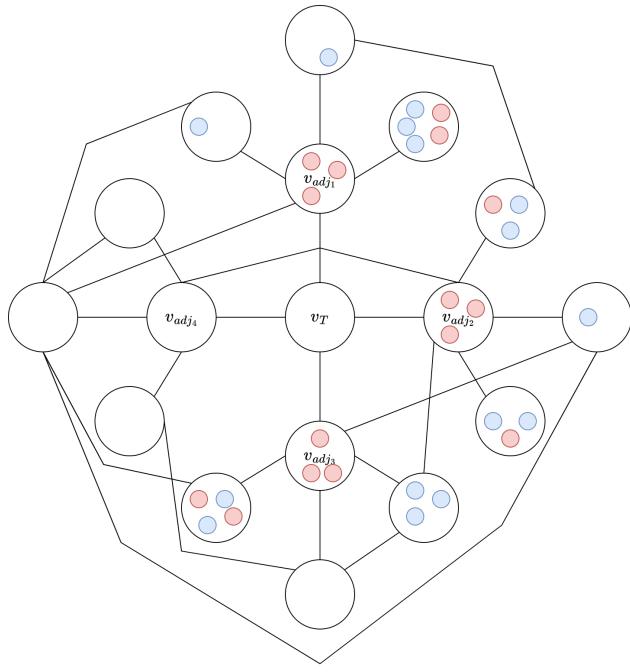


Figure A.1: An example problem instance for the unit movement domain showing the initial starting position (blue circles) and a valid solution (red circles).

## A Additional Figures

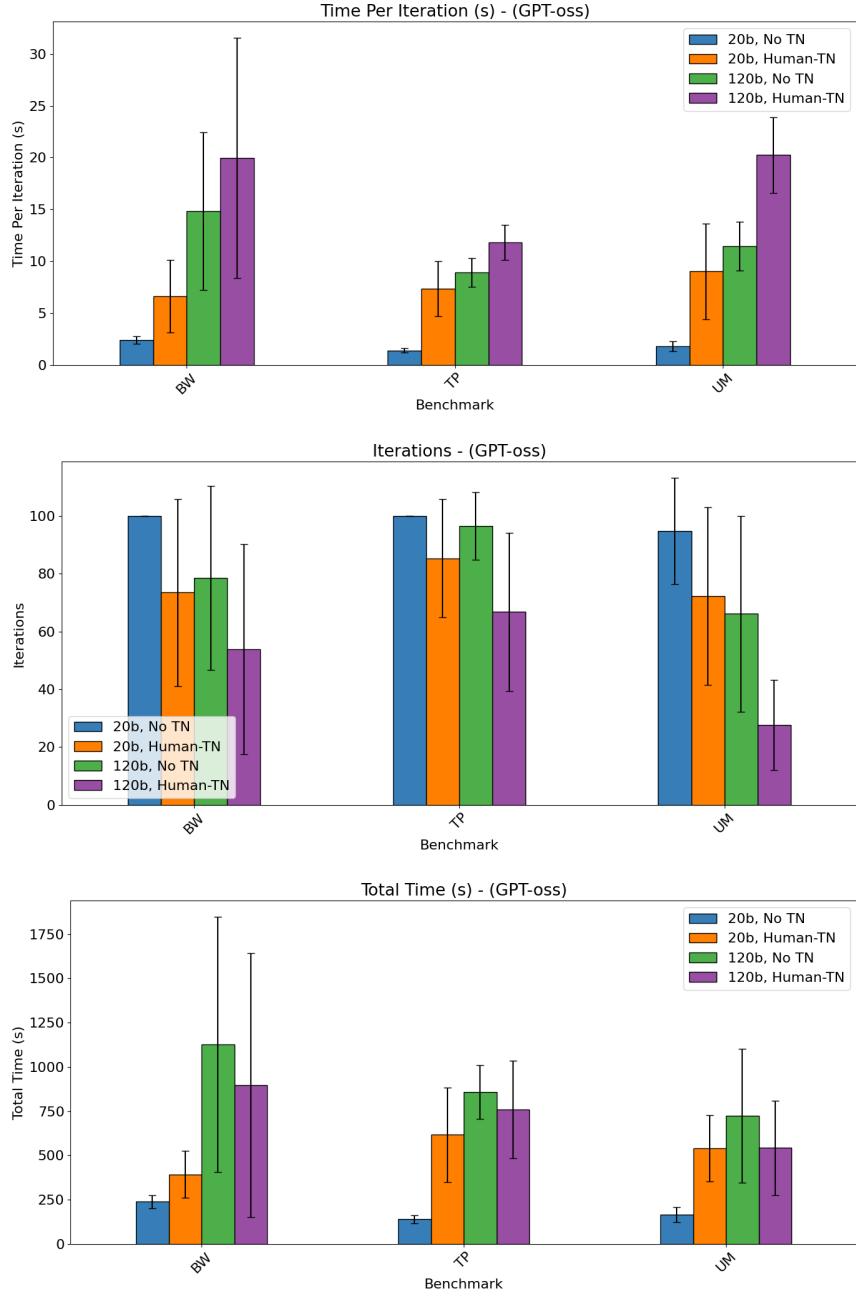


Figure A.2: A comparison of average runtime statistics for GPT-oss 120b across different benchmarks (BW - BlocksWorld, TP - Travel Planner, UM - Unit Movement).

## B Prompts

### B.1 Standard Agent Prompt

This prompt is given to the agent at each iteration for all domains. Only the content of the read and writeable files may differ across domains.

Your goal is to interact with a system that processes commands in order to solve a given task. The task is specified in the problem specification document and the task instance in the request file. You MUST read and take notes on these files to solve the problem. You will only be able to see the information from the immediately preceding action, so make sure you take notes such that this information is not lost.

You have the following actions and files available to you:

1. Read: reads [action\_arg1]. Line numbers will be displayed.  
Read [action\_arg1]

2. Write: overwrites text file [action\_arg1] with [action\_arg2].  
Write [action\_arg1] [action\_arg2]

3. Append: appends the text file [action\_arg1] with [action\_arg2].  
Append [action\_arg1] [action\_arg2]

4. Verify: Send a request to check if current task has been achieved.  
Verify

You are allowed to read and write the following file (r/w):  
solver.py (r/w): this file is automatically executed after every revision. The standard output of this file is written to output.txt  
files/notes.txt (r/w): the contents of this file will always be displayed in the notes section down below.  
answer.txt (r/w): a file to hold the final answer to user request

You are allowed to access the following read-only files (r):  
files/request.txt (r): the user request  
files/problem\_specification.txt (r): a problem specification document  
files/tools\_specification.txt (r): a document specifying tools  
output.txt (r): the stdout of solver.py

The contents of files/notes.txt is:  
\*\*\*NOTES START\*\*\*  
{}  
\*\*\*NOTES END\*\*\*

Your last response was:  
\*\*\*LAST RESPONSE START\*\*\*  
{}  
\*\*\*LAST RESPONSE END\*\*\*

The last ten commands executed were:  
\*\*\*LAST COMMAND START\*\*\*  
{}  
\*\*\*LAST COMMAND END\*\*\*

Here is the output of your last command:  
\*\*\*LAST COMMAND OUTPUT\*\*\*  
{}  
\*\*\*LAST COMMAND OUTPUT END\*\*\*

Your Limitations:

1. Do not attempt to read or write to any files not listed. This will result in  
↳ a file access denied error.
2. You can only open one file at a time.
3. If you get a file access denied error, make sure you use the correct path and  
↳ extension.

General Instructions:

1. Your response should be in a json format (formatted with newlines) and  
↳ include the following components:  
 \* observation: a detailed observation about the current state of the task and  
↳ what you have done so far  
 \* thought: reasoning about what next steps are valid and how to proceed  
 \* action {{name: , action\_arg1: , action\_arg2: }}: action [read, write, append,  
↳ or verify] and arguments for the action
2. There is no limit on how long your json can be, but do not include additional  
↳ nested components.
3. If you receive a JSONDecode error, this means your json response is  
↳ improperly formatted.
4. Think step by step. Justify each step with something from existing notes or  
↳ previous responses.
5. Respond only with your json action. Use an empty string if any arguments are  
↳ unneeded.
6. A task planner will give you a task to follow with an effect. Focus on  
↳ achieving this effect.
7. Call the verify action when the current effect is satisfied to move onto the  
↳ next task.
8. You MUST take notes to solve the problem. Make sure to take notes after a  
↳ read.
9. Do not include section headings (e.g. \*\*\* ) in your response.

Your current task is:

Task: {}

The effect that you are trying to achieve is:

Effect: {}

## B.2 Agent Action Examples

The agent is expected to respond with a JSON that will be parsed and processed as an action. Here are some examples:

```
{
  "observation": "Currently, no notes have been taken and no files have been  
↳ read. The task is to take notes on the problem specification file.",  

  "thought": "To achieve the effect of having notes contain the information from  
↳ the problem specification file, I first need to read the problem  
specification file. Then, I will append the contents of this file to the  
notes.txt file to ensure persistence of the information.",  

  "action": {
    "name": "Read",
    "action_arg1": "files/problem_specification.txt",
    "action_arg2": ""
  }
}
```

```
{
  "observation": "Having read the request file (files/request.txt), I've
  ↵ obtained additional context: a 3-day trip from Missoula to Dallas (March
  ↵ 23rd-25th, 2022) with a budget of $1,900. This aligns with my initial
  ↵ notes but adds a budget constraint. My current understanding is that I
  ↵ need to use Flights.py for flights and Accommodations.py for lodging
  ↵ within this budget. However, I still lack detailed insights from the
  ↵ problem specification file beyond the initial notes. To ensure
  ↵ completeness, I'll append these new details to notes.txt.",

  "thought": "With the request's context in mind, particularly the budget, my
  ↵ next steps should involve utilizing the specified tools (Flights.py and
  ↵ Accommodations.py) while considering the $1,900 budget. Before executing
  ↵ code, it's prudent to update my notes with the latest understanding,
  ↵ including the budget constraint, to maintain a comprehensive record of my
  ↵ approach.",

  "action": {
    "name": "Append",
    "action_arg1": "files/notes.txt",
    "action_arg2": "Additional context from request file: Budget for Missoula to
      ↵ Dallas trip (March 23rd-25th, 2022) is $1,900. Next steps involve using
      ↵ Flights.py and Accommodations.py with budget consideration."
  }
}
```

```
{
  "observation": "Read the problem specification which defines the allowed
  ↵ actions and required output format. Notes provide the initial state, goal
  ↵ state, and a sequence to unstack all blocks to the table. To achieve the
  ↵ goal, we will first execute the unstack/put sequence to clear the table,
  ↵ then pick and stack blocks in the order required by the goal: cyan at
  ↵ bottom, then blue, orange, green, yellow, purple. This yields a complete
  ↵ plan using only pick, put, unstack, and stack actions.",
  "thought": "Construct a plan consisting of the 8 unstack/put steps from the
  ↵ notes, followed by pick/stack steps to build the goal tower. Write this
  ↵ plan to answer.txt in the exact line-by-line format required.",
  "action": {
    "name": "Write",
    "action_arg1": "answer.txt",
    "action_arg2": "unstack black green\nput black\nunstack green cyan\nput
      ↵ green\nunstack cyan blue\nput cyan\nunstack blue purple\nput blue\npick
      ↵ blue\nstack blue cyan\npick orange\nstack orange blue\npick green\nstack
      ↵ green orange\npick yellow\nstack yellow green\npick purple\nstack purple
      ↵ yellow"
  }
}
```

### B.3 Example mid-iteration prompt

Your goal is to interact with a system that processes commands in order to solve a given task. The task is specified in the problem specification document and the task instance in the request file. You MUST read and take notes on these files to solve the problem. You will only be able to see the information from the immediately preceding action, so make sure you take notes such that this information is not lost. You have the following actions and files available to you:

1. Read: reads [action\_arg1]. Line numbers will be displayed.  
Read [action\_arg1]

2. Write: overwrites text file [action\_arg1] with [action\_arg2].  
Write [action\_arg1] [action\_arg2]

3. Append: appends the text file [action\_arg1] with [action\_arg2].  
Append [action\_arg1] [action\_arg2]

4. Verify: Send a request to check if current task has been achieved.  
Verify

You are allowed to read and write the following file (r/w):  
solver.py (r/w): this file is automatically executed after every revision. The standard output of this file is written to output.txt  
files/notes.txt (r/w): the contents of this file will always be displayed in the notes section down below.  
answer.txt (r/w): a file to hold the final answer to user request

You are allowed to access the following read-only files (r):  
files/request.txt (r): the user request  
files/problem\_specification.txt (r): a problem specification document  
files/tools\_specification.txt (r): a document specifying tools  
output.txt (r): the stdout of solver.py

The contents of files/notes.txt is:

\*\*\*NOTES START\*\*\*

The user will provide a request to make a travel plan. You do not need to complete the entire travel plan, but only the flight and accommodations for the user's trip. You will need to use code files Flights.py and Accommodations.py to access the database for required to solve the user request

Your solution should be returned in a json format in answer.txt with two flights (departing and returning) as well as the accommodation used for the duration of the traveler's stay:

departing\_flight: flight number  
returning\_flight: flight number  
accommodation: name

Be aware that the accommodations returned by the Accommodations tool may have names that do not seem to correspond with the city that you used for the search; this is expected behavior and you should proceed with the results you obtained using the tools.

```
Origin: Detroit
Destination: San Diego
Departure Date: 2022-03-05
Returning Date: 2022-03-07
Duration: 3 days
Budget: $3000
Traveler: solo
```

```
FlightSearch.run method:
- Inputs:
  * Departure City (string)
  * Destination City (string)
  * Date (string in YYYY-MM-DD format)
- Expected Output:
  * List of flight options (each containing flight number, departure/arrival
    ↳ times, price, etc.)
```

```
***NOTES END***
```

```
Your last response was:
***LAST RESPONSE START***
```

```
{
  "observation": "We have read the tools specification which shows how to import
    ↳ and use the Flights tool. To obtain flight options we need to write code
    ↳ in solver.py that imports the Flights class, creates an instance, runs the
    ↳ search for Detroit to San Diego on 2022-03-05, and prints the results. The
    ↳ solver will then be executed and its stdout captured in output.txt for
    ↳ later analysis.",
  "thought": "Next step is to write the described code into solver.py so that
    ↳ the flight search is performed and results are printed. This will allow us
    ↳ to later read output.txt and extract a departing flight number and cost.",
  "action": {
    "name": "Write",
    "action_arg1": "solver.py",
    "action_arg2": "from tools.Flights import Flights\n\ndef main():\n    FlightSearch = Flights()\n    results = FlightSearch.run(\"Detroit\",
      ↳ \"San Diego\", \"2022-03-05\")\n    print(results)\n\nif __name__ ==
      ↳ \"__main__\":\n        main()"
  }
}
***LAST RESPONSE END***
```

```
The last ten commands executed were:
```

```
***LAST COMMAND START***
Verify
Read files/request.txt
Append files/notes.txt
Verify
Read files/tools_specification.txt
Append files/notes.txt
Verify
Read solver.py
Read files/tools_specification.txt
```

```

Write solver.py
***LAST COMMAND END***

Here is the output of your last command:
***LAST COMMAND OUTPUT***
Updated solver.py:
from tools.Flights import Flights

def main():
    FlightSearch = Flights()
    results = FlightSearch.run("Detroit", "San Diego", "2022-03-05")
    print(results)

if __name__ == "__main__":
    main()

Code executed with stdout:
    Flight Number  Price DepTime ... OriginCityName DestCityName Distance
3306155      F3528556    720   08:33 ...        Detroit     San Diego    1956.0
[1 rows x 9 columns]

***LAST COMMAND OUTPUT END***

Your Limitations:
1. Do not attempt to read or write to any files not listed. This will result in
   ↵ a file access denied error.
2. You can only open one file at a time.
3. If you get a file access denied error, make sure you use the correct path and
   ↵ extension.

General Instructions:
1. Your response should be in a json format (formatted with newlines) and
   ↵ include the following components:
   * observation: a detailed observation about the current state of the task and
   ↵ what you have done so far
   * thought: reasoning about what next steps are valid and how to proceed
   * action {name: , action_arg1: , action_arg2: }: action [read, write, append,
   ↵ or verify] and arguments for the action
2. There is no limit on how long your json can be, but do not include additional
   ↵ nested components.
3. If you receive a JSONDecode error, this means your json response is
   ↵ improperly formatted.
4. Think step by step. Justify each step with something from existing notes or
   ↵ previous responses.
5. Respond only with your json action. Use an empty string if any arguments are
   ↵ unneeded.
6. A task planner will give you a task to follow with an effect. Focus on
   ↵ achieving this effect.
7. Call the verify action when the current effect is satisfied to move onto the
   ↵ next task.
8. You MUST take notes to solve the problem. Make sure to take notes after a
   ↵ read.
9. Do not include section headings (e.g. ***) in your response.

```

Your current task is:  
 Task: choose a departing flight  
 The effect that you are trying to achieve is:  
 Effect: notes contains details about a flight number and cost for a flight from  
 ↵ origin to destination on departure date consistent with request preferences

### B.3.1 Example response to the previous prompt

```
{
  "observation": "The solver.py execution printed flight search results to
  ↵ output.txt, showing a flight with number F3528556, price 720, departure
  ↵ time 08:33, etc. We need to capture these details in notes.txt.",
  "thought": "To satisfy the effect, I will read output.txt to obtain the flight
  ↵ information, then append a concise note with the departing flight number
  ↵ and its cost to notes.txt.",
  "action": {
    "name": "Read",
    "action_arg1": "output.txt",
    "action_arg2": ""
  }
}
```

## C Domain Specification

For each domain, we write problem specification documents that the agent needs to read in order to complete a task (e.g. what they need to actually do to complete the agentic task). Each problem instance also has an automatically generated `request.txt` that describes the problem details for a specific problem instance.

### C.1 Domain: Travel Planner

#### C.1.1 Problem Specification

The user will provide a request to make a travel plan. You do not need to  
 ↵ complete the entire travel plan, but only the flight and accommodations for  
 ↵ the user's trip. You will need to use code files `Flights.py` and  
 ↵ `Accommodations.py` to access the database for required to solve the user  
 ↵ request

Your solution should be returned in a json format in `answer.txt` with two flights  
 ↵ (departing and returning) as well as the accommodation used for the duration  
 ↵ of the traveler's stay:

```
departing_flight: flight number
returning_flight: flight number
accommodation: name
```

Be aware that the accommodations returned by the Accommodations tool may have  
→ names that do not seem to correspond with the city that you used for the  
→ search; this is expected behavior and you should proceed with the results  
→ you obtained using the tools.

### C.1.2 Example User Request

Please create a travel plan for a 3-day trip from Missoula to Dallas scheduled  
→ from March 23rd to March 25th, 2022. The budget for this trip is set at  
→ \$1,900.

## C.2 Domain: Recipe Generator

### C.2.1 Problem Specification

The user will provide a request containing a list of ingredients, your job is to  
→ select a valid recipe using those ingredients.

You do not need to use all ingredients provided by the user. Any recipe that can  
→ be made from the ingredients provided by the user is a valid recipe. For  
→ example, if the user provides the ingredient list:

Chicken Breast  
Tomatoes  
Soy Sauce  
Sesame Oil  
Brown Sugar  
Garlic

The following recipe:

Dish: Chicken and Tomato Stir Fry  
Ingredients:  
Chicken Breast  
Tomatoes  
Soy Sauce  
Sesame Oil

is valid solution to the user request, and a valid final output is:

Chicken and Tomato Stir Fry

### C.2.2 Example User Request

Hello, I'd like to request a recipe for the following ingredients:  
Chicken Breast  
Tomatoes  
Soy Sauce

Sesame Oil  
Garlic  
Ginger

### C.3 Domain: Blocks World

#### C.3.1 Problem Specification

I am playing with a set of blocks where I need to arrange the blocks into stacks.

→ Here are the actions I can do:

Pick up a block

Unstack a block from on top of another block

Put down a block

Stack a block on top of another block

I have the following restrictions on my actions:

I can only pick up or unstack one block at a time.

I can only pick up or unstack a block if my hand is empty.

I can only pick up a block if the block is on the table and the block is clear.

→ A block is clear if the block has no other blocks on top of it and if the

→ block is not picked up.

I can only unstack a block from on top of another block if the block I am  
→ unstacking was really on top of the other block.

I can only unstack a block from on top of another block if the block I am  
→ unstacking is clear.

Once I pick up or unstack a block, I am holding the block.

I can only put down a block that I am holding.

I can only stack a block on top of another block if I am holding the block being  
→ stacked.

I can only stack a block on top of another block if the block onto which I am  
→ stacking the block is clear.

Once I put down or stack a block, my hand becomes empty.

Once you stack a block on top of a second block, the second block is no longer  
→ clear.

Please provide a plan as a list of actions with space delimitated arguments line

→ by line in the following format in answer.txt:

pick [block]: pick a block up

put [block]: put a block down

stack [block1] [block2]: stack block1 on block2

unstack [block1] [block2]: unstack block1 from on top of block2

An example plan:

unstack blue red

put blue

pick red

stack red green

Write an answer to answer.txt. The plan will be checked automatically so please

→ follow the above format. The only actions allowed are [pick, put, stack,

→ unstack].

### C.3.2 Example User Request ( $n = 9, k = 6$ )

```
As initial conditions I have that:  
the blue block is on the table  
the gray block is on top of the blue block  
the red block is on top of the gray block  
the orange block is on top of the red block  
the yellow block is on top of the orange block  
the black block is on top of the yellow block  
the cyan block is clear  
the cyan block is on top of the black block  
the purple block is clear  
the purple block is on the table  
the green block is clear  
the green block is on the table  
My goal is to have that:  
the orange block is on top of the gray block  
the blue block is on top of the orange block  
the black block is on top of the blue block  
the yellow block is on top of the black block  
the red block is on top of the yellow block
```

## C.4 Domain: Unit Movement

### C.4.1 Problem Specification

You are an AI strategist tasked with controlling a team of infantry units in a wargame. You are given the current game state and must output a list of actions for your units to execute in a single turn.

Game State:

The game is played on a network. Units have location, and a unit ID.

Example unit:

```
{"unit_id": "Eagle_0", "location": "Westwood"},
```

Available Actions:

\* Move:

```
{"unit_id": "unit_id", "action_type": "move", "location": target_location}
```

The unit will move to target\_location.

Output Format:

Each action must be a valid JSON object as described above in a list. Do not include any other text besides the JSON list.

Example Output for two units:

```
[  
  {"unit_id": "Blue_0", "action_type": "move", "location": "Village"},  
  {"unit_id": "Blue_1", "action_type": "move", "location": "Southshore"}]
```

To make the task easier for more complex problem instances, it can be beneficial

- ↪ to group units together, decide on a general top level movement strategy,
- ↪ and then decide on unit movement within each group to follow the top level strategy.

#### C.4.2 Example User Request ( $n = 15, k = 9$ )

##### Goal:

Surround the target location {Eastfield} from at least three neighboring

- ↪ locations with your units. A neighboring location is considered covered
- ↪ if there are at least 9 units at that location.

##### Units:

```

Infantry (Hotel_0) at (Crestview)
Infantry (Hotel_1) at (Meadow)
Infantry (Hotel_2) at (Hillcrest)
Infantry (Hotel_3) at (Hillcrest)
Infantry (Hotel_4) at (Hillcrest)
Infantry (Hotel_5) at (Crestview)
Infantry (Hotel_6) at (Hillcrest)
Infantry (Hotel_7) at (Hillcrest)
Infantry (Hotel_8) at (Hillcrest)
Infantry (Hotel_9) at (Hillcrest)
Infantry (Hotel_10) at (Meadow)
Infantry (Hotel_11) at (Crestview)
Infantry (Hotel_12) at (Hillcrest)
Infantry (Hotel_13) at (Hillcrest)
Infantry (Hotel_14) at (Hillcrest)
Infantry (Lima_0) at (Bayview)
Infantry (Lima_1) at (Creekbend)
Infantry (Lima_2) at (Bayview)
Infantry (Lima_3) at (Townsend)
Infantry (Lima_4) at (Bayview)
Infantry (Lima_5) at (Creekbend)
Infantry (Lima_6) at (Bayview)
Infantry (Lima_7) at (Creekbend)
Infantry (Lima_8) at (Creekbend)
Infantry (Lima_9) at (Creekbend)
Infantry (Lima_10) at (Bayview)
Infantry (Lima_11) at (Creekbend)
Infantry (Lima_12) at (Creekbend)
Infantry (Lima_13) at (Townsend)
Infantry (Lima_14) at (Creekbend)
Infantry (Mike_0) at (Prairie)
Infantry (Mike_1) at (Prairie)
Infantry (Mike_2) at (Prairie)
Infantry (Mike_3) at (Summit)
Infantry (Mike_4) at (Prairie)
Infantry (Mike_5) at (Pineside)
Infantry (Mike_6) at (Summit)
Infantry (Mike_7) at (Prairie)
Infantry (Mike_8) at (Prairie)
Infantry (Mike_9) at (Prairie)
Infantry (Mike_10) at (Prairie)
```

```

Infantry (Mike_11) at (Prairie)
Infantry (Mike_12) at (Pineside)
Infantry (Mike_13) at (Summit)
Infantry (Mike_14) at (Summit)

Location Network (location - neighbors):
Eastfield - ['Seabreeze', 'Skyline', 'Moonlight', 'Centerville']
Seabreeze - ['Eastfield', 'Meadow', 'Hillcrest', 'Crestview']
Skyline - ['Eastfield', 'Townsend', 'Bayview', 'Creekbend']
Moonlight - ['Eastfield', 'Summit', 'Pineside', 'Prairie']
Centerville - ['Eastfield', 'Riverbend', 'Lakeside', 'Sunnyside']
Meadow - ['Seabreeze', 'Lakeside', 'Prairie']
Hillcrest - ['Seabreeze', 'Sunnyside', 'Creekbend']
Crestview - ['Seabreeze']
Townsend - ['Skyline', 'Bayview']
Bayview - ['Skyline', 'Pineside', 'Lakeside', 'Riverbend', 'Townsend']
Creekbend - ['Skyline', 'Hillcrest']
Summit - ['Moonlight', 'Pineside']
Pineside - ['Moonlight', 'Summit', 'Bayview']
Prairie - ['Moonlight', 'Meadow']
Riverbend - ['Centerville', 'Bayview']
Lakeside - ['Centerville', 'Bayview', 'Meadow']
Sunnyside - ['Centerville', 'Hillcrest']

```

## D Methods

### D.1 Travel Planner

```

{
    "method1": {
        "task": "process user request",
        "subtasks": {
            "subtask1": "take notes on origin, destination, departure date,
                        ↳ returning date and other preferences",
            "subtask2": "choose a departing flight",
            "subtask3": "choose a returning flight",
            "subtask4": "choose an accommodation"
        },
        "effect": "answer.txt contains a solution in the correct format that
                  ↳ follows the problem specification",
        "effect_files": {
            "file1": "answer.txt",
            "file2": "files/problem_specification.txt"
        }
    },
    "method2": {
        "task": "take notes on origin, destination, departure date, returning
                ↳ date and other preferences",
        "subtasks": {
            "subtask1": "take notes on the problem specification file"
        },
    }
}

```

```

"effect": "notes contains same details as request and contains origin,
↪ destination, departure date and returning date for the trip",
"effect_files": {
    "file1": "files/notes.txt",
    "file2": "files/request.txt"
}
},
"method3": {
    "task": "choose a departing flight",
    "effect": "notes contains details about a flight number and cost for a
↪ flight from origin to destination on departure date consistent with
↪ request preferences",
    "subtasks": {
        "subtask1": "understand flights tool"
    },
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/problem_specification.txt",
        "file3": "files/request.txt"
    }
},
"method4": {
    "task": "choose a returning flight",
    "effect": "notes contains details about a flight number and cost for a
↪ returning flight consistent with request preferences",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/problem_specification.txt",
        "file3": "files/request.txt"
    }
},
"method5": {
    "task": "choose an accommodation",
    "effect": "notes contains details about an accomodation consistent with
↪ request preferences",
    "subtasks": {
        "subtask1": "understand accommodation tool"
    },
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/problem_specification.txt",
        "file3": "files/request.txt"
    }
},
"method6": {
    "task": "understand flights tool",
    "effect": "notes contains details about flights tool methods inputs and
↪ outputs",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/tools_specification.txt"
    }
},
"method7": {
    "task": "understand accommodation tool",

```

```

    "effect": "notes contains details about accommodation tool methods
    ↳ inputs and outputs",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/tools_specification.txt"
    }
},
"method8": {
    "task": "take notes on the problem specification file",
    "effect": "notes contain the information from the problem specification
    ↳ file",
    "effect_files": {
        "file1": "files/problem_specification.txt",
        "file2": "files/notes.txt"
    }
}
}

```

## D.2 Recipe Generator

```

{
    "method1": {
        "task": "process user request",
        "subtasks": {
            "subtask1": "take notes on user request and problem specification",
            "subtask2": "understand the tools specification"
        },
        "effect": "the answer to the user request can be found in answer.txt",
        "effect_files": {
            "file1": "answer.txt",
            "file2": "files/request.txt",
            "file3": "files/problem_specification.txt"
        }
    },
    "method2": {
        "task": "solve user request",
        "effect": "a valid recipe is chosen",
        "effect_files": {
            "file1": "files/request.txt",
            "file2": "files/problem_specification.txt",
            "file3": "files/notes.txt"
        }
    },
    "method3": {
        "task": "understand the tools specification",
        "effect": "details of provided tools and correct module import path are
        ↳ recorded in notes",
        "effect_files": {
            "file1": "files/tools_specification.txt",
            "file2": "files/notes.txt"
        }
    },
    "method4": {

```

```

    "task": "take notes on the problem specification file",
    "effect": "notes contain the information from the problem specification
    ↳ file",
    "effect_files": {
        "file1": "files/problem_specification.txt",
        "file2": "files/notes.txt"
    }
},
"method5": {
    "task": "take notes on user request",
    "effect": "notes contain the information from the user request",
    "effect_files": {
        "file1": "files/request.txt",
        "file2": "files/notes.txt"
    }
},
"method6": {
    "task": "take notes on user request and problem specification",
    "effect": "notes contain information from the user request and problem
    ↳ specification files",
    "effect_files": {
        "file1": "files/request.txt",
        "file2": "files/problem_specification.txt",
        "file3": "files/notes.txt"
    }
}
}

```

### D.3 Blocks World

```

{
    "method1": {
        "task": "process user request",
        "subtasks": {
            "subtask1": "take notes on problem specification",
            "subtask2": "take notes on user request",
            "subtask3": "unstack all blocks"
        },
        "effect": "answer.txt contains a solution to the user request in the
        ↳ correct format",
        "effect_files": {
            "file1": "files/problem_specification.txt",
            "file2": "files/request.txt",
            "file3": "answer.txt"
        }
    },
    "method2": {
        "task": "take notes on problem specification",
        "effect": "notes contain the information from the problem specification
        ↳ file",
        "effect_files": {
            "file1": "files/problem_specification.txt",

```

```

        "file2": "files/notes.txt"
    }
},
"method3": {
    "task": "solve the user request",
    "effect": "notes contain steps to solve the user request",
    "effect_files": {
        "file1": "files/request.txt",
        "file2": "files/problem_specification.txt",
        "file3": "files/notes.txt"
    }
},
"method4": {
    "task": "unstack all blocks",
    "effect": "notes contain the intermediate steps to unstack all blocks
    ↳ such that they are all clear and no blocks are being held",
    "effect_files": {
        "file1": "files/request.txt",
        "file2": "files/problem_specification.txt",
        "file3": "files/notes.txt"
    }
},
"method5": {
    "task": "take notes on user request",
    "effect": "notes contain a copy of the user request about the initial
    ↳ condition and goal",
    "effect_files": {
        "file1": "files/request.txt",
        "file2": "files/notes.txt"
    }
}
}
}

```

#### D.4 Unit Movement

```

{
    "method1": {
        "task": "process user request",
        "subtasks": {
            "subtask1": "take notes on problem specification",
            "subtask2": "take notes on user request",
            "subtask3": "group units",
            "subtask4": "move units"
        },
        "effect": "the answer to the user request can be found in answer.txt in
        ↳ the correct format",
        "effect_files": {
            "file1": "answer.txt",
            "file2": "files/request.txt",
            "file3": "files/problem_specification.txt"
        }
    },
    "method2": {

```

```

"task": "move units",
"subtasks": {
    "subtask1": "move group 1",
    "subtask2": "move group 2",
    "subtask3": "move group 3"
},
"effect": "notes contain movement actions to achieve the user request",
"effect_files": {
    "file1": "files/request.txt",
    "file2": "files/problem_specification.txt",
    "file3": "files/notes.txt"
}
},
"method3": {
    "task": "move group 1",
    "effect": "notes contain moves for units in group 1",
    "effect_files": {
        "file1": "files/notes.txt"
    }
},
"method4": {
    "task": "move group 2",
    "effect": "notes contain moves for units in group 2",
    "effect_files": {
        "file1": "files/notes.txt"
    }
},
"method5": {
    "task": "move group 3",
    "effect": "notes contain moves for units in group 3",
    "effect_files": {
        "file1": "files/notes.txt"
    }
},
"method6": {
    "task": "group units",
    "effect": "notes contain a grouping of units into three distinct groups
    ↳ and a group level strategy for moving each group to achieve the
    ↳ request goal",
    "effect_files": {
        "file1": "files/request.txt",
        "file2": "files/notes.txt"
    }
},
"method7": {
    "task": "take notes on user request and problem specification",
    "effect": "notes contain information from the user request and problem
    ↳ specification files",
    "effect_files": {
        "file1": "files/request.txt",
        "file2": "files/problem_specification.txt",
        "file3": "files/notes.txt"
    }
},
"method8": {

```

```

        "task": "take notes on user request",
        "effect": "notes contain information from the user request file",
        "effect_files": {
            "file1": "files/request.txt",
            "file2": "files/notes.txt"
        }
    },
    "method9": {
        "task": "take notes on problem specification",
        "effect": "notes contain information from the problem specification
        ↳ file",
        "effect_files": {
            "file1": "files/problem_specification.txt",
            "file2": "files/notes.txt"
        }
    }
}

```

## E LLM-generated Task Networks

### E.1 Blocks World

```

{
    "method1": {
        "task": "process user request",
        "subtasks": {
            "subtask1": "formulate plan",
            "subtask2": "write final plan to answer.txt"
        },
        "effect": "the answer to the user request can be found in answer.txt in the
        ↳ correct format",
        "effect_files": {
            "file1": "answer.txt",
            "file2": "files/request.txt",
            "file3": "files/problem_specification.txt"
        }
    },
    "method2": {
        "task": "formulate plan",
        "subtasks": {
            "subtask1": "take notes on problem and request",
            "subtask2": "generate plan to clear misplaced blocks",
            "subtask3": "generate plan to build goal stacks"
        },
        "effect": "the full plan to solve the blocksworld problem is stored in
        ↳ notes.txt",
        "effect_files": {
            "file1": "files/notes.txt",
            "file2": "files/request.txt"
        }
    },
    "method3": {
        "task": "take notes on problem and request",

```

```

    "subtasks": {
        "subtask1": "take notes on problem specification",
        "subtask2": "take notes on user request"
    },
    "effect": "notes contain the initial state, goal state, and rules of the
    ↵ problem",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/request.txt",
        "file3": "files/problem_specification.txt"
    }
},
"method4": {
    "task": "take notes on problem specification",
    "effect": "notes contain the rules of the blocksworld problem",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/problem_specification.txt"
    }
},
"method5": {
    "task": "take notes on user request",
    "effect": "notes contain the initial and goal block configurations",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/request.txt"
    }
},
"method6": {
    "task": "generate plan to clear misplaced blocks",
    "effect": "notes contains the sequence of unstack and put actions to move
    ↵ all misplaced blocks to the table",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/request.txt"
    }
},
"method7": {
    "task": "generate plan to build goal stacks",
    "effect": "notes contains the sequence of pick and stack actions to build
    ↵ the goal configuration from the blocks on the table",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/request.txt"
    }
},
"method8": {
    "task": "write final plan to answer.txt",
    "effect": "the final, combined plan from notes.txt is written to answer.txt
    ↵ in the correct format",
    "effect_files": {
        "file1": "answer.txt",
        "file2": "files/notes.txt"
    }
}

```

```
}
```

## E.2 Unit Movement

```
{
  "method1": {
    "task": "process user request",
    "effect": "the answer to the user request can be found in answer.txt in the
    ↵ correct format",
    "effect_files": {
      "file1": "answer.txt",
      "file2": "files/request.txt",
      "file3": "files/problem_specification.txt"
    },
    "subtasks": {
      "subtask1": "compose actions",
      "subtask2": "write answer file"
    }
  },
  "method2": {
    "task": "compose actions",
    "effect": "notes.txt contains the JSON list of unit actions for the turn",
    "effect_files": {
      "file1": "files/notes.txt",
      "file2": "files/request.txt",
      "file3": "files/problem_specification.txt"
    },
    "subtasks": {
      "subtask1": "take notes on request and spec",
      "subtask2": "plan strategy",
      "subtask3": "write actions to notes"
    }
  },
  "method3": {
    "task": "take notes on request and spec",
    "effect": "notes.txt contains relevant information extracted from the
    ↵ request and problem specification",
    "effect_files": {
      "file1": "files/notes.txt",
      "file2": "files/request.txt",
      "file3": "files/problem_specification.txt"
    },
    "subtasks": {
      "subtask1": "read request",
      "subtask2": "read problem specification",
      "subtask3": "write notes"
    }
  },
  "method4": {
    "task": "read request",
    "effect": "the contents of files/request.txt have been read and can be used
    ↵ for note taking",
    "effect_files": {
```

```

        "file1": "files/request.txt"
    },
},
"method5": {
    "task": "read problem specification",
    "effect": "the contents of files/problem_specification.txt have been read
    ↳ and can be used for note taking",
    "effect_files": {
        "file1": "files/problem_specification.txt"
    }
},
"method6": {
    "task": "write notes",
    "effect": "files/notes.txt now contains extracted information from the
    ↳ request and specification",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/request.txt",
        "file3": "files/problem_specification.txt"
    }
},
"method7": {
    "task": "plan strategy",
    "effect": "files/notes.txt is updated with a high-level movement strategy
    ↳ for the infantry units",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/request.txt",
        "file3": "files/problem_specification.txt"
    }
},
"method8": {
    "task": "write actions to notes",
    "effect": "files/notes.txt now contains the final JSON list of unit
    ↳ actions",
    "effect_files": {
        "file1": "files/notes.txt",
        "file2": "files/request.txt",
        "file3": "files/problem_specification.txt"
    }
},
"method9": {
    "task": "write answer file",
    "effect": "answer.txt contains the JSON list of unit actions in the required
    ↳ format",
    "effect_files": {
        "file1": "answer.txt",
        "file2": "files/notes.txt",
        "file3": "files/request.txt",
        "file4": "files/problem_specification.txt"
    }
}
}

```

### E.3 Prompt for generating task networks

The following prompt was used as input to Gemini 2.5 Pro to generate the two LLM-generated task networks above. The empty section labeled task specification is filled with the Problem Spec file for the domain being tested.

```
There is a system that processes commands in order to solve a given task. The
↪ task is specified in the problem specification document and the task
↪ instance in the request file. An agent is given the following actions to
↪ perform:

1. Read: reads [action_arg1]. Line numbers will be displayed.
Read [action_arg1]

2. Write: overwrites text file [action_arg1] with [action_arg2].
Write [action_arg1] [action_arg2]

3. Append: appends the text file [action_arg1] with [action_arg2].
Append [action_arg1] [action_arg2]

4. Verify: Send a request to check if current task has been achieved.
Verify

The agent is allowed to read and write the following file (r/w):
files/notes.txt (r/w): the contents of this file will always be displayed in the
↪ notes section down below.
answer.txt (r/w): a file to hold the final answer to user request

The agent is allowed to access the following read-only files (r):
files/request.txt (r): the user request
files/problem_specification.txt (r): a problem specification document

The agent can only see the contents of the last action it performed (e.g.
↪ observations are Markovian), so the agent needs to take frequent notes. The
↪ agent will follow a task network with nodes of the form:
{
    "method1": {
        "task": "process user request",
        "subtasks": {
            "subtask1": "[other subtask]",
            "subtask2": "[other subtask]"
        },
        "effect": "the answer to the user request can be found in answer.txt in
↪ the correct format",
        "effect_files": {
            "file1": "answer.txt",
            "file2": "files/request.txt",
            "file3": "files/problem_specification.txt"
        }
    },
    ...
    "method7": {
        "task": "take notes on [example1] and [example2]",
        "subtasks": {
            "subtask1": "[other subtask]",
            "subtask2": "[other subtask]"
        }
    }
}
```

```

},
"effect": "notes contain information from [example1] and [example2]
    ↳ documents",
"effect_files": {
    "file1": "files/[example1].txt",
    "file2": "files/[example2].txt",
    "file3": "files/notes.txt"
}
}
}

```

which specify the name of the task, the effect of the task, the files that can  
 ↳ be used to confirm the result of the task and possibly a list of subtasks  
 ↳ that may need to be completed. For the following problem, please write a  
 ↳ task network for nodes of the above format to assist the agent in solving  
 ↳ the task.

General instructions:

1. Please respond only with the task network and no additional text.
2. The agent will need to use the verify action to move onto the next task, so  
 ↳ no verification subtasks are needed.
3. Make sure that the keys for subtasks are numbered backwards in the order that  
 ↳ the agent should complete the task in.
4. All subtasks should have a corresponding task node. The subtasks component is  
 ↳ not needed for leaf nodes of the task network.
5. The values in the subtask list "[other subtask]" should correspond to the  
 ↳ name of the subtask it is referring to (e.g. "take notes on xyz").
6. The top level task should be called "process user request".
7. Effect files denote the files needed for the verifier to check if a given  
 ↳ task node is completed (e.g. if the task is to write an answer, the problem  
 ↳ specification and request should be included as effect files so the answer  
 ↳ can be checked against it).
8. Make sure each task is numbered task[number].

The task specification:

```
*** TASK SPEC START ***
{}
*** TASK SPEC END ***
```

## F Verification

To verify whether an agent LLM has successfully completed a task, we send a request to a verifier LLM to check if the effect of the task network node has been satisfied. Only the contents of the effect files for a given task node are placed into the context of the verifier LLM. This allows for more consistent evaluation (and alleviates long context issues in using smaller LLMs for verification).

### F.1 Verification Code (success checking in Fig 2)

```

with open('verification_prompt.txt') as f:
    lines = f.readlines()
    prompt = ''.join(lines)

task_specification = get_current_task_network_node(current_task)

```

```

check = task_specification["effect"]

files_to_check = ""
counter = 1
while "file" + str(counter) in task_specification["effect_files"]:
    file_to_check = task_specification["effect_files"]["file" + str(counter)]
    files_to_check += "## Here are the contents of " + file_to_check[:-4] + ":" +
        "\n"
    with open(files_directory + "/" + file_to_check) as f:
        lines = f.readlines()
        files_to_check += ".join(lines) + "\n"
    counter += 1

prompt = prompt.format(check, files_to_check)
last_output = llm_model.instruct_query(prompt)

success = True if "PASS: TRUE" in last_output else False
if success:
    predicates.add(stack[-1][0])
    stack = stack[:-1]
    if len(stack) == 0:
        break
    current_task = update_task()

```

## F.2 Verification Prompt

This verification prompt was tuned with meta-prompting on GPT-4.

```

**Here is your task:**  

Your goal is to determine if the provided files satisfy the specified effect.  

→ Follow these steps carefully:  
  

### **Task Specification**  

The effect that needs to be satisfied is:  

{}  
  

### **File Contents**  

Here are the contents of the files provided:  

{}  
  

### **Instructions**  

1. **Understand the Effect:** Carefully read the effect and identify its  

   → conditions.  

2. **Analyze the File Contents:** Compare the file contents explicitly against  

   → the conditions of the effect.  

   - Break down the effect into specific criteria (e.g., "summarized version"  

     → means the main ideas in "request" should be present in "notes").  

   - Check whether these criteria are satisfied in the files.  

3. **Provide a Detailed Analysis:** State which criteria are satisfied and which  

   → are not, with reasons based on the file contents.  
  

### **Your Response Format**  

Begin your response with:

```

```  
ANALYSIS:  
<Your detailed analysis of whether the files satisfy the effect, with explicit  
→ reasoning.>  
```

End your response with one of the following lines:

```  
PASS: TRUE  
```

\*only if all conditions of the effect are satisfied.\*

```  
PASS: FALSE  
```

\*if any condition of the effect is not satisfied.\*

---