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*
Iconium Labs
Tempe, AZ, United States
makro@iconiumlabs.com

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

ABSTRACT

Large language models (LLMs) often struggle when performing agentic tasks without substantial tool support, prom-pt 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

ACM Reference Format:
Vincent Hsiao, Mark Roberts*, and Leslie Smith. 2026. Procedural Knowledge Improves Agentic LLM Workflows. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 9 pages. https://doi.org/10.65109/VEVZ5917

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 [2, 29], 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]).

*Work performed while at NRL.

This work is licensed under a Creative Commons Attribution International 4.0 License.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). https://doi.org/10.65109/VEVZ5917

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 complete consistently and correctly. Part of the gap is that LLMs are mostly constructed to predict the next token, while planning may require 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. 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 planning task 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, procedural 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 an agentic LLM as a large language model framework that can act, use tools, and autonomously plan to accomplish tasks. Many problems cannot be solved in a single forward pass of an LLM, motivating the development of structured workflow techniques that use multiple LLM passes. Early approaches such as Chain-of-Thought (CoT) [34] used prompt engineering to elicit multi-step reasoning. This concept has been extended to non-linear structures like Tree-of-Thoughts (ToT) [38] and X-of-Thoughts (XoT) [21], which enable iterative refinement with verification and branching search. Complementary to these reasoning strategies, prompt chaining [35] allows the output of one prompt to become the input to another. Early community-driven agent frameworks like AutoGPT [37] 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.

One of the first academic approaches in this area was the ReAct framework [39], where agent reasoning choices are interleaved with action selection in order to interact with external environments. Another notable contribution was ExpeL [42], 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 Reflexion (Fig. 2 in [25]). The Reflexion agent employs an LLM (denoted 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 compiles a research strategy from the user into a list of steps. The output structure is enforced by incorporating code comments and yield directives that specify the goal of each 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 [33] and there are numerous papers covering various aspects of this field of research [12, 20]. Early demonstrations included Toolformer [23], 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 [41], an orchestrator agent creates a plan that is executed by specialized sub-agents. Unlike our explicit task network, 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 [40] 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 [43] employs an LLM to translate planning problems in natural language into PDDL and then iteratively calls the external planner to verify LLM-generated 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 [28] [27] [14]. There have been impressive gains in this area of research; for example, a recent notable example is by Verma et al. [32] which claims up to 94% planning accuracy using a fine-tuning approach paired with CoT. Despite these gains, planning remains a challenging task [26] [31] with many results showing that LLMs

still have many weaknesses on 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.

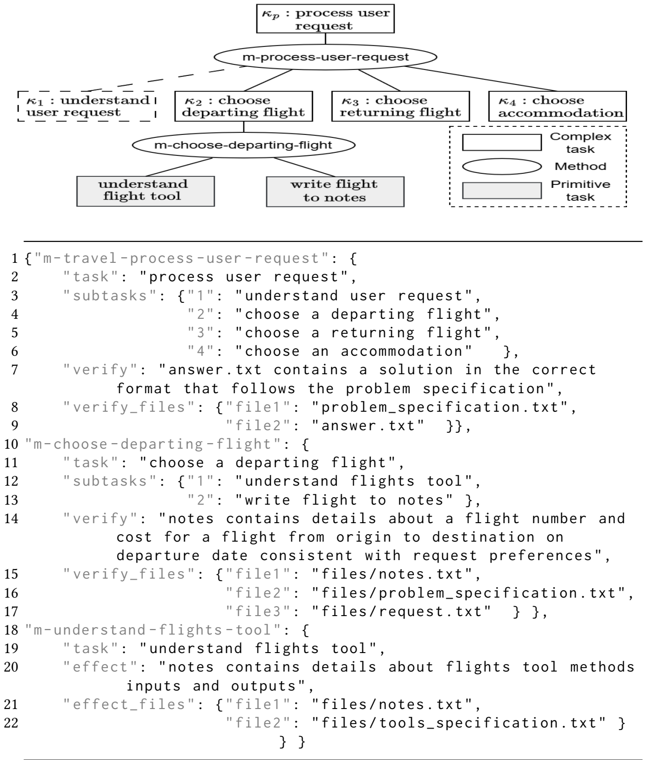
Another similar work is ChatHTN [22], which uses LLMs to assist in HTN planning. This is exactly opposite from our work where HTNs are used to guide LLMs through complex tasks. Unlike other works, our focus is not directly on the ability of an LLM to produce a plan (i.e., a sequence of steps solving 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 κ_p into $\langle \kappa_1, .., \kappa_4 \rangle$ (Lines 3-6), where the κ_1 (dashed box) has passed the check to `verify` (Lines 7-9, formalized later). Method `m-choose-departing-flight` (Lines 10-17) decomposes κ_2 into: $\langle$`understand flights tool`, `write flight to notes`$\rangle$, further decomposed by `m-understand-flights-tool` (Lines 18-22).

For a set of methods $\mathcal{M}$, a method $m \in \mathcal{M}$ is a tuple (*head*, *task*, *pre*, *subtasks*), where the *head* is the name and any paramaters 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 $\langle \kappa_1, \kappa_2, .., \kappa_n \rangle$. 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).




```
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    "task": "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: A simplified 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 κ_1 indicates it has passed `verify`. Bottom: Example simplified methods.

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

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

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

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

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

3.1 Initial Analysis

In this work, we investigate the use of an agentic LLM system as the policy π that operates within this MDP framework. We show that divide-and-conquer mechanisms such as task decomposition can be used to constrain π 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 $\mathcal{A} : \mathcal{S} \times K \rightarrow C$ 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 $(a_0, \ldots, a_{H-1})$ 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 θ, which governs action selection:

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

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, $(\kappa_0, \ldots, \kappa_{n-1})$. Let $P_\theta(\kappa_j)$ be the probability of successfully completing subtask j. We make a standard planning assumption of subtask independence where subtask completion depends only on its preconditions being met, regardless of prior execution history[1]. 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 κ_j, the assembly function $\mathcal{A}(s_i, \kappa_j)$ 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:

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

Since the probability of success for a subtask κ_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.

[1] In practice, the soundness of this assumption depends on the verifier chosen and the stringency of task preconditions.

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 requests.
- r `Request` contains problem instance details to be solved such as the initial configuration of the problem and the goal configuration.
- 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 $C = \mathcal{A}(e_1, e_2, ..., e_k)$ containing one or more elements e_i. For `ProcLLM` this includes the action context $C_a = \mathcal{A}_a(s, \texttt{trace}, \kappa))$, and the verify context $C_v = \mathcal{A}_v(s, \texttt{trace}, \kappa)$.

Actions 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.

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

`read` (`file`): copies `file` into `Notes`
`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. 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 differ slightly from this simple form. For the methods in this system, no

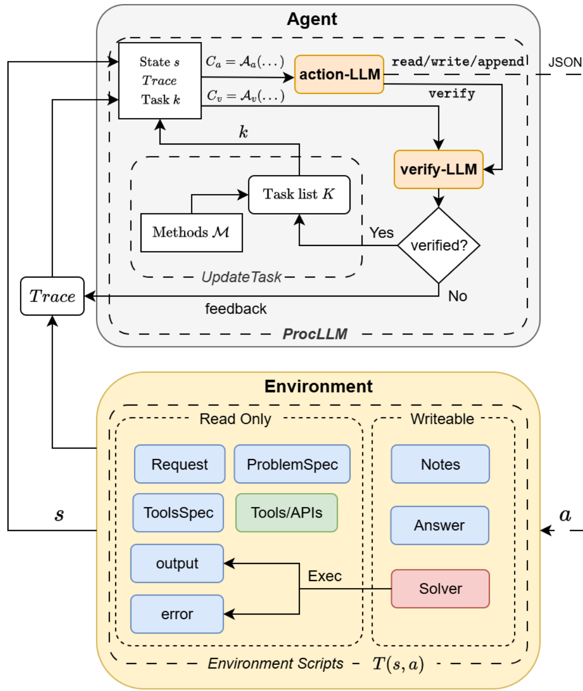


Figure 2: System overview, labeled with components of the MDP. Blue boxes denote text files, orange boxes denote LLMs, red boxes denote python files, and green boxes denote API files (scripts/databases/etc.).

preconditions are specified. In addition to the $(head, task, subtask)$, methods in `ProcLLM` also include details for the `verify` step.

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 κ. 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

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

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 [36], 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 [30], requires rearranging stacks of blocks. **Unit Movement (UM)**, another synthetic benchmark, requires moving units around a graph.

	<u>Solver</u>?	<u>Tools</u>?	Problem Type	Source
TP	Y	Y	Tool Use	[36]
RG	Y	Y	Algorithmic Tool Use	This paper
BW			Planning	[30]
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 problem instance; it functions similar to a vanilla reflexion agent [25]. Experiments are performed

Table 1: Comparison: Mean success rate on subtasks in reduced Travel Planner benchmark (Verification set from [36])

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	**0.395**

on an AMD EPYC 7H12 64 core CPU with the LLMs being served through ollama hosted on A100 GPUs.

5.1 Travel Planning

This tool-use problem is based on the TravelPlanner benchmark [36], 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 [36]. 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 [36] 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 [36]. `ProblemSpec` specifies that the agent should only fulfill parts of the request in order (in contrast to [36]).

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`.

5.2 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`.

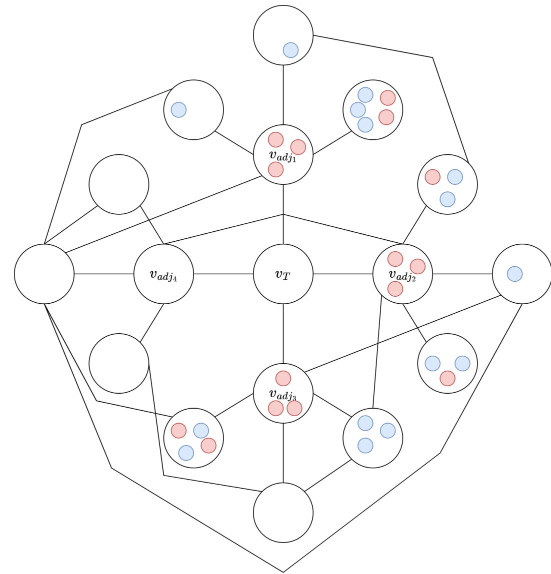


Figure 3: An example problem instance for the unit movement domain showing the initial starting position (blue circles) and a valid solution (red circles)

5.3 Blocks World (BW)

This problem adapts the BlocksWorld benchmark from PlanBench [30], 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 4 (top) 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 `No-TN` and is often close in performance to 120b `Human-TN`.

5.4 Unit Movement (UM)

In this task, for each problem instance we generate a graph G with the following properties (Figure 3 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).

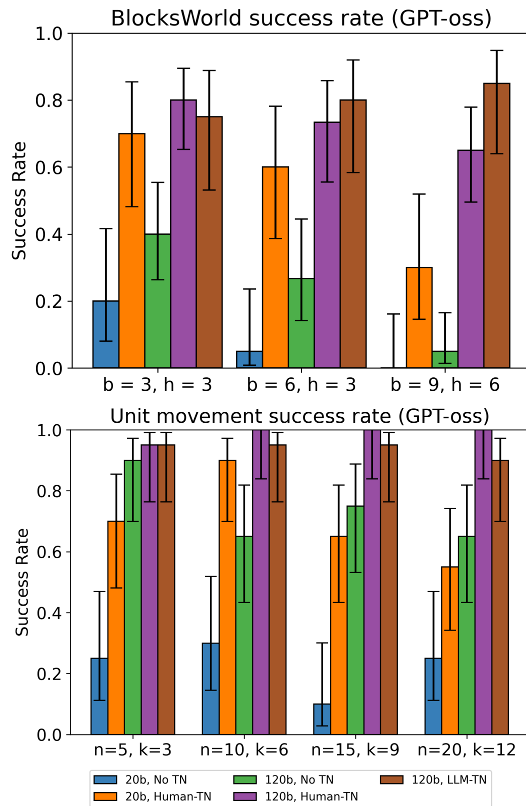


Figure 4: Success rates for BlockWorld and Unit Movement domains, evaluated across increasing problem complexity (b starting blocks and h final stack height for BW, number of units n for UM). Bars show mean success rates, with 95% Wilson confidence intervals.

- 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 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 4 (bottom) 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 4, 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? Using a task network typically increases the average time per iteration while decreasing the total number of iterations. Figure 5 shows runtime statistics for GPT-oss in `ProcLLM` across the evaluation benchmarks. 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

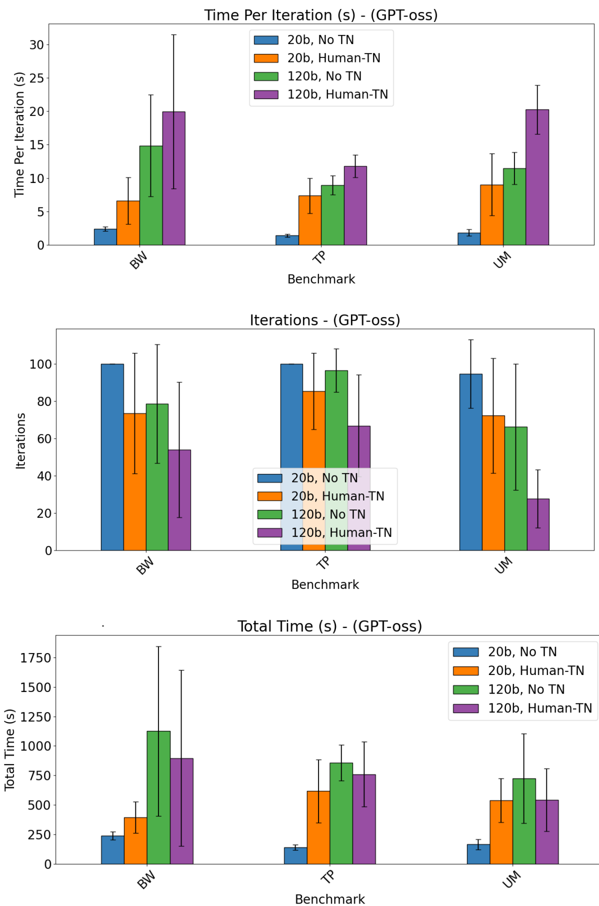


Figure 5: A comparison of average runtime statistics for GPT-oss across different benchmarks (BW - BlocksWorld, TP - Travel Planner, UM - Unit Movement).

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 shortcutting behavior mirrors earlier findings ([3]) that LLMs can default to superficial strategies unless constrained externally.

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.

6.1 On LLMs and Code Safety

In our experiments with agentic LLMs, one concerning behavior we noticed was the execution of out-of-scope code in Python. To sandbox our agent LLM, we set up the environment that processes agent actions to allow the LLM to only read and write to a pre-specified list of approved files. However, we did not limit any code generation capabilities to allow for greater problem-solving creativity. This lack of limitations led to unforeseen side effects. In one instance, the agentic LLM generated code that depended on Python libraries that were not installed in the development environment it was running in. Instead of changing the code to a supported library, it attempted to programmatically install the Python library within the Python script being written. It is important to emphasize that prompt restrictions (or even restrictions imposed by a task network) are not a sufficient guard against unintended agent behavior. Reproduction of this work should be carried out in a carefully sandboxed environment.

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 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. 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., [24]) or fine-tuning (e.g., [32]) to improve the distillation of task networks from domain documents

ACKNOWLEDGMENTS

We thank NRL for funding this research.

REFERENCES

[1] Fahiem Bacchus and Froduald Kabanza. 2000. Using Temporal Logics to Express Search Control Knowledge for Planning. *AIJ* 116, 1 (Jan. 2000), 123–191.

[2] Fahiem Bacchus and Qiang Yang. 1994. Downward refinement and the efficiency of hierarchical problem solving. *AIJ* 71, 1 (1994), 43–100.

[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. *JAIR* 28 (Feb. 2007), 119–156.

[7] Christopher W. Geib and Robert P. Goldman. 2009. A Probabilistic Plan Recognition Algorithm Based on Plan Tree Grammars. *AIJ* 173, 11 (July 2009), 1101–1132.

[8] Malik Ghallab, Dana Nau, and Paolo Traverso. 2025. *Acting, Planning and Learning*. Cambridge University Press. Author preprint available at 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.

[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 Khizbullin, 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] Tengxiao Liu, Qipeng Guo, Yuqing Yang, Xiangkun Hu, Yue Zhang, Xipeng Qiu, and Zheng Zhang. 2023. Plan, verify and switch: Integrated reasoning with diverse x-of-thoughts. *arXiv preprint arXiv:2310.14628* (2023).

[22] Hector Munoz-Avila, David W Aha, and Paola Rizzo. 2025. ChatHTN: Interleaving Approximate (LLM) and Symbolic HTN Planning. *arXiv preprint arXiv:2505.11814* (2025).

[23] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, 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).

[24] 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).

[25] 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.

[26] 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).

[27] 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.

[28] 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.

[29] Austin Tate. 1977. Generating Project Networks. In *Proc. of IJCAI (IJCAI'77)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 888–893.

[30] 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.

[31] 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).

[32] 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

[33] 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).

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[35] 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.

[36] 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).

[37] Hui Yang, Sifu Yue, and Yunzhong He. 2023. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224* (2023).

[38] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems* 36 (2023), 11809–11822.

[39] 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).

[40] 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.

[41] 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).

[42] 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.

[43] Zhehua Zhou, Jiayang Song, Kunpeng Yao, Zhan Shu, and Lei Ma. 2024. Isrllm: 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.