THREAD: Thinking Deeper with Recursive Spawning

Philip Schroeder, **Nathaniel Morgan**, **Hongyin Luo**, **James Glass**

MIT Computer Science and Artificial Intelligence Lab, Cambridge MA, USA

Abstract

Large language models (LLMs) have shown impressive capabilities across diverse settings, but still struggle as the length and complexity of the context increases. To address this challenge, we propose Thinking Recursively and Dynamically (ThReaD). THREAD frames model generation as a thread of execution that, based on the context, can run to completion or dynamically spawn new threads. By spawning, threads can offload work (e.g., thinking, retrieving information) to child threads, which only return tokens needed for the parent thread to do its work. We apply THREAD in the settings of LLM task solving and question answering, where the dynamic threading allows the model to recursively decompose the given task or question into progressively simpler subproblems that can be solved by separate child threads. We test THREAD, implemented using a few-shot learning approach, on diverse benchmarks for agent tasks and data-grounded question answering. THREAD achieves state-of-the-art performance with GPT-4 and GPT-3.5 on these benchmarks, including ALFWorld, TextCraft, and WebShop, along with two new benchmarks, DataCommons QA and MIMIC-III ICU QA. In addition, THREAD outperforms existing frameworks by 10% to 50% absolute points with smaller models, including Llama-3-8b and CodeLlama-7b.

1 Introduction

Large Language Models (LLMs) have shown success in diverse settings (Wei et al., 2022; Huang and Chang, 2023), but their performance degrades as context length and complexity grows (Dziri et al., 2023; Liu et al., 2024; Qin et al., 2023). This constraint limits their efficacy in settings that require more work (thinking, retrieving information, analyzing, etc.) than can fit into a concise line of generation. To address this limitation, we propose Thinking Recursively and Dynamically (ThReaD).

THREAD is a general framework where model generation is treated as a thread of execution that, based on the context, can independently run to completion or dynamically spawn new threads in a recursive fashion. When a thread spawns a child, the child generates conditioning on context that derives from the parent's token sequence. Spawning child threads allows work, such as internal thinking or interacting with an external environment, to be completed on behalf of the parent, without directly adding to the parent's context. Child threads return only the information (tokens) needed for the parent to complete its work. In effect, spawning enables the model to dynamically adapt the amount of work or intermediate computational steps used to produce different parts of its token sequence.

The synchronization and spawning mechanisms of THREAD can vary based on the setting. Figure 1 shows an example of applying THREAD in a synchronous setting, where a parent waits for a child in a form analogous to `Thread.join()` in multi-threaded programming. In this example, a parent thread pauses generation until the child execution completes and the child returns output tokens that are appended directly to the token sequence of the parent before the parent proceeds generation.

THREAD improves the flexibility of model generation, allowing the model to adapt, through recursive spawning, the amount of work it does based on the given problem, without overextending its context. Importantly, this unfolds with limited need for explicit rules or hard coded logic. Instead, THREAD relies on the model's ability to infer the appropriate continuation given the context of each thread at the time of execution. Finally, THREAD is agnostic to the type of sequence. Depending on the underlying model, THREAD can be applied in varied settings, including multi-modal applications, and fulfill varied purposes (e.g., carrying out calcu-

Correspondence to Philip Schroeder at pschro@mit.edu. Source code is available at https://github.com/philipmit/thread.

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 8418–8442

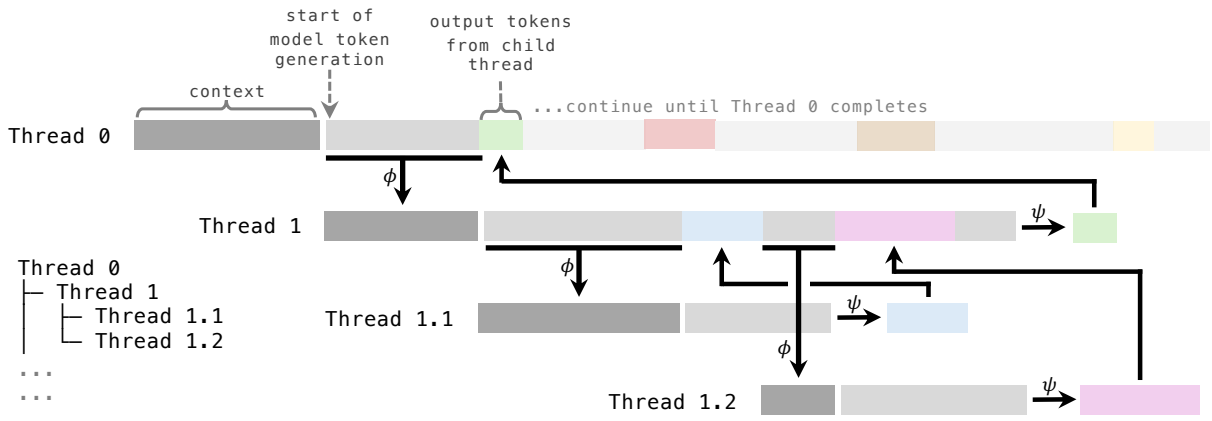


Figure 1: **THREAD with join synchronization.** THREAD frames model generation as an execution thread that can dynamically spawn new threads. In the example with join synchronization, when a thread spawns a child, it pauses generation until feedback is returned. Child threads generate starting from context derived from their parent's token sequence. When the child completes, it returns output tokens (colored bars), which are added to the context of the parent before it continues generating. ϕ and ψ are functions that control information flow from parent to child and from child to parent, respectively, by defining the tokens that are propagated based on the thread's token sequence.

lations, generating thoughts, retrieving information, robot manipulation).

In this paper, we consider THREAD in the settings of agent tasks (Figure 2a) and question answering (Figure 2b). In these settings, THREAD enables the LLM to recursively decompose a task, or question, into progressively simpler sub-problems that can be solved by child threads in a compartmentalized manner. A child thread infers the specifications of its sub-problem from its parent's token sequence. If needed, child threads can troubleshoot their sub-problem, and spawn their own threads, all without distracting their parent. We test THREAD, using a few-shot learning approach, on benchmarks consisting of question-answering and agent tasks. For these problems, we apply THREAD with join synchronization, allowing parent threads to see feedback from child threads before defining their next step. For example, in the agent setting, if the model is given the task of cleaning a bowl and putting it on the countertop, the main thread may spawn a child with the sub-task of finding a bowl. This child can then spawn further threads for handling the complexities of navigating to and checking different locations for the bowl, with each thread returning only the information needed for its parent to proceed with its sub-task. Based on the feedback with regard to finding the bowl, the main thread can spawn a new thread to re-attempt the sub-task or spawn threads to execute the remaining sub-tasks (i.e., washing the bowl, putting the bowl on the countertop).

THREAD has several advantages over existing frameworks in these settings. To start, THREAD addresses limitations of methods that require the model to solve the problem in one line of context (Yao et al., 2023; Shinn et al., 2024; Sun et al., 2023a) by allowing the model to dynamically offload work by spawning child threads. Further, unlike methods for task decomposition (Khot et al., 2023; Sun et al., 2023b; Prasad et al., 2023; Wang et al., 2024, 2023), THREAD enables the model to adapt its decision-making in real time as it receives feedback from each step, as depicted in Figure 2. Existing methods either do not allow for plan adaptation (Sun et al., 2023b; Wang et al., 2023) or only adapt plans by calling pre-existing sub-task handlers (Khot et al., 2023) or by adding more sub-steps within the existing plan when a sub-task fails (Prasad et al., 2023; Wang et al., 2024). Finally, THREAD provides a more unified framework compared to these methods, which require separate prompting mechanisms for separate planner and executor modules. Instead, THREAD can be implemented with the same few-shot prompt used for every thread at every step of the task completion.

We evaluate THREAD on agent tasks and data-grounded question answering. THREAD significantly outperforms prior methods, achieving state-of-the-art performance with GPT-4 and GPT-3.5 on ALFWorld, TextCraft, and WebShop, along with two new benchmarks, DataCommons QA and MIMIC-III ICU QA. THREAD also shows success with smaller models, outperforming prior methods by 10% to 50% absolute points across the benchmarks with Llama-3-8b and CodeLlama-7b.

2 THREAD Framework

THREAD frames model generation as an execution thread that, given the context, can run to completion or spawn new threads. A thread consists of generating with a model, G, given a specific starting context, c. For the main thread, c is the initial seed context, c_0 (e.g., context provided by a user). For each child thread, c derives from the parent's token sequence. A thread continues until it meets some termination criteria, such as a stop token.

Algorithm 1 THREAD with join synchronization

function THREAD(c, Y)
  while `True` **do**
    $Y = Y + G(c + Y)$
    if Y spawns a child thread **then**
      $Y = Y + \psi(\text{THREAD}(\phi(Y), [\,]))$
    else if Y ends the thread **then**
      return Y

2.1 THREAD with Join Synchronization

We present THREAD in a setting where, analogous to how the `join()` method is used in multithreaded programming, a parent thread waits for the execution of the child to complete before proceeding. When the child execution completes, the child's output is returned directly to the parent. In this setting, THREAD can be implemented using the recursive function shown in Algorithm 1. The function takes two inputs: the context for the thread, c, and the tokens generated so far by the thread, Y. We treat a token sequence as a list of tokens.

THREAD begins with $c = c_0$ and $Y = [\,]$. The model generates, $G(c + Y)$, one token at a time conditioning on the given context, c, appended with the growing sequence of generated tokens, Y.

If Y spawns a child thread, then a new thread is created with THREAD($\phi(Y), [\,]$) where Y for the child thread is initialized as an empty list and c is based on the token sequence of the parent, $\phi(Y)$. The output tokens of the child thread are appended to the parent's token sequence and the parent continues generating. If Y ends the thread, then the thread returns Y (its full token sequence).

The ϕ function defines the context for a child thread based on the full token sequence of the parent thread at the time the child is spawned, including tokens directly generated by the parent or returned as output from previous child threads of that parent. The ψ function defines the output tokens of a child thread based on its full token sequence at the time the thread ends, including tokens directly generated by the child or returned as output from threads that it spawned.

2.2 Alternative Synchronization and Spawning Mechanisms

Above, we show THREAD in a setting where a parent always waits for a child to complete and the child's output is always appended directly to the token sequence of the parent before the parent proceeds. However, depending on the setting, THREAD can involve varied mechanisms for synchronization and spawning and, in scenarios without sequential dependencies, can include asynchronous multithreading, allowing parent threads to continue generating without waiting for child threads to complete (improving overall efficiency).

2.3 Defining Parent-Child Information Exchange with ϕ and ψ

The functions ϕ and ψ control the propagation of information from parent to child and from child to parent, respectively. Similar to the synchronization and spawning mechanisms mentioned above, these functions can vary based on the setting in which THREAD is implemented. For example, ϕ and ψ can range from simple functions that compress or decompress information in the token sequence to entirely separate models that transform the sequence in more complex ways. In the section below, we describe how we define ϕ and ψ when implementing THREAD in the settings of agent tasks and question answering.

2.4 Dynamically Adapting Intermediate Work in Token Generation

The THREAD framework enables a thread of model generation to spawn a child thread to produce output tokens that can serve as future tokens for the parent. In effect, this enables the model to adapt, as needed, the amount of work or computational steps used to produce different tokens. The intermediate work added through recursive spawning can range from deeper thinking and reasoning to interacting with external environments or sources of information. As depicted in Figure 3, the work associated with each part of the parent's token sequence is represented by a thread tree organized based on connections between threads. These trees reflect how threads interact to form output tokens and the computational work associated with each component of the overall model generation.

Figure 2: When given a task (a) or question (b), THREAD can be used to help the model, through recursive spawning, decompose the problem into progressively simpler sub-problems that are solved by child threads. In these examples, the context for a child thread is based on the last line of the parent's token sequence.

3 Applying THREAD for Agent Tasks and Question Answering

In this paper, we test THREAD in the settings of question answering and agent tasks with LLMs, where THREAD enables the model to dynamically decompose the given problem into simpler sub-problems that are completed by separate threads. We apply THREAD using an in-context learning approach, where the same few-shot prompt is used for every thread at every step of the task completion or question answering. Since these problems benefit from the model adapting to feedback as it defines its next steps, we apply THREAD with the join synchronization described above.

Thread spawning and termination based on special stop tokens. We implement a spawning mechanism using a special stop token, ω_{listen}, which pauses the thread generation and marks the start of the output the thread expects from the child. The context for the child thread is defined by ϕ based on the token sequence of the parent that occurs before ω_{listen}. A thread ends when it generates the end token, ω_{end}. As exemplified in Figure 2, we use => as ω_{listen} and END as ω_{end}.

Implementing THREAD with few-shot learning. We leverage a few-shot learning approach to implement THREAD. The few-shot examples are comprised in a single prompt with examples of successful spawning and problem solving at different thread depths, including how the token ω_{listen} should be used. As a result, the approach does not require explicit rules to define when new threads should be spawned and how parent threads should respond to feedback from a child. Instead, these mechanisms are all implied by the examples provided in the few-shot prompt. Changing the spawning and parent-child dynamics simply requires modifying the prompt. All threads are provided the same few-shot prompt for every step of the problem completion. Thus, THREAD only requires generating a single prompt for a given setting. This prompt, q, is prepended to the context for each thread, $G(q + c + Y)$, forming the full context from which the thread will generate.

Defining ϕ and ψ. As described above, the functions ϕ and ψ control information flow from parent to child and from child to parent, respectively. The function ψ returns the result from a child thread based on its full token sequence. As depicted in Fig-

ure 2, we implement ψ as a function that returns the tokens included in the print statement at the end of the child thread's token sequence. In addition, ψ appends the token <= to the output to mark the end of the child's output within the parent's token sequence. We implement ϕ as a function that extracts the last line of the parent's token sequence to create the context for the child thread. To improve the ability of parent and child threads to efficiently organize, use, update, and propagate shared information, we leverage the coding-based skills of the model to define variables that represent pieces of information that are important for the given problem. For example, if the task is to find an item on an e-commerce site that has certain attributes, the model can define a variable, such as `obj_attributes`, and spawn a child to search for items with `obj_attributes` instead of having to list all of the attributes. Then, when ϕ processes the token sequence from the parent, it instantiates the variables, allowing the child to see, for example, the full list of attributes. We also test THREAD implemented without using these variables, with results shown in the appendix.

Threads performing actions in an environment. Threads can listen for feedback when performing an action in an environment the same way they listen for feedback from a child thread. They simply generate tokens that represent the action and then produce the token ω_{listen} to listen for feedback from the environment. The action, a, is then executed in the environment, E, and the output, $o = E(a)$, is appended to the thread's token sequence the same way it is done for the feedback from a child thread (Algorithm 2). The thread can then continue generating based on the feedback.

4 Experiments

We evaluate THREAD on 5 benchmarks for agent tasks and question answering using large and small models, including GPT-4, GPT-3.5, Llama-3-8b, Llama-2-7b, and CodeLlama-7b. For each benchmark, we use the same prompt for all models.

We include details regarding models, prompts, and experiments (including ablations) in the appendix. Examples of the prompts are shown in Appendix G. Details regarding the QA benchmarks are provided in Appendix F. We release all code and data at `https://github.com/philipmit/thread`.

4.1 ALFWorld

ALFWorld is a suite of text-based environments, implemented in TextWorld, designed to align with the embodied ALFRED benchmark (Shridhar et al., 2021). It includes 6 different task types that instruct the agent to accomplish a goal by interacting with a simulated household with actions defined in text. Following Yao et al. (2023), we evaluate an agent on 134 unseen evaluation games, including six task types: Pick, Clean, Heat, Cool, Look, and Pick2. Like previous methods, such as ReAct (Yao et al., 2023), we implement THREAD with one few-shot prompt per task. We also test THREAD using task-general prompting.

Results. THREAD significantly outperforms all prior methods (Tables 1 and 2), including those that require the agent to have access to external memory consisting of past experiences with the task. With GPT-4, THREAD achieves a combined success rate of 98.5% with 100% success in 4 of the 6 tasks. With GPT-3.5, THREAD shows a combined success rate of 95.5%, outperforming all methods by over 9% absolute points. With smaller models (Llama-3-8b and CodeLlama-7b), THREAD improves upon prior methods by 30% to 55% points.

Finally, Table 8 shows the results when testing THREAD with task-general prompting. To implement the task-general prompt, we split the prompt into one set of examples for the main thread and a second set of examples for all other threads (with the same sets of examples used for all tasks). We see that GPT-3.5 achieves the same combined success rate with the task-general prompting as it does with the task-specific prompting. Further, while the performance of Llama-3-8b and CodeLlama-7b degrades with task-general prompting, it remains a significant improvement over task-specific prompting with prior methods (Table 2).

4.2 TextCraft

TextCraft is a text-based environment inspired by the crafting component of Minecraft (Prasad et al., 2023). The tasks involve building Minecraft items with crafting commands using available resources from the environment. Following the work of Prasad et al. (2023), we evaluate THREAD on the test set containing 200 samples. We implement THREAD using a single few-shot prompt and compare its performance to previous methods used for TextCraft (Prasad et al., 2023; Wang et al., 2024; Shinn et al., 2024; Yao et al., 2023).

Table 1: **ALFWorld task-specific success rates (%).** Results are separated based on whether the method requires the agent to access to external memory. *As reported in Kagaya et al. (2024) (RAP) and Fu et al. (2024) (AutoGuide).

Model	Requires ext. mem.	Method	All	Pick	Clean	Heat	Cool	Look	Pick2
GPT-3.5	Yes	Reflexion (Shinn et al., 2024)	76.1	75.0	80.6	69.6	76.2	83.3	70.6
		AdaPlanner (Sun et al., 2023a)	82.8	91.7	87.1	82.6	**95.2**	50.0	82.4
		RAP* (Kagaya et al., 2024)	85.8	95.8	87.1	78.3	90.5	88.9	70.6
		AutoGuide* (Fu et al., 2024)	79.1	-	-	-	-	-	-
	No	ReAct (Yao et al., 2023)	53.7	45.8	48.4	69.6	66.7	55.6	35.3
		DecomP (Khot et al., 2023)	84.3	91.7	87.1	82.6	90.5	83.3	64.7
		ADaPT (Prasad et al., 2023)	82.1	87.5	83.9	78.3	90.5	83.3	64.7
		THREAD	**95.5**	**95.8**	**93.5**	**95.7**	**95.2**	**100**	**94.1**
GPT-4	Yes	RAP* (Kagaya et al., 2024)	94.8	95.8	90.3	**100**	**95.2**	**100**	88.2
	No	ReAct (Yao et al., 2023)	87.3	83.3	77.4	95.7	85.7	**100**	88.2
		DecomP (Khot et al., 2023)	89.6	87.5	87.1	91.3	90.5	94.4	88.2
		ADaPT (Prasad et al., 2023)	91.0	91.7	87.1	95.7	90.5	94.4	88.2
		THREAD	**98.5**	**100**	**100**	**100**	**95.2**	**100**	**94.1**

Table 2: ALFWorld success rates (%) for all tasks combined.

Requires ext. mem.	Method	Llama-3-8b	Llama-2-7b	CodeLlama-7b
Yes	Reflexion (Shinn et al., 2024)	25.4	11.2	27.6
	AdaPlanner (Sun et al., 2023a)	28.4	11.9	29.9
No	ReAct (Yao et al., 2023)	20.1	12.7	23.1
	DecomP (Khot et al., 2023)	37.3	14.2	33.6
	ADaPT (Prasad et al., 2023)	30.6	15.7	35.1
	THREAD	**71.6**	**22.4**	**91.0**

Results. THREAD outperforms prior methods by at least 20% absolute points with GPT-3.5 and at least 40% points with Llama-3-8b and 30% points CodeLlama-7b (Table 3). Further, Llama-3-8b with THREAD outperforms GPT-3.5 with all prior methods and CodeLlama-7b with THREAD outperforms GPT-3.5 with all prior methods except TDAG.

4.3 WebShop

WebShop is an online shopping environment with over a million real-world products (Yao et al., 2022). The benchmark requires the model to interact with the website to purchase a product based on specifications provided by a user. The evaluation metrics include the success rate, which is the percentage of products that were purchased with full success, and the score, which is the average percentage of desired attributes covered by the purchased items. Following prior work (Shinn et al., 2024; Prasad et al., 2023; Zhou et al., 2023), we evaluate THREAD on a test set of 100 instructions.

Results. Table 4 shows the success rate and score for each method. We again separate the results based on whether the method requires the model to have access to external memory. With GPT-3.5, THREAD outperforms other prompt-only methods by an absolute 4% in success rate and over 10% in score and outperforms RAP (Kagaya et al., 2024) by 1% in success rate (with a similar score). THREAD achieves an absolute 10%, or greater, improvement in success rate with Llama-3-8b and CodeLlama-7b. Llama-3-8b with THREAD achieves a higher success rate than GPT-3.5 with all prior methods, with the exception of RAP.

4.4 DataCommons QA

DataCommons QA is a benchmark consisting of questions that can be answered using data provided by Google DataCommons (Guha, 2019) `https://datacommons.org`. The questions range from comparing statistics in different locations to making predictions regarding future trends. The test

Table 3: TextCraft success rate (%). *As reported in Prasad et al. (2023).

Method	GPT-3.5	Llama-3-8b	Llama-2-7b	CodeLlama-7b
Reflexion* (Shinn et al., 2024)	32.0	-	-	-
ReAct (Yao et al., 2023)	20.5	12.5	8.0	10.5
DecomP (Khot et al., 2023)	68.5	47.0	12.0	38.5
ADaPT (Prasad et al., 2023)	52.5	23.5	12.0	18.0
TDAG (Wang et al., 2024)	73.5	48.5	14.0	31.0
THREAD	**93.5**	**92.0**	**20.0**	**71.0**

Table 4: WebShop success rate (SR; %) and score (%).

Requires ext. mem.	Method	GPT-3.5		Llama-3-8b		Llama-2-7b		CodeLlama-7b	
		SR	**Score**	**SR**	**Score**	**SR**	**Score**	**SR**	**Score**
Yes	Reflexion (Shinn et al., 2024)	38	64.4	32	59.8	8	19.7	17	57.3
	LATS (Zhou et al., 2023)	40	76.0	34	61.5	12	30.1	21	60.7
	RAP* (Kagaya et al., 2024)	48	76.1	-	-	-	-	-	-
	AutoGuide* (Fu et al., 2024)	46	73.4	-	-	-	-	-	-
No	ReAct (Yao et al., 2023)	37	59.5	31	54.1	10	18.5	17	49.2
	DecomP (Khot et al., 2023)	43	58.7	35	56.3	14	29.6	21	57.1
	ADaPT (Prasad et al., 2023)	44	60.0	35	58.2	13	28.7	21	56.4
	TDAG (Wang et al., 2024)	45	64.5	37	63.5	14	29.8	23	58.5
	THREAD	**49**	**76.3**	**47**	**70.4**	**20**	**48.5**	**40**	**68.9**

set includes a total of 140 questions. We implement THREAD with a few-shot prompt and compare its performance to three baselines: Reflexion (Shinn et al., 2024), Natural Language Embedded Programs (NLEP) (Zhang et al., 2023), and NLEP+ReAct, an extension of NLEP that allows the model to evaluate intermediate outputs of its analysis as it answers the question.

Results. Table 5 shows the accuracy of each method on DataCommons QA. THREAD outperforms prior methods by over 10% absolute points with GPT-3.5, Llama-3-8b, and CodeLlama-7b. Llama-3-8b with THREAD outperforms GPT-3.5 with all prior methods.

4.5 MIMIC-III ICU QA

The MIMIC-III ICU QA benchmark consists of patient-focused questions based on clinical time-series data made available by MIMIC-III (Johnson et al., 2016). The benchmark reflects a setting in which a healthcare provider can ask natural language questions about patients in the intensive care unit (ICU) and receive an answer from the language model based on the relevant patient data. The test set includes 160 questions. Similar to above, we implement THREAD with a few-shot prompt and compare its performance relative to Reflexion (Shinn et al., 2024), NLEP, and NLEP+ReAct.

Results. As shown in Table 6, THREAD significantly outperforms prior methods across all models except Llama-2-7b. All methods show the lowest performance with Llama-2-7b, which is consistent across all benchmarks.

5 Related Work

The THREAD framework relates to prior approaches for dynamically adapting work, or computational steps, used during model generation. Previous approaches for dynamically adapting computation based on the given problem require special training (Goyal et al., 2023; Nye et al., 2022) or novel model architectures (Graves, 2016; Banino et al., 2021; Dehghani et al., 2018). THREAD provides a more general and flexible framework that, as we show, can be applied without modifications to the underlying model architecture and without further training. In addition, unlike methods that allow the model to adapt the amount of internal computations (Goyal et al., 2023; Graves, 2016; Banino et al., 2021; Dehghani et al., 2018), the intermediate work used to supplement model generation with THREAD can involve work that ex-

Table 5: DataCommons QA accuracy (%).

Method	GPT-3.5	Llama-3-8b	Llama-2-7b	CodeLlama-7b
Reflexion (Shinn et al., 2024)	37.9	24.3	10.7	20.7
DecomP (Khot et al., 2023)	64.3	57.9	15.7	31.4
NLEP (Zhang et al., 2023)	28.6	21.4	11.4	19.3
NLEP+ReAct	41.4	27.1	14.3	24.3
THREAD	**77.1**	**67.9**	**22.1**	**62.1**

Table 6: MIMIC-III ICU QA accuracy (%).

Method	GPT-3.5	Llama-3-8b	Llama-2-7b	CodeLlama-7b
Reflexion (Shinn et al., 2024)	38.1	19.4	8.8	16.9
DecomP (Khot et al., 2023)	61.9	51.3	13.1	30.6
NLEP (Zhang et al., 2023)	35.6	17.5	8.1	15.6
NLEP+ReAct	43.1	23.8	12.5	21.9
THREAD	**71.3**	**61.9**	**18.8**	**58.1**

tends beyond internal thinking, such as retrieving information or interacting with an external environment. In addition, THREAD allows for greater interpretability, since all of the intermediate work is performed by generating meaningful tokens.

In this paper, we apply THREAD in a setting where it improves the ability of LLMs to decompose a given problem into simpler sub-problems. There has been significant prior work involving the decomposition of problems with neural modular modeling architectures (Andreas et al., 2016; Talmor and Berant, 2018; Min et al., 2019; Jiang and Bansal, 2019; Gupta et al., 2019; Perez et al., 2020; Khot et al., 2021). Later work has used fine-tuning or in-context learning with modern LLMs for decomposition with multi-step tasks (Khot et al., 2023; Wang et al., 2023), mathematical reasoning (Gao et al., 2023; Lee and Kim, 2023), and program synthesis (Murali et al., 2018; Nye et al., 2019; Zheng et al., 2023). To handle complex tasks, a more recent approach, called ADaPT, has improved upon these methods by using a planner to further decompose a task when a failure is encountered (Prasad et al., 2023), which was further extended to a multi-agent setting with TDAG (Wang et al., 2024). Unlike prior methods, such as Decomposed Prompting (Khot et al., 2023), ADaPT and TDAG enable the model to adapt a plan upon task failure. In contrast to these approaches, THREAD has the advantage of allowing the model to adapt its decision-making in real time as it receives feedback from each step. With THREAD, each step of the plan is specified only after receiving feedback from the previous step. In addition, THREAD enables the model to flexibly specify and, if needed, re-specify sub-tasks when a sub-task fails or when unexpected feedback is returned. Finally, THREAD, which uses the same prompting for every thread at every step of the problem, provides a more unified framework than ADaPT and TDAG, which involve separate prompting and few-shot examples for the "planner" and "executor" modules. With these methods, the planner module outlines a full plan and the executor follows all steps of the plan. If the executor fails, the planner is tasked with further decomposing the failed step into more sub-steps. Instead of outlining a full plan and attempting all steps, THREAD involves specifying one step, allocating this to a child thread, and receiving feedback from the child thread before proceeding.

6 Conclusion

We introduce THREAD, a general framework in which model generation is treated as a thread of execution that can dynamically offload work, such as thinking or retrieving information, by spawning new threads. THREAD enables a model to adapt, through recursive spawning, the amount of intermediate work it uses to produce different parts of its token sequence. We apply this framework in the settings of agent tasks and question answering. We show that THREAD, implemented using a few-shot learning approach, achieves state-of-the-art performance on diverse benchmarks.

Acknowledgements

This work was supported by Quanta Computer, Inc. We also thank the Google DataCommons team for their help with developing the DataCommons QA benchmark.

Limitations

The implementation of THREAD in this paper does not involve any explicit mechanisms for error handling and instead relies on the inherent reasoning ability of LLMs to respond to unexpected feedback from child threads or from the environment. An advantage of THREAD is that individual threads can fail without directly affecting other threads. However, the context necessary to self-correct may be lost if error reporting is not handled appropriately. More work is needed to develop robust error detection and recovery mechanisms to ensure that the information necessary for self-correction is preserved and utilized effectively. In addition, our implementation of THREAD involves limited communication between the parent and child threads, where the context for the child thread is based on the last line of the parent's token sequence. This can result in the child missing information that could help with its work, leading to less efficient or effective generation. Overall, more work is need to improve the propagation of information, as defined by the functions ϕ and ψ, between parent and child threads.

Ethical Statement

In this work, we show how THREAD can be applied to improve LLM agent task completion and question answering. LLMs are being increasingly deployed to autonomously interact with external environments and humans. By improving this capacity, our work has the potential for amplifying risk associated with automated decision-making or facilitate harmful use of LLMs. Addressing these risks requires careful consideration of ethical guidelines, robustness checks, and transparency in algorithm deployment to ensure that advancements in LLMs contribute positively to societal welfare.

References

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 39–48.

Andrea Banino, Jan Balaguer, and Charles Blundell. 2021. Pondernet: Learning to ponder. In *8th ICML Workshop on Automated Machine Learning (AutoML)*.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. 2018. Universal transformers. In *International Conference on Learning Representations*.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. 2023. Faith and fate: Limits of transformers on compositionality. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. 2024. Autoguide: Automated generation and selection of state-aware guidelines for large language model agents. *arXiv preprint arXiv:2403.08978*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. 2023. Think before you speak: Training language models with pause tokens. In *The Twelfth International Conference on Learning Representations*.

Alex Graves. 2016. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*.

Ramanathan Guha. 2019. Datacommons. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1–1. IEEE.

Nitish Gupta, Kevin Lin, Dan Roth, Sameer Singh, and Matt Gardner. 2019. Neural module networks for reasoning over text. In *International Conference on Learning Representations*.

Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards reasoning in large language models: A survey. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1049–1065.

Yichen Jiang and Mohit Bansal. 2019. Self-assembling modular networks for interpretable multi-hop reasoning. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4474–4484.

Alistair EW Johnson, Tom J Pollard, Lu Shen, Li-wei H Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi,

and Roger G Mark. 2016. Mimic-iii, a freely accessible critical care database. *Scientific data*, 3(1):1–9.

Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. 2024. Rap: Retrieval-augmented planning with contextual memory for multimodal llm agents. *arXiv preprint arXiv:2402.03610*.

Tushar Khot, Daniel Khashabi, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2021. Text modular networks: Learning to decompose tasks in the language of existing models. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1264–1279.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed prompting: A modular approach for solving complex tasks. In *The Eleventh International Conference on Learning Representations*.

Soochan Lee and Gunhee Kim. 2023. Recursion of thought: A divide-and-conquer approach to multi-context reasoning with language models. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 623–658.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2019. Multi-hop reading comprehension through question decomposition and rescoring. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6097–6109.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2022. Show your work: Scratchpads for intermediate computation with language models. In *Deep Learning for Code Workshop*.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870. PMLR.

Ethan Perez, Patrick Lewis, Wen-tau Yih, Kyunghyun Cho, and Douwe Kiela. 2020. Unsupervised question decomposition for question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. 2023. Adapt: As-needed decomposition and planning with language models. *arXiv preprint arXiv:2311.05772*.

Guanghui Qin, Yukun Feng, and Benjamin Van Durme. 2023. The nlp task effectiveness of long-range transformers. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 3774–3790.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. Alfworld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2023a. Adaplanner: Adaptive planning from feedback with language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Simeng Sun, Yang Liu, Shuohang Wang, Chenguang Zhu, and Mohit Iyyer. 2023b. Pearl: Prompting large language models to plan and execute actions over long documents. *arXiv preprint arXiv:2305.14564*.

Alon Talmor and Jonathan Berant. 2018. The web as a knowledge-base for answering complex questions. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 641–651.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634.

Yaoxiang Wang, Zhiyong Wu, Junfeng Yao, and Jinsong Su. 2024. Tdag: A multi-agent framework based on dynamic task decomposition and agent generation. *arXiv preprint arXiv:2402.10178*.

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*, 35:24824–24837.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

Tianhua Zhang, Jiaxin Ge, Hongyin Luo, Yung-Sung Chuang, Mingye Gao, Yuan Gong, Xixin Wu, Yoon Kim, Helen Meng, and James Glass. 2023. Natural language embedded programs for hybrid language symbolic reasoning. *arXiv preprint arXiv:2309.10814*.

Wenqing Zheng, SP Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. 2023. Outline, then details: Syntactically guided coarse-to-fine code generation. In *International Conference on Machine Learning*, pages 42403–42419. PMLR.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

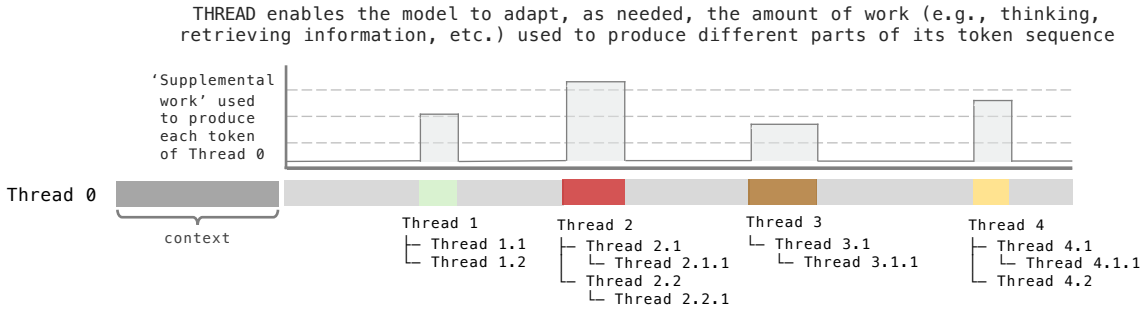


Figure 3: THREAD allows model to adapt amount of supplemental work used to produce tokens.

A Illustration of Intermediate Work used to Produce Tokens

Figure 3 illustrates how THREAD enables the model to dynamically adapt the amount of work or computational steps used to produce different parts of its token sequence. The illustration is based on the Thread 0 from Figure 1. The "supplemental work" for a token reflects the amount of additional tokens, generated by spawned threads, used to produce each token within Thread 0. The work associated with each part of the token sequence is represented by a thread tree organized based on connections between parent and child threads. These trees reflect how threads interact to form output tokens and the computational work associated with each component of the overall model generation.

B Implementing THREAD for Agent Tasks and Question Answering

We release all code and data at `https://github.com/philipmit/thread`. As described in section 3, we implement THREAD with join synchronization for the problems evaluated in this paper. We implement a spawning mechanism using a special stop token, ω_{listen}. A thread ends when it generates the end token, ω_{end}. We use `=>` as ω_{listen} and `END` as ω_{end}.

We implement THREAD in each setting using a fixed few-shot prompt showing examples of successful spawning and problem solving at different thread depths. All threads are provided the same few-shot prompt for every step of the problem completion. This prompt, q, is prepended to the context, $c = q + c$, for each thread, forming the full context from which the thread generates.

As shown in Algorithm 2 below, threads can get feedback from the environment the same way they get feedback from a child thread. They generate tokens that represent the action and then produce the token ω_{listen} to listen for feedback from the environment. As shown in Algorithm 2, the action, a, is executed in the environment, E, and the output, $o = E(a)$, is appended to the thread's token sequence before it continues generation. The function ξ parses the action from the token sequence. As with prior methods such as ReAct (Yao et al., 2023), actions are specified following the '>' token.

Algorithm 2 THREAD with join synchronization for tasks involving actions in environment

function THREAD(c, Y)
  while `True` **do**
    $Y = Y + G(c + Y)$
    if Y performs an action **then**
      $a = \xi(Y)$
      $o = E(a)$
      $Y = Y + o$
    else if Y spawns a child thread **then**
      $Y = Y + \psi(\text{THREAD}(\phi(Y), [\,]))$
    else if Y ends the thread **then**
      return Y

To ensure consistency across benchmarks and with prior work, we use `Meta-Llama-3-8B` for Llama-3-8b, `Llama-2-7b-hf` for Llama-2-7b, and `CodeLlama-7b-hf` for CodeLlama-7b available on Huggingface (Wolf et al., 2019) and use `gpt-4-0613` for GPT-4 and `gpt-3.5-turbo-instruct` for GPT-3.5 from the OpenAI API, with the temperature set to 0 for all experiments.

C Additional Experiments with GPT-3.5

Table 7 shows the mean and standard error of the performance of THREAD with GPT-3.5 across 5 runs on each benchmark. Overall, the results are consistent across multiple runs. Table 7 also shows the average and max thread depths for each task.

Table 7: Mean and standard error of THREAD performance and thread depths with GPT-3.5.

Benchmark	Mean (Std. Err.)	Avg. thread depth	Max thread depth
DataCommons QA	76.7 (.17)	4.1	6
MIMIC-III ICU QA	71.4 (.36)	4.3	6
ALFWorld	95.7 (.28)	3.7	7
TextCraft	93.7 (.37)	5.8	10
WebShop	48.6 (.40)	3.6	7

Table 8: ALFWorld task-specific success rates (%) using task-general prompting with THREAD.

Model	**All**	**Pick**	**Clean**	**Heat**	**Cool**	**Look**	**Pick2**
GPT-3.5	95.5	100	96.8	82.6	95.2	100	100
Llama-3-8b	49.3	58.3	38.7	56.5	57.1	72.2	11.8
CodeLlama-7b	61.9	41.7	87.1	65.2	61.9	38.9	64.7

D Task-general Prompting for ALFWorld

Table 8 shows the results when testing THREAD with task-general prompting in ALFWorld. To implement the task-general prompt, we split the prompt into one set of examples for the main thread and a second set of examples for all other threads. The same sets of examples used for all tasks. GPT-3.5 achieves the same combined success rate with the task-general prompting as it does with the task-specific prompting. Further, while the performance of Llama-3-8b and CodeLlama-7b degrades with task-general prompting, it remains a significant improvement over task-specific prompting with prior methods (Table 2).

E Error and Ablation Analysis

E.1 Error Types

To identify what types of errors THREAD reduces, we classify the errors that are responsible for each method's failures. Descriptions and examples of each error type are provided in Tables 9 and 10 for the agent tasks and Tables 11 and 12 for the QA tasks. We classified these errors through hand review of each failure case, where the first error type to occur within the failure case was identified as the error type responsible for the failure. Figures 4a, 4b, 5, and 6a show the error counts across different methods and tasks. We focus this analysis on the prompt-only methods to provide a more clear evaluation of how the novel aspects of THREAD change performance relative to other methods that leverage few-shot learning.

E.2 Ablations

Figures 4c, 4d, and 6b show how the error counts change when applying the following three modifications to THREAD:

1. Removing the variables used by parent and child threads to manipulate and organize shared information.
2. Replacing ϕ with a function that returns the parent thread's full token sequence.
3. Using a separate prompt for Thread 0 (which performs the initial task decomposition) instead of using the same few-shot prompt for all threads.

E.3 Agent Tasks

Task decomposition and flexible sub-task specification reduce failures caused by inadequate plans. In Figures 4a and 4b, we see that the methods that involve task decomposition (i.e., Decomposed prompting, ADaPT, and THREAD) show fewer failures that are caused by inadequate planning. This is likely because there is a dedicated line of reasoning for generating the plan, instead of it being part of the single line of model generation that is responsible for both defining the plan and carrying out the steps of the plan, like in ReAct.

We see that THREAD, which allows for flexible sub-task specification, further reduces failures caused by inadequate planning. This is due to the fact that, when a child thread fails, the parent thread can accommodate the child thread's feedback, adapt the sub-task specification, and spawn a new thread. Prior methods do not allow for this flexible sub-task re-specification, as they either require

the model to select from a fixed set of pre-defined sub-task handlers (e.g., Decomposed Prompting) or do not accommodate sub-task feedback when handling failures (e.g, ADaPT and TDAG). Figure 7 shows an example where a child thread, which is tasked with checking a cabinet for a plate, returns feedback saying the cabinet is closed. This is a common problem, where a child thread (or a sub-task handler/executor in Decomposed Prompting or ADaPT) return some unpredictable intermediate result instead of fully completing its task (which, in this case, involves opening the cabinet and checking for a plate). To address this with THREAD, the parent thread spawns a new child thread with a modified sub-task specification (which includes opening the cabinet and checking if there is a plate) in order to prevent the new child thread from repeating the mistake made by the previous child. Prior methods do not allow for these dynamic, nuanced adjustments to the plan to accommodate the full range of possible intermediate feedback involved in complex tasks.

Shortening action sequences reduces failures caused by misinterpreting environment feedback. We see that errors in interpreting environment feedback are lowest with Decomposed Prompting and THREAD. One potential explanation for this is that, by decomposing the task into more sub-components than other methods, Decomposed Prompting and THREAD require each line of model generation to execute shorter sequences of consecutive actions. As a result, each line of model generation needs to manage shorter action-observation sequences and, therefore, is able to more accurately interpret and keep track of the information returned from the environment.

Figure 5 shows the number of times GPT-3.5 misinterprets environment feedback when executing action sequences of different lengths when using the different methods. By significantly reducing the number of cases where the model has to perform long action sequences, Decomposed Prompting and THREAD reduce the number of interpretation errors.

E.4 QA Tasks

Task decomposition reduces failures that are caused by performing the wrong analysis. In Figure 6a, we see that Decomposed Prompting and THREAD show fewer failures caused by the model performing the wrong analysis for the given question. This is similar to what we see with the agent tasks where problem decomposition improves the planning. By separating the line of model generation that defines the plan (defining the analysis type) from the line of model generation that is responsible for executing the plan (completing the analysis), the model can more effectively accomplish both tasks.

Flexible sub-task specification reduces failures that are caused by runtime errors. We see that THREAD shows the fewest failures that are caused by runtime errors. With THREAD, when an error occurs, the error is returned to the parent thread, which can adapt its plan by modifying the sub-task specification and spawning a new thread (as discussed above). In addition, based on the ablation analysis (Figure 6b), we see that the variables used in THREAD may also reduce failures due to runtime errors, especially with smaller models such as Llama-3-8b.

F DataCommons QA and MIMIC-III ICU QA Benchmarks

We built the DataCommons QA benchmark utilizing data provided by Google Data Commons (Guha, 2019) with a focus on U.S. data. Data Commons contains publicly available data, aggregated from sources such as the Center for Disease Control and Prevention and the World Health Organization.

Generation of the benchmark started with the production of questions which fit the following criteria: 1) answerable by information provided by Google Data Commons, 2) broad enough to encompass information sampled between different locations (e.g., two counties), 3) lacking in subjectivity for the creation of a verifiable answer. The benchmark consists of question templates, which generate the questions of the benchmark, and ground-truth programs, which generate the answers to the questions of the benchmark. This format allows for the creation of a large quantity of diverse questions. Each question template consists of a sentence containing placeholders such as `{variable_text}` which, upon random sampling of a variable, are altered accordingly, facilitating the automated production of a variety of questions based on one question template. An example of a question template is as follows: 'Was `{variable_text}` in `{entity1}` increasing or decreasing from `{time1}` to `{time2}`?'. Time series data was sampled at the city, state, county, and country level for the United States. Data retrieval from Google Data Commons

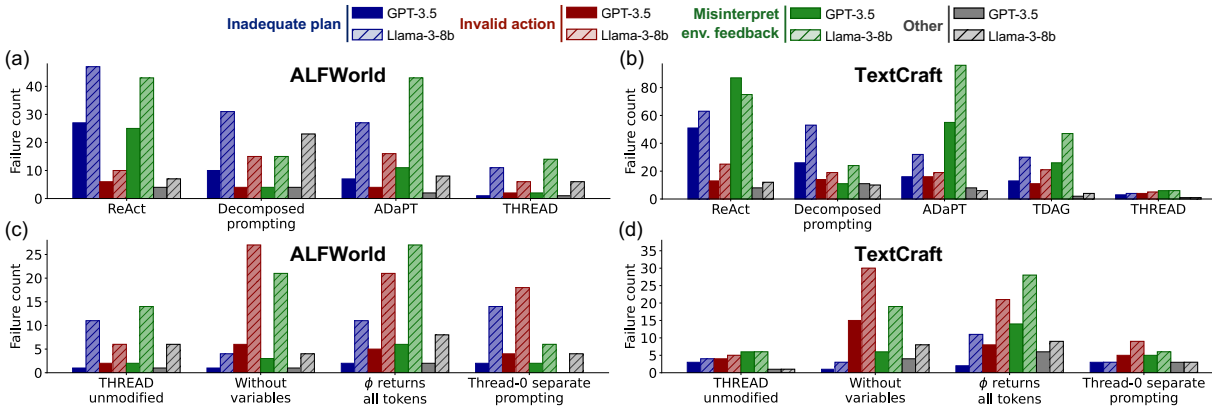


Figure 4: Failure counts in ALFWorld and TextCraft for different methods (a and b) and for modified versions of THREAD (c and d).

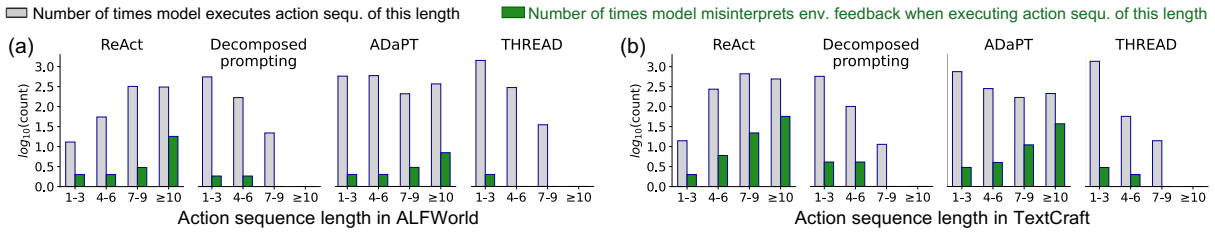


Figure 5: Number of times GPT-3.5 misinterprets environment feedback when executing action sequences in ALFWorld (a) and TextCraft (b). Values are shown as $\log_{10}$(count) to allow the total counts (shown in gray) to fit in the same figure as the error counts (shown in green).

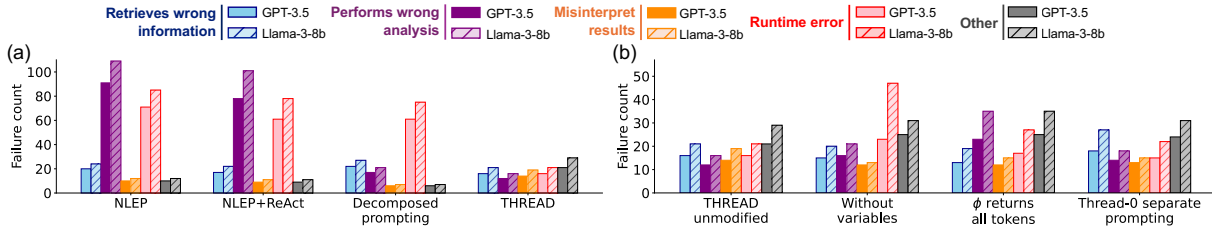


Figure 6: Failure counts, combined for DataCommons QA and MIMIC-III ICU QA, for different methods (a) and modified versions of THREAD (b).

Table 9: Description of error types for agent tasks.

Error type	Description
Inadequate plan	The plan does not include steps sufficient to complete the task (including steps to recover from unexpected intermediate results).
Invalid action	The model attempts to perform an invalid action in the environment.
Misinterprets environment feedback	The model incorrectly interprets information returned from the environment.

Table 10: Examples of error types for agent tasks.

Error type	Examples
Inadequate plan	**ALFWorld:** The plan does not include cleaning the plate when the model is tasked with putting a clean plate on the table. The plan does not involve opening the cabinet to check for a plate when an unexpected sub-task result is returned (e.g., "The cabinet is closed"). **TextCraft:** The plan involves crafting an object before acquiring one of the precursors. The plan does not finish checking if a material (e.g., bamboo) can be retrieved from the environment when an unexpected result is returned (e.g., "Could not find valid recipe for bamboo.").
Invalid action	**ALFWorld:** The model incorrectly specifies the action for examining an object with the desklamp. **TextCraft:** The model incorrectly states a crafting recipe.
Misinterprets environment feedback	**ALFWorld:** The model continues looking for a plate after the environment indicates "You see a plate 1". **TextCraft:** The model proceeds as though it has successfully fetched bamboo despite the environment returning "Could not find bamboo".
Other	**ALFWorld:** The model reaches the maximum number of actions allowed in the ALFWorld environment. **TextCraft:** The model runs out of context.

Table 11: Description of error types for QA tasks.

Error type	Description
Retrieves wrong information	The model does not retrieve the information needed to answer the question.
Performs wrong analysis	The model retrieves the correct information, but does not perform the analysis needed to answer the question.
Misinterprets results of analysis	The model performs the correct analysis, but misinterprets the results.
Runtime error	Error occurs during the execution of the model's code.

```
TASK:
You are in the middle of a room. . .  you see a cabinet 1, a countertop 1, ...
Your task is to: clean some plate and put it in sidetable.
```

```
########
ADaPT
########

### Planner:
I need to find plate. The plate is likely to be in cabinet, countertop, or diningtable…
Step 1: Find and take plate from cabinet
If plate not found so far, I will next look in the countertop.
Step 2: Find and take plate from countertop
If plate not found so far, I will next look in the diningtable.
Step 3: Find and take plate from diningtable
Execution Order: (Step 1 OR Step 2 OR Step 3)

### Executor 1:
Your task is to: Take plate from cabinet 1
> go to cabinet 1
The cabinet 1 is closed.
> think: The cabinet is closed. Task failed!

### Executor 2 (moving to Executor 2 because Executor 1 failed):
Your task is to: Take plate from countertop 1
> go to countertop 1
On the countertop 1, you see a cup 1, . . .
> think: There is no plate. Task failed!

### Executor 3 (moving to Executor 3 because Executor 2 failed):
Your task is to: Take plate from diningtable 1
> go to diningtable 1
On the diningtable 1, you see a apple 1, . . .
> think: There is no plate. Task failed!
```

```
########
THREAD
########

### Thread 0:
obj = 'plate'
likely_loc_IDs_for_obj = ['cabinet 1', 'countertop 1', . . . ]
. . .
First, I need to find the {obj}. =>

### Thread 1:
My sub-task is to find the {obj}.
. . .
loc_i = likely_loc_IDs_for_obj[0]
I need to check if there is the {obj} in the {loc_i}. =>

### Thread 1.1:
My sub-task is to check if there is the {obj} in the {loc_i}.
> go to {loc_i} =>The cabinet 1 is closed<=
. . .
print('The {loc_i} is closed.')
END

### Thread 1 (continue Thread 1 after Thread 1.1 has ended):
. . .
I need to check if there is the {obj} in the {loc_i}. =>The {loc_i} is closed.<=
I need to open the {loc_i} and check if there is the {obj} in the {loc_i}. =>

### Thread 1.2:
My sub-task is to open the {loc_i} and check if there is the {obj} in the {loc_i}.
> open {loc_i} =>You see a cup 1, a plate 1, . . .<=
. . .
obj_ID = 'plate 1'
print('You have opened the {loc_i} and found the {obj_ID}')
END

### Thread 1 (continue Thread 1 after Thread 1.2 has ended):
. . .
I need to open the {loc_i} and check if there is the {obj} in the {loc_i}. =>You have opened the {loc_i} and found the {obj_ID}<=
. . .
print('You have found the {obj_ID}')
END
```

Figure 7: Example of task in ALFWorld where ADaPT fails to productively respond to the feedback from a sub-task, while THREAD successfully accommodates the feedback to re-specify a new sub-task.

Table 12: Examples of error types for QA tasks.

Error type	Examples
Retrieves wrong information	The model retrieves data on smoking prevalence of all people when asked specifically about the female population.
Performs wrong analysis	The model compares values from 2014 to 2021 when asked to compare values from 2018 to 2021.
Misinterprets results of analysis	The model indicates a variable is increasing despite its analysis showing that it is decreasing.
Runtime error	The model tries to concatenate an integer to a string.
Other	The model runs out of context.

was conducted through the REST API provided by the website.

A "ground-truth program" was created for each question template to generate a verifiably correct answer for each question within the benchmark. These programs varied in length from 100-200 lines of code and analyzed data acquired from the Google Data Commons API in accordance with the constraints tailored by the question it was answering. For example, if a question template included a constraint about the years of data to be sampled, the ground-truth program was constructed to generate a year range for the answer. Code within the program would subsequently be created to filter and analyze the data acquired from Google Data Commons accordingly by applying whichever mathematical concept was required for the answer (e.g., correlation, linear regression, median, mean). Consequently, ground-truth programs formed the backbone of the benchmark through the creation of answers to the diverse questions which comprised the question template set. Figure 8 shows an example with the question "From 2015 to 2021, was the rate of asthma increasing faster in Boston or LA?".

The MIMIC-III ICU QA was created using the same approach as described above. However, questions were instead based on clinical time-series data provided by MIMIC-III (Johnson et al., 2016). We outline all steps, along with the code, to reproduce the benchmarks at https://github.com/philipmit/thread. We release the full dataset for DataCommons QA. Due to restrictions with MIMIC-III data access, we cannot directly release the dataset for MIMIC-III ICU QA. However, upon gaining access to MIMIC-III following the instructions outlined at https://physionet.org/content/mimiciii/1.4/, you can reproduce the full dataset using the code that we release.

From 2015 to 2021, was the rate of asthma increasing faster in Boston or LA?

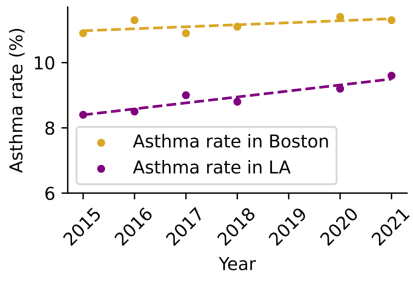
A table with the asthma rate from 2015 to 2019 in Boston and LA is shown below:
=>{table_dir}<=

Year	Asthma rate Boston	Asthma rate LA
2015	10.9	8.4
2016	11.3	8.5
2017	10.9	9.0
2018	11.1	8.8
2020	11.4	9.2
2021	11.3	9.6

Based on the data in {table_dir}, the slope of the asthma rate in Boston is =>0.06<= and in LA is =>0.18<=.

Therefore, we see that the rate of asthma increased faster in LA than Boston.

A figure representing the data in {table_dir} is shown below:
=>



<=

Figure 8: Question and example response with the DataCommons QA benchmark.

G Prompts

Below, we provide examples of the prompt used for each benchmark. The given task or question is highlighted in yellow. The text corresponding to the spawning of child threads is highlighted in blue, pink, orange, and green. The full prompts for each task can be seen at https://github.com/philipmit/thread.

G.1 DataCommons QA

Question: From 2018 to 2021 was the percentage of people with diabetes increasing faster in Willows or Villa Ridge?
Here is a table showing the percentage of people with diabetes during 2018 to 2021 for Willows and Villa Ridge:
=>{table_dir}<=
Based on the data in {table_dir} from 2018 to 2021 the slope of the percentage of people with diabetes in Willows was =>0.3357<= and in Villa Ridge was =>0.1642<=
Therefore, the percentage of people with diabetes was increasing faster in Willows.
Final Answer: Willows
#END#

Here is a table showing the percentage of people with diabetes during 2018 to 2021 for Willows and Villa Ridge: =>
location1 = 'Willows'
location2 = 'Villa Ridge'
timeframe = '2018 to 2021'
variable = 'percentage of people with diabetes'
# The {variable} during {timeframe} for location number 1, {location1}, is =>{data_location1_in_timeframe}<=
# The {variable} during {timeframe} for location number 2, {location2}, is =>{data_location2_in_timeframe}<=
data_location1_in_timeframe = [data_location1_in_timeframe] if not isinstance(data_location1_in_timeframe, list) else data_location1_in_timeframe
data_location2_in_timeframe = [data_location2_in_timeframe] if not isinstance(data_location2_in_timeframe, list) else data_location2_in_timeframe
timeframe_dates = np.unique([x['date'] for x in data_location1_in_timeframe] + [x['date'] for x in data_location2_in_timeframe])
data_location1_in_timeframe_values = ...
data_location2_in_timeframe_values = ...
...
...
table = pd.DataFrame('date': timeframe_dates, location1: data_location1_in_timeframe_values, location2: data_location2_in_timeframe_values)
table_dir = DATA_DIR
table.to_csv(table_dir, sep='', index=False)
print('{table_dir}')
#END#

# The percentage of people with diabetes during 2018 to 2021 for location number 1, Willows, is =>
location1 = 'Willows'
timeframe = '2018 to 2021'
time1 = '2018'
time2 = '2021'
variable = 'percentage of people with diabetes'
query = f'{variable} in {location1}'
response = requests.post(URL_EXPLORE + '?q=' + query, headers=HEADERS, json=JSON)
response = json.loads(response.text)
> print(response['variables']) =>['dc/topic/Diabetes', 'Percent_Person_WithDiabetes', 'Percent_Person_20OrMoreYears_WithDiabetes', 'dc/topic/PopulationWithDiabetesByAge', 'Count_Person_20To79Years_Diabetes_AsFractionOf_Count_Person_20To79Years', 'WHO/SDG_SH_DTH_RNCOM_DiabetesMellitus', 'dc/topic/DiabetesFemalePopulationByAge', 'dc/nh3s4skee5483']<=
variable_ID = 'Percent_Person_WithDiabetes'
> print(response['entities']) =>['geoId/0685684']<=
location_ID = 'geoId/0685684'
# I will now use the location_ID and variable_ID to get the data.
data_location1_all = collect_data_commons1(location_ID, variable_ID)
data_location1_in_timeframe = [x for x in data_location1_all if (int(x['date'])>=int(time1) and int(x['date'])<=int(time2))]
> print(data_location1_in_timeframe) =>['date': '2018', 'value': 10.1, 'date': '2020', 'value': 10.8, 'date': '2021', 'value': 11.1]<=
print('You have retrieved {data_location1_in_timeframe}.')
#END#

Based on the data in /path/to/data_table.txt from 2018 to 2021 the slope of the percentage of people with diabetes in Willows was =>
table_dir = '/path/to/data_table.txt'
variable = 'percentage of people with diabetes'
location = 'Willows'
table = pd.read_csv(table_dir, sep='^')
# > print(table) =>
# date Willows Villa Ridge
# 0 2018 10.1 8.2
# 1 2020 10.8 8.5
# 2 2021 11.1 8.7
# <=
location_col_name = 'Willows'
X = table['date']
y = table[location_col_name]
slope, intercept, r_value, p_value, std_err = stats.linregress(X, y)
print(slope)
#END#

G.2 MIMIC-III ICU QA

Question: Was the average systolic blood pressure of patient X from hour 10 to 20 of their ICU stay higher or lower than the average among all patients who expired in the hospital?
Here is a table showing the systolic blood pressure of patient X from hour 10 to 20 of their ICU stay: =>{table_dir}<=
Based on the data in {table_dir} the average systolic blood pressure of patient X was =>138.18<=
The average systolic blood pressure among all patients who expired in the hospital was =>120.81<=
Therefore, the average systolic blood pressure of patient X from hour 10 to 20 was higher than the average among all patients who expired in the hospital.
Final Answer: higher
#END#

Here is a table showing the systolic blood pressure of patient X from hour 10 to 20 of their ICU stay: =>
patient_ID = 'X'
timeframe = 'hour 10 to 20'
variable = 'systolic blood pressure'
# The {variable} during {timeframe} for patient {patient_ID} was =>{data_patient_in_timeframe}<=
...
...
table = pd.DataFrame('time': timeframe_dates, variable: data_patient_in_timeframe_values)
table_dir = DATA_DIR
table.to_csv(table_dir, sep='^', index=False)
print('{table_dir}')
#END#

# The systolic blood pressure during hour 10 to 20 for patient X was =>
patient_ID = 'X'
timeframe = 'hour 10 to 20'
time1 = '10'
time2 = '20'
variable = 'systolic blood pressure'
...
...
print('You have retrieved {data_patient_in_timeframe}.')
#END#

Based on the data in /path/to/data_table.txt the average systolic blood pressure of patient X was =>
table_dir = '/path/to/data_table.txt'
variable = 'systolic blood pressure'
table = pd.read_csv(table_dir, sep='^')
# > print(table) =>
...
...
variable_col_name = 'Systolic blood pressure'
y = table[variable_col_name]
y_mean=np.mean(y)
print(y_mean)

#END#

The average systolic blood pressure among all patients who expired in the hospital was =>
variable = ‘systolic blood pressure’
table = pd.read_csv(PATIENT_DATA_DIR, sep=‘‘)
# > print(table) =>
...
...
filter_col_name = ‘MORTALITY_INHOSPITAL’
variable_col_name = ‘Systolic blood pressure’
target_patient_group = table[table[variable_col_name]==1]
y = target_patient_group[variable_col_name]
y_mean=np.mean(y)
print(y_mean)
#END#

G.3 ALFWorld

You are in the middle of a room. Looking quickly around you, you see a cabinet 4, a cabinet 3, a cabinet 2, a cabinet 1, a coffeemachine 1, a countertop 1, a diningtable 3, a diningtable 2, a diningtable 1, a drawer 1, a fridge 1, a garbagecan 1, a microwave 1, a sidetable 1, a sinkbasin 1, a stoveburner 4, a stoveburner 3, a stoveburner 2, a stoveburner 1, and a toaster 1.
Your task is to: clean some apple and put it in sidetable.
target_location_ID = ‘sidetable 1’
cleaning_location_ID = ‘sinkbasin 1’
likely_location_IDs_for_obj = [‘fridge 1’, ‘countertop 1’, ‘cabinet 4’, ‘cabinet 3’, ‘cabinet 2’, ‘cabinet 1’, ‘diningtable 3’, ‘diningtable 2’, ‘diningtable 1’, ‘sidetable 1’, ‘sinkbasin 1’, ‘garbagecan 1’, ‘drawer 1’, ‘stoveburner 4’, ‘stoveburner 3’, ‘stoveburner 2’, ‘stoveburner 1’]
obj = ‘apple’
I need to perform the following steps:
First, I need to find and take the {obj}. =>You have found and taken the {obj_ID}.<=
Next, I need to clean the {obj_ID} with the {cleaning_location_ID}. =>You have cleaned the {obj_ID} with the {cleaning_location_ID}.<=
Finally, I need to put the {obj_ID} in the {target_location_ID}. =>You have put the {obj_ID} in the {target_location_ID}.<=
#END#

I need to find and take the apple. =>
obj = ‘apple’
My sub-task is to find and take the {obj}. I will check the locations in {likely_location_IDs_for_obj} one by one.
location_i = likely_location_IDs_for_obj[0]
I need to check if there is the {obj} in the {location_i}. =>There is no {obj} in the {location_i}.<=
location_i = likely_location_IDs_for_obj[1]
...
...
I need to check if there is the {obj} in the {location_i}. =>There is no {obj} in the {location_i}.<=
location_i = likely_location_IDs_for_obj[2]
I need to check if there is the {obj} in the {location_i}. =>There is no {obj} in the {location_i}.<=
location_i = likely_location_IDs_for_obj[3]
I need to check if there is the {obj} in the {location_i}. =>There is no {obj} in the {location_i}.<=
location_i = likely_location_IDs_for_obj[4]
I need to check if there is the {obj} in the {location_i}. =>There is no {obj} in the {location_i}.<=
location_i = likely_location_IDs_for_obj[5]
I need to check if there is the {obj} in the {location_i}. =>There is no {obj} in the {location_i}.<=
location_i = likely_location_IDs_for_obj[10]
I need to check if there is the {obj} in the {location_i}. =>You have found and taken the {obj_ID} from the {location_i}.<=
print(‘You have found and taken the {obj_ID}.’)
#END#

I need to check if there is the apple in the fridge 1. =>
location_i = ‘fridge 1’
obj = ‘apple’
My sub-task is to check if there is the {obj} in the {location_i}.
> go to {location_i} =>The {location_i} is closed.<=
> open {location_i} =>You open the {location_i}. The {location_i} is open. In it, you see a lettuce 2, a mug 2, a potato 2, and a tomato 1.<=

obj_IDs_found = [‘lettuce 2’, ‘mug 2’, ‘potato 2’, ‘tomato 1’]
obj_number = ‘There is no obj_number’
obj_ID = ‘There is no obj_ID’
print(‘There is no {obj} in the {location_i}.’)
#END#

I need to clean the apple 3 with the sinkbasin 1. =>
cleaning_location_ID = ‘sinkbasin 1’
obj_ID = ‘apple 3’
My sub-task is to clean the {obj_ID} with the {cleaning_location_ID}.
> go to {cleaning_location_ID} =>On the {cleaning_location_ID}, you see nothing.<=
> clean {obj_ID} with {cleaning_location_ID} =>You clean the {obj_ID} using the {cleaning_location_ID}.<=
print(‘You have cleaned the {obj_ID} with the {cleaning_location_ID}.’)
#END#

I need to put the apple 3 in the sidetable 1. =>
target_location_ID = ‘sidetable 1’
obj_ID = ‘apple 3’
My sub-task is to put the {obj_ID} in the {target_location_ID}.
> go to {target_location_ID} =>On the {target_location_ID}, you see a cup 1, a lettuce 1, a peppershaker 3, a potato 1, and a saltshaker 1.<=
> put {obj_ID} in/on {target_location_ID} =>You put the {obj_ID} in/on the {target_location_ID}.<=
print(‘You have put the {obj_ID} in the {target_location_ID}.’)
#END#

G.4 WebShop

Instruction: i would like a 3 ounce bottle of bright citrus deodorant for sensitive skin, and price lower than 50.00 dollars
obj_attributes = [‘3 ounce bottle’, ‘Bright citrus’, ‘For sensitive skin’]
max_price = 50.00
obj = ‘deodorant’
I need to perform the following steps:
First, I need to retrieve search results for {obj} that are less than {max_price} dollars with the attributes: {obj_attributes}. =>You have retrieved {results_under_max_price}.<=
Next, I need to identify the item in {results_under_max_price} that matches the most attributes: {obj_attributes}. =>You have identified {item_to_purchase}.<=
Finally, I need to purchase {item_to_purchase} with {obj_attributes}. =>You have purchased {item_to_purchase}.<=
#END#

I need to retrieve search results for deodorant that are less than 50.00 dollars with the attributes: [‘3 ounce bottle’, ‘Bright citrus’, ‘For sensitive skin’]. =>
obj = ‘deodorant’
max_price = 50.00
obj_attribute1 = ‘3 ounce bottle’
obj_attribute2 = ‘Bright citrus’
obj_attribute3 = ‘For sensitive skin’
> search[{obj}; {obj_attribute1}; {obj_attribute2}; {obj_attribute3}] =>
[Back to Search]
Page 1 (Total results: 50)
[Next >]
[B08KBVJ4XN]
Barrel and Oak - Aluminum-Free Deodorant, Deodorant for Men, Essential Oil-Based Scent, 24-Hour Odor Protection, Cedar & Patchouli Blend, Gentle on Sensitive Skin (Mountain Sage, 1 oz, 2-Pack)
$15.95
[B078GWRC1J]
Bright Citrus Deodorant by Earth Mama | Pregnancy and Breastfeeding, Contains Organic Calendula 3-Ounce
$10.99
[B078GTKVXY]
Ginger Fresh Deodorant by Earth Mama | Pregnancy and Breastfeeding, Contains Organic Calendula 3-Ounce
$60.99
<=
I will now filter for results that are less than {max_price}.
results = [‘B08KBVJ4XN’, ‘B078GWRC1J’, ‘B078GTKVXY’]
results_prices = [15.95, 10.99, 60.99]

results_under_max_price = [result for result, price in zip(results, results_prices) if price < max_price]
print('You have retrieved {results_under_max_price}.')
#END#

I need to identify the item in ['B08KBVJ4XN', 'B078GWRC1J'] that matches the most attributes: ['3 ounce bottle', 'Bright citrus', 'For sensitive skin']. =>
item1 = 'B08KBVJ4XN'
item2 = 'B078GWRC1J'
obj_attributes = ['3 ounce bottle', 'Bright citrus', 'For sensitive skin']
My sub-task is to identify the item in [{item1}, {item2}] that matches the most attributes: {obj_attributes}.
I need to count the number of attributes in {obj_attributes} that {item1} has. =>This item has 1 attribute.<=
I need to count the number of attributes in {obj_attributes} that {item2} has. =>This item has 2 attributes.<=
items = [{item1}, {item2}]
item_attributes = [1, 2]
item_to_purchase = next(iter(items[item_attributes.index(max(item_attributes))]))
print('You have identified {item_to_purchase}.')
#END#

I need to count the number of attributes in ['3 ounce bottle', 'Bright citrus', 'For sensitive skin'] that B08KBVJ4XN has. =>
item_to_check = 'B08KBVJ4XN'
obj_attributes = ['3 ounce bottle', 'Bright citrus', 'For sensitive skin']
My sub-task is to count the number of attributes in {obj_attributes} that {item_to_check} has.
> click[{item_to_check}] =>
[Back to Search]
[< Prev]
Barrel and Oak - Aluminum-Free Deodorant, Deodorant for Men, Essential Oil-Based Scent, 24-Hour Odor Protection, Cedar & Patchouli Blend, Gentle on Sensitive Skin (Mountain Sage, 1 oz, 2-Pack)
Price: $15.95
Rating: N.A.
[Description]
[Features]
[Reviews]
[Attributes]
[Buy Now]
<=
I need to check if {item_to_check} has the attribute '3 ounce bottle'. {item_to_check} does not have this attribute because it is described as '1 oz, 2-Pack'.
I need to check if {item_to_check} has the attribute 'Bright citrus'. {item_to_check} does not have this attribute because it is described as 'Cedar & Patchouli Blend'.
I need to check if {item_to_check} has the attribute 'For sensitive skin'. {item_to_check} has this attribute because it is described as 'Gentle on Sensitive Skin'.
{item_to_check} has 1 attribute.
print('This item has 1 attribute.')
#END#

I need to purchase B078GWRC1J with ['3 ounce bottle', 'Bright citrus', 'For sensitive skin']. =>
item_to_purchase = 'B078GWRC1J'
obj_attributes = ['3 ounce bottle', 'Bright citrus', 'For sensitive skin']
My sub-task is to purchase {item_to_purchase} with {obj_attributes}.
> click[{item_to_purchase}] =>
[Back to Search]
[< Prev]
scent [assorted scents][bright citrus][calming lavender][ginger fresh][simply non-scents]
size [travel set (4-pack)][3 ounce (pack of 1)][3-ounce (2-pack)]
Bright Citrus Deodorant by Earth Mama | Pregnancy and Breastfeeding, Contains Organic Calendula 3-Ounce
Price: $10.99
Rating: N.A.
[Description]
[Features]
[Reviews]
[Attributes]
[Buy Now]
<=
I will select from the 'scent' buttons: [assorted scents], [bright citrus], [calming lavender], [ginger fresh], [simply non-scents].
> click[bright citrus] =>You have clicked bright citrus.<=
I will select from the 'size' buttons: [travel set (4-pack)], [3 ounce (pack of 1)], [3-ounce (2-pack)].
> click[3 ounce (pack of 1)] =>You have clicked 3 ounce (pack of 1).<=

I can now select [Buy Now].
> click[Buy Now] =>You have clicked Buy Now.<=
print('You have purchased {item_to_purchase}.')
#END#

G.5 TextCraft

Crafting commands:
craft 3 dark oak sign using 6 dark oak planks, 1 stick
craft 4 dark oak planks using 1 dark oak log
craft 1 stick using 1 planks
craft 4 stick using 2 bamboo
craft 4 oak planks using 1 oak log
craft 1 dark oak fence using 2 stick, 4 dark oak planks
craft 1 warped stairs using 6 warped planks
craft 3 oak sign using 6 oak planks, 1 stick
Goal: craft dark oak sign.
craft_command_list = ['craft 3 dark oak sign using 6 dark oak planks, 1 stick', 'craft 4 dark oak planks using 1 dark oak log', 'craft 1 stick using 1 planks', 'craft 4 stick using 2 bamboo', 'craft 4 oak planks using 1 oak log', 'craft 1 dark oak fence using 2 stick, 4 dark oak planks', 'craft 1 warped stairs using 6 warped planks', 'craft 3 oak sign using 6 oak planks, 1 stick']
craft_command_item_list = ['dark oak sign', 'dark oak planks', 'stick', 'stick', 'oak planks', 'dark oak fence', 'warped stairs', 'oak sign']
target_item = 'dark oak sign'
target_item_count_total = 1
idx = craft_command_item_list.index(min([x for x in craft_command_item_list if target_item in x], key=len))
target_craft_command = craft_command_list[idx]
To craft {target_item}, I need to perform the following action until I have at least {target_item_count_total} {target_item}: {target_craft_command} =>You have completed the action.<=
#END#

I need to perform the following action until I have at least 1 dark oak sign: craft 3 dark oak sign using 6 dark oak planks, 1 stick =>
target_item = 'dark oak sign'
target_item_count_total = 1
target_craft_command_result_count = 3
precursor1 = 'dark oak planks'
precursor1_count_command = 6
precursor2 = 'stick'
precursor2_count_command = 1
Since I need at least 1 {target_item} and each action produces 3 {target_item}, The number of times I need to perform the action is ceiling of 1/3, which is 1.
target_craft_command_reps = 1
precursor1_count_total = precursor1_count_command * target_craft_command_reps
precursor1_count_total = precursor2_count_command * target_craft_command_reps
Next, I need to get or craft {precursor1}.
To start, I first need to check if I can get {precursor1_count_total} {precursor1}. =>You cannot get the material.<=
Since I cannot get {precursor1}, I need to craft it.
idx = craft_command_item_list.index(min([x for x in craft_command_item_list if precursor1 in x], key=len))
precursor1 = craft_command_item_list[idx]
precursor1_craft_command = craft_command_list[idx]
To craft {precursor1}, I need to perform the following action until I have at least {precursor1_count_total} {precursor1}: {precursor1_craft_command} =>You have completed the action.<=
Next, I need to get or craft {precursor2}.
To start, I first need to check if I can get {precursor2_count_total} {precursor2}. =>You cannot get the material.<=
Since I cannot get {precursor2}, I need to craft it.
idx = craft_command_item_list.index(min([x for x in craft_command_item_list if precursor2 in x], key=len))
precursor2 = craft_command_item_list[idx]
precursor2_craft_command = craft_command_list[idx]
To craft {precursor2}, I need to perform the following action until I have at least {precursor2_count_total} {precursor2}: {precursor2_craft_command} =>You have completed the action.<=
Finally, I will perform the action 1 time.
> craft {target_craft_command_result_count} {target_item} using {precursor1_count_command} {precursor1}, {precursor2_count_command} {precursor2} =>Crafted 3 minecraft:{target_item}.<=
print('You have completed the action.')
#END#

I first need to check if I can get 6 dark oak planks =>
get_item = ‘dark oak planks’
get_item_count_total = 6
> get {get_item_count_total} {get_item} =>Could not find {get_item}.<=
I cannot get the {get_item}.
print(‘You cannot get the material.’)
#END#
