Interactive and Expressive Code-Augmented Planning with Large Language Models

Anthony Z. Liu[1 2]**, Xinhe Wang**[2]**, Jacob Sansom**[2]**, Yao Fu**[2]**,**
Jongwook Choi[2]**, Sungryull Sohn**[3]**, Jaekyeom Kim**[3]**, and Honglak Lee**[2 3]

[1]Correspondence anthliu@umich.edu
[2]University of Michigan
[3]LG AI Research

Abstract

Large Language Models (LLMs) demonstrate strong abilities in common-sense reasoning and interactive decision-making, but often struggle with complex, long-horizon planning tasks. Recent techniques have sought to structure LLM outputs using control flow and code to improve planning performance. However, code-based approaches can be error-prone and insufficient for handling ambiguous or unstructured data. To address these challenges, we propose REPL-Plan, an LLM planning approach that is *fully code-expressive* (it can utilize all the benefits of code) while also being *dynamic* (it can flexibly adapt from errors and use the LLM for soft reasoning). In REPL-Plan, an LLM tackles tasks by interacting with a Read-Eval-Print Loop (REPL) that iteratively executes and evaluates code—similar to language shells or interactive notebooks—enabling the model to flexibly correct errors and adapt dynamically to task requirements. We demonstrate that REPL-Plan achieves strong results across various planning domains compared to previous methods.

1 Introduction

Large Language Models (LLMs) have shown strong capabilities in planning and decision making tasks (Wang et al., 2023b; Xi et al., 2023). LLMs have strong domain-knowledge and common-sense priors (Kojima et al., 2023; Brown et al., 2020), which are required in many of these planning tasks. For instance, answering *"What steps should I take to prepare dinner?"* requires knowledge about food preparation and kitchen environments. However, LLMs are imperfect and can struggle to make accurate decisions for complex and long-horizon tasks — often making "hallucinations" and incorrect short-term decisions (Kambhampati et al., 2024; Valmeekam et al., 2023; Liu et al., 2023).

Prior work has improved planning performance by guiding an LLM by enforcing levels of **structure** to the LLM's outputs and decisions, guiding LLM-planning to output sequences or trees of thoughts and subtasks (Yao et al., 2022b; Kojima et al., 2022; Yao et al., 2023; Schroeder et al., 2024). Prior work has also structured LLM decisions by using **code**, which we call here code-augmented LLM-planning. As recent LLMs have been trained on large code bases (Chen et al., 2021; Yang et al., 2024), LLMs have the ability to utilize the *control flow* structures of code to plan accurately. Some works prompt an LLM to write code or pseudocode solely to help the LLM self-structure its decisions, without explictly executing code (Chae et al., 2024). Other works prompt an LLM to write code that is directly executed to solve the given task (Gao et al., 2023; Wang et al., 2023a).

However, using LLM-written code also introduces the inherent limitations of code to planning. In this work, we address the following limitations: (1) *Soft reasoning* sub-problems: many tasks require a planner to answer or solve some "soft reasoning" task that is not easily solved using code. A planner may need to interpret unstructured observation data, or make decisions on possibly subjective choices (e.g. *"purchase the item that best matches the user's request"*). (2) *Bottom-up nature* of writing code: solving tasks with code requires a planner to often solve tasks from a bottom-up manner. This includes writing functions for sub-processes before writing code to solve the main tasks and writing case statements that must consider every possible output of a function or result of a sub-task. Generating accurate code in this manner requires precise and accurate forethought. (3) *Coding bugs*: writing accurate code in one pass is difficult, even for skilled human coders.

In this work, we aim to address these challenges by developing a **top-down** framework for code-augmented LLM planning that is both **dynamic** and **expressive**. Inspired by the way human developers prototype and debug code, we leverage interactive environments such as REPLs (Read-Eval-

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 20330–20354
July 27 - August 1, 2025

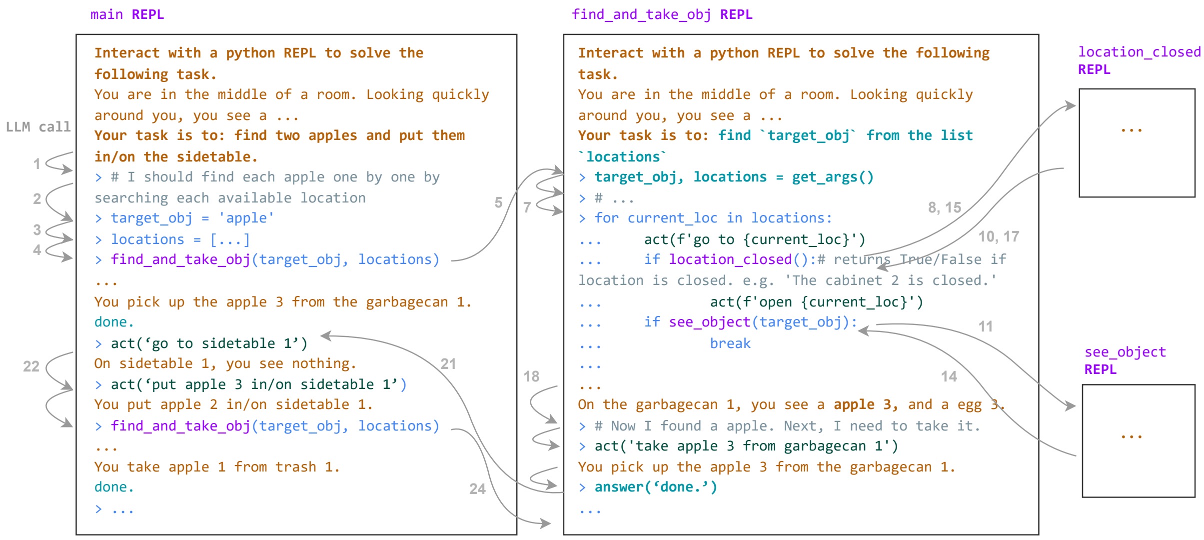

Figure 1: **REPL-Plan** is an approach for using code to augment LLM-planning where an LLM interacts with an **LLM-REPL**, iteratively writing code and interacting with the environment by calling the `act` function. The interactive nature allows the LLM to adapt to changes in the environment and errors in the code. An LLM-REPL also allows the LLM to recursively spawn other LLM-REPL by calling undefined functions (e.g. `find_and_take_obj`). This allows the LLM to recursively solve subtasks in a top-down divide-and-conquer manner, and to separate subtasks that may be ill-suited for code, such as soft-reasoning tasks (e.g. `location_closed` and `see_object`). An LLM can combine these tools (child LLM-REPLs, iterative coding) to express complex workflows that can solve tasks in a compact way.

Print Loops, commonly referred to as *language shells*) and other computing paradigms (Steele and Gabriel, 1996). Many researchers are already familiar with tools like iPython and Jupyter Notebook—practical examples of REPLs—where code is executed one line at a time, providing immediate feedback that supports incremental development and on-the-fly error correction. This interactive, step-by-step process is central to our approach, enabling the LLM to iteratively refine its plans and responses.

We introduce **REPL-Plan**: an approach for solving complex planning problems (shown in Figure 1) by using LLMs to interact with **LLM-REPLs**, where the LLM writes code line-by-line, writing and calling functions to interact with the planning environment. Specifically, LLM-REPLs are an extension of REPLs — in addition to REPL functionality, code in LLM-REPLs can *recursively "spawn" child LLM-REPLs*. This recursive spawning enables the LLM to abstract and solve subtasks in a top-down divide-and-conquer manner, isolating those tasks that require heuristic reasoning or involve uncertainty due to incomplete or noisy information—challenges that are not readily addressed by conventional, rigid rule-based coding approaches.

Our work makes the following contributions:

- **REPL-Plan**: a novel approach for planning using **LLM-REPL**s, which are an extension of REPLs (e.g. language shells, code notebooks). LLM-REPLs enable a LLM to make decisions in a top-down way that is both dynamic and expressive.
- We show that REPL-Plan achieves strong performance in challenging text-based sequential decision making environments ALFWorld (Shridhar et al., 2021) and WebShop (Yao et al., 2022a). We also show REPL-Plan achieves strong performance on a novel real-world web navigation task that requires handling complex web observations.
- We test the robustness of REPL-Plan by conducting ablations — testing the code correction/prediction ability by giving the LLM faulty demonstrations.

2 Approach

2.1 Problem Statement

In a typical interactive planning task, a large language model agent (LLM-agent) is given a task description t and must interact with an environment for a number of timesteps to accomplish the task t. At each timestep i, the agent is given a language observation o_i of the current state of the environment and can predict a language action a_i which can affect the environment (for the agent to accomplish t). To help the LLM-agent, it is given k-trajectories (or demonstrations) $\{\tau_1, \ldots, \tau_k\}$ of solving some prior tasks that are created by an expert. The LLM-agent must follow the demonstrations to solve the current task t.

A simple way to apply the LLM is to simply model the next action using the language-modeling probabilities of the LLM. However, many real-world tasks are complex — they may require solving multiple smaller sub-tasks, and long-horizon — they require many action predictions to solve the task. Enforcing structure, such as writing *code*, to the LLM-agent's decisions, can be used to address these issues.

In this work, we propose a novel method for using code to augment LLM-agents, which addresses the limitations of using code with planning, described in Section 1: (1) "Soft reasoning" subproblems, (2) planning bottom-up, and finally (3) coding errors.

2.2 LLM-REPLs

We base our approach on the concept of a REPL (Read-Eval-Print-Loop), also known as a language shell or code notebook, is a programming environment that tracks program state (global/local variables) at each timestep where (1) a user inputs a statement of code (variable assignment, loop, function definition), (2) this line of code is *evaluated*, to obtain the next program state, (3) the output of the previous line of code is shown to the user, and finally (4) repeat step (1).

A REPL is useful for task solving in a *dynamic* manner — results (and possible errors) are immediately reflected in the output of the program state. Then, a user (or LLM), can simply write another statement of code that can correct their previous mistake, without re-writing the entire program.

We extend the idea of using a REPL *recursively* — where any REPL can spawn *another* REPL in a top-down manner, in order to solve any sub-task — a "subtask" REPL. We call this idea an **LLM-REPL**. In Section 3.3, we show experimentally adding recursive spawning is crucial for solving tasks.

To create a **LLM-REPL**, we simply add predefined functions, or *primitives*, to the program state. The LLM can call these primitives to choose to *recursively spawn* child LLM-REPLs, and manage context-passing between parent-child LLM-REPLs.

- `[subtask](args)`: spawns an LLM-REPL with the name `[subtask]`. The LLM is then queried using the `[subtask]` REPL until context is passed back to this REPL or another child REPL is spawned. If the `[subtask]` LLM-REPL has previously been spawned, then execution history will *continue* from previous. Otherwise, a task description for the LLM-REPL will be queried, and a new LLM-REPL will be created. Note the new child LLM-REPL does not share any variable state with the parent, so the following primitives can be used to pass context.
- `get_args()`: returns the arguments passed from the parent of the current LLM-REPL.
- `answer(a)`: passes and returns the value `a` back to the parent LLM-REPL. Execution is passed back to the parent.

Pseudo-code for how a practical implementation of LLM-REPLs is given in Appendix A.2.

2.3 REPL-Plan

REPL-Plan is simply planning and decision-making through collection of LLM-REPLs. As all the planning and actions can be done in a LLM-REPL, REPL-Plan is code-*expressive*.

To execute **REPL-Plan**, we can adapt an LLM-REPL by adding more primitive functions for interacting with the environment.

- `act(a)`: passes and executes the action `a` in the environment.
- `get_obs()`: passes the recent observation as a string. This may be used by an planner to execute some string processing code when an observation is too large (e.g. a whole web page).

For each timestep i, we run the LLM-REPL until the `act(a)` function is called, then pass in any observations as needed to the in-code-environment-outputs of the LLM-REPLs. We demonstrate how LLM calls, code, and actions from `act` are executed across different parent-child LLM-REPLs in Figure 2.

Few-shot Setting. We use a few-shot setting where all agents (REPL-Plan and baselines) are given k expert demonstrations — including environment observations and code outputs (Yao et al., 2022b; Schroeder et al., 2024; Sun et al., 2023). For a fair comparison, demonstrations for all approaches are derived from the same training environment trajectories. Specifically, REPL-Plan uses these demonstrations to keep a *global REPL pool*, a set of spawned REPLs from previous task executions and demonstrations. In this way, an filter_search() REPL

```
Task: Find any matching items in the search results.
>>> description = get_args()
>>> matching = []
>>> for i in range(5):
...     matching.extend(filter_page(description))
...     act('click [Next >]')
...
>>> answer(matching)
```

filter_page() REPL

```
Task: Find any matching items on the current page
>>> description = get_args()
>>> item_links = parse_items()
>>> matching = []
>>> for item in range(item_links):
...     act(f'click [{item}]')
...     if item_matches(item, description):
...         matching.append(item)
...     act(f'click [< Back]')
...
>>> answer(matching)
```

filter_search() Trace	filter_page() Trace
$\ell_2 \leftarrow$ LLMQuery()	
description = get_args()	
$\ell_3 \leftarrow$ LLMQuery()	
matching = []	
$\ell_7, \ldots, \ell_7 \leftarrow$ LLMQuery()	
for i in range(5):	
NameError(filter_page)	
create REPL: subtask $\leftarrow$ LLMQuery()	
matching.extend(	
filter_page(description))	
	$\ell_2 \leftarrow$ LLMQuery()
	description = get_args()
	$\ell_3 \leftarrow$ LLMQuery()
	NameError(parse_items)
	create REPL: subtask $\leftarrow$ LLMQuery()
	item_links = parse_items()
	$\ell_4 \leftarrow$ LLMQuery()
	matching = []
	$\ell_5, \ldots, \ell_9 \leftarrow$ LLMQuery()
	for item in range(item_links):
	act(f'click [{item}]')
	NameError(item_matches)
	...
	act(f'click [< Back]')
	...
	answer(matching)
act('click [Next >])'	
...	...
answer(matching)	

Figure 2: A toy example of context passing that is possible in REPL-Plan. In the toy example, the task is parse all items on a search page that match a given description. We show a sample generated code in the two code snippets above, where the agent splits the task into 3 different sub-tasks: (1) `filter_page`, parsing any matching items on the current page, (2) `parse_items`, parse any item links on the current page, and (3) `item_matches`, determine if the current item page matches the description.

LLM can perform in-context-learning from previous tasks and demonstrations to output correct and consistent code.

3 Experiments

We evaluated REPL-Plan against baselines on various language-based environments. Each of these environments is challenging — to be successful, an agent must be able to understand and predict long-horizon complex action sequences, and have enough language-understanding to interpret the language observations and actions. We show that REPL-Plan is an effective balance in using code for planning — using code to structure complex, repeating, and language-understanding sub-tasks.

3.1 Environments

We tested on the following environments:

ALFWorld. ALFWorld (Shridhar et al., 2021) is a text-based simulated-household embodied agent environment, where an agent must accomplish 6 different types of tasks related to navigating and interacting with a household, e.g. searching the kitchen for an apple, heating it, then placing it on a table.

WebShop. WebShop (Yao et al., 2022a) is an interactable e-commerce environment, where an agent is given a target product description—in which a human annotator described a category, key attributes, and price limit—and the agent must navigate the e-commerce site by searching for and buying a product that matches the target description. An agent's success is measured based on how many of the purchased item's attributes match with the ground truth attributes.

In WebShop, the search page is configured to return the top k results from the search query. Prior approaches have set $k = 3$, and tested strategies where the best item is selected only from the first page (Top-3). We additionally test a setting where $k = 10$. In this setting, we tested the strongest baseline, THREAD (Schroeder et al., 2024) —an LLM approach that uses a text-based divide-and-conquer strategy— by (1) using the same Top-3 strategy, and (2) using a Top-20 strategy — searching twice and selecting the best among 20. Details included in Appendices A.4 and A.5.

Real-World Web Tasks. To test the scalability of REPL-Plan, we wanted to test on real-world web navigation tasks on (1) live real websites and (2) very long-horizon tasks ($\geq$ 30 visited pages). We were not aware of existing benchmarks that fit these requirements (Zhou et al., 2023; Deng et al., 2024), so designed a novel set of web tasks. These tasks are difficult as actual websites can be extremely large (4k-20k tokens, compared to $\leq$ 500 tokens per web-page in WebShop). We tested the agents on distributions of tasks where the goal is to navigate an E-commerce website and add item(s) that match a given description to the cart. We divided the tasks into a "simple" category, where the agent only needs to add an item of any description (e.g. add any laptop to the cart from `bestbuy.com`) and a "complex" category, where the agent must add as many items as it can find that match a specific description (e.g. loop through each search result and add every printer that is capable of printing at least 12 pages per minute to the cart). The full task descriptions and examples of observations and actions can be found in Appendix A.7.

3.2 Results

We find that elements of *top-down recursive, dynamic*, and *code-expressivity* are important for approaches to score a high success rate on all environments (top-down recursive and dynamic for ALFWorld, dynamic and code-expressivity for WebShop and Real-World Web). For example, the baseline THREAD (Schroeder et al., 2024) has top-down recursive and dynamic elements, but not code-expressivity, has high success rate on ALFWorld and $k = 3$ WebShop, but fails to scale in $k = 10$ WebShop and Real-World Web.

External Memory	Method	ALFWorld SR (%)
Yes	Reflexion[1]	76.1
	AdaPlanner[2]	82.8
	RAP[3]	85.8
	AutoGuide[4]	79.1
No	ReAct[5]	53.7
	ADaPT[6]	82.1
	THREAD[7]	95.5
	REPL-Plan	**97.0**

Table 1: The average success rate (SR (%)) of running each method on ALFWorld with GPT-3.5-instruct (OpenAI, 2023a). We note that methods 1-4 use external memory — they save information from previous test tasks to new test tasks. [1] (Shinn et al., 2023) [2] (Sun et al., 2023) [3] (Kagaya et al., 2024) [4] (Fu et al., 2024) [5] (Yao et al., 2022b) [6] (Prasad et al., 2024) [7] (Schroeder et al., 2024)

	Real-World Web (% of Expert Score)	
Method	Simple	Complex
ReACT	**86.7**	17.6
THREAD	13.3	0.0
REPL-Plan	**86.7**	**39.6**

Table 2: We tested the baselines ReACT and THREAD against REPL-Plan in the Real-World Web Tasks with GPT4o-mini. For each method, we present the percentage of the expert score reached.

ALFWorld. We show the results of testing approaches on ALFWorld in Table 1. Our approach, REPL-Plan, achieves the highest success rate (SR) of 97.0% compared to the best baseline, THREAD (Schroeder et al., 2024), with 95.5% SR.

We find that *top-down recursive* and *dynamic* approaches are crucial to solving this task, which ours and THREAD both have. Both REPL-Plan and THREAD effectively divide-and-conquer various sub-tasks of ALFWorld (e.g. dividing a task into (1) finding an apple, (2) heating it, then (3) placing it on the table), and are able to effectively change plans if there are environment issues (e.g. object is misplaced or mis-referenced).

Model	Setting	Strategy	External mem.	Method	WebShop SR(%)	WebShop Score(%)
GPT-3.5-instruct	Page $k = 3$	Top-3	Yes	Reflexion	38	64.4
				LATS[1]	40	76.0
				RAP	48	76.1
				AutoGuide	46	73.4
			No	ReAct	37	59.5
				ADaPT	44	60.0
				TDAG[2]	45	64.5
				THREAD	**49**	**76.3**
				REPL-Plan	47	74.2
GPT4o-mini	Page $k = 10$	Top-3	No	THREAD*	21	42.1
		Top-3		REPL-Plan	37	69.9
		Top-20		REPL-Plan	**52**	**77.1**

Table 3: The success rate (SR) and average score of each model-method-setting on Webshop. For the $k = 3$ setting, models were run using GPT-3.5-instruct and with a top-3 strategy (examine only the top-3 items). For the $k = 10$ setting, we ran THREAD and REPL-Plan using GPT4o-mini (OpenAI, 2023b), and showed the performance of each the viable strategy. * note that we made modifications to run THREAD as intended, described in Appendix A.4. [1](Zhou et al., 2024) [2](Wang et al., 2024b)

WebShop. We show the results of various approaches on different LLMs, settings, and strategies on WebShop in Table 3.

For simple strategies such as Top-3, i.e. simply searching and then comparing the Top-3 items, many baselines score a high ($\geq 46\%$) success rate: RAP, AutoGuide, THREAD, and REPL-Plan, with THREAD reaching the the highest SR of 49%. However, this strategy is simplistic and limited — trajectories can be completed within 10 actions, and success primarily relies on the LLM's ability to understand web-page observations.

By testing on the Page $k = 10$ setting with GPT4o-mini on REPL-Plan and the strongest baseline THREAD, we find that a combination of *top-down recursive* and code-*expressive* approaches are valuable in attaining a high SR in this setting and being able to successfully execute a Top-20 strategy.

REPL-Plan, using loops and variables (that can be passed across LLM-REPLs), is able to reach a 52% SR with GPT4o-mini.

Real-World Web Environment. We show the results of running ReACT, THREAD, and REPL-Plan in the Real-World Web Environment in Table 2. We find that each agent is able to solve tasks from the Simple distribution (tasks outlined in Appendix A.7). These tasks are short horizon and can be completed in 5 actions. However, they still require the agent to be able to manage and interpret the long web page observations, which can be 4k-20k tokens long. Despite multiple attempts to improve THREAD described in Appendix A.4, we found THREAD could not effectively plan when there are long observation contexts. REPL-Plan is able to more effectively break down the observations using full code expressitivity — scoring 39.6% of the expert score on the complex tasks, where the baselines ReACT and THREAD score 17.6% and 0.0% respectively. We also give a qualitative analysis of this result in Figure 3. In this analysis, we show why REPL-Plan is able to (1) effectively manage large observations through code and LLM-REPL spawning, and (2) mitigate the effects of LLMs hallucinating using code.

3.3 Ablations

We tested REPL-Plan's ability to run under imperfect conditions in the page $k = 3$ WebShop environment, using the Top-3 strategy.

Buggy Demonstrations. In order to test the error-correcting abilities of REPL-Plan, we injected minor coding mistakes into the demonstrations and tested the performance.

Specifically, we injected two errors: (1) we saved variable descriptions using quotes: `description = "..."`. This would cause errors when there are quotes in the desired description contained quotes:

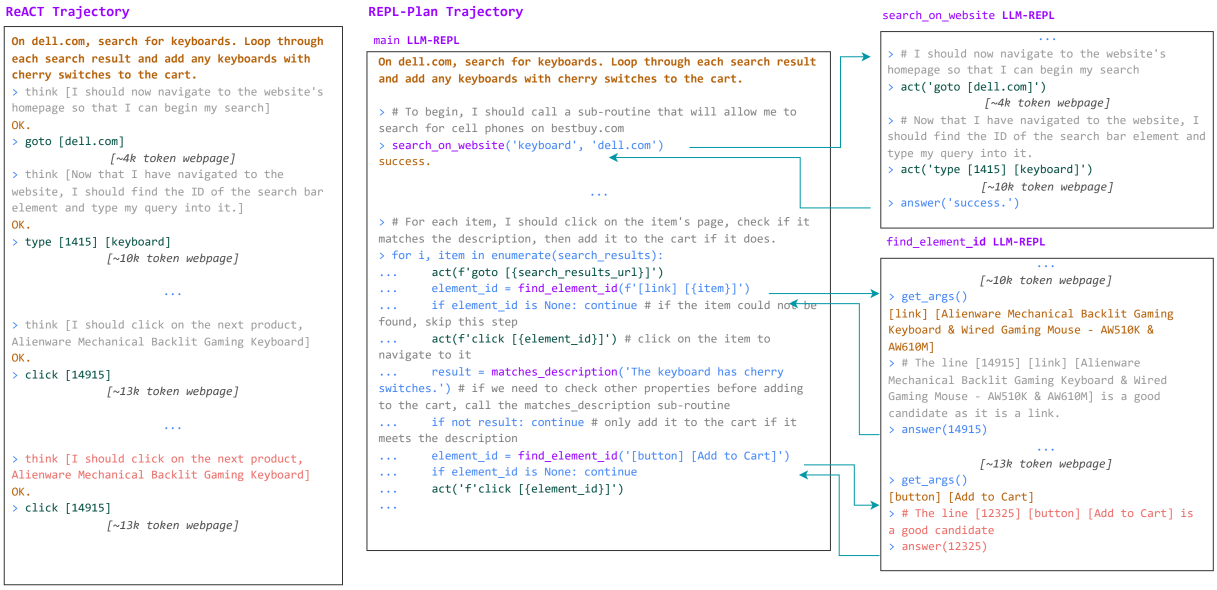

Figure 3: For a qualitative analysis, we include truncated versions of trajectories from REPL-Plan and the baseline ReACT on a real-world web loop-like task. In this task (Appendix A.7), the agent is shown long web pages (4k-15k tokens long), and must interact with the page using element IDs (labeled with integer IDs). In the trajectories, (1) REPL-Plan is able manage large observations and long prompt contexts by sub-dividing the tasks into different LLM-REPLs. And (2), we find that in both trajectories, ReACT and REPL-Plan both run into hallucination errors from GPT4o-mini (highlighted in red). On the left, in ReACT, the LLM gets "lost", and re-checks a product it already checked before. This causes the ReACT agent to loop infinitely. On the right, in REPL-Plan, agent hallucinates a link element ID. However, due to code in the main loop REPL-Plan mitigates the effect of the hallucination — the agent clicks the wrong element ID, but still continues to search for candidate products.

e.g. `" 12" ..."`, and (2) we removed key variable definitions that are used later in the demonstration. We found that the LLM is relatively robust to these errors, only failing one more task on GPT-4o-mini than without any errors. We show a small sample of these corrections in Appendix A.10.

No-Subtask-REPLs. Next, we tested how the top-down decomposition of tasks affects the performance of LLM-REPL. We created a "No-Subtask-REPL" ablation where the agent aims to solve the task without spawning any child LLM-REPLs.

We find this strategy is conceptually similar to a code-augmented version of prior approaches, such as ReACT (Yao et al., 2022b). However, given that this approach cannot de-compose tasks, the REPL code/observation history becomes large and untenable for the agent to solve WebShop tasks. The performance decreases by around half: $52\% \rightarrow 24\%$ and $44\% \rightarrow 20\%$ respectively for GPT-3.5-instr. and GPT-4o-mini.

Zero-shot Subtask-REPL. As k-shot demonstrations are given as a pool of LLM-REPLs which are used in REPL-Plan, we tested the performance under the condition that one of those LLM-REPLs was removed from this pool, and the LLM must zero-shot infer both the task description and code writing for this LLM-REPL.

To do this, we removed the `get_requirements` LLM-REPL, which is designed to examine the item on the current page, and return an integer count of the number of matching attributes. This is a crucial function that is necessary for completing the task and called multiple times.

We found that in half of the trials the LLM is able to correctly zero-shot infer a subtask description and code for the LLM-REPL, with a final performance of 28% (from 52%) and 16% (from 44%) respectively for GPT-3.5-instr. and GPT-4o-mini. When the LLM incorrectly infers this subtask, it often causes the agent to enter an error loop, causing the agent to fail the overall task. We analyze success and failure cases more closely and give qualitative analysis in Appendix A.11. Although zero-shot task/code inference is not the focus of this work, this indicates an extension work to understand REPLs and LLM-REPLs in these settings.

4 Related Work

LLM-Agents. With the increasing capabilities of LLMs, several works have proposed robust agents that interact with text-based environments (LLM-agents). These works show promising results in challenging benchmarks such as embodied

Method	Ablation	WebShop sample SR (%, $n = 25$)	
		GPT-3.5-instr.	GPT-4o-mini
REPL-Plan	Full Model	52	44
	(1) Buggy Demo.	52	40
	(2) No-Subtask-REPLs	24	20
	(3) Zero-shot Subtask-REPL	28	16

Table 4: We tested REPL-Plan on the WebShop Page $k = 3$ setting with various corruptions to the demonstrations. (1) Buggy Demo.: We added minor bugs (non-escaped strings, missing variables) to the demonstration. (2) No-Subtask-REPLs: We tested a version of REPL-Plan (and demonstrations) without recursive spawning. (3) Zero-shot Subtask-REPL: We removed one of the Subtask-REPLs from the demonstration, such that the agent needed to zero-shot infer code.

agent tasks (Shridhar et al., 2021) and web navigation (Yao et al., 2022a; Zhou et al., 2023). A number of these LLM-agents, based on works by Yao et al. (2022b) and Shinn et al. (2023), take in a sequence of text observations-action pairs as history and generate single actions at each time-step. However, as the LLMs tend to hallucinate incorrect actions, and incorrectly plan ahead, several works, including REPL-Plan, have been proposed that incorporate *code*-augmentation to LLM-agents, and recursively dividing tasks for an LLM-agent.

Code-Augmented LLMs. Current LLMs (without guidance) suffer from long context understanding (Li et al., 2024) and logical and arithmetic mistakes (Wei et al., 2023). To alleviate these problems, recent works have used external tooling such as code. For example, Program-Aided Language Models (PAL) (Gao et al., 2023) generate programs and run a Python interpreter to get solutions for mathematical, symbolic, and algorithmic reasoning problems. LLMs and code have also been used in robotics to generate robust policies (Liang et al., 2023). Compared to these works, our work REPL-Plan focuses on the interactive decision-making setting, and focuses on how we can fix the problems of running code in these settings by creating an approach that is fully interactive (the LLM interacts with a LLM-REPL to code) and top-down recursive (with spawning child LLM-REPLs). CodeAct and Mint (Wang et al., 2024a, 2023c) are works that proposes an interactive LLM-code generating paradigm where the agent interacts with a human user to solve a task using code. However, compared to REPL-Plan, these works are not designed for the LLM-agent space, and also do not split interactive code in a recursive manner. AdaPlanner (Sun et al., 2023) also focuses on the planning setting by writing and refining code, but is not interactive, as a code plan must be written entirely in one pass.

Decomposing Tasks. In the few-shot in-context learning setting, LLMs can struggle with increased task complexity. To deal with this, Khot et al. (2023) have proposed to decompose challenging tasks to simpler sub-tasks and solve these individually. ADaPT (Prasad et al., 2024) recursively decomposes complex sub-tasks as-needed to adapt to LLMs' capability and task complexity. Prior works (Wang et al., 2023a; Liu et al., 2024) have decomposed tasks into re-usable code blocks, or prompts, which can be re-used in new tasks. THREAD (Schroeder et al., 2024) is a work that recursively divides LLM context into *thread*, which each individually solve some sub-task of the whole. Thread also contains partial Python code executability, by saving variables and computing one-line Python statements. REPL-Plan is different from THREAD: (1) REPL-Plan is fully code-expressive, any variables can be passed to other REPLs, loops can be run, etc. (2) REPL-Plan handles recursive sub-task spawning in a code-expressive way (using functions). In THREAD, these sub-tasks are spawned "anonymously", and cannot be re-used. These two differences allow REPL-Plan to efficiently handle program contexts in many planning situations.

5 Conclusion

In this work, we proposed REPL-Plan, a code-augmented planning approach for Large Language Models (LLMs) that is both dynamic and code-expressive, enabling effective top-down planning. By extending the Read-Eval-Print Loop (REPL) paradigm, we create LLM-REPLs, which allow LLMs to interactively generate code, recursively spawn child LLM-REPLs, and dynamically adapt to complex tasks. Through evaluations on diverse planning benchmarks, REPL-Plan demonstrates superior performance compared to baseline methods, showcasing its ability to handle complex, long-horizon tasks with flexibility and robustness.

6 Limitations

One limitation of this work is the abilities of Large Language Models (LLMs) to generalize beyond k-shot settings — particulary when using REPL-Plan. As we showed in the ablation experiments in Table 4, LLMs have some capabilities of generalizing beyond the k-demonstrations (when the k-demonstrations contain partially buggy code, zero-shot subtask-REPL inference). However, beyond this, such as writing zero-shot code for new LLM-REPL-code for more significant subtasks, and generalizing from improperly structured code (No-Subtask-REPLs), it can be difficult for the LLMs to successfully use REPL-Plan. Future work could address this by examining how to demonstrate REPL-Plan under out of distribution settings, and also examining how to adapt LLMs for REPL-coding in a more manner beyond in-context learning (e.g. fine tuning LLMs to write code for REPLs).

In addition, using code generation for LLM-planning, including REPL-Plan, introduces the possibility of catastrophic code failures, such as infinite loops. While this is a limitation in all code-augmented LLM-planning approaches, REPL-Plan gives LLMs a chance to recover from these failures, e.g. by showing the LLM error traces of memory overflow errors.

Another limitation is the efficiency of some use-cases of REPL-Plan. In many scenarios, a subtask can be solved using a pure code approach — an LLM writes code for an LLM-REPL that can solve the subtask without intervention. Future work could address this by having a mechanism for determining when code written by the LLM is independent of LLM decisions, and "automate" subtasks if possible.

Acknowledgments

This work was supported in part by funding from LG AI Research and NSF Graduate Research Fellowship.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Hyungjoo Chae, Yeonghyeon Kim, Seungone Kim, Kai Tzu-iunn Ong, Beong-woo Kwak, Moohyeon Kim, Seonghwan Kim, Taeyoon Kwon, Jiwan Chung, Youngjae Yu, et al. 2024. Language models as compilers: Simulating pseudocode execution improves algorithmic reasoning in language models. *arXiv preprint arXiv:2404.02575*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2024. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36.

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. *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. *Preprint*, arXiv:2211.10435.

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. *Preprint*, arXiv:2402.03610.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Kaya Stechly, Mudit Verma, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. 2024. Llms can't plan, but can help planning in llm-modulo frameworks. *arXiv preprint arXiv:2402.01817*.

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. *Preprint*, arXiv:2210.02406.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large language models are zero-shot reasoners. *Preprint*, arXiv:2205.11916.

Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context llms struggle with long in-context learning. *Preprint*, arXiv:2404.02060.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. *Preprint*, arXiv:2209.07753.

Anthony Zhe Liu, Jongwook Choi, Sungryull Sohn, Yao Fu, Jaekyeom Kim, Dong-Ki Kim, Xinhe Wang, Jaewon Yoo, and Honglak Lee. 2024. Skillact: Using skill abstractions improves llm agents. In *ICML 2024 Workshop on LLMs and Cognition*.

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

OpenAI. 2023a. Gpt-3.5. `https://platform.openai.com/docs/models/gpt-3-5`. Accessed via OpenAI API.

OpenAI. 2023b. Gpt-4. `https://openai.com/research/gpt-4`. Accessed via OpenAI API.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. 2024. Adapt: As-needed decomposition and planning with language models. *Preprint*, arXiv:2311.05772.

Philip Schroeder, Nathaniel Morgan, Hongyin Luo, and James Glass. 2024. Thread: Thinking deeper with recursive spawning. *Preprint*, arXiv:2405.17402.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *ICLR*.

Michael Sipser. 1996. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29.

Guy L Steele and Richard P Gabriel. 1996. The evolution of lisp. In *History of programming languages—II*, pages 233–330.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2023. Adaplanner: Adaptive planning from feedback with language models. *Preprint*, arXiv:2305.16653.

Karthik Valmeekam, Sarath Sreedharan, Matthew Marquez, Alberto Olmo, and Subbarao Kambhampati. 2023. On the planning abilities of large language models (a critical investigation with a proposed benchmark). *Preprint*, arXiv:2302.06706.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023a. Voyager: An open-ended embodied agent with large language models. *Preprint*, arXiv:2305.16291.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2023b. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. Executable code actions elicit better llm agents. *Preprint*, arXiv:2402.01030.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2023c. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *arXiv preprint arXiv:2309.10691*.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.

Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R. Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, Heng Ji, and Chengxiang Zhai. 2024. If llm is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. *Preprint*, arXiv:2401.00812.

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

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models, 2023. *URL https://arxiv. org/pdf/2305.10601. pdf*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022b. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning acting and planning in language models. *Preprint*, arXiv:2310.04406.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

A Appendix

A.1 Interactive Coding and REPLs

In this section, we provide a brief introduction to the history of interactive coding and REPLs. The REPL (Read-Eval-Print-Loop) paradigm originates with Lisp in the 1960s (Steele and Gabriel, 1996) and is the core of interactive coding. In interactive coding, a user iteratively interacts with a coding environment in a REPL loop: A user inputs a code expression which is "Read" into the code interpreter, "Evaluated" by the interpreter, the code output is "Printed" to the user, and the process "Loops" and the user inputs another expression of code.

Compared to traditional programming, where a user writes a program, compiles, then runs and debugs, interactive coding with REPLs allows a user to write code iteratively during runtime which facilitates exploratory programming and debugging.

The REPL paradigm is now used ubiquitously in command-line shells and scripting languages such as Python and Javascript. The REPL paradigm has further expanded into popular interactive coding applications and concepts, such as computing notebooks, such as Jupyter Python notebooks and Colab which use a browser-based REPL.

A.2 LLM-REPL Implementation

Algorithm 1: LLM-REPL Pseudocode

```
class LLMREPLFunction
    def __init__(task_description)
        task ← task_description
        history ← [] // REPL code/output history
        locals ← {get_args, answer, act, get_obs} // Insert REPL primitives to
            local state
        pool ← /* global REPL pool */
    def __call__(*args)
        while True do
            if /* need new REPL block */ then
                code ← QueryLLM(task, history)
            try
                exec(code, locals) // Return if answer is called
                history.append(/* format code, stdout, stderr */) // only reached
                    if code finishes without interrupt
            except NameError
                // When LLM calls an unseen function
                fname ← /* get undefined variable name */
                /* If REPL not in global REPL pool */
                subtask ← QueryLLM(fname, history)
                locals[fname] ← LLMREPLFunction(subtask)
            /* Return to code state after last interrupt */
```

Implementing an LLM-REPL, as described in Algorithm 1, can be accomplished in various approaches. The primary implementation challenge is how to handle child LLM-REPL spawning — whenever there is an undefined function call, an LLM-REPL should be created and run for that undefined function. Given the undecidable nature of determining whether a function or variable is undefined (Sipser, 1996), the most flexible approach is to handle any undefined variable/function errors, `NameError`s in Python, during runtime.

One approach is to modify a code interpreter (e.g. the Python interpreter) to automatically intercept these errors and continue execution. Continuing execution between parent and child LLM-REPLs can be synced by running different threads or co-routines with various concurrency synchronization primitives.

We implemented LLM-REPL without modifying the interpreter or running any concurrency routines by saving execution state for each REPL loop iteration and re-running code when LLM-REPLs are spawned. When a `NameError` is encountered, we create an LLM-REPL and add it as a variable to the last saved execution state. Finally, we re-run the code from this last execution state. However, an issue with this approach is multiple context sensitive functions (other LLM-REPLs, `act`) can be called again after an interrupt. For example, consider the execution trace from Figure 2. After the `NameError(count_even)`, `act(i*2+1)` will be called again after re-running the code. Avoiding incorrect and duplicate computation can avoided by keeping a function call counter for each LLM-REPL, then when a context-sensitive function called more than once, the return value is returned using a cache. These call counters must be updated and synced between parent-child LLM-REPLs after each context-passing function.

Another implementation detail is how to differentiate between a code statement that intends to spawn a new child LLM-REPL, and a code statement that makes a mistake by forgetting to define a variable. For example, consider the following lines of code:

1. `[id for id, price in id_to_price.items() for price < max_price]`
2. `[id for id, price in id_to_price.items() for price < get_max_price()]`

In the case that `max_price` and `get_max_price` are undefined, both lines would raise a `NameError`. In both cases, in our framework, an LLM-REPL will be created and named `max_price` and `get_max_price` respectively. However, in the implementation of an LLM-REPL, we override *any non-function method* (such as comparators and string representations) to raise a custom error: `REPLNameError`. Because of this, the first line of code 1. will result in a `REPLNameError` raised where the LLM will be given feedback that it forgot to define the `max_price` variable. The second line of code 2. will jump to the `get_max_price()` LLM-REPL execution.

We have attached our implementation for LLM-REPLs with example test cases in the supplementary material.

`main()` REPL

```
1 Task: Count to 4.
2 >>> for i in range(2):
3 ...     act(i*2+1)
4 ...     count_even()
5 ...
```

`count_even()` REPL

```
1 Task: Count only evens to 4.
2 >>> for i in range(2):
3 ...     act((i+1)*2)
4 ...     answer(f'Counted {i*2}.')
5 ...
```

Actions	main() Trace	count_even() Trace
	$\ell_2, \ldots, \ell_5 \leftarrow$ LLMQuery()	
	for i in range(2):	
act(1)	act(i*2+1)	
	NameError(count_even)	
	create REPL: subtask ← LLMQuery()	
	add count_even REPL to variables	
	"rewind" state back to ℓ_4	
	count_even()	
		$\ell_2, \ldots, \ell_5 \leftarrow$ LLMQuery()
		for i in range(2):
act(2)		act((i+1)*2)
		answer('...')
act(3)	act(i*2+1)	
	count_even()	
act(4)		act((i+1)*2)

Figure 4: Another toy example of context passing that is possible in REPL-Plan, where context is "interleaved" between LLM-REPLs. In the toy example, the task is to count to 4. We show the generated code in the two code snippets above. In the code, main REPL could spawn another REPL to help it count only the even numbers. By passing context back and forth, we show how the final actions count to 4 in the REPL's execution trace in the bottom table.

A.3 GPT API Parameters

For GPT-3.5-turbo-instruct and GPT-4o-mini, we used the same sampling parameters across REPL-Plan and baselines. We used a temperature of 0.8, and defaults for other sampling parameters (no frequency penalty, logit bias, presence penalty, or top p).

A.4 Modifications to THREAD for GPT4o-mini

We found that running THREAD (Schroeder et al., 2024) on the author's codebase with newer OpenAI models such as GPT4o and GPT4o-mini tended to always cause infinite loops — after seeing an observation, the LLMs would always generate #START# tokens, which indicate the start of a prompt, causing the model to re-generate the actions and text before seeing the observation, and repeat this #START# generation. Under normal generation, the LLM should never generate these tokens.

We believe that the GPT4o series may not listen to literal commands to follow demonstrations exactly when viewing web demonstrations, compared to GPT3.5-instruct, which was used in the original paper for web evironments such as WebShop.

To correct this, we modified the algorithm THREAD as follows: for each LLM generation, $G(c+Y)$, if #START# is a substring, we prompt the LLM to re-generate the text. If this happens, still, we then prompt using the following "don't loop" prompt $L =$ 'Do NOT output #START#. Look at the examples for what to do next. If your sub-task is finished or failed, report the status of your task with:
print("<status here>")
#END#
Continue output:'

We re-query the LLM with $G(c+Y+L)$, then proceed using THREAD.

We find this change is able to improve THREAD from always looping on WebShop with GPT4o-mini, to a 21% success rate.

A.5 WebShop GPT4o-mini experiment details

For the WebShop with GPT4o-mini setting, we tested THREAD and REPL-Plan with both Top-3 and Top-20 strategies. For the Top-3, we used the same prompts as used for Top-3 with GPT-3.5 for both THREAD and REPL-Plan. We implemented a version of the Top-20 strategy for THREAD, however the method would enter a fail state by looping forever.

A.6 Comparison of Code-augmented LLM Planning Approaches

	REPL-Plan	THREAD[1]	AdaPlanner[2]
Code for Planning	✓	✓	✓
Interactive Coding	✓	✓	✗
Top-down Divide-and-conquer	✓	✓	✗
Code-error recovery	✓	✗	✗
Loops	✓	✗	✓
Subtask Function Re-usage	✓	✗	✗

Table 5: A detailed comparison of code-augmented LLM planning approaches. [1]Schroeder et al. (2024). [2]Sun et al. (2023).

A.7 Real-World Web Task Details

Environment. We constructed the Real-World Web Task environment by interacting with live websites on the browser. Observations are taken from the *accessibility tree* of the website. Each website HTML markup is represented in a browser as a tree of elements (e.g. links, images, buttons, etc.) called a DOM tree. The DOM tree is then converted to an accessibility tree by the browser, by adding text information to DOM nodes that may be useful for users of assistive technologies, such as screen readers. This also makes the accessibility tree readable for text-only observations given to LLMs.

Our environment specifically takes the accessibility tree from the browser, then strips unnecessary nodes from the tree, and adds unique element IDs to each interactive element. For example, the website `google.com` gives the following observation:

```
[6] [RootWebArea] [Google]
        [11] [link] [About]
        [12] [link] [Store]
        [20] [link] [Gmail]
        [22] [link] [Search for Images]
                Images
        [27] [button] [Google apps] [expanded: False]

        [31] [link] [Sign in]
        [323] [IframePresentational] []
        [281] [image] [Google]
        [46] [search] []
                [35] [combobox] [Search] [focused: True, autocomplete: both,
    hasPopup: listbox, required: False, expanded: False]
                [81] [button] [Search by voice]
                [87] [button] [Search by image]
                [212] [button] [Google Search]
                [213] [button] [I'm Feeling Lucky]
        [225] [contentinfo] []
                [228] [link] [Advertising]
                [229] [link] [Business]
                [230] [link] [How Search works]
                [232] [link] [Our third decade of climate action: join us]
                [235] [link] [Privacy]
                [236] [link] [Terms]
                [240] [button] [Settings] [hasPopup: menu, expanded: False]
                        [241] [generic] [] [hasPopup: menu]
```

A link in the above web page can be clicked with the following action: `click [11]` (to click on the `[About]` page).

The search bar can be used with: `type [46] [What is a REPL?]`.

These observations can be very long and difficult to parse. Note that most web pages are 10-20 times more tokens than the `google.com` webpage shown above.

Task Distribution. We tested the agents on distributions of tasks where the goal is to navigate one of 5 different E-commerce websites, and add item(s) that match a given description to the cart. We divided the tasks into a "simple" category, where the agent only needs to add an item of any description (e.g. add any laptop to the cart from `bestbuy.com`) and a "complex" category, where the agent must find as many items as it can that match a specific description (e.g. loop through each product page and add each printer to the cart that prints at least 12 pages per minute).

In each of these tasks, we tested each approach on each task for 3 trials, and averaged the score. An expert score was obtained by having one of the authors use a web browser to complete the task.

Simple tasks:

1. On bestbuy.com, search for laptops. Add any laptop to the cart.
2. On dell.com, search for desktop computers. Add any desktop computer to the cart.
3. On staples.com, search for monitors. Add any monitor to the cart.
4. On zappos.com, search for backpacks. Add any backpack to the cart.
5. On nordstrom.com, search for bath towels. Add any bath towel to the cart.

Complex tasks:

1. On bestbuy.com, search for printers. Loop through each search result and add every printer that is capable of printing at least 12 pages per minute to the cart.

2. On dell.com, search for keyboards. Loop through each search result and add every keyboard with cherry key switches to the cart.
3. On staples.com, search for shredders. Loop through each search result and add every shredder with at least 6 gallons of capacity to the cart.
4. On zappos.com, search for sunglasses. Loop through each search result and add every pair of sunglasses that is polarized to the cart.
5. On nordstrom.com, search for watches. Loop through each search result and add every watch that is swiss made to the cart.

Approaches. We tested REPL-Plan and the baselines ReACT and THREAD on the Real-World Web environment. For each approach, to help the LLM deal with long pages, we truncate un-necessary web page observations before querying the LLM. In each approach, we constructed a demonstration from one training task: On bestbuy.com, search for cell phones. Add cell phones with a rating of at least 4.5/5 to cart. The same search strategy and task decomposition was used for THREAD and REPL-Plan. However, as described in Appendix A.4, we modified THREAD to run with GPT4o-mini and our web environment.

A.8 WebShop Example Top-3 Trajectory

For brevity, we include a short Top-3 strategy WebShop trajectory/log file of running REPL-Plan. In the following, `##### ENTER REPL ######` and `##### EXIT REPL ######` indicate when context is switched to and from which REPL.

```
WebShop
Instruction:
i want a noise cancelling cosycost usb microphone, and price lower than 70.00 dollars
[Search]
##### ENTER REPL `_main` #####
>>> description = """noise cancelling cosycost usb microphone, and price lower than 70.00 dollars"""
>>> attributes = ["""noise cancelling""", """cosycost""", """usb microphone"""] # save the list of desired attributes besides price, escape any quotes
>>> search_query = generate_query(description) # get a search statement to find the item
Name error: generate_query. injecting with cached LLM func.
##### ENTER REPL `generate_query` #####
>>> description = get_args() # call this once to get the description of the target product so that you can generate search queries accordingly
>>> print(description) # print out the description so that you know what it is
noise cancelling cosycost usb microphone, and price lower than 70.00 dollars

>>> query = """noise cancelling cosycost usb microphone"""
>>> answer(query) # answer the query to return it.
##### EXITING REPL `generate_query`#####
>>> act(f'search[{search_query}]')
##### EXITING REPL `_main`#####
> search[noise cancelling cosycost usb microphone]

[Back to Search]
Page 1 (Total results: 3)
[Next >]
[B0972Q1T8T]
Cosycost USB Microphone,Condenser Computer PC Gaming Microphone for PS4/5 Laptop Windows Mac OS Android Phone,Noise Cancelling Instant Mute,Studio Mic for Voice,Music Recording,Podcasting,Streaming
$32.99
[B072L2D6LY]
Andrea Communications NC-255VM USB On-Ear Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
$34.59
[B071H84LTJ]
Andrea Communications NC-455VM USB Over-Ear Circumaural Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
```

```
$49.24
##### ENTER REPL `_main` #####
>>> valid_ids = filter_ids_by_price(description)
Name error: filter_ids_by_price. injecting with cached LLM func.
##### ENTER REPL `filter_ids_by_price` #####
>>> description = get_args() # call this once to get the description of the target product so that you can generate search queries accordingly
>>> print(description) # print out the target description so that you know what it is
noise cancelling cosycost usb microphone, and price lower than 70.00 dollars

>>> max_price = 70.00 # store the max price from the description
>>> print_page() # call this function to print out the list of products
>>> id_to_price = {} # record an id to price dictionary
>>> id_to_price['B0972Q1T8T'] = 32.99 # item B0972Q1T8T has a price of $32.99
>>> id_to_price['B072L2D6LY'] = 34.59 # item B072L2D6LY has a price of $34.59
>>> id_to_price['B071H84LTJ'] = 49.24 # item B071H84LTJ has a price of $49.24
>>> results = [id for id, price in id_to_price.items() if price < max_price]
>>> answer(results) # answer with the result of the ids that are below max_price
##### EXITING REPL `filter_ids_by_price`#####
>>> best_id = select_best_product(description, attributes, valid_ids)
Name error: select_best_product. injecting with cached LLM func.
##### ENTER REPL `select_best_product` #####
>>> description, attributes, candidates = get_args() # get the target description and product candidates
>>> scores = {}
>>> for product_id in candidates:
...     act(f'click[{product_id}]') # go to the product's page details
...     scores[product_id] = check_requirements(description, attributes)
...     act(f'click[< Prev]') # always perform this step to get back to the list of results
...
##### EXITING REPL `select_best_product`#####
##### EXITING REPL `_main`#####
> click[B0972Q1T8T]

[Back to Search]
[< Prev]
Cosycost USB Microphone,Condenser Computer PC Gaming Microphone for PS4/5 Laptop Windows Mac OS Android Phone,Noise Cancelling Instant Mute,Studio Mic for Voice,Music Recording,Podcasting,Streaming
Price: $32.99
Rating: N.A.
[Description]
[Features]
[Reviews]
[Attributes]
[Buy Now]
##### ENTER REPL `_main` #####
##### ENTER REPL `select_best_product` #####
Name error: check_requirements. injecting with cached LLM func.
##### ENTER REPL `check_requirements` #####
>>> target_description, requirements = get_args() # get the requirements and product_info
>>> print(target_description) # print the target_description so that have a better understanding about the requirements
noise cancelling cosycost usb microphone, and price lower than 70.00 dollars

>>> print(requirements) # print out the requirements
['noise cancelling', 'cosycost', 'usb microphone']

>>> print_page() # print the product page so we can check the requirements
>>> # I'll check each requirement one by one.
>>> # 1. the product has 'noise cancelling'? Yes, it is explicitly mentioned in the product description as 'Noise Cancelling'.
>>> # 2. the product has 'cosycost'? Yes, it is explicitly mentioned in the product name as 'Cosycost USB Microphone'.
>>> # 3. the product is a 'usb microphone'? Yes, it is explicitly mentioned in the product description as 'USB Microphone'.
>>> answer(3) # the number of yes to the questions above
```

```
##### EXITING REPL `check_requirements`#####
##### EXITING REPL `select_best_product`#####
##### EXITING REPL `_main`#####
> click[< Prev]

[Back to Search]
Page 1 (Total results: 3)
[Next >]
[B0972Q1T8T]
Cosycost USB Microphone,Condenser Computer PC Gaming Microphone for PS4/5 Laptop Windows Mac OS Android Phone,Noise Cancelling Instant Mute,Studio Mic for Voice,Music Recording,Podcasting,Streaming
$32.99
[B072L2D6LY]
Andrea Communications NC-255VM USB On-Ear Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
$34.59
[B071H84LTJ]
Andrea Communications NC-455VM USB Over-Ear Circumaural Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
$49.24
##### ENTER REPL `_main` #####
##### ENTER REPL `select_best_product` #####
##### EXITING REPL `select_best_product`#####
##### EXITING REPL `_main`#####
> click[B072L2D6LY]

[Back to Search]
[< Prev]
Andrea Communications NC-255VM USB On-Ear Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
Price: $34.59
Rating: N.A.
[Description]
[Features]
[Reviews]
[Attributes]
[Buy Now]
##### ENTER REPL `_main` #####
##### ENTER REPL `select_best_product` #####
##### ENTER REPL `check_requirements` #####
>>> target_description, requirements = get_args() # get the requirements and product_info
>>> print(target_description) # print the target_description so that have a better understanding about the requirements
noise cancelling cosycost usb microphone, and price lower than 70.00 dollars

>>> print(requirements) # print out the requirements
['noise cancelling', 'cosycost', 'usb microphone']

>>> print_page() # print the product page so we can check the requirements
>>> # I'll check each requirement one by one.
>>> # 1. the product has 'noise cancelling' feature? Yes, it mentions 'Noise-Canceling Microphone' which qualifies as meeting the requirement.
>>> # 2. the product has 'cosycost' brand? No, it does not mention 'cosycost' anywhere in the product info.
>>> # 3. the product is a 'usb microphone'? No, it is described as a 'USB On-Ear Stereo USB Computer Headset' which does not specifically mention 'usb microphone'.
>>> answer(1) # Only one requirement is met based on the checks above.
##### EXITING REPL `check_requirements`#####
##### EXITING REPL `select_best_product`#####
##### EXITING REPL `_main`#####
> click[< Prev]

[Back to Search]
Page 1 (Total results: 3)
[Next >]
[B0972Q1T8T]
```

```
Cosycost USB Microphone,Condenser Computer PC Gaming Microphone for PS4/5 Laptop Windows Mac OS Android Phone,Noise Cancelling Instant Mute,Studio Mic for Voice,Music Recording,Podcasting,Streaming
$32.99
[B072L2D6LY]
Andrea Communications NC-255VM USB On-Ear Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
$34.59
[B071H84LTJ]
Andrea Communications NC-455VM USB Over-Ear Circumaural Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
$49.24
##### ENTER REPL `_main` #####
##### ENTER REPL `select_best_product` #####
##### EXITING REPL `select_best_product`#####
##### EXITING REPL `_main`#####
> click[B071H84LTJ]

[Back to Search]
[< Prev]
Andrea Communications NC-455VM USB Over-Ear Circumaural Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
Price: $49.24
Rating: N.A.
[Description]
[Features]
[Reviews]
[Attributes]
[Buy Now]
##### ENTER REPL `_main` #####
##### ENTER REPL `select_best_product` #####
##### ENTER REPL `check_requirements` #####
>>> target_description, requirements = get_args() # get the requirements and product_info
>>> print(target_description) # print the target_description so that have a better understanding about the requirements
noise cancelling cosycost usb microphone, and price lower than 70.00 dollars

>>> print(requirements) # print out the requirements
['noise cancelling', 'cosycost', 'usb microphone']

>>> print_page() # print the product page so we can check the requirements
>>> # I'll check each requirement one by one.
>>> # 1. the product has 'noise cancelling' feature? Yes, it mentions 'Noise-Canceling Microphone' which qualifies as meeting the requirement.
>>> # 2. the product has 'cosycost' brand? No, it does not mention 'cosycost' anywhere in the product info.
>>> # 3. the product is a 'usb microphone'? No, it describes a 'USB Over-Ear Circumaural Stereo USB Computer Headset' which does not qualify as a standalone 'usb microphone'.
>>> answer(1) # Only one requirement is met, which is 'noise cancelling'.
##### EXITING REPL `check_requirements`#####
##### EXITING REPL `select_best_product`#####
##### EXITING REPL `_main`#####
> click[< Prev]

[Back to Search]
Page 1 (Total results: 3)
[Next >]
[B0972Q1T8T]
Cosycost USB Microphone,Condenser Computer PC Gaming Microphone for PS4/5 Laptop Windows Mac OS Android Phone,Noise Cancelling Instant Mute,Studio Mic for Voice,Music Recording,Podcasting,Streaming
$32.99
[B072L2D6LY]
Andrea Communications NC-255VM USB On-Ear Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
$34.59
[B071H84LTJ]
```

```
Andrea Communications NC-455VM USB Over-Ear Circumaural Stereo USB Computer Headset with Noise-Canceling Microphone, in-Line Volume/Mute Controls, and Plug
$49.24
##### ENTER REPL `_main` #####
##### ENTER REPL `select_best_product` #####
>>> products_with_max_score = [product_id for product_id, estimated_score in scores.items() if estimated_score == max(scores.values())]
>>> answer(products_with_max_score[0]) # return the id of the first best candidate
##### EXITING REPL `select_best_product`#####
>>> buy_best_product(best_id, attributes)
Name error: buy_best_product. injecting with cached LLM func.
##### ENTER REPL `buy_best_product` #####
>>> product_asin, attributes = get_args()
>>> act(f'click[{product_asin}]')
##### EXITING REPL `buy_best_product`#####
##### EXITING REPL `_main`#####
> click[B0972Q1T8T]

[Back to Search]
[< Prev]
Cosycost USB Microphone,Condenser Computer PC Gaming Microphone for PS4/5 Laptop Windows Mac OS Android Phone,Noise Cancelling Instant Mute,Studio Mic for Voice,Music Recording,Podcasting,Streaming
Price: $32.99
Rating: N.A.
[Description]
[Features]
[Reviews]
[Attributes]
[Buy Now]
##### ENTER REPL `_main` #####
##### ENTER REPL `buy_best_product` #####
>>> select_options(attributes) # select the item options based on the attributes before buying
Name error: select_options. injecting with cached LLM func.
##### ENTER REPL `select_options` #####
>>> attributes = get_args()
>>> get_page_options() # return the list of page options (if any)
Name error: get_page_options. injecting with cached LLM func.
##### ENTER REPL `get_page_options` #####
>>> print_page() # print out the current observation to see if there are any buying variations to select before buying.
>>> variations = [] # save variations (if any) in a list
>>> answer("no options found.")
##### EXITING REPL `get_page_options`#####
'no options found.'
>>> print(attributes) # print out the target attributes to confirm there are no buying variations to select.
['noise cancelling', 'cosycost', 'usb microphone']

>>> answer('success.')
##### EXITING REPL `select_options`#####
'success.'
>>> act('click[Buy Now]') # Buy the item to complete the task.
##### EXITING REPL `buy_best_product`#####
##### EXITING REPL `_main`#####
> click[Buy Now]
```

A.9 WebShop REPL-Plan Prompt Samples

```
# Top 3 strategy

Your task is to: Navigate a shopping website to purchase an item matching the following request: i would like a 3 ounce bottle of bright citrus deodorant for sensitive skin, and price lower than 50.00 dollars
>>> description = """3 ounce bottle of bright citrus deodorant for sensitive skin, and price lower than 50.00 dollars"""
>>> attributes = ["""3 ounce bottle""", """bright citrus""", """sensitive skin"""] # save the list of desired attributes besides price, escape any quotes
```

```
>>> search_query = generate_query(description) # get a search statement to find the item
>>> act(f'search[{search_query}]')
>>> valid_ids = filter_ids_by_price(description)
>>> best_id = select_best_product(description, attributes, valid_ids)
>>> buy_best_product(best_id, attributes)
'success.'
>>> answer('done.')



# generate_query
Your task is to: Generate a good search query for the given arg `description`. Answer the result with `answer(query)`. Make sure any quotes in the search string are escaped.
>>> description = get_args() # call this once to get the description of the target product so that you can generate search queries accordingly
>>> print(description) #print out the description so that you know what it is
3 ounce bottle of bright citrus deodorant for sensitive skin, and price lower than 50.00 dollars
>>> query = """3 ounce bright citrus deodorant sensitive skin"""
>>> answer(query) # answer the query to return it.


# select_best_product
Your task is to: Given the description, attributes, and a few product candidates, determine which of them fits the description the best (meets the most requirements). answer best product's id with `answer('ID')`.
>>> description, attributes, candidates = get_args() # get the target_description anget_product_idsd candidates
>>> scores = {}
>>> for product_id in candidates:
...     act(f'click[{product_id}]') # go to the product's page details
...     scores[product_id] = check_requirements(description, attributes)
...     act(f'click[< Prev]') # always perform this step to get back to the list of results
...
>>> products_with_max_score = [product_id for product_id, estimated_score in scores.items() if estimated_score == max(scores.values())]
>>> answer(products_with_max_score[0]) # return the id of the first best candidates

# check_requirements
Your task is to: Given a list of requirements, and a task description, for each requirement, check if the product meets the requirement or not, then answer how many requirements it meets.
>>> target_description, requirements = get_args() # get the requirements and product_info
>>> print(target_description) # print the target_description so that have a better understanding about the requirements
3 ounce bottle of bright citrus deodorant for sensitive skin, and price lower than 50.00 dollars
>>> print(requirements) # print out the requirements
['deodorant', '3 ounce bottle', 'bright citrus', 'sensitive skin', '<50.00 dollars']
>>> print_page() # print the product page so we can check the requirements
[Back to Search]
[< Prev]
scent [assorted scents][bright citrus][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, Natural and Safe for Sensitive Skin, Pregnancy and Breastfeeding, Contains Organic Calendula 3-Ounce
Price: $10.99
Rating: N.A.
[Description]
[Features]
[Reviews]
[Attributes]
[Buy Now] (You must select buying variation for scent, color before buying this product)
```

```
Selected Buying Variation Options: scent: None, size: None
>>> # I'll check each requirement one by one.
>>> # 2. the product has '3 ounce bottle' size? Yes, it meets the requirement because it has [3 ounce] size option.
>>> # 3. the product has 'bright citrus' scent? Yes, it has the exact option [bright citrus] listed.
>>> # 4. the product is good for 'sensitive skin'?, Yes, it is described as 'Gentle on Sensitive Skin'.
>>> answer(3) #the number of yes to the questions above


# buy_best_product
Your task is to: Given a product_asin, click on the product asin, select the required buying variations that the attributes requires, and buy it.
>>> product_asin, attributes = get_args()
>>> act(f'click[{product_asin}]')
>>> select_options(attributes) # select the item options based on the target_description before buying
'success.'
>>> act('click[Buy Now]') # Buy the item to complete the task.
>>> answer('success.')
```

A.10 WebShop Ablation Bug correction Samples

We show a few hand-picked instances where an LLM is able recover from bug(s) in the demonstration.

```
>>> description = "gingko light and 20"x20" pillow cover that is hand painted, and price lower than 50.00 dollars" # there is a syntax error in the description, the quotes around "20"x20"" need to be escaped
SyntaxError('invalid syntax', ('<unknown>', 1, 36, 'description = "gingko light and 20"x20" pillow cover that is hand painted, and price lower than 50.00 dollars" # there is a syntax error in the description, the quotes around "20"x20"" need to be escaped\n', 1, 39))
>>> description = 'gingko light and 20"x20" pillow cover that is hand painted, and price lower than 50.00 dollars' # corrected syntax error
>>> attributes = ["gingko light", "20\"x20\"", "hand painted"] # escaped quotes in attributes list
>>> search_query = generate_query(description) # get a search statement to find the item


>>> all_requirements = ["car subwoofer", "high speed", "dual style", "12" power amplifier", "<200.00 dollars"] # you must put the item category in the first place, each term should be as short/broken down as possible
SyntaxError('unterminated string literal (detected at line 1)', ('<unknown>', 1, 105, 'all_requirements = ["car subwoofer", "high speed", "dual style", "12" power amplifier", "<200.00 dollars"] # you must put the item c
ategory in the first place, each term should be as short/broken down as possible, 1, 105))
>>> all_requirements = ["car subwoofer", "high speed", "dual style", "12\" power amplifier", "<200.00 dollars"] # corrected the syntax error by escaping the double quotes


>>> results = [id for id, price in id_to_price.items() if price < max_price]
REPLNameError("name 'max_price' not defined.")
>>> max_price = 30.00 # set the maximum price to $30.00
>>> results = [id for id, price in id_to_price.items() if price < max_price] # create a list of ids that have a price lower than the maximum price
>>> answer(results) # answer with the result of the ids that are below max_price
```

A.11 WebShop Ablation Zero-shot Subtask-REPL Analysis

In this section, we provide details on the Zero-shot Subtask-REPL ablation experiment. Specifically, in this experiment, we modify the expert demonstration for the WebShop environment by removing the `check_description` LLM-REPL. In WebShop, REPL-Plan loops through each product page, and calls `check_description` to obtain an integer score of the number of matching attributes of the product from the given item description. To execute this LLM-REPL correctly, the agent must write code that succesfully (1) prints the target description and requirements from the `get_args` function, (2) prints the current page (3) calls the `answer` function with a count of the number of matching attributes.

Note that to infer the subtask description and LLM-REPL code, the agent is given demonstrations of out-of-task LLM-REPLs from the WebShop domain.

We include the code from the expert-demonstration as a reference for the intended behavior of the LLM-REPL. The code is removed for this experiment and must be inferred.

```
# Expert written task description and code demonstration.
Your task is to: Given a list of requirements, and a task description, for each
    requirement, check if the product meets the requirement or not, then answer how
     many requirements it meets.
>>> target_description, requirements = get_args() # get the requirements and
    product_info
>>> print(target_description) # print the target_description so that have a better
    understanding about the requirements
3 ounce bottle of bright citrus deodorant for sensitive skin, and price lower than
    50.00 dollars
>>> print(requirements) # print out the requirements
>>> print_page() # print the product page so we can check the requirements
>>> # I'll check each requirement one by one.
>>> # 2. the product has '3 ounce bottle' size? Yes, it meets the requirement
    because it has [3 ounce] size option.
>>> # 3. the product has 'bright citrus' scent? Yes, it has the exact option [
    bright citrus] listed.
>>> # 4. the product is good for 'sensitive skin'?, Yes, it is described as 'Gentle
     on Sensitive Skin'.
>>> answer(3) #the number of yes to the questions above
```

Successful Code Samples We include the following code samples from the agent with GPT3.5-instruct where it successfully inferred the correct code.

```
# Zero-shot Subtask-REPL SUCCESSES
# Example 1
Your task is to: (1) Print out the description and attributes with `get_args()`.
    (2) Go to the product page by clicking on the product. (3) Check the current
    page for matching attributes. (4) Return the number of matching attributes with
     `answer(INTEGER)`
>>> description = get_args() # call this once to get the description of the target
    product so that you can generate search queries accordingly
>>> attributes = get_args() # call this once to get the attributes of the target
    product so that you can check for matching attributes
>>> print(description) # print out the description so that you know what it is
('long clip-in hair extension which is natural looking, and price lower than 40.00
    dollars', ['long', 'clip-in', 'natural looking'])
>>> print(attributes) # print out the attributes so that you know what they are
('long clip-in hair extension which is natural looking, and price lower than 40.00
    dollars', ['long', 'clip-in', 'natural looking'])
>>> print_page() # print out the current observation
>>> # With the page printed, I should manually analyze the page to fulfill my goal
>>> # The only attribute that matches is "long"
>>> answer(1) # return the number of matching attributes, which is 1


# Example 2
Your task is to: (1) Print out the description and attributes with `get_args()`.
    (2) Go to the product page by clicking on the product. (3) Check how many
    attributes match the description on the current page. (4) Return the number of
    matching attributes with `answer(INTEGER)`
>>> description = get_args() # call this once to get the description of the target
    product so that you can generate search queries accordingly
>>> print(description) #print out the description so that you know what it is
('noise cancelling cosycost usb microphone, and price lower than 70.00 dollars', ['
    noise cancelling', 'cosycost', 'usb microphone'])
>>> attributes = get_args()[1] # retrieve the attributes from the description
>>> print(attributes) # print out the attributes to see what they are
['noise cancelling', 'cosycost', 'usb microphone']
>>> print_page() # print out the current observation
>>> # Now I need to go to the product page by clicking on the product
```

```
>>> # Once on the product page, I need to check how many attributes match the description
>>> # In this case, all three attributes match: noise cancelling, cosycost, and usb microphone
>>> matching_attributes = len(attributes) # use the len() function to get the number of matching attributes
>>> answer(matching_attributes) # return the number of matching attributes



# Example 3: The agent mis-understands and tries to click on the product, but recovers and completes the task
Your task is to: (1) Print out the description with `get_args()`. (2) print out the current web page with `print_page()`. (3) Look at the webpage for the attributes. (4) Answer with the number of attributes that match the description with `answer(INTEGER)`
>>> description = get_args() # call this once to get the description of the target product so that you can generate search queries accordingly
>>> attributes = get_args() # call this once to get the attributes of the target product so that you can compare them with the description
>>> print(description) # print out the description so that you know what it is
('high speed 3 foot red usb cable, and price lower than 40.00 dollars', ['high speed', '3 foot', 'red'])

>>> print(attributes) # print out the attributes so that you know what they are
('high speed 3 foot red usb cable, and price lower than 40.00 dollars', ['high speed', '3 foot', 'red'])

>>> act(f'click[{product_id}]') # go to the product page by clicking on the product ID
REPLNameError("name 'product_id' not defined.")
>>> # I cannot complete this task without knowing the product ID, so I will need to ask for it or define it myself
>>> # Assuming the product ID is defined as "12345"
>>> print_page() # print out the current observation
>>> # With the page printed, I should manually analyze the page to fulfill my goal
>>> # There is one attribute that matches the description: "red"
>>> matching_attributes = 1
>>> answer(matching_attributes)
```

Failed Code Samples We include the following code samples from the agent with GPT3.5-instruct where it incorrectly infers the subtask or the code. We generally find that the LLMs write code that contains the "right intentions" for solving the task, but critically fail by containing bugs or crash the system in some way.

```
# Zero-shot Subtask-REPL FAILS
# Example 1: this fails as it mis-understands how to parse and string match for attributes.
Your task is to: (1) Print out the description and attributes with `get_args()`. (2) Go to the product page by clicking on the product. (3) Check the current page for matching attributes. (4) Return the number of matching attributes with `answer(INTEGER)`
>>> description = get_args() # call this once to get the description of the target product so that you can generate search queries accordingly
>>> attributes = get_args() # call this once to get the attributes of the target product so that you can check for matching attributes
>>> print(description) # print out the description so that you know what it is
('long clip-in hair extension which is natural looking, and price lower than 40.00 dollars', ['long', 'clip-in', 'natural looking'])

>>> print(attributes) # print out the attributes so that you know what they are
('long clip-in hair extension which is natural looking, and price lower than 40.00 dollars', ['long', 'clip-in', 'natural looking'])

>>> print_page() # print out the current observation
>>> # With the page printed, I should manually analyze the page to fulfill my goal
>>> # The attributes on the page are "long", "curly", "black", "golden", "straight", "african", "hair", "ladies", "chemical fiber", "silk", "corn perm", "women"
```

```
>>> # I will check for matching attributes by comparing them to the attributes retrieved from get_args()
>>> matching_attributes = 0
>>> for attribute in attributes[1]:
...     if attribute in ["long", "curly", "black", "golden", "straight", "african", "hair", "ladies", "chemical fiber", "silk", "corn perm", "women"]:
...         matching_attributes += 1
...
>>> answer(matching_attributes)


# Example 2: this fails as the task description is mistakenly inferred, then causes the agent to enter a looping fail state
Your task is to: (1) Print out the description and attributes with `get_args()`. (2) Use the act function to click on each product and go to its page details. (3) Use the check_current_page_requirements function to check how many attributes match the description on the current page. (4) Use a list comprehension to find the products with the maximum score and return a list of those products with `answer(LIST)
>>> # I will now create a list of products with the maximum score by looping through the scores dictionary with a list comprehension
>>> products = [product for product, score in scores.items() if score == max(scores.values())]
AttributeError("'LLMREPL' object has no attribute 'items'")
>>> # I will then answer with the list of products with the maximum score
>>> answer(products)
REPLNameError("name 'products' not defined.")
>>> # now its your turn
>>> # I will now create a list of products with the maximum score by looping through the scores dictionary with a list comprehension
>>> products = [product for product, score in scores.items() if score == max(scores.values())]
AttributeError("'LLMREPL' object has no attribute 'items'")
...


# Example 3: this solution fails to match the attributes to the page and instead counts with string matching the description
Your task is to: (1) Print out the description and attributes with `get_args()`. (2) Click on each product candidate to check its details. (3) Count how many attributes match the description and attributes provided, and return that count as an integer.
>>> description, attributes = get_args() # call this to get both description and attributes for analysis
>>> print(description, attributes) # print out the description and attributes for analysis
space-saving ottoman bench to match my blue living room, pick that one that's 100x45x45cm, and price lower than 430.00 dollars ['space-saving', 'blue living room', '100x45x45cm']
>>> # Now that I have the description and attributes, I need to analyze the product candidates by clicking on them and checking their details.
>>> print_page() # print out the current observation to analyze product candidates
>>> # I need to analyze the current page for matching attributes and description
>>> matching_count = 0 # Initialize a counter for matching attributes
>>> if "space-saving" in description and "blue living room" in description and "100x45x45cm" in attributes: matching_count += 1
>>> if "space-saving" in attributes: matching_count += 1
>>> if "blue living room" in attributes: matching_count += 1
>>> if "100x45x45cm" in attributes: matching_count += 1
>>> answer(matching_count)
```