Statler: State-Maintaining Language Models for Embodied Reasoning

Takuma Yoneda*[1], Jiading Fang*[1], Peng Li*[2], Huanyu Zhang*[3], Tianchong Jiang[3], Shengjie Lin[1], Ben Picker[3], David Yunis[1], Hongyuan Mei[1], and Matthew R. Walter[1]

[1]Toyota Technological Institute at Chicago
{takuma,fjd,slin,dyunis,hongyuan,mwalter}@ttic.edu
[2]Fudan University
lip21@m.fudan.edu.cn
[3]University of Chicago
{huanyu,tianchongj,bpicker}@uchicago.edu

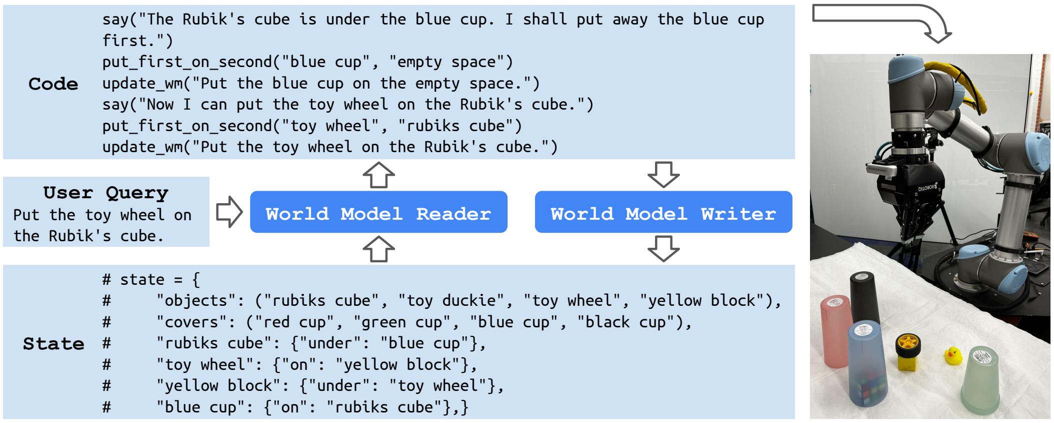
Abstract: Large language models (LLMs) provide a promising tool that enable robots to perform complex robot reasoning tasks. However, the limited context window of contemporary LLMs makes reasoning over long time horizons difficult. Embodied tasks such as those that one might expect a household robot to perform typically require that the planner consider information acquired a long time ago (e.g., properties of the many objects that the robot previously encountered in the environment). Attempts to capture the world state using an LLM's implicit internal representation is complicated by the paucity of task- and environment-relevant information available in a robot's action history, while methods that rely on the ability to convey information via the prompt to the LLM are subject to its limited context window. In this paper, we propose Statler, a framework that endows LLMs with an explicit representation of the world state as a form of "memory" that is maintained over time. Integral to Statler is its use of two instances of general LLMs—a world-model reader and a world-model writer—that interface with and maintain the world state. By providing access to this world state "memory", Statler improves the ability of existing LLMs to reason over longer time horizons without the constraint of context length. We evaluate the effectiveness of our approach on three simulated table-top manipulation domains and a real robot domain, and show that it improves the state-of-the-art in LLM-based robot reasoning. Project website: https://statler-lm.github.io/.

Keywords: Large language models, Long-horizon planning, World state model

1 Introduction

Large language models (LLMs) are capable of generating intricate free-form text and complex code with an impressive level of proficiency [1, 2, 3]. Recently, researchers have shown that the success of LLMs extends to robotics domains, where the capacity for LLMs to perform complex reasoning using language enables robots to perform tasks that require sophisticated planning and language understanding [4, 5, 6]. These methods either rely solely on the implicit in-context memory that is internal to the LLM [5] or they augment LLMs with scene information extracted from an ego-centric image captured at the current time step [4]. Both approaches have proven effective for difficult embodied reasoning tasks, however they struggle when faced with planning tasks that require planning over long time horizons, due to the limited context window of contemporary LLMs. Although there have been recent efforts to enlarge the context window of LLMs [7], the size of the

[*] Equal contribution.



```
say("The Rubik's cube is under the blue cup. I shall put away the blue cup
first.")
put_first_on_second("blue cup", "empty space")
update_wm("Put the blue cup on the empty space.")
say("Now I can put the toy wheel on the Rubik's cube.")
put_first_on_second("toy wheel", "rubiks cube")
update_wm("Put the toy wheel on the Rubik's cube.")
```

Put the toy wheel on the Rubik's cube.

```
# state = {
#     "objects": ("rubiks cube", "toy duckie", "toy wheel", "yellow block"),
#     "covers": ("red cup", "green cup", "blue cup", "black cup"),
#     "rubiks cube": {"under": "blue cup"},
#     "toy wheel": {"on": "yellow block"},
#     "yellow block": {"under": "toy wheel"},
#     "blue cup": {"on": "rubiks cube"},}
```

Figure 1: Our Statler framework enables robots to carry out complex tasks specified in natural language that require reasoning over long time horizons. Integral to our model are its world model writer and world model reader, two instances of general LLMs that are responsible for maintaining the explicit world state and generating code that enables the robot to carry out the task.

context window remains fundamentally bounded. Further, providing the model with long-range context improves prediction accuracy only on on a small number of tokens—LLMs struggle to exploit information conveyed in long-term context beyond what can be directly copied [8]. Meanwhile, reliance on the robot's current ego-centric view prohibits the language model from reasoning over aspects of the scene that are not directly observable, e.g., the fruit located in the (closed) kitchen refrigerator or an object in a room that the robot previously visited.

In this paper, we propose Statler (STATe-maintaining Language models for Embodied Reasoning), a framework that maintains an external world model as explicit memory to improve the long-term reasoning capabilities of LLMs for robot planning. Integral to our approach, as shown in Figure 1, it maintains and interfaces with this world model over time using two instances of general LLMs—a **world-model reader** and a **world-model writer**. The world-model reader interfaces with the world model to generate code that answers user queries. The world-model writer is responsible for predicting the next world state based on the current world state and a query given by the reader. We employ a structured representation of the world state, which has been found to improve the performance of LLMs [9, 10], particularly when the output is also structured, and has the advantage of being human-readable and concise for efficient processing. Note that while we individually tailor each world model's design to its general task type (see Prompts 12, 8, 7, and 9), the design is highly flexible because the reader and writer are both LLMs and are instructed with in-context-learning to understand how to parse and manipulate the world model. This is in contrast to domain-specific formal languages [11], where the designs are fixed and parsing and writing requires that specific rules be followed.

We evaluate Statler on a series of simulated and real-world robot manipulation domains. Experimental results demonstrate that Statler improves the long-term embodied reasoning capabilities of LLMs and that it outperforms the current state-of-the-art [5].

2 Motivational Example

As a demonstration of the challenges to temporal reasoning with LLMs, we consider a *three-cups-and-a-ball* version of the classic shell game. In this game, three visually identical cups are placed upside down on a table with a ball hidden under one of the cups. At the start, the player knows under which of the three cups the ball lies. In each of the subsequent K rounds, the dealer swaps the position of two randomly selected cups. After the K rounds, the player is asked which of the three

```
1 # Initial state
2 cups = [False, True, False]
3 Swapping cup 1 with cup 2
4 Swapping cup 0 with cup 2
5 Swapping cup 1 with cup 2
6 cups = [True, False, False]
```

Prompt 1: The prompt and desired output of a vanilla LLM.

```
1 # Initial state
2 cups = [False, True, False]
3 Swapping cup 1 with cup 2
4 Swapping cup 0 with cup 2
5 Swapping cup 1 with cup 2
6 cups = [False, False, True]
7 cups = [True, False, False]
8 cups = [True, False, False]
```

Prompt 2: The prompt and desired output of an LLM w/ CoT.

```
1 # Initial state
2 cups = [False, True, False]
3 Swapping cup 1 with cup 2
4 cups = [False, False, True]
5 Swapping cup 0 with cup 2
6 cups = [True, False, False]
7 Swapping cup 1 with cup 2
8 cups = [True, False, False]
```

Prompt 3: The prompt and desired output of an LLM w/ state.

cups contains the ball. Because the cups are visually indistinguishable, the player must keep track the ball's location as the cups are swapped in order to successfully identify its final location.

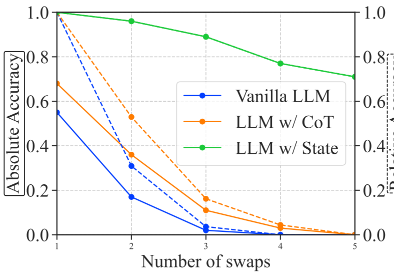


Figure 2: The accuracies of different methods for different numbers of swaps in the three-cups-and-a-ball shell game. LLM w/ State is a simplified version of our proposed Statler framework. For each method, the solid line shows how its accuracy $a(n)$ changes with the number of swaps n. The dashed line is the *relative* accuracy: $r(n) = a(n)/a(1)$. Intuitively, it measures how fast the performance decreases from a *hypothetically perfect* one-swap performance. Note that LLM w/ State indeed achieves $a(1) = 100\%$.

We simulate this three-cups-and-a-ball game using text as the interface. Prompt 1 presents the setup of the game. In Line 2, the Boolean value indicates the location of the ball and the subsequent lines describe the sequence of dealer swaps. After providing the LLM with multiple in-context learning examples prior to the prompt, the model is then asked to identify the location of the ball by generating the list highlighted in green after the K swaps.

We evaluate three different approaches that attempt to solve this task: a vanilla LLM, an LLM with chain-of-thought (CoT) [12], and a state-maintaining LLM, a simplified version of our Statler model. The vanilla LLM (see Prompt 1) provides only the final location of the ball at the end of the game given the initial location and sequence of swaps. The LLM with CoT (see Prompt 2) generates the sequence of ball positions after the final swapping action. This triggers the model to reason over the state transitions (i.e., changes in the cup positions) that can help to identify the final location of the ball. The state-maintaining LLM (see Prompt 3) stores and updates a state representation at every step. In contrast to the other models, the state-maintaining LLM processes each query step-by-step conditioned on the previous (generated) state representation, and then updates the representation.

We evaluate the accuracy with which these three models predict the location of the ball for different numbers of dealer swaps. We use the `text-davinci-003` version of GPT-3 as our LLM using the OpenAI API.[1] We prompt the LLM with 30 demonstration examples with a randomized number of swaps, and one final prompt for each episode. We evaluate the three models using 100 episodes, each of which involves querying the model for the location of the ball after every dealer swap. We terminate the episode if the response to the query is incorrect.

Figure 2 visualizes the average absolute accuracy of each model as well the accuracy relative to the model's one-swap accuracy. As we increase the number of swaps, the absolute accuracy of the vanilla LLM drops precipitously, reaching a near-zero value after only three swaps. This behavior is consistent with existing work that highlights the difficulty of maintaining the world state implicitly in LLMs [13, 14]. The LLM with CoT performs slightly better after one swap, but also experiences a pronounced decrease in absolute and relative accuracy. In contrast, the state-maintaining model consistently achieves higher absolute accuracy. More importantly, the relative accuracy of the state-maintaining model decreases far more gradually than the other methods, retaining more than 75% (absolute and relative) accuracy after five rounds of swaps.

[1] https://openai.com/api

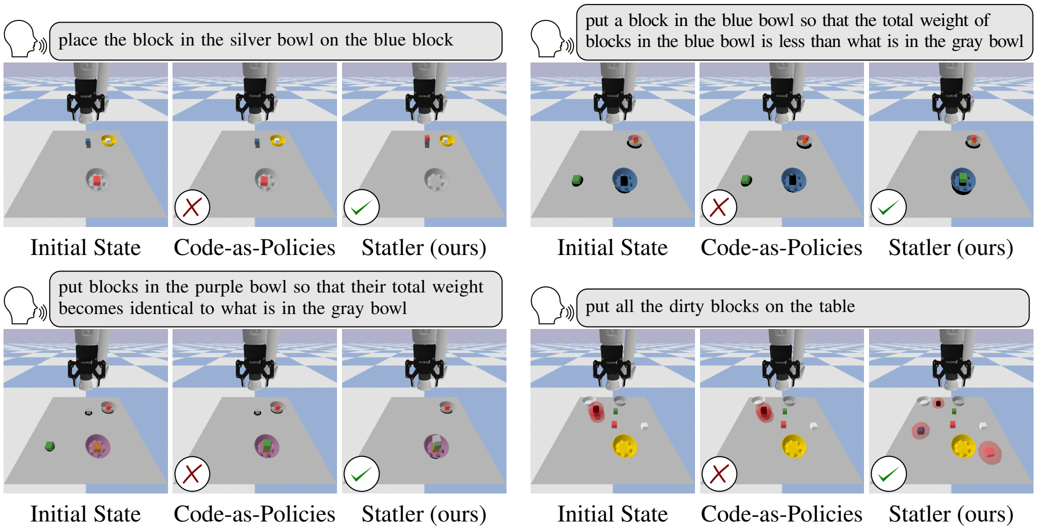

place the block in the silver bowl on the blue block
put a block in the blue bowl so that the total weight of blocks in the blue bowl is less than what is in the gray bowl
Initial State
Code-as-Policies
Statler (ours)
Initial State
Code-as-Policies
Statler (ours)
put blocks in the purple bowl so that their total weight becomes identical to what is in the gray bowl
put all the dirty blocks on the table
Initial State
Code-as-Policies
Statler (ours)
Initial State
Code-as-Policies
Statler (ours)

Figure 3: Examples of simulations that show the result of executing different natural language instructions using a vanilla LLM and our state-maintaining LLM.

Next, we present our full method (Statler)—a generalized version of this simple state model—and demonstrate its ability to produce plans in the context of more realistic scenarios that require reasoning with significantly greater complexity.

3 Method

As exemplified in Section 2, the key to our approach is to allow the LLM to describe the next state while responding to each user query. The motivating example is simple in that the next state description *is* the response. Instead, we consider a more challenging and arguably more realistic scenario, such as manipulating objects on a table as depicted in Figure 4. In this setting, there is a significant burden on the LLM to track the state updates as well as generate responses. Inspired by the concept of modularity, we propose to *split* the burden across multiple different prompted LLMs. Precisely, we maintain a separate prompt that includes instructions and demonstrations for each subtask (state tracking or query responding) and then use the prompt to elicit an LLM to perform the particular subtask. As we will discuss shortly, our framework includes **world-model reader** that responds to the user query and a **world-model writer** that is responsible for updating the state representation. Our framework, also shown in Figure 1 does not pose any limitation on what domain it can be applied to, or how many number of subtasks there are. We note that our approach can be seen as an extension to Code-as-Policies, where the state-managing mechanism is additionally embedded without affecting the fundamental capability of Code-as-Policies (i.e., hierarchical code generation).

To give a better idea of how the world-model reader and writer operate, we show example prompts and what each model is expected to generate. Prompt 4 is an example of the input passed to the world-model reader. Given a user query "Put the cyan block on the yellow block" and the current state representation (Lines 1–12), The world-model reader is expected to generate the code that responds to the query, taking into account the current state. The expected code to be generated is highlighted in green. After generating the code, our model executes it to complete the query. When the state needs to be updated, the generated code contains an `update_wm` function, which triggers the world-model writer with the query specified in its argument. In Prompt 5, we show the corresponding example for the world-model writer. Similar to the world-model reader, we prepend the current state representation before the user query and the model generates the updated state representation

```
# state = {
#     "objects": ["cyan block", "yellow block", "brown block", "purple block", "blue block", "green bowl", "red bowl", "disinfector"],
#     "relations": [],
#     "disinfector": {"contains": []},
#     "cyan block": {"is": ["dirty"]},
#     "yellow block": {"is": ["clean"]},
#     "brown block": {"is": ["clean"]},
#     "purple block": {"is": ["dirty"]},
#     "blue block": {"is": ["clean"]},
#     "green bowl": {},
#     "red bowl": {}
# }
# query:  Put the cyan block on the yellow block
put_first_on_second("cyan block", "yellow block")
update_wm("Put the cyan block on the yellow block")
```

Prompt 4: world-model reader. The text highlighted in green represents the part that the model is expected to generate.

```
# state = {
#     "objects": ["cyan block", "yellow block", "brown block", "purple block", "blue block", "green bowl", "red bowl", "disinfector"],
#     "relations": [],
#     "disinfector": {"contains": []},
#     "cyan block": {"is": ["dirty"]},
#     "yellow block": {"is": ["clean"]},
#     "brown block": {"is": ["clean"]},
#     "purple block": {"is": ["dirty"]},
#     "blue block": {"is": ["clean"]},
#     "green bowl": {},
#     "red bowl": {}
# }
# query:  Put the cyan block on the yellow block.
# state = {
#     "objects": ["cyan block", "yellow block", "brown block", "purple block", "blue block", "green bowl", "red bowl", "disinfector"],
#     "relations": [["cyan block is on yellow block"]],
#     "disinfector": {"contains": []},
#     "cyan block": {"is": ["dirty"]},
#     "yellow block": {"is": ["dirty"]},
#     "brown block": {"is": ["clean"]},
#     "purple block": {"is": ["dirty"]},
#     "blue block": {"is": ["clean"]},
#     "green bowl": {},
#     "red bowl": {},
# }
```

Prompt 5: world-model writer. The text rendered in blue highlights the updated part of the state.

(highlighted in green). Whenever the writer updates the state representation, we store it in external memory and refer to it as the current state representation.

4 Experiments

To demonstrate the capability of our approach, we evaluate our method on three tabletop domains (shown in Figure 4): simple pick-and-place, block disinfection, and relative weight reasoning. For each domain, we designed a training prompt and consider 20 evaluation episodes, where each episode consists of between 5 and 16 consecutive steps of user queries. We ensure every episode contains at least one query that requires reasoning over the interaction history (i.e., it requires “memory” across steps). This section is organized as follows: First, we provide a description of the three evaluation domains. Second, we present the details of our prompt design. Third, we discuss the evaluation results and then provide qualitative analyses.

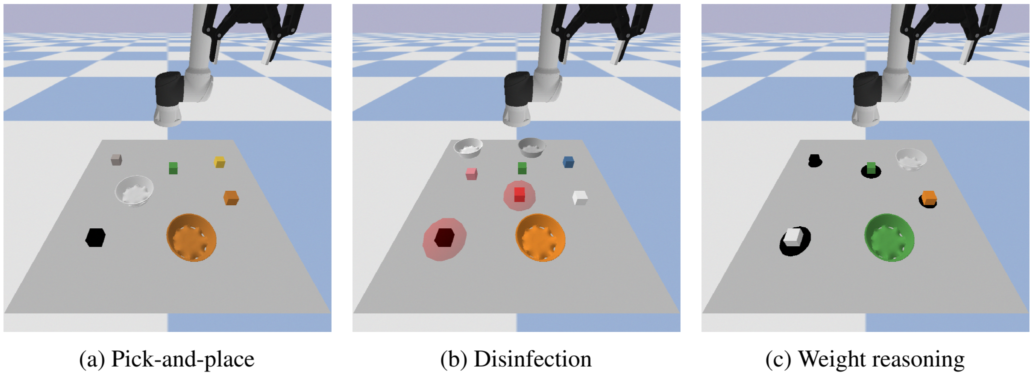

(a) Pick-and-place (b) Disinfection (c) Weight reasoning

Figure 4: The simulated domains we consider include (a) pick-and-place; (b) block disinfection, where the translucent sphere around a block represents its dirtiness (this is not visible to the robot); and (c) relative weight reasoning, where the radius of the disk under each block provides an indication of its weight. These disks are rendered there only for visual aids.

4.1 Simulated Table-top Manipulation Domains

The **simple pick-and-place** domain involves scenarios that require a robot arm to sequentially pick up a block and place it onto another block, bowl, or the table. The model needs to remember and reason over the block locations. The example user queries are “Put the green block in the red bowl”, “What is the color of the block under the pink block?”, and “How many blocks are in the green bowl?”

In the **block disinfection** domain, we consider the scenario in which a block can be either *dirty* or *clean*. When a clean block touches a dirty block (for example by stacking a dirty block on a clean block), it becomes dirty. There is a *disinfector* on the table that cleans any block placed inside it. This scenario emulates a clean-up task in which you might ask a robot to put dirty dishes in a dishwasher or dirty clothes in a washing machine. The user query contains pick-and-place commands similar to those in the simple pick-and-place domain as well as textual utterances that require reasoning over which blocks are clean and dirty, such as “Put all the clean blocks in the green bowl.” This domain presents a particular challenge as the model must effectively track the current cleanliness status of each block and accurately capture the state mutations that happens when a dirty block comes into contact with another clean block.

Relative weight reasoning involves memorizing and reasoning over the relative weights of the blocks. User queries provide information about the weight of blocks (e.g., “The red block is twice the weight of the bronze block”), which are followed by queries that require reasoning over the weights (e.g., “Put blocks in the purple bowl so that their total weight becomes identical to what is in the gray bowl”).

Table 1: Number of successful steps until failure (normalized by episode length) and the success rate for each domain.

	Simple Pick-and-Place		Block Disinfection		Rel. Weight Reasoning	
	successful steps	success rate	successful steps	success rate	successful steps	success rate
Code-as-Policies	0.54	0.00 (0/20)	0.68	0.00 (0/20)	0.84	0.00 (0/20)
Statler (ours)	**0.88**	**0.50 (10/20)**	**0.82**	**0.40 (8/20)**	**0.93**	**0.55 (11/20)**

We run the baseline (Code-as-Policies) and our Statler state-maintaining model on each domain. Table 1 reports the success rates of each method as well as their step count until the first failed attempt to generate the correct code. We normalize the successful steps by the total number of steps for each episode.

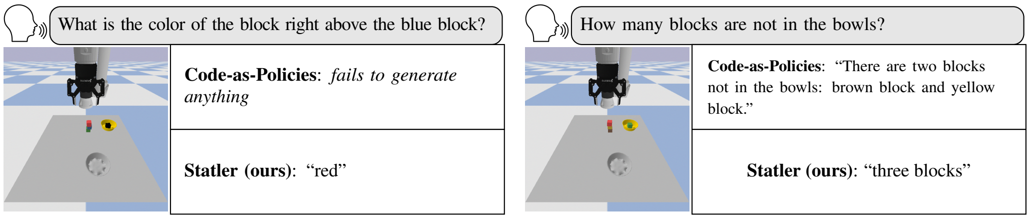

What is the color of the block right above the blue block?
Code-as-Policies: *fails to generate anything*
Statler (ours): "red"
How many blocks are not in the bowls?
Code-as-Policies: "There are two blocks not in the bowls: brown block and yellow block."
Statler (ours): "three blocks"

Figure 5: Examples that show the result of querying language models with and without state maintenance for the environment depicted in the image. In the scenario depicted on the left, a standard language model fails to produce an answer, while our state-maintaining language model produces the correct response. On the right, one of the blocks is currently not visible and so a standard language model (Code-as-Policies) incorrectly identifies two blocks as not being in the bowls. By maintaining a persistent model of the world, our method is aware of the third block and correctly answers the query.

Table 2: Success rates of Code-as-Policies and Statler for non-temporal and temporal queries, truncating at the first failure of each model.

	Non-temporal		Temporal	
	Code-as-Policies	Statler (ours)	Code-as-Policies	Statler (ours)
Simple Pick-and-Place	1.00 (62/62)	1.00 (68/68)	0.31 (9/29)	**0.83 (48/58)**
Block Disinfection	0.99 (148/149)	0.98 (164/168)	0.05 (1/20)	**0.65 (15/23)**
Weight Reasoning	1.00 (107/107)	1.00 (107/107)	0.00 (0/20)	**0.55 (11/20)**

We observe that the baseline Code-as-Policies model correctly processes most of the user queries that do not require reasoning over the past steps, such as "Put the red block on the blue block" or "The red block has the same weight as the blue block" (in this case, `noop()` is the correct code to generate). However, when it comes to the queries that require non-trivial operation of the memory, such as "Put all the dirty blocks in the pink bowl" and "What is the color of the block under the purple block?", the baseline model tends to generate incorrect code or often fails to generate any code at all (see Figure 5 (left)). In contrast, our Statler model successfully handles the majority of cases that require complex logical reasoning over the past history. In each of the three domains, we find that Statler outperforms the baseline in the majority of scenarios.

In order to better understand the behavior of Statler, we analyze the success rate of code generation based on the type of textual utterance. Specifically, we categorize each query as either *temporal* or *non-temporal* depending on whether it involves temporal reasoning. Table 2 summarizes the performance of Statler in comparison to Code-as-Policies on both types of queries. We note that we consider the sequence of steps up until the point that model fails to generate a correct code, including the step on which it failed. The difference in denominator between the two models under the same setting results from the fact that the models fail at different steps in some episodes. We also report an alternative way to calculate the success rate in Table 2, by aligning the set of queries evaluated by both of the models.

Examining the failure cases reveals some interesting observations. Firstly, we find that both models generally successfully handle the basic pick-and-place tasks. However the baseline model consistently fails to generate a response when presented with a non-trivial query that involves reasoning over the past. Secondly, thanks to its state-updating mechanism, our model demonstrated superior comprehension of complex queries, resulting in a better performance. For instance, in queries like "Put the block in the golden bowl on the block in the silver bowl" our model executed flawlessly, whereas the baseline model consistently failed.

Despite its robustness, our model is not without errors. It occasionally generates incorrect responses and still suffers from hallucinations. For example, it hallucinates block conditions (clean or not)

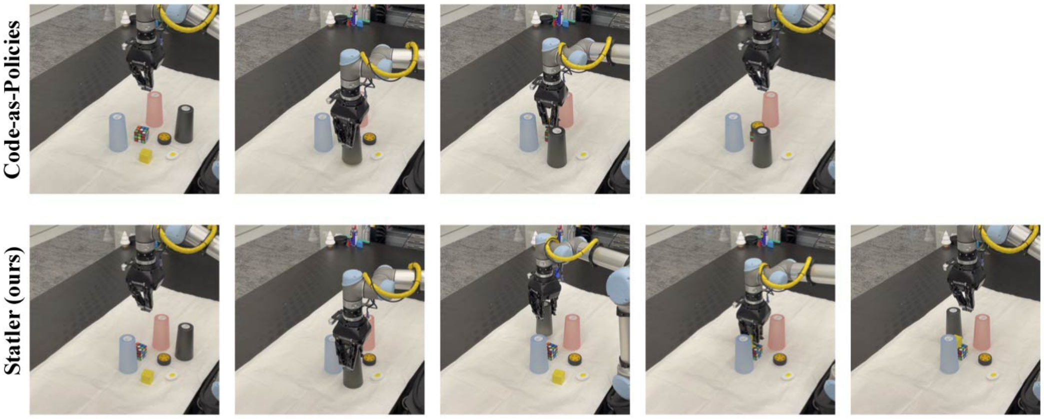


Figure 6: A comparison of the resulting behavior for (top) Code-as-Policies and (bottom) our Statler model for the real robot experiments given the multi-sentence instruction “Put the black cup on the yellow block. Put the yellow block on the Rubik’s cube.” Frames are arranged with time increasing from left to right, and correspond to instances when the robot has placed a (possibly imaginary) object. In order to successfully carry out the instruction, the robot must remove the black cup after placing it above the yellow block in order to place the block on the Rubik’s cube. However, the the baseline Code-as-Policies (top row, third frame) fails to move the black cup aside, leaving the yellow block covered, and instead places an imaginary object on top of the Rubik’s cube.

or locations when the cleanliness of the block is never explicitly described. Moreover, the model’s reasoning strategy seems to predominantly focus on evaluating the weight relationships between blocks, e.g., contemplating whether a block is light or heavy, rather than executing mathematical computations. This weakness became evident when asked to accumulate blocks in a bowl until their total weight surpassed another bowl’s content, as the model underfilled the bowl. Additionally, our model also makes other mistakes and struggles to comprehend ambiguous terms like “other” in queries such as “the other blocks are clean.” In the disinfection domain, it wrongly inferred from the training prompt that a block *at the bottom* becomes dirty when another block is placed on top of it, independent of the cleanliness of the placed block, rather than “a block becomes dirty when it is in contact with another block.”

4.2 Real Robot Experiments

In order to validate our method on a real robot, we implement it on a UR5 arm in a similar tabletop domain as the simulated experiments. Because ground-truth position of objects is not available, unlike in simulation, we use MDETR [15], an open-vocabulary segmentation model, to obtain segmentation masks for objects from an RGB camera on the gripper. Through camera transforms of the masks and a depth camera also located on the gripper, we obtain the (x, y, z) positions for grasping and placement. Besides these details, all of the primitive functions are the same as in simulation. In this domain, the robot is asked to stack objects and to cover objects with different colored cups. At any point, an object is only permitted to be covered by a single object or cover. If the robot is asked to manipulate the bottom object, it must remove the top object. If it is asked to use a new cover, it must remove the old cover. In Figure 6, we provide a short example where the vanilla language model approach fails. The difficulty is in recognizing that the black cup must be removed in order to move the yellow block, which Statler correctly spots. Instead, the vanilla approach assumes that the object does not need to be uncovered, which leads MDETR to incorrectly detect the toy wheel that has yellow color in it as the yellow block.

5 Related Work

Language Understanding for Robotics There is a large body of work on language understanding for robotic agents dating back several decades. A common approach involves symbol grounding [16], whereby words and phrases are mapped to symbols in the robot's world model. Early work [17, 18] relies upon hand-engineered rules to perform this mapping. More recent methods replace these rules with statistical models the parameters of which are trained on annotated corpora [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]. Other methods use neural network-based architectures to jointly reason over natural language utterances and the agent's (visual) observations of the scene [31, 32, 33, 34, 35].

LLMs for Robotics Since LLMs are trained with enormous Internet corpora, their infused common sense have shown to help in the domain of robotics in terms of high-level planning from natural language instructions [4, 5, 36, 4] for both object manipulation [37, 38] and navigation tasks [39, 40, 41, 42]. Combining LLMs with expressive visual-language embeddings also enables impressive capabilities [43]. This has led to efforts to push for general multi-modality embodied models [44, 45].

Code Generation with LLMs Code generation has been one of the most successful use cases for LLMs [2, 46, 47, 48, 49, 3]. Since code can connect with executable APIs for tasks including computation, vision and manipulation, a large chunk of work has focused on code generation with different tools [50, 51, 52]. In particular, Code-as-policies [5] has been one of the first to use code generation within the robotics context.

State Representation in Reasoning State representation is a common formulation in robotics to summarize and provide necessary information for the agents to perform actions [53, 54]. For example, in a Markov chain, the state is constructed so that future predictions are independent of the past given the current state. This saves the agent from remembering all details of the history [55]. State representation has also been helpful in algorithmic reasoning tasks [56, 57]. Instead of using one forward pass to predict the execution result for the whole code snippet, Nye et al. [56] [56] propose to spell out step-by-step intermediate outputs to help infer the final execution results. Also relevant are research efforts that aim to enhance language modeling by rolling out possible future tokens [58].

6 Conclusion

In this paper, we presented Statler, a state-maintaining language model that consists of a world-model reader and a writer. The world-model reader responds to a user query taking into account the current internal state, while offloading the state update to the world-model writer. Our model does not pose any limitations in how the state representation should be formatted, as long as it is represented in the form of a string, leaving some space for flexibility in its design. We evaluated our approach on various simulated and real tasks. The experimental results suggest that our approach effectively maintains state representation and handles non-trivial reasoning over the past steps, whereas the baseline approach (Code-as-Policies) fails to generate correct code on such queries. Since the capability of the world-model reader depends directly on the language model behind it, our model has a potential to handle various challenging scenarios as well as various types of state representations, given a strong backbone LLM.

In addition, having separate models (i.e., the world-model reader and the world-model writer) suggests that it may be possible to use a lightweight language model for some components. For example, if the task for the world-model writer is much easier than the reader, one can utilize a smaller LLM with reduced API costs or one that is hosted locally to complete the task without sacrificing performance.

A potential extension of our work is to integrate numerical representation, such as coordinates and sizes of the objects, into the state. An ability to reason over these quantities will be an important step toward embodied intelligence.

7 Limitations

There are several limitations with the current approach. Firstly, although highly flexible, the world models are designed by hand individually for each task. Ideally there should be an automatic way of generating it, maybe from the LLMs themselves. Secondly, the current world models are still purely text-based, so it does not directly reason about visual information. It will be interesting to see how it will work out when more capable multi-modal models are accessible. Thirdly, in this paper, we assume that the generated code executes successfully, thus if there are issues in execution the updated state will be incorrect. This could be alleviated by providing some feedback from external modules such as image captioning models.

Acknowledgements

We are grateful to National Science Foundation for enabling this work under HDR TRIPODS (No. 2216899), and to Adobe for supporting the second-to-last author through an Adobe Research gift. We thank Luzhe Sun for his help with prompt writing at an early stage, and Richard Xu for helping to set up the simulator.

References

[1] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, A. Herzog, D. Ho, J. Hsu, J. Ibarz, B. Ichter, A. Irpan, E. Jang, R. J. Ruano, K. Jeffrey, S. Jesmonth, N. J. Joshi, R. Julian, D. Kalashnikov, Y. Kuang, K.-H. Lee, S. Levine, Y. Lu, L. Luu, C. Parada, P. Pastor, J. Quiambao, K. Rao, J. Rettinghouse, D. Reyes, P. Sermanet, N. Sievers, C. Tan, A. Toshev, V. Vanhoucke, F. Xia, T. Xiao, P. Xu, S. Xu, M. Yan, and A. Zeng. Do as I can, not as I say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[5] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. R. Florence, and A. Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

[6] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. R. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, N. Brown, T. Jackson, L. Luu, S. Levine, K. Hausman, and B. Ichter. Inner monologue: Embodied reasoning through planning with language models. In *Proceedings of the Conference on Robot Learning (CoRL)*, 2022.

[7] Anthropic introducing 100k Context windows. `https://www.anthropic.com/index/100k-context-windows`. Accessed: 2023-05-11.

[8] S. Sun, K. Krishna, A. Mattarella-Micke, and M. Iyyer. Do long-range language models actually use long-range context? *arXiv preprint arXiv:2109.09115*, 2021.

[9] A. Madaan, S. Zhou, U. Alon, Y. Yang, and G. Neubig. Language models of code are few-shot commonsense learners. In Y. Goldberg, Z. Kozareva, and Y. Zhang, editors, *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

[10] P. Li, T. Sun, Q. Tang, H. Yan, Y. Wu, X. Huang, and X. Qiu. CodeIE: Large code generation models are better few-shot information extractors. *arXiv preprint arXiv:2305.05711*, 2023.

[11] A. Nordmann, N. Hochgeschwender, and S. B. Wrede. A survey on domain-specific languages in robotics. In *Simulation, Modeling, and Programming for Autonomous Robots*, 2014.

[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[13] S. Toshniwal, S. Wiseman, K. Livescu, and K. Gimpel. Chess as a testbed for language model state tracking. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2021.

[14] C.-H. Lee, H. Cheng, and M. Ostendorf. Dialogue state tracking with a language model using schema-driven prompting. *arXiv preprint arXiv:2109.07506*, 2021.

[15] A. Kamath, M. Singh, Y. LeCun, G. Synnaeve, I. Misra, and N. Carion. MDETR - Modulated detection for end-to-end multi-modal understanding. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2021.

[16] S. Harnad. The symbol grounding problem. *Physica D*, 42:335–346, 1990.

[17] T. Winograd. *Procedures As A Representation for Data in a Computer Program for Understanding Natural Language*. PhD thesis, Massachusetts Institute of Technology, 1971.

[18] M. MacMahon, B. Stankiewicz, and B. Kuipers. Walk the talk: Connecting language, knowledge, and action in route instructions. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2006.

[19] T. Kollar, S. Tellex, D. Roy, and N. Roy. Toward understanding natural language directions. In *Proceedings of the ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, 2010.

[20] C. Matuszek, D. Fox, and K. Koscher. Following directions using statistical machine translation. In *Proceedings of the ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, 2010.

[21] D. L. Chen and R. J. Mooney. Learning to interpret natural language navigation instructions from observations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2011.

[22] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. Teller, and N. Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2011.

[23] C. Matuszek, E. Herbst, L. Zettlemoyer, and D. Fox. Learning to parse natural language commands to a robot control system. In *Proceedings of the International Symposium on Experimental Robotics (ISER)*, 2012.

[24] J. Thomason, S. Zhang, R. J. Mooney, and P. Stone. Learning to interpret natural language commands through human-robot dialog. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[25] T. M. Howard, S. Tellex, and N. Roy. A natural language planner interface for mobile manipulators. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[26] D. K. Misra, J. Sung, K. Lee, and A. Saxena. Tell me Dave: Context-sensitive grounding of natural language to manipulation instructions. *International Journal of Robotics Research*, 35 (1-3):281–300, January 2016.

[27] J. Thomason, J. Sinapov, M. Svetlik, P. Stone, and R. J. Mooney. Learning multi-modal grounded linguistic semantics by playing "I spy". In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2016.

[28] J. Thomason, J. Sinapov, R. J. Mooney, and P. Stone. Guiding exploratory behaviors for multi-modal grounding of linguistic descriptions. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2018.

[29] M. Shridhar and D. Hsu. Interactive visual grounding of referring expressions for human-robot interaction. In *Proceedings of Robotics: Science and Systems (RSS)*, 2018.

[30] R. Paul, J. Arkin, D. Aksaray, N. Roy, and T. M. Howard. Efficient grounding of abstract spatial concepts for natural language interaction with robot platforms. *International Journal of Robotics Research*, 37(10):1269–1299, June 2018.

[31] H. Mei, M. Bansal, and M. Walter. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2016.

[32] P. Anderson, Q. Wu, D. Teney, J. Bruce, M. Johnson, N. Sünderhauf, I. D. Reid, S. Gould, and A. van den Hengel. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[33] D. Fried, R. Hu, V. Cirik, A. Rohrbach, J. Andreas, L.-P. Morency, T. Berg-Kirkpatrick, K. Saenko, D. Klein, and T. Darrell. Speaker-follower models for vision-and-language navigation. In *Advances in Neural Information Processing Systems (NeurIPS)*, Dec. 2018.

[34] F. Zhu, Y. Zhu, X. Chang, and X. Liang. Vision-language navigation with self-supervised auxiliary reasoning tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[35] S. Y. Min, D. S. Chaplot, P. Ravikumar, Y. Bisk, and R. Salakhutdinov. FILM: Following instructions in language with modular methods. *arXiv preprint arXiv:2110.07342*, 2021.

[36] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2022.

[37] R. Wang, J. Mao, J. Hsu, H. Zhao, J. Wu, and Y. Gao. Programmatically grounded, compositionally generalizable robotic manipulation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[38] A. Z. Ren, B. Govil, T.-Y. Yang, K. R. Narasimhan, and A. Majumdar. Leveraging language for accelerated learning of tool manipulation. In *Proceedings of the Conference on Robot Learning (CoRL)*, 2023.

[39] A. Majumdar, A. Shrivastava, S. Lee, P. Anderson, D. Parikh, and D. Batra. Improving vision-and-language navigation with image-text pairs from the Web. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020.

[40] S. Y. Gadre, M. Wortsman, G. Ilharco, L. Schmidt, and S. Song. Cows on pasture: Baselines and benchmarks for language-driven zero-shot object navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[41] D. Shah, B. Osiński, S. Levine, et al. LM-Nav: Robotic navigation with large pre-trained models of language, vision, and action. In *Proceedings of the Conference on Robot Learning (CoRL)*, 2023.

[42] C. Huang, O. Mees, A. Zeng, and W. Burgard. Visual language maps for robot navigation. *arXiv preprint arXiv:2210.05714*, 2022.

[43] M. Shridhar, L. Manuelli, and D. Fox. CLIPort: What and where pathways for robotic manipulation. *arXiv preprint arXiv:2109.12098*, 2021.

[44] A. Zeng, A. S. Wong, S. Welker, K. Choromanski, F. Tombari, A. Purohit, M. S. Ryoo, V. Sindhwani, J. Lee, V. Vanhoucke, and P. R. Florence. Socratic models: Composing zero-shot multimodal reasoning with language. *arXiv preprint arXiv:2204.00598*, 2022.

[45] D. Driess, F. Xia, M. S. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, et al. PaLM-E: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[46] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. X. Song, and J. Steinhardt. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*, 2021.

[47] Y. Li, D. H. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, Tom, Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de, M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, Alexey, Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de, Freitas, K. Kavukcuoglu, and O. Vinyals. Competition-level code generation with AlphaCode. *Science*, 378:1092–1097, 2022.

[48] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen. CodeT: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[49] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. J. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[50] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[51] D. Sur'is, S. Menon, and C. Vondrick. ViperGPT: Visual inference via Python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023.

[52] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*, 2023.

[53] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[54] D. Abel, D. Arumugam, L. Lehnert, and M. L. Littman. State abstractions for lifelong reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[55] P. A. Gagniuc. *Markov Chains: From Theory to Implementation and Experimentation*. Wiley, 2017.

[56] M. Nye, A. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, C. Sutton, and A. Odena. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, abs/2112.00114, 2021.

[57] A. J. H. Nam, M. Ren, C. Finn, and J. L. McClelland. Learning to reason with relational abstractions. *ArXiv*, abs/2210.02615, 2022.

[58] L. Du, H. Mei, and J. Eisner. Autoregressive modeling with lookahead attention. *arXiv preprint arXiv:2305.12272*, 2023.

A World Model Design

While our method does not impose any restriction on the format of the state representation, we adopt a structured (JSON-like) representation of the state in each of the domains that we consider. This choice is based on recent results demonstrating the advantages of structured representations, particularly when the output is also structured [10]. There is no limitation in the design of state representation as long as it forms a string. Through the training prompt, we encourage the model to store object-level information in the state representation (e.g., whether the block is clean or not, relative weight of the block, or if a bowl contains some blocks). Since this structure is not specific to any particular domain, the state information of all three domains can be represented in a similar fashion.

Our experiments evaluate the advantages of maintaining a state representation using the world-model reader and writer in the context of non-hierarchical code generation. While our framework allows hierarchical code generation, which supports more complicated queries and improves robustness (by recursively defining undefined functions) as highlighted in [5], it is orthogonal to the way in which the response is generated. More concretely, in our experiments we inform the model what set of functions the model has access to at the code execution time. In addition to the built-in python functions and statements, we allow the baseline model to use `put_first_on_second`, `say` and `noop` functions. For the state-maintained model we additionally allow `update_wm` function that takes a string and triggers world-model writer to update the state.

B State Representations for Each Domain

Prompt 7, 8, 9, 12 show examples of the state representations we have in an intermediate step of an evaluation episode. In our experiments, the world state is kept in a JSON-like format, comprising three key aspects: a list of objects, object relations, and object-specific data. The ’objects’ key contains a list of objects present in the scene (excluding the table, due to our tabletop environment setup). The ’relations’ key represents block arrangements, like ’red block is on green block’. Each object also gets a key, with the stored information varying per object and experiment. For instance, the bowls and the disinfector are considered as containers, storing data about contained blocks. For blocks, information storage changes with each experiment: no information for the simple pick-and-place, weight relations for relative weight reasoning, and cleanliness status for block disinfection. By default, all objects are placed on the table unless explicitly designated to be inside one of the containers.

C Additional Analyses of Simulation-based Experiments

Table 2 and Table 3 display the results of the same experiment, but take different approaches to collate the results. We feed both the baseline (Code-as-Policies) and our Statler models with a series of sequential queries for each evaluation episode. An important aspect to consider is the inter-dependency of these queries. For instance, if a model fails to perform the query “Place the red block on the green block”, the subsequent queries that presume its success, such as “Place the block on the green block on the table” cannot be addressed. In other words, if the model fails to respond to a query, the subsequent queries are likely invalid given the configuration of the environment.

Table 3: Success rates of Code-as-Policies and Statler for non-temporal and temporal queries, truncating at the first failure of any model.

	Non-temporal		Temporal	
	Code-as-Policies	Statler (ours)	Code-as-Policies	Statler (ours)
Simple Pick-and-Place	1.00 (62/62)	1.00 (62/62)	0.32 (9/28)	**0.86 (24/28)**
Block Disinfection	0.99 (148/149)	0.99 (147/149)	0.00 (0/18)	**0.61 (11/18)**
Weight Reasoning	1.00 (107/107)	1.00 (107/107)	0.00 (0/20)	**0.55 (11/20)**

```
# state = {
#      "objects": ["green block", "orange block", "white block", "black block", "golden bowl", "silver bowl"],
#      "relations": ["black block is on green block"],
#      "green block": {},
#      "orange block": {},
#      "white block": {},
#      "black block": {},
#      "golden bowl": {"contains": ["white block"]},
#      "silver bowl": {"contains": ["orange block"]},
# }
```

Prompt 6: State representation (stacking)

```
# state = {
#     "objects": ["purple block", "bronze block", "green block", "red block", "transparent bowl", "blue bowl"],
#     "relations": [],
#     "purple block": {"weight": green_block.weight * 2},
#     "bronze block": {"weight": red_block.weight / 2},
#     "green block": {"weight": purple_block.weight / 2},
#     "red block": {},
#     "transparent bowl": {},
#     "blue bowl": {"contains": ["bronze block"]},
# }
```

Prompt 7: State representation (weight)

```
# state = {
#     "objects": ["green block", "white block", "black block", "blue block", "pink block", "transparent bowl", "platinum bowl", "disinfector"],
#     "relations": [],
#     "disinfector": {"contains": []},
#     "green block": {"is": ["dirty"]},
#     "white block": {},
#     "black block": {},
#     "blue block": {"is": ["clean"]},
#     "pink block": {"is": ["clean"]},
#     "transparent bowl": {"contains": ["green block"]},
#     "platinum bowl": {}
# }
```

Prompt 8: State representation (disinfection)

```
# state = {
#     ’blocks’: {’yellow block’: None, ’toy wheel’: None, ’rubiks cube’None, ’toy egg’: None},
#     ’covers’: (’black cup’, ’blue cup’, ’red cup’)
# }
```

Prompt 9: State representation (real robot)

Consequently, we’ve chosen to truncate the testing episodes in two ways. The first approach truncates the evaluation episode as soon as either of the models fails a query, thus aligning the evaluation episode length for both models (Table 3 summarizes these results). The second approach involves independent truncation for each model , leading to different episode length to consider by the model (Table 2 summarizes these results).

Although both approaches have inherent shortcomings, we believe the true evaluation of these models’ performance lies somewhere in between. Neither is ideal, but together they offer a more comprehensive understanding of each model’s performance.

Following are some examples of the evaluation episodes for the simple pick-and-place, weight reasoning, and block disinfection scenarios. The query “Put the red block in the orange bowl” (Prompt 10, Line 34) is a non-temporal query, while “Put all the dirty blocks on the table” (Prompt 10, Line 40) is a temporal query.

```
eval_episode = {
    "init_state": ’’’
    # state = {
    #     "objects": ["green block", "white block", "black block", "blue block", "pink block", "red block", "orange bowl
", "silver bowl", "disinfector"],
    #     "relations": [],
    #     "disinfector": {"contains": []},
    #     "green block": {},
    #     "white block": {},
    #     "black block": {},
    #     "blue block": {},
    #     "pink block": {},
    #     "red block": {},
    #     "orange bowl": {},
    #     "silver bowl": {}
    # }
    ’’’,
    "dirty_list": ["red block"],
    "init_simple_state": ’’’
    objects = ["green block", "white block", "black block", "blue block", "pink block", "red block", "orange bowl", "
silver bowl", "disinfector"],
    ’’’,
    "episode": [
        {"user_query": ’’’the red block is dirty.’’’,
        "gold_code": ’’’update_wm("the red block is dirty.")’’’},

        {"user_query": ’’’the pink block is clean.’’’,
        "gold_code": ’’’update_wm("the pink block is clean.")’’’},

        {"user_query": ’’’Put the pink block in the disinfector’’’,
        "gold_code": ’’’
        put_first_on_second("pink block", "disinfector")
        update_wm("Put the pink block in the disinfector. the pink block becomes clean.")
        ’’’},

        {"user_query": ’’’Put the red block in the orange bowl’’’,
        "gold_code": ’’’
        put_first_on_second("red block", "orange bowl")
        update_wm("Put the red block in the orange bowl.")
        ’’’},

        {"user_query": ’’’Put all the dirty blocks on the table.’’’,
        "gold_code": ’’’
        put_first_on_second("red block", "table")
        update_wm("Put the red block on the relations.")
        ’’’},

        {"user_query": ’’’Put all the clean blocks on the table.’’’,
        "gold_code": ’’’
        put_first_on_second("pink block", "table")
        update_wm("Put the pink block on the relations.")
        ’’’},

        {"user_query": ’’’Put the red block on the pink block’’’,
        "gold_code": ’’’
        put_first_on_second("red block", "pink block")
        update_wm("Put the red block on the pink block. the pink block becomes dirty.")
        ’’’},

        {"user_query": ’’’Put the red block in the orange bowl’’’,
        "gold_code": ’’’
        put_first_on_second("red block", "orange bowl")
        update_wm("Put the red block in the orange bowl.")
        ’’’},

        {"user_query": ’’’Put the red block on the table.’’’,
        "gold_code": ’’’
        put_first_on_second("red block", "table")
        update_wm("Put the red block on the table.")
        ’’’},

        {"user_query": ’’’Put the pink block on the red block’’’,
        "gold_code": ’’’
        put_first_on_second("pink block", "red block")
        update_wm("Put the pink block on the red block.")
        ’’’},

        {"user_query": ’’’Put the red block and the pink block in the disinfector’’’,
        "gold_code": ’’’
        put_first_on_second("red block", "disinfector")
        put_first_on_second("pink block", "disinfector")
        update_wm("Put the red block and the pink block in the disinfector. the red block and the pink block become
clean.")
        ’’’},

        {"user_query": ’’’Put all the clean blocks on the table.’’’,
        "gold_code": ’’’
        put_first_on_second("red block", "table")
        put_first_on_second("pink block", "table")
        update_wm("Put the red block and the pink block on the relations.")
        ’’’}
    ]
}
```

Prompt 10: Sample Evaluation Episode (Block Disinfection)

```
eval_episode = {
    "obj_name_to_weight": {"green block": 4.,
                           "white block": 4.,
                           "black block": 2.,
                           "orange block": 2.,},
    "init_state": ’’’
    # state = {
    #     "objects": ["green block", "orange block", "white block", "black block", "transparent bowl", "green bowl"],
    #     "relations": [],
    #     "green block": {},
    #     "orange block": {},
    #     "white block": {},
    #     "black block": {},
    #     "transparent bowl": {},
    #     "green bowl": {},
    # }
    ’’’,
    "init_simple_state": ’’’
    # objects = ["green block", "orange block", "white block", "black block", "transparent bowl", "green bowl"],
    ’’’,
    "episode": [
        {
            "user_query": "The green block has the same weight as the white block",
            # weight: green block == white block
            "gold_code": ’’’’’’,
            "gold_next_state": ’’’’’’,
        },
        {
            "user_query": "The white block is twice the weight of the black block",
            # weight:
            # - green block == white block
            # - white block == black block x 2
            "gold_code": ’’’ ’’’,
            "gold_next_state": ’’’ ’’’,
        },
        {
            "user_query": "The orange block is half the weight of the green block",
            # weight:
            # - green block == white block
            # - white block == black block x 2
            # - orange block == white block / 2 == black block
            "gold_code": ’’’ ’’’,
            "gold_next_state": ’’’ ’’’,
        },
        {
            "user_query": "Put the orange block in the transparent bowl",
            "gold_code": ’’’put_first_on_second("orange block", "transparent bowl")’’’,
            "gold_next_state": ’’’ ’’’,
        },
        {
            "user_query": "Put the blocks in the green bowl so that their total weight becomes identical to what is in the transparent bowl",
            "gold_code": ’’’put_first_on_second("black block", "green bowl")’’’,
            "gold_next_state": ’’’ ’’’,
        },
    ]
}
```

Prompt 11: Sample Evaluation Episode (Weight Reasoning)

```
eval_episode = {
    "obj_name_to_weight": {"green block": 4.,
                           "white block": 4.,
                           "black block": 2.,
                           "orange block": 2.,},
    "init_state": ’’’
    # state = {
    #     "objects": ["green block", "orange block", "white block", "black block", "transparent bowl",
"green bowl"],
    #     "relations": [],
    #     "green block": {},
    #     "orange block": {},
    #     "white block": {},
    #     "black block": {},
    #     "transparent bowl": {},
    #     "green bowl": {},
    # }
    ’’’,
    "init_simple_state": ’’’
    # objects = ["green block", "orange block", "white block", "black block", "transparent bowl", "
green bowl"],
    ’’’,
    "episode": [
        {
            "user_query": "The green block has the same weight as the white block",
            # weight: green block == white block
            "gold_code": ’’’’’’,
            "gold_next_state": ’’’’’’,
        },
        {
            "user_query": "The white block is twice the weight of the black block",
            # weight:
            # - green block == white block
            # - white block == black block x 2
            "gold_code": ’’’ ’’’,
            "gold_next_state": ’’’ ’’’,
        },
        {
            "user_query": "The orange block is half the weight of the green block",
            # weight:
            # - green block == white block
            # - white block == black block x 2
            # - orange block == white block / 2 == black block
            "gold_code": ’’’ ’’’,
            "gold_next_state": ’’’ ’’’,
        },
        {
            "user_query": "Put the orange block in the transparent bowl",
            "gold_code": ’’’put_first_on_second("orange block", "transparent bowl")’’’,
            "gold_next_state": ’’’ ’’’,
        },
        {
            "user_query": "Put the blocks in the green bowl so that their total weight becomes
identical to what is in the transparent bowl",
            "gold_code": ’’’put_first_on_second("black block", "green bowl")’’’,
            "gold_next_state": ’’’ ’’’,
        },
    ]
}
```

Prompt 12: Sample Evaluation Episode (Simple Pick-and-Place)