Robotics Tasks Generation through Factorization

Stanford CS224N Custom Project

Yihan Zhou, Angela Yi, Ranajit Gangopadhyay
Department of Computer Science
Stanford University
zhouy043@stanford.edu, angelayi@stanford.edu, rganguli@stanford.edu

Abstract

Training general-purpose robotic policies and embodied agents requires significant amounts of real-world interaction data, which is often very expensive to collect. Nowadays, most learning is done through simulators, but to generalize at the task level requires increasing the task level diversity. However, this is a challenge as it requires specifying a lot more data in order for the system to understand new assets. With the recent significant progress in Large Language Models (LLMs), people have been working on applying LLMs for both tasks and simulation code generation. In this paper, we propose a different way of generating rich simulation environments through exploiting the modularity and sparse structure in robotics scene and tasks.

1 Key Information to include

- External Mentor: Fan-Yun Sun (fanyun@stanford.edu)
- TA Mentor: Yuhui Zhang (yuhuiz@stanford.edu)
- Team Contributions:
  Yihan Zhou: Research, Code Development/Testing on Approach 2
  Angela Yi: Research, Code Development/Testing on Approach 1
  Ranajit Gangopadhyay: Research

2 Introduction

Simulation environments have become the predominant approach in training diverse robotics skills, as it provides an accessible way to explore an unlimited amount of environments and collect data at scale through parallel computation. However, constructing these environments require a considerable amount of effort, including designing tasks, selecting meaningful assets, and designing plausible scenes. Due to these challenges, typical human-curated benchmarks have only included tens to hundreds of tasks (Shridhar et al., 2021).

In light of the recent progress in Large Language Models (LLMs), a new paradigm termed *Generative Simulation* (Wang et al., 2023b) has been presented, which involves using these foundation models to generate all the stages needed for diverse robotic skill learning in simulation, such as the scene descriptions, asset selection and generation, and reward distributions. These foundation models also are able to generate scenes and tasks that closely resemble real-world scenarios. Some recent work in this area include GenSim (Wang et al., 2023a), which utilizes few-shot prompting and a task library for doing code generation, RoboGen (Wang et al., 2023b) which goes through a series of stages to generate information used for skill learning, and Holodeck (Yang et al., 2023) which uses a structured way to generate code for diverse 3D environments.

In this paper, we will propose a different direction in generating simulations through a factorized representation. We approached the problem by breaking down complex tasks into modular and

composable pieces with the minimal context needed. Given a complex task, we decomposed the toplevel task into the shared state variables and independent subtasks. By reducing the complexity of the task into subtasks with a limited set of states, we decrease the possibility for errors on the later generated code produced by the foundational model. The decomposition attempts to exploit the world knowledge from LLMs and reduces the need for referencing any existing task library. We also included an intermediate verification phase which brought feedback to the foundation model if code was generated incorrectly.

The effectiveness of the factorized representation was assessed through comparing the generated code between GenSim and our framework. Correctness of the generated code was evaluated on 3 angles: syntax correctness, runtime correctness, and task completion. We found that on 55 tasks, our framework was able to generate significantly more runtime-correct code, and slightly more task-completable code.

3 Related Work

The paper that inspired our work to explore generative simulation using LLMs is GenSim, (Wang et al., 2023a). The paper contains 3 components: (1) a two-stage prompting mechanism that generates task descriptions, (2) a few-shot prompting process for code generation, and (3) a task library that caches high-quality instruction code for fine-tuning. Specifically, when generating code for the simulation, GenSim queries its previously curated task library for code related to similar tasks. Then prompt the LLM to use the selected code template as reference to generate the target task.

Holodeck (Yang et al., 2023) takes a different approach in generating scenes through breaking down the problem into different modules: the floor and wall, doorway and window, object selection, and layout design. It uses LLMs to generate spatial relational constraints between objects. We drew inspiration from its scaleable approach for scene generation and use it to inform our task generation goal.

RoboGen (Wang et al., 2023b) used a self-guided approach of propose-generate-learn cycle. In this case, the agent proposes tasks and skills to learn, then generates simulation environments by populating assets, Next, the agent decomposes the tasks into subtasks and identifies learning approaches (reinforced learning, motion planning, or trajectory optimizations), and learns the policy to acquire the skill. However, the challenge with this approach was to verify if the learned skills solved the corresponding tasks.

4 Approach

As stated previously, our approach takes a robotics task and decomposes it into a sequence of subtasks where each subtask only pertains to a small set of skills. The simulations can be viewed as systems of interlinked events at a fundamental level.

To put it concretely, let $\mathcal{M}$ denotes a Contextual Markov Decision Process (CMDP), $\mathcal{M} = \langle S, A, O, T, \Omega, R, \pi_0 \rangle$, where S denotes a set of states, A denotes a set of robot actions, O denotes a set of observations from how the robot conducts task in the simulation, $T : S \times A \to S^*$ is a transition function where S^* is the new state, $\Omega : S \to O$ is an observation function to detect whether the subtask function goal was achieved, $R : S \to \mathbb{R}$ is the reward model, and π_0 is the initial state distribution. We aim to generate a new target CMDP $\mathcal{M}^*$ through some natural language instruction sent to the LLM, which we'll denote as q_{text}. We explain our two approaches as following.

$$
\begin{aligned}
p(\mathcal{M}^*|\mathcal{M}, q_{\text{text}}) =& p(S^*, T^*, R^*, \Omega^*, \pi_0^*|\mathcal{M}, q_{\text{text}}) && (1)\\
=& p(S^*|\mathcal{M}, q_{\text{text}})\cdot && (2)\\
& p(T^*|S^*, \mathcal{M}, q_{\text{text}})\cdot && (3)\\
& p(R^*|S^*, T^*, \mathcal{M}, q_{\text{text}})\cdot && (4)\\
& p(\Omega^*|S^*, T^*, R^*, \mathcal{M}, q_{\text{text}})\cdot && (5)\\
& p(\pi_0^*|S^*, T^*, R^*, \Omega^*, \mathcal{M}, q_{\text{text}}) && (6)
\end{aligned}
$$

We first take the initial $\mathcal{M}$ and query the LLM with q_{text} to describe the ideal end space state S^* in order to accomplish the task. Then, we instruct the LLM to design a series of T^*, the subtask functions, R^*, the reward function of each subtask, Ω^* observation function (through the `addgoal` function in subtask code) to define the subtask's acceptance criteria, and π_0^* the initial state distribution of where and how each assets to be rendered. Leveraging factorization, we can prompt LLMs to create modular subtasks that can be used to train agents in acquiring specific skills that can be combined to help them complete more complex tasks.

4.1 Approach 1

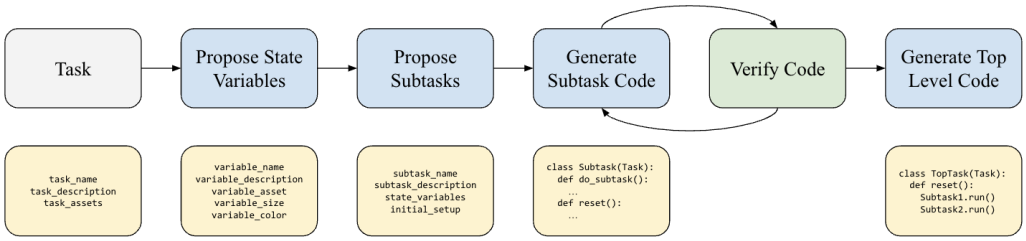


Figure 1: Approach 1 workflow

We started by forking the GenSim (Wang et al., 2023a) codebase which already contains code for testing the simulated environments using CLIPort's frameworks (Shridhar et al., 2021). To modularize the code generation and model off of our factorized graph method, we split the workflow into a couple of steps (1) generate state variables, (2) generate subtasks, (3) generate subtask code, and (4) generate the toplevel code. Prompting was done on a pre-trained gpt-4-1106-preview model. A diagram of the workflow can be found in Figure 1.

4.1.1 Generating state variables

The first step is to define the initial state space of the task through constructing state variables representing what the robotic agent will be engaging within the environment. Given a task, its description, and the assets used (in the form of urdf files), we prompted the LLM to think about how to come up with the higher level variables that are needed to accomplish the task. Whereas original assets can be considered as low level representations of each object in the graph, specifically what urdf file to display in the simulation, the state variables represent semantically what each element in the task represent. Some examples include the blocks used to build the base of a car, or the cylinders used as the wheels of the car. The variables were returned in the format of the variable name, their description, their size, urdf file and their color.

4.1.2 Generating subtasks

The second step is to generate the subtasks of the factor graph which splits up the reward distribution based on these subtasks. Given the task, description, assets, and the newly generated state variables, we asked the LLM to think about how to split the task into sizable subtasks while selecting a subset of the state variables. By prompting the LLM to create subtasks based on a subset of state variables, we create edges between the factorized functions and states, rather than having all the states in one higher level task. This not only prepare for a simpler code generation request for next step, but also allows us to modularize the task and potentially reuse these subtasks for future higher level tasks. To encourage the LLM to come up with independent subtasks, we also prompted the LLM to think about what is the intended setting of environment before each subtask is run.

4.1.3 Generating subtask code

The third step is to generate code for each subtask. As we have already split the top-level task into subtasks with their needed variables, we can now generate the code for the task by separately generating code for each subtasks. This generates more error resistant code as the LLM can narrow its mind to only thinking about the current subtask. We also provided external knowledge of the

physical environment and to the coding environment by adding in information about the available APIs. For example, how to place an asset at a random location for the robotic arm to grab, or how to rotate a given asset in the scene.

As each subtask is now its own independent task, we can also separately verify the correctness of each subtask before generating the entire toplevel task. This is done by checking for syntax errors, runtime errors, and task completion errors. If any errors occurred, we would then provide the information back to the LLM and ask it to regenerate the subtask code, avoiding those errors.

Figure 2: A sample output of the top-level code generated for the task StackBlocksInContainer.

4.1.4 Generating top level code

The final step is to generate the toplevel code by piecing together the previously generated subtask code. By modularizing the subtask code previously, the LLM has an easier task of calling the subtask components rather than needing to generate the entire code. An example output of the toplevel code can be found in Figure 2.

4.2 Approach 2

We also took a different approach aiming to achieve the same objective. A diagram of the workflow can be found in Figure 3.

Initially, given a task proposal, we instruct the LLMs to delineate it into a task name, a detailed description, and the necessary assets for constructing the simulation environment. This task name and description are then broken down into a series of subtasks, each identified by its own name and detailed description. For each subtask, we employ the LLM to generate the required state variables and execution code using the subtask's name and description as directives. The generated state variables and code for each subtask are then scrutinized for syntax and runtime errors. If errors are detected, the feedback is incorporated back into the original prompt for subtask variable and code generation, thereby refining the results. The culmination of this process is the assembly of all the

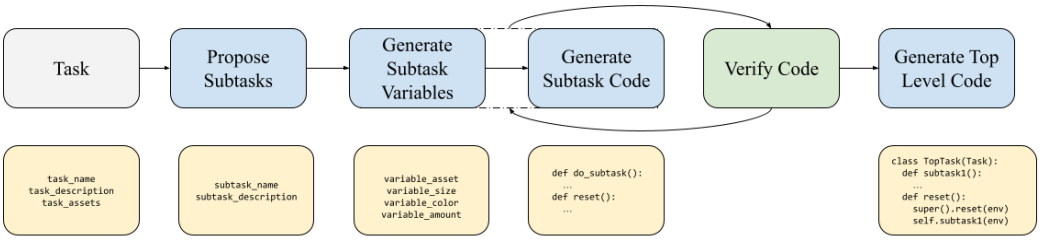


Figure 3: Approach 2 workflow

state variables and functions corresponding to the subtasks. An example output of the code can be found in Figure 4

5 Experiments

5.1 Data

Our evaluation dataset is from a list of 152 previously generated tasks from GenSim which contains the task name, task description, and assets used for this task. We compared against GenSim's "top-down-chain-of-thought" prompting. We evaluated on a subset of these tasks where GenSim was able to run successfully on 10, and failed on 50. The subset of tasks we chose contained those that required multiple steps, rather than very simple tasks which could be very easily generated in one shot.

5.2 Evaluation method

We evaluated the code generation from 3 angles: syntax correctness, runtime success, and agent task completion.

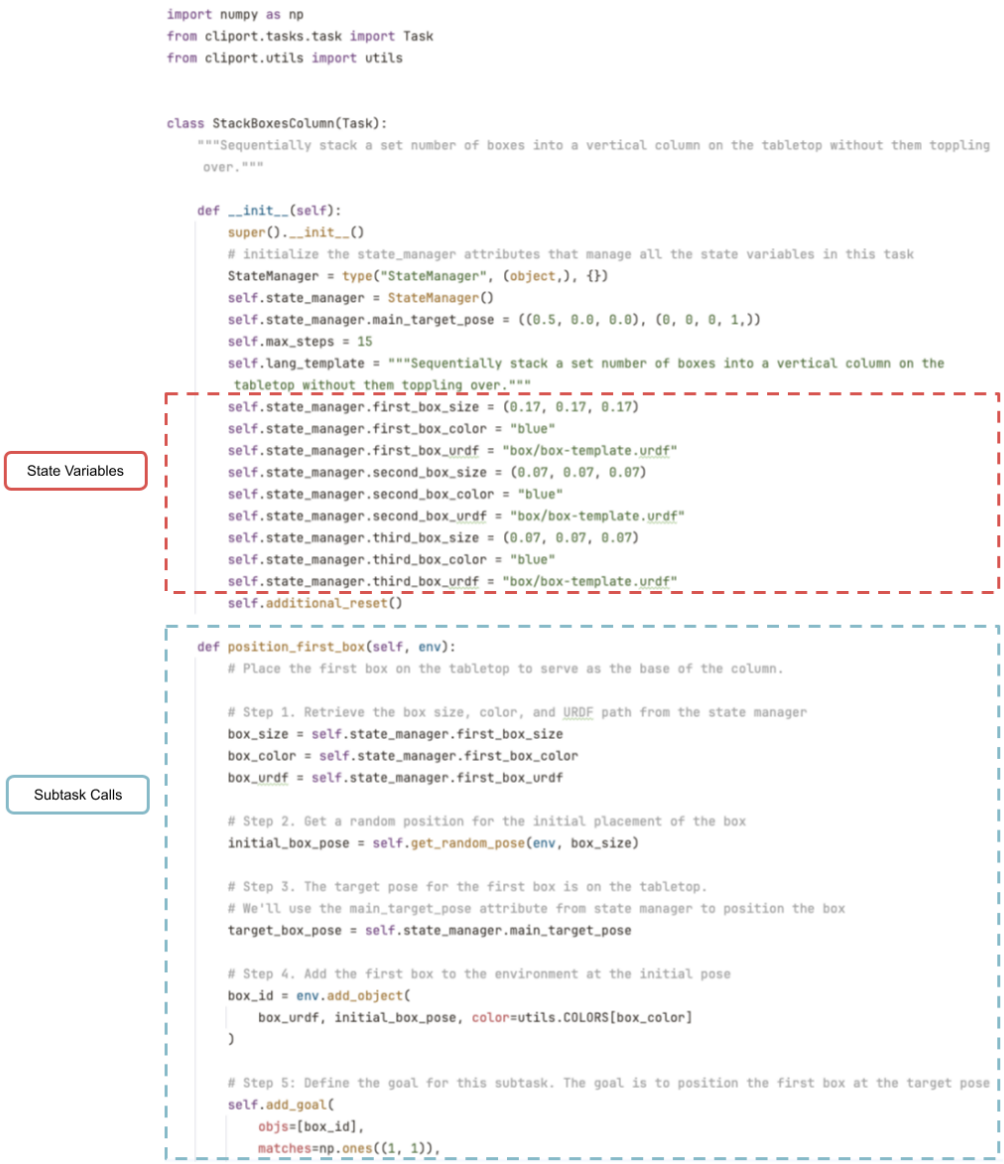
To evaluate syntax correctness, we counted the number of times the generated code could be loaded by python. To evaluate runtime correctness, we counted the number of times an environment simulation instance initiating successfully and an agent running the generated code would not error out. And to evaluate task completion, we used the oracle agent from GenSim to train on the task and see if it has completed the goals added in the generated task. For each step, agent iterates through all objects that do not have their goal completed and attempts to take an action to complete the task. An episode is considered "successful" when the agent accomplishes every subtask and achieves a total reward of 99%, and a task is considered "completed" when half of the total run episodes are passing.

5.3 Results

	Syntax Pass Rate	Runtime Pass Rate	Task completion Rate
Baseline Generated Code	95.0%	53.3%	16.7%
Approach 1	96.4%	78.2%	29.1%
Approach 2	98.3%	90.2%	16.7%

6 Analysis

Overall, our work aimed to validate that factorizing the tasks into subtasks, and giving run-time feedback to LLM improved code generation accuracy, specifically around syntax and runtime correctness. Instead of generating the task code in one attempt, we provide tailored context of the previous code when each subtask was generated. This ensured that the code generated for the subsequent task had some understanding of the subtask completed prior.




```python
import numpy as np
from cliport.tasks.task import Task
from cliport.utils import utils


class StackBoxesColumn(Task):
    """Sequentially stack a set number of boxes into a vertical column on the tabletop without them toppling
     over."""

    def __init__(self):
        super().__init__()
        # initialize the state_manager attributes that manage all the state variables in this task
        StateManager = type("StateManager", (object,), {})
        self.state_manager = StateManager()
        self.state_manager.main_target_pose = ((0.5, 0.0, 0.0), (0, 0, 0, 1,))
        self.max_steps = 15
        self.lang_template = """Sequentially stack a set number of boxes into a vertical column on the
         tabletop without them toppling over."""
        self.state_manager.first_box_size = (0.17, 0.17, 0.17)
        self.state_manager.first_box_color = "blue"
        self.state_manager.first_box_urdf = "box/box-template.urdf"
        self.state_manager.second_box_size = (0.07, 0.07, 0.07)
        self.state_manager.second_box_color = "blue"
        self.state_manager.second_box_urdf = "box/box-template.urdf"
        self.state_manager.third_box_size = (0.07, 0.07, 0.07)
        self.state_manager.third_box_color = "blue"
        self.state_manager.third_box_urdf = "box/box-template.urdf"
        self.additional_reset()

    def position_first_box(self, env):
        # Place the first box on the tabletop to serve as the base of the column.

        # Step 1. Retrieve the box size, color, and URDF path from the state manager
        box_size = self.state_manager.first_box_size
        box_color = self.state_manager.first_box_color
        box_urdf = self.state_manager.first_box_urdf

        # Step 2. Get a random position for the initial placement of the box
        initial_box_pose = self.get_random_pose(env, box_size)

        # Step 3. The target pose for the first box is on the tabletop.
        # We'll use the main_target_pose attribute from state manager to position the box
        target_box_pose = self.state_manager.main_target_pose

        # Step 4. Add the first box to the environment at the initial pose
        box_id = env.add_object(
            box_urdf, initial_box_pose, color=utils.COLORS[box_color]
        )

        # Step 5: Define the goal for this subtask. The goal is to position the first box at the target pose
        self.add_goal(
            objs=[box_id],
            matches=np.ones((1, 1)),
```

Figure 4: A sample output of the top-level code generated for the task StackBoxesColumn.

6.1 Syntax Correctness

In general, LLM was able to provide syntax correct code throughout our experiments. The exception was when the LLM was unaware of an existing utility function, and it would try to implement its own custom function. This would oftentimes result in an error because it was incompatible with the system. By providing more details about the available APIs to the LLM, fixed most of the syntax issues.

6.2 Runtime Success

We initially observed many runtime failures because the subtasks being too granular, such as "pick up the ball" and "put down the ball". Due to the specific setting of the simulation environment, each goal of the subtask needs to be evaluated against the position and rotation updates of the targeted assets. When the subtask design become excessively granular, it becomes incompatible to the goal

evaluation of the simulation environment. By providing more detailed instruction and some successful subtask design as examples, we enable the in-context training for LLMs break down the subtasks with learnable and feasible subtask granularity.

6.3 Agent Task Completion

Both Gensim and our work appear to struggle with having the oracle agent successfully complete the task (earn a 99% reward) after the environment was created successfully. We found this to be a challenging problem to solve as more real-life factors came into play here

Task generation needs to comply certain rules from the simulation which could be trivial for certain tasks. For example, the existing reward function measures object location through the center of the object. When an object was placed in the targeted location but not being perfectly oriented, this requires the reward function to be flexible enough to calibrate the successfulness of the task. This therefore requires additional support from the simulation environment. A similar issue for reward calculation is that placing objects in circles could also be challenging.

Additionally, we anticipate a more rigorous spatial reasoning step that provides targeted poses would help to increase the task completion rate. For task "build car", to create the targeted pose of the 2nd subtask, "build car body", the LLM needs to understand where the car base was placed in the previous subtask "build car base". Accurately reasoning through the targeted pose of each subtask and eventually compose the final task outcome required, requires LLM to have strong sense of spatial understanding of the simulation environment.

7 Conclusion

Utilizing LLMs and their abundant world knowledge to generate embodied simulations robustly and at scale has been an active research area. Our work attempted to contribute to this domain by task decomposition, we proved that we could reduce the difficulty of task code generation by breaking down a task into more granular subtasks, providing code validation feedback of a smaller task, and providing minimally needed context to help mitigate LLM imput limitation and help with implementation robustness.

Overall, it seems like factorizing the tasks into subtasks, and giving feedback to models on their code does help with the accuracy of code generation in terms of syntax and runtime correctness. However, there are some limitations to our work. Similar to the GenSim paper, we did not do evaluations on more complex robotics agents; instead we stuck to the Universal Robot UR5e equipped with a suction gripper which limits the tasks that could be included from the Ravens and CLIPort datasets such as sweeping piles, folding clothes, and manipulating rope. Additionally, we could not determine proper metric to automatically measure if the asset composition in the simulation are reflective of the initially provided task and task description. Future investigation that focusing on the mentioned directions would help solidify the research findings further.

References

Mohit Shridhar, Lucas Manuelli, and Dieter Fox. 2021. Cliport: What and where pathways for robotic manipulation.

Lirui Wang, Yiyang Ling, Zhecheng Yuan, Mohit Shridhar, Chen Bao, Yuzhe Qin, Bailin Wang, Huazhe Xu, and Xiaolong Wang. 2023a. Gensim: Generating robotic simulation tasks via large language models. In *Arxiv*.

Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory Erickson, David Held, and Chuang Gan. 2023b. Robogen: Towards unleashing infinite data for automated robot learning via generative simulation.

Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, Chris Callison-Burch, Mark Yatskar, Aniruddha Kembhavi, and Christopher Clark. 2023. Holodeck: Language guided generation of 3d embodied ai environments. *arXiv preprint arXiv:2312.09067*.

8 Appendix

8.1 Intermediate results from Approach 1

8.1.1 Example of state variables generated for the task "Build Bridge"

```
"state_variables": [
    {
        "variable_description": "Base block to construct bridge foundation",
        "variable_name": "base_block_yellow",
        "variable_size": "[0.1, 0.02, 0.05]",
        "variable_asset": "box/box-template.urdf",
        "variable_color": "'yellow'"
    },
    {
        "variable_description": "Horizontal block to place on top of the yellow blocks",
        "variable_name": "top_block_blue",
        "variable_size": "[0.3, 0.02, 0.1]",
        "variable_asset": "box/box-template.urdf",
        "variable_color": "'blue'"
    },
]
```

8.1.2 Example of subtasks generated for the task "BuildBridge"

```
"subtask_prompt": [
    {
        "description": (
            "Add the blue base blocks on each side of the yellow "
            "supports to reinforce the structure."
        ),
        "name": "place_yellow_supports",
        "state_variables": ["base_block_yellow"],
        "initial_setup": (
            "No initial setup needed; space must be available for "
            "positioning the block on the base."
        )
    },
    {
        "description": (
            "Place the blue deck block horizontally across the top "
            "of the yellow supports to complete the bridge."
        ),
        "name": "position_blue_deck",
        "state_variables": ["top_block_blue"],
        "initial_setup": (
            "The two yellow base blocks need to be in position and "
            "parallel to each other to support the blue top block."
        )
    },
]
```

8.1.3 Example of generated factorized code for the task "BuildBridge"

```
import numpy as np
from cliport.tasks.task import Task
from cliport.utils import utils


class PlaceYellowSupports(Task):
    def __init__(self):
```

```
        super().__init__()
        self.max_steps = 15
        self.lang_template = (
            "Position two yellow support blocks parallel "
            "to each other on the bases."
        )
        self.task_completed_desc = (
            "Two yellow support blocks are positioned "
            "parallel on the bases."
        )
        self.additional_reset()

    @staticmethod
    def place_yellow_supports(task, env, yellow_support_ids):
        # Calculate the placement positions for yellow supports
        yellow_block_size = [0.1, 0.05, 0.05]
        gap_between_blocks = 0.05
        base_z = 0.025  # height of the blue base blocks
        yellow_support_height = yellow_block_size[2] / 2

        yellow_support_poses = [
            ((0.5, -0.1, base_z + yellow_support_height), (0, 0, 0, 1)
    ),
            ((0.5, 0.1, base_z + yellow_support_height), (0, 0, 0, 1))
        ]

        # Add the goal for placing the yellow supports
        task.add_goal(
            objs=yellow_support_ids,
            matches=np.array([
                [1, 0],
                [0, 1]
            ]),
            targ_poses=yellow_support_poses,
            replace=False,
            rotations=True,
            metric='pose',
            params=None,
            step_max_reward=1.0 / 3,
            language_goal=(
                "Position two yellow support blocks parallel "
                "to each other on the bases."
            )
        )

    def reset(self, env):
        super().reset(env)

        # Add yellow support blocks to the environment
        yellow_support_size = [0.1, 0.05, 0.05]
        yellow_support_urdf = "block/block.urdf"
        yellow_support_ids = []

        # We add two yellow support blocks with random poses
        for _ in range(2):
            yellow_support_pose = self.get_random_pose(
                env, yellow_support_size
            )
            yellow_support_id = env.add_object(
                yellow_support_urdf,
                yellow_support_pose,
                color='yellow'
            )
            yellow_support_ids.append(yellow_support_id)

```

```
        PlaceYellowSupports.place_yellow_supports(
            self, env, yellow_support_ids
        )


class PositionBlueDeck(Task):
    def __init__(self):
        super().__init__()
        self.max_steps = 15
        self.lang_template = (
            "Place the blue deck block horizontally across the "
            "top of the yellow supports to complete the bridge."
        )
        self.task_completed_desc = (
            "Blue deck is positioned across yellow supports."
        )
        self.additional_reset()

    @staticmethod
    def position_blue_deck(task, env, blue_deck_id, yellow_support_ids
    ):
        # Calculate the center y position between the two supports
        y_center = 0.0
        # The z position is the height of the supports plus
        # half the height of the blue deck
        z_pos = 0.025 + 0.025 + 0.05 / 2

        # The target pose for the blue deck block
        blue_deck_pose = ((0.5, y_center, z_pos), (0, 0, 0, 1))

        # Add the goal for positioning the blue deck across the yellow
     supports
        task.add_goal(
            objs=[blue_deck_id],
            matches=np.array([[1]]),
            targ_poses=[blue_deck_pose],
            replace=False,
            rotations=False,
            metric='pose',
            params=None,
            step_max_reward=1.0 / 3,
            language_goal=(
                "Place the blue deck block horizontally across the "
                "top of the yellow supports to complete the bridge."
            )
        )

    def reset(self, env):
        super().reset(env)

        # Add the blue deck block to the environment
        blue_deck_size = [0.1, 0.1, 0.05]
        blue_deck_urdf = "block/block.urdf"

        # Generate a random pose for the blue deck block
        blue_deck_pose = self.get_random_pose(env, blue_deck_size)
        blue_deck_id = env.add_object(
            blue_deck_urdf,
            blue_deck_pose,
            color='blue'
        )

        # Retrieve the IDs of the yellow supports which have been
        # placed by a previous subtask. Replace these placeholders
        # with actual IDs in a real implementation
```

```
        yellow_support_ids = [
            'yellow_support_id_1', 'yellow_support_id_2'
        ]

        PositionBlueDeck.position_blue_deck(
            self, env, blue_deck_id, yellow_support_ids
        )

class BuildBridge(Task):

    def __init__(self):
        super().__init__()
        self.max_steps = 15
        self.lang_template = (
            "Construct a bridge using two yellow blocks and :
            "three blue blocks. "
        )
        self.task_completed_desc = "done building bridge."
        self.additional_reset()

    def reset(self, env):
        super().reset(env)

        # Initialise yellow supports
        yellow_support_size = [0.1, 0.05, 0.05]
        yellow_support_urdf = "block/block.urdf"
        yellow_support_ids = []

        # Add two yellow support blocks to the environment
        for _ in range(2):
            yellow_support_pose = self.get_random_pose(
                env, yellow_support_size
            )
            yellow_support_id = env.add_object(
                yellow_support_urdf,
                yellow_support_pose,
                color='yellow'
            )
            yellow_support_ids.append(yellow_support_id)

        # Initialise blue deck
        blue_deck_size = [0.1, 0.1, 0.05]
        blue_deck_urdf = "block/block.urdf"

        # Add the blue deck block to the environment
        blue_deck_pose = self.get_random_pose(env, blue_deck_size)
        blue_deck_id = env.add_object(
            blue_deck_urdf, blue_deck_pose, color='blue'
        )

        # Call subtask methods
        PlaceYellowSupports.place_yellow_supports(
            self, env, yellow_support_ids
        )
        PositionBlueDeck.position_blue_deck(
            self, env, blue_deck_id, yellow_support_ids
        )
```

8.2 Intermediate results from Approach 2

8.2.1 Example of state variables and subtask code generated for the 1st subtask of task "Place Box In Pallet":

```
{"python_code": """
```

```
def position_first_box(self, env):
    # Place the first box on the tabletop to serve as the base of the column.

    # Step 1. Retrieve the box size, color,
    # and URDF path from the state manager
    box_size = self.state_manager.first_box_size
    box_color = self.state_manager.first_box_color
    box_urdf = self.state_manager.first_box_urdf

    # Step 2. Get a random position for the initial placement of the box
    initial_box_pose = self.get_random_pose(env, box_size)

    # Step 3. The target pose for the first box is on the tabletop.
    # We'll use the main_target_pose attribute from state manager to
    # position the box
    target_box_pose = self.state_manager.main_target_pose

    # Step 4. Add the first box to the environment at the initial pose
    box_id = env.add_object(box_urdf, initial_box_pose,
    color=utils.COLORS[box_color])

    # Step 5: Define the goal for this subtask.
    # The goal is to position the first box at the target pose
    self.add_goal(objs=[box_id],
                  matches=np.ones((1, 1)),
                  targ_poses=[target_box_pose],
                  replace=False,
                  rotations=True,
                  metric='pose',
                  params=None,
                  step_max_reward=1./3,
                  language_goal="Place the first box on the tabletop to
                  serve as the base of the column.")
""",
"object_variables": [
        "self.state_manager.first_box_size = (0.17, 0.17, 0.17)",
        "self.state_manager.first_box_color = 'blue'",
        "self.state_manager.first_box_urdf = 'box/box-template.urdf'"
]}
```

8.2.2 Example of state variables and subtask code generated for the 2nd subtask of task "Place Box In Pallet":

```
{"python_code": """
def stack_second_box(self, env):
    # Position the second box on top of the first box,
    # aligning their edges to form a stable structure.

    # Step 1: Retrieve the box size, URDF path,
    # and the pose of the first box (target pose for the second box).
    box_size = self.state_manager.second_box_size
    box_urdf = self.state_manager.second_box_urdf
    first_box_pose = self.state_manager.main_target_pose

    # Step 2: Calculate the pose for the second box.
    # It should be directly on top of the first box,
    # with the same orientation, and aligned edges.
    # We'll use utils.apply to offset the position
    # along the z-axis by the size of one box.
```

```
    second_box_target_pose = (utils.apply(first_box_pose, (0, 0, box_size[2])),
    first_box_pose[1])

    # Step 3: Get a random, collision-free initial pose for the second box.
    second_box_initial_pose = self.get_random_pose(env, box_size)

    # Step 4: Add the second box to the environment at the initial pose.
    second_box_id = env.add_object(box_urdf, second_box_initial_pose,
    color=utils.COLORS[self.state_manager.second_box_color])

    # Step 5: Define the goal for this subtask.
    # The goal is to position the second box on top of the first box.
    self.add_goal(objs=[second_box_id],
                  matches=np.ones((1, 1)),
                  targ_poses=[second_box_target_pose],
                  replace=False,
                  rotations=False,
                  metric='pose',
                  params=None,
                  step_max_reward=1./3,
                  language_goal="Position the second box on top of the first box,
                  aligning their edges to form a stable structure.")
""",
"object_variables": [
        "self.state_manager.second_box_size = (0.07, 0.07, 0.07)",
        "self.state_manager.second_box_color = 'blue'",
        "self.state_manager.second_box_urdf = 'box/box-template.urdf'"
]}
```

8.2.3 Example of state variables and subtask code generated for the 3rd subtask of task "Place Box In Pallet":

```
response =
{"python_code": """
def stack_third_box(self, env):
    # Carefully place the third box on top of the second box,
    # ensuring that the edges remain aligned and the column is stable.

    # Step 1: Retrieve the box size, URDF path, and color from the state manager.
    box_size = self.state_manager.third_box_size
    box_urdf = self.state_manager.third_box_urdf
    box_color = self.state_manager.third_box_color

    # Step 2: Calculate the target pose for the third box.
    # It should be directly on top of the second box,
    # aligned with the main target pose used for the previous boxes.
    second_box_pose = self.state_manager.main_target_pose
    third_box_target_pose = (utils.apply(second_box_pose, (0, 0, 2 * box_size[2])),
    second_box_pose[1])

    # Step 3: Get a random, collision-free initial pose for the third box.
    third_box_initial_pose = self.get_random_pose(env, box_size)

    # Step 4: Add the third box to the environment at the initial pose.
    third_box_id = env.add_object(box_urdf, third_box_initial_pose,
    color=utils.COLORS[box_color])

    # Step 5: Define the goal for this subtask.
    # The goal is to stack the third box on top of the second box.
```

```
    self.add_goal(objs=[third_box_id],
                  matches=np.ones((1, 1)),
                  targ_poses=[third_box_target_pose],
                  replace=False,
                  rotations=False,
                  metric='pose',
                  params=None,
                  step_max_reward=1./3,
                  language_goal="Carefully place the third box on
                  top of the second box, ensuring that the edges
                  remain aligned and the column is stable.")
"""",
"object_variables": [
        "self.state_manager.third_box_size = (0.07, 0.07, 0.07)",
        "self.state_manager.third_box_color = 'blue'",
        "self.state_manager.third_box_urdf = \"box/box-template.urdf\""
]}
```

8.2.4 Example of generated factorized code for the task "Place Box In Pallet"

```python
import numpy as np
from cliport.tasks.task import Task
from cliport.utils import utils


class StackBoxesColumn(Task):
    """Sequentially stack a set number of boxes into a vertical column
    ↪  on the tabletop without them toppling over."""

    def __init__(self):
        super().__init__()
        # initialize the state_manager attributes that manage all the
    ↪ state variables in this task
        StateManager = type("StateManager", (object,), {})
        self.state_manager = StateManager()
        self.state_manager.main_target_pose = ((0.5, 0.0, 0.0), (0, 0,
    ↪  0, 1,))
        self.max_steps = 15
        self.lang_template = """Sequentially stack a set number of
    ↪ boxes into a vertical column on the tabletop without them
    ↪ toppling over."""
        self.state_manager.first_box_size = (0.17, 0.17, 0.17)
        self.state_manager.first_box_color = "blue"
        self.state_manager.first_box_urdf = "box/box-template.urdf"
        self.state_manager.second_box_size = (0.07, 0.07, 0.07)
        self.state_manager.second_box_color = "blue"
        self.state_manager.second_box_urdf = "box/box-template.urdf"
        self.state_manager.third_box_size = (0.07, 0.07, 0.07)
        self.state_manager.third_box_color = "blue"
        self.state_manager.third_box_urdf = "box/box-template.urdf"
        self.additional_reset()

    def position_first_box(self, env):
        # Place the first box on the tabletop to serve as the base of
    ↪ the column.

        # Step 1. Retrieve the box size, color, and URDF path from the
    ↪  state manager
        box_size = self.state_manager.first_box_size
        box_color = self.state_manager.first_box_color
        box_urdf = self.state_manager.first_box_urdf

```

```
        # Step 2. Get a random position for the initial placement of the box
        initial_box_pose = self.get_random_pose(env, box_size)

        # Step 3. The target pose for the first box is on the tabletop.
        # We'll use the main_target_pose attribute from state manager to position the box
        target_box_pose = self.state_manager.main_target_pose

        # Step 4. Add the first box to the environment at the initial pose
        box_id = env.add_object(
            box_urdf, initial_box_pose, color=utils.COLORS[box_color]
        )

        # Step 5: Define the goal for this subtask. The goal is to position the first box at the target pose
        self.add_goal(
            objs=[box_id],
            matches=np.ones((1, 1)),
            targ_poses=[target_box_pose],
            replace=False,
            rotations=True,
            metric="pose",
            params=None,
            step_max_reward=1.0 / 3,
            language_goal="Place the first box on the tabletop to serve as the base of the column.",
        )

    def stack_second_box(self, env):
        # Position the second box on top of the first box, aligning their edges to form a stable structure.

        # Step 1: Retrieve the box size, URDF path, and the pose of the first box (target pose for the second box).
        box_size = self.state_manager.second_box_size
        box_urdf = self.state_manager.second_box_urdf
        first_box_pose = self.state_manager.main_target_pose

        # Step 2: Calculate the pose for the second box. It should be directly on top of the first box,
        # with the same orientation, and aligned edges.
        # We'll use utils.apply to offset the position along the z-axis by the size of one box.
        second_box_target_pose = (
            utils.apply(first_box_pose, (0, 0, box_size[2])),
            first_box_pose[1],
        )

        # Step 3: Get a random, collision-free initial pose for the second box.
        second_box_initial_pose = self.get_random_pose(env, box_size)

        # Step 4: Add the second box to the environment at the initial pose.
        second_box_id = env.add_object(
            box_urdf,
            second_box_initial_pose,
            color=utils.COLORS[self.state_manager.second_box_color],
        )

        # Step 5: Define the goal for this subtask. The goal is to position the second box on top of the first box.
```

```
        self.add_goal(
            objs=[second_box_id],
            matches=np.ones((1, 1)),
            targ_poses=[second_box_target_pose],
            replace=False,
            rotations=False,  # We assume the boxes have the same orientation, only position matters.
            metric="pose",
            params=None,
            step_max_reward=1.0 / 3,
            language_goal="Position the second box on top of the first box, aligning their edges to form a stable structure.",
        )

    def stack_third_box(self, env):
        # Carefully place the third box on top of the second box, ensuring that the edges remain aligned and the column is stable.

        # Step 1: Retrieve the box size, URDF path, and color from the state manager.
        box_size = self.state_manager.third_box_size
        box_urdf = self.state_manager.third_box_urdf
        box_color = self.state_manager.third_box_color

        # Step 2: Calculate the target pose for the third box. It should be directly on top of the second box,
        # aligned with the main target pose used for the previous boxes.
        second_box_pose = self.state_manager.main_target_pose
        third_box_target_pose = (
            utils.apply(second_box_pose, (0, 0, 2 * box_size[2])),
            second_box_pose[1],
        )

        # Step 3: Get a random, collision-free initial pose for the third box.
        third_box_initial_pose = self.get_random_pose(env, box_size)

        # Step 4: Add the third box to the environment at the initial pose.
        third_box_id = env.add_object(
            box_urdf, third_box_initial_pose, color=utils.COLORS[box_color]
        )

        # Step 5: Define the goal for this subtask. The goal is to stack the third box on top of the second box.
        self.add_goal(
            objs=[third_box_id],
            matches=np.ones((1, 1)),
            targ_poses=[third_box_target_pose],
            replace=False,
            rotations=False,  # We assume the boxes have the same orientation, only position matters.
            metric="pose",
            params=None,
            step_max_reward=1.0 / 3,
            language_goal="Carefully place the third box on top of the second box, ensuring that the edges remain aligned and the column is stable.",
        )

    def reset(self, env):
        super().reset(env)
```

```
139         self.position_first_box(env)
140         self.stack_second_box(env)
141         self.stack_third_box(env)
```