RoboFactory: Exploring Embodied Agent Collaboration with Compositional Constraints

Supplementary Material

A. Data Generation

In this section, we provide a detailed description of the process for effectively generating expert data. First, we elaborate on the details of RoboBrain, explaining how it generates the next subgoal and constraints. Next, we introduce the method for generating agent trajectories based on subgoals and constraints. Then, we describe the implementation of RoboChecker, an interface to integrate various constraints with data generation pipeline. Finally, we present tasks in the *RoboFactory* benchmark along with its corresponding descriptions.

RoboBrain In RoboBrain, we structure the following prompts for GPT-4o [1] to generate new subgoals based on the given information, such as task instructions, previous subgoals, and constraint violation feedback. Along with each subgoal, a set of constraints is generated, which can be categorized into three levels: logical, temporal, and spatial. These constraints are formulated as structured text to ensure that RoboChecker can accurately recognize the corresponding functions and verify whether the constraints are satisfied. The detailed description of prompts is as follows.

```
You are an AI system responsible for generating subgoals and constraints for a multi-agent robotic task.
 Your goal is to ensure that each agent receives a clearly defined subgoal while adhering to
 well-structured constraints. Constraints must be formatted correctly to enable validation and enforce
 coordination and collaboration among agents.

Key Requirements
- Generate at least one subgoal per agent based on the given task description.
- Define explicit constraints for each agent, ensuring every constraint involves at least one agent.
- Follow specific formatting rules to categorize constraints accurately.
- Ensure all constraints are clear, actionable, and unambiguous to guide robotic agents effectively.

Input Structure
- Task Instruction: "{General description of the task}"
- Global Observation: <image_global>
- Agents Observation: [<image_1>,<image_2>....,<image_n>]
- Previous Subgoals: "{Subgoals executed by each agent}"
- Constraint Violation Feedback: "{List of feedback from violated constraints, if any}"

Output Content
- A set of subgoals and constraints based on the task requirements. Each constraint should follow these
 formatting rules according to its category:
1. Logical Constraints:
   - Agent-specific condition: Agent-Specific Condition: Specifies a requirement for the behavior of a
 single agent.
       Example: "The gripper of Agent_1 must be perpendicular to {Object}."
   - Multi-agent condition: Defines a coordination rule between multiple agents.
       Example: "Agent_1 and Agent_2 must maintain a consistent gripper height.".
2. Temporal Constraints:
   - Synchronization: Specifies whether agents can perform tasks simultaneously or share the same space.
       Example: "Agent_1 and Agent_3 perform tasks simultaneously without interference.".
   - Sequence: Defines the required order of actions between agents.
       Example: "Agent_2 must complete the task before Agent_4 can begin their action.".
3. Spatial Constraints:
   - Collision avoidance: Ensures agents do not interfere with each other or the environment.
       Example: "Agents must avoid colliding with each other when moving in close proximity.".
   - Space occupancy: Specifies spatial positioning rules to prevent conflicts.
       Example: "Agent_1 should not occupy the same space as Agent_3 in the designated area.".

Output Example
{
  "Subgoals": {
    "Agent_1": "{Clear and structured subgoals for Agent_1}",
    "Agent_2": "{Clear and structured subgoals for Agent_2}",
    ...
  },
  "Constraints": {
    "Logical": [
      {
```

```
            "Agent": "Agent_1",
            "Constraint": "The gripper of Agent_1 is perpendicular to {Object}."
          },
          {
            "Agents": ["Agent_2", "Agent_3"],
            "Constraint": "Keep the gripper height consistent between Agent_2 and Agent_3 to make the camera
    remain horizontal."
          }
        ],
        "Temporal": [
          {
            "Agents": ["Agent_2", "Agent_4"],
            "Constraint": Agent_2 and Agent_4 could share the same space chronologically."
          }
        ],
        "Spatial": [
          {
            "Agent": "Agent_2",
            "Constraint": "Avoid collision between Agent_2 and other Agents."
          },
          {
            "Agent": "Agent_4",
            "Constraint": "Avoid collision between Agent_4 and other Agents."
          }
        ]
      }
    }
```


Trajectory Generation Effectively converting these conceptual subgoals into precise robotic motion trajectories remains a significant challenge for large language models. Inspired by RoboTwin [2], we define a set of motion primitives, each represented as a Python function interface. By providing specific input parameters, these primitives generate corresponding motion trajectories. For instance, the MOVE primitive inputs an agent ID and a target position. Then, it computes a trajectory based on the current position of the robotic arm to generate the appropriate motion sequence. This approach allows large language models to focus on understanding the high-level logic of action interactions while avoiding direct involvement in low-level control signal computations.

RoboChecker RoboChecker is designed to evaluate the validity and efficiency of generated motion trajectories, ensuring smooth execution while preventing collisions and inconsistencies. To achieve this, we define four key validation functions as optional interface type, each addressing a specific aspect of trajectory assessment: agent movement direction, interaction at contact points, spatial occupancy of trajectories, and the correctness of trajectory scheduling. The definitions of these functions are as follows:

- **Movement Direction Validation**: Ensures that the movement of each agent aligns with logical constraints. For instance, a robotic gripper should maintain an appropriate angle when grasping an object or adhere to a specific orientation during task execution to guarantee stable and effective interactions.
- **Contact Point Interaction Validation**: Verifies whether an interaction with object or other agents meets expected conditions. In collaborative manipulation tasks, for example, multiple agents should grasp objects at appropriate positions to maintain stability during joint handling.
- **Spatial Occupancy Validation**: Analyzes the spatial feasibility of an agent's trajectory, ensuring that it does not enter restricted zones or cause spatial conflicts. For instance, in confined environments, different agents' paths should remain non-overlapping to avoid collisions.
- **Trajectory Scheduling Validation**: Assesses whether the execution order of motion trajectories adheres to temporal constraints. This includes ensuring that actions requiring synchronization occur simultaneously and that tasks with sequential dependencies are executed in the correct order. It will also analyze whether these operations can be executed simultaneously or follow a predefined sequence along the trajectory. For example, in a task where a lid must be opened before placing an object inside, the action of "open lid" should precede the action of "place object" in the trajectory plan.

These functions take two types of inputs: the current agent's trajectory and the constraints generated by RoboBrain. The constraints produced by RoboBrain are represented in textual form. To process these constraints effectively, we construct the following prompt to match each constraint with its corresponding function, extract the relevant parameters from the constraint text, and integrate them with the agent's trajectory for validation. In Fig 1, we present the constraints of the Take Photo task along with the CheckCode generated through visual programming based on these constraint, which serves as the evaluation

protocol, bridging textual constraints with the corresponding trajectory.

```
You are an expert in robotic motion validation, responsible for ensuring that a given set of motion
 trajectories adheres to logical, spatial, and temporal constraints. Your task is to validate these
 trajectories based on predefined requirements, ensuring compliance with movement logic, spatial
 integrity, and execution order.
To achieve accurate validation, you must:
- Match each constraint to the appropriate validation function.
- Extract relevant parameters from the constraint description.
- Apply the corresponding validation rule to assess compliance.

Validation Functions and Parameter Extraction
Each constraint is assigned to a specific validation function, which extracts relevant parameters and
 applies the appropriate validation rule.
1. Movement Direction Validation: Ensures that an agent maintains the required orientation during
 interactions.
    Extracted Parameters:
        Agent ID: The agent executing the movement.
        Target Object: The object involved in the interaction.
        Required Orientation: The necessary orientation for the gripper of the agent.
    Formal Representation:
        (Agent_ID, Target_Object, Required_Orientation) -> Validate_Direction()
    Example Constraint:
        "The gripper of Agent_1 must be perpendicular to Object_A when grasping."
        (Agent_1, Object_A, perpendicular) -> Validate_Direction()
2. Contact Point Interaction Validation: Ensures that agents interact with objects or other agents at the
 designated contact points.
    Extracted Parameters:
        Agent ID: The agent performing the interaction.
        Target Object: The Object involved in the interaction.
        Contact Point: The designated interaction point (\eg, left side of the object).
    Formal Representation:
        (Agent_ID, Target_Object, Contact_Point) -> Validate_Interaction()
    Example Constraint:
        "Agent_3 must grasp Object_B at its left point."
        (Agent_3, Object_B, left) -> Validate_Interaction()
3. Spatial Occupancy Validation: Ensures that the movement of an agent does not result in spatial
 conflicts, such as entering restricted zones or colliding with other agents.
    Extracted Parameters:
        Agent IDs: The agents whose trajectories require validation.
    Formal Representation:
        (Agent_IDs) -> Validate_Spatial_Occupancy()
    Example Constraints and its parameters:
        "Agent_2 must not intersect with the trajectories of other agents."
        (Agent_2) -> Validate_Spatial_Occupancy()
4. Trajectory Scheduling Validation: Ensures that the execution order of actions adheres to temporal
 constraints,  including:
    - Sequential dependencies, where one action must precede another.
    - Synchronized execution, where multiple agents must act simultaneously.
    Extracted Parameters:
        Agent IDs: The agents involved in the scheduling constraint.
        Task Dependency Type:
            - Sequential: Specifies an ordered execution sequence.
            - Simultaneous: Requires two agents to perform actions at the same time.
    Formal Representation:
        (Agent_IDs, Task_Dependency_Type) -> Validate_Scheduling()
    Example Constraints and its parameters:
        "Agent_4 must place Object_C only after Agent_5 opens the container."
        ([Agent_5, Agent_4], "Sequential") -> Validate_Scheduling()
```

Tasks Our benchmark dataset includes 11 tasks. Table 1 presents the number of agents for each task, the task description, and the corresponding target condition. While single-agent tasks can assess the robotic arm's interaction capabilities, our primary focus is on multi-agent tasks, which evaluate the coordination and cooperation abilities between agents.

B. Experimental Setup

B.1. Training Details

We adopt the CNN-based Diffusion Policy as our base model, with a prediction horizon of 8, observation steps are set to 3, and action steps are set to 6. For the dataloader, we use a batch size of 128. The optimizer is set to `torch.optim.AdamW` with a learning rate of 1.0×10^{-4}, betas in the range of $[0.95, 0.999]$, and ϵ set to 1.0×10^{-8}. The learning rate warmup lasts

500 steps, and we train for 300 epochs for all tasks in the benchmark. The training process is conducted on a single Nvidia RTX 4090 GPU. For 150 demonstration samples with average episode length of 205, the training time is around 5 hours.

Different Training Strategies Each Franka Emika Panda robotic arm consists of seven rotational joints, with an additional one-dimensional action for the gripper (as both left and right grippers maintain the same width), resulting in an action space of dimension 8 per agent. We design four different multi-agent DP strategy modes:

- Global View and Shared Policy: All agents share a global observation that includes every agent in the environment. For a task with N agents, their actions are concatenated to form a joint action of dimension $8N$. A single model is trained using the global view and the $8N$-dimensional joint action.
- Local View and Shared Policy: Each agent has an individual local observation centered on its own action. To prevent catastrophic forgetting during training, we randomly shuffle the training data of all agents before inputting them into a shared model. A single model is trained with multiple local observations and the corresponding agent's 8-dimensional action.
- Global View and Separate Policy: All agents share a global observation, ensuring that all agents are included within it. However, each agent trains its own model to determine actions independently. The training input consists of the global view and an individual agent's 8-dimensional action.
- Local View and Separate Policy: Each agent has an individual local observation centered on its own action, with its own perspective prioritized while incorporating surrounding environmental information. Each agent trains a separate model using its local view and corresponding 8-dimensional action.

All global and local views used for training are RGB images with a resolution of 320×240. We select the fourth training strategies as the baselines for the benchmark experiments.

B.2. Evaluation Details

To ensure smooth robotic arm movements during simulations, we employ interpolating operation for action trajectories generated by the Diffusion Policy. For each task, we conduct evaluations across 100 distinct scene configurations, varying initial object placements and environmental conditions. We introduce a maximum action step limit for each task for evaluation of success rates. A failure is determined if the task is not completed within this limit. To set a reasonable threshold, we perform a warm-up test among 20 samples to estimate the average number of steps required to complete the task. The maximum action step limit is set twice this average. The success criteria for each task, including the target conditions, are detailed in Table 1.

C. Demonstrations of Benchmark

Fig. 2 demonstrates several tasks in the *RoboFactory* benchmark. We visualize the observation of key timestamps with the corresponding subgoals generated from RoboBrain across 4 tasks (*Camera Alignment*, *Place Food*, *Two Robot Stack Cube*, *Lift Barrier*).

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. arXiv preprint arXiv:2303.08774, 2023. 1

[2] Yao Mu, Tianxing Chen, Shijia Peng, Zanxin Chen, Zeyu Gao, Yude Zou, Lunkai Lin, Zhiqiang Xie, and Ping Luo. Robotwin: Dual-arm robot benchmark with generative digital twins (early version). arXiv preprint arXiv:2409.02920, 2024. 2

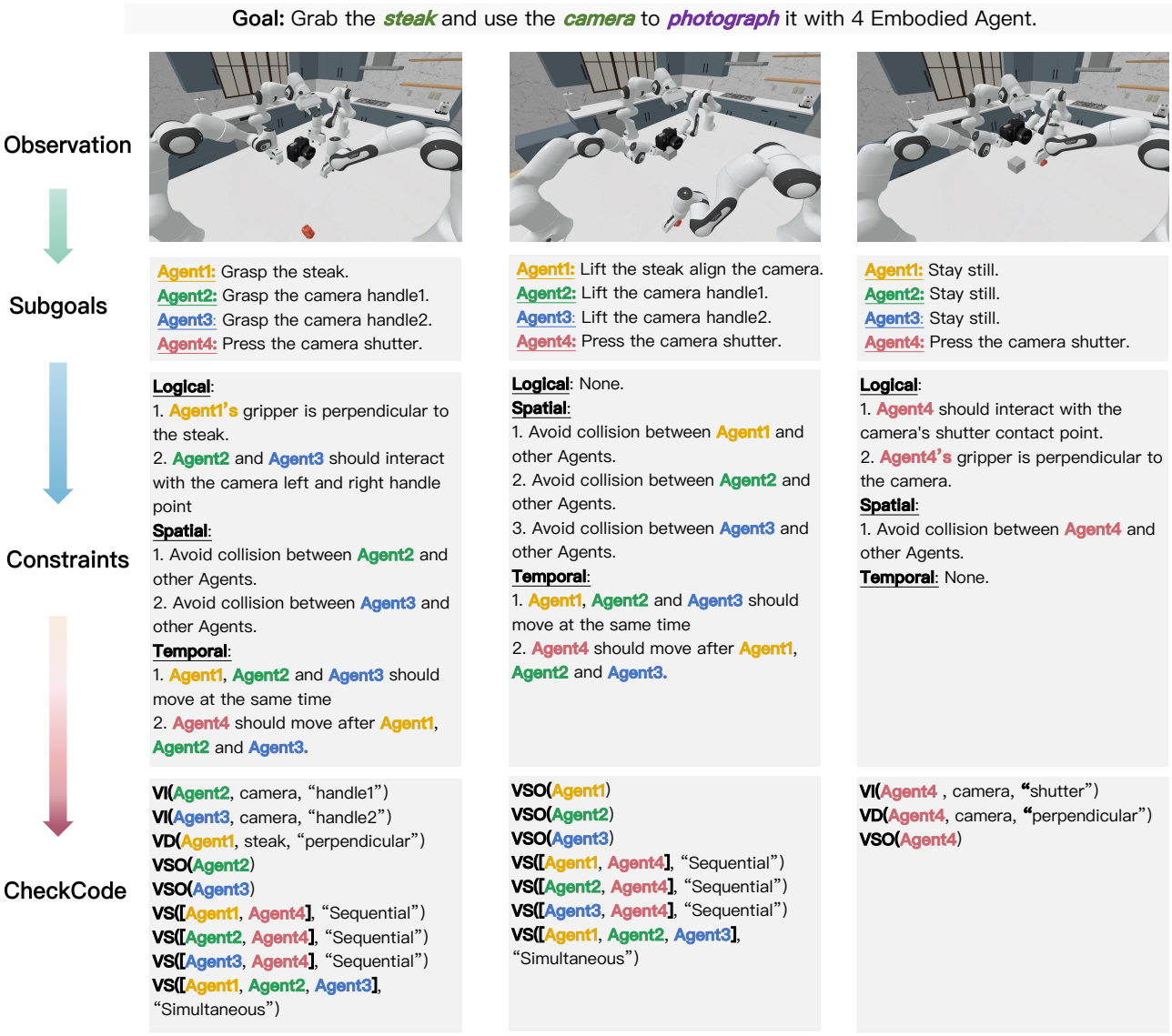

Figure 1. Demonstration of *RoboChecker* is showcased in the complete execution of the Take Photo task. By analyzing constraints, *RoboChecker* generates **CheckCode**, a composition of multiple interfaces. Specifically, VI stands for **Validate Interaction**, VD for **Validate Direction**, VSO for **Validate Spatial Occupancy**, and VS for **Validate Scheduling**. The CheckCode returns true only when all interfaces pass validation, indicating that the generated motion trajectory adheres to the compositional constraints. Otherwise, CheckCode identifies the failed interfaces and sends the feedback to *RoboBrain*.

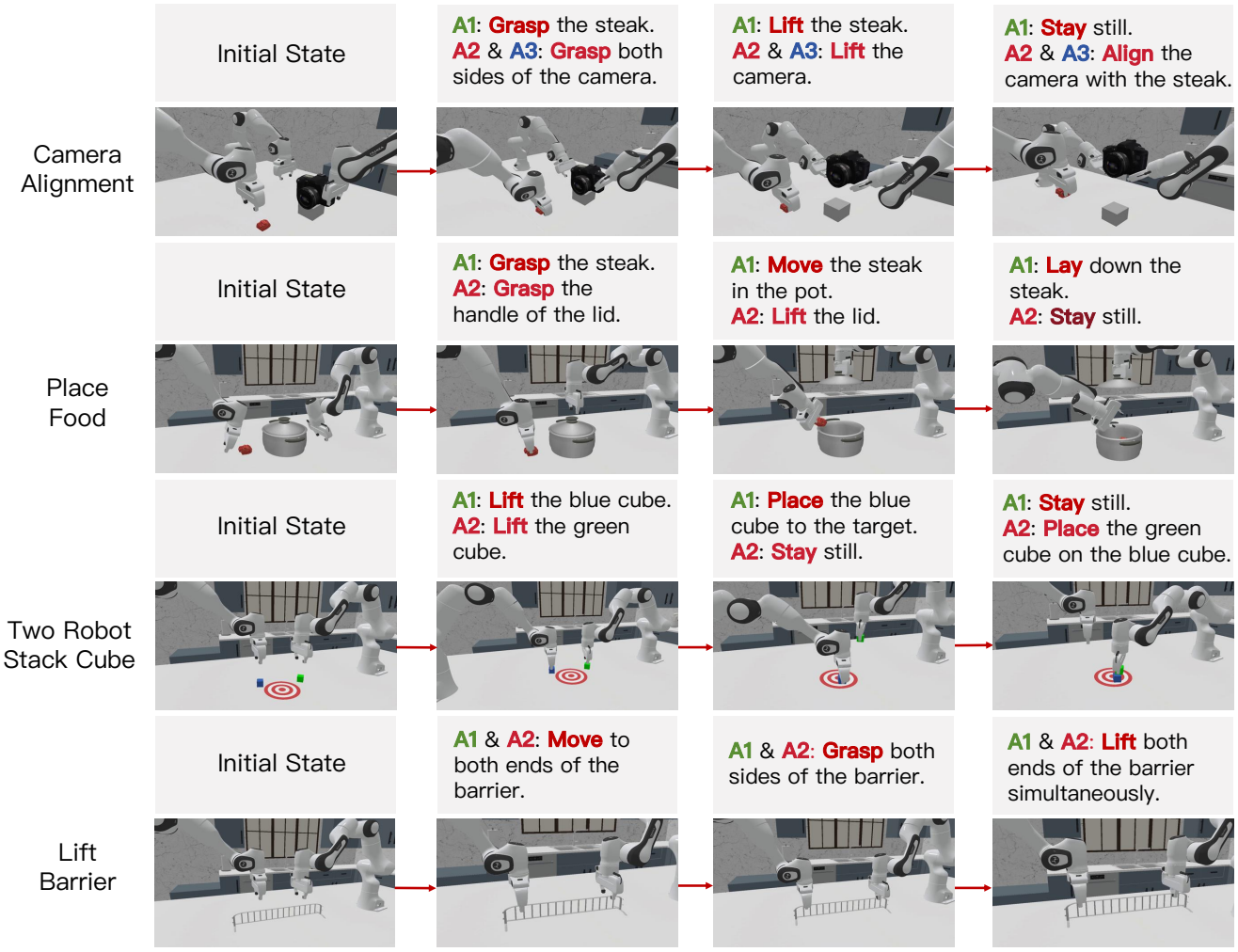

Figure 2. Demonstrations of tasks in the *RoboFactory* Benchmark. For each task, the subgoals in each timestamp are displayed in the top row, and the observation is shown in the bottom row.

Task	Agent Number	Description	Target Condition
Pick Meat	1	There is a piece of meat placed on the table. A robotic arm picks up the meat and lifts it to a specified height.	The height of the meat reaches a predefined threshold.
Stack Cube	1	A blue cube and a red cube are placed on the table. A robotic arm picks up the blue cube and places it on top of the red cube.	The distance between the blue and red cubes is within a threshold, with the blue cube positioned at a greater height than the red cube.
Strike Cube	1	A hammer and a cube are placed on the table. A robotic arm first identifies an optimal grasping position to pick up the hammer, then moves it to a suitable position to strike the cube.	The hammerhead is positioned directly above the cube within a predefined distance threshold.
Lift Barrier	2	A long barrier is placed on the table. Two robotic arms simultaneously grasp both ends of the barrier and lift it to a specified height.	The barrier is elevated to the specified height while maintaining stability.
Pass Shoe	2	A shoe is placed on the table. One robotic arm grasps the shoe and passes it to the other one, which then places it at the target location.	The distance between the shoe and the target location is within a predefined threshold.
Place Food	2	A pot and a kind of food are placed on the table. One robotic arm lifts the pot's lid, while the other picks up the food and places it inside the pot.	The food is placed inside the pot, with the distance between the food and the center of the pot being within a predefined threshold.
Two Robots Stack Cube	2	A blue cube and a red cube are placed on the table. A robotic arm picks up the blue cube to a specified position, while the other places the red cube on top of it.	The blue cube is within the specified threshold distance from the target position. The distance between the blue and red cubes remains within a defined threshold, with the red cube positioned at a greater height than the blue cube.
Camera Alignment	3	A camera and an object are placed on the table. One robotic arm picks up the object to a specified position. The other two robotic arms grasp both sides of the camera and align it to the object.	The camera reaches a specified height, and the object is placed at the designated position that aligns with the camera.
Three Robots Stack Cube	3	A blue cube, a red cube and a green cube are placed on the table. One robotic arm picks up the blue cube to a specified position. Another arm places the red cube on top of the blue one. The last arm places the green cube on top of the red one.	The blue cube is positioned within the specified target range. Additionally, the red cube is successfully placed on top of the blue cube, and the green cube is positioned atop the red cube.
Take Photo	4	A camera and an object are placed on the table. One robotic arm picks up the object and places it to a specified position. Another two robotic arms grasp both sides of the camera and align it to the object. The last robotic arm clicks the shutter.	The camera reaches a specified height, and the object is placed at the designated position that aligns with the camera. Additionally, the distance between the end effector of the last robotic arm and the camera's shutter is within a certain threshold.
Long Pipeline Delivery	4	A shoe is placed on the table. Three robotic arms grasp the shoe and pass it to the next robotic arm. The last robotic arm places the shoe to a specified position.	The distance between the shoe and the target location is within a predefined threshold.

Table 1. Task Descriptions for the *RoboFactory* Benchmark