Code-BT: A Code-Driven Approach to Behavior Tree Generation for Robot Tasks Planning with Large Language Models

Siyang Zhang[1,2], **Bin Li**[2,*], **Jingtao Qi**[2], **Xueying Wang**[2], **Fu Li**[2], **Jianan Wang**[1,2], **En Zhu**[1] and **Jinjing Sun**[2]

[1]College of Computer Science and Technology - National University of Defense Technology
[2]Intelligent Game and Decision Lab (IGDL)

Abstract

Behavior trees (BTs) provide a systematic and structured control architecture extensively employed in game AI and robotic behavior control, owing to their modularity, reactivity, and reusability. Nonetheless, manual BTs design requires significant expertise and becomes inefficient as task complexity increases. Recent automation technologies have avoided manual work, but often have high application barriers and face challenges in adapting to new tasks, making it difficult to easily configure them to specific requirements. Code-BT introduces a novel approach that utilizes large language models (LLMs) to automatically generate BTs, representing the task planning process as the process of coding and organizing sequences. By retrieving control flow information from the generated code, BTs can be efficiently constructed to address the complexity and diversity of task planning challenges. Rather than relying on manual design, Code-BT uses task instructions to guide the selection of relevant APIs, and then systematically assembles these APIs into modular code to align with the BTs structure. Finally, action sequences and control logic are extracted from the generated code to construct the BTs. Our approach not only ensures the automation of BTs generation but also guarantees the scalability and adaptability for long-term tasks. Experimental results demonstrate that Code-BT substantially improves LLM performance in BTs generation, achieving improvements ranging from 16.67% to 29.17%.

1 Introduction

Recently, the application of Large Language Models (LLMs) in robotics has attracted considerable attention. Models from the GPT series, in particular, have demonstrated remarkable capabilities in analogical reasoning, emergent intelligence [Brown *et al.*, 2020], and code comprehension and generation [Jiang *et al.*, 2024a]. Several investigations have successfully leveraged LLMs for task planning in robotics. This includes translating natural language instructions into Planning Domain Definition Language (PDDL) [Aeronautiques *et al.*, 1998], replacing traditional planners with LLMs to generate task plans [Zhou *et al.*, 2024b], and utilizing LLMs to produce Python planning programs for specific domains and tasks [Silver *et al.*, 2024]. Moreover, some studies have introduced world-state information to enable dynamic expansion and updates of object properties [Chen *et al.*, 2023]. These works demonstrate that LLMs possess strong domain knowledge and common-sense priors, which are essential for many of planning tasks.

However, despite the promising prospects of these studies, several inherent challenges have been revealed. A significant challenge is the development of domain knowledge libraries, which is still laborious and time-intensive, particularly when it necessitates frequent manual revisions. For example, a knowledge base predefined for task planning in a factory environment must be manually modified whenever there are changes in processes or equipment, otherwise, the generated task plans will fail. Another challenge is the limited generalization and flexibility across different task types. While LLMs can generate task plans for specific domains, current methods often struggle to generalize these plans to other types of tasks. For instance, a planning approach designed specifically for gear assembly may not be flexible or adaptable enough to handle household tasks. These issues highlight the gap between existing methods and the actual requirements of robotic task planning.

BTs are a control architecture that allows agents to flexibly switch between different tasks [Colledanchise and Ögren, 2018], and are widely used in robot control [Colledanchise and Natale, 2021][Wen *et al.*, 2024a]. Compared to traditional task representation methods such as finite state machines and decision trees, behavior trees offer advantages in modularity and flexibility. However, the manual design of BTs requires complex domain knowledge and tedious experiential details, which results in increased time and cost. This has spurred some explorations into automatic BTs generation methods based on LLMs. [Izzo *et al.*, 2024][Lykov *et al.*, 2023] construct BTs datasets to fine-tune models to improve the direct output of BTs, while [Ao *et al.*, 2024] and [Chen *et al.*, 2024] combine planning for BTs generation, using LLMs to transform goals into planning languages, and then using BTs expansion algorithms to construct behavior trees. How-

*Corresponding author.

ever, these methods either require substantial costs in collecting or constructing datasets for fine-tuning, or building domain knowledge libraries for each individual task scenarios, which lacks generalization and flexibility.

Code represents a fundamental medium for the realization and deployment of BTs [1] [2]. A fact is that LLMs have been extensively trained on high-quality code data, which has concurrently enhanced their code generation capabilities, now a key metric of their performance. This has inspired us to use code generation for the automatic design of BTs. Compared to directly generating BTs or planning languages for robotic tasks, LLMs have been extensively trained on high-level programming languages like Python. Code's primary benefit is its generality and its ability to encapsulate complex logic with expressive power. Utilizing the strong code generation abilities of LLMs, we intend to create a comprehensive and adaptable framework for BTs generation, facilitating effective task planning in diverse robotic scenarios.

We introduce a code driven approach for Behavior Trees generation **Code-BT**. Code-BT consists of a three-phase framework that utilizes code to assist LLMs in planning BTs for solving robotic tasks. This code generation approach is bottom-up, emulating the programmer's methodology of crafting subtask functions prior to tackling the complete task within the code. In the first phase, the LLM selects relevant code APIs according to the task instructions. Subsequently, the LLM organizes these APIs into modular solutions, while adhering to constraint rules to regulate the control flow structure. Given the inherent complexity, this process is iterative and feedback-driven, as producing an accurate and reliable solution in a single attempt is rarely feasible. Finally, we introduce a parsing algorithm to extract control flow from the generated code, mapping it into a Behavior Tree representation.

The principal contributions of this paper are as follows.

- We propose Code-BT, the first framework leveraging LLMs' code generation capabilities to automate Behavior Trees creation for robotic tasks.
- We design targeted rules to enforce structural constraints on code generated by LLMs, ensuring alignment with the BTs. To improve reliability, we incorporate an iterative feedback mechanism, organizing APIs in a bottom-up manner and preserving the modularity of BTs.
- We devise an innovative algorithm leveraging Abstract Syntax Trees (ASTs) to accurately extract control flow from Python code and map it into an equivalent XML-formatted BTs representation.
- Compared to generalizable methods like Chain-of-Thought (CoT) and code generation approaches, our method significantly improves BTs generation pass rate and quality across diverse task scenarios.

[1] https://py-trees.readthedocs.io/en/devel/
[2] https://www.behaviortree.dev/

2 Related Work

2.1 Planning and Learning Methods for BTs Generation

The automatic generation of BTs is a significant research direction in robotics and artificial intelligence. Some studies have focused on planning-based approaches to ensure the generated tree structures are both logical and complete. For instance, [Cai *et al.*, 2021] introduced a STRIPS-based BTs expansion method that constructs coherent and comprehensive behavior trees using state space formulations. [Gugliermo *et al.*, 2023] combined decision trees and logical decomposition to create interpretable behavior trees. [Rovida *et al.*, 2017] utilized a PDDL planner to initialize task sequences, followed by hierarchical task network (HTN) decomposition to obtain hierarchical trees, which were then reorganized and optimized to generate behavior trees. [Chen *et al.*, 2024] convert task instructions into first-order logic formulas and using the Optimal Behavior Tree Expansion Algorithm (OBTEA) to construct minimum-cost behavior trees.

Moreover, learning-based BTs generation methods have demonstrated their potential in representing patterns within BTs structures effectively. [Potteiger and Koutsoukos, 2023] employed reinforcement learning to develop an Evolving Behavior Tree (EBT) model. [Iovino *et al.*, 2021] used genetic programming to learn the structure of behavior trees, addressing robotic tasks in unpredictable environments. [Jain *et al.*, 2024] adopted a learning from demonstration approach to derive behavior trees from task demonstrations.

While planning-based methods ensure the reliability of the generated behavior trees, they often require substantial time and effort to manually write PDDL domain and problem files. Additionally, learning-based methods are computationally expensive, and they have limited generalization capabilities.

2.2 Integrating LLMs for Robot Program Generation

Code generation aims to automatically produce executable code from high-level abstract descriptions, such as natural language or requirement specifications, thereby reducing the workload of manual coding. In recent years, large language models (LLMs) have demonstrated significant potential in natural language processing and code generation. Recent advancements have seen the introduction of numerous LLMs, including DeepSeek v2[Liu *et al.*, 2024] , Qwen 2.5 [Yang *et al.*, 2024], and GPT-4[Achiam *et al.*, 2023], which have achieved state-of-the-art (SOTA) results in various code-related tasks, such as code generation [Wen *et al.*, 2024b] and code summarization [Zhou *et al.*, 2024c].

Program code serves as one of the primary interfaces for specifying robotic planning and control commands, playing a crucial role in implementing robotic functionalities and tasks[Yang *et al.*, 2022]. Consequently, LLM-generated code can serve as an expressive means to control robots. For example, [Vemprala *et al.*, 2024] demonstrated the ability of LLMs to generate deployable code based on task descriptions and available API information. [Liang *et al.*, 2023] showed how LLMs can recursively generate undefined functions, enabling

the development of more complex code policies for robots. Additionally, [Burns *et al.*, 2024] illustrated the potential of LLMs to generate relevant code for high-precision manipulation tasks within the correct action space.

These studies highlight the significant potential of code generation techniques in the field of robotics. However, current methods primarily focus on low-level action control and lack the capability to handle long-term complex tasks.

2.3 Integrating LLMs for BTs Generation

The strong language understanding and reasoning capabilities of large language models (LLMs) have led researchers to explore innovative methods for directly constructing BTs using LLMs. For example, [Cao and Lee, 2023] designed a Phase-Step Prompt method that converts natural language descriptions into robot actions, thereby building complete behavior trees. In [Zhou *et al.*, 2024a], a keyword parser based on BERT was developed to parse sequential steps into an initial behavior tree, which is subsequently dynamically expanded. Some studies have used LLMs to directly generate XML-formatted behavior trees [Izzo *et al.*, 2024], [Mower *et al.*, 2024]. [Ao *et al.*, 2024] employs LLMs as planners by providing initial states, goal descriptions, and PDDL knowledge, using different context learning methods to generate behavior trees.

Unlike the aforementioned works, our approach parses behavior trees from generated high-level programming languages rather than from rough natural language descriptions. This method incorporates more precise API and control flow information and does not rely on additional training or predefined planning domain knowledge.

3 Method

In this section, we introduce the workflow and design details of our method, including API selection, code generation, and control flow to BTs mapping. Our method framework is illustrated in Figure 1. To better leverage the code generation capabilities of LLMs for generating usable behavior trees for robotic tasks, we have constructed a pipeline consisting of the following phases.

3.1 API Selection

This phase ultimately yields a set of callable API functions, which fall into two categories: condition checks and action executions.

We instruct the LLM to combine the task description with the API library to select the necessary and relevant APIs for the task, and output the results in JSON format. Given the diversity of platforms, scenarios, and tools in the robotics domain, these APIs are not limited to any specific platform or library. The API library provides a diverse set of robotic actions, including essential descriptive names and corresponding functional annotations, while hiding the intricate details of the functions, similar to the definition in [Vemprala *et al.*, 2024].

The advantage of this method is that it significantly reduces the complexity of constructing the function library, requiring only the function names and their corresponding descriptions, which enhances its generality. Whether manually or automatically created, there is no need for specific, strict implementations or runtime details, making it easy to construct an API library. Furthermore, additional parameters or detailed comments can be added to the function headers if necessary.

Specifically, regarding the API function library, when a new scenario or task arises, relevant APIs can be easily incorporated into the library in code form, without the need for detailed function implementations. This increases the generalization across different scenarios. Additionally, the highly abstracted nature of the APIs in the library facilitates easy updates, modifications, or deletions.

3.2 Rule-Constrained Multi-Round Code Generation

Decision trees have been shown to be equivalent to behavior trees when action nodes always return "running" [Colledanchise and Ögren, 2016], as illustrated in Figure 2. In practice, decision trees can be easily implemented using "if-else" branching structures in code. However, behavior tree nodes may return either "success" or "failure." Therefore, simple "if-else" structures are insufficient to fully represent behavior trees. There are already open-source libraries, such as pytrees, that assist in implementing behavior trees, but these libraries have not achieved the same level of popularity as ROS in the robotics operating system domain. Consequently, we aim to generate more general-purpose code to represent behavior trees without relying on complex predefined libraries.

In previous code generation tasks, benchmarks like Humaneval[Chen *et al.*, 2021] and MBPP[Austin *et al.*, 2021] have been used to evaluate the performance of models in generating code. However, there is currently no benchmark available for evaluating the generation capability of personalized code, such as robot functions or specific coding conventions. Notably, the work on SCoT (Structure Chain of Thought) [Li *et al.*, 2023] explored the method of incorporating structured intermediate steps into code generation by leveraging sequence, branching, and looping structural information, which effectively improved the quality of code generation.

Inspired by decision trees and SCoT, we aim to effectively extract control structure information (control flow) from code to generate BTs. Based on the set of available API functions obtained from the previous step, we leverage the LLMs to organize these API functions through code generation. Specifically, the code generation step has three core settings:

Constraint Rules:

1. In addition to using the `if-else` keywords, we allow the use of `not` to evaluate the execution results of nodes. The `return` keyword is used to indicate the result, which can only be failure or success, represented by boolean values (`False` or `True`).
2. The control flow within code blocks follows two logical patterns:If the condition is not met (`if not`), execute the corresponding function; otherwise, return `True`.If the condition is met or the operation is successful (`if`), execute the corresponding function; otherwise, return

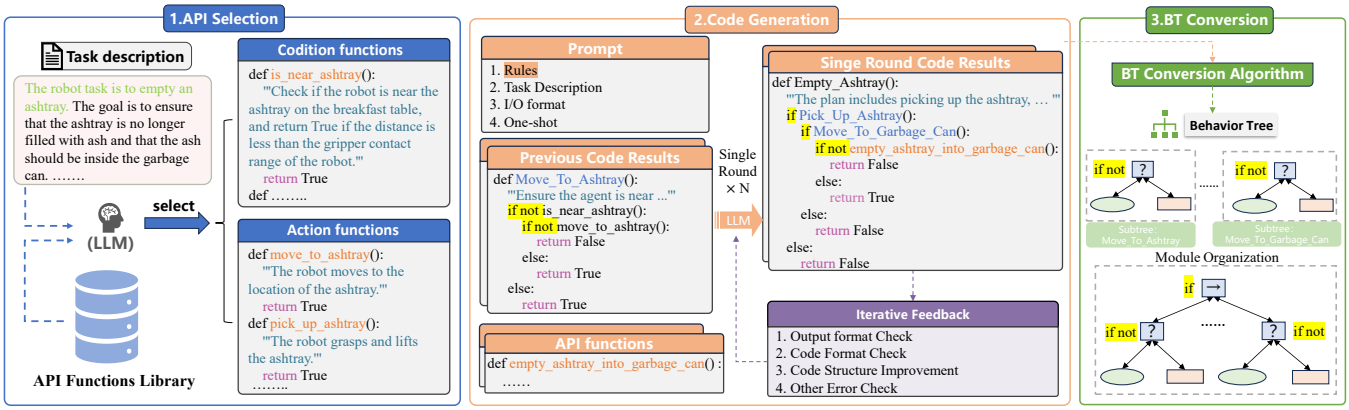


Figure 1: Overview of Code-BT. **Phase 1: API Selection**, where the task description for the agent (emptying an ashtray) is provided. The LLM selects appropriate condition functions and action functions from the library. **Phase 2: Code Generation**, where prompts are crafted based on constraint rules, task descriptions, I/O formats, and one-shot examples. This step is iterative, with each response leveraging historical code results. This step integrates a feedback process to enhance instruction adherence, optimize code structure, and reduce errors. **Phase 3: Control Flow to BTs Mapping** (BT conversion), we transform the generated code into a behavior tree. The conversion algorithm maps the control flow in the code to subtrees in the behavior tree. Each node in the behavior tree corresponds to a specific action or condition, with control nodes determined by keywords such as "if (not)-else" and "return".

`False`.If all steps are successful, return `True`; otherwise, return `False`.

3. Each code block can only use nested `if(not)` statements: Specifically, nested `if(not)` keywords are used to determine whether to proceed with the execution of the next function. Within a code block, only one if-else structure with the lowest indentation level is allowed, and this if-else structure contains all the control structure information of a behavior tree (subtree).

These constraint rules ensure that the generated code adheres to a structured and predictable control flow. By limiting the complexity of the if-else structures and using boolean values for results, the rules facilitate the generation of reliable and efficient behavior trees.

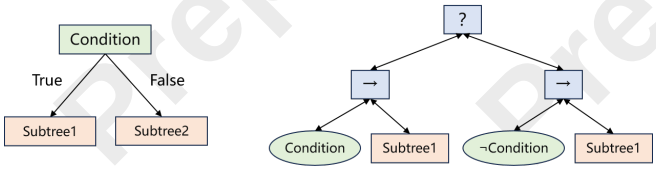


Figure 2: Decision Tree to Behavior Tree Conversion

Multi-Round Generation: For tasks involving multiple APIs, requiring the generation of code to organize all APIs in a single response increases the challenge for LLMs. Therefore, we allow the LLM to use only a subset of the API collection in a single response to generate a code block. The multi-round generation process corresponds to the generation of multiple subtrees. Each round of generation involves organizing subtrees and nodes, which fully leverages the modular advantages of behavior trees. This approach aligns the code generation process with the bottom-up modular design process of behavior trees. For example, when a robot completes the task of cleaning an ashtray, it involves stages such as moving, judging, picking up, and cleaning. During the generation process, APIs are organized into these modules, and these modules are then used to generate the final code.

Iterative Feedback: We mandate that the LLM generates responses in JSON format. To ensure compliance with constraint rules and optimize the code structure, we introduce a feedback mechanism that mitigates errors and hallucinations. This mechanism primarily includes: Output Format Validation: Verify that the output adheres to the JSON format, ensuring the completeness and accuracy of keys and values. Syntax Validation: Ensure that the generated code complies with the syntax rules of the target programming language. Rule Validation: Verify that the generated code structure aligns with the predefined rules. If the code fails validation, the system provides specific feedback to guide the model in regenerating the code. This process continues until all rules are satisfied or the maximum feedback limit is reached.

In summary, compared to approaches that either utilize sequential API functions (as illustrated in Figure 3) or adopt decision tree control architectures, our solution incorporates a richer set of keywords and control flow information. By embedding rule constraints into the code generation process, our framework subsequently maps the generated code to behavior trees for robotic control, ensuring alignment with the modular design principles of behavior trees.

3.3 Control Flow to BTs Mapping

The goal of this phase is to convert the generated Python code into XML format representing a behavior tree. Compared to the intricate details of the pytrees library, such as the writing and updating processes of nodes to the blackboard, we focus more on how the generated code organizes available APIs according to control flow to accomplish tasks. We have designed a conversion algorithm that leverages Abstract Syntax Trees (ASTs) to parse the control flow in the code and map it to the structure of a behavior tree. The main steps of

Chatgpt for robotics

```
def get_ingredients():
    # Go to the fridge and get the eggs.
    go_to_location(locate ("fridge"))
    pick_up("eggs")

    # Go to the counter and get the bowl.
    go_to_location(locate ("bowl"))
    pick_up("bowl")

    # Crack the eggs into the bowl.
    use_item("eggs")
    ……
```

Code-BT

```
def get_ingredients():
    """ Go to the fridge and get the eggs. """
    if go_to_location(locate ("fridge")):
        if pick_up("eggs"):
            if go_to_location(locate ("bowl")):
                if pick_up("bowl"):
                    if use_item("eggs") :
                        return True
                    else:
                        return False
                else:
                    return False
    ……
```

Figure 3: Comparison of Two Different Code Structures for an Example Robotic Task.

the conversion algorithm are as follows:

1. **Abstract Syntax Tree (AST) Parsing:** Initially, we use Python's ast module to parse the generated code and generate the corresponding AST. AST is a tree-like structure that clearly extracts the syntactic structure of the code.
2. **Control Flow Extraction:** Based on the AST parsing, we extract control flow information from the code. Control flow information primarily includes conditional statements (such as if-else) and function calls. This control flow information will be used to construct nodes and subtrees in the behavior tree.
3. **Behavior Tree Node Mapping:** We map conditional statements in the code to condition nodes in the behavior tree, and API function calls to action nodes(Encapsulated functions are mapped to subtrees). For control flow structures (such as nested if-else statements), we map them to control flow nodes in the behavior tree, such as Sequence Nodes or Fallback Nodes.
4. **Behavior Tree Construction:** After completing the node mapping, we organize these nodes according to the structural rules of behavior trees to form a complete behavior tree, which is then output in XML format.

4 Experiments

4.1 Experimental Setup

Benchmarks: Our experiments are conducted using the BEHAVIOR-1K benchmark[Li *et al.*, 2024], a comprehensive simulation framework designed for robotics. The tasks in BEHAVIOR-1K are characterized by their long-term nature and reliance on complex manipulation skills, presenting significant challenges for state-of-the-art robotic learning solutions. We selected 24 tasks from this benchmark to ensure a diverse representation of task scenarios and categories. For each task, we extracted their corresponding task descriptions and developed an API function library necessary for all tasks, ensuring comprehensive coverage of all required operations.

These API functions are highly abstracted but include detailed functional annotations, as illustrated in Figure 1.

LLMs: Code-BT is designed as a versatile framework, independent of any specific LLM. For our experiments, we selected four prominent LLMs: DeepseekV2.5-Chat [DeepSeek-AI, 2024], GPT-4o[Hurst *et al.*, 2024], Qwen2.5-72B-Instruct[Yang *et al.*, 2024] (denoted as qwen2.5-72b), and Llama3.1-70B-Instruct [Dubey *et al.*, 2024] (denoted as llama3.1-70b). These models, representing a range of parameter sizes and including both powerful open-source (Deepseek) and closed-source (GPT-4o) models, ensure diverse results and accessibility.

Compared Methods: For the task of generating behavior trees, there is currently no specific prompting method designed to directly improve performance. Therefore, we compared our method with general CoT (Chain of Thought) method, as well as code generation methods such as SCoT and CodeCoT (Code Chain of Thought). Our method and all baselines use one-shot examples and are constructed using the same problem setup. All comparison methods have been adjusted to fit the behavior tree generation process.

- **CoT**[Wei *et al.*, 2022]: The standard CoT method, which generates intermediate reasoning steps before solving language reasoning tasks, has been proven effective in improving the performance of LLMs in commonsense and arithmetic reasoning.
- **SCoT**: SCoT is a novel code generation prompting technique that requires the LLM to use program structures (i.e., sequence, branching, and looping structures) to construct the CoT, resulting in SCoT prompts. Compared to CoT, SCoT prompts explicitly constrain the LLM to think from the perspective of source code, further enhancing the LLM's performance in code generation.
- **CodeCoT**: Referring to the [Jiang *et al.*, 2024b] setup, we manually wrote detailed annotations for the behavior tree XML in the example, enabling the LLM to use the annotations as intermediate steps to guide the XML generation process.

Evaluation Metrics: BTs evaluation usually relies on manual inspection, where experts assess the correctness, functionality, and performance of the BTs. However, this approach presents several challenges: it is time-consuming, susceptible to human error, and often inconsistent due to the inherent subjectivity of manual judgment. In addition, traditional simulation tools, such as Gazebo[Koenig and Howard, 2004], are commonly used for BTs evaluation. While these simulators accurately model physical processes and environments, they come with high computational demands and inefficiencies. Moreover, these tools often require considerable effort to adapt to different scenarios, making them unsuitable for evaluating BTs across diverse task environments.

To overcome these limitations, we employ two automated evaluation methods to effectively demonstrate the improvements introduced by Code-BT: the simulator and the evaluator. The simulator allows BTs to be tested in virtual environments, ultimately assessing whether the BT can successfully complete the task. Meanwhile, the evaluator systematically evaluates various aspects of the BTs, such as its structure and logic, and provides a corresponding score.

- **LLM-based Simulator**: This metric is derived from the validation results of a behavior simulator [Wang *et al.*,

Model	Step	Methods	Passed (%)	Evaluation
gpt-4o	Two-Step	**Ours**	**83.33**	**81.57**
		Code CoT	58.33	79.38
		CoT	66.67	80.79
		SCoT	66.67	81.21
	One-Step	Code CoT	25.00	79.28
		CoT	54.17	80.42
		Scot	33.33	80.35
deepseek-2.5	Two-Step	**Ours**	**87.50**	77.88
		Code CoT	45.83	77.72
		CoT	41.67	81.39
		SCoT	37.50	80.19
	One-Step	Code CoT	70.83	80.68
		CoT	45.83	**82.28**
		SCoT	66.67	82.32
llama3.1-70b	Two-Step	**Ours**	**66.67**	**79.67**
		Code CoT	29.17	77.92
		CoT	16.67	78.17
		SCoT	8.33	78.92
	One-Step	Code CoT	41.67	78.31
		CoT	50.00	78.25
		Scot	29.17	78.42
qwen2.5-72b	Two-Step	**Ours**	**79.17**	**80.42**
		Code CoT	16.67	79.85
		CoT	16.67	79.54
		SCoT	20.83	79.97
	One-Step	Code CoT	50.00	79.29
		CoT	45.83	80.00
		SCoT	50.00	78.56

Table 1: Results of Comparative Experiments on Benchmarks.

2024]. The simulator rapidly executes behavior trees and categorizes the outcomes into three types: Good, Bad Logic, and Unreachable. Behavior trees that yield a "Good" result are deemed successful, and we calculate the proportion of such successful outcomes.

- **LLM-based Evaluator**: We utilize Deepseek to evaluate the behavior trees, providing a comprehensive score based on five critical dimensions: logic and task completion, structure and readability, efficiency and performance, error handling and robustness, and scalability and maintainability. The total score is 100.

4.2 Method Performance

In our experiments, we divided the comparative methods into two distinct settings. *One-step:* The LLM directly receives the task description and outputs the XML-formatted behavior tree. *Two-step:* The LLM first generates code with format constraints, which is then converted into a behavior tree by the LLM. These settings allow us to systematically evaluate the effectiveness of each approach in generating behavior trees for robotic tasks. Table 1 summarizes the final experimental results. We measured the extent to which our method enhances the reasoning and planning capabilities of LLMs. For the baseline methods, we provided the robot task descriptions and available API functions as prompts to the LLMs, enabling them to integrate planning through multi-round generation. During the generation process, errors were identified, and feedback was provided for improvement. To ensure a fair comparison, we included a few demonstration examples. Across all models, Code-BT demonstrated significant performance improvements. The results indicate that Code-BT substantially increases the pass rate of behavior trees, showcasing its overall capability to generate correct behavior trees. Specifically, in terms of pass rate, our method achieved a 16.67% improvement over the highest-performing baseline method (DeepSeekV2.5, Llama3.1-70B-Instruct, GPT-4o), and a 29.17% improvement (Qwen2.5-72B-Instruct).

Our method showed slightly lower improvements in LLM Evaluation, which can be attributed to the inherent limitations of LLMs. Even for humans, it is challenging to provide precise scores for behavior trees. Nevertheless, it is noteworthy that LLM Evaluation still reflects the overall effectiveness of the experiment. The suboptimal performance of the baseline methods can be attributed to the lack of training data on behavior trees for LLMs, as well as the limited effectiveness of intermediate steps in enhancing the planning of behavior tree tasks. In contrast, the application of our method brought about significant improvements. This demonstrates that the proposed Code-BT effectively addresses the challenges faced by existing LLMs in using behavior trees for task planning.

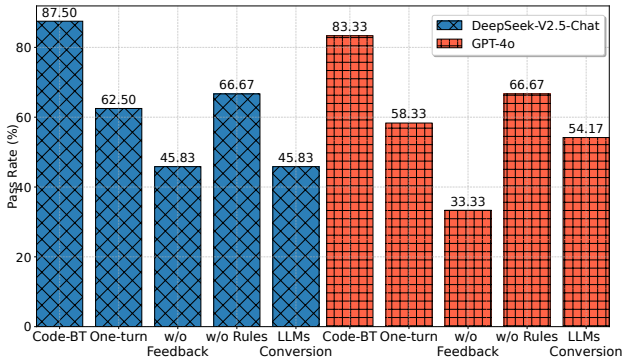
4.3 Ablation Study

In this section, we validate the effectiveness of several core mechanisms in Code-BT by exploring the following questions:

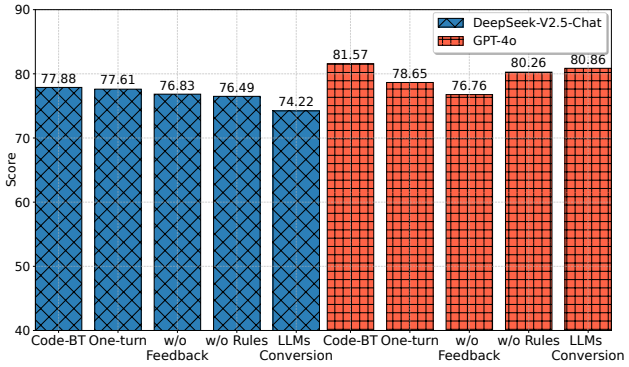
- Can the multi-round generation setting reduce the reasoning difficulty for LLMS when organizing complex tasks, thereby helping them produce better code responses?
- Can rule-constrained code generation align the control flow in the code with the control logic in behavior trees?
- Can iterative feedback correct various errors in the output of LLMs?
- Can the conversion algorithm accurately and reliably transform code into XML-formatted behavior trees?

We observe that DeepSeek and GPT-4o outperform other models, so we choose them as the main LLMs for ablation studies. We explore several ablation settings to reflect the specific effect of each mechanism. The results of the ablation study are detailed in Figure 4.

Effectiveness of Multi-Round Generation: We compared multi-round generation with one-turn generation. In multi-round generation, the model does not need to consider all functions at once. For example, in a pool-cleaning task, the model is allowed to output only the code block for "obtaining a brush" in one round, whereas one-turn generation may require the model to output the entire code for the task at once. We found that one-turn generation led to a significant performance drop 23% (DeepSeek and GPT-4o), which is attributed to the complexity of the task. Organizing all API information to write code in a single round is challenging for



(a) Pass Rate of DeepSeek and GPT-4o.



(b) Score of DeepSeek and GPT-4o.

Figure 4: The ablation experiment results on DeepSeek-V2.5-Chat and GPT-4o. "w/o Rules" represents the removal of format constraint prompts, "w/o Feedback" represents the ablation of iterative feedback, and "LLMs conversion" indicates replacing the conversion algorithm with LLMs.

LLMs, leading to a substantial increase in logical errors and hallucinations. In contrast, multi-round generation reduces the reasoning difficulty for LLMs by allowing them to focus on completing "subtasks," thereby improving the reliability of the generated results.

Effectiveness of Rule Constraints: Rule constraints play a central role in the Code-BT framework, not only standardizing the structure of code generation but also ensuring that the generated code can be seamlessly converted into the control logic of behavior trees. To explore the effectiveness of rule constraints, we analyzed the following aspects:

Without rule constraints, the code generated by LLMs tends to be more free-form, potentially leading to unclear control flow or logic that does not align with behavior trees. For example, the generated code may contain non-standard return results or redundant conditional judgments, which increases the difficulty of subsequent conversion to behavior trees and may lead to logical errors in the behavior trees.

In contrast, rule-constrained code generation restricts the structure of the code (e.g., using return statements for code block execution results, limiting the complexity of if-else structures), allowing the LLMs to focus on necessary operations and ensuring that the generated code has clear control flow information. Experimental results show that rule-constrained code generation significantly improves the pass rate and evaluation scores of behavior trees.

Effectiveness of Iterative Feedback: The iterative feedback mechanism is another core component of the Code-BT framework. When generating code, LLMs may produce format errors, logical errors, or outputs that do not comply with rule constraints. Without a feedback mechanism, these errors cannot be corrected in time, leading to lower-quality behavior trees. Therefore, the iterative feedback mechanism is designed to optimize code quality through multi-round generation and feedback, ensuring that the generated code meets task requirements.

To validate the effectiveness of iterative feedback, we conducted comparative experiments with and without iterative feedback. The results show that iterative feedback significantly improves the quality of the output (with a 41.67% improvement in pass rate on DeepSeek and a 50% improvement on GPT-4o).Through multi-round generation and feedback, the LLM can gradually optimize the code structure, generating behavior trees that better align with task requirements. Although multiple feedback rounds are allowed, most tasks achieve satisfactory results within 3 feedback iterations, indicating that the iterative feedback mechanism can quickly identify and correct issues during the generation process.

Effectiveness of the Conversion Algorithm: Mapping the generated Python code into XML-formatted behavior trees is the final phase in the Code-BT framework and a crucial step in ensuring that the generated results can be directly applied to robotic tasks. To validate the effectiveness of the conversion algorithm, we compared two methods: using the LLMs to convert Python code into XML BTs and using an AST-based parsing algorithm to convert code into behavior trees.

The LLM-based conversion method struggles to accurately capture the control logic and action sequences in the code when dealing with complex control flows, leading to a 41.67% drop in performance on deepseek and a 29.16% drop on gpt-4o. In contrast, the conversion algorithm is designed to precisely parse the control flow information in the code. Experimental results show that the conversion algorithm can accurately map the control flow to the structure of behavior trees (nodes and subtrees), and generate behavior trees with clear logic.

5 Conclusion

This paper introduces Code-BT, a framework for the automatic generation of Behavior Trees leveraging the code generation capabilities of LLMs. Code-BT addresses the challenges of designing behavior trees for complex robotic tasks by integrating the code generation prowess of LLMs with the modular design principles of behavior trees. Experimental results demonstrate that Code-BT significantly outperforms existing baseline methods, particularly in complex tasks. Future research could focus on further optimizing Code-BT's generation mechanisms, expanding its application scenarios, and exploring more powerful LLM support to meet the demands of even more complex tasks.

References

[Achiam *et al.*, 2023] 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.

[Aeronautiques *et al.*, 1998] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins Sri, Anthony Barrett, Dave Christianson, et al. Pddl— the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[Ao *et al.*, 2024] Jicong Ao, Fan Wu, Yansong Wu, Abdalla Swikir, and Sami Haddadin. Llm as bt-planner: Leveraging llms for behavior tree generation in robot task planning. *arXiv preprint arXiv:2409.10444*, 2024.

[Austin *et al.*, 2021] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[Brown *et al.*, 2020] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[Burns *et al.*, 2024] Kaylee Burns, Ajinkya Jain, Keegan Go, Fei Xia, Michael Stark, Stefan Schaal, and Karol Hausman. Genchip: Generating robot policy code for high-precision and contact-rich manipulation tasks. *arXiv preprint arXiv:2404.06645*, 2024.

[Cai *et al.*, 2021] Zhongxuan Cai, Minglong Li, Wanrong Huang, and Wenjing Yang. Bt expansion: a sound and complete algorithm for behavior planning of intelligent robots with behavior trees. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6058–6065, 2021.

[Cao and Lee, 2023] Yue Cao and CS Lee. Robot behavior-tree-based task generation with large language models. *arXiv preprint arXiv:2302.12927*, 2023.

[Chen *et al.*, 2021] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[Chen *et al.*, 2023] Siwei Chen, Anxing Xiao, and David Hsu. Llm-state: Expandable state representation for long-horizon task planning in the open world. *arXiv preprint arXiv:2311.17406*, 2023.

[Chen *et al.*, 2024] Xinglin Chen, Yishuai Cai, Yunxin Mao, Minglong Li, Wenjing Yang, Weixia Xu, and Ji Wang. Integrating intent understanding and optimal behavior planning for behavior tree generation from human instructions. *arXiv preprint arXiv:2405.07474*, 2024.

[Colledanchise and Natale, 2021] Michele Colledanchise and Lorenzo Natale. On the implementation of behavior trees in robotics. *IEEE Robotics and Automation Letters*, 6(3):5929–5936, 2021.

[Colledanchise and Ögren, 2016] Michele Colledanchise and Petter Ögren. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Transactions on robotics*, 33(2):372–389, 2016.

[Colledanchise and Ögren, 2018] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[DeepSeek-AI, 2024] DeepSeek-AI. Introducing deepseek v2.5 blog. https://platform.deepseek.com/api-docs/zh-cn/news/news0905/, 2024.

[Dubey *et al.*, 2024] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[Gugliermo *et al.*, 2023] Simona Gugliermo, Erik Schaffernicht, Christos Koniaris, and Federico Pecora. Learning behavior trees from planning experts using decision tree and logic factorization. *IEEE Robotics and Automation Letters*, 8(6):3534–3541, 2023.

[Hurst *et al.*, 2024] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[Iovino *et al.*, 2021] Matteo Iovino, Jonathan Styrud, Pietro Falco, and Christian Smith. Learning behavior trees with genetic programming in unpredictable environments. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4591–4597. IEEE, 2021.

[Izzo *et al.*, 2024] Riccardo Andrea Izzo, Gianluca Bardaro, and Matteo Matteucci. Btgenbot: Behavior tree generation for robotic tasks with lightweight llms. *arXiv preprint arXiv:2403.12761*, 2024.

[Jain *et al.*, 2024] Aayush Jain, Philip Long, Valeria Villani, John D Kelleher, and Maria Chiara Leva. Cobt: Collaborative programming of behaviour trees from one demonstration for robot manipulation. *arXiv preprint arXiv:2404.05870*, 2024.

[Jiang *et al.*, 2024a] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[Jiang *et al.*, 2024b] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024.

[Koenig and Howard, 2004] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. Ieee, 2004.

[Li *et al.*, 2023] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 2023.

[Li *et al.*, 2024] Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Wensi Ai, Benjamin Martinez, et al. Behavior-1k: A human-centered, embodied ai benchmark with 1,000 everyday activities and realistic simulation. *arXiv preprint arXiv:2403.09227*, 2024.

[Liang *et al.*, 2023] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023.

[Liu *et al.*, 2024] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[Lykov *et al.*, 2023] Artem Lykov, Maria Dronova, Nikolay Naglov, Mikhail Litvinov, Sergei Satsevich, Artem Bazhenov, Vladimir Berman, Aleksei Shcherbak, and Dzmitry Tsetserukou. Llm-mars: Large language model for behavior tree generation and nlp-enhanced dialogue in multi-agent robot systems. *arXiv preprint arXiv:2312.09348*, 2023.

[Mower *et al.*, 2024] Christopher E Mower, Yuhui Wan, Hongzhan Yu, Antoine Grosnit, Jonas Gonzalez-Billandon, Matthieu Zimmer, Jinlong Wang, Xinyu Zhang, Yao Zhao, Anbang Zhai, et al. Ros-llm: A ros framework for embodied ai with task feedback and structured reasoning. *arXiv preprint arXiv:2406.19741*, 2024.

[Potteiger and Koutsoukos, 2023] Nicholas Potteiger and Xenofon Koutsoukos. Safe explainable agents for autonomous navigation using evolving behavior trees. In *2023 IEEE International Conference on Assured Autonomy (ICAA)*, Jun 2023.

[Rovida *et al.*, 2017] Francesco Rovida, Bjarne Grossmann, and Volker Krüger. Extended behavior trees for quick definition of flexible robotic tasks. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6793–6800. IEEE, 2017.

[Silver *et al.*, 2024] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Kaelbling, and Michael Katz. Generalized planning in pddl domains with pretrained large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 20256–20264, 2024.

[Vemprala *et al.*, 2024] Sai H Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. *IEEE Access*, 2024.

[Wang *et al.*, 2024] Jianan Wang, Bin Li, Xueying Wang, Fu Li, Yunlong Wu, Juan Chen, and Xiaodong Yi. Besimulator: A large language model powered text-based behavior simulator. *arXiv preprint arXiv:2409.15865*, 2024.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[Wen *et al.*, 2024a] Shanghua Wen, Wendi Wu, Ning Li, Ji Wang, Shaowu Yang, Chi Ben, and Wenjing Yang. Auction-based behavior tree evolution for heterogeneous multi-agent systems. *Applied Sciences*, 14(17):7896, 2024.

[Wen *et al.*, 2024b] Yeming Wen, Pengcheng Yin, Kensen Shi, Henryk Michalewski, Swarat Chaudhuri, and Alex Polozov. Grounding data science code generation with input-output specifications. *arXiv preprint arXiv:2402.08073*, 2024.

[Yang *et al.*, 2022] Shuo Yang, Xinjun Mao, Yao Lu, and Yong Xu. Towards a behavior tree-based robotic software architecture with adjoint observation schemes for robotic software development. *Automated Software Engineering*, 29(1):31, 2022.

[Yang *et al.*, 2024] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[Zhou *et al.*, 2024a] Haotian Zhou, Yunhan Lin, Longwu Yan, Jihong Zhu, and Huasong Min. Llm-bt: Performing robotic adaptive tasks based on large language models and behavior trees. *arXiv preprint arXiv:2404.05134*, 2024.

[Zhou *et al.*, 2024b] Zhehua Zhou, Jiayang Song, Kunpeng Yao, Zhan Shu, and Lei Ma. Isr-llm: Iterative self-refined large language model for long-horizon sequential task planning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2081–2088. IEEE, 2024.

[Zhou *et al.*, 2024c] Ziyi Zhou, Mingchen Li, Huiqun Yu, Guisheng Fan, Penghui Yang, and Zijie Huang. Learning to generate structured code summaries from hybrid code context. *IEEE Transactions on Software Engineering*, 2024.