Middle Code Prediction: Enhancing Code Generation for Uncommon Programming Languages in Robotics

Zixi Jia[*] jiazixi@mail.neu.edu.cn
Hongbin Gao ghb2323@163.com
Hexiao Li lihexiao2021@163.com
Faculty of Robot Science and Engineering, Northeastern University, Shenyang, China

Editors: Vu Nguyen and Hsuan-Tien Lin

Abstract

Generating executable code through natural language instructions to drive robotic movements is considered a crucial step towards achieving embodied intelligence. However, in the robotics domain, the scarcity of programming language data often necessitates manually encapsulating high-level APIs to enable Large Language Models(LLMs) to predict code correctly, which is time-consuming and incomplete. Therefore, this paper proposes a three-stage Middle Code Prediction(MCP) scheme, by injecting appropriate prompts at different stages, the LLMs can shift towards predicting middle code that it understands more easily. This middle code can then be converted into the final code through specific scripts, accomplishing the task of generating code in uncommon programming languages automatically and without the need for manually encapsulating high-level APIs. We tested our approach on Hospital Item Transport Dataset(HITD) and found that MCP could improve the mean accuracy of various baseline models to varying degrees, with an overall increase of 31%, while also enhancing the noise resistance of fine-tuned models. We conducted real-world experiments on industrial robotic arms, verifying the feasibility of MCP in scenarios with no API and partial API encapsulation. The method proposed in this paper provides a guideline for code generation in uncommon programming languages within the context of LLMs. Our experimental dataset is available at `https://github.com/Ghbbbbb/MCP`.

Keywords: Code generation, Large Language models, Robot, Prompt learning

1. Introduction

Large Language Models(LLMs) have made significant progress in generating code for common programming languages, as seen with models like CodeT5 Wang et al. (2021), CodeLlama Rozière et al. (2023), and WizardCoder Luo et al. (2024), which are often trained on vast amounts of data. However, a recent study Pigott (2020) indicates that there are currently 8,495 programming languages, with the top 20 languages accounting for 95% of project repositories. This implies the existence of hundreds of domain-specific programming languages with scarce or hard-to-collect training data, making direct training of LLMs time-consuming and prone to overfitting Yang et al. (2024). Therefore, it is crucial to find a method that enables code generation for uncommon programming languages.

In the field of robotics, different robot brands use unique programming languages, which are typical examples of uncommon programming languages. Automatically generating code from natural language instructions to drive these robots is one of the hottest topics today.

[*] Corresponding author

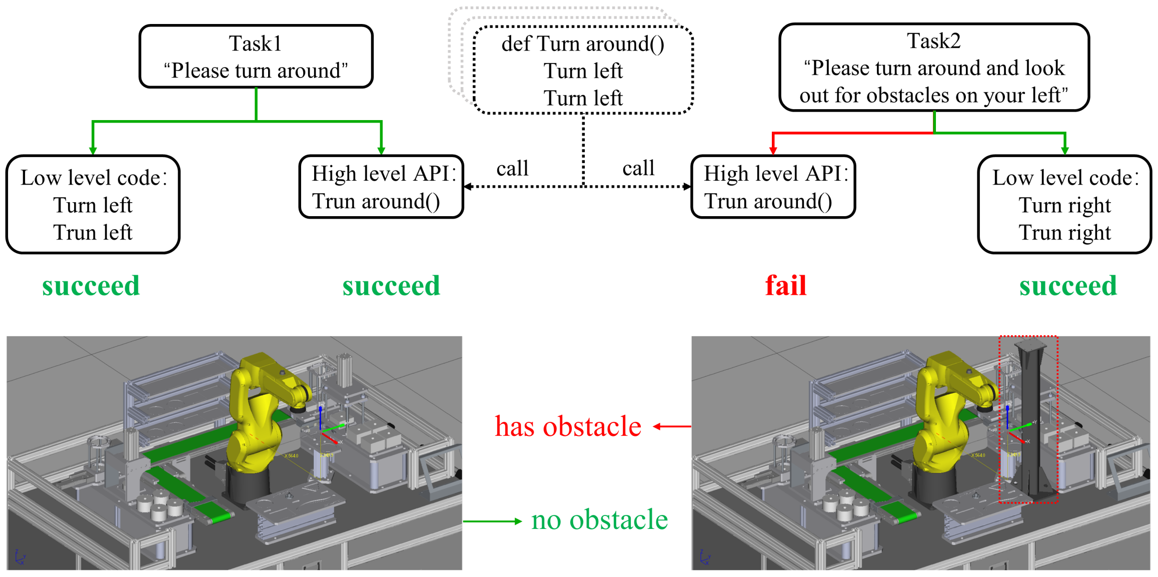


Figure 1: Comparison of the high-level APIs and the low-level code across different tasks and scenarios. Low-level code can be adapted to a wider range of tasks but often results in increased code length.

Traditional methods mainly focus on disambiguation and semantic extraction Wang et al. (2017); Liu and Zhang (2018); Weigelt et al. (2020), involving complex semantic parsing work and generally having poor generalization. Recently, with the rise of LLMs, it is possible for LLMs to act as the brain of the robot. By describing environmental information and specifying APIs in the prompts, LLMs can autonomously plan and generate corresponding code based on the task Chen and Huang (2023); Bärmann et al. (2023), thereby driving the robot to perform tasks. Although this approach solves the problem of semantic parsing, it involves the encapsulating of high-level APIs Liang et al. (2023), which often requires the involvement of experienced encapsulating engineers, making it still unfriendly to users. Additionally, encapsulated high-level APIs may reduce task applicability. As shown in Figure 1, due to the absence of a high-level API suitable for Task 2 in the API library, the model selects the closest-semantic API, leading to a motion failure. Conversely, if low-level code is predicted, the robot can perform more complex and comprehensive movements through different combinations of low-level codes, but this will increase the burden of code prediction on the LLMs. Our paper focuses on the latter, aiming to propose a scheme for generating long code in uncommon programming languages.

How can LLMs maintain strong long-code generation capabilities for uncommon programming languages? Inspired by how humans write pseudo-code: tackling a programming problem by directly writing its final code is challenging, but writing its pseudo-code significantly reduces the task's difficulty. We applied this idea to LLMs and proposed a three-stage Middle Code Prediction(MCP) scheme. By having LLMs predict middle code(a form combining natural language and programming language, similar to pseudo-code), they can accomplish more complex tasks. Specifically, in Stage 1, we manually wrote high-quality *[instructions, final code, simplified rules, middle code]* quadruples. In Stage 2, these quadruples are prompted to LLMs, allowing them to learn specific simplification rules so that for new given tasks, LLMs can generate the corresponding middle code and code extension scripts for the final code. In Stage 3, we use few-shot prompting to let LLMs predict the middle code for new tasks. The predicted middle code can then be translated into the final code using the code extension scripts in a one-to-one mapping process. Thus, through this method, LLMs only need to predict the middle code for a given task, reducing the burden of code prediction and potentially enabling long-code generation for uncommon programming languages.

We constructed a dataset composed of low-level code, namely the Hospital Item Transport Dataset(HITD) (Section 4.1), on which the main experiments of this paper will be conducted. We found that MCP can improve the task mean accuracy of various baseline models to varying degrees, with an overall improvement of 31% (Section 4.3.1). Furthermore, an ablation study on the middle code revealed that the Sequential and SubSummary modules are key to the correct prediction of middle code by LLMs (Section 4.3.2). The combination of MCP with fine-tuned models resulted in the highest task performance, demonstrating resistance to noise interference (Section 4.3.3).

The contributions of this paper can be summarized in three aspects:

- We introduce Middle Code Prediction(MCP), a scheme that allows LLMs to adapt to various low-level code prediction tasks through the injection of prompts at different stages. This approach offers strong robustness and generalization without the need for manually encapsulating APIs.

- We create the Hospital Item Transport Dataset(HITD), composed of natural language instructions and low-level codes. This dataset includes tasks of three different difficulty levels and serves as an effective benchmark for evaluating the low-level code generation capabilities of LLMs.

- We validate the feasibility of MCP in scenarios with no API and partial API encapsulation, achieving the process from natural language input to final code generation and robotic arm movement. This provides a viable solution for robotic arm control in the context of LLMs.

2. Related work

2.1. Robot code generation

In the field of robot code generation, traditional work involves complex tasks of semantic disambiguation and extraction, which have poor generalization capabilities when handling different types of tasks Thomas et al. (2022). However, with the rise of LLMs and prompt learning, guiding LLMs to generate code through prompts to drive robot actions has become a viable approach. Vemprala et al. (2024) proposed a method using ChatGPT and prompt engineering to guide robots in completing various tasks, observing the robot's task execution in the virtual environment AirSim. Kannan et al. (2023) introduced the SMART-LLM model, which feeds the output from each stage into the GPT-4 model to achieve a multi-robot collaborative framework. Singh et al. (2023) proposed a programmatic LLMs prompt structure called ProgPrompt, which can generate plans in different contexts, robot functions, and tasks, demonstrating high success rates in the VirtualHome household tasks.

However, the aforementioned methods generally rely on pre-existing high-level APIs and LLMs with massive parameters for code prediction. In contrast, our "middle code" based approach effectively avoids the need for encapsulated high-level APIs by directly predicting long low-level code. Additionally, by simplifying tasks to the prediction of "middle code", it is expected that inference deployment on small parameter LLMs can be achieved.

2.2. In-Context Learning and Chain-of-Thought

In-Context Learning is a type of machine learning that aims to enable LLMs to understand tasks and perform task reasoning through prompts Dong et al. (2022). Currently, many works on robot code generation employ In-Context Learning Liang et al. (2023); Arenas et al. (2024). This lightweight approach eliminates the need for retraining and fine-tuning, effectively reducing computational costs and allowing LLMs to quickly adapt to new tasks.

Chain-of-Thought(CoT) emphasizes thinking. For LLMs, focusing more on the thought process of generating answers during prediction often yields better results. Wei et al. (2022) were the first to propose using CoT to tackle complex reasoning tasks. They used a few-shot approach to provide the model with data in the form of *[input, CoT, output]*, allowing the model to automatically reason solution steps and answers, thereby improving model performance. Kojima et al. (2022) found LLMs to be effective zero-shot reasoners by simply adding "Let's think step by step" before each answer. Wang et al. (2023) inspired by human intuition in solving problems where multiple methods can yield correct answers but incorrect answers differ, proposed a coherent method to enhance model accuracy through diversity. Zhou et al. (2023) introduced a least-to-most prompting strategy, where LLMs first decompose complex problems into several sub-problems based on prompts, and then recursively solve each sub-problem, thereby solving complex problems step by step.

Our approach is similar to Zhou et al. (2023) least-to-most method. However, unlike their approach, ours does not explicitly decompose the problem and solve each sub-problem separately. Considering that code generation tends to follow a specific pattern for a given scenario and to minimize time costs, we merge these two processes into one, implicitly allowing LLMs to learn this process through few-shot prompting.

3. Middle Code Prediction

In this section, we will detail the method of Middle Code Prediction(MCP). Our approach consists of three stages, as illustrated in Figure 2: (1)Simplified Rule Interpretation: Manually writing high-quality quadruples. (2)Knowledge Extraction in LLMs: Predicting the middle code and extension script for new given task. (3)Code Generation in LLMs: Generating the final code for new given task.

3.1. Simplified Rule Interpretation

If LLMs were directly tasked with predicting final code, the generated language would be highly abstract code, which is difficult for LLMs to understand without fine-tuning in the domain. Conversely, using a form that combines natural language with code, namely middle code, can to some extent bridge this gap. Here, we define middle code as a combination of natural language reasoning and simplified code. Natural language reasoning, akin to CoT,

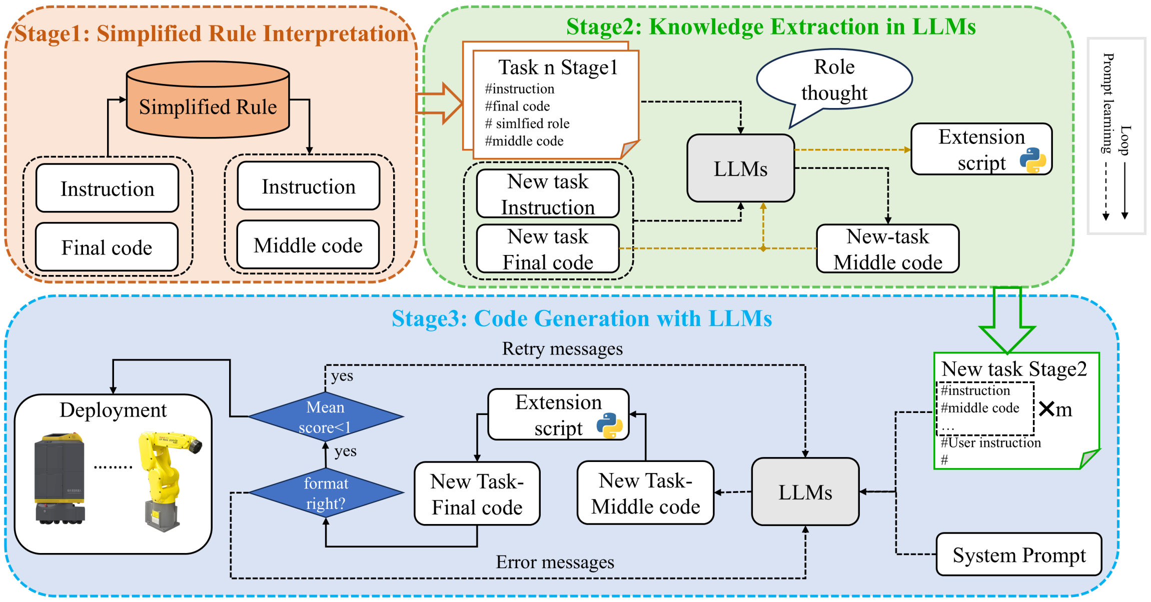


Figure 2: Three-stage Framework of MCP.

aims to help the model better understand relevant tasks, while simplified code represents a streamlined version of the final code, aiming to reduce the model's burden in code prediction.

Inspired by the work of Wang et al. (2017), we introduce the concept of Simplification Rules (SR), which transform final code into middle code. SR include but is not limited to the following guidelines:

1) Repetitive code segments can be replaced with key information represented by numbers or letters.

2) The order of replacements should match the original appearance order in the code.

3) Replacements should facilitate easy extraction in script languages.

4) During replacement, summarization of tasks can be performed.

5) Results after replacement should be simplified as much as possible while ensuring no loss of information.

Taking the Hospital Item Transport Dataset(HITD) as an example, we present a case of simplifying from final code to middle code, as shown in Figure 3. Specifically, given a *Taski* consisting of input $\left(x^{(i)},\ y_F^{(i)}\right)$, where $x^{(i)}$ represents the user command for the *ith* task and $y_F^{(i)}$ represents the final code for the *ith* task, we apply simplified rules to transform it into middle code:

$$\left(x^{(i)},\ y_M^{(i)}\right) = f_{sr}^{(i)}\left(x^{(i)},\ y_F^{(i)}\right) \tag{1}$$

where $y_M^{(i)}$ represents the middle code for the *ith* task, which consists of natural language reasoning (in black font) and simplified code (in red font) from Figure 3. $f_{sr}^{(i)} \subseteq SR$ denotes

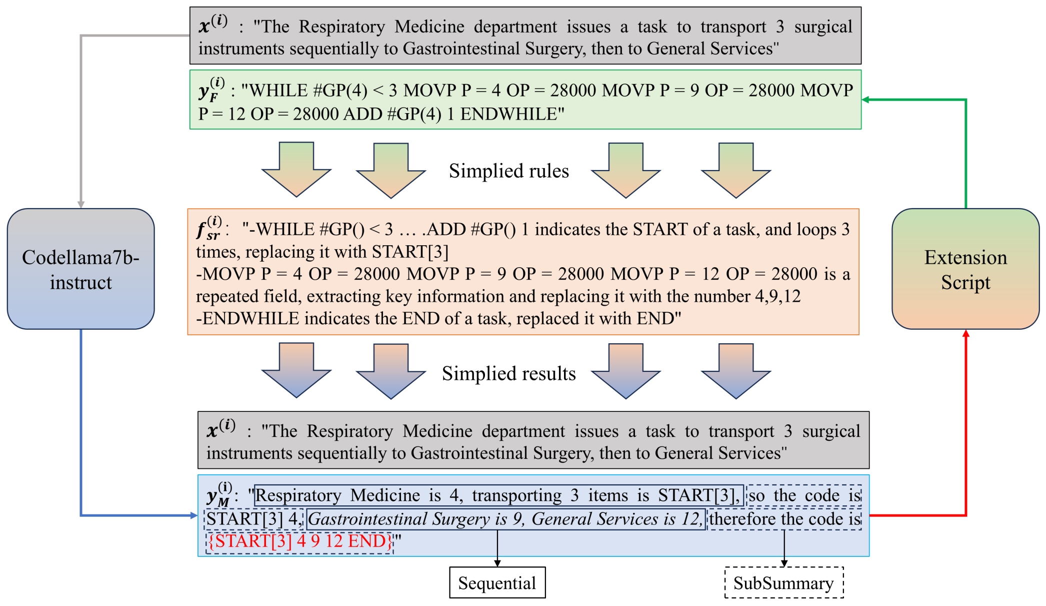

$x^{(i)}$: "The Respiratory Medicine department issues a task to transport 3 surgical instruments sequentially to Gastrointestinal Surgery, then to General Services"
$y_F^{(i)}$: "WHILE #GP(4) < 3 MOVP P = 4 OP = 28000 MOVP P = 9 OP = 28000 MOVP P = 12 OP = 28000 ADD #GP(4) 1 ENDWHILE"
Simplied rules
$f_{sr}^{(i)}$: "-WHILE #GP() < 3 … .ADD #GP() 1 indicates the START of a task, and loops 3 times, replacing it with START[3]
-MOVP P = 4 OP = 28000 MOVP P = 9 OP = 28000 MOVP P = 12 OP = 28000 is a repeated field, extracting key information and replacing it with the number 4,9,12
-ENDWHILE indicates the END of a task, replaced it with END"
Simplied results
Codellama7b-instruct
Extension Script
$x^{(i)}$: "The Respiratory Medicine department issues a task to transport 3 surgical instruments sequentially to Gastrointestinal Surgery, then to General Services"
$y_M^{(i)}$: "Respiratory Medicine is 4, transporting 3 items is START[3], so the code is START[3] 4, Gastrointestinal Surgery is 9, General Services is 12, therefore the code is START[3] 4 9 12 END"
Sequential
SubSummary

Figure 3: Transformation from Final Code to Middle Code under Simplified Rules

the simplification rules applied to the *ith* task. Thus, *Taski* can be redefined as:

$$Taski = \left(x^{(i)}, y_F^{(i)}, y_M^{(i)}, f_{sr}^{(i)}\right) \qquad i = 1, 2, ..., n \qquad (n < 10) \tag{2}$$

where we set $n < 10$, considering the constraints of LLMs' context window.

To enhance LLMs' reasoning capabilities, our natural language reasoning consists primarily of two parts: Sequential and SubSummary. Sequential ensures that the order of replacements aligns with the original user instructions, as depicted by the solid boxes in Figure 3 *[Respiratory Medicine - Gastrointestinal Surgery - General Services]*, following the guideline 2). SubSummary involves summarizing the replaced information, as illustrated by the dashed boxes in Figure 3, following the guideline 4).

Through the above steps, we can collect quadruples corresponding to n tasks. These high-quality quadruples will be used in subsequent prompting of the LLMs.

3.2. Knowledge Extraction in LLMs

In the stage2, the LLMs learns the mapping relationship between n high-quality quadruples through Few-shot prompting. This enables LLMs, given input $\left(x^{(n+1)}, y_F^{(n+1)}\right)$ for the *n+1th* task, to think the corresponding simplification rule $f_{sr}^{(n+1)}$ and generate middle code $y_M^{(n+1)}$. Additionally, prompts are used to guide the LLMs in generating extension scripts f_{ES} from middle code to final code:

$$y_M^{(n+1)} = f_{LLM}\left[\left(Task1,\ Task2,\ ...\ ,Taskn\right)\ ,f_{sr}^{(n+1)}\ \left(x^{(n+1)},\ y_F^{(n+1)}\right)\right] \tag{3}$$

$$f_{ES}^{(n+1)}\left(y_M^{(n+1)}\right) = y_F^{(n+1)} \tag{4}$$

where f_{LLM} is the process of In-context Learning from LLMs, and f_{ES} is the corresponding extension script. It can extract the final simplified code(red code part in Figure 3) from the middle code and expand it into the final code with extension script, which is a one-to-one mapping process.

Through knowledge extraction in the LLMs, construction of middle code and extension scripts for new tasks is achieved, laying the foundation for subsequent LLMs code generation.

3.3. Code Generation in LLMs

In the stage3, we still use In-Context Learning to prompt LLMs to predict the middle code for new tasks. Different from stage2, the samples here are some examples of the same task, such as $\left(x_i^{(n+1)},\ y_{M_i}^{(n+1)}\right)$, which represents the user instructions and middle code of the *ith* sample in the *n+1th* task. The prediction process of the LLMs is as follows:

$$y_{M_{m+1}}^{(n+1)} = f_{LLM}\left(\left[\left(x_1^{(n+1)},\ y_{M_1}^{(n+1)}\right),\ \left(x_2^{(n+1)},\ y_{M_2}^{(n+1)}\right),...,\left(x_m^{(n+1)},\ y_{M_m}^{(n+1)}\right)\right],\ x_{m+1}^{(n+1)}\right) \quad (5)$$

For the *m+1th* user instruction, the corresponding middle code $y_{M_{m+1}}^{(n+1)}$ can be predicted, which can be converted into the final code $y_{F_{m+1}}^{(n+1)}$ through Equation (4), thereby achieving the process from user instruction input to final code generation.

We filtered the final code, and if there was a format error(meaning the final code could not be extracted), we provided feedback to LLMs to generate the correct format of middle code. Similarly, if the *Mean Score* of the final code was less than 1, we also provided feedback to LLMs to initiate a new round of code generation. Considering time and API-call costs, we set a maximum feedback limit of 3 for both types of feedback.

4. Experiments

In this section, we first introduce the Hospital Item Transport Dataset(HITD). Then, we conduct a series of experiments on this dataset with different baseline models and code prediction methods, demonstrating the strong robustness and generalization capability of MCP as a code prediction method.

4.1. Experimental setup

Tasks and dataset. HITD is a dataset consisting of user instructions and long sequences of low-level code that can drive logistics robots for item transportation. The user instructions comprises *[originating department, intermediate transport department, final department]*. In the example shown in Figure 3, the *originating department* is the Respiratory Medicine Department, the *intermediate transport department* is the Gastrointestinal Surgery Department, and the *final department* is the General Services. Additionally, the *originating department* can determine whether to include "priority" based on user instructions. If "priority" is included, this portion of the code will be written first, followed by the tasks of departments without priority. Based on the code modeling length and reasoning complexity, the dataset is divided into three levels from simple to difficult, denoted as Task1, Task2, and Task3. Table 1 presents the main information for each level of tasks.

Table 1: Task information of HITD

Task Name	Code Length	Inference Complexity	Number Of Departments (originate/ intermediate)	Priority
Task1	202	simple	1/ 1-9	×
Task2	371	moderate	1-4/ 1-5	×
Task3	399	difficult	1-4/ 1-5	✓

Baseline models. We compared different baseline models, including GPT-3.5-Turbo-16k, which supports long contexts, and the Llama2 series models Touvron et al. (2023), including Llama2(7b), Llama2(13b), CodeLlama2(7b), and CodeLlama2(13b). Additionally, we introduced the ChatGLM2(6b) Du et al. (2022), which will be used for subsequent fine-tuning. All experiments were conducted on 2 NVIDIA RTX 3090-24GB GPU.

Baseline methods. Currently, there is still lack of a general and effective method for code prediction in uncommon programming languages. We compared our method with the currently popular methods based on high-level APIs Vemprala et al. (2024); Arenas et al. (2024); Liang et al. (2023). We adapted these methods for low-level code prediction by prompting LLMs with environment information, task descriptions, status descriptions, and low-level code information, allowing LLMs to output the final code, we named these methods Final Code Prediction(FCP). Decompose Module first decomposes high-level tasks into sub tasks and then recursively solves each sub task, which has been proven effective for decision-making LLMs in several papers Zhou et al. (2023); Zhang et al. (2024), and we called this method to Final Code Prediction with Decompose Module(FCP-DM). Additionally, we introduced the efficient fine-tuning method P-tuning v2 Liu et al. (2022), as a representative of specific domain fine-tuning methods.

4.2. Metrics

Since the inputs and outputs of the tasks are in a one-to-one correspondence, we focus on the correctness of code generation and introduce the Accuracy *(Acc)* metric, which represents the proportion of correctly predicted code among all task samples. Considering that incorrectly predicted code can still vary in quality, we use the *BLEU* score Papineni et al. (2002) to evaluate how close the generated code is to the reference code. Additionally, since the output of LLMs sometimes does not conform to the specified output format, even after few-shot prompting and error retrying, it may present correct code but not extracted. To address this issue, we set the *BLEU* score to 0.5 in such cases:

$$BLEU = \begin{cases} 0.5 & if\ not\ extract\ code \\ BP \cdot exp\left(\frac{1}{n}\sum_{i=1}^{n}\log P_i\right) & otherwise \end{cases} \tag{6}$$

Where BP is the length penalty factor, and P_i is the precision of n-gram exact matches. To comprehensively evaluate the results of code generation, we introduce the *Mean Score*, which is the weighted sum of *Acc* and *BLEU* scores:

$$MeanScore = \lambda \cdot Acc \ + \ (1-\lambda) \cdot BLEU \tag{7}$$

In this paper, we set λ to 0.5.

4.3. Main Results

4.3.1. MCP Can Improve the Mean Scores of Baseline Models

We introduce Task1, Task2, and Task3, and randomly sample 100 test cases for each task. The shot number of prompting methods are both set to 5, which is considered an effective shot number as discussed in Section 4.3.2. P-Tuning v2 is tested using checkpoints after 1000 steps.

Table 2 presents the results. Among the Llama2 series models, the base model, such as Llama2(7b) and Llama2(13b), is not well-suited for this type of code prediction task, the best MCP method only completed 26% of the tasks on Llama2(13b). However, CodeLlama2(7b), which is fine-tuned on code data and instructions based on Llama2, is better at handling code-related problems. We found that when using the MCP method, this model performs well on all three tasks, with an average *Acc* improvement of 46% and an average *BLEU* improvement of 13.8% compared to the FCP method. CodeLlama2(13b) once again set new best results for this series on multiple metrics. After using the MCP method, the average *Acc* reached 63%, indicating that the MCP method can achieve better results as the parameter size of the baseline model increases. GPT-3.5-Turbo-16k also achieved good results in code prediction. It achieved an *Acc* of 85% on Task1 using the FCP and FCP-DM methods. However, as the task difficulty increased, these methods experience a rapid decline in performance. In contrast, the performance of MCP was more stable, achieving excellent average *Acc* and *BLEU* scores of 74% and 97.8%, respectively. Fine-tuned models adapt better to this type of task, furthermore, fine-tuning with *(instruction, middle code)* pair data enables the models focus more on code reasoning, thus result in the greatest performance gains.

4.3.2. Ablation study of MCP

The middle code consists of natural language reasoning and simplified code. For the natural language reasoning part, we adopted the least-to-most approach, implicitly breaking down the problem into several sub-code issues and summarizing each sub-code issue. Here, we studied the importance of Sequential and SubSummary to the MCP method. We randomly selected 100 test data from Task1, Task2, Task3, and Mix-Task, where Mix-Task is composed of a mix of the above three tasks. We enabled the CodeLlama2(7b) baseline model to predict the code through few-shot prompting. As shown in Figure 4, the MCP method always outperforms the FCP method in all tasks, even when lacking the Sequential and SubSummary modules individually. As the number of shots increases, most methods show a performance growth trend, but there is also a saturation state. When the number of shots reaches 7, the code prediction performance of some methods decreases, which may be attributed to the inherent limitations of the LLM's context window. Therefore, considering both time cost and performance, we set the optimal number of shots to 5. Unless otherwise specified, all the results discussed in this paper are derived from the 5-shot condition.

The impact of the Sequential and SubSummary modules varies across different tasks. In simpler tasks like Task1, the SubSummary module has a lesser effect, and its absence has minimal impact on MCP because Task1 requires fewer subdivisions of sub-code issues and thus fewer summaries. In contrast, the Sequential module plays a significant role here. Without the Sequential module, there is a decrease of 7.05% in the *Mean Score*(85.05% vs.

Table 2: Comparison of code prediction methods on different baseline models.

Model	**Method**	**Task1**		**Task2**		**Task3**		**Average**	
		Acc	BLEU	Acc	BLEU	Acc	BLEU	Acc	BLEU
Llama2(7b)	FCP	-	0.642	-	0.380	-	0.125	-	0.382
	FCP-DM	-	0.657	-	0.398	-	0.382	-	0.479
	MCP	-	0.716	-	0.611	-	0.529	-	0.619
Llama2(13b)	FCP	0.03	0.702	0.01	0.521	-	0.220	0.01	0.481
	FCP-DM	0.05	0.721	0.02	0.515	0.02	0.552	0.03	0.596
	MCP	0.36	0.842	0.25	0.830	0.18	0.815	0.26	0.829
Code Llama2(7b)	FCP	0.12	0.825	0.05	0.738	0.02	0.760	0.06	0.774
	FCP-DM	0.12	0.832	0.06	0.770	0.05	0.762	0.08	0.788
	MCP	0.73	0.971	0.55	<u>0.929</u>	0.27	0.836	0.52	0.912
Code Llama2(13b)	FCP	0.13	0.918	0.10	0.855	0.06	0.835	0.10	0.869
	FCP-DM	0.14	0.921	0.12	0.871	0.12	0.854	0.13	0.882
	MCP	0.76	0.972	<u>0.75</u>	0.899	<u>0.37</u>	0.848	<u>0.63</u>	0.906
GPT-3.5-Turbo-16k	FCP	<u>0.85</u>	0.980	0.30	0.908	0.20	0.916	0.45	0.935
	FCP-DM	<u>0.85</u>	<u>0.982</u>	0.33	0.910	0.36	<u>0.921</u>	0.51	<u>0.938</u>
	MCP	**0.89**	**0.985**	**0.86**	**0.990**	**0.48**	**0.959**	**0.74**	**0.978**
ChatGLM2(6b)	P-Tuning v2	<u>1.00</u>	<u>1.000</u>	<u>0.91</u>	<u>0.994</u>	<u>0.77</u>	<u>0.972</u>	<u>0.89</u>	<u>0.989</u>
	P-Tuning v2*	**1.00**	**1.000**	**0.94**	**0.997**	**0.88**	**0.984**	**0.94**	**0.994**

The best scores are highlighted in bold, and the second-best scores are underlined. P-Tuning v2 indicates that the fine-tuning data is in the form of (instruction, final code), while P-Tuning v2* indicates that the fine-tuning data is in the form of (instruction, middle code).

78%). However, as tasks become more complex and the code modeling length increases, requiring more sub-code summaries. At this point, the SubSummary module becomes more influential. In Task2, Task3, and Mix-Task, the absence of the SubSummary module leads to *Mean Score* decreases of 21.75%, 11%, and 5.9%, respectively.

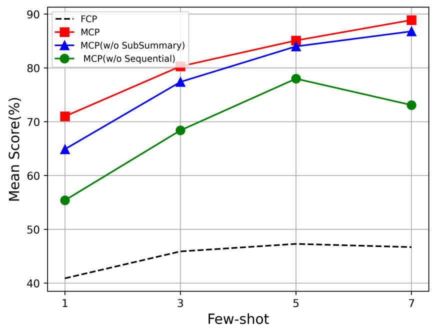
These results demonstrate that both the Sequential and SubSummary modules play crucial roles in the MCP method. The Sequential module aligns user inputs with sub-code replacements, reducing the burden of reasoning for the model. Meanwhile, the SubSummary module breaks down code issues into individual sub-problems and summarizes them, simplifying the difficulty of long code tasks. This capability enables the CodeLlama2 model with 7b parameters to handle long code prediction tasks effectively.

4.3.3. MCP Enhances the Noise Robustness of Fine-tuned Models

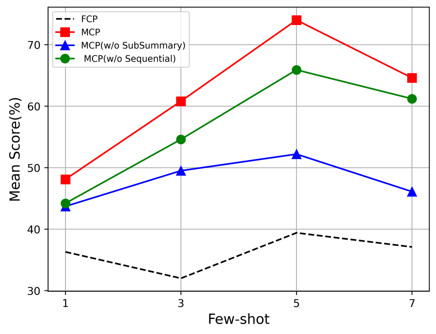
This section explores whether fine-tuned models can benefit from the MCP method. We introduced different types of noise disturbances into the original test data, which are commonly encountered in real-life scenarios. Specifically, they are categorized into three types:

1) noise1: Altering the user expression while preserving the overall meaning, including replacing synonyms, rephrasing sentence structures, etc.

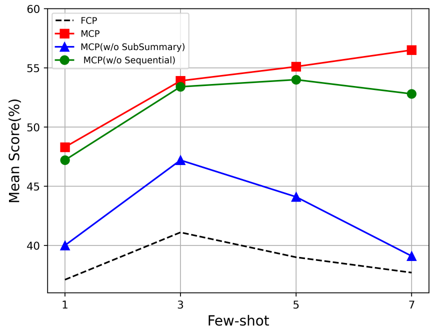
2) noise2: Adding an additional department, such as Dermatology department.



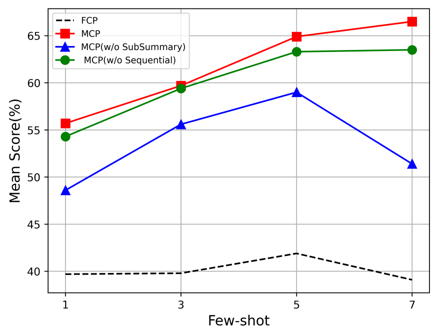
(*a*) Task1



(*b*) Task2



(*c*) Task3



(*d*) Mix-Task

Figure 4: Ablation study of the MCP method.

3) noise3: Randomly changing the serial number of one department, for instance, changing the serial number of Cardiac Surgery department from 1 to 0.

We randomly sampled 20 data points from the Mix-Task each round, totaling 10 rounds to form the original test set. Subsequently, we added noise1, noise2, and noise3 to form the remaining three noise datasets, testing the robust performance of various methods. MCP used the CodeLlama2(7b) model, while the fine-tuned model used ChatGLM2(6b).

The results are shown in Figure 5. After adding different noise datasets, MCP showed minimal changes in *Mean Score*, with a maximum median change of only 4% (0.622 vs 0.583 vs 0.601 vs 0.594). This indicates that MCP maintains strong resistance to interference and robustness. P-Tuning v2, fine-tuned using final code, showed little change in performance after adding noise1 data (0.961 vs 0.948). However, after adding noise2 and noise3 data, the *Mean Score* decreased by 33.6% and 28.0%, respectively. This significant change suggests that P-Tuning v2 lacks good noise resistance because it only memorizes the original dataset and struggles with noise like adding a new department or changing department numbers not included in the training data. In contrast, when we fine-tuned using middle code, the situation changed. We found that the model decreased by only 0.1%, 17.3%, and 9.6% on noise1, noise2, and noise3, respectively, reducing the impact of noise disturbance on the model's *Mean Score*. This may be because middle codes involve natural language reasoning, allowing the fine-tuned model to adapt its code output based on changes mentioned in user instructions, even for data the model has not seen before.

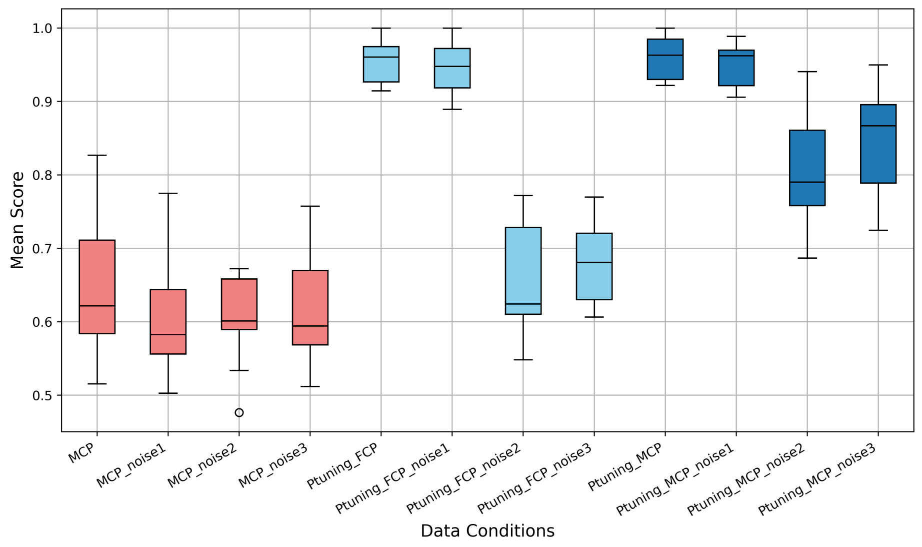


Figure 5: Robustness of different methods to different noise datasets

4.3.4. Generalization Ability of MCP in Different Scenarios

To verify whether MCP can be applied to different scenarios, we further developed the Robopal simulation environment Zhou et al. (2024) and created a robot desktop block placement scenario. In this scenario, each data consists of a pair *[user instruction, final code]*. The final code is a custom scarce code that follows certain syntactic rules, which can be extracted and executed in the Robopal. It should be noted that we simplified the visual task by recording all object positions in advance. We tested the Gripper-move, Block-stack, Block-move, and integrated four scenarios. Table 3 shows the detailed descriptions of these tasks.

Table 3: The task description of desktop block placement

Type	**Task Descriptions**
Gripper-move	Move the gripper <length><orientation>to the <block/mug>
Block-stack	Stack the <block>on the <block/mug>
Block-move	Place the <block><length><orientation>to the <block/mug>
Integrated	[Gripper-move], [Block-stack],. . . ,[Block-move]

We tested the CodeLlama2(13b) and the GPT-3.5-Turbo-16k model using In-Context Learning to prompt each model. Ten test cases for each scenario were manually annotated, and the desktop results were evaluated by running the final code. The results are shown in Table 4. The performance of CodeLlama2 in each task was slightly inferior to that of the GPT3.5 model, which may be due to the inherent capabilities of the model. Additionally, the MCP achieved higher success rates on both models compared to previous methods. In the Block-stack scenario with the GPT3.5 model, all test instructions were successfully executed. As the instruction length increased, FCP and FCP-DM failed to summarize effectively and

Table 4: Success Rate on Evaluation Task Set

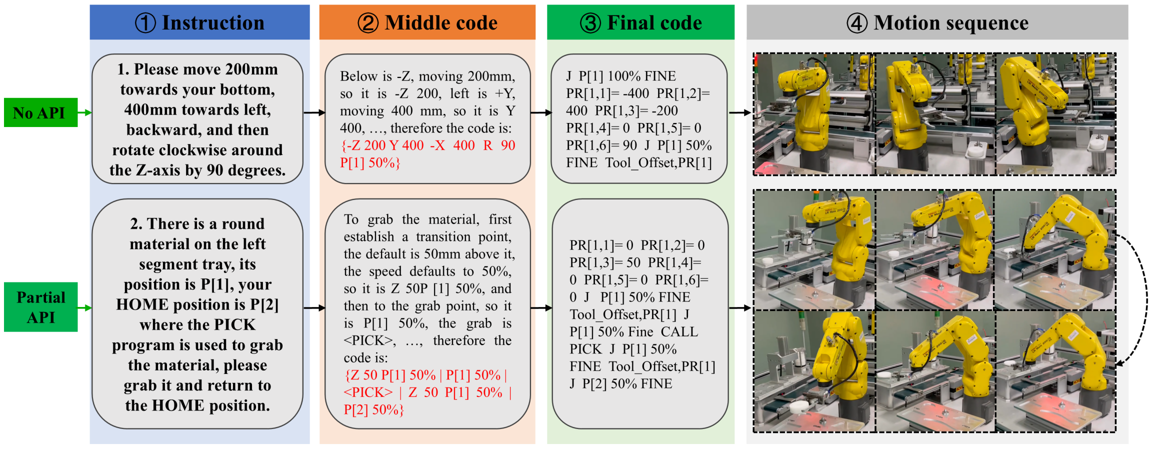
Type	CodeLlama2(13b)			GPT-3.5-Turbo-16k		
	FCP	FCP-DM	MCP	FCP	FCP-DM	MCP
Gripper-move	4/10	5/10	8/10	5/10	5/10	**9/10**
Block-stack	2/10	4/10	6/10	4/10	5/10	**10/10**
Block-move	2/10	2/10	6/10	3/10	4/10	**9/10**
Integrated	0/10	0/10	5/10	1/10	2/10	**6/10**

performed poorly in integrated tasks. In contrast, MCP's SubSummary module could better adapt to long sequence tasks, successfully running 6 out of 10 integrated tasks.

4.4. Real-world robot Manipulation with MCP

This section we conducted code generation experiments on a real robot, named FANUC LR Mate 200iD/4S. We tested the code generation effectiveness of LLMs in scenarios with no API and partial API encapsulation. The GPT-3.5-Turbo-16k model was used for testing.

For scenarios without API encapsulation, The first task involved simple motion instructions. The model correctly generated simplified code {- Z 200 Y 400 -X 400 R 90 P[1] 50%} for this task using few-shot prompting. Subsequently, this was extended into final code and successfully executed the motion to drive the robotic arm, as shown in Figure 6, In the second scenario, we attempted to encapsulate the actions of the robotic arm into high-level APIs. Here, the grasping action was encapsulated as "PICK". We supplemented these examples in the context of LLMs. We found that even in this partially encapsulated scenario, the model still generated middle code and produced the correct simplified code through reasoning {Z 50 P[1] 50% | P[1] 50% | <PICK> | Z 50 P[1] 50% | P[2] 50%}, which also extend the final code and successfully drove the robotic arm to complete the material grasping.



1. Please move 200mm towards your bottom, 400mm towards left, backward, and then rotate clockwise around the Z-axis by 90 degrees.

Below is -Z, moving 200mm, so it is -Z 200, left is +Y, moving 400 mm, so it is Y 400, …, therefore the code is: {-Z 200 Y 400 -X 400 R 90 P[1] 50%}

```
J P[1] 100% FINE PR[1,1]= -400 PR[1,2]= 400 PR[1,3]= -200 PR[1,4]= 0 PR[1,5]= 0 PR[1,6]= 90 J P[1] 50% FINE Tool_Offset,PR[1]
```

2. There is a round material on the left segment tray, its position is P[1], your HOME position is P[2] where the PICK program is used to grab the material, please grab it and return to the HOME position.

To grab the material, first establish a transition point, the default is 50mm above it, the speed defaults to 50%, so it is Z 50P [1] 50%, and then to the grab point, so it is P[1] 50%, the grab is <PICK>, …, therefore the code is: {Z 50 P[1] 50% | P[1] 50% | <PICK> | Z 50 P[1] 50% | P[2] 50%}

```
PR[1,1]= 0 PR[1,2]= 0 PR[1,3]= 50 PR[1,4]= 0 PR[1,5]= 0 PR[1,6]= 0 J P[1] 50% FINE Tool_Offset,PR[1] J P[1] 50% Fine CALL PICK J P[1] 50% FINE Tool_Offset,PR[1] J P[2] 50% FINE
```

Figure 6: MCP realizes the robotic arm movement on both no API and partial API encapsulation

5. Limitations

In this section, we highlight two limitations of MCP. The first limitation is that the stage1 of MCP requires the provision of several high-quality quadruples, which may necessitate the involvement of an experienced prompt engineer to optimize for different types of LLMs. The second limitation pertains to the generalization experiments. Although we successfully controlled the robot using the final code, the object pose information was pre-written into a file. Further research is needed to integrate MCP with vision detection models to enable motion in dynamic scenarios.

6. Conclusion

This paper addresses the issue of code generation for uncommon programming languages that typically rely on high-level API encapsulation by proposing a three-phase approach called Middle Code Prediction(MCP). By injecting prompts at different stages, LLMs shift towards predicting middle code, which is easier for them to understand, and this middle code can then be mapped to the final code through extension scripts. This approach eliminates the need for high-level API encapsulation while improving the *Mean Score* of various baseline models. We tested MCP for robustness and generalization, finding that combining MCP with fine-tuned models yields significant performance benefits. Furthermore, this approach can be effectively deployed in industrial robotic arm, including scenarios with no API and partial API encapsulation. In the future, we plan to incorporate visual modal information for more complex robotic arm motion testing.

Acknowledgments

This work is supported by the National Key R&D Program of China (No. 2023YFB3710901), the Liaoning Applied Basic Research Program Project, China (2023JH2/101300146).

References

Montserrat Gonzalez Arenas, Ted Xiao, Sumeet Singh, Vidhi Jain, Allen Z. Ren, Quan Vuong, Jake Varley, Alexander Herzog, Isabel Leal, Sean Kirmani, et al. How to prompt your robot: A promptbook for manipulation skills with code as policies. In *IEEE International Conference on Robotics and Automation(ICRA)*, pages 4340–4348, 2024.

Leonard Bärmann, Rainer Kartmann, Fabian Peller-Konrad, Alex Waibel, and Tamim Asfour. Incremental learning of humanoid robot behavior from natural interaction and large language models. *arXiv preprint arXiv:2309.04316*, 2023.

Juo-Tung Chen and Chien-Ming Huang. Forgetful large language models: Lessons learned from using llms in robot programming. In *Proceedings of the AAAI Symposium Series*, volume 2, pages 508–513, 2023.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, et al. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. GLM: general language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics(ACL)*, pages 320–335, 2022.

Shyam Sundar Kannan, Vishnunandan L. N. Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models. *arXiv preprint arXiv:2309.10062*, 2023.

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

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, 2023.

Rui Liu and Xiaoli Zhang. Generating machine-executable plans from end-user's natural-language instructions. *Knowledge-Based Systems*, 140:15–26, 2018.

Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics(ACL)*, pages 61–68, 2022.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations(ICLR)*, 2024.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics(ACL)*, pages 311–318, 2002.

D. Pigott. Online Historical Encyclopaedia of Programming Languages, 2020. URL `https://hopl.info/`.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530, 2023.

Julien Joseph Thomas, Vishnu Suresh, Muhammed Anas, Sayu Sajeev, and KS Sunil. Programming with natural languages: A survey. In *Computer Networks and Inventive Communication Technologies: Proceedings of Fourth ICCNCT 2021*, pages 767–779, 2022.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

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

Sida I. Wang, Samuel Ginn, Percy Liang, and Christopher D. Manning. Naturalizing a programming language via interactive learning. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics(ACL)*, pages 929–938, 2017.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations(ICLR)*, 2023.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing(EMNLP)*, pages 8696–8708, 2021.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems(NeurIPS)*, 35: 24824–24837, 2022.

Sebastian Weigelt, Vanessa Steurer, Tobias Hey, and Walter F. Tichy. Programming in natural language with fuse: Synthesizing methods from spoken utterances using deep natural language understanding. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics(ACL)*, pages 4280–4295, 2020.

Zhen Yang, Jacky Wai Keung, Zeyu Sun, Yunfei Zhao, Ge Li, Zhi Jin, Shuo Liu, and Yishu Li. Improving domain-specific neural code generation with few-shot meta-learning. *Information and Software Technology*, 166:107365, 2024.

Jiatao Zhang, Lanling Tang, Yufan Song, Qiwei Meng, Haofu Qian, Jun Shao, Wei Song, Shiqiang Zhu, and Jason Gu. FLTRNN: faithful long-horizon task planning for robotics with large language models. In *IEEE International Conference on Robotics and Automation(ICRA)*, pages 6680–6686, 2024.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, et al. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations(ICLR)*, 2023.

Haoran Zhou, Yichao Huang, Yuhan Zhao, and Yang Lu. robopal: A Simulation Framework based Mujoco, April 2024. URL https://github.com/NoneJou072/robopal.