Developments in the Built Environment 19 (2024) 100488

Contents lists available at ScienceDirect

Developments in the Built Environment

journal homepage: www.sciencedirect.com/journal/developments-in-the-built-environment

Large language model-based code generation for the control of construction assembly robots: A hierarchical generation approach

Hanbin Luo [a,b], Jianxin Wu [a,b], Jiajing Liu [a,b,c,*], Maxwell Fordjour Antwi-Afari [d]

[a] National Center of Technology Innovation for Digital Construction, Huazhong University of Science and Technology, 430074, Wuhan, China
[b] School of Civil and Hydraulic Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China
[c] School of Artificial Intelligence and Automation, Huazhong University of Science and Technology, 430074, Hubei, China
[d] Department of Civil Engineering, College of Engineering and Physical Sciences, Aston University, B4 7ET, Birmingham, UK

ARTICLE INFO

Keywords:
Construction assembly robot
Large language model
Code generation
ChatGPT
Human–robot collaboration

ABSTRACT

Offline programming (OLP) is a mainstream approach for controlling assembly robots at construction sites. However, existing methods are tailored to specific assembly tasks and workflows, and thus lack flexibility. Additionally, the emerging large language model (LLM)-based OLP cannot effectively handle the code logic of robot programming. Thus, this paper addresses the question: *How can robot control programs be generated effectively and accurately for diverse construction assembly tasks using LLM techniques?* This paper describes a closed user-on-the-loop control framework for construction assembly robots based on LLM techniques. A hierarchical strategy to generate robot control programs is proposed to logically integrate code generation at high and low levels. Additionally, customized application programming interfaces and a chain of action are combined to enhance the LLM's understanding of assembly action logic. An assembly task set was designed to evaluate the feasibility and reliability of the proposed approach. The results show that the proposed approach (1) is widely applicable to diverse assembly tasks, and (2) can improve the quality of the generated code by decreasing the number of errors. Our approach facilitates the automation of construction assembly tasks by simplifying the robot control process.

1. Introduction

Assembly tasks are among the most common tasks in construction. Notably, the temporal allocation dedicated to assembly undertakings constitutes approximately 70% of the aggregate project duration during construction (Ding et al., 2020). Additionally, the process of assembling structural elements is an example of physically exacting vocations, often necessitating iterative engagements with contorted stances involving bending and twisting to perform manual handling and repetitive activities (Gao et al., 2023). With advancements in robotics, intelligent robots provide a promising solution for preventing musculoskeletal disorders during assembly tasks and improving construction efficiency.

Offline programming (OLP) is a mainstream method for controlling assembly robots on construction sites (Zhang et al., 2023). It stores codes relevant to tasks as a corpus, captures the intrinsic logic of the action plan based on the task and environmental descriptions, and organically combines the corpus to generate an executable robot control program. OLP employs virtual robots for programming and simulation, offering a flexible solution suitable for intricate environments and tasks, such as the collision-free path planning of robots (Vann et al., 2023). Additionally, OLP can collaborate with technologies such as building information modeling, facilitating the workflow planning of robot construction, and allocating multi-robot tasks (Kim et al., 2021; Zhu et al., 2021).

Three primary categories of automated OLP methods have been investigated: template-based (Hu et al., 2022; Rogeau et al., 2021), skill library-based (Wallhoff et al., 2010; Zheng et al., 2022), and neural machine translation (Bonilla and Ugalde, 2019; Kahuttanaseth et al., 2018) methods. However, despite the valuable contributions from the extensive research, these methods often have a limited scope, are tailored for specific assembly tasks and workflows, and are difficult to extend to intricate scenarios. Limited corpora of templates, skill libraries, and small models have restricted the range of generated codes. Moreover, crucial task-related details such as the construction process

* Corresponding author. National Center of Technology Innovation for Digital Construction, Huazhong University of Science and Technology, 430074, Wuhan, China.

E-mail address: liujiajing@hust.edu.cn (J. Liu).

https://doi.org/10.1016/j.dibe.2024.100488
Received 8 March 2024; Received in revised form 5 May 2024; Accepted 18 June 2024
Available online 20 June 2024

and masonry forms are rarely considered in these approaches, which results in poor adaptability of the generated codes to unconventional construction processes, requiring a large amount of code modification work.

Emerging large language model (LLM) techniques have demonstrated significant potential in robot programming and intricate task resolution (Saka et al., 2024; Zhao et al., 2023), which is promising an effective avenue for enhancing robot control. Notably, an impressive study by the Autonomous Systems and Robotics Group at Microsoft™ harnessed *ChatGPT* to generate robotic action plans for diverse applications, enabling users to be supervisors in a closed loop to iteratively refine a solution (Vemprala et al., 2023). However, the execution of the generated plan relies on the collaboration of pre-defined functions, which necessitates the inclusion of low-level policies within these functions. A recent study by Google™ proposed a hierarchical code-writing approach that recursively defines and nests functions to create programs capable of controlling robotic movements (Liang et al., 2023). However, the generated programs were not flattened at the same code logic level and a clear demarcation between the different levels of robot programming was not evident, resulting in excessive functional nesting and a more intricate code structure.

Against this contextual background, this paper addresses the following research question: *How can robot control programs be generated effectively and accurately for diverse construction assembly tasks using LLM techniques?* To address this research question, we refer to the seminal work of Vemprala et al. (2023) and develop a closed user-on-the-loop construction assembly robot control framework based on LLM techniques. This framework encompasses the construction of an API library, hierarchical robot control program generation (HRCPG), simulation and optimization, and execution phases.

Firstly, a customized inventory of basic robot manipulation functions for construction assembly tasks is compiled to form an API library. Subsequently, HRCPG strategy is proposed to logically integrate code generation at high and low levels with the API library, while augmenting LLM's comprehension of assembly action logic through a chain of action (CoA) within prompts. Following this, the AI-generated code is executed on a robot simulation platform to evaluate its accuracy and efficiency. Task completion-related metrics (i.e., code executability and condition recall) and task performance-related metrics (e.g., construction efficiency and energy expenditure) are used to ensure the successful completion of assembly tasks and to optimize task execution performance, respectively. During this phase, LLM identifies all errors in the generated code and provides feasible solutions. After several rounds of LLM-assisted optimization, the generated codes are seamlessly implemented on a physical robot. The proposed framework enables automated robot control from task planning to low-level policy codes generation through conversational programming with the LLM, thereby liberating users from arduous and demanding programming tasks and mitigating the knowledge barrier and interaction complexity associated with deploying construction robots. Additionally, the accuracy and reliability of the LLM outputs are guaranteed through HRCPG, CoA strategy and the closed-loop control process, providing a tangible example of LLM's efficacy in construction robotics applications.

The remainder of this paper is organized as follows. Section 2 reviews previous research on assembly robot programming in construction, LLMs for robot programming, and prompt engineering. The proposed LLM-based user-on-the-loop robot control approach is described in Section 3. Next, the feasibility and reliability of the proposed approach were verified through experiments on an assembly task set, as discussed in Section 4. Sections 5 and 6 present the contributions and limitations, respectively, and concluding remarks are presented in Section 7.

2. Literature review

In this section, we first examine the control method of construction assembly robots, focusing particularly on the OLP approach. It is highlighted that while the emerging LLM technology holds promise in simplifying complex task programming, its application in construction robot assembly requires further exploration. Consequently, in Section 2.2, we conduct a comprehensive review of current research on LLM in robot tasks, revealing its limitations in low-level code implementation, which is also the main obstacle to its application in construction assembly tasks. A well-crafted prompt is indispensable for directing LLM to generate the robot control code. Thus, in Section 2.3, we delve into the principles of prompt engineering, laying the groundwork for designing a holistic framework for integrating LLM into the realm of construction assembly robot control.

2.1. Assembly robot programming in construction

Assembly tasks are among the most common activities in construction projects. In recent years, the use of robots for assembly tasks in construction projects has attracted significant interest in improving the productivity and safety of assembly work (Cai et al., 2019, 2020; Wang et al., 2021). In typical real-world scenarios, robots are programmed by end users to achieve motion control and autonomous decision-making based on different construction environments and assembly task requirements.

Robot programming mainly comprises online programming and OLP. In online programming, engineers directly code and debug within the robot's operational environment in real-time, while OLP simulates the robot's operations in a virtual environment. Unlike online programming, OLP removes users from the workspace, allowing early detection of potential mechanical collisions through simulation and adaptation to the evolving construction environment in real-time using sensors and advanced algorithms (Cai et al., 2019, 2020; Bruckmann and Boumann, 2021; Carvalho et al., 1998; Gao et al., 2022). In terms of efficiency, although OLP is hard-coded, it excels in highly repetition tasks such as assembly through program reuse (Carvalho et al., 1998). Additionally, OLP provides easier integration with building information models and digital twins, with numerous current studies conducted within an OLP environment (Kim et al., 2021; Zhang et al., 2023).

Traditionally, OLP is manually encoded by engineers, which is time-consuming and requires a high level of robotics expertise for effective robot operation (Huang et al., 2018). To address the limitations of manual programming, several methods have been developed for automatically generating robot commands for construction assembly.

The prevalence of analogous action sequences in element assembly procedures has resulted in the frequent adoption of template-based approaches, in which a fixed-action logical framework is established (Ding et al., 2020; Hu et al., 2022; Huang et al., 2021). For example, Huang et al. (2021) devised an action template and planned a skeleton for a discrete bar assembly to implement robot motion planning for additive bar structure construction. This template comprised four motion primitives: transit, pick, transfer, and place. The fixed-action logical framework can be represented as a sequence of robot instructions (Ding et al., 2020). However, most templates are tailored to specific assembly tasks and are difficult to extend to other scenarios.

Another OLP approach for generating robot assembly programs involves the creation of a skill library, which can automatically learn action logic and combine skills. Previous studies attempted to integrate this method with expert systems, knowledge models, and reinforcement learning methods to determine sequences of defined skills (Wallhoff et al., 2010; Zheng et al., 2022). For instance, Wallhoff et al. (2010) registered the available skills in an expert system, manually decomposed a task into these skills, and invoked them through voice input. However, the escalating number of skills required for task solutions correspondingly increases the demand for time and processing power (Wallhoff et al., 2010). Additionally, the skills in the library delineate the spectrum of actions that the robot can undertake, thus resulting in potential security concerns when the library is not aligned with the task requirements.

The above two methods require operators with high levels of knowledge and expertise. In contrast, more recently, neural machine translation has been utilized to translate natural language into robot commands for simpler human–robot interactions (Bonilla and Ugalde, 2019; Kahuttanaseth et al., 2018). However, it also encounters challenges in scaling to handle more intricate tasks and diverse instructions, owing to the limited size of training corpora.

LLM leverages its extensive training dataset, endowing it with capabilities that are not readily apparent in smaller models and the potential to effectively solve complex tasks (Zhao et al., 2023). Only a few studies have explored the application of LLMs to construction assembly robots, with only two studies from the same research group (You et al., 2023a; Ye et al., 2023). In the first study, a system called RoboGPT, based on *ChatGPT*, was developed for automatic sequence planning. When the system receives a task description from users, it generates sequential solution commands, which are then decoded by the command decoder system, and finally implements object manipulation (You et al., 2023a). In the second study, a user's voice was used as the input for RoboGPT. It combines contextual information to evaluate the ambiguity of voice information and clarifies operational instructions to users through conversation. When the information is sufficient for decision-making, the response is sent to the command decoder to trigger the robot control function to perform the task. In these two studies, LLM was used for the automatic planning of tasks and the invocation of predefined skills, but the customization of low-level policies was not adequately discussed. This is similar to the skill library-based method, which relies on predefined policies to map each step of the action plan. A summary and comparison of existing LLM-based OLP and general OLP methods is presented in Table 1.

In this study, we harnessed the saliency ability of LLMs to automatically generate robot programs, establishing a full path from task description to task planning, trajectory planning, and robot policy code generation, thereby reducing the difficulty of robot programming and improving the flexibility and security of codes in various assembly tasks.

2.2. LLMs for robot programming

In recent years, the emerging LLM technology has introduced a novel application paradigm for robots. Models such as *LLaMA* (Touvron et al., 2023) and *ChatGPT* (OpenAI and ChatGPT, 2023) have been successfully used to analyze user inputs and translate them into executable robot code. LLMs enable nonexpert users to communicate with robots and

Table 1
A summary and comparison of existing LLM-based OLP and general OLP methods.

Method	Description	Pros	Cons
Manual OLP	Hand coding	Flexible to different task requirements	Time-consuming and high knowledge requirement
Template-based OLP	Programming in a fixed template or action framework	Easy to use with only parameter setup	Weak adaptation to tasks with different workflow
Skill library-based OLP	Programming with encapsulated skills	Automated action logic learning and skills combination	Skills tailored for specific task requirements with limited application scope
Neural machine translation-based OLP	Directly translate natural language into robot codes using small models	Simpler human-robot interactions	Limited by the size of training corpora
LLM-based OLP	Directly translate natural language into robot codes using LLMs	More extensive training dataset and emergent ability for solving complex tasks	Lack customization of low-level policies in predefined skills

increase the trust of users in human–robot collaborations owing to their high communication effectiveness (Ye et al., 2023).

Task planning is an important research domain in the context of LLMs in robotics. In essence, when provided with prompts that describe the task and environment, the LLM decodes the task into a sequence of actions and arranges them logically through commonsense reasoning and code comprehension (Singh et al., 2023). Although previous studies have demonstrated the feasibility of robotic task planning using LLMs, optimizing an artificial intelligence (AI)-generated plan by modifying the order of action sequences poses challenges, primarily because of the open-loop nature of pipelines, which cannot receive feedback from the environment.

Recently, the Autonomous Systems and Robotics Group at Microsoft™ introduced a framework that facilitates testing, verification, and validation of the generated code by enabling user involvement in the loop. This framework employs *ChatGPT* to generate a robotic action plan, incorporating task-relevant APIs, and subsequently iteratively refining the solution by evaluating output quality and safety (Vemprala et al., 2023). However, the execution of the generated plan relies on the collaboration of pre-defined functions, requiring the inclusion of low-level policies within these functions. Based on this, ROSGPT (Koubaa, 2023) and KGGPT (Mu et al., 2023) harness an ontology to mitigate the limitation of the output of LLMs. However, they merely decompose a high-level task command into several sub-tasks or action sequences, without considering the coding work required for robots to execute these sub-tasks or actions. This underscores the significance of hierarchically generating robot code that seamlessly integrates high-level modules from task planning with their corresponding low-level policies.

The implementation of low-level policies involves research on code generation. Previous studies demonstrated the considerable advantages of LLMs in generating robot codes (Jain et al., 2022). However, the question of generating robot code hierarchically while simultaneously creating high-level modules and low-level policies using LLM techniques remains largely unexplored. In a recent study conducted by Google™, valuable insights were obtained to address this challenge (Liang et al., 2023). Researchers have proposed a code as policies (CaP) method that employs a hierarchical code-writing approach by recursively defining and nesting functions, thereby composing code ranging from simple *Python* instructions to more complex programs capable of controlling robotic movements. However, this approach is similar to most existing LLM-based robot task planning methods and is primarily applicable to housework environments, emphasizing the encapsulation and reuse of control functions (Huang et al., 2022; Singh et al., 2023). In addition, the generated programs were not flattened at the same code logic level and a clear demarcation between the different levels of robot programming was not evident, resulting in excessive function nesting and a more intricate code structure.

In this study, task planning and code generation of low-level policies by LLMs were logically integrated according to different levels of robot control, which simplifies the comprehension and application of robot control commands for users and augments the capability of LLMs to generate robotic solutions tailored to construction scenarios.

2.3. Prompt engineering

Prompts serve as instructions from users to the LLMs for intentional communication and information interaction (Liu et al., 2023). In LLM tasks, prompt engineering is a vital component, as a meticulous prompt design can result in a deeper comprehension of human intentions and result in more favorable outcomes. Many studies in the LLM domain have focused on prompt design, through both manual and automatic approaches, across diverse tasks such as task planning and text generation (Li and Liang, 2021; Teven and Alexander, 2021).

Prompts can be formulated in three forms: structured language, unstructured language, and a hybrid of the two. In an experiment (Liang et al., 2023), the use of code prompts for robot-relevant reasoning resulted in a higher success rate compared with the use of natural language prompts. While writing prompts in a structured language can mitigate potential syntax errors in the output, it requires users in the construction domain to engage in time-consuming and knowledge-intensive efforts to describe the usage of all necessary APIs in the codes. With the advancement of LLM technologies, studies have demonstrated the increased flexibility of prompting in natural language (Vemprala et al., 2023).

Thus, the approach described in this paper adopts natural language to describe the usage of APIs and task requirements for prompts. To facilitate a streamlined and expedited prompt construction process, we developed a dedicated API library tailored for robotic assembly in construction. Moreover, the logic chain of robot actions is added to the prompts, which draws inspiration from the concept of the chain of thought (CoT) (Wei et al., 2022) and collaborates with customized APIs to enhance the logical coherence of the generated outputs.

3. Methodology

As outlined in Section 2, existing programming approaches for construction assembly robots demand considerable effort from users to translate task instructions or action sequences into a robot programming language. Typically, construction workers are limited to participating solely in the robot execution stage, maintaining a passive collaboration with the robots. Leveraging the robust capabilities of LLMs in robot planning and control, we delve into its application within construction assembly tasks in this section. Here, we introduce a closed-loop framework designed to empower workers with limited programming skills to seamlessly engage in the entire process of robot control with the assistance of the LLM.

3.1. User on the loop of assembly robot control

Robot control can typically be divided into four levels: task, action, primitive, and servo (Siciliano et al., 2008). The primary objective of the task and action levels revolves around decomposing the overall task into a coherent sequence of logical actions, considering environmental constraints. Conversely, the primitive and servo levels focus on motion trajectory computation and driving to joint servomotors. Consequently, these control levels exhibit distinct requirements concerning the APIs to be utilized within robot codes as well as the essential task-related information necessary for effective robot programming.

Considering the distinctive attributes of the four control levels, a novel framework integrated with a strategy of hierarchical code generation is proposed for users in the loop of assembly robot control, as shown in Fig. 1. Given the APIs and task descriptions in natural language, the LLM parses the intentions of users and hierarchically generates robot control codes at high and low levels, which are then executed in the simulation to provide feedback to the LLM for code optimization. The proposed pipeline comprises the following four steps.

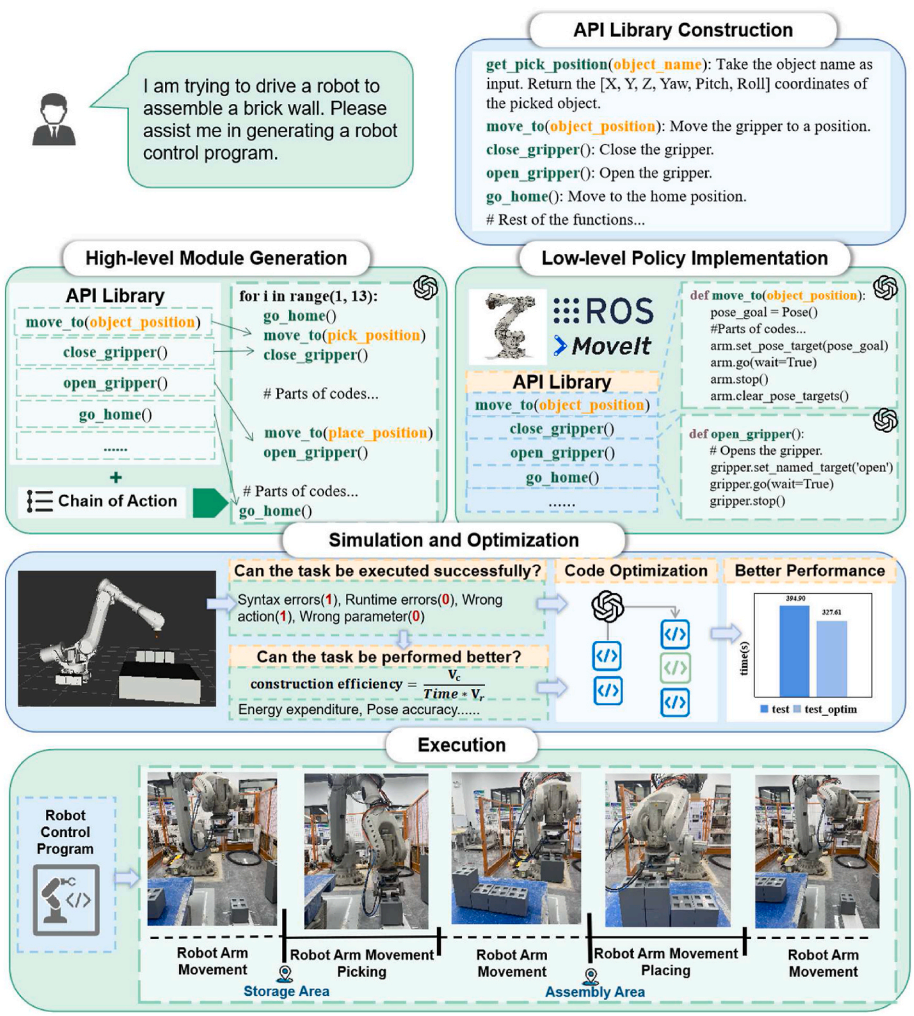

Fig. 1. Framework of the user on the loop of assembly robot control.

- ***API Library Construction:*** Before interacting with the LLM, a robotic API library is developed for construction assembly tasks, designed to seamlessly integrate with prompts for both high-level and low-level code generation. The API library comprises functions that operate at the action level with detailed descriptions of their inputs, outputs, and roles provided in natural language.
- ***HRCPG:*** The LLM is leveraged to build high-level modules and implement low-level policies using the bespoke API library. High-level module generation adopts a CoA strategy that collaborates with API calls to augment the LLM's reasoning capabilities in assembling action logic. Furthermore, third-party libraries are employed to create low-level operational plans within high-level functions, culminating in the synthesis of a comprehensive robot control program.
- ***Simulation and Optimization:*** The assessment of code quality entails the execution of the generated program on a virtual agent, followed by the application of metrics to evaluate its performance. The feedback obtained is incorporated into iterative prompts, providing valuable guidance for optimizing the AI-generated robot control program.
- ***Execution:*** After multiple rounds of optimization, the robot control codes can be interpreted and deployed on an actual robot for construction assembly tasks.

Since most third-party libraries for robots handle the implementation and encapsulation of control logic at a primitive level, we set the control levels of the third-party libraries and the API library as primitive and action levels, respectively. Specifically, in the proposed framework, the process of high-level module generation refines robot commands from the task level to the action level, also referred to as task planning. In low-level policy implementation, the LLM further refines the action sequences from high-level modules to the primitive level by leveraging third-party libraries. These two phases are logically connected through the API library at the action level.

3.2. API library for LLM prompting in construction assembly

In this section, the process of building the API library is described to ensure its ability to fulfill the specific demands of robot assembly tasks. These tasks often contain general action primitives, such as grasping and placing. Considering this, we present an inventory of basic robot manipulation functions for construction assembly tasks grouped into three categories: perception, action, and optional functions (Table 2).

Perception functions are used to access environmental information using sensors, and their outputs are used by action functions to generate motion trajectories. Action functions encompass a spectrum of action sequences executed by a robot, including arm movement, picking, placing, and robot movement. In addition to the basic functions in the workflow, supplementary functions that can be tailored to task specifications in the API library are used, such as *check_gripper()*. The efficacy of descriptive names and detailed descriptions of APIs has been proven to facilitate the utilization of functions and ensure accurate code generation (Liang et al., 2023; Vemprala et al., 2023). Hence, the bespoke functions are named according to their specific functionalities while providing lucid explanations of the types of inputs and outputs, as presented in Table 2.

Using the functions listed in Table 2, users can readily select APIs and modify their names and descriptions to form an API library for LLM prompting. Although certain motions may recur at different steps of the assembly, the environmental conditions and state of the robot can differ at each step. Hence, the assembly process should be meticulously scrutinized and the relevant functions selected accordingly. For instance, as shown in Fig. 2, the robot is always under load when moving from the storage area to the assembly area, which requires checking the state of the end effector while moving and reducing the moving velocity to minimize energy consumption and ensure safety. Consequently, the function of *check_gripper()* should be listed in the API library and the description of the function related to robot movement should be modified into "*navigate_to(assembly_area_position): Navigates the mobile robot to the assembly area position at the speed of* …" However, when the robot moves from the assembly area to the storage area, it always operates under no-load conditions, and the function requirement is relatively simple.

3.3. Hierarchical robot control program generation

3.3.1. HRCPG framework

As described in Section 3.1, HRCPG consists of two parts: high-level module generation and low-level policy implementation. Based on the connotation of the different levels of robot control, in this paper, the task and action levels of robotic programming are considered high level, and the primitive and servo levels are regarded as low level. The HRCPG pipeline is shown in Fig. 3.

Prompts are designed for the LLM at both high-level and low-level, respectively containing necessary information about the robot and the task. For high-level module generation, based on the customized API library and CoA in prompts, the LLM is given a hint to break an assembly task into action sequences and accordingly generate the main control unit of the program with APIs. The implementation of low-level policies involves the integration of third-party libraries, including open-source robot control libraries (Chitta and Koubaa, 2016; Sucan et al., 2012) and algorithms (Ren et al., 2021; Yang et al., 2022; You et al., 2023b). These libraries and algorithms are used to generate low-level code functionalities within high-level functions. Finally, responses to these

Table 2
APIs for LLM prompting in construction assembly tasks.

Types		Names	Description
Perception Function		get_area_position()	Return the area location [X, Y, Z].
		get_pick_position(object_name)	Take the object name as input. Return the [X, Y, Z, Yaw, Pitch, Roll] coordinates of the picked object.
		get_place_position(assembly_order, object_name)	Take the assembly order of the current object and object name as input. Return the [X, Y, Z, Yaw, Pitch, Roll] coordinates of the placed object.
Action Function	Robotic Arm Movement	move_to(object_position)	Move the end effector to a position specified by [X, Y, Z, Yaw, Pitch, Roll] coordinates. Return nothing.
		move_joints(object_position)	Move the end effector to the position along the fastest path using jointed movements. Input the [X, Y, Z, Yaw, Pitch, Roll] coordinates and return nothing.
		go_home()	Move the robotic arm and its end effector to the home position. Return nothing.
	Picking	close_gripper()	Close the gripper and grab the object. Return nothing.
		turn_on_suction_pump()	Turn on the suction pump and grab the object. Return nothing.
	Placing	open_gripper()	Open the gripper and place the object. Return nothing.
		turn_off_suction_pump()	Turn off the suction pump and place the object. Return nothing.
	Robot Movemnet	navigate_to(area_position)	Navigate the mobile robot to the area position. Input the [X, Y, Z] coordinates and return nothing.
		fly_to(area_position)	Fly the drone to the area position. Input the [X, Y, Z] coordinates and return nothing.
Optional Function		check_gripper()	Check the state of the gripper. If it is closed, open it.

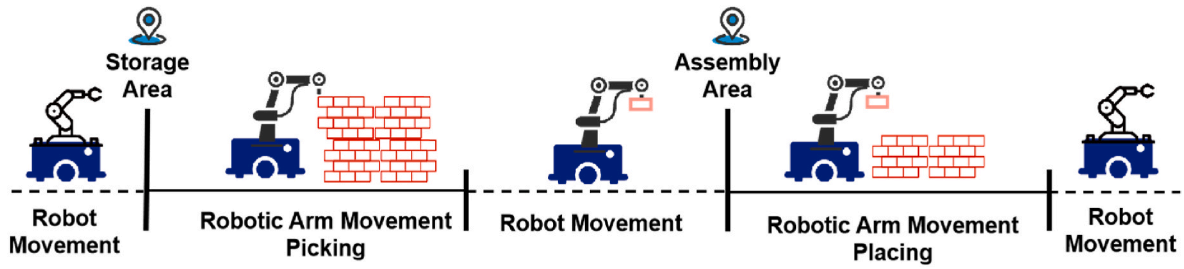

Fig. 2. Working process of a typical construction assembly task.

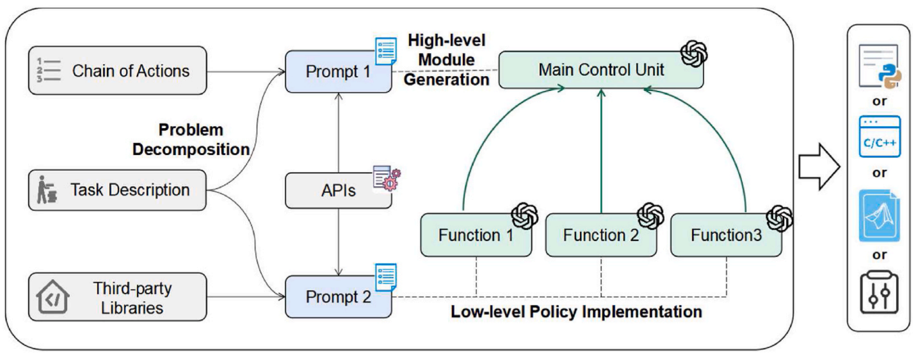

Fig. 3. Pipeline of HRCPG.

two prompts are integrated into a complete executable script. Throughout the process, the APIs act as a bridge between the high-level module and low-level policies of the robot control codes. Ensuring consistent API descriptions in both prompts contributes to the harmonious synchronization of function usage across the two programming levels.

3.3.2. High-level module generation

In this section, we introduce the prompt-setting rules specific to high-level module generation using the API library, in which the prompts are designed to contain the following parts:

- **Environment:** Describes obstacles that the robot may encounter during the assembly process and the relative spatial relationship between the robot and the working space. Within the context of the construction assembly task, the workspace primarily involves material storage and assembly areas, wherein the position and orientation of these regions should be informed to the LLM for the picking and placing of assembly components.
- **Robot:** The type of assembly robot and the description of its end effector should be introduced to the LLM, which is conducive for the AI to interpret and comprehend the available APIs, such as *open_gripper()* and *activate_suction_pump()*.
- **Task:** Inform the LLM of information related to task execution, such as the construction process and masonry form. For example, in the process of laying floor tiles, users can specify that tiles should be arranged in a straight lay pattern, forming a grid with dimensions of 5 × 3. Additionally, they can define a safe distance above the material before the picking operation. To utilize vision-based techniques to perceive the assembly materials and environment, the user should integrate the input information for the corresponding model into the prompts.
- **Chain of Action:** Describes the action execution sequences of the assembly task. Inspired by the concept of CoT (Wei et al., 2022), the proposed approach involves designing prompts that offer a brief description of action logic in natural language, thereby enhancing the ability of the LLM for robot action logic reasoning. The description of the action in the CoA should align with the provided APIs, which avoids incorrect action sequences that misguide the LLM.

Additional information, such as the LLM's role and output format, is an optional component of the prompts. In robotic assembly tasks, the LLM typically acts as a robotic algorithm engineer, collaborating with a designated robot. The API library is seamlessly integrated into the prompt. Thus, the LLM can generate the main control unit of a robot program comprising the provided APIs.

3.3.3. Low-level policy implementation

Executable third-party libraries are used to build prompts for the LLM to implement low-level policies in high-level functions. The detailed design principles of prompt engineering are as follows:

- **Introduction of third-party libraries**: Demonstrates the usage of libraries with explicit descriptions of their inputs, outputs, and roles. Because of the well-established familiarity of LLMs with widely used third-party libraries such as *NumPy* and *MoveIt*, merely mentioning the names of these libraries within the prompts is sufficient, without necessitating the inclusion of redundant information.
- **Initialization of the robot:** Defines the mechanical structure of the robot and its sensor configurations. Additionally, serial, network, and other interfaces are provided to the LLM to establish communication with the robot through code. Motion constraints, such as goal joint tolerance and maximum velocities, can be included in prompts to satisfy task requirements.

The API library can be directly integrated into this prompt. Alternatively, users have the flexibility to transform APIs into a standardized function format, whereby function names and input arguments serve as function declarations, and the accompanying text descriptions serve as code documentation. When robot information is provided in a programming language, it can be effectively incorporated into a class structure, thereby facilitating parameter invocations within functions. The LLM achieves practical implementation of each function on a robot platform with third-party libraries. Notably, policy codes exhibit low interfunction dependency, making it easy to identify errors within specific functions using this hierarchical method.

3.4. Simulation and optimization

After robot control codes are generated hierarchically, they are composed and assembled into a cohesive control script. The script is subsequently transmitted to the simulation platform, where it is executed to assess the validity and accuracy of the robot control program generated by the LLM. Users can then iteratively optimize the code based on the simulation findings. A schematic of the interaction between the user and the LLM is presented in Fig. 4.

Throughout the simulation process, users typically prioritize two key concerns: (1) Can the task be executed successfully? (2) If so, can it perform better? The first concern pertains to task completion. Hence, two task completion-related metrics are employed: code executability and condition recall. Code executability measures whether a robot control script can be interpreted successfully, including syntactic correctness, verification of indispensable libraries and resources, and compatibility with the robot operating system and computer hardware. Condition recall measures the percentage of conditions in prompts that are satisfied in the codes.

For further analysis, the reasons for incomplete task execution are divided into four types of errors: syntax errors, runtime errors, wrong action, and wrong parameters (Skreta et al., 2023). Among them, syntax errors are errors that occur during code compilation, such as the use of incorrect punctuation; runtime errors occur during the execution phase of code, such as an array out of bounds; wrong parameter refers to logical errors related to parameters, which typically do not impede the standard program execution, albeit fail to align with predetermined task prerequisites; wrong action refers to a logical error related to the action sequence, such as a missing action. The latter two types of errors are logical errors related to condition recall.

Regarding the second concern, task performance-related metrics are applied to improve robot performance. Robot performance can be optimized from several aspects, such as construction efficiency, energy expenditure, and pose accuracy (Dakhli and Lafhaj, 2017; Petersen et al., 2019). For the assembly task in load-bearing construction, the metric proposed in (Petersen et al., 2019) is adopted to assess the construction efficiency of a single-material assembly:

$$\text{construction efficiency} = \frac{V_c}{Time * V_r} \tag{1}$$

where V_c is the constructed volume, V_r is the volumetric size of all deployed robots, and *Time* is the cumulative duration required by the robots to complete the task. When comparing metrics across iterations, the specific objectives of code optimization are identified and prompted into conversations with the LLM.

Thus, prompts are built with the relevant code snippets, errors, and optimization goals, which are then sent to the LLM as feedback. In response, the LLM rectifies the identified errors in the program and provides suggestions to the user for code improvements, such as adjusting parameter settings and altering control algorithms.

4. Experiment

4.1. Assembly task set

To evaluate the feasibility of the proposed approach in different construction scenarios, ten assembly tasks are used to test the method, forming a construction assembly task set, as presented in Table 3. The use of diverse robot types and end effectors can result in disparities in the codes of low-level policies within APIs. Thus, three distinctive types

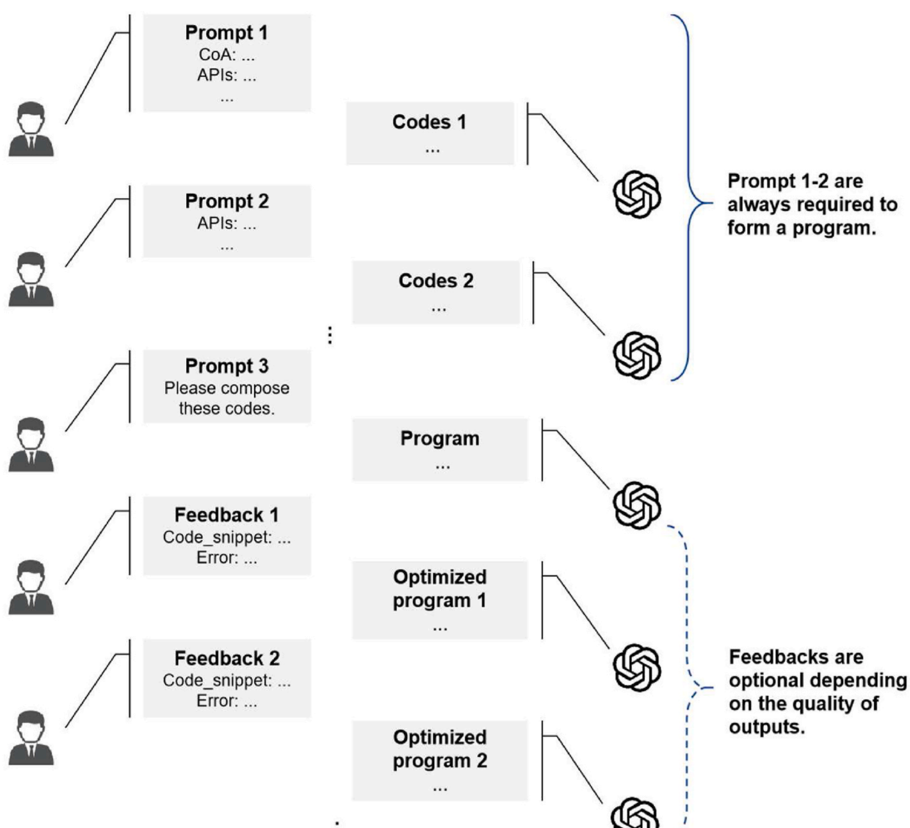

Fig. 4. Interaction between the user and the LLM.

Table 3
Construction assembly task set.

Number	Type of Robot	Task Description	Task Characteristics	Reference
1	Fixed Robot Arm	Use 12 bricks to construct a brick wall.	–	Ding et al. (2020)
2	Mobile Manipulator	Eight beams (four for short edge and four for long edge) and four columns are assembled to a rectangular frame.	Multi-materials assembly	Gao et al. (2022)
3	UAV	Use 40 bricks to construct a foam brick tower.	–	Augugliaro et al. (2014)
4	Fixed Robot Arm	Assemble a rectangular frame with four beams (two for 1.45 m and two for 1.83 m)	Multi-materials assembly	Chong et al. (2022)
5	Mobile Manipulator	A total of nine wooden struts needs to be assembled, three of which are placed by the robot.	Human–robot collaboration; Stability requirement	Mitterberger et al. (2022)
6	Fixed Robot Arm	16 square floor tiles are laid in 4 rows and 4 columns.	–	King et al. (2014)
7	Mobile Manipulator	A suspended grid is built. The target grid is located at the top of the robot arm, and the robot needs to install 9 ceiling tiles.	An angle is required to move the ceiling tile above the suspended grid.	(Liang et al., 2020, 2022)
8	Fixed Robot Arm	Alternated build 8 bricks on the front or back side of the vault.	A small drawing-out movement is required before each brick fit-in step to apply epoxy putty.	Parascho et al. (2020)
9	Mobile Manipulator	Install four slabs on the wall.	Three basic motion modes are defined: move, slide, and lift.	Hu and Cao (2022)
10	UAV	A UAV is used to assemble six square floor tiles.	–	–

of construction assembly robots were employed in the task set: fixed robotic arm, mobile manipulator, and unmanned aerial vehicle (UAV). Owing to the less application of UAV in construction assembly, the ratio of fixed robotic arms, mobile manipulators, and UAV in the task set was set to 4:4:2.

In addition, the action sequence provided in a high-level module can be influenced by the assembly material, construction process, safety requirements, and robot type. Therefore, for the same robot type, the task requirements were tailored differently to introduce variations into the action sequence. Additionally, nine tasks in the task set adopted experimental designs from existing studies, as shown in Table 3. The assembly task set aimed to provide a comprehensive evaluation of the robustness and generalizability of the proposed method by covering common construction assembly materials (e.g., bricks, bars, and tiles), different types of robots, and different assembly processes designed for specific working environments.

4.2. Setting and metrics

To validate the efficacy of the proposed approach, *ChatGPT* (GPT-3.5 Version), a highly acclaimed LLM, was used to generate *Python* codes for robot control (van Dis et al., 2023; OpenAI and ChatGPT, 2023). *ChatGPT* has been trained using instruction following human alignment and large dialog data, making it outstanding in conversational understanding, reasoning, and programming (OpenAI and code, 2023; Zhao et al., 2023). Previous studies, such as AirSim-ChatGPT (Vemprala et al., 2023), RoboGPT (You et al., 2023a; Ye et al., 2023), and KGGPT (Mu et al., 2023), have also applied *ChatGPT* to generate robot programs. In contrast to other models within the GPT family, the utilization of *ChatGPT* in this study eliminates constraints related to cost and account access, thereby empowering construction professionals across various income brackets to engage with, advance, and implement the proposed method. In the following experiments, the controllers and communication modules of mechanical devices were initialized in prompts to inform the LLM of the mechanical structure and sensor configurations of the robot used. Additionally, all robot control programs were generated and executed within a consistent software environment. The detailed settings for software environment and hardware configurations are shown in Table 4.

To verify the quality of AI-generated codes in the experiments, the codes were evaluated on the strict solution of code analysis and code errors were reported. Two key performance indicators, the average number of errors and number of optimization rounds, were used to evaluate the performance. Specifically, the average number of errors in the generated codes of the ten tasks was used to determine the quality of the outputs. For this metric, only errors present in the initial set of generated robot programs were considered. Furthermore, the number of optimization rounds required for each task to rectify all the errors in the programs was reported to indicate the efficacy of iterative code optimization within a closed-loop paradigm.

4.3. Evaluation of results

To evaluate the superiority of the proposed method over existing approaches, we conducted a set of experiments across ten tasks in the task set to evaluate its performance in four critical aspects: (1) the ability to generate programs for diverse assembly tasks, (2) the impact of the HRCPG strategy on the quality of code, (3) the impact of APIs and CoA on the quality of code, and (4) the ability to solve tasks through iterative optimization.

Ability to generate programs for diverse assembly tasks. The availability of the developed approach was compared with methods that use fixed-action templates to generate code automatically (Ding et al., 2020; Huang et al., 2021; Rogeau et al., 2021). Three main factors that determine the diversity of the assembly tasks, considered when constructing the task set in Table 3, were used to evaluate the applicability of the method. The approach is considered feasible when it satisfies the conditions.

- Assembly material: The approach has control primitives for handling assembly materials used, such as grabbing control primitives for bricks and rods and suction cup control primitives for plate materials.
- Robot type: The approach satisfies the control requirements of robots. For example, UAVs require wireless signal transmission and flight attitude control.
- Construction process: The plan skeleton of the approach does not need to be changed to adapt to the construction process. However, adding intermediate points or adjusting parameters to the original action sequence is permitted.

Table 4
The settings of software environment and hardware configurations in the experiments.

Setting	Category	Tool	Code initialization
Software Environment	Operating System	Ubuntu 18.04	–
	Framework	ROS Melodic	–
	Simulation Platform	Gazebo 9	–
	Programming language	Python	–
Mechanical Devices	Robotic Arm	Moveit	moveit_commander.MoveGroupCommander('manipulator')
	Mobile Robot	move_base	actionlib.SimpleActionClient('move_base', MoveBaseAction)
	Gripper	Moveit	moveit_commander.MoveGroupCommander('gripper')
	Pump	–	rospy.init_node('vacuum_pump_controller', anonymous = True) pump_pub = rospy.Publisher('vacuum_pump_control', Bool, queue_size = 1)
	UAV (offboard)	mavors	rospy.wait_for_service('/mavros/cmd/arming') arm_client = rospy.ServiceProxy('/mavros/cmd/arming', CommandBool) state_sub = rospy.Subscriber('mavros/state', State, callback = state_cb) local_pos_pub = rospy.Publisher('mavros/setpoint_position/local', PoseStamped, queue_size = 10) attitude_pub = rospy.Publisher('/mavros/setpoint_raw/attitude', AttitudeTarget, queue_size = 10) tool_pub = rospy.Publisher('/tool_command', Bool, queue_size = 10)

As presented in Fig. 5, the findings reveal that the method proposed in this study exhibits universal applicability to all ten tasks from these three aspects. Approaches relying on fixed templates can adapt to different construction processes but are less adaptable to robot types. Notably, the approach presented in (Ding et al., 2020) encounters additional constraints related to the programming language.

Impact of HRCPG strategy on the quality of code. To evaluate the effectiveness of the proposed HRCPG strategy, two code generation methods were used for comparison: (1) a flat method using a single prompt that encapsulates information about the environment, task, robot, and CoA, as presented in Fig. 6, and (2) the CaP approach proposed in (Liang et al., 2023), an example of which is shown in Fig. 7.

Fig. 8 presents a comparison of the results of the flat, CaP, and proposed methods in terms of code generation efficiency, coherence, and redundancy. Generation efficiency evaluates the cost of a code generation method and is determined by the number of prompts used and the average number of errors in generating the code. As shown in Fig. 8(a), the points of the proposed method are clustered in the lower-left corner, indicating that the number of prompts used and the number of code errors generated were relatively low. In Fig. 8(b), code coherence analysis was performed by quantifying the total number of errors resulting from function encapsulation and hierarchical generation in ten tasks. This included errors resulting from misaligned input–output parameters in function calls and redundant operations across different code blocks. The proposed method achieved the same code coherence performance as the flat method. As shown in Fig. 8(c), code redundancy analysis was conducted by calculating the average number of valid code lines (excluding blank and comment lines) in the generated programs. This result indicates that redundant code and unnecessary multi-layer nesting may exist in the programs generated by the flat and CaP methods.

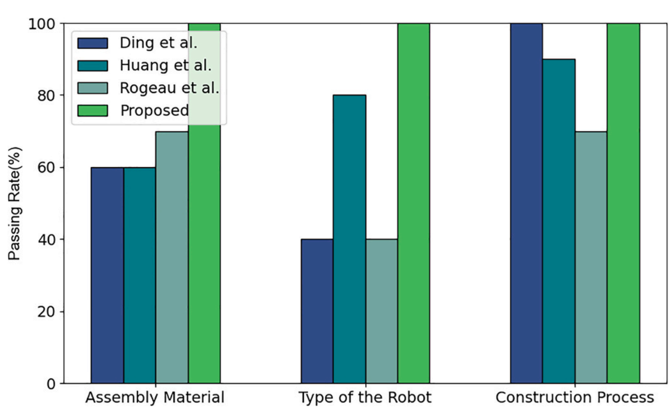

Fig. 5. Applicability comparison of four methods in ten assembly tasks.

Fig. 9 depicts the distribution of the average number of errors among the four types of errors for the flat, CaP, and proposed methods. The code quality generated by the proposed method was significantly better than those generated by the other two methods. Additionally, the results indicate that the proposed method exhibited significantly fewer errors than the other methods in terms of wrong parameter and wrong action. Moreover, wrong action errors were the most frequently occurring error type among the three methods.

Impact of APIs and CoA on the quality of code. Four sets of experiments were conducted across the ten tasks to assess the impact of APIs, CoAs, and their interactions on the quality of AI-generated codes, as presented in Table 5. Two natural language reasoning strategies were adopted: the CoA proposed in this paper, and Vanilla, which does not provide step-by-step information on the assembly process in the dialogs. The results presented in Table 5 indicate that the use of the CoA resulted in a reduction of 1.8 errors per task when APIs were provided to the LLM. However, when APIs were not included, the influence of CoA on code quality appeared to be negligible, as evidenced by the comparable error counts observed in both the CoA and Vanilla scenarios.

The CoA strategy is primarily applied during the high-level module-generation phase, effectively rectifying the action sequences and API invocations presented within the main control unit. To further assess the impact of CoA, we conducted an error analysis of the wrong action in the main control unit separately in the APIs + Vanilla and APIs + CoA scenarios. Remarkably, the average number of errors in the APIs + CoA setting was 1.4, whereas the APIs + Vanilla setting exhibited 0.9 more errors. Fig. 10 depicts the performance of each assembly task for both settings, emphasizing the favorable effects of CoA in generating code pertinent to action sequences.

Ability of task solving through iterative optimization. It was observed that only one out of ten tasks managed to generate the robot program without any syntax and logical errors in the initial interaction round. This observation underscores the significance of iteratively providing feedback to the LLM to optimize the control codes. As shown in Fig. 11, after three rounds of optimization, five tasks resulted in executable control programs, and all tasks were successfully solved after five rounds. On average, nearly three interaction rounds were required for code optimization.

4.4. Use case of brick assembly using a fixed robot arm

In this section, a use case of brick assembly in construction (the first task in the task set) is presented to illustrate how the proposed LLM-based user-on-the-loop program generation approach can improve construction assembly robot control. Notably, it is required to establish a local workspace with essential packages and incorporate robot 3D models, controllers, and third-party libraries for offline robot control.

As depicted in Fig. 12, the brick assembly scenario incorporated a

Imagine we are working with a manipulator robot in a construction site. There is a robot that has a gripper attached to its end effector. I'd like you to use MoveIt! to write Python codes that can be executed at the platform of ROS to drive this robot for implementing the assembly task. Some settings of the robot are as follows:

```
moveit_commander roscpp_initialize(sys.argv)
rospy.init_node('Assembly_Demo')
arm = moveit_commander.MoveGroupCommander('manipulator')
gripper = moveit_commander.MoveGroupCommander('gripper')
end_effector_link = arm.get_end_effector_link()
reference_frame = 'base_link'
arm.set_pose_reference_frame(reference_frame)
arm.allow_replanning(True)
gripper.allow_replanning(True)
arm.set_goal_position_tolerance(0.001)
arm.set_goal_orientation_tolerance(0.005)
```

There are 12 bricks in the scene. In each assembly step, I want you to move robot to the home state and check the state of gripper first, then reach the brick, grab it and move it to the place position. Note that you should move a safe distance (0.3 m) above the object before and after the completion of grabbing and placing.Note that you have initialized 'home' pose for the arm and 'open', 'close' poses for the gripper.

Fig. 6. Example of the prompt for flat code generation.

plane measuring approximately 1.4 m × 1.4 m × 0.42 m, which was designated as the assembly area. The material storage area was located 0.4 m away from the assembly area, housing 12 bricks, each with a volume of 0.2 m × 0.2 m × 0.2 m. An ABB IRB6700-235 robot (ABB, 2023) with a gripper attached to its end-effector was used to complete the assembly task.

In this case, the robot was tasked with constructing a brick wall structure with a straight lay pattern, in which the bricks were assembled in a three-row, four-column arrangement along the XZ-axis plane. For safety, the robot was required to maintain a safe distance of 0.3 m above the bricks before and after manipulation.

Fig. 13 shows the interaction between the user and the LLM to generate a high-level module in the program. As the robot was a fixed arm, the API library was constructed without any functions related to the movement of the robot. It was observed that *ChatGPT* first generated the action plan (the outputs in gray) based on the CoA and then effectively invoked the provided APIs following this action plan. The action plan was accompanied by code comments in natural language, offering valuable insights into how the LLM organized actions to accomplish the task. Thus, these comments proved beneficial for code comprehension and management within the high-level module. However, we must acknowledge that certain errors were identified in the LLM's responses, necessitating correction in the optimization step.

To implement low-level policies, the LLM was directed to employ the third-party library *MoveIt* to generate instruction codes, as depicted in Fig. 14. A class named *AssemblyDemo* contains built-in prompts to furnish the LLM with essential details regarding robot communication and control. The example usage provided in the output demonstrates *ChatGPT*'s swift understanding of the usage of the provided APIs through its ability to discern the input and output requirements for each function from the provided natural language descriptions.

Fig. 15 shows the feedback loop employed to optimize the code using the metrics of code executability and condition recall. Before conducting the simulation in ROS, we integrated the codes generated at the high and low levels into a *Python* script. In the first iteration, an invalid syntax error occurred in the code snippet *brick_name = f"Brick{i}"*, and an error of wrong action was detected in the subsequent iteration. The response from *ChatGPT* contained an assessment of errors and provided corresponding solutions. Through this feedback, *ChatGPT* effectively rectified errors and the task requirements were satisfied.

Fig. 16 presents another feedback loop used to enhance the performance of the robot based on task-related metrics. *ChatGPT* was used to optimize the action sequences or adjust the parameter settings to reduce the total working time while preserving assembly accuracy. In response, *ChatGPT* offered four suggestions along with an updated version of the codes, which were subsequently modified and used to improve robot performance.

To compare the quality of the codes, we denote the optimized codes generated in the first loop as version 1, and those generated from the second loop as version 2. Three experiments were conducted on each version to ensure reliability. The resulting average working times and axial position errors are listed in Table 6. The analysis revealed that the robot driven by the code in version 2 exhibited a significantly reduced working time compared with version 1, with only a marginal increase of 0.02 mm in the average position error.

After the simulation and optimization, a real-world experiment was conducted in a laboratory using an ABB IRB6700-235 robot, as shown in Fig. 17. The bricks used in the experiment were connected using a mortise and tenon structure, which enabled the assembly wall to satisfy the construction strength requirements but required high-precision assembly positions. Before the experiment, we captured the positions of the starting grabbing and placing points by teleoperating the robot and generated the position of each brick by iterating point offsets. Additionally, the *get_pick_position()* and *get_place_position()* functions were defined to support precise assembly.

The *Python* script optimized in the simulation scenario was then run in ROS to control the robot and perform the brick assembly. Following the construction process written in the script, the robot cycled through the six operations depicted in Fig. 17 until all bricks were assembled. Benefiting from the simulation and optimization, the AI-generated codes were executed without errors. However, brick slipping occurred occasionally because of insufficient friction between the gripper jaws and the bricks. This successful application verified the effectiveness of the approach in grounding an AI-generated program in a physical robotic system.

5. Discussion

A framework to generate executable robot control programs specific to construction assembly tasks through iterative interactions between users and LLMs has been proposed. This approach benefits automation in construction assembly by simplifying the robot control process, and it is adaptable to diverse construction assembly tasks. The contributions of this study to the knowledge of construction robot programming are twofold.

First, an HRCPG strategy is designed to logically integrate code

```
import numpy as np
import rospy
import moveit_commander
from env_utils import get_pick_position, get_place_position, get_area_position, move_to, move_joints, go_home, close_gripper,
turn_on_suction_pump, navigate_to, check_gripper
from plan_utils import parse_assembly_order, transform_shape_pts

objects = {'assembly_order': 1, 'object_name': 'white block'}
# rotate the white block clockwise by 40 degrees and place it.
shape_pts = get_place_position(object['assembly_order'], object['object_name'])
new_shape_pts = transform_shape_pts('rotate it clockwise by 40 degrees', shape_pts=shape_pts)
move_joints(new_shape_pts)
open_gripper()

objects = ['yellow block', 'green block', 'yellow bowl', 'blue block', 'blue bowl', 'green bowl']
# imagine that the bowls are different biomes on earth and imagine that the blocks are parts of a building.
# now show the assembly order of objects to build a tower in the grasslands.
assembly_order = parse_assembly_order( 'build a tower in the grasslands' )

# (rest of the examples omitted for brevity)

objects = [f"brick_{i}" for i in range(1, 12)]
# There is a robot arm that has a gripper attach to its end effector. I want you to check the state of gripper first, then reach the brick,
grab it, lift it up and move it to the specific position. Note that you should move a safe distance above the object (0.3 m) before and
after grabbing and placing.
```

```
import sys
import rospy
import moveit_commander
from env_utils import get_pick_position, get_place_position

moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('moveit_move_line', anonymous=True)
robot = moveit_commander.RobotCommander()
arm = moveit_commander.MoveGroupCommander("manipulator")
gripper = moveit_commander.MoveGroupCommander("gripper")

# Define function: move_line(object_position).
def move_line(object_position):
    target_pose = geometry_msgs.msg.Pose()
    target_pose.position.x = object_position[0]
    target_pose.position.y = object_position[1]
    target_pose.position.z = object_position[2]
    arm.set_pose_target(target_pose)
    plan = group.plan()
    arm.execute(plan)
# Define put_first_on_second(arg1, arg2).
def put_first_on_second(arg1, arg2):
    pick_pos = self.get_pick_position(arg1) if isinstance(arg1, str) else arg1
    place_pos = self.get_pick_position(arg2) if isinstance(arg2, str) else arg2
    move_line(pick_pos)
    move_line(place_pos)

# (rest of the examples omitted for brevity)
# Define function: open_gripper()
# Please complete the logic of open_gripper() according to the above function generation examples.
```

Fig. 7. Example of the prompt for CaP approach to generate code.

generation at high and low levels according to the different levels of robot control. In construction assembly tasks, low-level policies of the control function, such as parameter settings and control algorithms, are conditioned by robot configurations and task requirements. The encapsulation and repeated application of control functions across different robot systems and tasks can result in system errors and safety problems in real-world construction scenarios. Therefore, task planning and low-level policy implementation of functions must be customized for different robot assembly tasks. The proposed HRCPG strategy plans the construction process, combines related functions according to construction requirements, and customizes the control functions based on robot configurations. As shown in Figs. 8 and 9, the proposed HRCPG strategy enables the customization of robot control codes at a lower cost and higher accuracy than the flat and CaP methods while effectively avoiding the problems of poor code coherence and high redundancy caused by hierarchical code generation.

In addition, LLMs continuously predict the next token based on a probabilistic model and eventually produce a complete control script by generating codes. The probability of generating a correct script is the product of the probabilities of all the tokens in the control program. Hence, more intricate tasks present a greater challenge in obtaining accurate robot codes, as they often result in longer scripts with a higher number of tokens in the LLM response. Previous studies have acknowledged the limitations of LLMs in generating lengthy programs (Merow

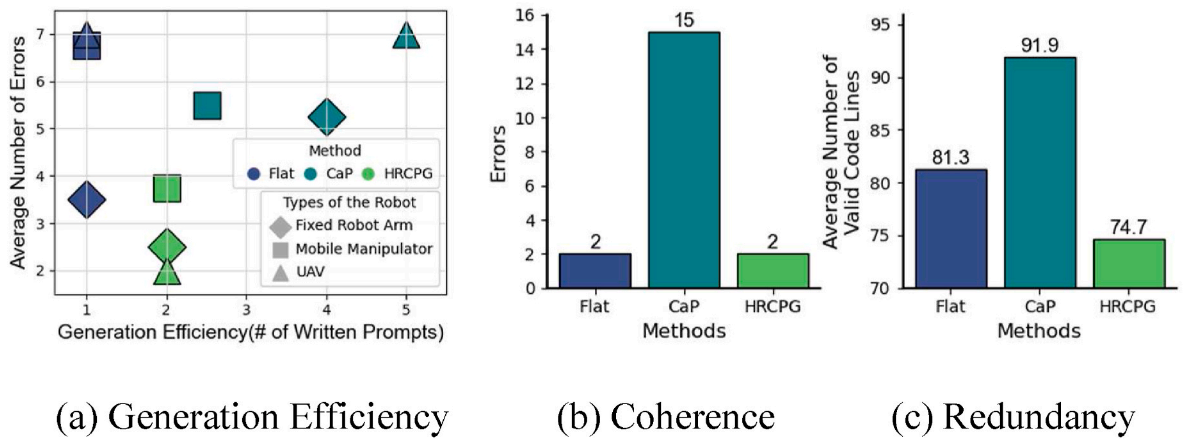

Fig. 8. Comparative analysis of the flat, CaP, and the proposed method in terms of code generation efficiency, coherence, and redundancy.

Fig. 9. Distribution of average number of errors of the flat, CaP, and proposed methods.

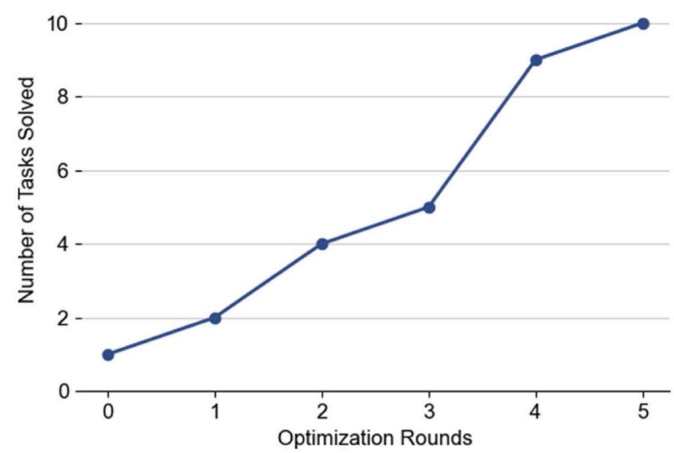

Fig. 11. Number of tasks solved with increasing number of optimization rounds.

Table 5
Number of code errors in different settings of CoA and APIs.

Average number of total errors	APIs	Without APIs
CoA	2.9	5.5
Vanilla	4.7	5.9

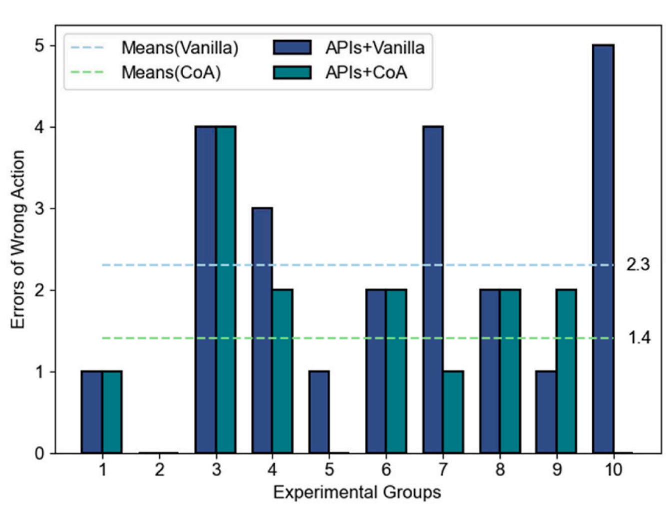

Fig. 10. Error analysis of wrong action in ten tasks.

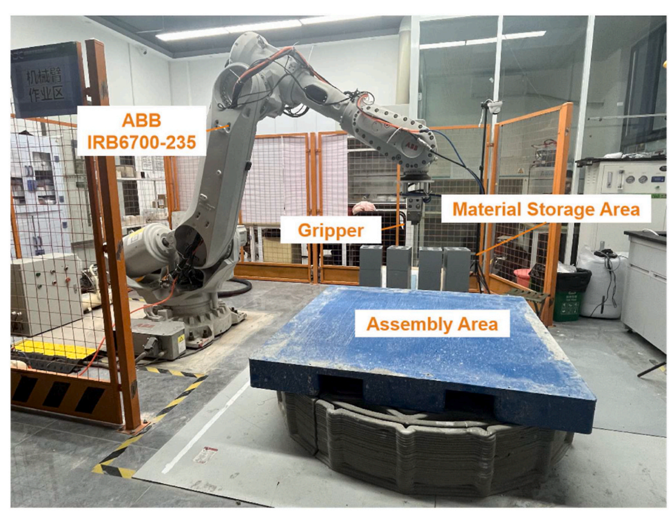

Fig. 12. Experimental setup of the use case.

et al., 2023). The proposed HRCPG strategy can reduce the length of AI-generated tokens and increase the likelihood of generating correct responses. This is achieved by decomposing the generation of complex robot action logic into high-level module generation and low-level policy implementation.

Imagine we are working with a manipulator robot in a construction site. I need your assistance in sending assembly commands to the ABB IRB6700-235 robot equipped with a gripper on its end effector.
You are always accessible to the following functions:
get_pick_position(object_name): Take the object name as input. Return the [X, Y, Z, Yaw, Pitch, Roll] coordinates of the picked object.
get_place_position(assembly_order, object_name=None): Take the assembly order of the current object and object name as input. Note that the attribute 'object_name' is optional when there is only one category of fabrication components, and mandatory when there are two or more. Return the [X, Y, Z, Yaw, Pitch, Roll] coordinates of the object.
move_to(object_position): Move the gripper to a position specified by [X, Y, Z, Yaw, Pitch, Roll] coordinates. Return nothing.
close_gripper(): Close the gripper. Return nothing.
open_gripper(): Open the gripper. Return nothing.
go_home(): Move the robotic arm and its end effector to the home position. Return nothing.
check_gripper(): Check the state of the gripper before grabbing. If it is closed, open it.
You are not allowed to use any other hypothetical functions. Keep the solutions simple and clear.
Use Python code to express your solution.

There are 12 bricks in the scene. In each assembly step, I want you to move robot to the home position and check the state of gripper first, then reach the brick, grab it, lift it up and move it to the specific position. Note that you should move a safe distance above the object (0.3 m) before and after the completion of grabbing and placing.

```
def assemble_bricks():
  for i in range(1, 13):
    brick_name = f"Brick{i}"
    # Move to home state and check gripper state
    go_home()
    check_gripper()
    # Get the pick position of the brick
    pick_position = get_pick_position(brick_name)
    # Calculate the safe distance above the object
    above_pick_position = [pick_position[0], pick_position[1], pick_position[2] + 0.3, pick_position[3], pick_position[4],
pick_position[5]]
    # Move above the brick before grabbing
    move_to(above_pick_position)
    # Close the gripper to grab the brick
    close_gripper()
    # Move the gripper back to the safe distance above the object
    move_to(above_pick_position)
    # Get the place position for the current assembly order and brick name
    place_position = get_place_position(i, brick_name)
    # Calculate the safe distance above the place position
    above_place_position = [place_position[0], place_position[1], place_position[2] + 0.3, place_position[3], place_position[4],
place_position[5]]
    # Move above the place position before placing the brick
    move_to(above_place_position)
    # Move the gripper down to the place position
    move_to(place_position)
    # Open the gripper to release the brick
    open_gripper()
    # Move the gripper back to the safe distance above the place position
    move_to(above_place_position)
  # After assembling all the bricks, go back to the home state
go_home()
```

Fig. 13. Input and output of the LLM for high-level module generation.

Second, an API library was built to enable the LLM to adapt to diverse task requirements and hardware configurations of construction assembly tasks, thereby effectively augmenting the efficiency of code generation. Additionally, the concept of CoA, which provides a lucid sequence of actions to indicate the order of API calls in the main control unit, is introduced to operate in conjunction with the selected APIs. As depicted in Fig. 10, the combined utilization of APIs and CoA enhances the understanding of the LLM in assembly action logic. In the context of the prompts, the accuracy of the API and CoA in delineating assembly tasks and conditions correlates directly with the quality of the LLM output. Our ablation study, detailed in Table 5, investigated the API library and CoA, revealing that AI-generated code errors were halved with the inclusion of API and CoA support. This finding underscores the specificity and effectiveness of the prompts devised in this study for conveying information, indicating that the prompt designed in this paper is more specific and adequate in expressing the information. Similar observations have been reported in previous studies (Merow et al., 2023).

In the experimental analysis presented in Fig. 18, we observed that *ChatGPT* exhibited a spontaneous output of action logic even without explicit requirements provided in the prompt. This finding indicated the possibility that *ChatGPT* may have been trained with CoT, which has also been reported in recent research (Chen et al., 2023). As shown in Table 5, CoA had no impact on the quality of coding in the absence of APIs because the LLM implicitly followed the CoT strategy. However, the CoA significantly improved the performance of the LLM with APIs in the prompt. This enhancement can be attributed to the high consistency between the action descriptions in the CoA and APIs, resulting in more accurate functional invocations. By contrast, the CoT in *ChatGPT* is learned from the training data, and the generated action sequences may not necessarily align with the provided APIs, rendering it less effective as

I'd like you to use the MoveIt! to write codes that can be executed at the platform of ROS to control the moving of ABB IRB6700-235 robot and a gripper. Assuming we have defined and initialized the AssemblyDemo class as follow:

```
class AssemblyDemo:
    def __init__(self):
        moveit_commander.roscpp_initialize(sys.argv)
        rospy.init_node('Assembly_Demo')
        self.arm = moveit_commander.MoveGroupCommander('manipulator')
        self.gripper = moveit_commander.MoveGroupCommander('gripper')
        self.end_effector_link = self.arm.get_end_effector_link()
        self.reference_frame = 'base_link'
        self.arm.set_pose_reference_frame(self.reference_frame)
        self.arm.allow_replanning(True)
        self.gripper.allow_replanning(True)
        self.arm.set_goal_position_tolerance(0.001)
        self.arm.set_goal_orientation_tolerance(0.005)

    def move_to(object_position):
    # Move the gripper to a position specified by [X, Y, Z, Yaw, Pitch, Roll] coordinates. Return nothing.

    def close_gripper():
    # Close the gripper. Return nothing

    def open_gripper():
    # Open the gripper. Return nothing.

    def go_home():
    # Return the robotic arm and its end effector to the home position. Return nothing.

    def check_gripper():
    # Check the state of gripper before grabbing. If it is closed, open it.
```

Note that we have initialized 'home' pose for the arm and 'open', 'close' poses for the gripper. Can you implement move_to(object_position), close_gripper(), open_gripper(), go_home(), check_gripper()?

```
import sys
import rospy
import moveit_commander
import geometry_msgs.msg
import tf

# Codes of AssemblyDemo class.

# Example usage
demo = AssemblyDemo()
demo.go_home()
demo.move_to([0.5, 0.2, 0.3, 0.0, 0.0, 0.0])
demo.check_gripper()
demo.close_gripper()
demo.move_to([0.6, 0.2, 0.3, 0.0, 0.0, 0.0])
demo.open_gripper()
demo.go_home()
```

Fig. 14. Input and output of the LLM for low-level policy implementation.

a substitute for CoA.

6. Limitations and future research

Despite the contributions of this study, we acknowledge that it has the following limitations. First, syntax errors are related to the training dataset of the LLM, whereas logical errors involving task condition implementation are more relevant to the expression of users and the form of prompts. As shown in Fig. 9, an average of 0.7 syntax errors resulted when utilizing our code generation approach. This finding suggests the presence of potentially false information regarding robot control within the training set of *ChatGPT*. Thus, a fine-tuned LLM tailored for code generation for construction robots is required for future research. Robot control programs with higher professionalism and quality can be generated by performing rigorous training data cleaning and efficient fine-tuning techniques such as human feedback-based reinforcement learning methods (Zhao et al., 2023).

Additionally, logical errors can be reduced through a more detailed expression of the task requirements. However, we acknowledge that an increase in the level of detail may introduce inconvenience to users. Thus, striking a balance between providing concise and prompt information and obtaining optimal model outputs is a crucial research question that warrants further investigation. Furthermore, although the prompts are meticulously designed in this paper, the challenge of prompt brittleness persists, manifesting in varied outputs across different LLMs—a significant research obstacle in LLM technology. Future research endeavors will tackle this issue by employing methods such as vector representations (Li and Liang, 2021) and data-driven prompt engineering (Li and Liang, 2021; Shin et al., 2020) to bolster the robustness of the prompts, thereby advancing the reliability and usability of LLM applications.

Second, the proposed approach assumes that the given task and

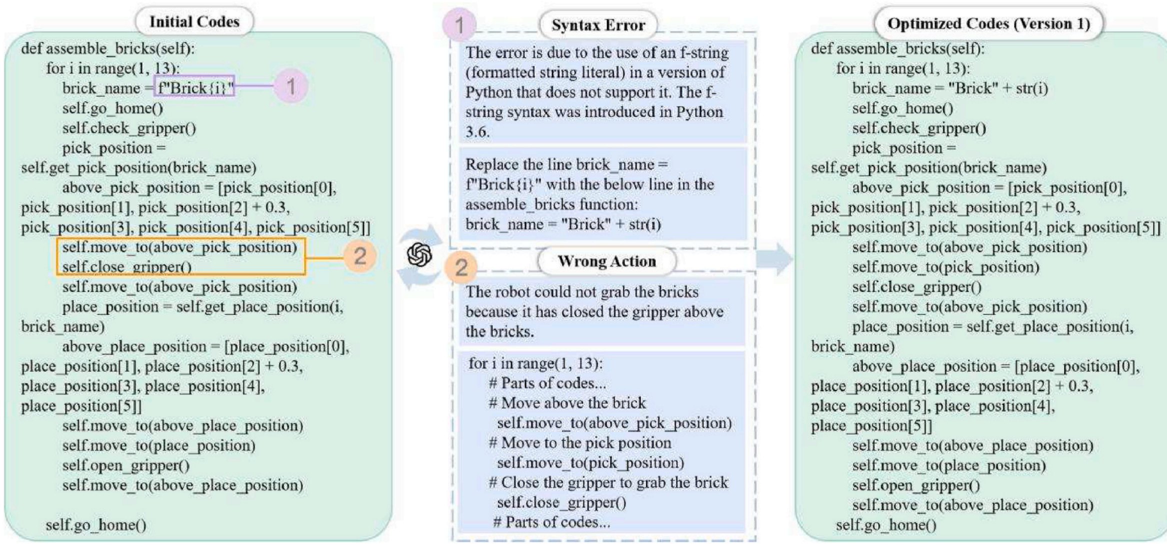

Fig. 15. Feedback loop to optimize codes for task completion.

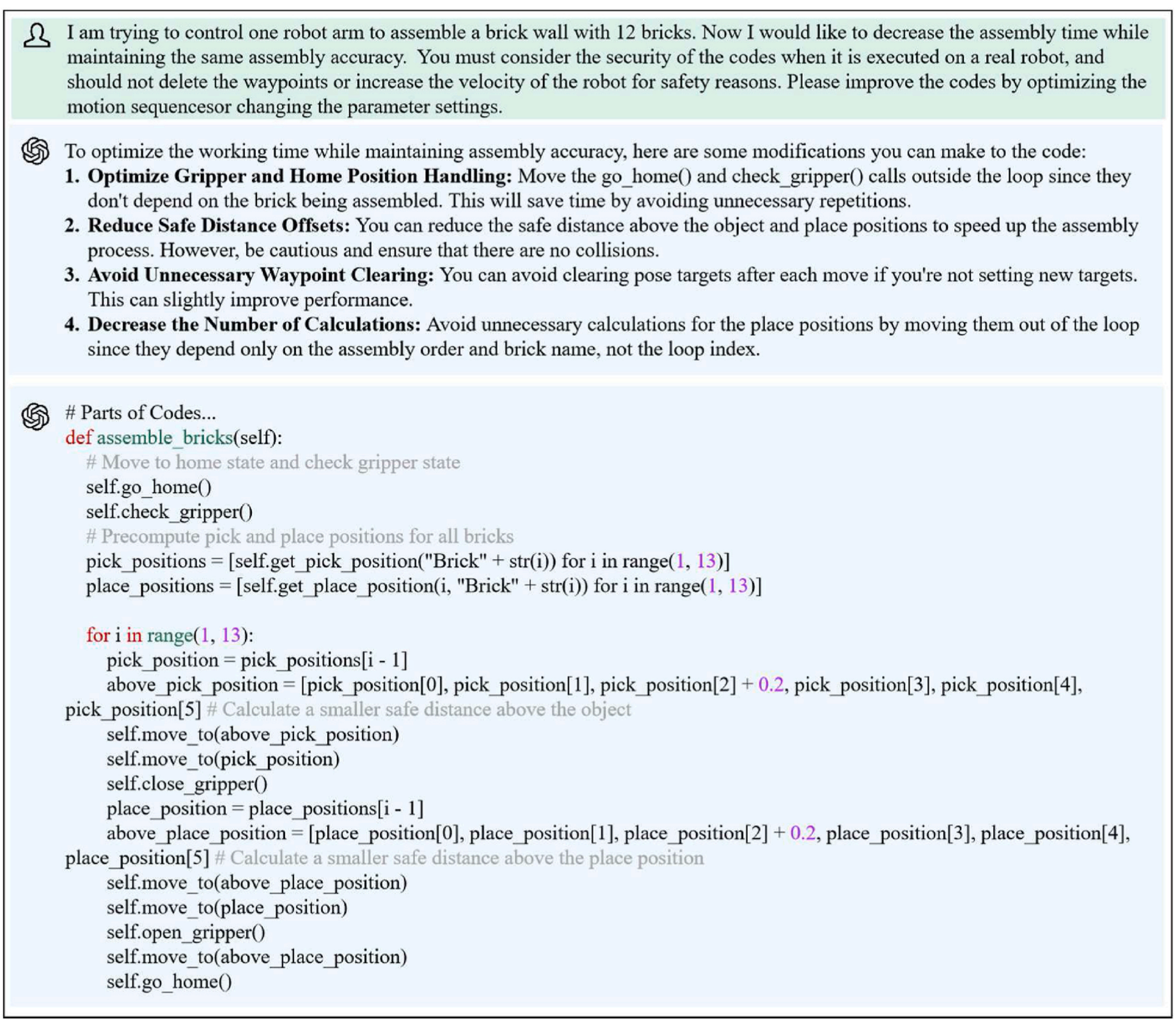

Fig. 16. Response of the LLM to optimize codes for better performance.

generated instructions are always supported and executable, but the constraints imposed by the robot's hardware configuration are not within the scope of consideration. Further research could incorporate retrieval-augmented generation (RAG) technology to assist the LLM in retrieving technical documents of the robot during code generation and optimization, enhancing its understanding of robot skills, and improving

Table 6
Position error and working time of the robot.

Code Version	Axial Position Error (*mm*)			Position Error (*mm*)	Time (*s*)
	X-axis	Y-axis	Z-axis		
Version 1	0.28	0.22	2.25	0.92	394.90
Version 2	0.37	0.22	2.25	0.94	327.61

the accuracy of API calls (Boiko et al., 2023). Additionally, complex physical constraints (such as data noise, static friction, and structural vibration) can affect the effectiveness of plan execution during real-world experiments. For instance, insufficient friction between gripper jaws and bricks can result in slips, whereas brick assemblies in the form of mortise and tenon joints can suffer severe collision problems owing to slight positional errors. However, these factors accurately are difficult to model using existing simulation platforms. Integrating multi-dimensional sensor data, such as force and vision sensing, as realistic feedback to the proposed framework can enable dual-loop optimal control of the robot in simulation and reality, thereby effectively avoiding safety problems and improving assembly accuracy.

Lastly, the utilization of *ChatGPT* introduces additional constraints. Rate limits, such as those governing requests and tokens per minute, impede extensive testing and continuous usage. Scaling the solution upwards escalates the expenses associated with API calls, thereby prompting financial considerations for long-term adoption. Moreover, transmitting sensitive user information to third-party services heightens the risk of data security breaches and privacy violations during transmission and storage. Therefore, this further underscores the imperative for developing a LLM tailored for code generation for construction robotics.

7. Conclusions

This study aimed to address the following research question: *How can robot control programs be generated effectively and accurately for diverse construction assembly tasks using LLM techniques?* To address this research question, a closed user-on-the-loop robot control workflow based on LLM techniques is proposed. An HRCPG strategy was designed to achieve high-level module generation and low-level policy implementation. Additionally, a customized API library and CoA were used to prompt an LLM to enhance the understanding of user intentions. To evaluate the effectiveness of the proposed approach, we conducted experiments involving ten distinct construction assembly tasks encompassing various construction materials (e.g., bricks, bars, and tiles), diverse robot types (e.g., fixed robot arm, mobile manipulator, and UAV), and different assembly processes. The experimental results show that (1) the approach can implement closed-loop control on the construction robot without the high burden of programming work on users, and (2) the proposed code generation method can effectively improve the quality of AI-generated codes by decreasing the number of code errors from 5.9 to 2.9.

The approach presented in this paper breaks the technical barriers between users and robots by employing an LLM as a medium intermediary to translate user intentions into comprehensible robot commands. This method facilitates automation in construction assembly by simplifying the process of robot control and can be extended to more complex scenarios. The contributions of this research are twofold: (1) the design of the HRCPG strategy to logically integrate code generation at high and low levels according to different levels of robot control; and (2) the combination of APIs and CoA to enhance the understanding of the LLM to the assembly action logic. Future studies should focus on developing a fine-tuned LLM tailored for code generation for construction robots and integrating multidimensional sensor data during operation.

CRediT authorship contribution statement

Hanbin Luo: Writing – original draft, Methodology, Funding acquisition, Conceptualization. **Jianxin Wu:** Writing – original draft, Methodology. **Jiajing Liu:** Writing – review & editing, Writing – original draft, Methodology, Funding acquisition, Conceptualization. **Maxwell Fordjour Antwi-Afari:** Software, Resources.

Declaration of competing interest

This manuscript has not been published or presented elsewhere in part or in entirety and is not under consideration by another journal. We have read and understood your journal's policies, and we believe that neither the manuscript nor the study violates any of these. There are no conflicts of interest to declare.

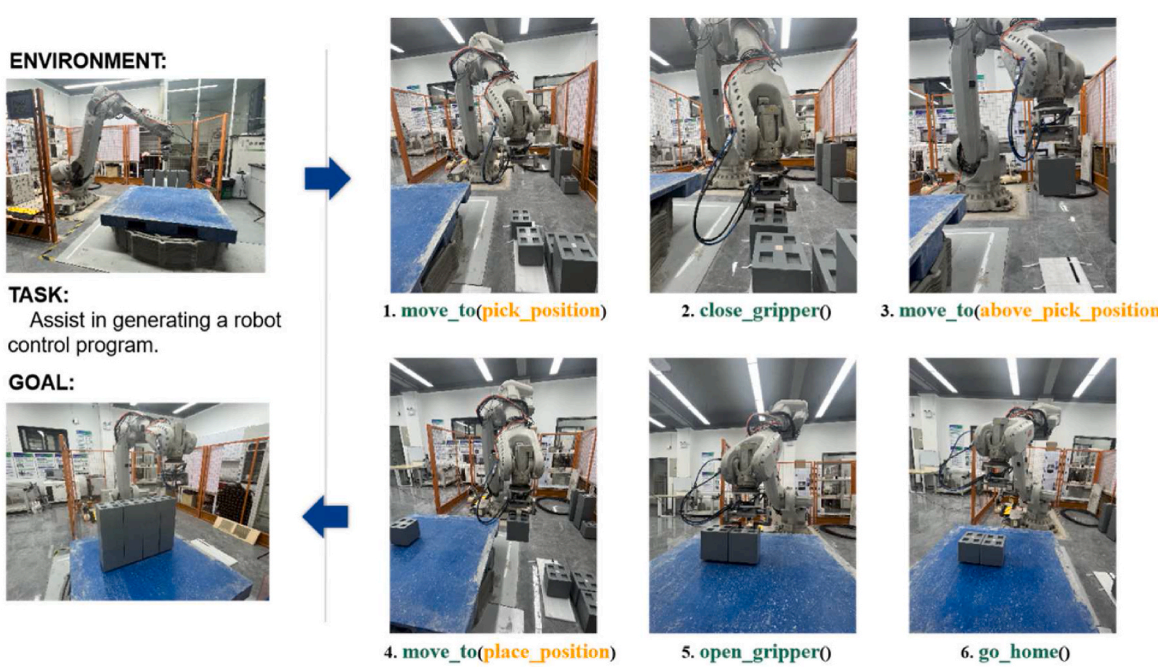

Fig. 17. Brick assembly process of the robot.

There are 12 bricks in the scene. I want you to assemble each brick one by one. Note that you should move a safe distance above the object (0.3 m) before and after the completion of grabbing and placing.

Sure, I can help you with that. Let's start by assembling each brick one by one. We'll follow these steps for each brick:
1. Get the pick position of the brick using the **get_pick_position** function.
2. Move the gripper to a safe distance above the brick by subtracting 0.3m from the Z-coordinate.
3. Move the gripper to the pick position obtained in step 1.
4. Check the gripper state using the **check_gripper** function and open it if necessary.
5. Close the gripper using the **close_gripper** function.
6. Move the gripper to a safe distance above the brick by adding 0.3m to the Z-coordinate.
7. Get the place position of the brick using the **get_place_position** function.
8. Move the gripper to the place position obtained in step 7.
9. Open the gripper using the **open_gripper** function.
10. Move the gripper to a safe distance above the brick by subtracting 0.3m from the Z-coordinate.
11. Repeat the above steps for the remaining bricks.

Here's a Python code snippet that implements the assembly process for the 12 bricks:

```
# Assembly process for each brick
for i in range(1, 13):
   brick_name = f"brick{i}"
   # Get the pick position of the brick
   pick_position = get_pick_position(brick_name)
   # Move gripper above the brick
   safe_pick_position = [pick_position[0], pick_position[1], pick_position[2] - 0.3, pick_position[3], pick_position[4],
pick_position[5]]
   move_to(safe_pick_position)
   # Move gripper to the pick position
   move_to(pick_position)
   # Check gripper state and open if necessary
   check_gripper()
   # Close the gripper
   close_gripper()
   # Move gripper above the brick
   safe_place_position = get_place_position(i, brick_name)
   safe_place_position[2] += 0.3
   move_to(safe_place_position)
   # Move gripper to the place position
   move_to(get_place_position(i, brick_name))
   # Open the gripper
   open_gripper()
   # Move gripper above the brick
   safe_place_position[2] -= 0.3
   move_to(safe_place_position)
```

Fig. 18. Experiment in setting up APIs + Vanilla.

Data availability

Data will be made available on request.

Acknowledgements

The authors acknowledge the financial support of the National Key R&D Program of China (No. 2023YFC3806605), the National Natural Science Foundation of China (Grant Nos. 72301114 and U21A20151) and China Postdoctoral Science Foundation (Grant No. 2023M731187) toward conducting the research presented in this paper. Access to the code used in this study will be made available upon request from the corresponding author.

References

ABB, I.R.B., 2023. 6700 robots - industrial robots from ABB robotics. https://new.abb.com/products/robotics/robots/articulated-robots/irb-6700. (Accessed 29 October 2023).

Augugliaro, F., Lupashin, S., Hamer, M., Male, C., Hehn, M., Mueller, M.W., Willmann, J. S., Gramazio, F., Kohler, M., D'Andrea, R., 2014. The flight assembled architecture installation: cooperative construction with flying machines. IEEE Control Syst. Mag. 34 (4), 46–64. https://doi.org/10.1109/Mcs.2014.2320359.

Boiko, D.A., MacKnight, R., Kline, B., Gomes, G., 2023. Autonomous chemical research with large language models. Nature 624 (7992), 570–578. https://doi.org/10.1038/s41586-023-06792-0.

Bonilla, F.S., Ugalde, F.R., 2019. Automatic translation of Spanish natural language commands to control robot comands based on lstm neural network. 2019 Third IEEE International Conference on Robotic Computing, pp. 125–131. https://doi.org/10.1109/IRC.2019.00026.

Bruckmann, T., Boumann, R., 2021. Simulation and optimization of automated masonry construction using cable robots. Adv. Eng. Inf. 50, 101388 https://doi.org/10.1016/j.aei.2021.101388.

Cai, S., Ma, Z., Skibniewski, M.J., Bao, S., 2019. Construction automation and robotics for high-rise buildings over the past decades: a comprehensive review. Adv. Eng. Inf. 42, 100989 https://doi.org/10.1016/j.aei.2019.100989.

Cai, S., Ma, Z., Skibniewski, M.J., Guo, J., 2020. Construction automation and robotics: from one-offs to follow-ups based on practices of Chinese construction companies. J. Construct. Eng. Manag. 146 (10), 05020013 https://doi.org/10.1061/(ASCE)CO.1943-7862.0001910.

Carvalho, G.C., Siqueira, M.L., Absi-Alfaro, S.C., 1998. Off-line programming of flexible welding manufacturing cells. J. Mater. Process. Technol. 78 (1–3), 24–28. https://doi.org/10.1016/S0924-0136(97)00458-5.

Chen, J., Chen, L., Huang, H., Zhou, T., 2023. When do you need Chain-of-Thought prompting for ChatGPT? arXiv preprint arXiv:2304.03262. https://doi.org/10.48550/arXiv.2304.03262.

Chitta, S., 2016. MoveIt!: an introduction. In: Koubaa, A. (Ed.), Robot Operating System (ROS). Studies in Computational Intelligence, vol 625. Springer, Cham. https://doi.org/10.1007/978-3-319-26054-9_1.

Chong, O.W., Zhang, J., Voyles, R.M., Min, B.-C., 2022. BIM-based simulation of construction robotics in the assembly process of wood frames. Autom. ConStruct. 137, 104194 https://doi.org/10.1016/j.autcon.2022.104194.

Dakhli, Z., Lafhaj, Z., 2017. Robotic mechanical design for brick-laying automation. Cogent Eng. 4 (1), 1361600 https://doi.org/10.1080/23311916.2017.1361600.
Ding, L., Jiang, W., Zhou, Y., Zhou, C., Liu, S., 2020. BIM-based task-level planning for robotic brick assembly through image-based 3D modeling. Adv. Eng. Inf. 43, 100993 https://doi.org/10.1016/j.aei.2019.100993.
Gao, Y., Meng, J., Shu, J., Liu, Y., 2022. BIM-based task and motion planning prototype for robotic assembly of COVID-19 hospitalisation light weight structures. Autom. ConStruct. 140, 104370 https://doi.org/10.1016/j.autcon.2022.104370.
Gao, Y., Shu, J., Xiao, W., Jin, Z., 2023. Polyhedron-bounded collision checks for robotic assembly of structural components. Autom. ConStruct. 152, 104904 https://doi.org/10.1016/j.autcon.2023.104904.
Hu, H., Cao, J., 2022. Adaptive variable impedance control of dual-arm robots for slabstone installation. ISA (Instrum. Soc. Am.) Trans. 128 (Pt A), 397–408. https://doi.org/10.1016/j.isatra.2021.10.020.
Hu, H., Chen, J., Liu, H., Li, Z., Huang, L., 2022. Natural language-based automatic programming for industrial robots. J. Grid Comput. 20 (3), 26. https://doi.org/10.1007/s10723-022-09618-x.
Huang, Y., Garrett, C.R., Mueller, C.T., 2018. Automated sequence and motion planning for robotic spatial extrusion of 3D trusses. Constr. Robot. 2 (1–4), 15–39. https://doi.org/10.1007/s41693-018-0012-z.
Huang, Y., Garrett, C.R., Ting, I., Parascho, S., Mueller, C.T., 2021. Robotic additive construction of bar structures: unified sequence and motion planning. Constr. Robot. 5, 115–130. https://doi.org/10.1007/s41693-021-00062-z.
Huang, W., Xia, F., Xiao, T., Chan, H., Liang, J., Florence, P., Zeng, A., Tompson, J., Mordatch, I., Chebotar, Y., Sermanet, P., Brown, N., Jackson, T., Luu, L., Levine, S., Hausman, K., Ichter, B., 2022. Inner monologue: embodied reasoning through planning with language models. arxiv preprint arxiv:2207.05608. https://doi.org/10.48550/arXiv.2207.05608.
Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., Sharma, R., 2022. Jigsaw: large language models meet program synthesis. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1219–1231. https://doi.org/10.1145/3510003.3510203.
Kahuttanaseth, W., Dressler, A., Netramai, C., 2018. Commanding mobile robot movement based on natural language processing with RNN encoderdecoder. In: 2018 5th International Conference on Business and Industrial Research, pp. 161–166. https://doi.org/10.1109/ICBIR.2018.8391185.
Kim, S., Peavy, M., Huang, P.C., Kim, K., 2021. Development of BIM-integrated construction robot task planning and simulation system. Autom. ConStruct. 127, 103720 https://doi.org/10.1016/j.autcon.2021.103720.
King, N., Bechthold, M., Kane, A., Michalatos, P., 2014. Robotic tile placement: tools, techniques and feasibility. Autom. ConStruct. 39, 161–166. https://doi.org/10.1016/j.autcon.2013.08.014.
Koubaa, A., 2023. ROSGPT: next-generation human-robot interaction with ChatGPT and ROS. Preprints, 2023040827. https://doi.org/10.20944/preprints202304.0827.v3.
Li, X.L., Liang, P., 2021. Prefix-tuning: optimizing continuous prompts for generation. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, pp. 4582–4597. https://doi.org/10.18653/v1/2021.acl-long.353.
Liang, C.J., Kamat, V.R., Menassa, C.C., 2020. Teaching robots to perform quasi-repetitive construction tasks through human demonstration. Autom. ConStruct. 120, 103370 https://doi.org/10.1016/j.autcon.2020.103370.
Liang, C.J., Kamat, V.R., Menassa, C.C., McGee, W., 2022. Trajectory-based skill learning for overhead construction robots using generalized cylinders with orientation. J. Comput. Civ. Eng. 36 (2), 04021036 https://doi.org/10.1061/(Asce)Cp.1943-5487.0001004.
Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P., Zeng, A., 2023. Code as policies: language model programs for embodied control. In: 2023 IEEE International Conference on Robotics and Automation, pp. 9493–9500. https://doi.org/10.1109/ICRA48891.2023.10160591.
Liu, P.F., Yuan, W.Z., Fu, J.L., Jiang, Z.B., Hayashi, H., Neubig, G., 2023. Pre-train, prompt, and predict: a systematic survey of prompting methods in natural language processing. ACM Comput. Surv. 55 (9), 1–35. https://doi.org/10.1145/3560815.
Merow, C., Serra-Diaz, J.M., Enquist, B.J., Wilson, A.M., 2023. AI chatbots can boost scientific coding. Nature Ecol. Evol. 7 (7), 960–962. https://doi.org/10.1038/s41559-023-02063-3.
Mitterberger, D., Atanasova, L., Dörfler, K., Gramazio, F., Kohler, M., 2022. Tie a knot: human-robot cooperative workflow for assembling wooden structures using rope joints. Constr. Robot. 6 (3–4), 277–292. https://doi.org/10.1007/s41693-022-00083-2.
Mu, Z., Zhao, W., Yin, Y., Xi, X., Song, W., Gu, J., Zhu, S., 2023. KGGPT: empowering robots with OpenAI's ChatGPT and knowledge graph. In: International Conference on Intelligent Robotics and Applications. Springer Nature Singapore, Singapore, pp. 340–351. https://doi.org/10.1007/978-981-99-6495-6_29.
OpenAI, ChatGPT, 2023. https://openai.com/blog/chatgpt/. (Accessed 8 April 2023).
OpenAI, code-davinci-002. https://platform.openai.com/docs/model-index-for-researchers, 2023–. (Accessed 13 November 2023).
Parascho, S., Han, I.X., Walker, S., Beghini, A., Bruun, E.P., Adriaenssens, S., 2020. Robotic vault: a cooperative robotic assembly method for brick vault construction. Constr. Robot. 4, 117–126. https://doi.org/10.1007/s41693-020-00041-w.
Petersen, K.H., Napp, N., Stuart-Smith, R., Rus, D., Kovac, M., 2019. A review of collective robotic construction. Sci. Robot. 4 (28), eaau8479. https://doi.org/10.1126/scirobotics.aau8479.
Ren, B., Wang, Y., Chen, J., Chen, S., 2021. A novel nonlinear disturbance observer embedded second-order finite time tracking-based controller for robotic manipulators. J. Comput. Inf. Sci. Eng. 21 (6), 061005 https://doi.org/10.1115/1.4050470.
Rogeau, N., Latteur, P., Weinand, Y., 2021. An integrated design tool for timber plate structures to generate joints geometry, fabrication toolpath, and robot trajectories. Autom. ConStruct. 130, 103875 https://doi.org/10.1016/j.autcon.2021.103875.
Saka, A., Taiwo, R., Saka, N., Salami, B.A., Ajayi, S., Akande, K., Kazemi, H., 2024. GPT models in construction industry: opportunities, limitations, and a use case validation. Dev. Built Environ. 17, 100300 https://doi.org/10.1016/j.dibe.2023.100300.
Shin, T., Razeghi, Y., Logan IV, R.L., Wallace, E., Singh, S., 2020. Autoprompt: eliciting knowledge from language models with automatically generated prompts. arXiv preprint arXiv:2010.15980. https://doi.org/10.48550/arXiv.2010.15980.
Siciliano, B., Sciavicco, L., Villani, L., Oriolo, G., 2008. Robotics: Modelling, Planning and Control. Springer Publishing Company, Incorporated. ISBN: 1846286417.
Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., Garg, A., 2023. Progprompt: generating situated robot task plans using large language models. In: 2023 IEEE International Conference on Robotics and Automation, pp. 11523–11530. https://doi.org/10.1109/ICRA48891.2023.10161317.
Skreta, M., Yoshikawa, N., Arellano-Rubach, S., Ji, Z., Kristensen, L.B., Darvish, K., Aspuru-Guzik, A., Shkurti, F., Garg, A., 2023. Errors are useful prompts: instruction guided task programming with verifier-assisted iterative prompting. arXiv preprint arXiv:2303.14100. https://doi.org/10.48550/arXiv.2303.14100.
Sucan, I.A., Moll, M., Kavraki, L.E., 2012. The open motion planning library. IEEE Robot. Autom. Mag. 19 (4), 72–82. https://doi.org/10.1109/MRA.2012.2205651.
Teven, L.S., Alexander, R., 2021. How many data points is a prompt worth?. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics, pp. 2627–2636. https://doi.org/10.18653/v1/2021.naacl-main.208.
Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., Lample, G., 2023. Llama: open and efficient foundation language models. arXiv preprint arXiv:2302.13971. https://doi.org/10.48550/arXiv.2302.13971.
van Dis, E.A.M., Bollen, J., Zuidema, W., van Rooij, R., Bockting, C.L., 2023. ChatGPT: five priorities for research. Nature 614 (7947), 224–226. https://doi.org/10.1038/d41586-023-00288-7.
Vann, W., Zhou, T., Zhu, Q., Du, E., 2023. Enabling automated facility maintenance from articulated robot Collision-Free designs. Adv. Eng. Inf. 55, 101820 https://doi.org/10.1016/j.aei.2022.101820.
Vemprala, S., Bonatti, R., Bucker, A., Kapoor, A., 2023. Chatgpt for robotics: design principles and model abilities. arXiv preprint arXiv:2306.17582. https://doi.org/10.48550/arXiv.2306.17582.
Wallhoff, F., Blume, J., Bannat, A., Rösel, W., Lenz, C., Knoll, A., 2010. A skill-based approach towards hybrid assembly. Adv. Eng. Inf. 24 (3), 329–339. https://doi.org/10.1016/j.aei.2010.05.013.
Wang, X., Liang, C.J., Menassa, C.C., Kamat, V.R., 2021. Interactive and immersive process-level digital twin for collaborative human-robot construction work. J. Comput. Civ. Eng. 35 (6), 04021023 https://doi.org/10.1061/(Asce)Cp.1943-5487.0000988.
Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, brian, Xia, F., Chi, E., Le, Q.V., Zhou, D., 2022. Chain of thought prompting elicits reasoning in large language models. Adv. Neural Inf. Process. Syst. 35, 24824–24837. In: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.
Yang, T., Zhang, B., Hong, H., Chen, Y., Yang, H., Wang, T., Cao, D., 2022. Motion control for earth excavation robot based on force pre-load and cross-coupling compensation. Autom. ConStruct. 141, 104402 https://doi.org/10.1016/j.autcon.2022.104402.
Ye, Y., You, H.X., Du, J., 2023. Improved trust in human-robot collaboration with ChatGPT. IEEE Access 11, 55748–55754. https://doi.org/10.1109/Access.2023.3282111.
You, H., Ye, Y., Zhou, T., Zhu, Q., Du, J., 2023a. Robot-enabled Construction Assembly with Automated Sequence Planning Based on ChatGPT: RoboGPT. https://doi.org/10.48550/arXiv.2304.11018 arxiv preprint arxiv:2304.11018.
You, K., Zhou, C., Ding, L., 2023b. Deep learning technology for construction machinery and robotics. Autom. ConStruct. 150, 104852 https://doi.org/10.1016/j.autcon.2023.104852.
Zhang, M., Xu, R., Wu, H., Pan, J., Luo, X., 2023. Human–robot collaboration for on-site construction. Autom. ConStruct. 150, 104812 https://doi.org/10.1016/j.autcon.2023.104812.
Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J., Wen, J., 2023. A survey of large language models. arXiv preprint arXiv:2303.18223. https://doi.org/10.48550/arXiv.2303.18223.
Zheng, C., Xing, J., Wang, Z., Qin, X., Eynard, B., Li, J., Bai, J., Zhang, Y., 2022. Knowledge-based program generation approach for robotic manufacturing systems. Robot. Comput. Integrated Manuf. 73, 102242 https://doi.org/10.1016/j.rcim.2021.102242.
Zhu, A., Pauwels, P., De Vries, B., 2021. Smart component-oriented method of construction robot coordination for prefabricated housing. Autom. ConStruct. 129, 103778 https://doi.org/10.1016/j.autcon.2021.103778.