Kaunas University of Technology

Faculty of Electrical and Electronics Engineering

Development and Research on Methodology for Intuitive Robot Programming Using Large Language Models

Master's Final Degree Project

Augustin Averkin

Project author

Assoc. Prof. Dr. Virginijus Baranauskas

Supervisor

Kaunas, 2026

Kaunas University of Technology

Faculty of Electrical and Electronics Engineering

Development and Research on Methodology for Intuitive Robot Programming Using Large Language Models

Master's Final Degree Project

Control Technology 6211EX014

Augustin Averkin

Project author

Assoc. Prof. Dr. Virginijus Baranauskas

Supervisor

Assoc. Prof. Dr. Gintaras Dervinis

Reviewer

Kaunas, 2026

Kaunas University of Technology

Faculty of Electrical and Electronics Engineering

Augustin Averkin

Development and Research on Methodology for Intuitive Robot Programming Using Large Language Models

Declaration of Academic Integrity

I confirm that the final project of mine, Augustin Averkin, on the topic „Development and Research on Methodology for Intuitive Robot Programming Using Large Language Models “ is written completely by myself; all the provided data and research results are correct and have been obtained honestly. None of the parts of this thesis have been plagiarised from any printed, Internet-based or otherwise recorded sources. All direct and indirect quotations from external resources are indicated in the list of references. No monetary funds (unless required by Law) have been paid to anyone for any contribution to this project.

I fully and completely understand that any discovery of any manifestations/case/facts of dishonesty inevitably results in me incurring a penalty according to the procedure(s) effective at Kaunas University of Technology.

Augustin Averkin

Aprooved electronicaly

Averkin Augustin. Development and Research on Methodology for Intuitive Robot Programming Using Large Language Models. Master's Final Degree Project / supervisor Assoc. Prof. Dr. Virginijus Baranauskas; Faculty of Electrical and Electronics Engineering, Kaunas University of Technology.

Study field and area (study field group): electronics engineering, engineering sciences.

Keywords: Intuitive robot programming; industrial robots; natural language processing; large language models; LLM-based task planning; skill-based robot programming; Behaviour Trees; CoppeliaSim simulation; modular robot control;

Kaunas, 2026. 51 pages.

Summary

Programming of industrial robots typically requires specialized knowledge of robot programming language, kinematics, and system configuration, which limits accessibility and slows deployment of robotic solutions. This paper proposes a methodology for intuitive robot programming using natural language instructions combined with large language model (LLM) based task planning. The proposed system employs a layered architecture consisting of a simulation environment, a Redis-based shared data layer, Python-based robot skills, a Behaviour Tree execution framework and an LLM-based planning module. Natural language instructions are converted into structured task plans that reference predefined robot skills and are executed through Behaviour Trees to ensure modular control and reliable failure handling. The system is evaluated in a simulated robot environment using CoppeliaSim, where 88 tasks with varying complexity are executed to assess planning reliability, execution success rate, and computational performance. Experimental results show that the proposed approach can successfully generate and execute task plans for most tested scenarios, particularly when instructions are clearly defined. The results demonstrate the potential of combining LLM-based planning with skill-based execution frameworks to support more intuitive and flexible robot programming.

Averkin Augustin. Metodikos, skirtos intuityviam robotų programavimui naudojant didžiuosius kalbos modelius, kūrimas ir tyrimas. Magistro baigiamasis projektas / doc. dr. Virginijus Baranauskas; Kauno technologijos universitetas, Elektros ir Elektronikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): elektronikos inžinerija, inžinerijos mokslai.

Reikšminiai žodžiai: Intuityvus robotų programavimas; pramoniniai robotai; natūralios kalbos apdorojimas; didieji kalbos modeliai; LLM grįstas užduočių planavimas; įgūdžiais grįstas robotų programavimas; elgsenos medžiai; CoppeliaSim simuliacija; modulinis robotų valdymas;

Kaunas, 2026. 51 p.

Santrauka

Pramoninių robotų programavimas paprastai reikalauja specializuotų žinių apie robotų programavimo kalbas, kinematiką ir sistemos konfigūravimą. Tai riboja šių sistemų prieinamumą ir lėtina robotizuotų sprendimų diegimą. Šiame darbe siūloma intuityvi robotų programavimo metodika, grįsta natūralios kalbos instrukcijomis ir didžiųjų kalbos modelių pagrindu veikiančiu užduočių planavimu. Siūlomoje sistemoje taikoma sluoksninė architektūra, kurią sudaro simuliacinė aplinka, Redis grįstas bendras duomenų sluoksnis, Python kalba aprašyti robotų įgūdžiai, elgsenos medžiu grįsta vykdymo sistema ir didžiuoju kalbos modeliu (LLM) grįstas planavimo modulis. Darbe pristatoma metodika, kuria natūralios kalbos instrukcijos paverčiamos struktūrizuotais užduočių planais, kuriuose nurodomi iš anksto apibrėžti roboto įgūdžiai. Šie planai vykdomi naudojant elgsenos medžius, užtikrinant modulinį valdymą ir patikimą klaidų apdorojimą. Sukurta Sistema įvertinta simuliacinėje roboto aplinkoje naudojant CoppeliaSim. Bandymų metu įvykdytos 88 skirtingo sudėtingumo užduotys, siekiant įvertinti planavimo patikimumą, vykdymo sėkmės rodiklį ir skaičiavimo našumą. Eksperimentiniai rezultatai rodo, kad siūlomas metodas daugumoje testuotų scenarijų gali sėkmingai generuoti ir vykdyti užduočių planus, ypač kai instrukcijos yra aiškiai apibrėžtos. Rezultatai atskleidžia LLM grįsto planavimo ir įgūdžiais paremto vykdymo sistemų derinimo potencialą kuriant intuityvesnį ir lankstesnį robotų programavimą.

Table of Contents

List of figures

List of tables

List of abbreviations and terms

Abbreviations:

AI - Artificial Intelligence; a field of computer science focused on creating systems capable of performing tasks that normally require human intelligence.

BT – Behaviour Tree; a modular control structure used to represent and execute robot behaviour as a tree of actions and conditions.

CAD – Computer-Aided Design; software-based design tools often used in indirect robot programming and simulation.

CLIP – Contrastive Language-Image Pre-training; a vision-language model used to connect visual information with natural language descriptions.

HRI – Human-Robot Interaction; the study and design of interactions between humans and robotic systems.

IK – Inverse Kinematics; a method for calculating robot joint positions required to reach a desired end-effector pose.

JSON – JavaScript Object Notation; a structured data format used in this research to represent task plans generated by LLM

LLM – Large Language Model; an artificial intelligence model capable of interpreting and generating natural language text.

NLP – Natural Language Processing; a field of AI focused on enabling computers to understand, interpret and generate human language.

Terms:

Grounding – The process of linking natural language references, such as “red cube” or “left shelf”, to real or simulated physical objects in the environment.

Natural Language Programming – A programming approach where users describe tasks using human language instead of traditional code.

Precondition – A condition that must be true before a robot skill or action is allowed to start.

Postcondition – A condition that must be true after a robot skill or action has completed successfully.

Robot Skill – A reusable robot control behaviour with defined input parameters, preconditions, expected outputs and success or failure conditions.

Skill-Based Programming – A robot programming approach where reusable robot capabilities, such as pick, place, move or inspect are combined to create complex behaviours.

Introduction

Industrial automation and robotics are becoming increasingly important across many sectors, including manufacturing, logistics and service industries. Robots improve productivity, precision, and operational consistency. However, their adoption is still limited in many environments due to the complexity of robot programming. Traditional robot programming methods typically require specialized technical knowledge, including understanding of programming languages, coordinate systems, motion planning and safety constrains. As a result, industrial robots programming requires skilled engineers, which increase development costs and slows down the deployment of robotic solutions. This limitation is particularly significant for small and medium-sized companies that may lack expertise required to integrate and maintain robotic systems.

In recent years, advances in artificial intelligence, particularly in large language models and natural language processing, have created new opportunities to simplify human-robot interaction. Text-based AI models can understand complex instructions expressed in natural language and transform them into structured representations that can be interpreted by machines. This capability opens the possibility of developing intuitive robot programming systems where users can describe tasks in natural language rather than writing traditional code. Such an approach could significantly reduce the entry barrier for robot programming and allow non-technical users to interact with robotic systems more effectively. Therefore, the aim of this work is to research and develop a methodology for intuitive robot programming using natural language processing model to interpret user instructions.

The main goal of the thesis: research and develop a methodology for intuitive robot programming using natural language processing model to interpret user instructions.

Main tasks of the thesis:

1. Analyse existing robot programming methods, with the focus on natural-language-based approaches.
2. Develop a simulation-based environment and an end-to-end pipeline for natural-language robot programming.
3. Perform research to determine the key components required for this type of robot programming and to identify the main limitations of the approach.

1. Literature review

The development of an intuitive robot programming methodology based on large language models requires an understanding of both conventional robot programming approaches and recent advances in artificial intelligence and its application in robotics. Therefore, this literature review examines the main programming paradigms used in industrial and collaborative robotics, with particular attention to methods that reduce the technical barrier for end users. The review begins with traditional robot programming approaches, since they form the foundation of industrial robot control and highlight the limitations that motivate more intuitive alternatives.

Next, the review discusses the programming requirements specific to collaborative robots, where flexibility, usability, and human-robot interaction become especially important. This is followed by an overview of intuitive programming techniques, including teaching-based methods, visual programming, and skill-based task representation.

Finally, the review focuses on natural-language-based robot programming and the role of modern language models in transforming human instructions into executable task plans. Relevant grounding and planning approaches are examined to identify how current systems connect natural language, perception, and robot skills.

1.1. Traditional Robot Programming Methods

To develop a methodology for intuitive robot programming, it is first necessary to review the traditional and widely adopted programming approaches used in industry. Understanding how classical robot programs are created, transferred to the controller and executed in real time provides the baseline for later development of more intuitive approaches.

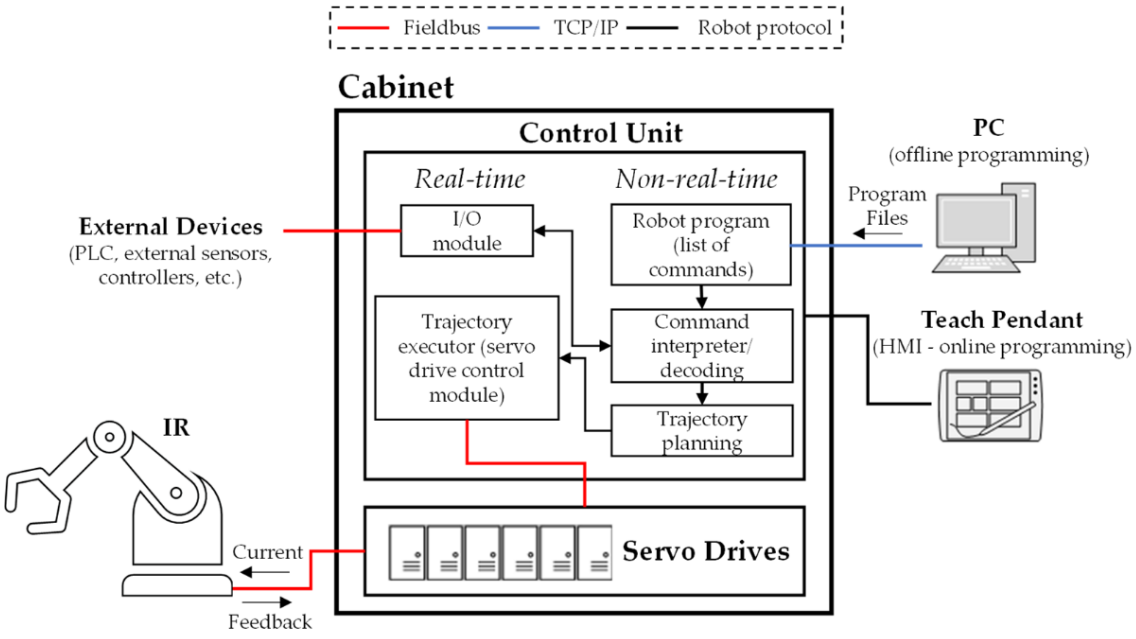


Fig. 1. Typical pipeline of traditional industrial robot programming.

A typical pipeline of traditional industrial robot programming is shown in Fig. 1. The robot task is defined as a sequence of vendor-specific instructions, created either offline on a PC or online using a

teach pendant, and then downloaded to the robot controller. Inside the cabinet, non-real-time modules read and interpret the instructions and perform path planning and trajectory generation, while the real-time layer executes the resulting trajectory and handles time-critical I/O. The controller cyclically sends joint position setpoints to the servo drives, which control the motors in closed loop using encoder feedback. External devices such as PLCs and sensors are integrated mainly via fieldbus connections to the controller I/O [1].

1.2. Direct Robot Programming

In direct (online) robot programming, the operator develops an application or modifies the robot directly in the work environment, using the physical robot itself [1]. When programming in this way, the operator uses a special training console – teach pendant to manually move the robot to the desired positions and record these points. Later robot replicates and moves along recorded path. This method is suitable for simple, repetitive tasks. Direct programming of the robot can also be done using kinesthetic training – the operator directly guides the robot's rear control unit through the desired trajectory by hand, and the robot controller records these movements. It is a more intuitive method that allows to program robots directly by guiding them on the required trajectory [2].

The main disadvantages of direct programming methods are that the programming is done with the robot itself, so the production line or work cell must be stopped during programming, which leads to downtime costs. Also, programming with these methods is a long-term process, and it is only suitable for simple and repetitive tasks, it is not suitable for frequent reconfiguration [2]. Direct programming methods also require specialized programming knowledge, as each robot manufacturer uses a different programming language, which creates challenges when integrating different robots into a single system [3].

1.3. Indirect robot programming

The indirect (offline) programming method allows the robot to be programmed separately from its workspace, often using a virtual robot model or simulation environment. The program is defined on a computer that is separated from a physical robot. This method creates ability to avoid production downtime during the programming phase, since the robot can perform other tasks while the program is being developed [1]. This type of programming also requires highly qualified professionals with advanced programming skills and deep knowledge of the robot's physical characteristics and limitations. When programming robots virtually, there are often inaccuracies in the virtual environment – the models cannot perfectly reflect real-world conditions, so programs created indirectly usually require thorough inspection and coordination with a real robot. For these reasons, indirect programming, like direct programming, is not suitable for small production batches or tasks that change frequently.

1.4. Programming Features Required for Collaborative Robots

The complexity of robot programming reduces their applicability and the possibility for companies to automate processes, as the costs of programming, installing and maintaining robots increase. This is especially noticeable in scenarios where robots work in tandem with humans – collaborative processes that use collaborative robots – cobots. Human participation in the cobot program goes beyond the traditional indirect role of the traditional programmer. The operator engages directly with

the cobot program during its execution, or in live mode. In this case, the operator must be directly or indirectly involved in the modification or impact of the cobot application.

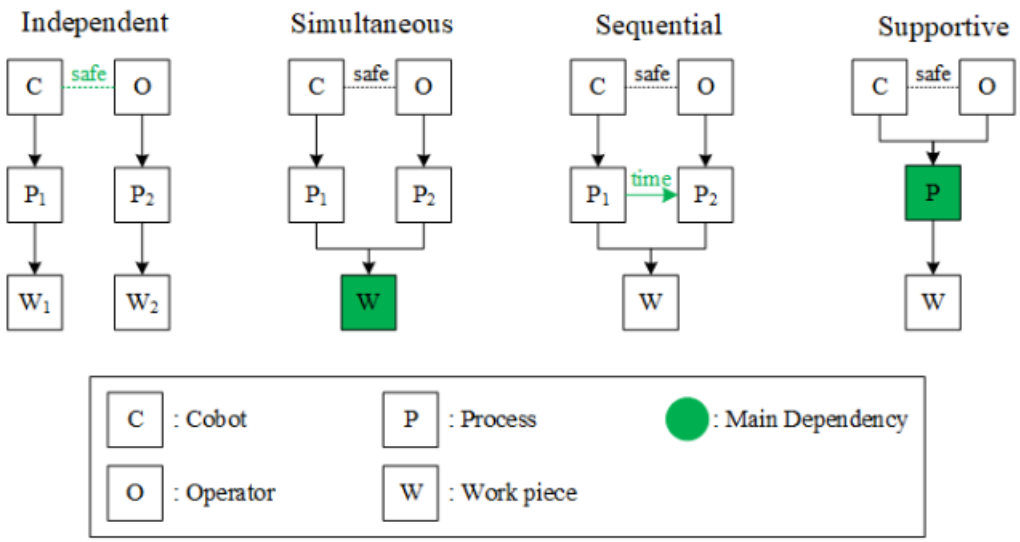


Fig. 2. Different Cobot and human collaboration principles

Fig. 2 highlights that the operator and the cobot may act independently, simultaneously, sequentially or in a supportive manner within the same process. Therefore, the operator can engage directly with the cobot program during execution, meaning that the operator is directly or indirectly involved in modifying or influencing the cobot's behaviour. Direct participation is manifested in clear communication between a human and a robot, i.e. a human sends information or instructions to the robot. Indirect participation occurs when a robot monitors human states and changes its course of action accordingly. Based on these modes of operator involvement, three main characteristics are created to ensure that the cobot is able to operate flexibly or to be programmed intuitively [4]:

1. **Communication:** The operator must be able to control the cobot through a communication channel, which can be verbal or non-verbal. Non-verbal communication includes gestures, gaze, head posture, and user interface [4].
2. **Optimization:** The program must be designed in such a way as to reduce the operator's workload rather than increase it or to keep the load unchanged [4].
3. **Learning:** Cobot must learn skills in a similar way to a human – by watching demonstrations, trial and error, getting feedback and asking questions. During operation, the operator must be able to directly influence the robot's skills by providing additional data, such as feedback, answers to questions, personalized demonstrations [4].

These requirements imply that cobot programming must move beyond conventional code - centric workflows toward interaction - centred approaches, where task intent can be specified, corrected and refined during operation. In practice, this has led to a range of intuitive programming techniques that aim to lower the expertise barrier and support rapid task configuration in collaborative settings.

1.5. Intuitive Robot Programming Techniques

Intuitive programming techniques can be seen as practical ways to satisfy the three cobot programming requirements (communication, optimization, learning).

For communication, common solutions include teaching pendants and kinesthetic teaching. In both cases, the operator can directly provide instructions to the robot, either by recording positions and actions or by physically guiding the robot during training [4].

For optimization, many companies use offline (indirect) programming supported by CAD and simulation tools. Environments such as ABB RobotStudio, Visual Components, and V-REP (CoppeliaSim) include user interfaces that help operators configure tasks, adjust parameters, and select operating modes while reducing production downtime during commissioning and changeover [4], [5].

For learning and accessibility, visual programming methods are often used. Icon-based or block-based systems (ABB AppStudio [6], Staubil VAL Blocks [7], Code3 [8], Omron TMflow [9]) allow users to build programs by combining graphical elements instead of writing code. These methods are usually suitable for simple tasks, but they can still make it difficult to define precise robot poses and parameters. For complex scenarios, especially those requiring conditional behaviour, additional tools or more advanced approaches are typically needed [4], [10].

Overall, intuitive programming techniques reduce the expertise barrier and simplify basic task setup, but they often become less effective when tasks require precise parametrisation, rich environment context, or complex conditional logic.

1.6. Skill and task based robotic management

The approach most widely discussed in the literature is task-based programming. It relies on a hierarchy of primitive skills and higher-level tasks. Skills usually represent reusable robot capabilities, such as motion primitives or sensor-based actions (e.g., opening a gripper or monitoring torque). Skills are goal-oriented and often object-centred, for example grasping an object or tightening a screw. Tasks are structured sequences of skills that aim to achieve a higher-level goal, such as an industrial operation or scenario [4].

Skills are the main foundation of task-based programming because they give a good balance between specificity and abstraction. They are general enough to be reused in different tasks but still structured enough to be understandable for human operators. A well-defined skill normally includes explicit preconditions and postconditions. These conditions are checked before and after execution to confirm that the action was done correctly and that the system state is consistent. In addition, skills are parameterised based on the current input state, and runtime monitoring is often used during execution to support safe and correct task progress.

In skill and task-based programming, a “skill” is usually treated as a reusable building block with a clear interface and a defined execution outcome. In practice, each skill is described by its purpose (what it should achieve), a set of parameters (what can be configured, such as target pose, speed, force limits), and execution semantics (what counts as success or failure). Several skill-based approaches emphasise that skills should include preconditions and postconditions: preconditions define when a skill is allowed to start, while postconditions define what must be true after execution to confirm correct progress and safety. Runtime monitoring is then used to detect violations and to trigger retries or recovery strategies [11], [12], [13].

Skills are also commonly organised hierarchically. Low-level skills represent motion or sensing primitives, while intermediate skills are compositions of low-level skills (e.g., pick-and-place as reach–grasp–lift–move–place). High-level skills are more related to perception, grounding, and task planning. This hierarchy supports modularity and reuse: the same low-level skills can be combined into different tasks, while the higher-level layers decide which skills to execute and how to set their parameters [11], [14].

A common way to execute skill sequences in a structured and robust manner is to use Behaviour Trees (BTs). BTs represent robot behaviour as a tree of modular nodes, where leaf nodes are actions (skills) or conditions, and internal nodes control the execution flow (for example, sequence and fallback). During runtime, nodes return a status such as SUCCESS, FAILURE, or RUNNING. This allows the system to react to unexpected events and handle errors without rewriting the whole program. Because of this modular and reactive structure, BTs are often presented as a practical alternative to large finite-state machines when task complexity increases [15].

In robotics, BTs are often used to combine reusability (skills reused as leaf nodes across tasks) with robustness (fallback nodes can implement recovery strategies, such as retrying a perception step or choosing an alternative action). For this reason, BTs fit naturally with task-based programming: tasks can be expressed as BTs that compose skills, while conditions and fallback branches provide a clear way to implement execution checks and recovery logic [16].

Overall, skill and task-based programming provide a modular way to represent robot capabilities and build complex behaviours through composition, while keeping clear interfaces, parameters, and execution outcomes. Behaviour Trees are one practical execution framework that can be used to organise such skill sequences.

1.7. Natural language-based programming methods

In natural language-based programming, free-form text instructions are converted into robot-understandable language. Text input is semantically processed and distinguishes the structure of the program, including controls, operations and related objects, from the grammatical dependency tree. This is done using language models. A language model is a neural network trained on large amounts of text to learn patterns in language and to predict the next words in a sequence. Because of this training, it can interpret instructions, generate structured outputs (such as actions or plans) and handle many ways of expressing the same intent [17].

The language model does not control the robot at the motor level. Instead, it mainly works as an interface for task specification and high-level planning (see Fig. 3). Because of this, the robot must already have a set of executable skills (e.g. pick, place, move, open or close gripper) and a reliable way to execute them. In other words, natural language is used to select and parameterize skills, while the actual execution is handled by the robot control system. Without a stable skill library and clear execution semantics, the generated plans become difficult to verify and unsafe to run in real environments. This is also reflected in natural language programming pipelines, where text is processed to extract actions, objects and control logic. The output is a structured program or plan that calls known robot skills [17].

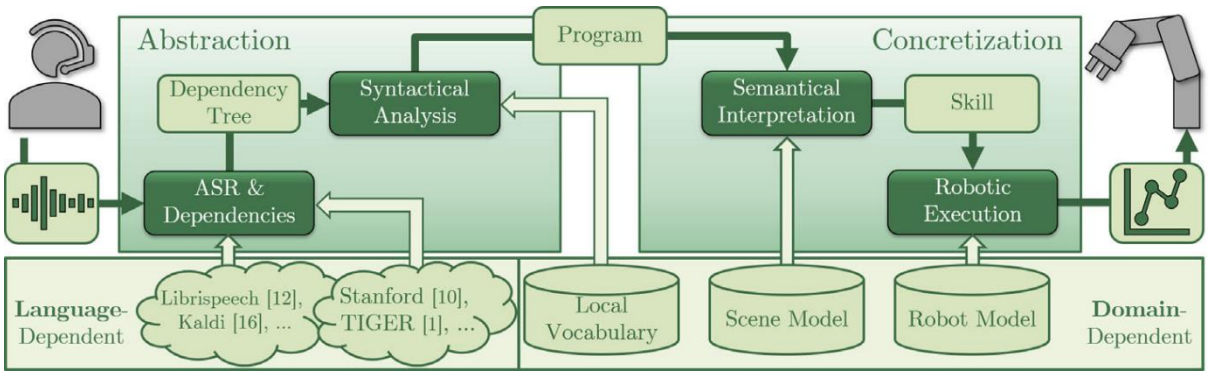


Fig. 3. Application of a natural language interpretation algorithm.

1.7.1. Grounding natural language

Natural language programming by itself is mainly useful for generating a structured program or action plan. However, to execute such a plan reliably, the system must ground the instruction in the current scene, meaning that object and part references in the text (e.g. “the red cup”, “the screw”, “the left shelf”) must be linked to real physical entities. In this context grounding can be understood as semantic interpretation supported by environmental context: the robot should not only parse the instruction but also understand what the referenced objects and events mean in the workspace at that moment. With grounding, the robot can avoid “blind” manipulation and instead select the correct object, choose an appropriate approach trajectory and decide which tool or strategy is suitable under the current conditions. In practice, grounding is typically supported by external sensor data, most commonly 2D or 3D cameras mounted on the robot or observing the workspace.

After natural language input is converted into a plan, the system still needs two key components to execute it:

1. Grounding, i.e., linking words to objects and states in the scene.
2. Skill execution layer, i.e. a set of available robot actions with defined outcomes.

The following subsections review representative methods that address these components in different ways: CLIP mainly supports grounding, while SayCan, PaLM-E, RT-2 and MALMM show how language models are combined with skills, perception and execution feedback to perform tasks from natural language.

1.7.2. CLIP model

One of the main visual-language models used for semantic grounding in robotics is CLIP. This model allows robots to efficiently match objects and understand visual concepts based on natural language (see Fig. 4). The CLIP is made up of a pair of neural networks that place pairs of text and image in a shared embedding space. The model is trained on 400 million pairs of image and text to predict which caption matches which image [18].

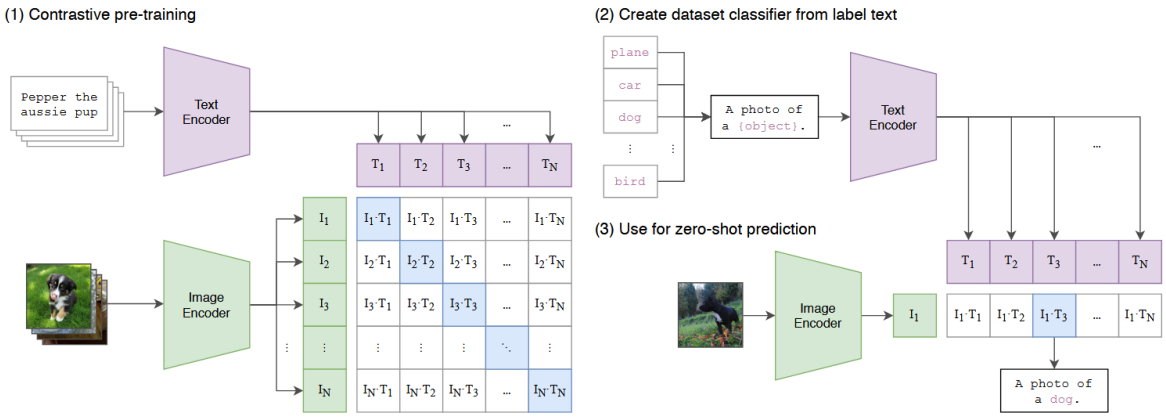


Fig. 4. Classic image models combine images with verbal equivalents. The CLIP model is trained by adding the text description generated for the photo

In the article [19] The authors study the task of rearranging objects, which uses a visible scene and an image of the final goal. Existing methods that rely only on visual properties, such as the properties of objects constructed by colour histograms or deep neural networks, generate poor results when the presented images are of the same class, but have different visual features, i.e. when testing is performed, in which the target image indicates a cup of one colour and a cup of another colour in the robot scene – the algorithm cannot distinguish that it is also an object with geometric properties matching the cup because its colour has changed. For this task, the CLIP model was applied by forming the CLIP-SemFEAT model (see Fig. 5), which, in addition to the visual properties, creates text descriptors for each object that best correspond to the specified objects. The CLIP-SemFeat method significantly outperforms methods (see Fig. 6) based only on visual descriptors, which allows to perform tasks of rearranging objects that correspond to the target image, although the objects in it are different from those in front of the robot.

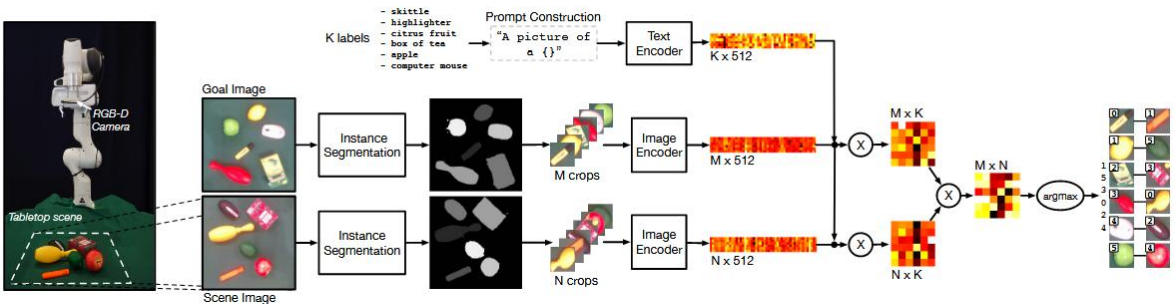


Fig. 5. Language-visual model training

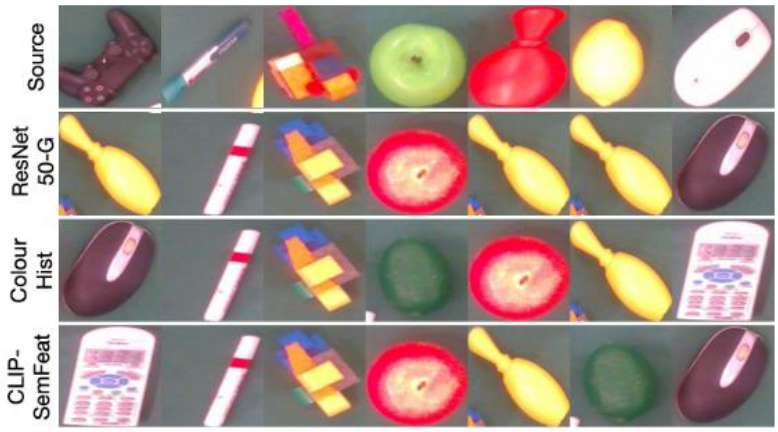


Fig. 6. Result of object rearrangement

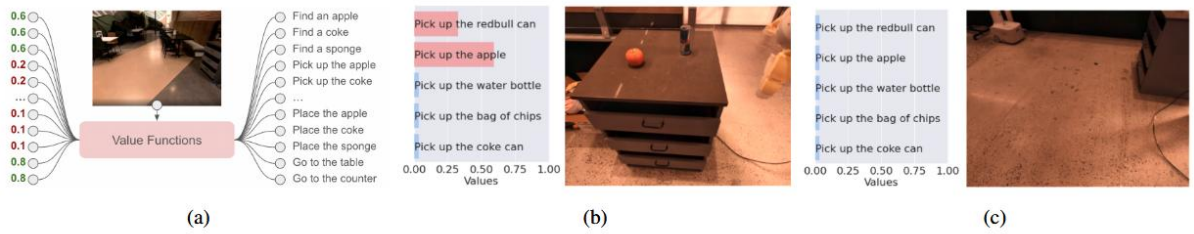
This is important for natural language programming because correct skill selection depends on resolving references like “the red cup” or “the mug on the left” and CLIP-style grounding helps reduce failures caused by visual variation.

1.7.3. SayCan Model

The SayCan model evaluates language instructions and known skills based on the robot's real-world capabilities, which are evaluated using a video camera. The algorithm combines large language models that provide high-level semantic knowledge with pre-trained skills that are linked to value functions. If a user says, "*I spilled my drink, could you help?"*, SayCan uses an LLM to suggest a sequence of actions (see Fig. 7).

Fig. 7. Example of how the SayCan model works using a natural language query to generate instructions that are appropriate for the bot.

Each action is evaluated by the value function, which determines its real-world feasibility based on the image seen by the camera (see Fig. 8).



(a) (b) (c)

Fig. 8. Query Formation (a) demonstrates how the model distinguishes known entity fetch actions based on the presented picture. In the Fig. (b) we can see that when the photo changes and two objects appear in it – an apple and a "redbull" can (c) demonstrates that with an empty frame no results are generated.

SayCan has been tested in 101 tasks related to the kitchen environment. Main results:

1. Planning success rate 84%
2. Execution success rate: 74%.

This shows that SayCan is able not only to plan the right sequence of actions according to the user's instructions, but also to successfully implement it in a real environment. SayCan innovatively combines the semantic understanding of language patterns with the physical capabilities of robots, allowing robots to efficiently perform complex tasks expressed in natural language [20].

SayCan shows a typical pattern for natural language robot programming: the LLM proposes a sequence of known skills, but execution is constrained by feasibility scoring based on the current observation, which reduces unsafe or impossible actions.

1.7.4. PaLM-E model

The PaLM-E model uses large language models, such as PaLM (up to 540B parameters), which is designed to understand and execute natural language instructions, including complex, multilingual queries, as well as demonstrating that it can produce intermediate reasoning steps in text before performing actions, which allows you to better examine what the model's answer to a question is based on. The language model is supplemented with a visual part of the modality, which allows the conversion of the visible image into textual information, as well as the ability to understand both 2D and 3D images and the ability to process multiple images, even if it has been trained with only single image requests. These modal parts are also supplemented by pre-trained robotic movements and skills. This allows the robot to perform tasks such as object manipulation, navigation, assembly operations [21].

From the methodology perspective, PaLM-E highlights that combining language with visual context improves instruction understanding, but task execution still depends on the available robot skills and how reliably they can be executed.

1.7.5. RT-2 model

Another example is the RT-2 video-language-action multimodal system (see Fig. 9). This model is unique in that it includes significantly more data about the robot's operation, its possible movement, and skills. The action coding is based on the RT-1 model, and the action space is covered by 6 degrees of freedom positions and rotational displacements in the limbs of the robot, the level of expansion of the gripper. The RT-2 models are based on existing image-language-action models such as PaLI-X and the aforementioned PaLM-E. They are combined with each other with both robotic trajectory

data and web-scale vision-language tasks. This multimodal system also surpasses others in that in previous models' robotic data accounted for less than 10% of the total data, but in the case of RT-2-PaLI-X, robotic data accounts for about 50% of the data, and in the case of RT-2-PaLM-e – about 66%. The system works with various objects, backgrounds, and can perform tasks that include picking, placing objects, pushing, opening drawers, and closing them. The RT-2 can interpret commands that were not present in the robot's training data, such as placing an object on a specific number or icon. Using "chain-of-thought," RT-2 can perform multi-layered semantic interpretation, such as deciding which object to take as an improvised hammer (stone), or which drink is best suited for a tired person (energy drink) [22].

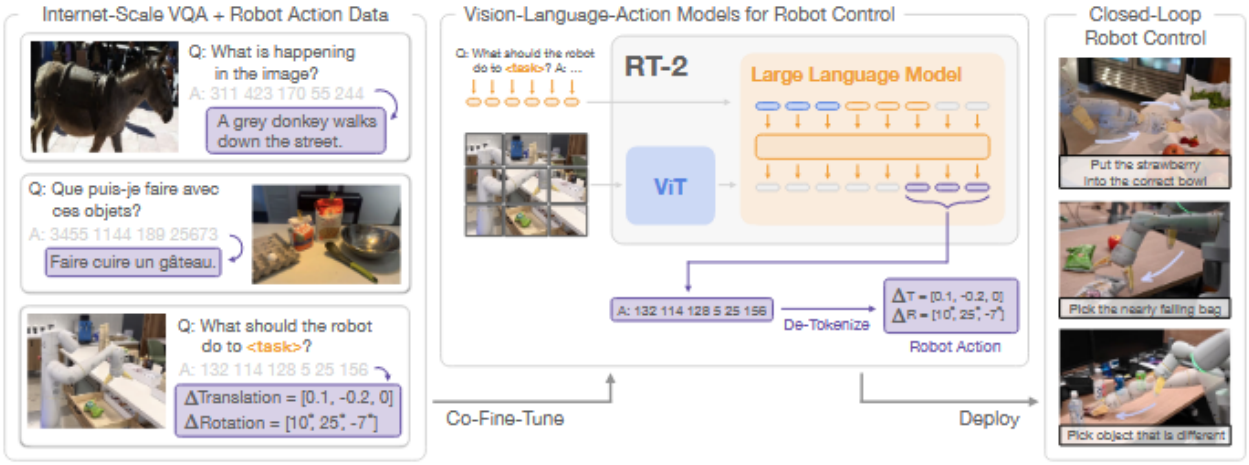


Fig. 9. Operation of the RT-2 model with closed-loop response to robotic actions.

The performance of the RT-2 models in performing tasks already seen is like that of the RT-1 but far surpasses other base models in various experiments. On average, the RT-2 model achieves about 2 times better results than previous models of this type and about 6 times better than other base models [22].

While RT-2 improves the ability to summarize semantic and visual concepts, the robot does not gain the ability to perform new movements because physical skills are still limited by the distribution of skills seen in the robot's data. For example, it has difficulty managing complex object dynamics, which are significantly different from those that the robot encountered during training. Also, a large amount of data is required to achieve the effectiveness of the model. Also, the system is still complex for practical tasks in real conditions[22].

RT-2 demonstrates strong generalisation from web-scale knowledge, but the practical limitations (data requirements, complexity and limited new physical behaviours) support the idea that for industrial use it is often safer to keep low-level control in engineered skills and use language models mainly for high-level planning.

1.7.6. MALMM model

In the article [23] the proposed MALMM (Multi-Agent Large Language Model for Manipulation) model addresses the development of skills using a three-level agent (see Fig. 10). Supervisor takes a human-generated request, processes it, and creates a request at the planner level, who arranges a sequence of actions required to complete the task, which is returned to the manager level. At the manager level, the information received from the planner is processed again and the task is passed to

the low-level programming agent to create the software code to execute the generated plan accordingly. The program created is returned to the manual level, from where it travels further to the program execution module. The execution module transmits commands to the robot and data about executed commands to the manager; the robot uses a video camera to generate visible information about the environment and transmits it to the manager. In this way, the language model constantly adjusts its actions to match the specified task, assesses the probability of completing the task, corrects itself if any action fails.

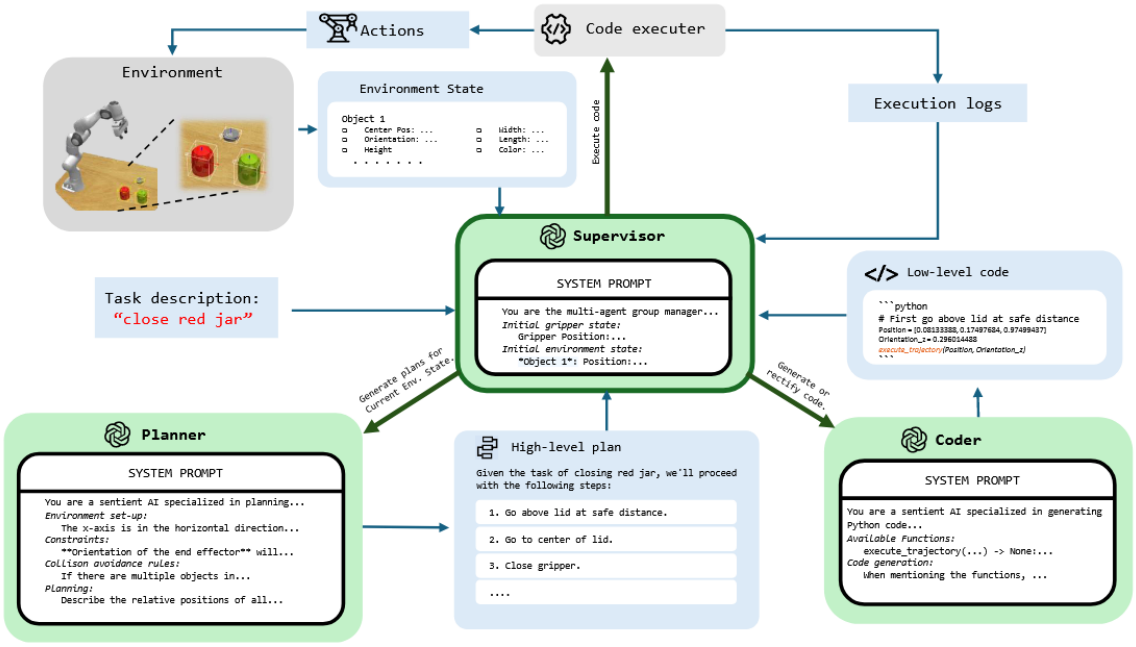


Fig. 10. The MALMM agent structure consists of a main management agent - manager, planner and programmer.

The MALMM model was tested in the CoppeliaSim virtual environment using gpt-4-turbo [24], LLaMA-3.3-70B [25] language models, additionally use "foundation" computer vision models, LangSam for generating object masks, and the M2T2 model for calculating precise poses of grasping objects, and the environment is monitored using 3D models. System tested with 9 task descriptions from RLBench [26] dataset. Comparison of different models can be seen in Fig. 11)

Methods	Basketball in Hoop	Close Jar	Empty Container	Insert in Peg	Meat off Grill	Open Bottle	Put Block	Rubbish in Bin	Stack Blocks	Avg
CAP [14]	0.00	0.00	0.00	0.08	0.00	0.00	0.76	0.00	0.00	0.09
VoxPoser [15]	0.20	0.00	0.00	0.00	0.00	0.00	0.36	0.64	0.32	0.17
Single Agent (SA) [11]	0.52	0.40	0.36	0.24	0.44	0.80	0.92	0.48	0.20	0.50
MALMM (†)	0.84	**0.88**	0.60	**0.80**	0.64	0.84	0.84	0.56	0.32	0.70
MALMM (‡)	**0.88**	0.84	**0.64**	0.68	**0.92**	**0.96**	**1.00**	**0.80**	**0.56**	**0.81**

Basketball in Hoop | *Close Jar* | *Empty Container* | *Insert onto Square Peg* | *Meat off Grill* | *Open Wine Bottle* | *Put Block* | *Put Rubbish in Bin* | *Stack Blocks*

Fig. 11. The results of the MALMM model were obtained by applying the model to nine RLBench tasks.

It is also interesting that the model shows successful results in tasks such as “Save the Cow”, in which toy bear, cow and wooden enclosure figures are placed in a simulation environment. The model is

given the task of rescuing the cow, and the model, after assessing the context, makes the decision to move the cow to a wooden enclosure (see Fig. 12).

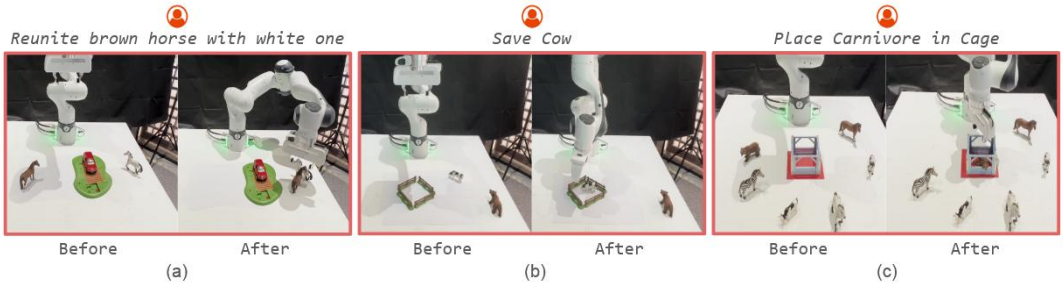


Fig. 12. Applying the MALMM model to an abstract "save the cow" task.

The MALMM model shows an advantage in all tasks, but it also has its own drawbacks. MALMM uses three gpt-4-turbo agents, which becomes extremely expensive, and MALMM is partially dependent on user-entered requests and pre-instructions, which has a significant impact on the final result, so even if the robot is able to develop its skills, associate objects with textual descriptions, smooth work can be hindered by an incorrectly submitted request or incorrectly formed system (instruction) prompt. Also, MALMM requires accurate object detection to correctly determine the points of grabbing objects.

MALMM formalises natural language programming as a pipeline (request, plan, code, execution) and uses execution feedback to correct failures. At the same time, its cost and dependence on prompt quality show why a controlled skill library and structured execution rule are important.

1.8. Overview of literature review

The literature review showed that intuitive robot programming cannot rely only on natural language, because free-form instructions become useful only when they are connected to a structured execution framework. From reviewed sources, it was learned that traditional direct and indirect robot programming methods remain effective but require significant technical expertise and are poorly suited for flexible human-oriented interaction. The literature review also showed that collaborative robot applications require programming approaches centred on communication, reduced operator burden and learning-oriented interaction. In addition, the analysis of intuitive programming techniques and skill-based approaches confirmed that reusable, parametrised skills with clear preconditions and postconditions provide and appropriate abstraction level between user intent and robot control, while Behaviour Trees offer a practical way to organise such skills into modular, reactive and verifiable task execution structures. Finally, the review of natural-language-based systems demonstrated that large language models are most suitable for high-level task interpretation and plan generation, whereas reliable execution still depends on grounding, constrained skill selection and monitored execution. This literature review provided insights that are used as the methodological basis for the proposed system in further chapters. Therefore, further research uses natural language only for task specification and structured plan generation, skill-based actions are used as the main execution units, Behaviour Trees are used for plan representation and execution control, and perception-grounding together with validation mechanisms are used to reduce ambiguity and improve execution reliability.

2. Methodological part

This methodological part explains the approach used in this research to develop and test a methodology for intuitive programming of robots using text-based AI model. The main idea of the work is that natural language can make robot programming easier for users, but it also creates ambiguity and safety risks if the instruction is executed directly. For this reason, in this research the text instruction is not converted straight into robot motion. Instead, it is transformed into a structured and limited plan format, which can be checked and then executed in controlled way.

2.1. Overview of the system structure

The proposed system is designed as a layered architecture (see Fig. 13). The simulation environment is used as the lowest layer, because it allows safe and repeatable experiments without risk for equipment or humans. In this work CoppeliaSim is selected as the simulation platform to model the robot cell, execute the physics simulation, and provide access to the complete scene (robot joints, object poses, sensors, etc.). The connection between CopeliaSim and the external control program is implemented in Python using the ZeroMQ Remote API. This integration allows the Python program to send commands to the simulator and read back the simulation state, which makes possible a closed-loop control workflow.

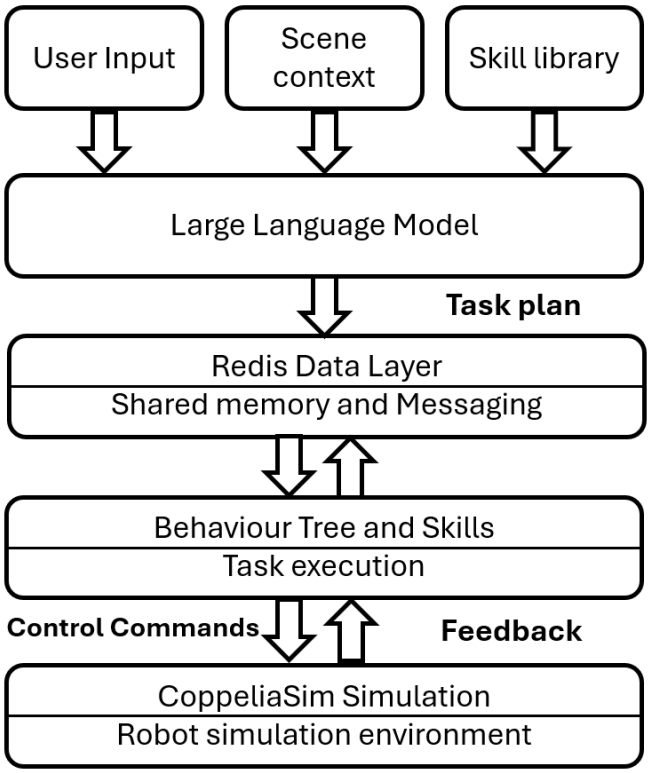


Fig. 13. Layered architecture of proposed system

On top of the simulation interface, the robot capabilities are implemented as Python “skills”. A skill in this research is defined as a reusable robot action with a clear interface: input parameters, required preconditions, expected postconditions, and possible failure outcomes. Skills are used as building blocks for more complex tasks.

For task execution and composition, this work uses Behaviour Trees (BT). Behaviour Trees are chosen because they provide a modular structure and a reactive execution model. They also support

clear separation between conditions and actions, and they allow explicit recovery logic when something fails. In the proposed methodology, the LLM is used only for planning: it generates a task plan in a constrained structure. The execution is done by a BT executor, which follows tree and calls skills as action nodes, while checking conditions and handling failures. With this design, the LLM output becomes easier to verify, and the system can avoid executing unsafe or incomplete instructions.

To connect the system modules, a Redis-based shared data layer is used. Redis works as a central storage for the system state, generated plans, and execution logs, and it also supports message-based communication between components. In this research Redis is used as a "blackboard", where different processes can read and write information: the planner reads the user request and environment context, the validator checks the generated plan, and the executor reports progress and outcomes. The LLM-based agent integrates in the planning layer by taking the user instruction together with the current context (available skills, robot and scene state, constraints) and producing a structured plan proposal. Before execution, the plan must pass validation rules. If the plan is not valid or some important parameters are missing, the system should ask clarification or replan, instead of starting execution.

2.2. CoppeliaSim environment

CoppeliaSim is an advanced robotics and automation simulation platform for creating, testing, and analysing robot behaviour in virtual environments. This environment is widely used for academic, scientific research and industrial projects. Development environment is based on a distributed control architecture: each object/model can be individually controlled via an embedded script, ROS / ROS2 nodes, remote API client, or a custom solution. This simulation environment has a wide range of physics engines that allow you to simulate object dynamics, collisions, friction and interactions between robots and the environment – providing an opportunity to reliably test real-world scenarios. CoppeliaSim allows you to control an unlimited number of robots and sensors in a single scene. Manipulators, mobile robots, drones, conveyors, industrial robots, LiDar, RGB-D cameras, force sensors, encoders and many other sensors are supported.

In this research, CoppeliaSim is used as the main simulation environment for development and experimental validation of the proposed intuitive robot programming methodology. The reason to use simulation is to enable safe and repeatable testing of task plans, motion logic and failure handling. The simulation environment (see Fig. 14) is inspired by the open-source project "LLM-BT-Robotics". The main purpose of adopting a similar scene structure is to use a proven layout for evaluating LLM-driven task planning combined with Behaviour Tree execution, while still adapting the scene to the specific requirements of this work. The scene is built as a pick-and-place work cell and includes an ABB IRB 4600-440 robot model. A vacuum gripper is attached to the robot end-effector to support grasping operations. Additionally, a camera sensor is included in the scene to enable perception-based task execution. The workspace contains a marked working area and several coloured objects used as task targets. The scene includes four coloured plates (yellow, red, green and blue) and three cubes (red, green and blue). These objects are used to create repeatable manipulation scenarios where robot needs to detect objects, decide the correct action sequence and execute motion and grasping. Known object poses from simulation environment are not used as ground truth for decision making. Instead, an overhead camera is placed above marked working zone and is used for plates and cubes localization. This approach is selected to simulate a closer to real-world setup, where object coordinates are not directly available and must be estimated from sensor data. Because localization

is performed from camera detection, the obtained object positions include measurement errors and small offsets. These perception errors are important for the research, since they allow testing whether the proposed planning and execution pipeline remains functional under realistic uncertainty.

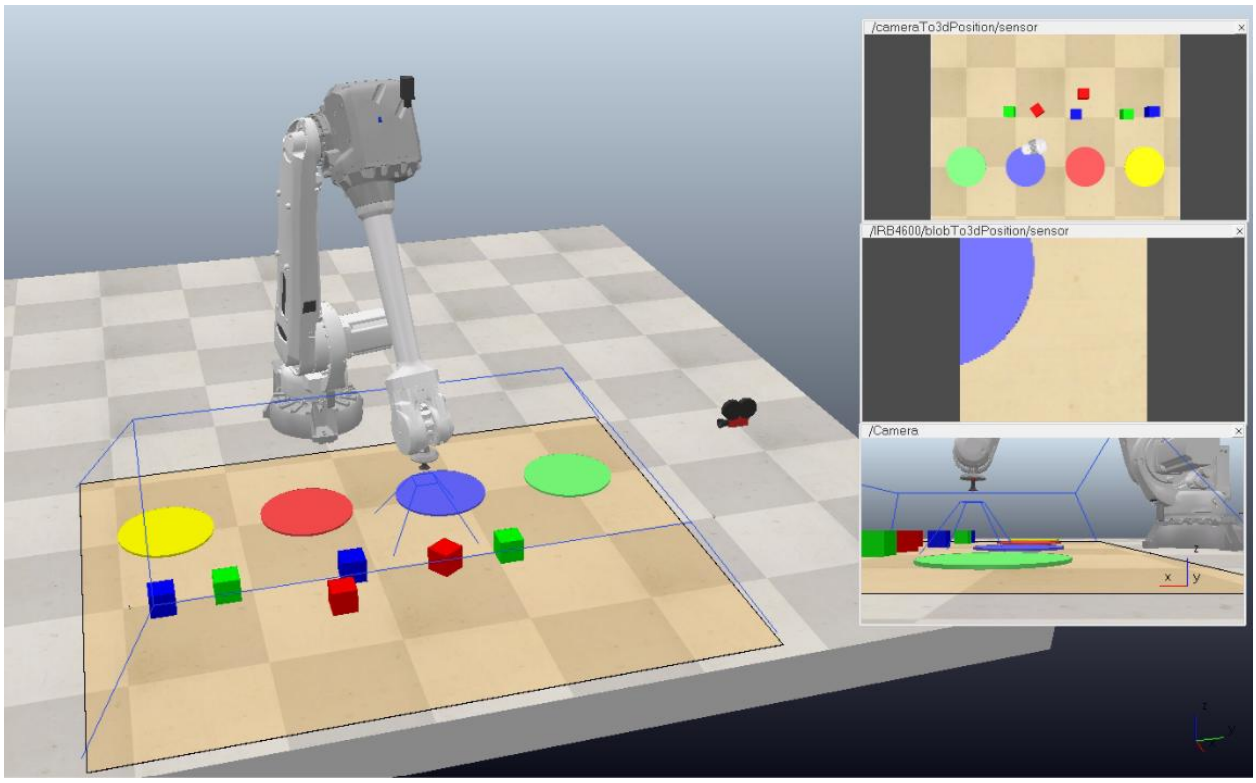


Fig. 14. CoppeliaSim environment overview.

For robot motion generation, the inverse kinematics functionality is enabled in the simulation. The project used as inspiration also applies inverse kinematics routines for robot movement control and the same principle is adopted here. This allows the high-level actions to be defined in task space, while joint configurations are computed through the IK solver inside the simulation environment.

Additionally, for further experiments an external Python script is developed to randomize the initial placement of objects in the scene. The randomization is implemented using a seed-based approach, meaning that for a selected seed value the same object placement configuration can be reproduced.

2.3. ZeroMQ Remote API communication and data flow

To control the CoppeliaSim scene from an external program ZeroMQ Remote API is used. Remote API enables communication between CoppeliaSim and an external application (running in a different process or even on a different machine) and supports the same API function calls as from within a CoppeliaSim script. CoppeliaSim supports two remote API frameworks, where the ZeroMQ remote API supports Python clients, therefore it is selected for this work [27].

The ZeroMQ Remote API follows a client-server principle. CoppeliaSim runs a ZeroMQ remote API server (enabled through the ZMQ plugin and the zmqRemoteApi add-on), While the Python program acts as a client and sends requests to the simulator. The official zmqRemoteApi protocol description specifies that communication is done through ZeroMQ REQ socket connected to an endpoint, and by default the server listens to TCP port 23000. The request is sent and a response is received for each call, which corresponds to a request-response RPC style interaction [28].

On the Python side, the connection is established using the official package coppeliasim-zmqremoteapi-client (see Fig. 15). This package implements RemoteAPIClient and allows the external program to "require" the sim namespace, which is then used to call CoppeliaSim API functions (e.g., reading object handles, getting positions, setting targets, starting/stopping simulation) [29]. Examples can be found in Fig. 16 and Fig. 17.

```
from coppeliasim_zmqremoteapi_client import RemoteAPIClient

client = RemoteAPIClient()          # connect to default endpoint (port 23000)
sim = client.require('sim')         # access CoppeliaSim API
simIK = client.require('simIK')     # optional: IK plugin

client.setStepping(True)            # step-by-step mode
sim.startSimulation()
```

Fig. 15. Example: Connect to CoppeliaSim and access sim.

```
handle = sim.getObject('/IRB4600')              # absolute path to object
pose = sim.getObjectPose(handle, -1)        # pose in world coordinates

position = sim.getObjectPosition(handle, -1)
```

Fig. 16. Example: Get object handle and pose.

```
sim.setInt32Signal('activated', 1)          # e.g., turn on suction pad
active = sim.getInt32Signal('activated')
```

Fig. 17. Example: Write/read simulator signals.

In terms of data transfer, two categories of information are exchanged between Python and CoppeliaSim:

1. Commands and control signals. These include starting or stopping the simulation, setting joint targets or end-effector targets, triggering gripper actions, and setting object properties. The external program can also call custom script functions inside CoppeliaSim if needed (for example, to group many operations into one call and reduce communication overhead), which is recommended in practice when many variables must be updated.
2. Observations and state feedback. These include robot state (joint positions, end-effector pose), object state (positions/orientations), and sensor data (for example camera-based detection outputs produced externally). This feedback is used by the Python control logic to decide which skill to execute next and to detect success or failure conditions.

Additionally, ZeroMQ remote API has a stepping and continuous simulation running mode. By default, CoppeliaSim runs freely and the client only sends requests – in continuous mode. In research stepping mode is used in which time advances when the client explicitly calls sim.step(). This is useful for deterministic experiments, synchronized logging, and closed-loop control, because each control cycle maps one simulation step.

2.4. Skills and Behaviour trees

An important concept in intuitive robot programming is the use of Behaviour Trees and Skills. Behaviour Trees are used to organise these skills into task execution structures, where actions and

conditions can be combined in a modular and reactive way. Skills represent reusable robot actions with clearly defined inputs, execution logic, and expected outcomes. Skills provide a structured way to describe robot capabilities at a higher level of abstraction than low-level motion commands. Together skills and Behaviour Trees form a practical foundation for building flexible, interpretable, and reliable robot task execution system.

2.4.1. Behavior trees

In this research, task execution is organised using Behaviour Trees (BTs). A BT is a hierarchical control structure in which task is represented as a tree of nodes with a single root. During execution, the root node is repeatedly ticked, and each tick propagates through the tree according to its structure. In this way, the BT determines which node should be executed next and evaluates whether the necessary conditions for continuing the task are satisfied.

Each node in a Behaviour Tree returns one of three main statuses:

1. SUCCESS – the node has completed its function successfully.
2. FAILURE – node cannot complete its function.
3. RUNNING – node has started execution but requires more time to finish.

This execution logic makes BTs suitable for robotic applications, where actions often take time and where task progress must be continuously monitored.

A Behaviour Tree is typically composed of three main groups of nodes. The first group consists of control flow nodes, which define the execution logic and ordering of child nodes. The most important control flow nodes are Sequence and Selector (or Fallback). A Sequence node executes its children in order and succeeds only if all of them succeed – if any child fails, the whole sequence fails. A Selector node also evaluates its children in order, but it succeeds as soon as one child succeeds and fails only if all children fail. The second group consists of decorator nodes, which modify the behaviour of a child node, for example by adding retry, timeout or inversion logic. The third group consists of leaf nodes, which perform the actual task – related operations. In this work leaf nodes are implemented either as action nodes, representing robot skills, or as a condition node, representing checks of the current system state.

Behaviour Trees were selected in this work because they match well with the requirements of an LLM-based robot programming pipeline. First, they support modular task composition, meaning that complex tasks can be constructed from smaller reusable units. The same skill or condition can therefore be applied in multiple tasks without rewriting the overall control logic. Second, BTs provide a clear separation between task structure and low-level execution. The tree itself describes what actions should be performed and in what order, while the detailed execution logic remains inside the skill implementations. The separation is especially important when task plans are generated automatically by a language model, because it constrains execution to a predefined and controlled set of allowed actions.

In addition, Behaviour Trees naturally support recovery and fault-handling strategies. For example, if a pick action fails, the tree can be designed to return to an earlier step, such as rescanning the workspace, selecting the target again, and retrying the action. Such recovery logic can be expressed explicitly and remains easy to inspect and modify. Another important advantage of BTs is their

transparency during execution. Since the execution progresses through clearly defined nodes and statuses, it becomes straightforward to log which plan was executed, which nodes succeeded or failed, and at which step the execution stopped. This property is particularly important in the context of this thesis, because it supports systemic experimentation, result analysis and failure investigation.

2.4.2. Skills structure and definition.

In this research, Skills refer to a bounded robot control behaviour implemented as a leaf action node in a Behaviour Tree. A skill represents a reusable robot capability designed to perform a specific function within the task execution pipeline. In contrast to low-level motion commands, a skill operates at a higher level of abstraction and provides a structured interface between task planning and robot execution. This makes skills suitable for a modular task composition, since complex robot tasks can be represented as sequences of smaller and well-defined actions.

Each skill in the proposed system is defined by several essential elements:

1. It has input parameters that specify the details required for execution, such as target colour, target zone or target plane.
2. It includes required preconditions, which define the state that must be satisfied before the skill can be executed. For example, the system may require that the scene map would already exist or that a target object has already been identified.
3. Each skill produces expected outputs, which typically correspond to updates in the blackboard, internal state variables or the world model. These outputs allow skills to use the information generated during earlier execution steps.
4. Each skill has explicit success and failure conditions, which make its execution outcome clear and allow BT to react appropriately.

In the developed system, all skills are implemented in Python as Behaviour Tree leaf nodes. During execution, they interact with the simulation environment through the ZeroMQ Remote API, which enables the external control program to send commands to simulation environment and receive state feedback. In addition, the skills use perception results obtained from the overhead camera pipeline, allowing execution decisions to depend not only on predefined commands but also on the current observed state of the workspace. In this way, skills form the main execution layer of the proposed methodology, connecting high-level task planning with controlled robot actions in the simulation environment.

The skills implemented in this research form the core execution layer of the proposed methodology. Each skill represents a specific robot capability and is designed as a modular Behaviour Tree action node with clearly defined responsibilities, state updates and execution outcomes. Table 1 summarizes the main skills used in the system, together with their goals, principal outputs and success, running, failure logic. This shows how the proposed skill library supports both simple manipulation actions and more structured execution, such as searching, picking, placing and stack-related operations.

Table 1. Skills and their definitions.

Skill	Goal	Main outputs and status logic
ScanSearchArea	Scan the workspace with the top camera and build an up-to-date world	Main outputs: Writes scene_map, plane_pos, color_pos_dict. Resets transient targeting keys: current_color_pos, stack_target_color, stack_pick_required. SUCCESS: at least

	model containing plane and cube poses	some objects/planes are detected. RUNNING: nothing is detected (continues scanning). FAILED: No objects were detected after set time (timeout).
StackingInit	Initialize stacking order and bookkeeping. This node does not move the robot – it prepares state variables such as colour order and stack index	Main outputs: Writes color_order, in_sequence, stack_index=0, stack_target_loc. Clears transient flags: current_color_pos, picked_up, current_color_source, stack_pick_required, stack_target_color. SUCCESS: Valid stack order is provided. RUNNING: Not used (instant). FAILED: No valid stack order was produced.
StackingAdvance	Advance stacking order pointer to the next colour object that needs to be picked up	Main outputs: Updates stack_index = stack_index + 1 when multi-colour sequence is active. SUCCESS: Pointer updated. RUNNING: Not used (instant). FAILED: Missing colour order.
TargetedSearch	Select the next cube to pick from scene map, optionally constrained by a named zone (left, right, top, bottom, centre) or from the stack	Main outputs: Writes current_color_pos = (colour, [x,y,z]), current_color_source = "surface". SUCCESS: Object exists in targeted area, and its colour and location was obtained from scene map. RUNNING: Not used (instant). FAILED: Object does not exist in targeted area based on scene map.
PickUpCube	Physically pick the cube based on targeted position using IK motion, proximity sensor vacuum enable signal.	Main outputs: sets picked_up=True, appends to pickup_history, removes cube from scene_map (remove_nearest_cube), updates color_pos_dict from updated scene_map. Uses stack_target_color/stack_pick_required to re-select the top-most stacked cube of that colour. SUCCESS: Cube was picked up using vacuum gripper and proximity sensor senses the object. RUNNING: Vacuum gripper is still not at set height and position; proximity sensor does not sense any object. FAILED: Cube was not picked up at set height and position.
PlaceCube	Place the currently held cube at target location and update world model	Main outputs: Sets picked_up = False, calls Processor.place_cube, updates scene_map by adding cube entry at placement location (add_cube) and updates color_pos_dict, logs into stack_history (writes to history.json). SUCCESS: placement completes for supported target_loc (e.g., "red plane", "green plane", "blue plane", "random", "sort_planes"). RUNNING: cube is still attached to vacuum gripper. FAILED: precondition fails (picked_up is False) or invalid/unsupported target_loc; in "sort_planes" (truncated in source).

As shown in Table 1, the implemented skills cover the main stages required for task execution in the proposed system: environment observation, target selection, object manipulation. This design enables complex tasks to be represented as combinations of smaller reusable behaviours, while preserving clarity between planning, state management and physical execution.

2.5. Redis as a communication and shared data layer

Redis is an in-memory data structure storage that is commonly used as a database, cache and message broker. It stores data in form of key-value pairs and provides read and write operations. Because Redis keeps its main data in memory, it is suitable for applications that require low-latency communication and rapid state updates. In addition, Redis supports more advanced data structures and communication mechanisms, such as lists, sets, hashes, streams and publish-subscribe channels.

In the publish-subscribe communication model, one process publishes a message to a named channel and other processes subscribed to that channel receive the message. This allows different system components to exchange information without being directly coupled to each other. For robotic systems, such an approach is useful when separate modules are responsible for planning, execution,

monitoring or state management, because each module can operate independently while still exchanging data through a common communication layer.

In this research, Redis is used as a shared data and communication layer between the main components of the proposed robot programming system. It provides a practical way to transfer task plans, store execution-related information and maintain coordination between modules. The use of Redis supports a modular system design, where planner, validator, executor and state management components can work as separate processes while still sharing the information required for task execution.

2.6. Plan representation and execution

In the proposed pipeline, task plans are represented in a serialized JSON format (see Fig. 18) and transmitted through Redis. This design provides a clarity between a planning layer and an execution layer, since the generated plan can be transferred, stored, validated and replayed independently of the execution process.

```
{
  "children": [
    {"behaviour": "ScanSearchArea"},
    {"behaviour": "TargetedSearch", "target_color": "green", "zone": "center"},
    {"behaviour": "PickUpCube"},
    {"behaviour": "PlaceCube", "target_loc": "red plane"}
  ]
}
```

Fig. 18. JSON scheme structure of “pick green cube and place ion red plane” task.

The JSON plan follows a simple list of children structure, where each element corresponds to a skill name. In this way, the plan does not directly define a low-level robot movement but instead specifies which predefined skills must be executed and in what order. Fig. 18 presents an example of such a plan for the command “pick green cube and place it on red plane”.

After JSON message is received, it is processed by a parser module that converts the serialized representation into Behaviour Tree (BT) nodes. Each plan is mapped to the corresponding BT leaf node with its required parameters, and the resulting children are attached to the root node of the tree. Once the tree is constructed, it is executed by repeatedly ticking the root node until overall execution reaches either SUCCESS or FAILURE. Therefore, the JSON plan defines the task structure, while the BT executor is responsible for its controlled execution.

2.7. Shared state – blackboard and scene map

The implemented skills are not independent, because many of them rely on information produced by previous execution steps. For example, object manipulation skills require target positions that are first obtained by perception-related skills. For this reason, the system uses two complementary state management mechanisms: a runtime blackboard and a persistent scene map.

The first mechanism is the blackboard, which acts as a shared key-value storage accessible to all behaviours during a single execution. Each behaviour explicitly declares which keys it reads and which keys it writes. This approach improves transparency of dependencies between skills and reduces hidden coupling between different parts of the execution pipeline. In the developed system,

the blackboard stores the most important temporary execution data, including the current scene map, detected plane positions by colour, the currently selected cube target, a flag indicating whether an object is attached to the gripper and additional stacking-related variables such as colour order, stack index and stack mode flags.

The second mechanism is a small persistent world model stored in the file in JSON format. This file contains a structured representation of the current scene, including plane positions by colour, a list of cubes with their identifiers, colours and positions as well as source information indicating whether an object is located on the surface or in a stack. When relevant, stacking metadata such as stack plane and stack index are also included. In contrast to the blackboard, which is intended for temporary execution data within a single BT run, the persistent scene map provides scene-level information that can be preserved across separate commands.

The persistent map is used for two main purposes.

1. It supports debugging and traceability, because the current world state can be inspected independently of the live execution process.
2. It allows the system to preserve stable scene knowledge between separate task requests.

2.8. LLM-based task planning in the proposed methodology

In the proposed methodology, the large language model (LLM) is not used as a low-level robot controller. It does not generate direct motion commands, does not call simulator API functions, and does not directly control inverse kinematics, actuator signals or other time-critical robot processes. Instead, the LLM is used only at the planning level, where its main role is to interpret a natural-language user instruction and convert it into a structured task plan that can be executed in controlled way using robot control software (see Fig. 19).

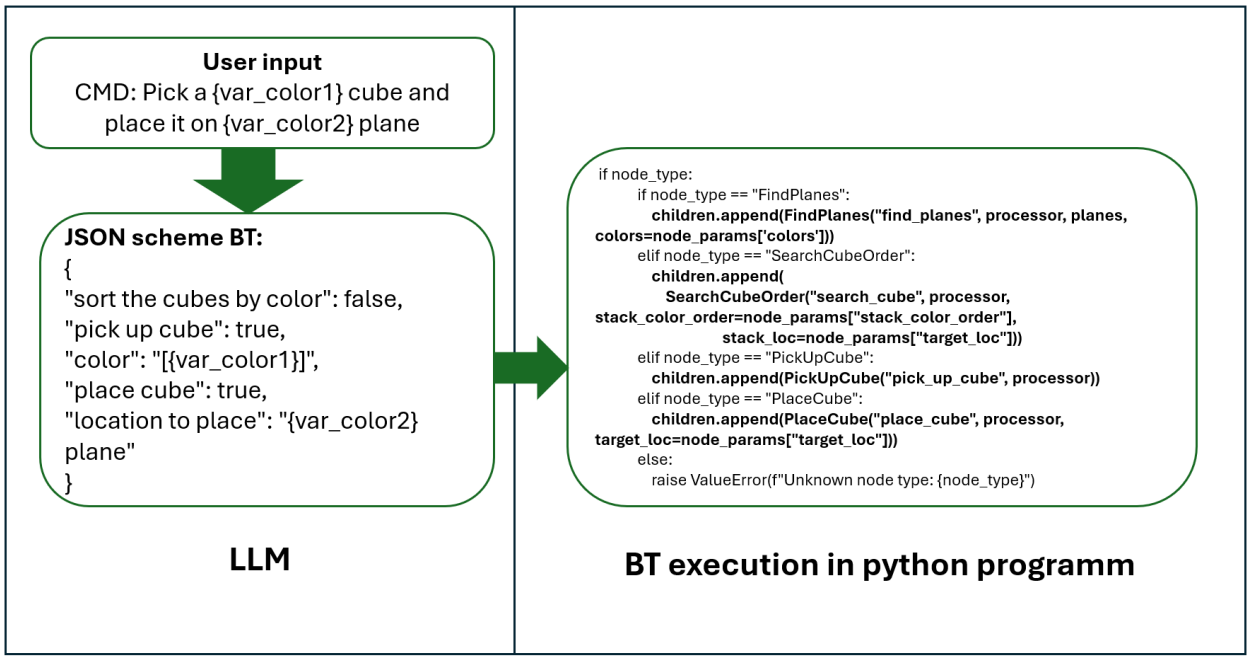


Fig. 19. Role of LLM in the proposed planning and execution pipeline

The LLM therefore operates as a high-level planner between the human user and the robot execution system. Its function is not to perform the task directly, but to determine which predefined robot behaviours should be used and in what order. This separation is important because it establishes a clear boundary between probabilistic language interpretation and deterministic robot execution. As a result, the flexibility of natural-language interaction can be used without giving the LLM direct control over robot motion or physical actions.

The input provided to the LLM consists of more than the user command alone. In the developed system, the planner receives three main types of contexts:

1. It receives the natural-language command describing the task to be performed, for example “pick a green cube and place it on the red plane”.
2. It receives the skill registry stored in YAML file, which defines the available skills, their purposes and the allowed parameter and their values. This registry acts as a contract between the planning module and the execution module, since the LLM must generate plans using only behaviours supported by the system.
3. The planner receives recent execution history stored in JSON type file, which provides some memory about previous actions and supports multi-step task reasoning, such as tasks involving stacking and subsequent unstacking. This history is only a contextual memory, not a complete or full up to date world representation.

To ensure generated output remains usable and safe, the LLM is guided by a dedicated system prompt stored in agent prompt file. The prompt is designed to make the model behave as a constrained planner rather than as a conversational assistant. It explicitly instructs the model to return only a JSON object, to follow the required plan structure, to use skills listed in skills registry and to select only parameter values that belong to allowed skills. If the command cannot be interpreted safely or lacks sufficient information, the model is instructed to return a warning object instead of an executable plan. In this way, prompt design serves as one of the main mechanisms for reducing ambiguity and constraining the output space during planning.

The resulting plans is represented in a serialized JSON format corresponding to a Behaviour Tree structure. After generation, the plan is passed to the validation and execution pipeline. The JSON output is first checked against structural and behavioural constraints and only then it is converted into BT nodes and executed by the BT executor. This design forms an important safety and correctness boundary: the LLM can only choose from fixed set of allowed robot skills, while actual motion generation, grasping logic, perception-based updates and state transitions remain fully controlled by the Python-based execution layer. Such an architecture improves interpretability, supports validation and makes the generated plans easier to analyse and reproduce during experiments.

2.9. Planner implementation and responsibilities

In the implemented system, the planner is realised as a Python program (see Fig. 20) that coordinates the interaction between the user command, the LLM and the execution pipeline. Its purpose is to collect the required planning context, construct the LLM request, receive the generated plan and forward it to the execution layer. In this way, the planner acts as the central software component responsible for transforming a natural-language task description into a structured plan suitable for BT execution.

```
agent_prompt = (Path(...)/"prompts/agent_prompt.txt").read_text()
tools_text = (Path(...)/"tools/registry.yaml").read_text()
history_text = (Path(...)/"history.json").read_text()

user_message = {
  "role": "user",
  "content": f"TOOLS_REGISTRY:\n{tools_text}\nHISTORY: {history_text}\nCMD:
{user_input}"
}

response = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[{"role":"system","content":agent_prompt}, user_message],
  temperature=0,
  max_tokens=512,
).choices[0].message.content

redis_client.publish("json_commands", response)
```

Fig. 20. Planner implementation in program.

The main responsibilities of the planner are as follows:

1. It loads the system prompt.
2. It loads skills registry.
3. It loads recent manipulation history.

After collecting these inputs, the planner combines them with the user command and sends the resulting request to the LLM. Once the model returns structured JSON plan, the planner receives this output and passes it further to the execution pipeline through Redis-based communication. This way planner remains separate from the BT executor and simulation environment.

2.10. LLM call and plan publication to Redis

The planner uses an OpenAI chat model, configured in the implementation as gpt-4o-mini, to generate structured task plans from natural-language commands. The model is called from Python module and uses chat completion interface. During each planning cycle, the planner first constructs the complete request consisting of the system prompt, the user command, skills registry and history. This combined input is then sent to the model to obtain constrained JSON plan.

After the response is received, the resulting plan in JSON form is forwarded to the Redis channel, where it can be accessed by the execution program. In this way, the planning module and the execution module are connected through a message-based communication layer rather than through direct internal function calls.

Redis is used here as a decoupling mechanism between planning and execution. The planner publishes the generated plan, while the executor independently listens to the corresponding channel and starts processing the plan when a new message arrives. This approach supports modularity, because the planner and executor can operate on separate processes. It also improves traceability in experimental conditions, since the exact generated JSON plan can be stored together with the random seed, execution logs and evaluation results. As a result, the system supports both reproducible experiments and clearer analysis of the relationship between planning output and execution outcome.

3. Experimental evaluation of the proposed methodology

This chapter presents the experimental evaluation of the proposed methodology for intuitive robot programming using text-based AI model. The purpose of this evaluation is to determine whether the developed system can reliably transform natural-language task descriptions into executable robot behaviour under controlled simulation conditions. In particular, the experiments are intended to assess the main functional stages of the pipeline, including task interpretation by the large language model, structured plan generation, BT construction and execution of predefined robot skills in the simulation environment.

The experimental analysis focuses on several key aspects of the proposed methodology.

1. It examines whether the LLM can generate valid BT plans that follow required JSON structure and use only predefined robot skills with appropriate parameters.
2. It evaluates whether these generated plans can be executed successfully in the simulation environment to achieve intended task outcome.
3. It considers how the system behaves under more difficult conditions, such as increased task complexity, ambiguous task descriptions.

In this way, the experimental part of the thesis is used not only to demonstrate successful operations of the developed system, but also to identify its limitations and the main sources of failures.

The experimentation is organised in two stages. The first stage consists of an initial pipeline test using simple manipulation task. Its purpose is to verify that all main system components work together correctly, from natural-language input to final task execution in simulation. The second stage consists of a broader experimental evaluation using multiple task classes with different levels of complexity. This structure makes it possible to first confirm the correctness of the end-to-end pipeline and then to analyse performance in a more systematic and comparative way.

3.1. Initial validation of the end-to-end pipeline

As an initial experiment, a simple manipulation task was selected to verify the complete operation of the developed pipeline. The chosen command was “Pick up green cube and place it on red plane”. This task was intentionally selected because it is simple enough to analyse clearly but still requires all essential stages of the proposed methodology: natural language interpretation, constrained task planning, conversion of the generated plan into a BT, execution in simulation environment. Therefore, this initial test serves as a baseline demonstration that the full pipeline operates as an integrated system.

In the first step of the experiment, the task description expressed in natural language was submitted to the planner through the LLM interface (see Fig. 21). Together with the user command, the planner also provided the language model with system prompt, skill registry, recent execution history. The purpose of this step is to evaluate whether the LLM could correctly interpret the instruction.

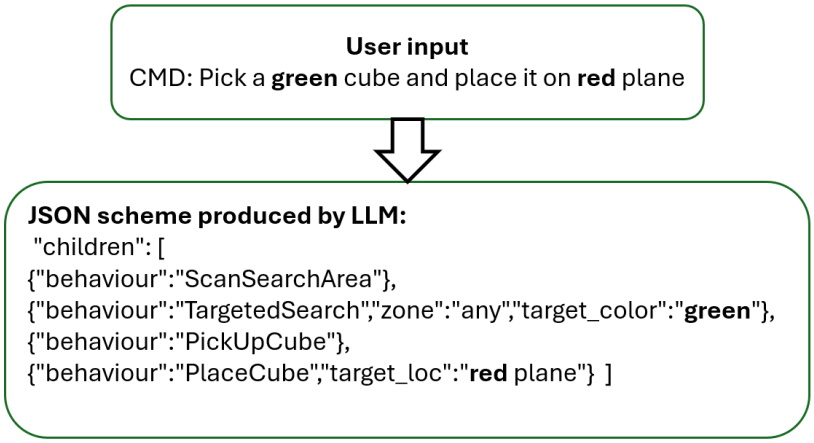


Fig. 21. Generated JSON plan for the initial task.

After receiving the input, the language model generated a structured JSON representation of the task plan. This JSON output described the required sequence of predefined robot skills and therefore served as a serialized BT specification. The generated plan was then passed to the parser and execution modules, where it was transformed into a concrete BT composed of the corresponding nodes (see Fig. 22). Each node activates second node if it successfully reached success state. Each node (skills) success, failure, running states are described in Table 1. If state – failed is reached, further execution is aborted.

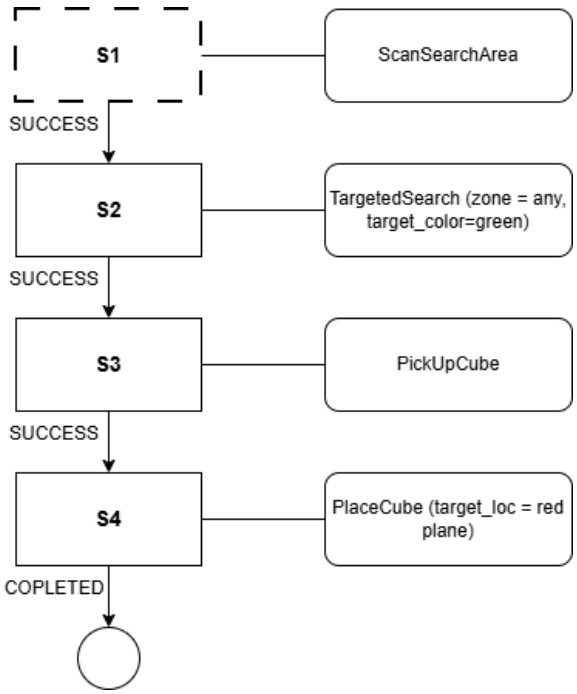


Fig. 22. BT execution steps.

Once the BT was constructed, it was executed in the CoppeliaSim environment (see Fig. 23). During execution, the BT executor sequentially triggered the required skills, such as workspace scanning, target selection, object pickup and object placement. At each stage of the execution, the system monitored the status of the corresponding BT nodes and checked whether the expected conditions for task progression were satisfied. This allowed the experiment to verify not only the outcome of the task, but also the internal correctness of the execution flow.

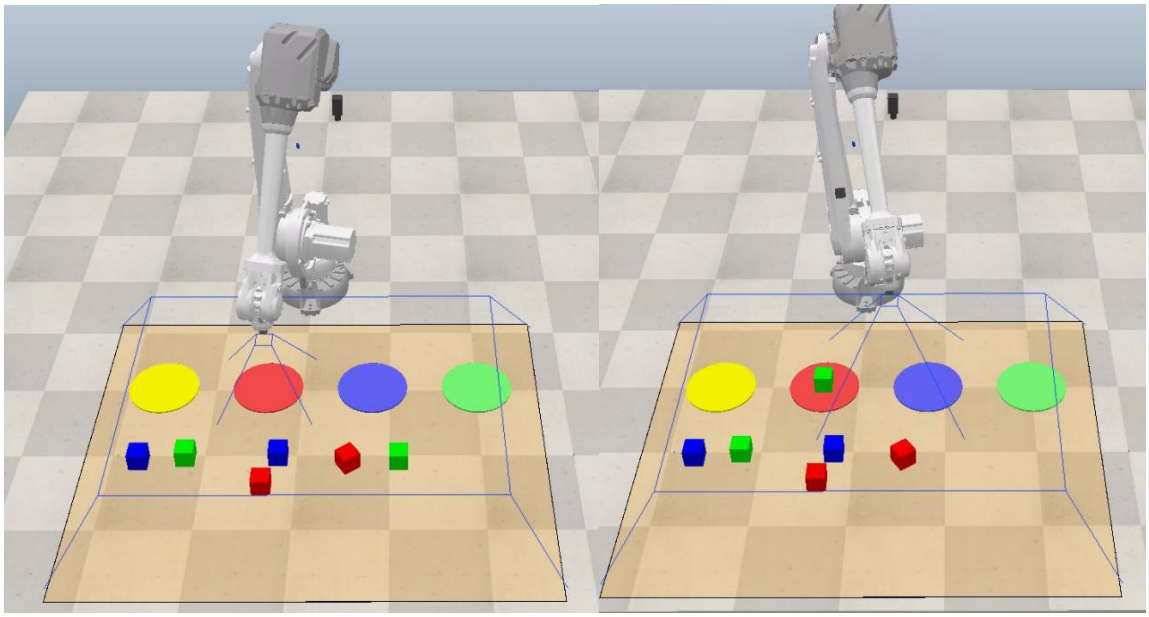


Fig. 23. execution of the initial test task in the simulation environment.

The simulation environment provided a visual representation of the full task progression, from the initial scene state to the final placement of the green cube on the red plane. This visual confirmation makes it possible to compare the generated plan with the observed physical outcome of the simulated manipulation task. As a result, the initial experiment demonstrates that the proposed methodology can link natural-language task input with structured planning and controlled execution in the robot simulation environment.

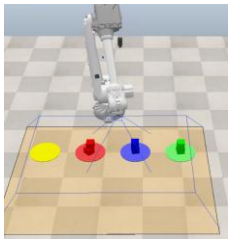
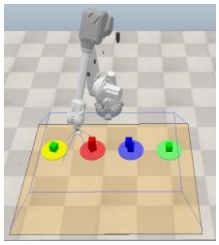
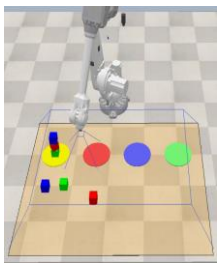
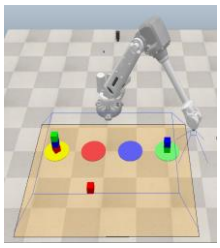
For this initial test scenario, the language model required 2.7 s to generate the execution plan. The total token usage during planning was 1790 tokens, of which 1731 tokens were used as input and 59 tokens were produced as output. These values indicate that for a simple single-object manipulation task, the planning overhead was relatively low compared to the total execution process.

The successful completion of this initial experiment shows that the developed pipeline is functionally capable of performing the complete sequence from instruction interpretation to robot task execution. More specifically, it confirms that the system can: interpret a natural-language command, generate a structured plan in the required format, construct a BT from that plan and execute the corresponding skills in the simulation environment until intendent goal is achieved. Therefore, this first experiment serves as an essential validation step before proceeding to the broader evaluation with larger and more complex task sets.

3.2. System testing using different task classes

To further evaluate the proposed methodology more systematically, a broader experimental study was carried out using multiple task classes with different levels of complexity. In total, 88 task descriptions were prepared and divided into four classes, with 22 tasks in each class. The purpose of this experiment was to examine how the developed system performs when required manipulation logic becomes more complex and when the natural-language instructions require more constrained reasoning about ordering, storage and multi-step operations. Table 2 presents task classes used in the experiments.

Table 2. Categories of tasks that were used in experiments.

Task class	Task description	Output example
C1 - Colour sorting	Tasks where the main goal is to sort cubes to matching colour planes.	
C2 - Sorting with post-sorting stack manipulation	Tasks where cubes are first sorted by colour, and after that one or more cubes are removed from stacks, relocated, or returned	
C3 - Mixed-stack construction	Tasks where the robot must build one or more mixed-colour stacks in specified order, sometimes leaving them unchanged	
C4 - Storage (yellow plane) manipulation	Tasks where the storage area is used as an intermediate buffer or temporary working area.	

The four task classes were designed to reflect different manipulation scenarios. Class C1 contains colour-sorting tasks, where cubes must be moved to the corresponding-coloured planes. A typical example is Task 01, which says: "*Sort all cubes by colour: place two red cubes on red plane…., two green…, and two blue…*". The phrasing is direct, declarative, and destination oriented. Most C1 tasks use a single main phrase such as sort, place or complete, followed by explicit constraints about counts, zones or order. Class C2 extends this scenario by adding post-sorting stack manipulation steps. This changes both the sentence structure and the phrasing. For example, Task 08 says: "*Sort cubes by colour. After sorting, move one green cube from green plane to random surface, then return it back to green plane*". It becomes a two-stage procedure usually marked by connectors such as *after sorting, then, return, unstack or move one top cube.* Compared with C1, these tasks use more state-depended language, because the second phase refers to an object that exists only after the first phase has produced a stack. Tasks in this class therefore require the planner not only to understand final destinations, but also to preserve task history and intermediate state. Within the class, the variation is mainly in how heavy the post-processing becomes. Task 08 has one leave-and-return loop, while tasks such as 64 or 67 add storage detours, random-surface relocation or repeated pickup of the same cube. Class C3 consists of mixed-stack construction tasks, where the system must build stacks in specified colour orders. It differs more from C1 and C2 because its central objective is not full sorting but ordered stack composition. An example is Task 39: "*Build one mixed stack on storage plane in order green, red, blue. Use one cube per colour only.*". The key phrases: *build one mixed stack, in order,*

from bottom to top, and *use one cube per colour only.* These expressions encode structure, sequence and selection limits in a compact way. The differences inside C3 can be seen by comparing Task 39 with Task 40. Task 39 describes one ordered stack, while Task 40 says: "*Build two mixed stacks: one on storage plane in order red, blue, green and one on green plane in order green, blue.*". The grammar is similar, but the second task increases complexity by introducing two simultaneous ordered subgoals instead of one. The Class C4 represents storage-mediated manipulation tasks, in which the yellow plane is used as an intermediate storage area (buffer) or temporary workspace. These tasks are built around the idea that cubes are first moved to storage, then moved again to their final destinations, sometimes followed by an additional cleanup or sorting phase. A clear example is Task 44: "*Move one red cube from centre zone and one blue cube from right zone to storage area in that order. Then move blue to blue plane and red to red plane. Finally, sort the remaining cubes by colour.*". Compared with other classes, C4 uses the largest number of temporal connectors: *then, after that, finally, in reverse order, immediately move* and similar phrases. Its sentence structure is therefore the most sequential and layered. The goal is also different from C3: in C3 the stack itself is often the final product, while in C4 storage is only intermediate state used to organize a longer manipulation sequence. In short, the classes progress from direct final-state specification in C1, to final state with additional post-processing in C2, to explicit ordered structure building in C3 and finally to multi-stage transport and redistribution workflows in C4. This made it possible to evaluate the system under gradually increasing planning and execution complexity.

The overall task completion results are shown in Fig. 24. According to the results, the highest number of successfully completed tasks was obtained in classes C1 and C3, where 18 tasks were completed successfully in each class. Class C2 achieved 15 successful executions, while class C4 showed the lowest result with 11 successfully completed tasks.

When these results are considered relative to the total number of tasks in each class, the success rates are 81.8% for C1, 68.2 % for C2, 81.8% for C3 and 50% for C4, so total average is 70.45% of successfully executed tasks (62 out of 88 tasks). The lowest performance in class C4 confirms that tasks involving storage as an intermediate manipulation area create the greatest difficulty for the proposed system. These tasks require more complex sequencing, stronger assumptions about object location and more careful handling of intermediate state transitions. By contrast, the high success rate of class C3 suggests that once a valid plan is generated, mixed-stack constructions can often be executed reliably, although this class still produced several planning failures before execution.

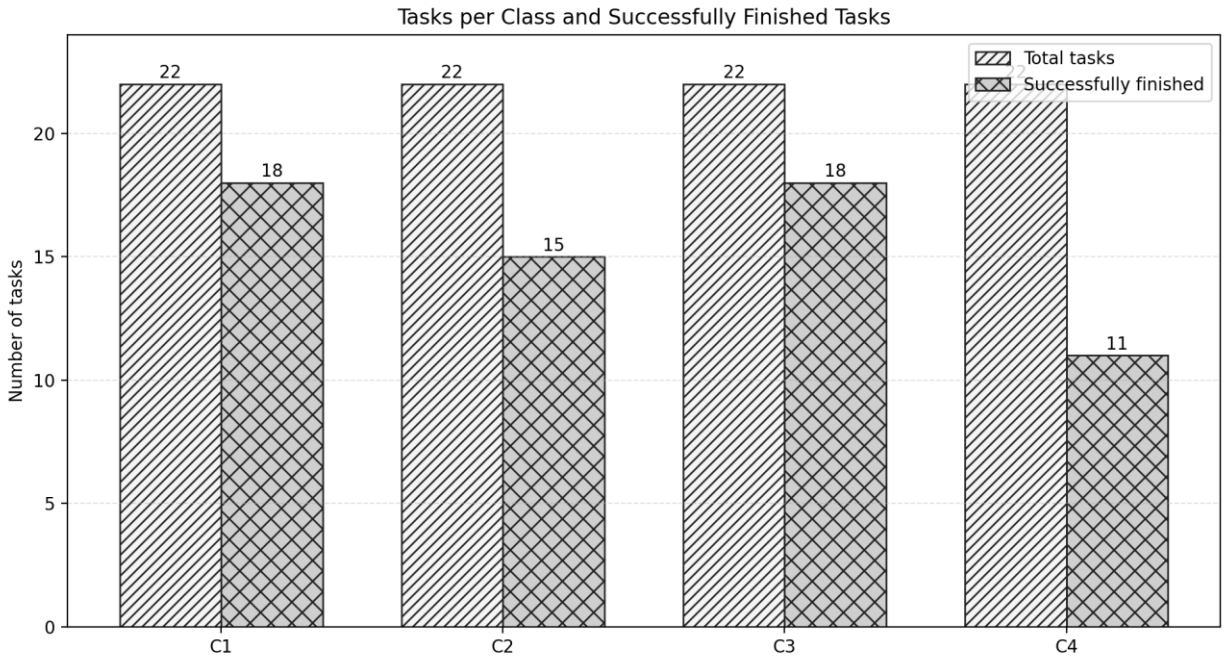


Fig. 24. Total tasks per class and successfully finished tasks.

The average number of BT steps per executed task is presented in Fig. 25. The largest average was observed in class C4 with 26.4 BT steps on average. A similarly high value was obtained in class C2 with 25.7 steps, while class C1 required 19.5 steps on average. The smallest value was recorded in class C3, where the average number of executed BT steps was 12.2. This difference indicates that successful mixed-stack tasks in C3 often resulted in shorter executable plans than the more storage-dependent scenarios in C2 and C4.

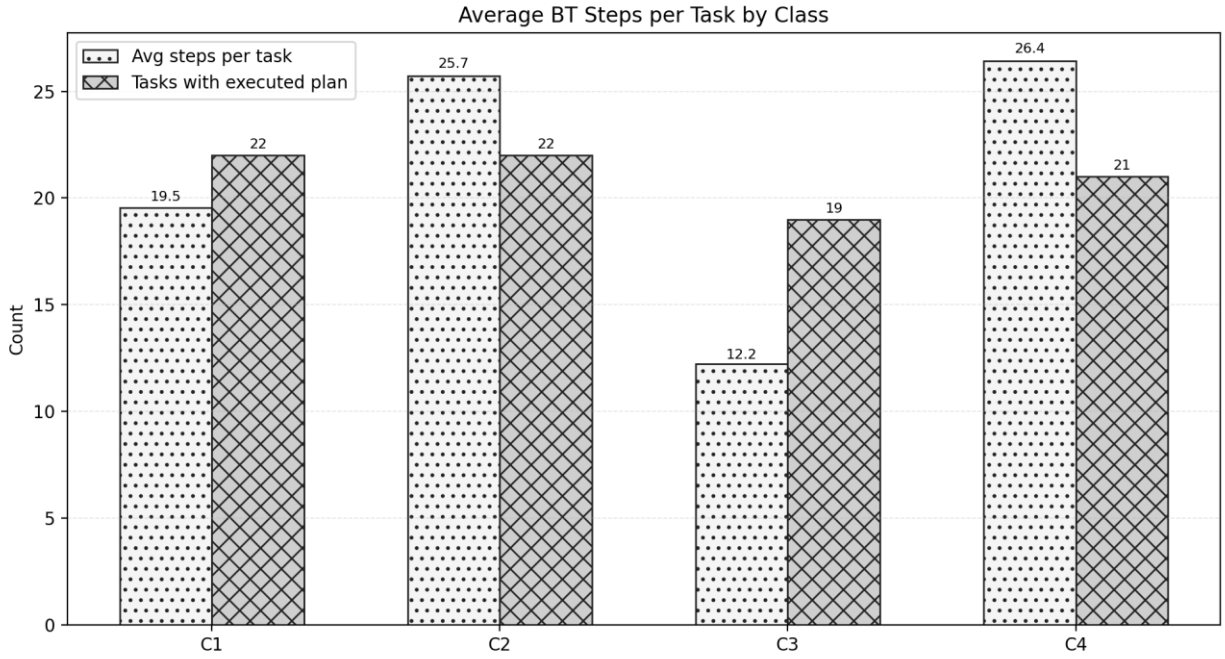


Fig. 25. Average Behaviour Tree steps used per task.

The planning cost in terms of token usage is shown in Fig. 26. Among all classes, the highest average total token usage was observed in class C4, with 2711.41 tokens per task. Class C3 followed with

2571.68 tokens, while C2 and C1 required 2269.32 and 2187.32 tokens respectively. The average prompt token (displaying the length of the task and additional information passed to LLM) count was highest in class C3 at 2662 and in class C4 at 2311.18, whereas the average completion-token (length of generated plan) count was highest in class C4 at 400.23 and in class C2 at 356.50. These results indicate that more complex tasks not only require larger context prompts, but may also lead to longer generated plans, especially in scenarios involving storage and post-sorting manipulation.

Fig. 26. Average token usage per class.

The detailed token distributions per task also reveals several outliers. While class C1 (see Fig. 27) and C2 (see Fig. 28) stayed consistent, in class C3 (see Fig. 29), tasks T72 and T78 required 7440 and 8048 tokens respectively, while in class C4 (see Fig. 30) task T24 required 7164 tokens and task T30 required 5027 tokens. These unusually large values are associated with multiple LLM attempts and indicate that certain tasks were difficult for the planner to resolve under the imposed constraints. Therefore, although average token values provide a useful overall comparison, the per-task distributions show that planning cost increases sharply when the model struggles to produce a valid executable structure.

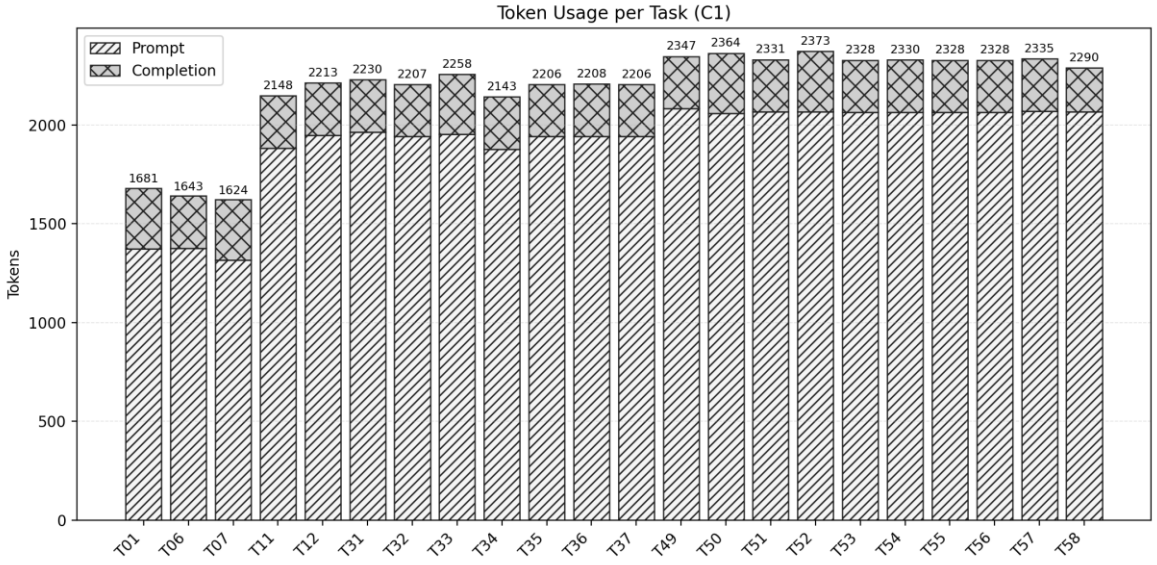


Fig. 27. Token usage per task, class C1.

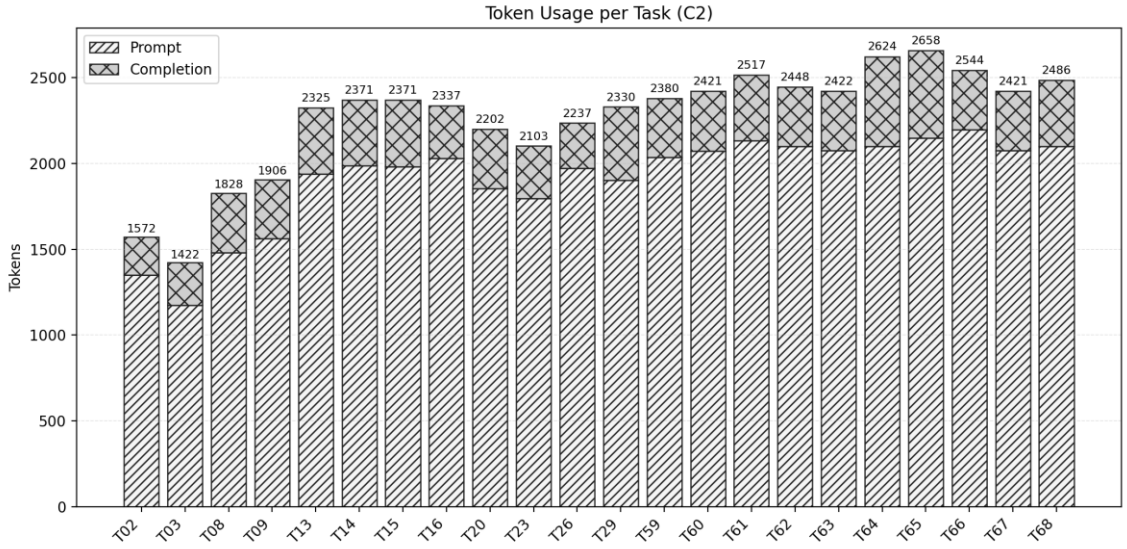


Fig. 28. Token usage per task, class C2

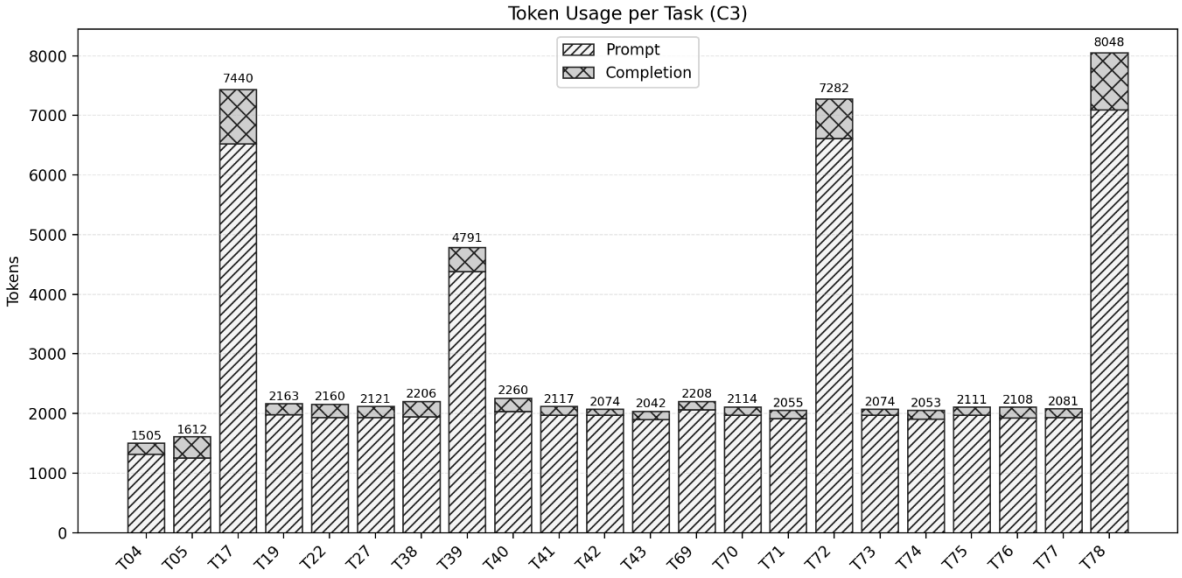


Fig. 29. Token usage per task, class C3

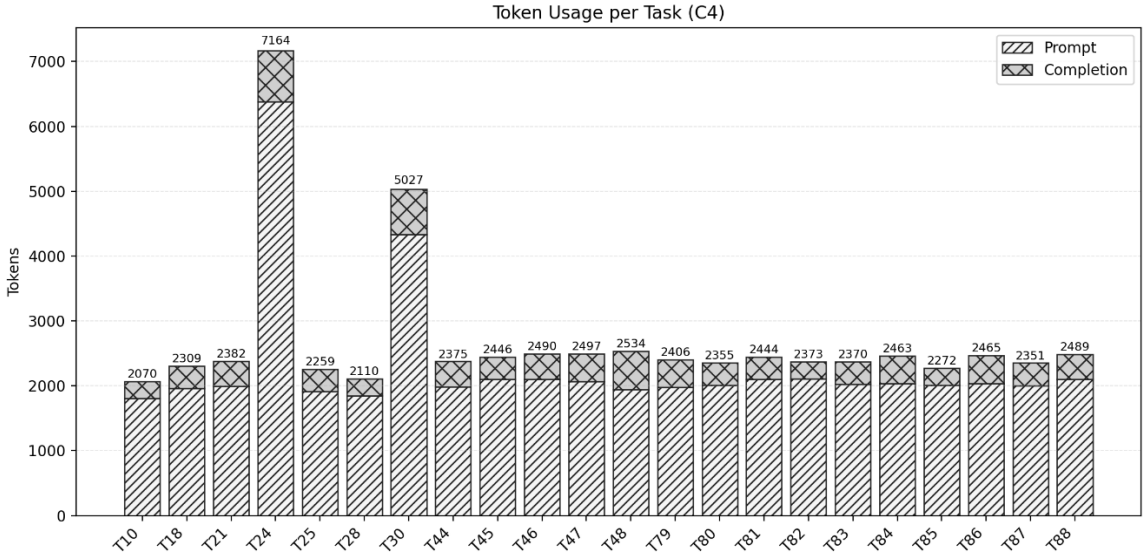


Fig. 30. Token usage per task, class C4.

Looking at plan generation time per task – longest planning time was recorder in class C3 (see Fig. 33), where task T17 required 21.5 s, followed by class C4 (see Fig. 34) where task T24 required 17.0 s and task T30 required 16.9s. In class C1 (see Fig. 31), the slowest task required 15.9s and in class C2 (see Fig. 32) slowest task required 14.2 s. In general, most tasks were generated within approximately 4 to 10 seconds, but the difficult cases show that plan generation time rises considerably when additional retries are needed. This trend is consistent with the token statistics and suggest that planning difficulty is reflected in both: prompt size and in plan generation time.

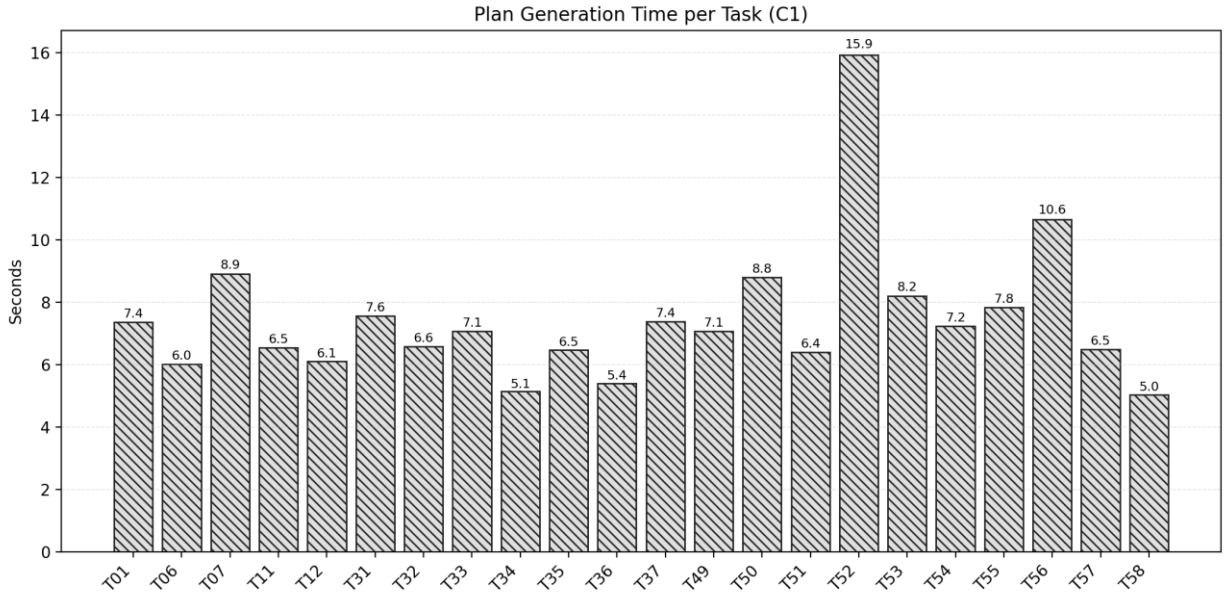


Fig. 31. Plan generation time per task, class C1.

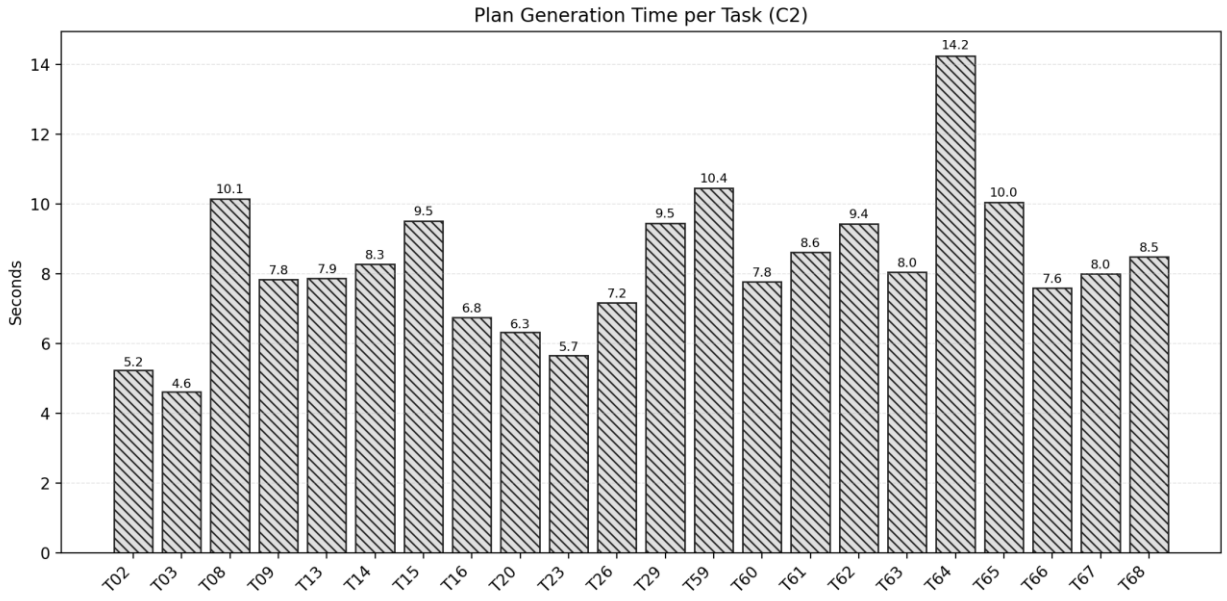


Fig. 32. Plan generation time per task, class C2.

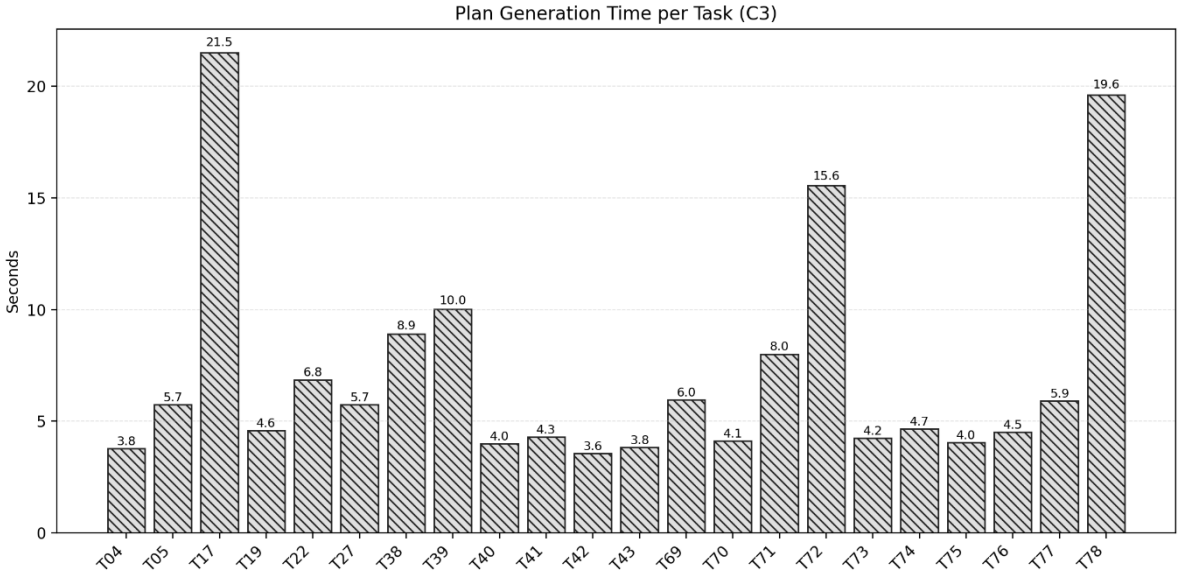


Fig. 33. Plan generation time per task, class C3.

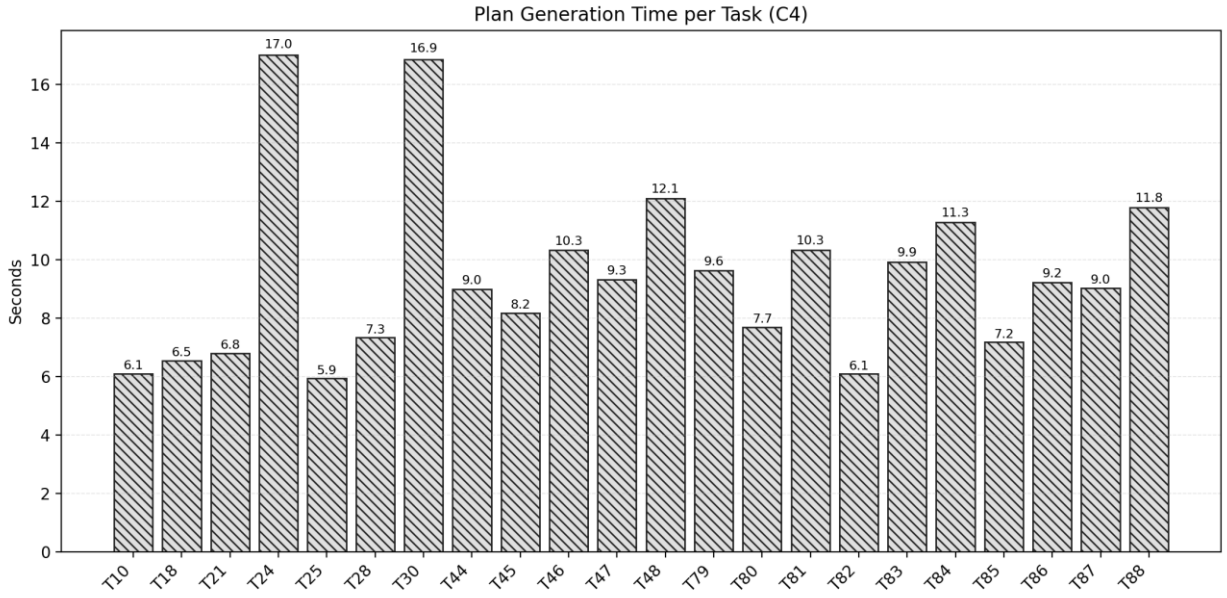


Fig. 34. Plan generation time per task C4.

Average attempts count (see Fig. 35) further supports evidence of this effect. Classes C1 and C2 had an average attempt count of 1 and no tasks requiring retry. In contrast, class C3 had an average attempt count of 1.18 and three tasks requiring retries, while class C4 had an average attempt count of 1.14 and two retry cases. These results show that the simpler sorting tasks were generally handled in a single attempt, whereas stack-construction and storage-mediated tasks were more likely to require repeated planning attempts before a usable result was obtained.

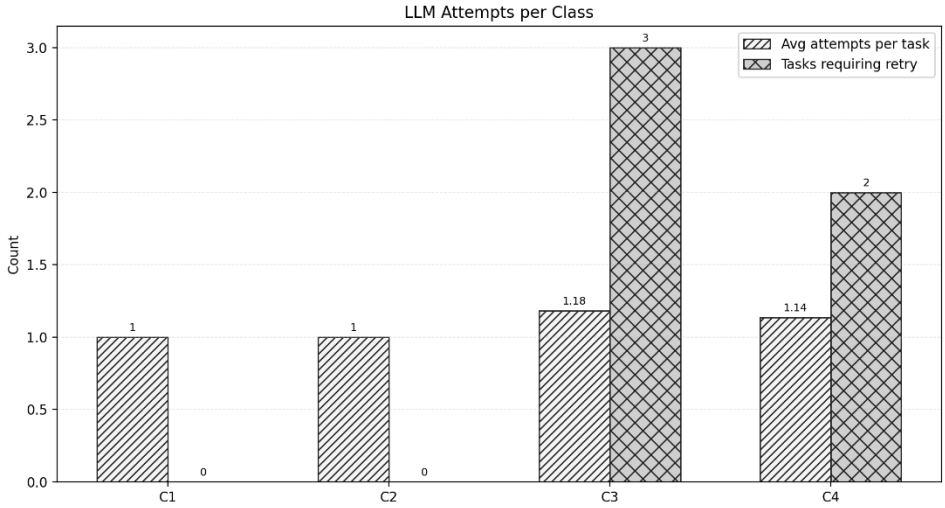


Fig. 35. Average LLM attempts per class and count of tasks requiring retry.

Task failures can be divided into two main groups: LLM planning issues and execution failures. The planning-related issues are summarized in Fig. 36. The most common planning problem was invalid storage stack build order, which occurred 6 times. Additional planning issues included 1 invalid storage stack unload order and 4 other LLM-related issues. This shows that LLM had difficulties providing clear plan on how blocks should be arranged in stacks based on provided information (chose wrong colour order, wrong stack building location) and difficulties to unload such stacks in correct order. Other LLM related issues consists of incorrect structure of provided plan, incorrect usage of the skills.

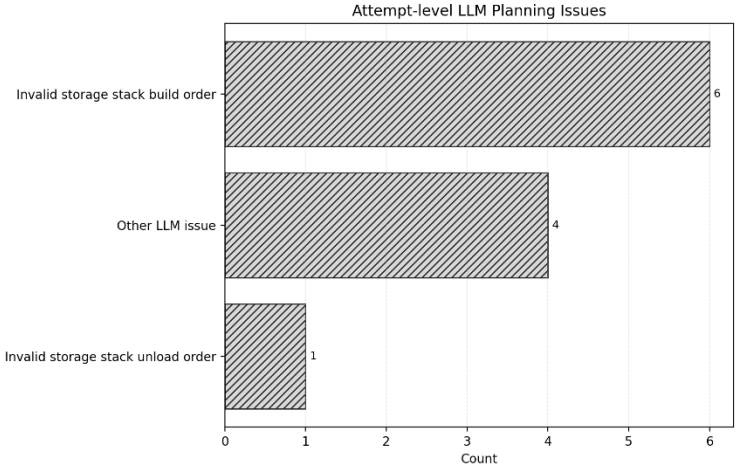


Fig. 36. Main LLM planning issues.

Execution failures are summarized in Fig. 37. The most frequent execution problem was the inability to find the target cube in the required zone, which occurred 14 times. Other execution failure categories included 3 cases where no stack cube was found in storage, 2 cases where no stack cube was found and 3 other execution failures. This distribution shows that perception-grounding mismatch and incorrect assumptions about object location were the dominant causes of unsuccessful execution. In other words, many failures did not arise because the robot could not perform a motion primitive, but because the expected object was not present in the assumed workspace region or was incorrectly taken from that location during previous manipulation steps.

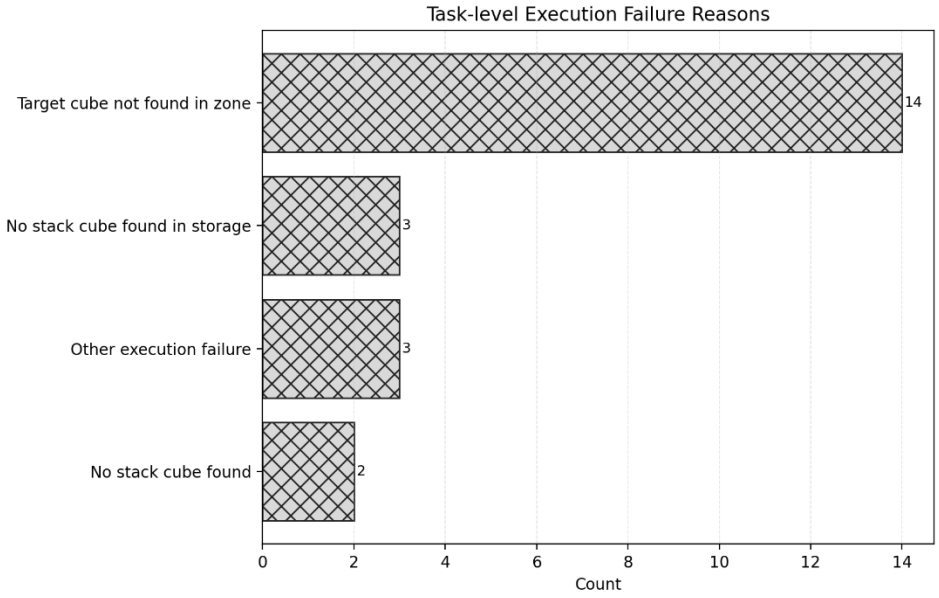


Fig. 37. Task-level execution failure reasons.

Overall, these results demonstrate that proposed methodology can solve a substantial portion of the tested tasks, but its reliability strongly depends on task structure. Simple colour-sorting tasks and many mixed-stack tasks can be completed successfully with relatively stable planning cost. However, tasks involving storage as an intermediate area or requiring precise stack-order reasoning are significantly more challenging. The main limitations observed in the experiments are related to incorrect planning of stack order, increased planning cost for more difficult tasks and execution failures caused by missing or incorrectly localized target objects. These findings support the conclusion that constrained LLM planning combined with Behaviour Tree execution is a feasible methodology for intuitive robot programming, but further improvements are needed in instruction disambiguation, stack-order reasoning and perception-grounding consistency.

Results and conclusions

1. The analysis of existing robot programming methods showed that traditional direct and indirect programming approaches are effective in industrial applications, but they remain difficult for non-expert users and require specialized technical knowledge. The literature review also showed that more intuitive robot programming becomes feasible when natural-language input is combined with structured task representation, reusable robot skills and controlled execution mechanisms such as Behaviour Trees.

2. A simulation-based end-to-end methodology for intuitive robot programming using text-based AI models was successfully developed. The created system integrates CoppeliaSim as the simulation environment, Python-based robot skills as the execution layer, Behaviour Trees as the task execution structure, Redis as the communication and shared-state layer and a large language model as a high-level planner. In the developed architecture, the LLM is not used as a low-level controller, but only as a module for generating constrained structured task plans. This solution established a clear separation between probabilistic language interpretation and deterministic robot execution. The developed system was initially validated with simplified experiment and confirmed that the full pipeline operates correctly: the LLM generated the task plan in 2.7 s using 1790 tokens, of which 1731 were prompt tokens and 59 were completion tokens.

3. The experimental evaluation showed that the proposed methodology is feasible for structure manipulation tasks, but its performance depends strongly on task complexity and on the consistency between language-based planning and execution state. In total, 88 tasks were evaluated across four task classes, and 62 tasks were completed successfully, resulting in an overall success rate of 70.45%. The highest success rates were obtained in classes C1 and C3, where 18 out of 22 tasks were completed successfully in each class (81.8 %), while class C2 achieved 15 successful executions (68.2%) and class C4 achieved 11 successful executions (50%). The most common planning issue was invalid storage stack build order, which occurred 6 times, while the most common execution failure was the inability to find target cube in required zone, which occurred 14 times. The highest average token usage was observed in class C4 (2711.41 tokens) and the most difficult tasks required repeated planning attempts, with the longest generation time reaching 21.5 s in class C3. These results show that the key required components of such a system are a constrained skill registry, Behaviour Tree-based execution, plan validation, shared state management and perception-based grounding, while the main limitations are related to stack-order reasoning, ambiguous instructions and mismatches between the generated plan and the actual scene state.

List of references

[1] P. Bilancia, J. Schmidt, R. Raffaeli, M. Peruzzini, and M. Pellicciari, 'An Overview of Industrial Robots Control and Programming Approaches', *Appl. Sci.*, vol. 13, p. 2582, Feb. 2023, doi: 10.3390/app13042582.

[2] S. Gong *et al.*, 'A Soft Collaborative Robot for Contact-based Intuitive Human Drag Teaching', *Adv. Sci.*, vol. 11, no. 24, p. 2308835, Apr. 2024, doi: 10.1002/advs.202308835.

[3] S. Lu, J. Berger, and J. Schilp, 'System of Robot Learning from Multi-Modal Demonstration and Natural Language Instruction', *Procedia CIRP*, vol. 107, pp. 914–919, Jan. 2022, doi: 10.1016/j.procir.2022.05.084.

[4] S. El Zaatari, M. Marei, W. Li, and Z. Usman, 'Cobot programming for collaborative industrial tasks: An overview', *Robot. Auton. Syst.*, vol. 116, pp. 162–180, Jun. 2019, doi: 10.1016/j.robot.2019.03.003.

[5] E. Rohmer, S. P. N. Singh, and M. Freese, 'V-REP: A versatile and scalable robot simulation framework', in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo: IEEE, Nov. 2013, pp. 1321–1326. doi: 10.1109/IROS.2013.6696520.

[6] 'AppStudio | ABB', ABB Group. Accessed: Jan. 31, 2026. [Online]. Available: https://www.abb.com/global/en/areas/robotics/products/software/appstudio

[7] 'VAL Blocks, Easy programming for your industrial robots', Staubli. Accessed: Jan. 31, 2026. [Online]. Available: https://www.staubli.com/global/en/robotics/products/robot-software/val-blocks.html

[8] J. Huang and M. Cakmak, 'Code3: A System for End-to-End Programming of Mobile Manipulator Robots for Novices and Experts', in *Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*, in HRI '17. New York, NY, USA: Association for Computing Machinery, Mar. 2017, pp. 453–462. doi: 10.1145/2909824.3020215.

[9] 'TMflow Software', OMRON Robotics. Accessed: Jan. 31, 2026. [Online]. Available: https://robotics.omron.com/products/collaborative-robots/software/

[10] D. Fogli, L. Gargioni, G. Guida, and F. Tampalini, 'A hybrid approach to user-oriented programming of collaborative robots', *Robot. Comput.-Integr. Manuf.*, vol. 73, p. 102234, Feb. 2022, doi: 10.1016/j.rcim.2021.102234.

[11] T. Lohi, S. Soutukorva, and T. Heikkilä, 'Programming of Skill-based Robots', in *2024 IEEE 19th Conference on Industrial Electronics and Applications (ICIEA)*, Aug. 2024, pp. 1–7. doi: 10.1109/ICIEA61579.2024.10664981.

[12] F. Ahmad, H. Ismail, J. Styrud, M. Stenmark, and V. Krueger, 'A Unified Framework for Real-Time Failure Handling in Robotics Using Vision-Language Models, Reactive Planner and Behavior Trees', Mar. 21, 2025, *arXiv*: arXiv:2503.15202. doi: 10.48550/arXiv.2503.15202.

[13] J. Saukkoriipi, T. Heikkilä, J. M. Ahola, T. Seppälä, and P. Isto, 'Programming and control for skill-based robots', *Open Eng.*, vol. 10, no. 1, pp. 368–376, Jan. 2020, doi: 10.1515/eng-2020-0037.

[14] T. Eiband, F. Lay, K. Nottensteiner, and D. Lee, 'Unifying Skill-Based Programming and Programming by Demonstration through Ontologies', *Procedia Comput. Sci.*, vol. 232, pp. 595–605, Jan. 2024, doi: 10.1016/j.procs.2024.01.059.

[15] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, ‘A survey of Behavior Trees in robotics and AI’, *Robot. Auton. Syst.*, vol. 154, p. 104096, Aug. 2022, doi: 10.1016/j.robot.2022.104096.

[16] P. Ögren and C. I. Sprague, ‘Behavior Trees in Robot Control Systems’, *Annu. Rev. Control Robot. Auton. Syst.*, vol. 5, no. 1, pp. 81–107, May 2022, doi: 10.1146/annurev-control-042920-095314.

[17] S. Sucker and D. Henrich, ‘A Layered Pipeline for Natural Language Robot Programming with Control Structures’, in *Annals of Scientific Society for Assembly, Handling and Industrial Robotics 2023*, S. Ihlenfeldt, T. Schüppstuhl, and K. Tracht, Eds, Cham: Springer Nature Switzerland, 2025, pp. 183–194. doi: 10.1007/978-3-031-74010-7_16.

[18] A. Radford *et al.*, ‘Learning Transferable Visual Models From Natural Language Supervision’, Feb. 26, 2021, *arXiv*: arXiv:2103.00020. doi: 10.48550/arXiv.2103.00020.

[19] W. Goodwin, S. Vaze, I. Havoutis, and I. Posner, ‘Semantically Grounded Object Matching for Robust Robotic Scene Rearrangement’, arXiv.org. Accessed: Jun. 08, 2025. [Online]. Available: https://arxiv.org/abs/2111.07975v1

[20] M. Ahn *et al.*, ‘Do As I Can, Not As I Say: Grounding Language in Robotic Affordances’, Aug. 16, 2022, *arXiv*: arXiv:2204.01691. doi: 10.48550/arXiv.2204.01691.

[21] D. Driess *et al.*, ‘PaLM-E: An Embodied Multimodal Language Model’, Mar. 06, 2023, *arXiv*: arXiv:2303.03378. doi: 10.48550/arXiv.2303.03378.

[22] A. Brohan *et al.*, ‘RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control’, Jul. 28, 2023, *arXiv*: arXiv:2307.15818. doi: 10.48550/arXiv.2307.15818.

[23] H. Singh, R. J. Das, M. Han, P. Nakov, and I. Laptev, ‘MALMM: Multi-Agent Large Language Models for Zero-Shot Robotic Manipulation’.

[24] OpenAI *et al.*, ‘GPT-4 Technical Report’, Mar. 04, 2024, *arXiv*: arXiv:2303.08774. doi: 10.48550/arXiv.2303.08774.

[25] H. Touvron *et al.*, ‘LLaMA: Open and Efficient Foundation Language Models’, Feb. 27, 2023, *arXiv*: arXiv:2302.13971. doi: 10.48550/arXiv.2302.13971.

[26] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison, ‘RLBench: The Robot Learning Benchmark & Learning Environment’, Sep. 26, 2019, *arXiv*: arXiv:1909.12271. doi: 10.48550/arXiv.1909.12271.

[27] ‘Remote API’. Accessed: Feb. 09, 2026. [Online]. Available: https://manual.coppeliarobotics.com/en/remoteApiOverview.htm

[28] CoppeliaRobotics, ‘zmqRemoteApi/PROTOCOL.md at master · CoppeliaRobotics/zmqRemoteApi’, GitHub. Accessed: Feb. 09, 2026. [Online]. Available: https://github.com/CoppeliaRobotics/zmqRemoteApi/blob/master/PROTOCOL.md

[29] *coppeliasim-zmqremoteapi-client: Client for the CoppeliaSim's zmqRemoteApi (protocol version 2)*. Python. Accessed: Feb. 09, 2026. [OS Independent]. Available: https://github.com/CoppeliaRobotics/zmqRemoteApi

Appendices

Appendix 1. Agent prompt

You are a planning agent. Given a user command and TOOLS_REGISTRY, return one JSON plan that fully satisfies the task.

Hard output rules:
- Return JSON only (no prose, no markdown).
- Use only behaviours supported by runtime: "ScanSearchArea", "StackingInit", "StackingAdvance", "TargetedSearch", "PickUpCube", "PlaceCube".
- Always start with {"behaviour":"ScanSearchArea"}.
- For every move action, use exactly: TargetedSearch -> PickUpCube -> PlaceCube.

Argument rules:
- Expand color codes: r/R -> red, g/G -> green, b/B -> blue.
- Allowed target_loc: "red plane", "green plane", "blue plane", "storage area", "<color> cube", "random".
- Allowed TargetedSearch zones: "left", "right", "top", "bottom", "center", "any", "stack", "storage".
- Include target_color in TargetedSearch unless color is controlled only by active sequence state.
- Prefer explicit target_color in TargetedSearch for reliability.

Ordering/completeness rules:
- Respect clause order strictly ("first", "then", "after sorting", "finally").
- Do not start post-processing until required sorting is complete.
- Cover every required object count from the command. Do not omit required cubes.

Stack/unstack rules:
- "from <color> stack" or "from <color> plane stack" must be:
  {"behaviour":"TargetedSearch","zone":"stack","target_color":"<color>"}.
- "from storage area" or "from storage stack" must use:
  {"behaviour":"TargetedSearch","zone":"storage","target_color":"<color>"}.
- After placing a cube to random and then "return it back", reacquire by color with zone "any", then place to its target plane.
- For "place to random, then pick it back", use direct reacquire flow:
  PlaceCube(random) -> TargetedSearch(zone="any", target_color=...) -> PickUpCube -> PlaceCube(target plane).
- If command says "make a stack in order A, B, C", interpret it as bottom->top:
  A is placed first and becomes bottom, B becomes middle, C becomes top.
- Later removals from that stack must follow the exposed top order:
  first C, then B, then A, unless the plan explicitly moves blocking cubes away.
- Do not confuse stack build order with stack removal order.
- When unloading a mixed stack, the next TargetedSearch target_color must match the cube currently on top, not the cube that was placed first.
- For mixed-stack unloading, prefer explicit TargetedSearch target_color steps in top-to-bottom order. Do not use StackingInit/StackingAdvance if that sequence would imply the wrong removal order.
- Example: if storage stack is built in order blue, red, green, then green is on top. Valid unload order is green -> red -> blue.
- If the command only asks to build one or more stacks and does not ask for later removal, return, unload, or unstack actions, then stop immediately after the final required placement.
- For build-only stack tasks, NEVER use TargetedSearch with zone "stack" or "storage".
- To build a stack, always source cubes from surface zones and place them directly onto the target location. Do not pick from a newly built stack and place it back onto the same stack.
- When the command contains multiple stack-build clauses, treat each destination independently and satisfy each required order exactly once.

Special constraint for the common fixed scene command:
- If command states:
  center has 2 red, 1 green, 1 blue; right has 1 green and 1 blue,
  and asks to sort all cubes by color,
  then sorting part must include all 6 placements before any post-step:
  - red x2 -> red plane (from center)
  - green x2 -> green plane (one center, one right)
  - blue x2 -> blue plane (one center, one right)

Special constraint for storage-stack command:
- If command says to make an intermediate mixed stack on storage plane in order blue, red, green,
  then the build phase must move exactly three cubes to storage in this exact order:
  blue -> red -> green.
- Do not move extra cubes to storage area for that task.
- Do not substitute another cube of the same color if the task describes one mixed stack with one cube per listed color.
- After that build phase, the storage unload order must be green -> red -> blue.
- A plan that builds a different storage order or unloads blue first from that stack is invalid.
- If command explicitly says which color becomes top of the stack, the first storage removal must use that color.

Use StackingInit/StackingAdvance only when it helps enforce explicit ordered color sequence.
If each TargetedSearch already has explicit target_color and zone, StackingInit/StackingAdvance can be omitted.

Schema:
```
{
  "children": [
    {"behaviour":"ScanSearchArea"},
    {"behaviour":"TargetedSearch","zone":"center","target_color":"red"},
    {"behaviour":"PickUpCube"},
    {"behaviour":"PlaceCube","target_loc":"red plane"}
  ]
}
```

If command is invalid (unsupported color/object), return:
{"warning":"instruction unclear or invalid: <reason>"}.

Appendix 2. Tools registry:

```
# Canonical tool registry for planners and validators.
# Each tool maps to a py_trees Behaviour in src/behaviours.py.

tools:
  - name: PickUpCube
    class: src.behaviours.PickUpCube
    description: Pick the currently targeted cube (uses IK + proximity + suction).
    params: {}
    blackboard:
      reads: [color_order, in_sequence, current_color_pos]
      writes: [picked_up, pickup_history, stack_history]

  - name: PlaceCube
    class: src.behaviours.PlaceCube
    description: Place the held cube at the target location.
    params:
      target_loc:
        type: string|int|null
        required: false
        allowed_values:
          - red plane
          - green plane
          - blue plane
          - storage area
          - red cube
          - green cube
          - blue cube
          - random
    blackboard:
      reads: [plane_pos, in_sequence, color_order, current_color_pos, picked_up,
color_pos_dict, pickup_history]
      writes: [picked_up, color_pos_dict, stack_history]

  - name: ScanSearchArea
    class: src.behaviours.ScanSearchArea
    description: Scan the search space with the top camera to detect plane and cube
positions by color.
    params:
      colors:
        type: list[string]|string|null
        required: false
        enum: [red, green, blue]
    blackboard:
      reads: []
      writes: [plane_pos, color_pos_dict, in_sequence]

  - name: StackingInit
    class: src.behaviours.StackingInit
    description: Initialize stacking order and related blackboard state.
    params:
      stack_color_order:
        type: list[string]|string
        required: true
        enum: [red, green, blue]
      target_loc:
```

```
        type: string|int|null
        required: false
    blackboard:
      reads: []
      writes: [color_order, in_sequence, stack_index, stack_target_loc,
stack_locations]

  - name: TargetedSearch
    class: src.behaviours.TargetedSearch
    description: Select a target cube from the scan map within a named zone.
    params:
      target_color:
        type: string|null
        required: false
        enum: [red, green, blue]
      zone:
        type: string|null
        required: false
        enum: [left, right, top, bottom, center, any, stack]
    blackboard:
      reads: [color_order, stack_index, color_pos_dict]
      writes: [current_color_pos]

  - name: StackingAdvance
    class: src.behaviours.StackingAdvance
    description: Advance to the next color in the stacking order.
    params: {}
    blackboard:
      reads: [color_order, in_sequence]
      writes: [stack_index]
```

Appendix 3. Task list class C1:

- T01 | C1 Color sorting | Sort all cubes by color: place two red cubes on red plane (both from center), two green cubes on green plane (one from center and one from right), and two blue cubes on blue plane (one from center and one from right).
- T06 | C1 Color sorting | Pick and place only cubes from the right zone first, then complete color sorting for all remaining cubes.
- T07 | C1 Color sorting | Place all red cubes first, then all green cubes, then all blue cubes, each to matching color plane.
- T11 | C1 Color sorting | Sort all cubes by color. Use center zone cubes before right zone cubes whenever both options exist.
- T12 | C1 Color sorting | Move one blue cube to blue plane, one green cube to green plane, one red cube to red plane. Then finish sorting the remaining cubes.
- T31 | C1 Color sorting | Sort all cubes by color. Always use right zone cubes before center zone cubes whenever both options exist.
- T32 | C1 Color sorting | Sort all cubes by color. Complete the blue plane first, then the green plane, and finally the red plane.
- T33 | C1 Color sorting | First move one red cube to red plane, one green cube to green plane, and one blue cube to blue plane. Then finish sorting all remaining cubes by color.
- T34 | C1 Color sorting | Sort all cubes by color. Finish all center zone cubes first, then sort the remaining right zone cubes.
- T35 | C1 Color sorting | Sort all cubes by color. Build the red stack first, then the green stack, and finally the blue stack.
- T36 | C1 Color sorting | First move both right-zone cubes to their matching color planes. After that, sort all remaining center-zone cubes by color.
- T37 | C1 Color sorting | Sort all cubes by color. Place all non-red cubes first, then place both red cubes on red plane.
- T49 | C1 Color sorting | Complete full color sorting, but process center-zone cubes first and only then process right-zone cubes, while still finishing with two cubes on each matching color plane.
- T50 | C1 Color sorting | Sort all cubes by color and prioritize building the blue stack before the green stack and the red stack.
- T51 | C1 Color sorting | First place one cube of each color onto its matching plane, then place the remaining three cubes so that each color plane ends with two cubes.
- T52 | C1 Color sorting | Perform complete color sorting with this sequence constraint: finish all green placements before any blue placement, and finish all blue placements before the second red placement.
- T53 | C1 Color sorting | Sort cubes to matching planes while taking at least one cube from the right zone before finishing the final placement on each color plane.
- T54 | C1 Color sorting | Create complete color stacks and ensure the first two placements are red to red plane and green to green plane before continuing with the remaining cubes.
- T55 | C1 Color sorting | Sort all cubes by color, starting with both center red cubes, then complete the green and blue planes with the remaining cubes.
- T56 | C1 Color sorting | Sort all cubes by color and handle all right-zone cubes consecutively in the middle of the sequence, not at the end.
- T57 | C1 Color sorting | Carry out full color sorting and force this high-level order: initialize red stack, then complete green stack, then complete blue stack, while keeping final counts correct.
- T58 | C1 Color sorting | Sort all cubes by color, but alternate destination planes as much as possible so that two consecutive placements to the same plane are avoided until unavoidable.

Appendix 4. Task list class C2:

- T02 | C2 Sort + stack manipulation | Sort all cubes by color, then unstack one cube from each color stack and place each picked cube on random free surface positions.

- T03 | C2 Sort + stack manipulation | First sort all cubes to matching color planes. Then move one top cube from red plane stack to green plane.

- T08 | C2 Sort + stack manipulation | Sort cubes by color. After sorting, move one green cube from green plane to random surface, then return it back to green plane.

- T09 | C2 Sort + stack manipulation | Sort cubes by color. After sorting, unstack blue top cube to random surface, then unstack red top cube to random surface.

- T13 | C2 Sort + stack manipulation | Sort all cubes by color, then clear all three stacks by moving one cube from each stack to random surface.

- T14 | C2 Sort + stack manipulation | Create full color sorting first. Then perform reverse unstacking order: blue stack, green stack, red stack; place each removed cube on random surface.

- T15 | C2 Sort + stack manipulation | Create three color stacks on matching planes, then move one cube from each stack to red plane in order green, blue, red.

- T16 | C2 Sort + stack manipulation | Sort all cubes by color, then move one top red cube from red stack to storage area.

- T20 | C2 Sort + stack manipulation | First sort all cubes by color. Then move one top blue cube from blue stack to storage area, and after that move the same blue cube from storage area to red plane.

- T23 | C2 Sort + stack manipulation | Sort all cubes by color. Then move one top cube from blue stack, one from green stack, and one from red stack to storage area in that order.

- T26 | C2 Sort + stack manipulation | First create full color stacks. Then move one top green cube from green stack to storage area, and return it back to green plane.

- T29 | C2 Sort + stack manipulation | First sort all cubes by color. Then move one top red cube and one top blue cube to storage area. After that, return blue to blue plane and red to red plane.

- T59 | C2 Sort + stack manipulation | First complete full color sorting. Then remove one top blue cube to random surface and one top green cube to random surface, in that order.

- T60 | C2 Sort + stack manipulation | Sort all cubes by color. After sorting, move one top red cube from red stack to green plane, then move one top green cube from green stack to blue plane.

- T61 | C2 Sort + stack manipulation | Complete color sorting first. Then unstack one cube from each color stack and place all three removed cubes on random surface positions in order red, green, blue.

- T62 | C2 Sort + stack manipulation | Sort by color; afterward, move one top green cube from green stack to storage area and immediately return that same cube to green plane.

- T63 | C2 Sort + stack manipulation | Build complete color stacks, then clear only the blue and red stacks by moving one top cube from each to random surface.

- T64 | C2 Sort + stack manipulation | Fully sort all cubes. Then remove one top cube from red stack to storage area, one top cube from blue stack to storage area, and return blue first and red second to their matching planes.

- T65 | C2 Sort + stack manipulation | Perform complete sorting. Then move one top cube from each stack onto red plane in order blue, green, red.

- T66 | C2 Sort + stack manipulation | Sort all cubes by color, then move one top red cube to random surface, pick it again from random surface, and place it back on red plane.

- T67 | C2 Sort + stack manipulation | First create full color stacks. Next, unstack one top green cube to storage area and one top blue cube to storage area; finally return both to matching planes in reverse order.

- T68 | C2 Sort + stack manipulation | After full sorting, move one top cube from green stack to random surface and one top cube from red stack to random surface, then move only the green cube back to green plane.

Appendix 5. Task list class C3:

- T04 | C3 Mixed-stack construction | Build one mixed stack on red plane in this order from bottom to top: red, green, blue. Use one cube per color only.

- T05 | C3 Mixed-stack construction | Build two mixed stacks: first stack on red plane with red then green then blue; second stack on blue plane with red then green then blue.

- T17 | C3 Mixed-stack construction | Build a mixed stack on storage plane in order red, blue, green. Use one cube per color only.

- T19 | C3 Mixed-stack construction | Build a temporary storage stack in order green, blue using one cube of each color. Then move blue to blue plane and green to green plane. Leave all other cubes untouched.

- T22 | C3 Mixed-stack construction | Build one mixed stack on green plane in order green, red, blue. Use one cube per color only.

- T27 | C3 Mixed-stack construction | Build a mixed stack on storage plane in order blue, green, red. Then move only the top red cube from storage area to red plane. Leave the other storage cubes in place.

- T38 | C3 Mixed-stack construction | Build one mixed stack on blue plane in order blue, red, green. Use one cube per color only.

- T39 | C3 Mixed-stack construction | Build one mixed stack on storage plane in order green, red, blue. Use one cube per color only.

- T40 | C3 Mixed-stack construction | Build two mixed stacks: one on storage plane in order red, blue, green, and one on green plane in order green, blue.

- T41 | C3 Mixed-stack construction | Build one mixed stack on blue plane in order red, green, blue. Use one cube per color only.

- T42 | C3 Mixed-stack construction | Build a temporary mixed stack on storage plane in order blue, green. Use one cube of each color only and leave all other cubes unchanged.

- T43 | C3 Mixed-stack construction | Build one mixed stack on red plane in order green, blue, red. Use one cube per color only.

- T69 | C3 Mixed-stack construction | Build one mixed stack on red plane in order blue, green, red. Use one cube per color only and stop after stack is built.

- T70 | C3 Mixed-stack construction | Build one mixed stack on storage plane in order red, green, blue. Use one cube per color only and stop after final placement.

- T71 | C3 Mixed-stack construction | Build one mixed stack on blue plane in order green, red, blue. Use one cube per color only.

- T72 | C3 Mixed-stack construction | Build two mixed stacks: first on red plane in order red, blue, green; second on storage plane in order green, blue. Use one cube per listed color in each stack and stop.

- T73 | C3 Mixed-stack construction | Construct a temporary mixed stack on storage plane in order blue, red. Use one cube of each color only and leave all other cubes untouched.

- T74 | C3 Mixed-stack construction | Build one mixed stack on green plane in order blue, green, red, using one cube per color only, and do not perform any unstacking action afterward.

- T75 | C3 Mixed-stack construction | Build one mixed stack on storage plane in order green, blue, red. Use one cube per color only and stop.

- T76 | C3 Mixed-stack construction | Create two build-only stacks: one on blue plane in order blue, red; one on green plane in order green, blue. Do not move cubes from built stacks.

- T77 | C3 Mixed-stack construction | Build one mixed stack on red plane in order red, green, blue using one cube per color, then end the plan.

- T78 | C3 Mixed-stack construction | Build one mixed stack on storage plane in order blue, green, red and one mixed stack on blue plane in order red, blue. Use only surface cubes to build both stacks.

Appendix 6. Task list class C4:

- T10 | C4 Storage-mediated | Make an intermediate mixed stack on storage plane in order blue, red, green. Then move each cube from that mixed stack in storage area to matching color plane.

- T18 | C4 Storage-mediated | Move both cubes from right zone to storage area first. Then move those storage cubes to matching color planes. After that, sort the remaining center cubes.

- T21 | C4 Storage-mediated | Move one red cube and one green cube to storage area first. Then return them to matching planes. After that, finish sorting the remaining cubes by color.

- T24 | C4 Storage-mediated | Build a mixed stack on storage plane in order red, green, blue. Then unstack it to random surface positions in order blue, green, red.

- T25 | C4 Storage-mediated | Move both right-zone cubes to storage area. Then move them from storage area to matching planes. Finally, sort all remaining cubes from center zone.

- T28 | C4 Storage-mediated | Move one blue cube from center zone, one green cube from right zone, and one red cube from center zone to storage area in that order. Then move them from storage area to matching planes in reverse order.

- T30 | C4 Storage-mediated | Build a temporary two-cube stack on storage plane in order green, blue using one cube of each color. Then move blue to blue plane and green to green plane. Finally, sort the remaining cubes by color.

- T44 | C4 Storage-mediated | Move one red cube from center zone and one blue cube from right zone to storage area in that order. Then move blue to blue plane and red to red plane. Finally, sort the remaining cubes by color.

- T45 | C4 Storage-mediated | Move one green cube and one blue cube from center zone to storage area. Then move them from storage area to matching planes. After that, sort the remaining cubes by color.

- T46 | C4 Storage-mediated | Build a temporary storage stack in order red, green using one cube of each color. Then move green to green plane and red to red plane. Finally, sort the remaining cubes by color.

- T47 | C4 Storage-mediated | Move both blue cubes to storage area one after another. Then move both blue cubes from storage area to blue plane. After that, sort all red and green cubes by color.

- T48 | C4 Storage-mediated | Move one red cube and one green cube to storage area first. Then move them to matching planes in reverse order. Finally, sort the remaining cubes by color.

- T79 | C4 Storage-mediated | Move one red cube and one blue cube to storage area. Then move red to red plane and blue to blue plane. Finally, sort remaining cubes by color.

- T80 | C4 Storage-mediated | Move one green cube from center zone and one blue cube from right zone to storage area in that order; then move them to matching planes in reverse order; then sort all remaining cubes.

- T81 | C4 Storage-mediated | First move both right-zone cubes into storage area. Next, move those two cubes from storage to their matching planes. Finally, finish sorting center-zone cubes by color.

- T82 | C4 Storage-mediated | Move one red cube to storage area, then one green cube to storage area, then one blue cube to storage area. After staging all three, return them from storage to matching planes in order blue, green, red.

- T83 | C4 Storage-mediated | Move one blue cube to storage area and immediately move it to blue plane; then move one green cube to storage area and immediately move it to green plane; then sort all remaining cubes by color.

- T84 | C4 Storage-mediated | Stage one red and one green cube in storage area, return green first then red to matching planes, and then perform full sorting for all cubes not yet sorted.

- T85 | C4 Storage-mediated | Place one green cube into storage area, place one blue cube into storage area, then move both from storage to matching planes and finish by sorting both red cubes to red plane.

- T86 | C4 Storage-mediated | Move one red cube and one blue cube from center zone to storage area; move blue to blue plane and red to red plane; then complete sorting of all remaining cubes.

- T87 | C4 Storage-mediated | Move one green cube from right zone and one red cube from center zone to storage area, return them to matching planes, and then sort the remaining cubes by color.

- T88 | C4 Storage-mediated | Use storage area as temporary buffer for one blue cube and one green cube, then place those cubes on matching planes, and finally perform complete color sorting for any cubes still on the surface.