Robotica (2026), **44**, pp. 1516–1533
doi:10.1017/S0263574726103646

CAMBRIDGE UNIVERSITY PRESS

RESEARCH ARTICLE

A dual-agent framework for physically grounded and syntactically verifiable industrial robot programming

Zhendao Chen[1], Haibo Chao[1], Yanhao He[2], Che Hou[2], Wenjie Chen[2] and Ning Tan[1,3]

[1]School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China
[2]State Key Laboratory of High-end Heavy-load Robots, Midea Group, Foshan, China
[3]State Key Laboratory of Intelligent Manufacturing Equipment and Technology, Huazhong University of Science and Technology, Wuhan, China
Corresponding author: Ning Tan; Email: tann5@mail.sysu.edu.cn

Received: 13 March 2026 UTC; **Revised:** 9 May 2026 UTC; **Accepted:** 21 May 2026 UTC;
First published online: 8 July 2026 UTC

Keywords: redundant manipulators; control of robotic systems; serial manipulator design and kinematics; novel applications of robotics; motion planning

Abstract

The integration of large language models (LLMs) into KUKA industrial robots offers a promising pathway toward intuitive natural language control and a higher level of intelligence. However, existing methods face two primary challenges: their reliance on predefined motion primitives and the syntactic fragility imposed by the strict constraints of proprietary KUKA Robot Language (KRL). To address these challenges, we propose MIKI (multi-agent integrated KUKA interface), a decoupled dual-agent framework that bridges high-level reasoning and low-level syntactic verification for industrial robot programming. The controller agent incorporates a chain-of-thought (CoT) reasoning mechanism to systematically decompose abstract human instructions into logical intermediate steps. By leveraging a visual toolchain, it translates these reasoned plans into a sequence of executable actions, each embedded with precise world coordinates. Subsequently, the code generation agent employs a syntax-guided iterative refinement process, underpinned by an ANTLR4 parser, to transform these actions into verified and executable KRL code. Experimental results demonstrate that this closed-loop approach substantially improves the syntactic success rate of code generation from 0.61 to 0.89. Furthermore, the framework achieves an average success rate of 0.6 across five desktop manipulation tasks of varying complexity using a physical KUKA robotic arm, validating its effectiveness in bridging high-level cognitive reasoning with deterministic industrial execution.

1. Introduction

Industrial manipulators within the KUKA ecosystem are foundational to modern automated manufacturing. Traditionally, these systems demand highly specialized expertise in the KUKA Robot Language (KRL), a process that is rigid, time-consuming, and poorly suited for the dynamic requirements of unstructured environments. While the emergence of large language models (LLMs) has opened new avenues for intuitive natural language interfaces in robotics [24, 26], direct end-to-end generation from human prompts to executable KUKA robot code remains a formidable challenge.

Two fundamental limitations hinder existing LLM-based approaches. First, most methods rely on a closed set of predefined motion primitives. This reliance inevitably sacrifices flexibility and adaptability, as the system cannot autonomously generate novel motions or adjust to unexpected scene variations. Second, existing methods exhibit severe syntactic fragility when generating codes in proprietary languages such as KRL. Unlike general-purpose languages like Python, KRL imposes strict, domain-specific constraints (e.g., exact structure initializations and mandatory data file declarations) that LLMs frequently hallucinate or omit. In industrial automation, where even minor spatial miscalculations or syntactic violations may result in costly damage or safety hazards, these limitations render purely end-to-end LLM solutions impractical.

© The Author(s), 2026. Published by Cambridge University Press.

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

To bridge this critical gap, a structural redesign is necessary. We observe that the two key requirements – open-vocabulary physical grounding of abstract instructions and strict adherence to a proprietary language's syntax – are fundamentally conflicting when forced into a single model. An LLM that attempts both simultaneously often compromises on one or both aspects, leading to degraded performance. Therefore, we propose a decoupled dual-agent collaborative framework that explicitly separates high-level cognitive reasoning from low-level syntactic implementation.

The core contributions of this work are as follows:

1. **Dual-agent architecture system:** We present a decoupled dual-agent framework designed to bridge the gap between high-level cognitive reasoning and deterministic industrial execution for KUKA robots. The controller agent decomposes complex instructions into logical sub-tasks. By integrating a visual toolchain, this agent grounds these plans into precise world coordinates, reducing spatial hallucinations. Concurrently, the code generation agent then takes these grounded plans and translates them into syntactically verifiable KRL code, ensuring seamless and robust execution on the physical robot. We validate the proposed framework through real-world experiments on a KUKA manipulator. The system achieves a 60% success rate in desktop manipulation tasks, demonstrating its capability to handle long-horizon planning and precise execution in practical scenarios.
2. **Syntax-guided refinement code synthesis:** We introduce an iterative refinement loop for code generation. This loop is primarily powered by an ANTLR4-based parser, which actively detects KRL-specific syntax errors and provides feedback. Furthermore, the system is augmented by a specialized KRL knowledge corpus, providing domain-specific insights for more accurate code generation. Our benchmarks demonstrate that the closed-loop approach enhances the syntactic reliability of KRL code, increasing the success rate from 0.61 to 0.89.
3. **KRL-bench: A dedicated code generation benchmark:** We introduce a dedicated benchmark suite for evaluating the ability of LLMs to generate syntactically correct and functionally accurate KRL code. This benchmark comprises 63 meticulously designed test cases, covering the full spectrum of KRL programming, from variable declarations, expressions, and logical control statements to functions, subprograms, and critical motion commands.

2. Related work

2.1. LLMs for robot planning and control

The emergence of LLMs has shifted the paradigm of robot programming from manual coding to natural language instruction. Some approaches utilized LLMs primarily for high-level task decomposition, breaking down complex instructions into sequences of predefined actions [1, 6]. Some other works have leveraged the code-generation capabilities of models like GPT-4 to directly produce executable policies, providing more flexible control ability. For instance, Code as Policies [11] generates Python functions for embodied control, while ProgPrompt [20] employs program-based prompting to simulate world states, enabling LLMs to more conveniently acquire domain knowledge. Inner Monologue [8] incorporates environment feedback to refine plans, and LLM + P [13] combines LLMs with classical planners using PDDL for long-horizon tasks. SayPlan [17] scales task planning by grounding abstract instructions in 3D scene graphs. ReAct [18] synergizes reasoning and acting to improve interactive decision-making, and KnowNo [28] provides statistical guarantees for task completion under uncertainty. LayoutGPT [5] extends compositional visual planning by generating structured layouts from language. Holodeck [27] leverages LLMs to generate interactive 3D environments conditioned on language descriptions. 3D-Generalist [22] proposes a self-improving vision-language-action model for crafting 3D worlds, demonstrating the potential of closed-loop interaction. Scenethesis [12] introduces a language-vision agentic framework for hierarchical 3D scene generation. LayoutVLM [21] combines vision-language models with differentiable optimization for 3D layout synthesis. Recently, several system-level frameworks have been proposed to bridge LLMs with robotic execution via intermediate logic or open-source

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

middleware. For instance, ROSA [19] employs a ReAct agent architecture to translate natural language into ROS/ROS2 commands, utilizing a predefined set of Python tools to interact with the robot operating system. In the context of Human–Robot Interaction (HRI), LaMI [23] leverages LLMs to process multi-modal inputs and trigger predefined ”atomic actions” alongside rule-based social expressions. Similarly, Chekam et al. [3] propose a framework that combines LLMs with Structured Behavior Trees (BTs), where the LLM acts as a cognitive interface to dynamically activate domain-specific perception and control plugins.

However, despite these impressive advancements, existing approaches share a fundamental limitation: they heavily depend on or abstract over a set of predefined robot motion primitives. Consequently, these methods typically output generic Python, JSON format, or high-level pseudo-code. These outputs cannot be directly executed by industrial controllers, which demand proprietary, strictly typed, and low-level languages such as the KRL. Our work addresses this critical gap by specifically targeting the syntax-compliant generation of strictly typed KRL code from scratch, rather than relying on generic motion scripts or external ROS wrappers.

2.2. Visual grounding and perception

To execute manipulation tasks, an agent must ground semantic concepts into physical coordinates. Traditional methods rely on fixed object detectors, which lack the flexibility to handle open-vocabulary queries and generalize to novel objects. The advent of vision-language models (VLMs) such as CLIP [10] and robust segmentation models like the Segment Anything Model (SAM) [16] has enabled more dynamic and flexible scene understanding. These models allow for semantic identification of objects based on natural language descriptions, greatly enhancing the adaptability of robotic systems. VoxPoser [7] uses VLMs to generate 3D value maps for trajectory planning, providing a cost function for robot movements based on semantic goals. Grounding DINO [14] marries transformer-based detection with grounded pre-training, achieving state-of-the-art performance in open-set object detection, which is crucial for identifying diverse objects in unstructured environments. Depth anything [9] repurposes diffusion-based image generators for robust monocular depth estimation, providing essential 3D information from 2D images. Molmo [4] provides open-weight and open-data VLMs that excel in vision-language tasks, offering versatile tools for robotic perception. Architect [25] employs hierarchical 2D inpainting to generate interactive 3D scenes, demonstrating the utility of visual priors for scene understanding and manipulation. While these models offer powerful semantic capabilities, translating semantic understanding into precise, actionable 3D coordinates for industrial robots remains a challenge. Many VLM-based approaches provide bounding boxes or coarse 3D estimates, which are insufficient for the sub-millimeter precision required in industrial settings. We employ a modular perception toolchain that explicitly transforms semantic identification into precise world coordinates (P_{WL}) via eye-to-hand calibration, providing a deterministic geometric basis for the subsequent code generation.

2.3. Closed-loop code generation

Despite their capabilities, LLMs are prone to “hallucinations,“ particularly regarding syntax rules of less common programming languages. This issue is exacerbated when dealing with proprietary industrial robot languages like KRL, which have strict and often idiosyncratic syntax. In software engineering, iterative refinement, where a model repairs its own code based on execution feedback, has proven effective. In robotics, similar feedback mechanisms have been explored [2, 29]. Our framework introduces a syntax-aware feedback loop utilizing an ANTLR4-based parser [15]. This ensures that the generated KRL code is syntactically validated and corrected before it reaches the robot controller, significantly enhancing system reliability and safety.

3. Methodology

In this section, we present MIKI, a multi-agent integrated KUKA interface that decomposes natural language instructions into verifiable robot code. The framework consists of two core agents: a controller agent for reasoning and visual grounding and a code generation agent for syntax-guided refinement.

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

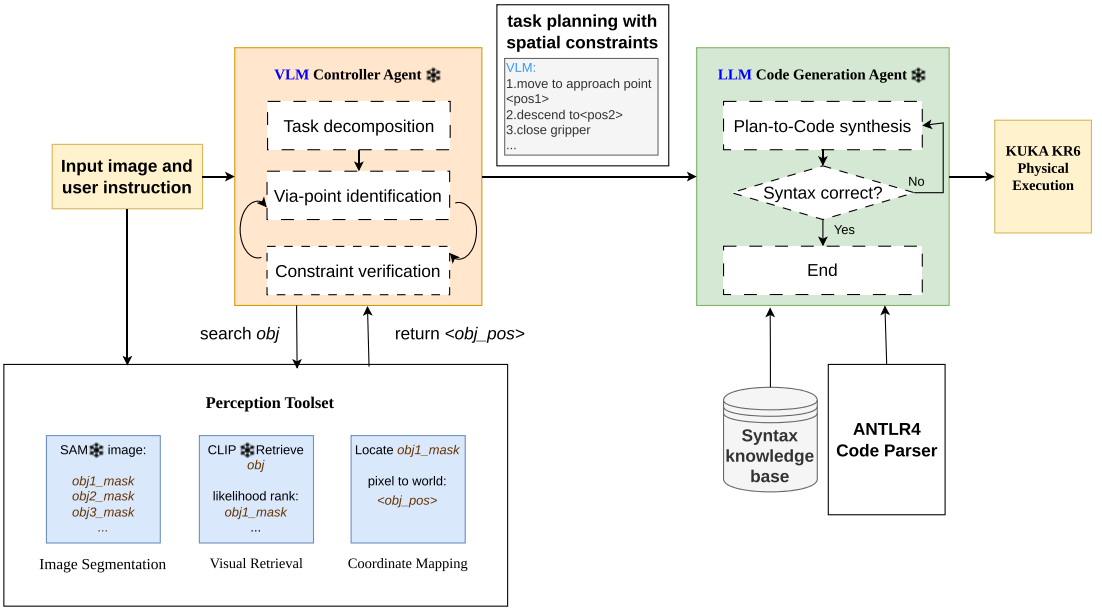


Figure 1. *Overview of the MIKI dual-agent framework. The controller agent decomposes natural language instructions into grounded action sequences via chain of thought with a visual toolchain. The code generation agent translates these plans into syntactically verified KRL code through an iterative generate-verify-refine loop powered by an ANTLR4 parser and a specialized KRL knowledge corpus.*

As illustrated in Figure 1, the system partitions the task into two distinct phases: r*easoning-guided task planning* performed by the controller agent ($\mathcal{A}_{ctrl}$) and s*yntax-compliant code generation* performed by the code generation agent ($\mathcal{A}_{krl}$).

3.1. System architecture

The overall workflow transforms a high-level natural language instruction T_{NL} into an executable KRL script C_{krl} through the following composition:

$$C_{krl} = \mathcal{A}_{krl}(\mathcal{A}_{ctrl}(T_{NL}, \mathcal{T}_{visual}), \mathcal{G}_{krl}) \tag{1}$$

where $\mathcal{T}_{visual}$ denotes the vision toolchain and $\mathcal{G}_{krl}$ represents the formal KRL grammar. In this architecture, $\mathcal{A}_{ctrl}$ functions as the cognitive core, generating a structured intermediate plan P_{NL} that includes precise 3D world coordinates. Subsequently, $\mathcal{A}_{krl}$ serves as the deterministic translator that ensures the plan is implemented within the rigid syntactic constraints of the robot controller.

3.2. Controller agent: Reasoning and visual grounding

The controller agent ($\mathcal{A}_{ctrl}$) is responsible for transforming a high-level natural language instruction T_{NL} into a structured intermediate plan P_{NL} that incorporates precise 3D world coordinates. This agent first decomposes complex tasks into logically ordered sub-tasks and then iteratively refines the plan through closed-loop interaction with the visual toolchain $\mathcal{T}_{visual}$. Specifically, $\mathcal{A}_{ctrl}$ follows a two-stage process:

1. Raw action sequence synthesis: The agent first generates an initial action plan through a structured reasoning process. Given a natural language instruction such as *"Stack the purple cube on top of the red cube,"* $\mathcal{A}_{ctrl}$ performs step-by-step reasoning to explicitly articulate intermediate decisions. An example the model's reasoning trace is illustrated below:
   Prompt (abbreviated): *"You are a robot controller. For a given task: (1) Plan an action sequence (approach, grasp, place, retract, etc.). Grasp at the geometric center. (2) List all objects with*

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

unique identifiers. Represent each coordinate in the action sequence as [ObjectName_x + bias, ObjectName_y + bias, ObjectName_z + bias], to be grounded later. Account for cumulative changes from previous steps."

Large language model's output (abbreviated):

(a) Identify objects: purple cube, blue cube, yellow cube (to be stacked), red cube (fixed base).
(b) Determine grasp strategy: All cubes are 2 cm tall. Grasp at geometric center → z = 1 cm above table. Safe travel height = 5 cm. Stack height increases by 2 cm after each placement.
(c) Plan motion sequence:

(i) Move to approach position above target: `[target_x, target_y, 5]`.
(ii) Descend to grasp position: `[target_x, target_y, 1]`. Grasp.
(iii) Retract to approach height: `[target_x, target_y, 5]`.
(iv) Move to approach position above current stack (red cube initially): `[red_x, red_y, 5]`.
(v) Descend to place position: place bottom of cube at current stack height H; its center is at `[red_x, red_y, H + 1]`. (Initial $H = 0$; after each placement $H \leftarrow H + 2$.)
(vi) Place the cube and retract to `[red_x, red_y, 5]`.
(vii) . . .

(d) Object list: `[purple_cube, blue_cube, yellow_cube, red_cube]`.

The raw sequence serves as a logical skeleton that decomposes the high-level instruction into a sequence of actions with symbolic object references. It also provides a structured blueprint for guiding subsequent perceptual queries and establishes initial spatial constraints like grasp heights or safe waypoints.

2. Reasoning-action loop and plan refinement: Following the initial plan generation, the agent engages in an iterative thought-execution-observation that progressively grounds the plan in physical reality. The loop operates as follows:

(a) *Perception call*: The agent invokes the corresponding function from the visual toolchain $\mathcal{T}_{visual}$. The toolchain returns the 3D coordinates P_w in the robot's base frame, as described in Section 3.3.
(b) *Plan update*: The received coordinates are inserted into the plan.
(c) *Constraint verification*: Before finalizing the plan, the agent checks whether the updated action satisfies predefined physical constraints. These constraints are encoded as part of the system prompt and are evaluated during the reasoning process. Examples include:

- Center-of-mass grasping: The grasp point must be the object's centroid, not a point on its surface, to ensure balanced lifting. The agent verifies that the z-coordinate of the grasp point corresponds to the object's vertical center (half-height above the table).
- Gripper clearance: The approach and retract trajectories must maintain a minimum safe distance from any obstacles, including the object itself during approach. The agent checks that the pre-approach waypoint is sufficiently above the object and that the path does not intersect other objects.
- Collision avoidance: In multi-object tasks, the agent ensures that intermediate waypoints are placed to avoid knocking over nearby objects.

In the current implementation, this constraint verification relies primarily on prompt-based arithmetic and explicitly defined rules rather than an external geometric solver or physical simulator. Once the visual toolchain provides precise metric coordinates, the LLM utilizes these values to perform basic scalar comparisons (e.g., verifying if the Z-axis approach height exceeds a predefined clearance threshold). We acknowledge that while this lightweight, heuristic approach is viable and effective for structured tabletop manipulation tasks with simple geometries, it serves mainly as a coarse logical check. For more complex, cluttered industrial environments that require dense collision avoidance and complex

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press



(a) Raw RGB Input (f_{obs}) (b) Multi-object Grounding (P_w)

Figure 2. *Visual perception and spatial grounding result in retrieving all cubes. (a) Original workspace view. (b) Processed output where all cubes are segmented and mapped to 3D world coordinates (P_w).*

inverse kinematics, relying solely on LLMs is insufficient. In such scenarios, integrating a dedicated motion planner or geometric simulator into the framework would be essential.

3.3. Visual toolchain and grounding mechanism

The controller agent interacts with the physical environment through a modular vision toolchain $\mathcal{T}_{\text{visual}}$ implemented as a set of Python-callable tools. These tools abstract raw sensor data into functional APIs, enabling open-vocabulary object grounding and precise spatial localization. The toolchain consists of six core tools: `GetObsImage`, `SAM`, `ImageCrop`, `CLIPRetrieval`, `GetObjPixelCenter`, and `Pixel2World`. The visual perception result is illustrated in Figure 2.

3.3.1. Open-vocabulary segmentation (f_{seg})

The `SAM` toolinvokes the SAM [10] to generate a dense set of object masks from the image. The automatic mask generator is configured with parameters: `points_per_side` = 32, `pred_iou_thresh` = 0.86, `stability_score_thresh` = 0.95, `crop_n_layers` = 1, `crop_n_points_downscale_factor` = 2, and `min_mask_region_area` = 100. These settings were tuned based on our robotic workspace. They were adjusted to account for the overhead lighting conditions and the physical scale of the target objects to minimize background noise and false positives, balancing mask quality and quantity for cluttered desktop scenes. Raw SAM masks are filtered to retain only those likely to represent physical objects:

- Size filtering: Masks with bounding box width or height < 10 px or > 400 px, or area < 700 px, are discarded.
- Non-Maximum Suppression: Overlapping masks are suppressed using an IoU threshold of 0.2. Masks are sorted by area (as a confidence proxy), and for any pair with IoU > 0.2, the smaller mask is removed.

The final output is a list of dictionaries, each containing: `segmentation`, a boolean numpy array of the mask; `bbox`, bounding box in [x, y, width, height] format; and `area`, pixel area of the mask.

3.3.2. Semantic retrieval via CLIP (f_{ret})

The `CLIP` tool then matches a natural language query against these cropped images using OpenCLIP [16] with a ViT-H-14 model. OpenCLIP was selected because it provides highly robust zero-shot open-vocabulary semantic matching. This capability allows the system to efficiently match natural language queries such as object colors or shapes to cropped image segments without requiring any task-specific fine-tuning or training. The model is loaded once and reused. For each cropped image $\mathbf{I}_i$, the vision encoder Φ_v produces a normalized feature vector $\mathbf{v}_i$. The text query T is tokenized and encoded by the text encoder Φ_t to obtain $\mathbf{t}$. Cosine similarity is computed for all images: $\text{sim}_i = \mathbf{v}_i \cdot \mathbf{t}$.

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

To obtain a probability-like ranking, the raw similarity scores are standardized and then passed through a softmax:

$$s_i = \text{softmax}\left(\frac{\text{sim}_i}{\sigma}\right)_i, \tag{2}$$

where σ is the mean and standard deviation of the similarity scores. This adaptive scaling avoids a fixed temperature parameter and works well across varying score distributions. Indices with $s_i > 0.1$ are returned in descending order of s_i. If multiple objects of the same category exist, the tool returns a ranked list, allowing the agent to select distinct instances.

3.3.3. Pixel to world coordinate mapping (f_{map})

Given a selected mask index j, the `GetObjPixelCenter` tool computes the pixel centroid (u_c, v_c) of its bounding box:

$$u_c = x_{\min} + \frac{w}{2}, \quad v_c = y_{\min} + \frac{h}{2}. \tag{3}$$

The `Pixel2World` tool converts this pixel coordinate to a 3D point in the robot base frame. It first reads the depth value d_{raw} at (u_c, v_c) from the depth image and converts it to meters using the depth scale s: $d = s \cdot d_{\text{raw}}$. Using the camera intrinsic matrix $\mathbf{K}$ (from `REALSENSE_RGB_INTRINSIC`), the point in camera coordinates is:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \mathbf{K}^{-1} \begin{bmatrix} u_c\, d \\ v_c\, d \\ d \end{bmatrix} = \begin{bmatrix} \dfrac{(u_c - c_x)d}{f_x} \\ \dfrac{(v_c - c_y)d}{f_y} \\ d \end{bmatrix}, \tag{4}$$

where (c_x, c_y) is the principal point and f_x, f_y the focal lengths. The point is then transformed to the robot base frame using the calibrated extrinsic matrix $\mathbf{T}_{\text{cam}\to\text{base}}$ (from `REALSENSE_T_CAM_TO_BASE`):

$$\begin{bmatrix} X_b \\ Y_b \\ Z_b \\ 1 \end{bmatrix} = \mathbf{T}_{\text{cam}\to\text{base}} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}. \tag{5}$$

Because the centroid lies on the visible top surface of the object, the resulting (X_b, Y_b, Z_b) corresponds to the top-surface center. The final coordinates are returned in millimeters to match the unit expected by KRL motion commands.

3.3.4. Integration with controller agent

The tools are designed to be invoked sequentially by the controller agent during its reasoning loop. For example, after generating an action plan that references "purple cube," the agent calls:

1. `GetObsImage` to obtain current scene data.
2. `SAM` to generate masks.
3. `ImageCrop` to extract object images.
4. `CLIP` with query to obtain the mask index.
5. `GetObjPixelCenter` and `Pixel2World` to compute the world coordinates.

This modular, query-driven interaction grounds spatial references in the physical scene, significantly reducing hallucinations and providing physically consistent coordinates that enable reliable execution by the code generation agent.

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

Table I. *KRL-bench benchmark suite structure.*

Unit name	Capabilities tested	Number of questions
Variable declaration	Declaring variables (e.g., INT, REAL, E6POS, FRAME)	21
Expressions	Arithmetic, logical, and string expressions	14
Control statements	IF-ELSE, FOR, WHILE loops, SWITCH statements	12
Functions & subprograms	Defining and calling functions	5
Motion commands	PTP, LIN, CIRC motions, relative movements	11
Total number of questions		**63**

3.4. Code generation agent: syntax-guided refinement synthesis

The code generation agent ($\mathcal{A}_{krl}$) is responsible for the final synthesis of the robot program. To handle the syntax fragility of KRL, we implement an iterative Generate-Verify-Refine loop.

3.4.1. Syntactic enforcement via ANTLR4

The agent translates the structured plan P_{NL} into KRL code C_k. Due to the strict requirements of KRL, such as specific `E6POS` structure initialization and `DAT` file variable declarations, a deterministic syntax parser built with **ANTLR4** is utilized to verify the output. The parser evaluates the code against the formal KRL grammar $\mathcal{G}_{krl}$:

$$F_k = \text{Parser}(C_k, \mathcal{G}_{krl}) \tag{6}$$

If the parser detects errors ($F_k \neq \emptyset$), such as missing semicolons or incorrect structure fields, it generates a structured error report including the line number and the expected token type.

3.4.2. Iterative refinement loop

If syntax errors are identified, the error report F_k is fed back to $\mathcal{A}_{krl}$ as a corrective prompt. The agent then regenerates the code segment, specifically targeting the reported violations. This closed-loop process continues until the code passes the ANTLR4 verification or reaches a maximum iteration limit k_{max}. This mechanism effectively transforms the LLM from a stochastic text generator into a more reliable code generator.

3.5. KRL-bench: A dedicated KRL code generation benchmark

To rigorously evaluate the capabilities of our framework and the underlying large language models in generating industrial robot programs, we developed *KRL-bench*. This benchmark suite consists of 63 meticulously curated test cases designed to assess the generation of syntactically correct and functionally accurate KRL code. KRL-bench covers a wide spectrum of programming constructs essential for industrial automation, including variable declarations, expressions, logical control statements (e.g., IF, FOR loops), function and subprogram definitions, and crucial robot motion commands. Each test case is defined by a natural language task description and a corresponding correct KRL reference solution, enabling a precise, automated evaluation of generated code against strict syntactic and semantic criteria. The comprehensive nature of KRL-bench allows for a nuanced assessment of LLM performance across various complexities of KRL programming, serving as a vital tool for both development and comparative analysis. The structure of KRL-bench is detailed in Table I. An example of a test case from KRL-bench, illustrating a function declaration and call, is shown in Figure 3.

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

```
p_name: Functions & Subprograms
p_inst: Declare function CalculateArea that accepts a REAL parameter (radius) and
returns area of the circle. Call this function to assign the area of a circle with
radius 3.0 to variable called area.
p_sol:
DEFFCT REAL CalculateArea(radius:IN)
DECL REAL PI
DECL REAL radius
DECL REAL result
PI = 3.14159
result = PI * radius * radius
RETURN (result)
ENDFCT
DEF MAIN_PROG()
DECL REAL area
BAS (#INITMOV,0 )
area = CalculateArea(3.0)
END
```

Figure 3. *An example from KRL-bench.*

The evaluation of KRL-bench is conducted through two steps to ensure both syntactic correctness and semantic fidelity. First, for each generated KRL code snippet, the ANTLR4-based parser is employed to perform a strict syntactic validation. Any code that fails to compile or contains syntax errors is immediately flagged as incorrect. Second, for syntactically valid code, a separate large language model (specifically, Qwen3) is utilized as a review agent to assess functional correctness against the provided reference solution. This review LLM acts as an ”expert” judge, comparing the generated code's behavior and intent with the ground truth to determine its semantic accuracy. This dual-layered evaluation ensures a robust and comprehensive assessment of the LLM's ability to produce deployable KRL programs. The exact prompt engineered for this evaluation process is provided in Appendix A.

4. Experiments

To comprehensively assess the efficacy of our proposed dual-agent framework (MIKI), we designed two complementary sets of experiments. The first experiment isolates the code generation component and evaluates its ability to produce syntactically correct and semantically faithful KRL code using our newly introduced KRL-bench benchmark. The second experiment integrates the full pipeline – from natural language instruction to physical execution – on a real KUKA robotic arm, testing the system's performance on five representative desktop manipulation tasks of increasing complexity. The following subsections detail the experimental setup, the evaluation protocols, and the obtained results, accompanied by in-depth analysis and discussion.

4.1. Experimental setup

Robotic hardware. All physical experiments were conducted on a KUKA KR 6 R900-2 industrial manipulator. The robot is controlled by a KUKA KRC4 controller, which runs the proprietary KRL environment. The controller is connected to our workstation via Ethernet, allowing for seamless deployment of generated KRL scripts. The robot is equipped with a two-finger parallel gripper (stroke: 0–28 mm) with custom 3D-printed fingertips, providing a maximum opening of 28 mm. Since the wooden cubes used in the experiments have a side length of 20 mm, the clearance between the gripper fingers and the cube is only 4 mm per side when grasping. This tight tolerance imposes strict precision requirements on both the computed grasp coordinates and the executed motion.

Perception system. An Intel RealSense D435 RGB-D camera is mounted in an eye-to-hand configuration, approximately 0.5 m above the workspace, pointing vertically downward. This placement ensures a clear, top-down view of the entire 50 cm $\times$ 80 cm tabletop area. The camera provides synchronized

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

Table II. *Overall success rates of KRL code generation (syntactic + semantic validation).*

Model backbone	Baseline (vanilla)	Ours (closed-loop)
Qwen-Plus	0.42	0.84
DeepSeek-V3.2	0.61	**0.89**
Qwen3-8b	0.22	0.75
Qwen3-14b	0.35	0.78
DeepSeek-R1	0.51	0.84
MiniMax-M2.5	0.51	0.87
Qwen3-1.7b	0.03	0.33

1,280 × 720 RGB images and aligned depth streams. The depth stream is used to obtain metric 3D information for each pixel. Prior to the experiments, the camera was calibrated extrinsically to the robot base frame using a standard hand–eye calibration procedure with a checkerboard pattern.

Workspace and objects. The workspace is a flat table of dimensions 50 cm × 80 cm, covered with a non-reflective mat to facilitate segmentation. Four wooden cubes (side length 2 cm) are used as target objects. Their size information is given in the task instruction. Each cube is painted a distinct, solid color: red, blue, yellow, and purple. In each experimental trial, the cubes are placed in random configurations to test the system's adaptability to varying object layouts. The configurations are generated by randomly perturbing the positions of the cubes within the table region, ensuring that cubes do not initially touch or occlude each other. The orientation of the cubes is always upright (faces parallel to the table), as they are simple cuboids.

4.2. Experiment I: Code generation robustness (syntax + semantics)

In this experiment, we evaluated the code generation agent's ($\mathcal{A}_{krl}$) ability to produce compilable KRL code. A generated code snippet is considered successful only if it both passes the ANTLR4 syntax checker and is judged by the reviewer LLM to be semantically equivalent to the ground truth reference. To achieve this, we utilized the KRL-bench benchmark suite, introduced in Section III-E, which comprises 63 diverse natural language instructions. These instructions range from simple motions (e.g., "Move to home") to complex logic involving conditional branching (e.g., "If input 1 is high, move to point A").

To mitigate this variance and ensure reproducibility, we configured the generation parameters with a temperature of 0.1 and a top-p of 0.9 across all code generation experiments. Furthermore, to verify the stability of the models, the success rates reported in Table II are averaged over 3 independent runs for each test case. We compared several LLMs as backbones for code generation, ranging from small models (1.7B parameters) to state-of-the-art large models. For each model, we tested two distinct configurations:

1. **Baseline (vanilla):** Direct code generation from the prompt without any external feedback or domain-specific corpus.
2. **Ours (closed-loop):** The proposed framework integrating an ANTLR4-based parser for iterative error feedback (up to $k_{max} = 3$) and a specialized KRL knowledge corpus.

For each test case, the generated code is evaluated in two stages. (1). Syntactic validation: The code is parsed by the ANTLR4 parser using the complete KRL grammar. If the parser reports any syntax errors, the test case is immediately marked as failed (syntactic failure). No further semantic evaluation is performed. (2). Semantic validation: If the code is syntactically valid, it is passed to a reviewer LLM (a separate instance of Qwen3) that compares the generated code against the ground-truth solution. The reviewer checks for functional equivalence, i.e., whether the code would perform the intended operations

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

Table III. *Ablation study results: Success rates (syntactic + semantic) for different component combinations.*

Configuration	DeepSeek-V3.2	Qwen-Plus
Baseline (no compiler, no corpus)	0.60	0.42
compiler only	0.63	0.40
corpus only	0.82	0.58
compiler & corpus (full)	0.88	0.84

when executed on a real robot. The reviewer outputs a binary judgment (pass/fail) and, in case of failure, a brief explanation. Only test cases that pass both syntactic and semantic checks are considered successful.

Results and analysis: As evidenced in Table II, the proposed closed-loop framework yields a substantial and consistent improvement in success rates across all evaluated LLM backbones. In the baseline "Vanilla" configuration, models exhibited a high propensity for syntax errors, frequently generating hallucinated keywords or omitting mandatory definitions for complex KRL structures. The integration of the iterative refinement loop effectively mitigated these deficiencies. The top-performing model, DeepSeek-V3.2, saw its success rate rise from 0.61 to 0.89, validating the effectiveness of the parser-guided feedback. Furthermore, the framework demonstrated significant robustness even with less capable models; for instance, Qwen3-8b achieved a substantial gain, jumping from 0.22 to 0.75.

4.2.1. Ablation studies

To systematically evaluate the contribution of each key component in our proposed framework – namely the ANTLR4-based syntax-guided refinement loop (denoted as "compiler") and the specialized KRL knowledge corpus (denoted as "corpus") – we conducted an ablation study on two representative LLM backbones: DeepSeek-V3.2 (the best-performing model) and Qwen-Plus (a moderately performing model). For each backbone, we tested two more configurations:

1. Compiler only: The syntax-guided iterative refinement loop is enabled, but the knowledge corpus is not provided.
2. Corpus only: The knowledge corpus is injected into the prompt, but no iterative refinement is performed.

All other experimental settings (e.g., prompts, evaluation protocol) remained identical to those in Section 4.2. The results are summarized in Table III. The ablation results reveal several important insights:

Effect of the knowledge corpus alone: Adding the KRL knowledge corpus without any compiler feedback yields a substantial improvement for both models: DeepSeek rises from 0.60 to 0.82, and Qwen from 0.42 to 0.58. This indicates that the corpus effectively supplies essential syntactic patterns, common code templates, and best-practice examples, significantly reducing the models' uncertainty about KRL's strict syntax. The gain is particularly pronounced for Qwen, whose baseline performance is lower, suggesting that weaker models benefit more from explicit domain guidance.

Effect of the compiler alone: Enabling only the syntax-guided refinement loop (without the corpus) produces a marginal gain for DeepSeek (0.60 → 0.63) but a slight degradation for Qwen (0.42 → 0.40). This mixed result can be explained by the nature of the feedback: the compiler reports errors in a structured format, but without a knowledge base, the model may struggle to correct those errors effectively, especially if its internal representation of KRL syntax is weak. In some cases, the feedback might even lead the model to over-correct or introduce new errors. Thus, the compiler alone is insufficient; it needs to be paired with adequate domain knowledge.

Synergistic effect of both components: When both the corpus and the compiler are used together, both models achieve their highest success rates (0.88 for DeepSeek and 0.84 for Qwen). The combination

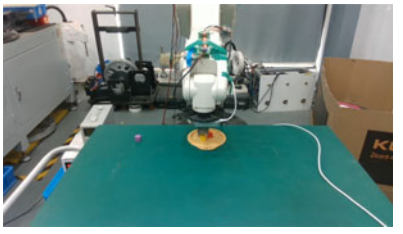
https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

Table IV. *Success rates on physical manipulation tasks.*

Task	Description	Success rate
Move block	Select a specific target and move it to a destination.	0.9
Classify	Identify objects by category and separate them.	0.7
Arrange	Pick and place objects into a specific layout.	0.6
Swap	Exchange the positions of two objects.	0.4
Stack	Place one block vertically on top of another.	0.4



(a) Intermediate execution steps (Grasp, Retract, Move, Descend)



(b) Final State

Figure 4. *Step-by-step execution of instruction: "Put all the blocks except the purple ones into the plate" in the "Arrange" task. The sequence (a) shows the trajectory execution, while (b) demonstrates the high-precision placement verified by the physical environment.*

outperforms the sum of individual gains, indicating a strong synergy: the corpus provides the necessary syntactic knowledge, and the compiler acts as a verifier that catches and helps fix the remaining errors. For Qwen, the leap from 0.58 (corpus only) to 0.84 (full system) is especially dramatic, demonstrating that even a model with moderate baseline capability can approach state-of-the-art performance when guided by both resources.

Model capacity considerations: DeepSeek, with its stronger inherent code-generation ability, already achieves 0.60 in the baseline and 0.82 with just the corpus; adding the compiler pushes it to 0.88. Qwen, starting from a lower baseline (0.42), relies more heavily on the corpus (0.58) and benefits enormously from the compiler's iterative corrections (0.84). This suggests that the closed-loop framework can help bridge the gap between weaker and stronger models, making the overall system more robust across different LLM backbones.

In summary, the ablation study confirms that both the KRL knowledge corpus and the syntax-guided refinement loop are crucial for achieving high-quality KRL code generation. The corpus provides the foundational syntactic knowledge, while the compiler ensures that the generated code adheres strictly to the language grammar. Their combination yields the best results, validating the design choices of our framework.

4.3. Experiment II: Real-world manipulation tasks

To validate the full pipeline from visual perception to physical execution, we set up a tabletop environment containing various colored blocks. We defined five distinct tasks of increasing complexity. Each task was attempted 10 times. Figure 4 illustrates the physical execution sequence of the 'Arrange' task as a representative example, where the robot sequentially executes approach, grasp, transfer, and retract motions, ultimately achieving the target placement state.

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

Table V. *Distribution of failure causes across five manipulation tasks (50 trials).*

Failure category	**Count**	**Percentage of trials**
Planning (missing or infeasible waypoints, buffer, retract)	13	26%
Grasp point miscalculation (surface vs. centroid)	6	12%
Semantic misunderstanding	1	2%
Total Failures	20	40%

The success rates are summarized in Table IV. A detailed analysis of the task execution is as follows:

Move block (success rate: 0.9). This task is the simplest and serves as a sanity check. The high success rate (9/10) demonstrates that the perception toolchain reliably extracts 3D coordinates under normal conditions. The single failure occurred when a blue cube was misclassified by the VLM as "cyan," which is a color not present in the scene, causing the system to conclude the target did not exist. This highlights limitations in fine-grained color discrimination under varying lighting.

Classify (success rate: 0.7). This task requires the robot to pick cubes one by one and place them in color-designated areas. The 70% success rate indicates that CLIP-based semantic retrieval performs effectively in most cases. Three failures were observed: two due to incorrect grasp point estimation (the agent failed to transform surface coordinates into the true geometric center), and one due to a missing retract point, causing collision with previously placed cubes during withdrawal. These failures highlight that while semantic understanding is adequate, spatial reasoning – particularly grasp planning and motion sequencing – needs further refinement.

Arrange (success rate: 0.6). The system succeeded in 6 out of 10 trials placing cubes in a specific layout. Among the four failures, one was caused by a surface-to-centroid conversion error leading to inaccurate placement; two occurred because the planner omitted intermediate waypoints, resulting in collisions; and one was due to insufficient retract height, causing the gripper to knock over adjacent cubes. These results indicate that while high-level sequencing is generally correct, more robust motion planning is required in defining safe approach, transfer, and retract trajectories to ensure collision-free execution in cluttered layouts.

Swap (success rate: 0.4). Swapping two objects requires a temporary buffer location. Out of 10 trials, six failures occurred: in three cases, the controller agent failed to generate a plan with an explicit buffer step, attempting direct exchange and causing collisions; in the other three, the agent defined a temporary location but placed it in midair rather than on a stable surface, causing the cube to drop. These failures highlight the difficulty of spatial reasoning for multi-step manipulation and the need for robust constraint checking to ensure physically feasible intermediate placements.

Stack (success rate: 0.4). Precise stacking succeeded in 4 out of 10 attempts. Among the six failures: one was caused by the omission of a retract point, leading to collision; three stemmed from incorrect grasp height calculations (grasping at the top surface instead of the centroid); and two occurred when retract height was not adjusted for increased stack height, causing collision with the tower. Even when grasping succeeded, placement accuracy remained challenging due to tight tolerances. These results underscore the need for precise centroid grounding and adaptive motion planning that accounts for changing scene geometry throughout multi-step tasks.

Across all tasks (50 trials total), the distribution of failure causes is summarized in Table V:

The table shows that planning errors dominate, accounting for 26% of trials. This suggests that while the perception and semantic components are reasonably robust (only 2% of failures attributable to perception and semantics), the translation from a high-level plan to a low-level motion sequence still requires refinement. In particular, the planner often omits intermediate waypoints that are necessary for collision avoidance in cluttered environments, and the open-loop execution (no visual feedback during motion) is vulnerable to small calibration errors.

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

Comparison with end-to-end baselines. Our framework presents two significant advantages compared to existing LLM-based robot systems. Notably, our approach uniquely targets proprietary industrial languages like KRL, differing significantly from methods that primarily output generic Python or rely on predefined skill libraries. This dual-agent architecture facilitates easier extension to other robot programming languages, enhancing its applicability across diverse industrial platforms. Furthermore, unlike systems that depend on high-level motion primitives, MIKI generates low-level motion code directly, offering greater flexibility and adaptability for complex tasks within industrial scenarios. The achieved 60% average success rate across five tasks is encouraging, particularly given the KUKA robot's stringent precision demands and the 2 cm cubes. This underscores the decoupled architecture's ability to effectively bridge semantic reasoning with deterministic industrial control.

4.4. Discussion

The experimental findings offer crucial insights into the MIKI framework's strengths and limitations. A key design strength lies in the deliberate separation of responsibilities between a controller agent for reasoning and grounding and a code generation agent for syntax. This decoupling allows the controller agent to concentrate on high-level planning and visual toolchain interaction, unburdened by low-level syntax details. Conversely, the code generation agent can leverage a specialized grammar and iterative refinement to generate error-free code, even with limited underlying LLM knowledge of KRL. This modularity also simplifies debugging and future enhancements, such as substituting the vision toolchain with a more advanced model like Grounding DINO, without impacting the code generation module. The vision toolchain, integrating SAM and CLIP, facilitates open-vocabulary object detection and precise metric coordinate extraction. The 0.9 success rate on the Move Block task validates its reliability in grounding object names in physical space. However, instances of failure in multi-step tasks due to omitted intermediate waypoints, infeasible buffer placements, and grasp height miscalculations highlight the current framework's limitations in translating high-level plans into collision-free motion sequences. Future enhancements could incorporate adaptive motion planning and closed-loop visual servoing to address these challenges.

Furthermore, syntactic verification via ANTLR4 dramatically improves code correctness, boosting the success rate from 0.61 to 0.89 for the optimal model. This is particularly crucial for industrial languages like KRL, where a single syntax error can halt production. The parser-based feedback is language-agnostic and can be adapted to other robot languages such as ABB RAPID or Universal Robots URScript with minimal modifications. Despite these strengths, tasks requiring precise multi-step coordination, like Swap and Stack, remain challenging. The core issue is not perception or syntax, but the absence of reactive planning during execution. The current framework generates a complete KRL program offline and executes it open-loop, leading to the accumulation of small errors and an inability to handle unexpected events, such as slightly displaced cubes. Our findings align with current trends in LLM-based robotics, demonstrating enhanced performance through combined reasoning, visual grounding, and iterative refinement. This work also uniquely targets proprietary industrial languages and provides syntactic guarantees through parser feedback. The 0.89 syntactic success rate stands as, to our knowledge, the first quantitative demonstration of reliable KRL code generation from natural language.

5. Conclusion

In this work, we introduced **MIKI (multi-agent integrated KUKA interface)**, a novel decoupled dual-agent framework that bridges the gap between high-level natural language understanding and low-level industrial robot programming. The framework tackles two fundamental challenges that have hindered the direct application of large language models in industrial settings: reliance on predefined motion primitives, which limits the system's adaptability to unstructured environments, and syntactic fragility, which causes frequent errors when generating code in proprietary languages like KUKA Robot Language.

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

The core innovation lies in the separation of concerns between a **controller agent** responsible for reasoning and visual grounding and a **code generation agent** dedicated to syntax-compliant code synthesis. The controller agent leverages a chain-of-thought mechanism to decompose complex instructions into structured sub-tasks and interacts with a modular vision toolchain – combining SAM, CLIP, and an eye-to-hand calibrated depth camera – to extract precise 3D world coordinates of objects. The code generation agent then translates these grounded plans into KRL code through an iterative generate-verify-refine loop powered by an ANTLR4-based parser. A specialized KRL knowledge corpus further enriches the prompts, providing domain-specific patterns and best practices. This closed-loop mechanism dramatically improves syntactic correctness while preserving semantic fidelity.

To rigorously evaluate our approach, we developed **KRL-bench**, the first dedicated benchmark for KRL code generation, comprising 63 test cases that span variable declarations, expressions, control flow, functions, and motion commands. Extensive experiments on both the benchmark and a real KUKA KR6 R900-2 manipulator demonstrate the effectiveness of MIKI. On KRL-bench, our best configuration (DeepSeek-V3.2 with full compiler and corpus) raises the syntactic-plus-semantic success rate from 0.61 (baseline) to 0.89 – a relative improvement of 46%. Ablation studies confirm that both the knowledge corpus and the compiler feedback are essential: the corpus alone lifts performance to 0.82, while the compiler alone yields marginal gains (0.63), but their combination achieves the highest results (0.88 for DeepSeek, 0.84 for Qwen-Plus). This synergy highlights the importance of pairing domain knowledge with automatic verification.

In physical experiments, MIKI successfully executed five representative manipulation tasks with an average success rate of 60%, despite the challenging precision requirements (2 cm cubes and 4 mm gripper clearance). The system reliably handled simple pick-and-place (90% success) and semantic classification (70%) and demonstrated initial capability in complex multi-step tasks such as swapping and stacking (40% each). These results validate that the decoupled architecture can translate abstract instructions into physically executable actions, while the syntax-guided refinement ensures that the generated KRL code is ready for deployment on industrial controllers.

Nevertheless, several limitations remain. The vision pipeline, though generally robust, occasionally suffers from misclassifications under challenging lighting (accounting for 2% of failures), suggesting that integrating temporal information or 3D scene graphs could further enhance its reliability. More critically, planning deficiencies – responsible for 26% of failures – remain the dominant bottleneck: the planner frequently omits intermediate waypoints necessary for collision avoidance or fails to define feasible buffer locations in multi-step tasks. These issues underscore the need for more sophisticated spatial reasoning, adaptive motion planning, and closed-loop visual servoing during execution. Future work will explore online replanning using real-time sensor feedback as well as extend MIKI to other industrial robot languages (e.g., ABB RAPID and Universal Robots URScript) and more complex scenarios involving tool use, assembly, or human–robot collaboration.

In conclusion, MIKI demonstrates that with careful architectural design, which separates reasoning from syntax, grounds language in vision, and enforces grammatical correctness through iterative feedback, large language models can be effectively harnessed for industrial robot programming. The framework provides a blueprint for integrating LLMs into other safety-critical, syntax-strict domains.

Supplementary material. The supplementary material for this article can be found at https://doi.org/10.1017/S0263574726103646.

Author contributions. Zhendao Chen and Haibo Chao conceived and designed the study. Yanhao He and Che Hou conducted data gathering. Wenjie Chen and Ning Tan secured the funding and provided research resources. Zhendao Chen and Haibo Chao wrote the article. Zhendao Chen, Haibo Chao, and Ning Tan reviewed and revised the article.

Financial support. This work is supported in part by the State Key Laboratory of High-end Heavy-load Robots Open Project (Grant No. HHR2025020205), in part by the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2025A1515050002), and in part by the Open Fund for State Key Laboratory of Intelligent Manufacturing Equipment and Technology (Grant No. IMETKF2026011).

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

Competing interests. The authors declare no conflicts of interest exist.

Ethical approval. Not applicable.

AI tools. The AI tool DeepSeek-V3.2 is used for refining the language and grammar. The authors have reviewed the results and take full responsibility for the contents of the paper.

References

[1] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, A. Herzog, D. Ho, J. Hsu, J. Ibarz, B. Ichter, A. Irpan, E. Jang, R. J. Ruano, K. Jeffrey, S. Jesmonth, N. J. Joshi, R. Julian, D. Kalashnikov, Y. Kuang, K.-H. Lee, S. Levine, Y. Lu, L. Luu, C. Parada, P. Pastor, J. Quiambao, K. Rao, J. Rettinghouse, D. Reyes, P. Sermanet, N. Sievers, C. Tan, A. Toshev, V. Vanhoucke, F. Xia, T. Xiao, P. Xu, S. Xu, M. Yan and A. Zeng, "Do As I Can, Not As I Say: Grounding Language in Robotic Affordances," **In:** *Proc. Conf. Robot Learn. (CoRL) 2023*, vol. **205** (2023) pp. 287–318.

[2] V. Bhat, A. U. Kaypak, P. Krishnamurthy, R. Karri and F. Khorrami, "Grounding large language models for robot task planning using closed-loop state feedback," *Adv. Robot. Res.* e202500072 (2025).

[3] I. M. Chekam, I. Pastor-Martinez, A. Tourani, J. A. Millan-Romera, L. Ribeiro, P. M. B. Soares, H. Voos and J. L. Sanchez-Lopez, "Interpretable robot control via structured behavior trees and large language models," *IEEE Access* **13**, 200905–200916 (2025).

[4] M. Deitke, C. Clark, S. Lee, R. Tripathi, Y. Yang, J. S. Park, M. Salehi, N. Muennighoff, K. Lo, L. Soldaini, J. Lu, T. Anderson, E. Bransom, K. Ehsani, H. Ngo, Y. Chen, A. Patel, M. Yatskar, C. Callison-Burch, A. Head, R. Hendrix, F. Bastani, E. Vanderbilt, N. Lambert, Y. Chou, A. Chheda, J. Sparks, S. Skjonsberg, M. Schmitz, A. Sarnat, B. Bischoff, P. Walsh, C. Newell, P. Wolters, T. Gupta, K.-H. Zeng, J. Borchardt, D. Groeneveld, C. Nam, S. Lebrecht, C. Wittlif, C. Schoenick, O. Michel, R. Krishna, L. Weihs, N. A. Smith, H. Hajishirzi, R. Girshick, A. Farhadi and A. Kembhavi, "Molmo and PixMo: Open Weights and Open Data for State-of-the-Art Vision-Language Models," **In:** *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR) 2025* (2025) pp. 91–104.

[5] W. Feng, W. Zhu, T.-J. Fu, V. Jampani, A. Akula, X. He, S. Basu, X. E. Wang and W. Y. Wang, "LayoutGPT: Compositional Visual Planning and Generation with Large Language Models," **In:** *Adv. Neural Inf. Process. Syst. (NeurIPS) 2024*, vol. **36** (2024).

[6] W. Huang, P. Abbeel, D. Pathak and I. Mordatch, "Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents," **In:** *Proc. Int. Conf. Mach. Learn. (ICML) 2022*, vol. **162** (2022) pp. 9118–9147.

[7] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu and L. Fei-Fei. "VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models," **In:** *Proc. Conf. Robot Learn. (CoRL) 2023a*, vol. **229** (2023a).

[8] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, N. Brown, T. Jackson, L. Luu, S. Levine, K. Hausman and B. Ichter, "Inner Monologue: Embodied Reasoning Through Planning with Language Models," **In:** *Proc. Conf. Robot Learn. (CoRL)*, vol. **205b** (2023b).1769–1782.

[9] B. Ke, A. Obukhov, S. Huang, N. Metzger, R. C. Daudt and K. Schindler, "Repurposing Diffusion-Based Image Generators for Monocular Depth Estimation," **In:** *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR) 2024* (2024) pp. 9492–9502.

[10] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár and R. Girshick, "Segment Anything," **In:** *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV) 2023* (2023) pp. 3992–4003.

[11] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence and A. Zeng, "Code as Policies: Language Model Programs for Embodied Control," **In:** *Proc. IEEE Int. Conf. Robot. Autom. (ICRA) 2023* (2023) pp. 9493–9500.

[12] L. Ling, C.-H. Lin, T.-Y. Lin, Y. Ding, Y. Zeng, Y. Sheng, Y. Ge, M.-Y. Liu, A. Bera and M. Li, "Scenethesis: A Language and Vision Agentic Framework for 3D Scene Generation," **In:** *Int. Conf. Learn. Represent. (ICLR) 2026* (2026).

[13] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas and P. Stone, "LLM+P: Empowering large language models with optimal planning proficiency," arXiv: 2304.11477 (2023).

[14] S. Liu, Z. Zeng, T. Ren, F. Li, H. Zhang, J. Yang, Q. Jiang, C. Li, J. Yang, H. Su, J. Zhu and L. Zhang. "Grounding Dino: Marrying Dino with Grounded Pre-Training for Open-Set Object Detection," **In:** *Proceedings of the 18th European Conference on Computer Vision (ECCV) 2025*, vol. **15105** of Lecture Notes in Computer Science (Springer, Milan, Italy, 2025) pp. 38–55.

[15] T. Parr, *The Definitive ANTLR 4 Reference*, 2 edition (The Pragmatic Bookshelf, Dallas, Texas, USA, 2014).

[16] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger and I. Sutskever, "Learning Transferable Visual Models from Natural Language Supervision," **In:** *Proc. Int. Conf. Mach. Learn. (ICML) 2021*, vol. **139** (2021) pp. 8748–8763.

[17] K. Rana, J. Haviland, S. Garg, J. Abou-Chakra, I. Reid and N. Sünderhauf, "SayPlan: Grounding Large Language Models using 3D Scene Graphs for Scalable Robot Task Planning," **In:** *Proc. Conf. Robot Learn. (CoRL) 2023*, vol. **229** (2023).

[18] A. Z. Ren, A. Dixit, A. Bodrova, S. Singh, S. Tu, N. Brown, P. Xu, L. Takayama, F. Xia, J. Varley, Z. Xu, D. Sadigh, A. Zeng and A. Majumdar, "Robots That Ask for Help: Uncertainty Alignment for Large Language Model Planners," **In:** *Proc. Conf. Robot Learn. (CoRL) 2023*, vol. **229** (2023).

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

[19] R. Royce, M. Kaufmann, J. Becktor, S. Moon, K. C. Carpenter, K. Pak, A. Towler, R. Thakker and S. Khattak, "Enabling Novel Mission Operations and Interactions with ROSA: The Robot Operating System Agent," **In:** *IEEE Aerospace Conference 2024* (2024) pp. 1–16.
[20] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason and A. Garg, "ProgPrompt: Generating Situated Robot Task Plans Using Large Language Models," **In:** *Proc. IEEE Int. Conf. Robot. Autom. (ICRA) 2023* (2023) pp.11523–11530.
[21] F.-Y. Sun, W. Liu, S. Gu, D. Lim, G. Bhat, F. Tombari, M. Li, N. Haber and J. Wu, "LayoutVLM: Differentiable Optimization of 3D Layout via Vision-language Models," **In:** *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR) 2025* (2025) pp. 29469–29478.
[22] F.-Y. Sun, S. Wu, C. Jacobsen, T. Yim, H. Zou, A. Zook, S. Li, Y.-H. Chou, E. F. Can, X. Wu, C. Eppner, V. Blukis, J. Tremblay, J. Wu, S. Birchfield and N. Haber, "3D-GENERALIST: Vision-Language-Action Models for Crafting 3D Worlds," **In:** *Int. Conf. 3D Vis. (3DV) 2026* (2026).
[23] C. Wang, S. Hasler, D. Tanneberg, F. Ocker, F. Joublin, A. Ceravola, J. Deigmoeller and M. Gienger, "LaMI: Large Language Models For Multi-modal Human–robot Interaction," **In:** *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems* (2024a).
[24] J. Wang, E. Shi, H. Hu, C. Ma, Y. Liu, X. Wang, Y. Yao, X. Liu, B. Ge and S. Zhang, "Large language models for robotics: Opportunities, challenges, and perspectives," *J. Autom. Intell.* **4**(1), 52–64 (2025).
[25] Y. Wang, X. Qiu, J. Liu, Z. Chen, J. Cai, Y. Wang, T.-H. Wang, Z. Xian and C. Gan, "ARCHITECT: Generating Vivid and Interactive 3D Scenes with Hierarchical 2D Inpainting," **In:** *Adv. Neural Inf. Process. Syst. (NeurIPS) 2024b*, vol. **37** (2024b).
[26] J. Xu, Q. Sun, Q.-L. Han and Y. Tang, "When embodied AI meets industry 5.0: Human-centered smart manufacturing," *IEEE/CAA J. Autom. Sin.* **12**(3), 485–501 (2025).
[27] Y. Yang, F.-Y. Sun, L. Weihs, E. Vanderbilt, A. Herrasti, W. Han, J. Wu, N. Haber, R. Krishna, L. Liu, C. Callison-Burch, M. Yatskar, A. Kembhavi and C. Clark, "Holodeck: Language Guided Generation of 3D Embodied AI Environments," **In:** *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR) 2024* (2024) pp. 16227–16237.
[28] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," **In:** *Proc. Int. Conf. Learn. Represent. (ICLR) 2023* (2023).
[29] P. Zhi, Z. Zhang, Y. Zhao, M. Han, Z. Zhang, Z. Li, Z. Jiao, B. Jia and S. Huang, "Closed-Loop Open-Vocabulary Mobile Manipulation with GPT-4V," **In:** Proc. IEEE Int. Conf. Robot. Autom. (ICRA) 2025 (2025) pp. 4761–4767.

Appendix A. Evaluation prompt for KRL-bench

The following prompt is used by the reviewer LLM to assess the semantic correctness of generated KRL code against the ground truth solution in the KRL-bench benchmark.

```
You are an expert in KUKA Robot Language (KRL); you are responsible for
evaluating the KRL code of a set of benchmarks. You will be given a task
instruction, an expected solution KRL code, and an output KRL code written by
someone to be evaluated. Your task is to evaluate the output KRL code against
the solution code and determine if it is correct.
You should follow these steps:
1. Read the task instruction and the expected solution KRL code.
2. Read the output KRL code.
3. Check if the output KRL code is syntactically correct. You should compare
the syntax of function declarations, variable declarations, and other KRL
constructs with the expected solution. If it is not syntactically correct,
return "No" and explain why it is not correct.
4. Compare the output KRL code with the expected solution KRL code. You
must analyze the solution code and understand its logic, what system, local,
global variables it declares or uses. You should check if the output KRL code
achieves the same functionality as the expected solution KRL code.
5. If the output code meets the requirements of the question, you should
classify it as correct, even if it differs from the reference answer.
6. If the output KRL code is correct, return just "Result: Correct." If its
logic is incorrect, return "Result: Wrong" and explain why.
```

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press

```
7. It is ok if output code uses "ptp home" at the beginning and the end while
the expected solution does not use it, because this command will improve
safety.
Let's start with the task.
Task: {p_name}
Instruction: {p_inst}
Expected Solution:
{p_sol}
Output Code:
{p_output}
Your evaluation:
```

Cite this article: Z. Chen, H. Chao, Y. He, C. Hou, W. Chen and N. Tan (2026). "A dual-agent framework for physically grounded and syntactically verifiable industrial robot programming", Robotica **44**, 1516–1533. https://doi.org/10.1017/S0263574726103646

https://doi.org/10.1017/S0263574726103646 Published online by Cambridge University Press