EAI Endorsed Transactions
on AI and Robotics

Research Article EAI.EU

Empowering Universal Robot Programming with Fine-Tuned Large Language Models

Tien Dat Le[1], Minhhuy Le[1,*]

[1]Intelligent Communication System Laboratory (ICSLab), Phenikaa University, Hanoi 12116, Vietnam

Abstract

LLMs are transforming AI but face challenges in robotics due to domain-specific requirements. This paper explores LLM-generated URScript code for Universal Robots (UR), improving automation accessibility. A fine-tuning dataset of 20,000 synthetic samples, based on 514 validated human-created examples, enhances performance. Using the Unsloth framework, we fine-tune and evaluate the model in real-world scenarios. Results demonstrate LLMs' potential to simplify UR robot programming, highlighting their value in industrial automation. The video demo is available at the following link, and the codebase will be added soon: https://github.com/t1end4t/llm-robotics

Received on 28 March 2025; accepted on 17 June 2025; published on 15 July 2025

Keywords: Large Language Models (LLMs), Fine-tuning, Synthesis dataset, URScript

doi:10.4108/airo.8983

1. Introduction

Automation and robotics are increasingly becoming integral components of modern industrial systems. Current advancements in this field are achieving remarkable outcomes, largely driven by the emergence of artificial intelligence (AI) and machine learning (ML) technologies Universal Robots (UR), with their versatile and collaborative robot arms, have emerged as a leader in this field. In 2024, Universal Robots unveiled its AI Accelerator, a comprehensive hardware and software toolkit aimed at advancing artificial intelligence (AI) applications in collaborative robots (cobots) [1]. By enhancing automation processes and expanding the scope of cobots' capabilities, the AI Accelerator demonstrates how AI can revolutionize industries ranging from manufacturing to healthcare. However, this rapid integration of AI into robotics presents both significant benefits and challenges, particularly concerning safety, reliability, and system stability. Ensuring that cobots maintain precision and adaptability without compromising human safety remains a critical focus in this evolving landscape.

Among the most promising innovations are generative models that can generate high-quality, human-like text and code such as Large Language Models, Multimodal AI, etc. Large Language Models (LLMs) have emerged as transformative tools in robotics, enabling advanced capabilities in natural language understanding, decisionmaking, and control. Research in this field has focused on integrating LLMs to enhance robotic performance across various dimensions, from task execution to multi-modal interaction.

The ability of LLMs to ground language in robotic affordances has been explored in "Do as I Can, Not as I Say" [2] by Chebotar et al., where natural language instructions are aligned with a robot's physical capabilities for high-level decision-making. Similarly, Huang et al. in "Voxposer" [9] demonstrate the use of LLMs to synthesize dense robotic trajectories and manipulate 3D value maps, showcasing their utility in spatial reasoning and task execution.

LLMs also play a critical role in state tracking and dynamic decision-making, as highlighted by Yoneda et al. in "Statler" [4], where state-maintaining LLMs enable consistent planning in dynamic environments. Furthermore, Liang et al. in "Code as Policies" [5] propose converting natural language inputs into executable robot control policies, allowing robots to perform complex tasks with minimal human intervention.

*Corresponding author. Email: huy.leminh@phenikaa-uni.edu.vn

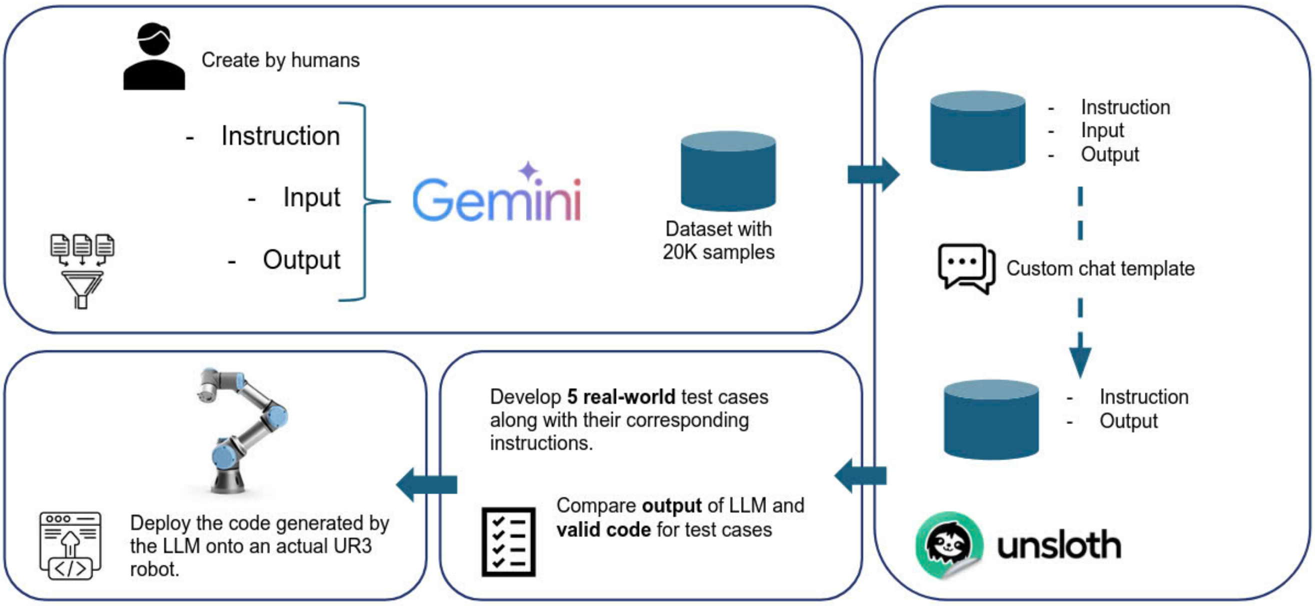


Figure 1. Methodological Workflow

In terms of adaptability, Mirchandani et al., in "Large Language Models as General Pattern Machines" [6], emphasize the ability of LLMs to extrapolate sequences and integrate large-scale robotic datasets, making them versatile tools for automation. Similarly, Ceravola et al., in "CoPAL", [7] introduce frameworks for real-time corrective planning of robot actions, ensuring robustness in unpredictable scenarios.

Multi-modal capabilities are another area of innovation. Driess et al., in "Palm-e", [8] detail the implementation of embodied multi-modal LLMs that enable perception and action, bridging the gap between sensory input and motor responses. Additionally, Huang et al. in "Instruct2Act" [9] explore the mapping of multi-modal human instructions to robotic actions, further advancing human-robot interaction. Scalability and collaboration are addressed in "Scalable Multi-Robot Collaboration with Large Language Models" [10] by Chen et al., where LLMs facilitate centralized and decentralized strategies for multi-robot systems. Finally, Firoozi et al., in "Foundation Models in Robotics", [11] provide a comprehensive overview of LLMs' foundational role in advancing robotic autonomy, discussing both their potential applications and the challenges ahead.

These advancements collectively illustrate the transformative impact of LLMs in robotics, paving the way for more intelligent, adaptable, and autonomous systems capable of seamlessly integrating into diverse environments. This study focuses on applying LLM to the task of generating URscript code to control the UR robot arm. While technologies LLMs techniques, have significant potential, their limitations, especially in niche domains [12], necessitate human oversight.

To address these challenges, we propose a fine-tuned model trained on a high-quality instruction-tuning dataset and evaluate its performance across a range of specific robotic control tasks. Our study follows a structured workflow, as illustrated in Figure 1, which outlines the key stages of dataset generation, model fine-tuning, and evaluation. Our main contributions in this study are as follows:

- **Data collection**: In this study, we constructed a specialized dataset for fine-tuning our model by utilizing URSim [13], a simulation software designed for programming and testing Universal Robots. Using URSim, we developed and collected URScript-based control programs, ensuring that all dataset instances were generated directly from URScript code. Following the methodology of the Alpaca dataset [14], we created seed tasks by deriving structured instructions from these URScript snippets. To expand the dataset, we employed **Gemini** to generate additional task instances, leveraging the initial seed tasks as a foundation. This process enabled us to scale the dataset to 20,000 samples while preserving consistency and task diversity. By systematically generating both concise and detailed task descriptions, we ensure that the dataset remains robust, syntactically and logically coherent, and highly relevant to real-world robotic applications. This comprehensive dataset serves as a reliable foundation for evaluating and fine-tuning our model.

- **Fine-tuning model**: To fine-tune our model, we employed the Unsloth framework, leveraging its advanced capabilities to refine multiple Qwen2.5 models, including Qwen-Base and Qwen-Coder in both 3B and 7B configurations. Using the dataset collected via URsim, we conducted supervised fine-tuning, enabling the models to adapt to task-specific patterns and complexities inherent in our data. This approach allowed for efficient optimization of specialized coding tasks, ensuring the models could accurately interpret and execute robotic commands.

- **Testing and deploying**: To validate our fine-tuned model, we conducted testing across five carefully selected use cases, each reflecting critical aspects of real-world robotic operations. These use cases included motion planning, object manipulation, error recovery, adaptive decision-making, and multi-step task execution. The performance of the model was first evaluated in a simulated environment using URsim to ensure alignment with the dataset used during fine-tuning. Subsequently, we deployed the code generated by the fine-tuned LLM directly onto a UR3 robotic arm, enabling us to assess its ability to perform physical tasks accurately and reliably.

2. Literature review

2.1. Large language models

Large Language Models (LLMs) represent a significant milestone in artificial intelligence (AI), designed to understand and generate human-like text with remarkable accuracy. Built using advanced deep learning architectures, particularly transformerbased frameworks, these models leverage massive datasets that span diverse domains of human knowledge, including books, articles, and websites. Their capabilities have revolutionized natural language processing (NLP) tasks such as machine translation, summarization, and more sophisticated applications like code generation, creative writing, and conversational AI.

At the core of LLMs lies the transformer architecture, introduced by Vaswani et al. [15], which introduced a self-attention mechanism that efficiently models long-range dependencies in sequential data. The self-attention mechanism is mathematically expressed as:

$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Here, Q, K and V represent the query, key, and value matrices, while d_K is the dimensionality of the keys. This mechanism allows LLMs to dynamically focus on relevant parts of the input sequence, capturing context effectively across large texts.

LLMs operate on the principle of pretraining and fine-tuning. In the pretraining phase, the model is optimized to minimize a loss function over a vast corpus of unstructured data. Typically, this involves a masked language modeling objective, such as:

$$\mathcal{L}_{MLM} = -\sum_{i=1}^{N} \log P(x_i \mid x_{<i})$$

where x_i represents the current token and $x_{<i}$ denotes the sequence of preceding tokens. Fine-tuning adapts this pretrained model to specific downstream tasks, refining its parameters to optimize task-specific objectives, often using supervised learning.

The emergence of models like GPT, BERT, and GPT-3 reflects iterative advancements in both architecture and training methodologies, made possible by increased computational resources. For instance, GPT-3, a 175-billion parameter model, exemplifies the scalability of LLMs and their ability to generalize across diverse tasks without explicit retraining.

Despite their versatility, LLMs pose significant challenges. Ethical concerns include biases in training data, risks of misinformation, and the environmental costs associated with training large-scale models, which can be quantified by the computational energy required, $E \propto n_p \cdot n_y \cdot n_t$ where n_p is the number of parameters, n_y the size of the dataset, and n_p the number of training iterations. Addressing these issues is a key focus in ongoing research.

Nonetheless, LLMs remain at the forefront of AI innovation, driving transformative advancements across industries while inspiring further exploration into optimizing their efficiency, fairness, and applicability.

2.2. Qwen LLM models

The Qwen large language models (LLMs), developed by Alibaba Cloud, represent a significant advancement in the field of artificial intelligence. These transformer-based models, initially introduced with the Qwen1.8B and later scaled to the Qwen2 and Qwen2.5 series, have been optimized for various applications, including general purpose natural language understanding, multilingual processing, and domain specific tasks like coding and mathematical reasoning [16]. Qwen models leverage stateof-the-art innovations in transformer architectures, such as Rotary Positional Embeddings (RoPE) and Grouped Query Attention (GQA), to enhance performance and scalability. The Qwen2.5 series offers models with parameters ranging from 3 billion to 72 billion, enabling fine-tuning and instruction tuning for task-specific applications. Notably, the Qwen2.5-72B model achieves state of-the-art results in reasoning and multilingual understanding, comparable to leading models like GPT-4 and Llama2.

Example of URscript

```
def move_joint():
    local Waypoint_1_p= p[-0.13,  -0.26, 0.14,  -0.001, 3.11,  0.03]
    local Waypoint_1_q=  [-1.6,   -1.72, -2.2,  -0.8,   1.59,  -0.03] # radian
    local Waypoint_2_p= p[-0.008, -0.26, 0.28,  -0.27,  -2.71, 0.27]
    local Waypoint_2_q=  [-0.92,  -1.26, -1.88, -1.73,  1.16,  0.88] #radian
    while (True):
        movej(get_inverse_kin(Waypoint_1_p, qnear=Waypoint_1_q ), a=1.39, v=1.04)

        movej(get_inverse_kin(Waypoint_2_p, qnear=Waypoint_2_q), a=0.78, v=0.87)
    end
end
```

Pretraining for Qwen models involves processing trillions of tokens from diverse multilingual datasets, with particular emphasis on English and Chinese. Post-training employs techniques such as Supervised Fine-Tuning (SFT) and Reinforcement Learning with Human Feedback (RLHF) to refine model alignment and improve response quality. These methods ensure that Qwen models not only deliver accurate responses but also demonstrate creativity and adherence to user instructions. In addition to their generalpurpose capabilities, specialized variants like Qwen2.5 Coder and Qwen2.5-Math are tailored for technical domains. Qwen2.5-Coder excels in programming tasks, such as debugging and code generation, while Qwen2.5-Math enhances mathematical reasoning using advanced techniques like Chain-of-Thought and Proof-of-Theorem reasoning.[16] [17]

2.3. Fine-tuning LLM models

Fine-tuning large language models (LLMs) is a process of adapting pre-trained models to specific tasks or domains by updating their parameters using a task-specific dataset. The core idea is to adjust the model's weights W to minimize a loss function $\mathcal{L}_{task}$, calculated as:

$$\mathcal{L}_{task} = \frac{1}{N}\sum_{i=1}^{N} \ell\Big(f(x_i; W), y_i\Big)$$

where x_i represents the input data, y_i is the target output, and ℓ is a loss function, such as cross-entropy. The updated weights W' are obtained using gradient descent:

$$W' = W - \eta \nabla \mathcal{L}_{task}$$

where η is the learning rate. However, fine-tuning all model parameters can be computationally expensive, especially for large models with billions of parameters.

To address this, techniques like **Low-Rank Adaptation (LoRA)** have been developed. LoRA introduces small, trainable low-rank matrices ∇W into the model, while keeping the original weights W frozen. For a weight matrix W, the modified forward pass becomes:

$$W' = W + \eta \nabla W$$

where $\Delta W = AB$, and $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times d}$ are low-rank matrices with $r \ll d$. This significantly reduces the number of trainable parameters to $\mathcal{O}(r \cdot d)$, making fine-tuning more efficient while retaining performance. Fine-tuning with LoRA thus enables the adaptation of LLMs to specific tasks with reduced computational and memory overhead, broadening their applicability to resource-constrained settings.

2.4. URScript: Flexible Robotic Programming Language

URScript is a high-level scripting language specifically developed by Universal Robots to program and control its robotic arms. It provides functionality beyond the graphical PolyScope interface, allowing users to write customized scripts for intricate control over robot movements, logical operations, and system interactions. Through URScript, developers can integrate external equipment, implement socket communications, and control robots in real time. [18]

URScript's design makes it accessible for diverse user expertise levels, enabling its adoption in complex industrial tasks and academic experiments alike. Key features of URScript include:

- Multi-threading: Manage simultaneous tasks, crucial for real-time robotic operations.
- Conditional Logic and Loops: Facilitate advanced automation scenarios.
- Integration Capabilities: Support for external devices and communication protocols.

3. Methodology

3.1. Data collection

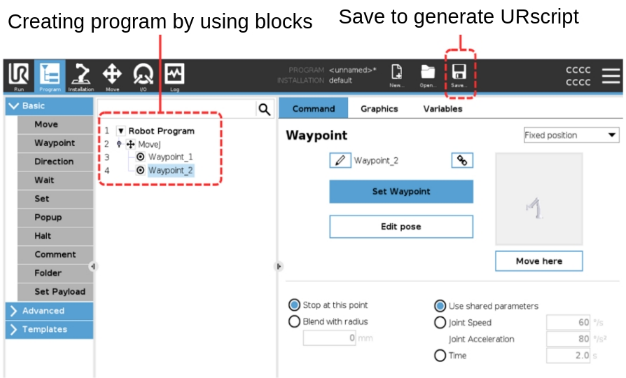


Figure 2. URSim GUI

To develop a high-quality dataset, we leverage URSim as a simulation platform to generate control programs written in URScript. This approach ensures that the produced code is syntactically correct, logically coherent, and adheres to predefined safety standards. As illustrated in Figure 2, URSim provides a graphical user interface (GUI) that allows users to create robot control programs using a block-based programming environment. The left panel of the GUI contains a set of predefined commands, such as movement, waypoints, and control logic, which can be sequentially arranged in the central programming workspace. Once a program is structured, it can be saved to generate a URScript file containing the corresponding commands in script format. After collecting URScript code snippets, we construct a set of seed tasks by deriving structured instructions from these snippets, following the methodology of the Alpaca dataset. [14] These seed tasks consist of both concise task descriptions and detailed explanations of the associated challenges, facilitating a comprehensive assessment of the model's fine-tuning effectiveness. To further expand the dataset, we employ the prompts provided in **Appendix A**, generating additional task samples while maintaining consistency and diversity. This structured approach ensures the dataset's robustness and applicability to real-world robotic control scenarios.

3.2. Fine-tune model by Unsloth

The Qwen model was chosen for this research due to its outstanding performance and extensive support for diverse applications, making it a suitable candidate for URScript code generation tasks. As a state-of-the-art language model, Qwen demonstrates strong capabilities in understanding and generating complex text structures, including programming languages, thanks to its advanced transformer architecture and large-scale pretraining. Furthermore, the availability of multiple parameter sizes, such as 3B and 7B, allows for a balance between computational efficiency and performance, enabling scalability based on resource constraints. Qwen's robust community support, frequent updates, and seamless integration with frameworks like Hugging Face Transformers ensure that it remains a versatile and reliable tool for fine-tuning in specialized domains. These factors collectively position Qwen as an optimal choice for enhancing the logical accuracy and contextual relevance required for generating URScript code. To enhance the capabilities of the Qwen-3B and Qwen-7B models for generating URScript code, we employed the Unsloth framework, a lightweight yet powerful finetuning approach designed for domain-specific applications. Unsloth facilitates efficient adaptation of large-scale language models by integrating optimized training workflows, flexible configuration options, and robust evaluation techniques. This section outlines the process of fine-tuning the Qwen models, leveraging Unsloth to maximize model performance and computational efficiency.

The decision to use the Qwen-3B and Qwen-7B models for this research was driven by the computational resources available, specifically the T4 GPU provided by Google Colab. These models strike a practical balance between performance and resource demands, making them well-suited for fine-tuning and experimentation within the constraints of a single-GPU environment. The T4 GPU, with its 16 GB of VRAM, is capable of handling the memory requirements of the 3B model efficiently and supports the 7B model with optimized batching and mixed-precision training techniques. This setup ensures that the research can be conducted without requiring expensive highperformance hardware, while still leveraging state-of-the-art models to achieve meaningful results in URScript code generation tasks.

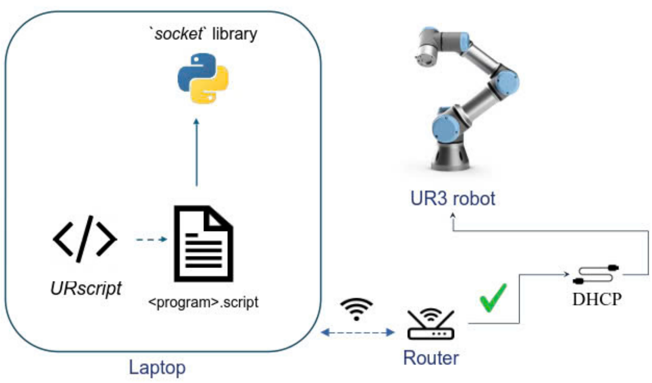
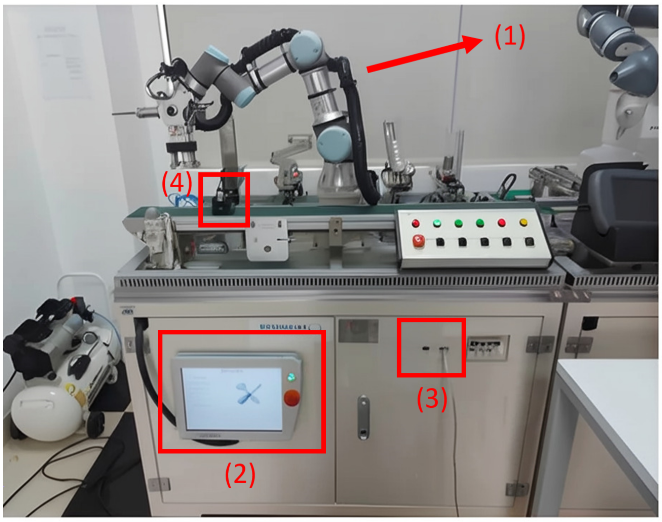
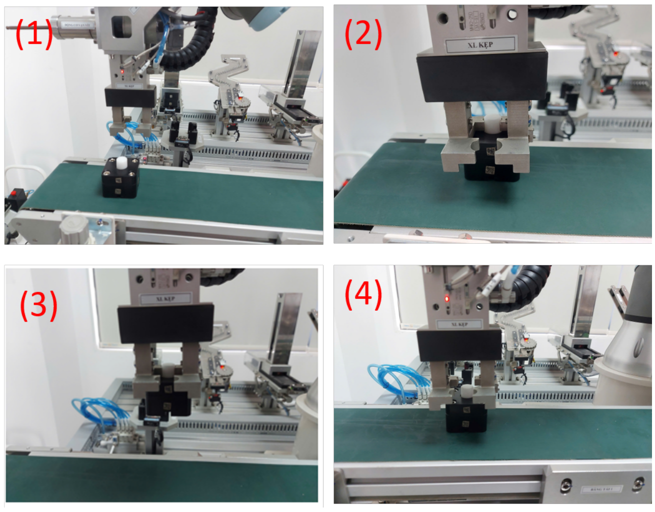


Figure 3. Deploying URscript code into UR3



(a) UR3 robotic system setup



(b) Pick-and-place operation sequence

Figure 4. Physical setup of the system

4. Experimental results

Evaluating the fine-tuning effectiveness of large language models (LLMs) in a narrow domain like controlling robots with URScript is challenging due to the specificity of the tasks. This research focuses on developing evaluation parameters for the fine tuning process and optimizing the workflows for pre-defined tasks. The goal is to enhance the adaptability and precision of LLMs in this specialized application.

4.1. Experimental setup

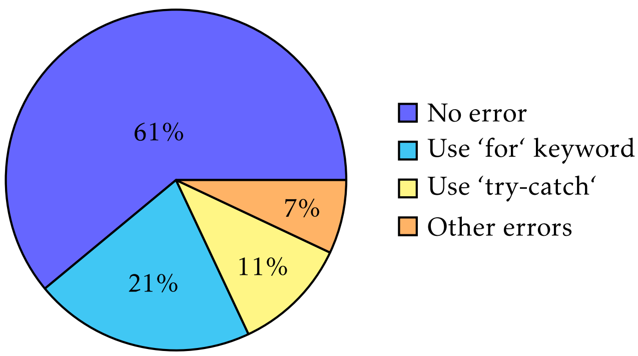


Figure 5. Error Distribution in URScript Generated by Qwen7B-Coder

The deployment of URScript code to the UR3 robotic system is facilitated through a networked communication framework, as illustrated in Figure 3. The process begins with the development of control scripts in URScript on a laptop, where a Python-based interface utilizing the socket library is employed to establish a communication channel. The URScript program is then transmitted over a local network via a router, which assigns an IP address to the UR3 robot through Dynamic Host Configuration Protocol (DHCP). Once connected, the UR3 robot receives and executes the transmitted script, enabling remote deployment and execution of control commands. This architecture ensures a flexible and efficient method for real-time programming and control of the robotic system.

Figure 4a illustrates the physical setup of the system, which consists of multiple interconnected components to facilitate automated robotic control. The primary element is the UR3 robotic arm (1), which is responsible for executing programmed tasks with high precision. The system is equipped with an HMI touchscreen interface (2), enabling real-time monitoring and user interaction for system control. Network communication is established through the Ethernet port (3), ensuring seamless data transmission between the control unit and external systems. Additionally, the system handles workpieces (4), which are manipulated during the automated process. This integrated setup enables efficient execution of URScript-based control programs, ensuring reliable operation in real-world applications.

For fine-tuning, we used the LoRA technique and 4-bit quantization model for less memory. For LoRA, we tried some parameter recommended by the original paper when set the *lora_alpha* to 16, set dropout to 0. With rank of this algorithm, we choose 16, and learning rate we set it to 2e-4, this learning rate is commonly used for f inetuning LLM and enables stable and incremental updates to the model's parameters throughout the training process. For optimizer, *admw_8bit* (AdamW optimizer that employs 8-bit precision) is selected for fine-tune, combine with weight decay is 0.01. The fine-tuning experiments were conducted on Google Colab T4 GPU with consistent settings across all datasets.

4.2. Results with high quality test-cases

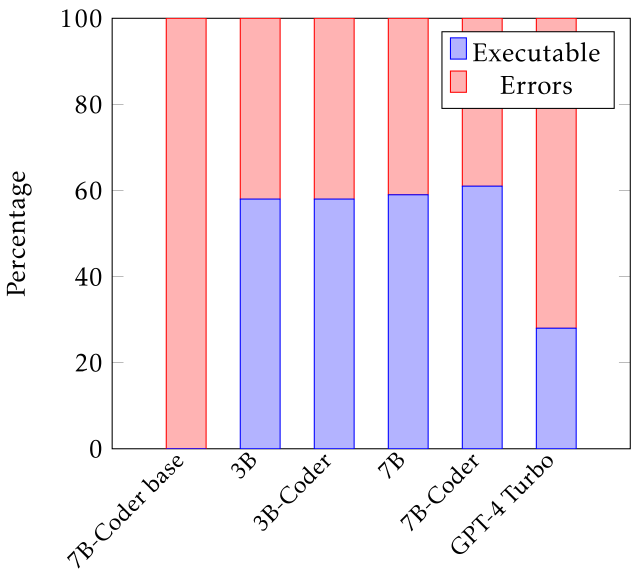


Figure 6. Comparison of Executable Code and Errors Across Models

To assess the reasoning capabilities and the effectiveness of URScript code generation of the fine-tuned model, we carefully selected five test cases with human-crafted prompts. These prompts, which are presented in **Appendix B**, were designed to evaluate the model's ability to understand and respond accurately to complex coding tasks. We systematically collected and analyzed responses from three categories of models: the Qwen2.5-7B-coder model without fine-tuning, the fine-tuned versions of the Qwen2.5-7B-coder, Qwen2.5-7B, Qwen2.5-3B-Coder, Qwen2.5-3B and the GPT-4 Turbo model. This comparative evaluation allows us to examine the impact of fine-tuning on performance and to identify differences in reasoning and code generation capabilities across the models.

The results, as shown in Table 1, highlight notable performance differences across the models. Qwen2.5-7B displayed the good performance, passing three test cases without errors. Qwen2.5-3B and Qwen2.5-Coder-7B (fine-tuned) followed closely, passing four out of five test cases, but each model failed one test case, indicating minor gaps in logical comprehension. Qwen2.5-3B-Coder demonstrated consistent performance, successfully handling all five test cases. In contrast, Qwen2.5-Coder-7B (without fine-tuning) and GPT4-Turbo struggled the most, failing multiple test cases and showing inconsistent outputs. These results suggest that model fine-tuning significantly enhances logical accuracy and generalization. Future steps could involve curating higher-quality datasets and employing advanced methods to further improve model performance.

Figure 4b illustrates the Pick-and-Place operation sequence executed using the URScript generated by the fine-tuned Qwen2.5-Coder-7B model. This sequence includes object detection, positioning, grasping, and placing the object at the target location. The successful execution demonstrates that fine-tuning improves the model's ability to generate accurate and reliable robot control scripts.

4.3. Results with test-dataset

To assess the effectiveness of fine-tuned large language models (LLMs) in generating more generalized URScript code, we evaluated four models—Qwen-3B, Qwen-3B-Coder, Qwen-7B, and Qwen-7B-Coder—using a dataset constructed with Gemini following the Alpaca methodology (as described in Section 3). This dataset contains 100 samples, each consisting of a generation prompt and the corresponding URScript code output. Human evaluation was conducted to verify the executability of the generated code. The results, presented in Figures 5 and 6, illustrate the error distribution and the proportion of executable code across the models

Figure 5 presents an analysis of errors in URScript generation using Qwen-7B-Coder. The majority of the generated scripts (61%) were error-free, while the remaining errors were classified into two common categories: incorrect usage of the **'for'** keyword (21%) and improper handling of **'try-catch'** blocks (7%), with other syntax or logical errors accounting for 11%.

Figure 6 compares the performance of all Qwen2.5 model variants and GPT-4 Turbo in terms of executable code and errors. The Qwen2.5-7B-Coder base model, which is not fine-tuned, exhibited the highest error rate, emphasizing the role of specialized training in enhancing code executability. Additionally, the four fine-tuned Qwen2.5 models (3B, 3B-Coder, 7B, and 7B-Coder) demonstrated comparable performance, indicating that fine-tuning helps narrow the performance gap across model sizes. In contrast, GPT-4 Turbo, despite not being fine-tuned for programming tasks, produced the lowest percentage of executable code among the models compared, suggesting that even advanced general-purpose models may underperform specialized fine-tuned models in URScript generation tasks.

5. Conclusion

This research demonstrated the potential of fine-tuning large language models (LLMs) for generating URScript code to enhance automation in robotics. By constructing

	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5
Qwen2.5-Coder-7B (without fine-tune)	✗	✗	✗	✗	✗
Qwen2.5-3B	✓	✓	✓	✗	✓
Qwen2.5-Coder-3B	✓	✓	✓	✓	✓
Qwen2.5-7B	✓	✓	✗	✓	✗
Qwen2.5-Coder-7B	✓	✓	✓	✗	✓
GPT4-Turbo	✗	✗	✓	✗	✓

Table 1. Error analysis of URScript generation across different Qwen model variants.

a custom dataset of 20,000 synthetic samples derived from validated human-created examples, we improved the model's ability to generate accurate, executable robotic control scripts. Using the Unsloth framework, we fine-tuned multiple Qwen2.5 models and evaluated their performance across critical robotic tasks, including motion planning, object manipulation, and adaptive decision-making.

Our experimental results highlighted the effectiveness of fine-tuned models, with the Qwen2.5-Coder variants showing the highest success rate in generating executable URScript code. Comparisons with general-purpose models, such as GPT-4 Turbo, underscored the advantage of domain-specific fine-tuning in improving both logical coherence and execution reliability. Despite these advancements, challenges remain in ensuring robustness in complex scenarios, particularly in error handling and safety compliance.

Deploying the generated scripts on a UR3 robotic arm validated the real-world applicability of our approach, though further refinement is needed to improve adaptability in dynamic environments. Future work will focus on expanding the dataset with more diverse robotic tasks, enhancing safety verification mechanisms, and exploring reinforcement learning techniques to further optimize fine-tuned LLMs for industrial automation. This study reinforces the transformative role of LLMs in robotic programming and paves the way for more accessible, AI-driven automation solutions.

References

[1] Universal Robots (2024) *Universal Robots unveils its AI Accelerator* [Internet]. Universal-robots.com. Available from: https://www.universal-robots.com/news-and-media/news-center/universal-robots-unveils-its-ai-accelerator/

[2] Ahn M., Brohan A., Brown N., Chebotar Y., Cortes O., David B., Finn C., Fu C., Gopalakrishnan K., Hausman K., Herzog A. (2022) *Do as I can, not as I say: Grounding language in robotic affordances*, arXiv preprint arXiv:2204.01691.

[3] Huang W., Wang C., Zhang R., Li Y., Wu J., Fei-Fei L. (2023) *Voxposer: Composable 3d value maps for robotic manipulation with language models*, arXiv preprint arXiv:2307.05973. 2023 Jul 12.

[4] Yoneda T., Fang J., Li P., Zhang H., Jiang T., Lin S., Picker B., Yunis D., Mei H., Walter M. R. (2024) *Statler: State-maintaining language models for embodied reasoning*, In 2024 IEEE International Conference on Robotics and Automation (ICRA), 2024 May 13 (pp. 15083-15091). IEEE.

[5] Liang J., Huang W., Xia F., Xu P., Hausman K., Ichter B., Florence P., Zeng A. (2023) *Code as policies: Language model programs for embodied control*, In 2023 IEEE International Conference on Robotics and Automation (ICRA), 2023 May 29 (pp. 9493-9500). IEEE.

[6] Mirchandani S., Xia F., Florence P., Ichter B., Driess D., Arenas M. G., Rao K., Sadigh D., Zeng A. (2023) *Large language models as general pattern machines*, arXiv preprint arXiv:2307.04721. 2023 Jul 10.

[7] Joublin F., Ceravola A., Smirnov P., Ocker F., Deigmoeller J., Belardinelli A., Wang C., Hasler S., Tanneberg D., Gienger M. (2024) *Copal: Corrective planning of robot actions with large language models*, In 2024 IEEE International Conference on Robotics and Automation (ICRA), 2024 May 13 (pp. 8664-8670). IEEE.

[8] Driess D., Xia F., Sajjadi M. S., Lynch C., Chowdhery A., Wahid A., Tompson J., Vuong Q., Yu T., Huang W., Chebotar Y. *PaLM-E: An embodied multimodal language model*.

[9] Huang S., Jiang Z., Dong H., Qiao Y., Gao P., Li H. (2023) *Instruct2Act: Mapping multi-modality instructions to robotic actions with large language model*, arXiv preprint arXiv:2305.11176. 2023 May 18.

[10] Chen Y., Arkin J., Zhang Y., Roy N., Fan C. (2024) *Scalable multi-robot collaboration with large language models: Centralized or decentralized systems?*, In 2024 IEEE International Conference on Robotics and Automation (ICRA), 2024 May 13 (pp. 4311-4317). IEEE.

[11] Firoozi R., Tucker J., Tian S., Majumdar A., Sun J., Liu W., Zhu Y., Song S., Kapoor A., Hausman K., Ichter B. (2023) *Foundation models in robotics: Applications, challenges, and the future*, The International Journal

of Robotics Research. 2023 Dec:02783649241281508.

[12] Chen X., Li L., Chang L., Huang Y., Zhao Y., Zhang Y., Li D. (2023) *Challenges and contributing factors in the utilization of large language models (LLMs)*, arXiv preprint arXiv:2310.13343. 2023 Oct 20.

[13] Universal Robots (2024) *Offline Simulator - UR Sim for non Linux 5.12.6 LTS* [Internet]. Available from: https://shorturl.at/zSc4y.

[14] Taori R., Gulrajani I., Zhang T., Dubois Y., Li X., Guestrin C., Liang P., Hashimoto TB. (2023) *Stanford Alpaca: An Instruction-following LLaMA model*, GitHub repository. Available from: https://github.com/tatsu-lab/stanford_alpaca.

[15] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez AN., Kaiser Ł., Polosukhin I. (2017) *Attention is all you need*, Advances in Neural Information Processing Systems, 2017;30.

[16] Yang A., Yang B., Zhang B., Hui B., Zheng B., Yu B., Li C., Liu D., Huang F., Wei H., Lin H. (2024) *Qwen2. 5 technical report*, arXiv preprint arXiv:2412.15115, 2024 Dec 19.

[17] Hui B., Yang J., Cui Z., Yang J., Liu D., Zhang L., Liu T., Zhang J., Yu B., Lu K., Dang K. (2024) *Qwen2. 5-coder technical report*, arXiv preprint arXiv:2409.12186, 2024 Sep 18.

[18] Universal Robots (2024) *Develop with URScript* [Internet]. Available from: https://www.universal-robots.com/developer/urscript/.

F. Appendix A

This appendix provides the prompts used with Gemini to generate a large dataset from seed tasks derived from URScript, following the Alpaca dataset method. These prompts ensure consistency, diversity, and relevance, guiding Gemini to expand task descriptions while maintaining the original structure and context of URScript-based robotic control.

The prompt was adapted from the Alpaca GitHub repository to better suit the problem addressed in this study.

Human prompt:

```
You are asked to come up with a set of 20 diverse URScript programming task instructions.
These task instructions will be evaluated based on their ability to test the capabilities
of a GPT model in generating URScript code for robotics tasks.

Here are the requirements:
1. Ensure diversity in the verbs used for each instruction to maximize variation.
2. Use a variety of linguistic styles, combining questions with imperative instructions.
3. Cover diverse types of programming tasks, such as open-ended code generation,
optimization, debugging, refactoring, and creating examples.
4. Focus exclusively on tasks related to URScript, ensuring they are realistic and align
with common robotics applications (e.g., motion planning, sensor integration, error
handling, etc.).
5. Avoid asking for non-URScript outputs or tasks that require external multimedia
creation or execution (e.g., visuals, audio, or real-world actions).
6. Instructions should be in English and at least 1-2 sentences long.
7. Provide appropriate input data where necessary. Inputs should contain realistic and
specific examples relevant to the instruction but should not exceed 100 words. When no
input context is needed, use "<noinput>".
8. Ensure outputs demonstrate correct and practical URScript implementations.
9. Use '###' to separate tasks clearly.

List of 20 tasks:
```

G. Appendix B

This appendix presents the responses generated by the four fine-tuned models (Qwen-3B, Qwen-3B-Coder, Qwen-7B, Qwen-7B-Coder) alongside GPT-4 for five test cases. These test cases were designed to evaluate model performance in generating accurate, executable, and logically coherent URScript code. The responses provide insights into the strengths and limitations of each model in handling robotic control tasks.

Test case 1: Move though 2 waypoints

Human prompt:

```
"Write a URScript function that moves a UR robot linearly in tool-space between
two waypoints.

Waypoint 1:
    Tool position: [1, 1, 1, 1, 1, 1]
    Joint acceleration: 1.39 rad/s²
    Joint speed: 1.04 rad/s

Waypoint 2:
```

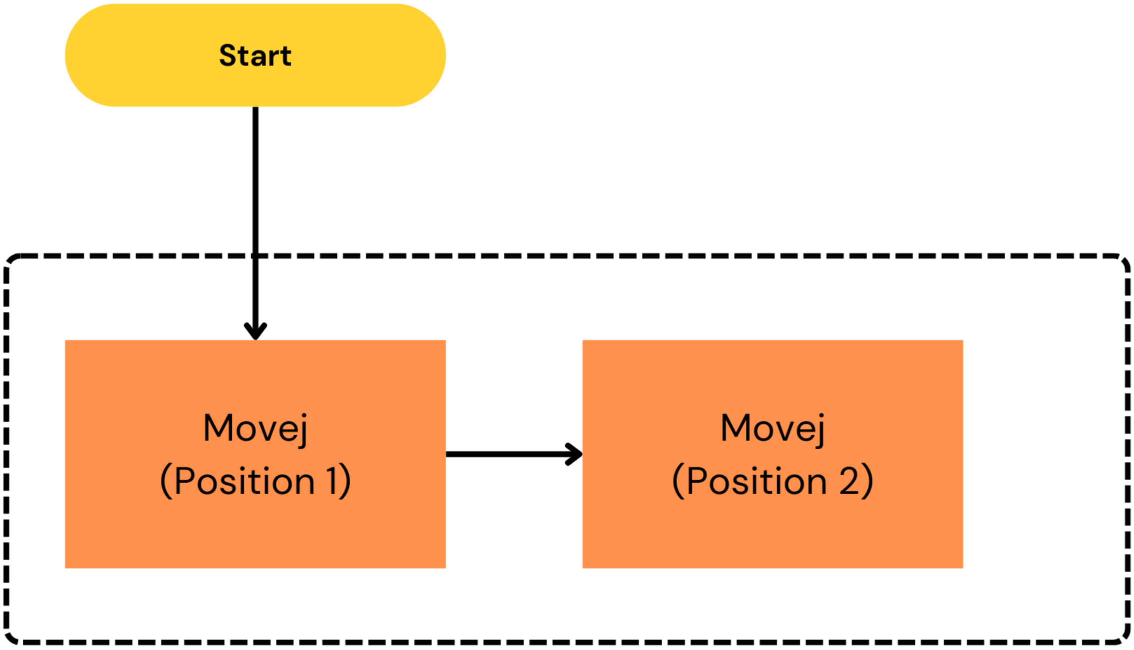


Figure G.1. Move though 2 waypoints

```
    Tool position: [-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    Joint acceleration: 0.78 rad/s²
    Joint speed: 0.87 rad/s

The function should execute the motion using active TCP. Please provide the complete
URScript code."
```

Qwen2.5-Coder-7B without fine-tuning Response:

```
def move_linear_tool_space():
    # Set the tool position for Waypoint 1
    tool_position_1 = [1, 1, 1, 1, 1, 1]
    # Set the joint acceleration for Waypoint 1
    joint_acceleration_1 = 1.39
    # Set the joint speed for Waypoint 1
    joint_speed_1 = 1.04

    # Set the tool position for Waypoint 2
    tool_position_2 = [-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    # Set the joint acceleration for Waypoint 2
    joint_acceleration_2 = 0.78
    # Set the joint speed for Waypoint 2
    joint_speed_2 = 0.87

    # Move the robot linearly in tool-space from Waypoint 1 to Waypoint 2
    move_l(tool_position_1, tool_position_2,
           joint_acceleration_1, joint_speed_1,
```

```
            joint_acceleration_2, joint_speed_2, active_tcp=True
    )
```

GPT-4 Turbo Response:

```
def move_linear_between_waypoints():
    # Waypoint 1: Tool position and motion parameters
    waypoint_1 = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
    acc_1 = 1.39  # Joint acceleration (rad/s²)
    speed_1 = 1.04  # Joint speed (rad/s)

    # Waypoint 2: Tool position and motion parameters
    waypoint_2 = [-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    acc_2 = 0.78  # Joint acceleration (rad/s²)
    speed_2 = 0.87  # Joint speed (rad/s)

    # Move to Waypoint 1
    movel(pose_trans(get_actual_tcp_pose(), waypoint_1), a=acc_1, v=speed_1)

    # Move to Waypoint 2
    movel(pose_trans(get_actual_tcp_pose(), waypoint_2), a=acc_2, v=speed_2)

move_linear_between_waypoints()
```

Qwen2.5-3B-Coder Response:

````
```URscript
def move_linear_waypoints():
  local waypoint1 = p[1, 1, 1, 1, 1, 1]
  local waypoint2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  movel(waypoint1, a=1.39, v=1.04, r=0.02)
  movel(waypoint2, a=0.78, v=0.87, r=0.02)
end
```
````

Qwen2.5-3B Response:

````
```URscript
def move_between_waypoints():
  local waypoint_1 = p[1, 1, 1, 1, 1, 1]
  local waypoint_2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  movej(get_actual_joint_positions(), a=1.39, v=1.04)
  movel(waypoint_1, a=0.78, v=0.87)
  movel(waypoint_2, a=0.78, v=0.87)
end
```
````

Qwen2.5-7B-Coder Response:

````
```URscript
def move_linear_toolspace():
  local waypoint1 = p[1, 1, 1, 1, 1, 1]
  local waypoint2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
````
````

```
  movej(get_inverse_kin(waypoint1, qnear=get_joint_positions()), a=1.39, v=1.04)
  movel(waypoint2, a=0.78, v=0.87)
end
```

Qwen2.5-7B Response:

```URscript
def move_linear_tool_space():
  local waypoint_1_pose = p[1, 1, 1, 1, 1, 1]
  local waypoint_1_joint_acc = 1.39
  local waypoint_1_joint_speed = 1.04

  local waypoint_2_pose = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_2_joint_acc = 0.78
  local waypoint_2_joint_speed = 0.87

  movel(waypoint_1_pose, a=waypoint_1_joint_acc, v=waypoint_1_joint_speed, r=0.02)
  movel(waypoint_2_pose, a=waypoint_2_joint_acc, v=waypoint_2_joint_speed, r=0.02)
end
```

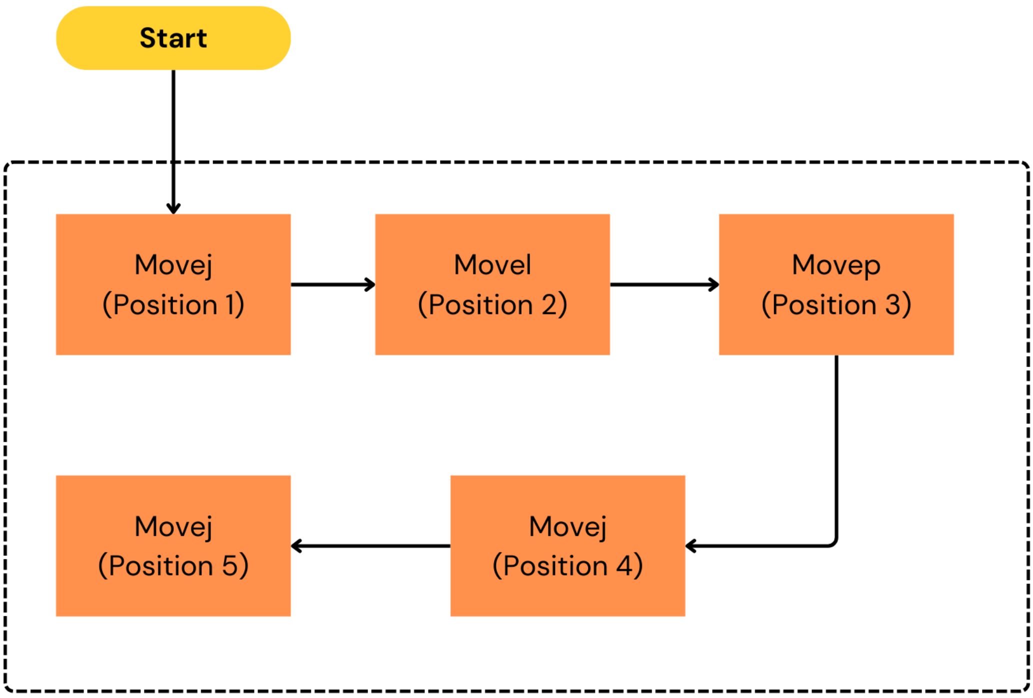


Figure G.2. Move through series of waypoints

Test case 2: Move through series of waypoints

Human prompt:

```
Write a URScript function to move a UR robot linearly in tool-space between
three waypoints.

Waypoint 1:
    Tool position: [1, 1, 1, 1, 1, 1]
    Joint acceleration: 1.39 rad/s²
    Joint speed: 1.04 rad/s

Waypoint 2:
    Tool position: [-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    Joint acceleration: 0.78 rad/s²
    Joint speed: 0.87 rad/s

Waypoint 3:
    Tool position: [-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    Joint acceleration: 0.78 rad/s²
    Joint speed: 0.87 rad/s

Waypoint 4:
  Tool position: [0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
  Joint acceleration: 0.9 rad/s²
  Joint speed: 0.8 rad/s

Waypoint 5:
  Tool position: [0, 0, 0, 0, 0, 0]
  Joint acceleration: 1.0 rad/s²
  Joint speed: 0.9 rad/s

The function should execute the motion using active TCP. Please provide the complete
URScript code.
```

Qwen2.5-Coder-7B without fine-tuning Response:

```
def move_linear_tool_space():
    # Set active TCP
    set_active_tool(""tool0"")

    # Waypoint 1
    move_l([1, 1, 1, 1, 1, 1], acc=1.39, vel=1.04)

    # Waypoint 2
    move_l([-0.008, -0.26, 0.28, -0.27, -2.71, 0.217], acc=0.78, vel=0.87)

    # Waypoint 3
    move_l([-0.008, -0.26, 0.28, -0.27, -2.71, 0.217], acc=0.78, vel=0.87)

    # Waypoint 4
    move_l([0.3, -0.4, 0.2, 0.1, -1.5, 0.5], acc=0.9, vel=0.8)

    # Waypoint 5
    move_l([0, 0, 0, 0, 0, 0], acc=1.0, vel=0.9)
```

GPT-4 Turbo Response:

```
"def move_through_waypoints():
    # Ensure the TCP is active
    set_tcp(p[0, 0, 0, 0, 0, 0])

    # Waypoint 1
    pose1 = p[1, 1, 1, 1, 1, 1]
    speed1 = 1.04
    accel1 = 1.39
    movel(pose1, a=accel1, v=speed1)

    # Waypoint 2
    pose2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    speed2 = 0.87
    accel2 = 0.78
    movel(pose2, a=accel2, v=speed2)

    # Waypoint 3 (Same as Waypoint 2)
    movel(pose2, a=accel2, v=speed2)

    # Waypoint 4
    pose4 = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
    speed4 = 0.8
    accel4 = 0.9
    movel(pose4, a=accel4, v=speed4)

    # Waypoint 5
    pose5 = p[0, 0, 0, 0, 0, 0]
    speed5 = 0.9
    accel5 = 1.0
    movel(pose5, a=accel5, v=speed5)

    popup(""Motion between waypoints completed"")

# Call the function
move_through_waypoints()"
```

Qwen2.5-3B-Coder Response:

````
```URscript
def move_through_waypoints():
  local waypoint_1 = p[1, 1, 1, 1, 1, 1]
  local waypoint_2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_3 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_4 = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
  local waypoint_5 = p[0, 0, 0, 0, 0, 0]

  movej(waypoint_1, a=1.39, v=1.04)
  movel(waypoint_2, a=0.78, v=0.87)
  movel(waypoint_3, a=0.78, v=0.87)
  movel(waypoint_4, a=0.9, v=0.8)
  movel(waypoint_5, a=1.0, v=0.9)
end
```
````

Qwen2.5-3B Response:

```URscript
def move_through_waypoints():
  local waypoint_1 = p[1, 1, 1, 1, 1, 1]
  local waypoint_2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_3 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_4 = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
  local waypoint_5 = p[0, 0, 0, 0, 0, 0]

  movej(waypoint_1, a=1.39, v=1.04)
  movej(waypoint_2, a=0.78, v=0.87)
  movej(waypoint_3, a=0.78, v=0.87)
  movej(waypoint_4, a=0.9, v=0.8)
  movej(waypoint_5, a=1.0, v=0.9)
end
```

Qwen2.5-7B-Coder Response:

```URscript
def move_through_waypoints():
  local waypoint_1 = p[1, 1, 1, 1, 1, 1]
  local waypoint_2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_3 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_4 = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
  local waypoint_5 = p[0, 0, 0, 0, 0, 0]

  movej(waypoint_1, a=1.39, v=1.04)
  movel(waypoint_2, a=0.78, v=0.87)
  movel(waypoint_3, a=0.78, v=0.87)
  movel(waypoint_4, a=0.9, v=0.8)
  movel(waypoint_5, a=1.0, v=0.9)
end
```

Qwen2.5-7B Response:

```URscript
def move_through_waypoints():
  local waypoint_1 = p[1, 1, 1, 1, 1, 1]
  local waypoint_2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_3 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_4 = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
  local waypoint_5 = p[0, 0, 0, 0, 0, 0]

  local acc_1 = 1.39
  local vel_1 = 1.04
  local acc_2 = 0.78
  local vel_2 = 0.87
  local acc_3 = 0.9
  local vel_3 = 0.8
  local acc_4 = 1.0
```

~~~~
  local vel_4 = 0.9

  movel(waypoint_1, a=acc_1, v=vel_1)
  movel(waypoint_2, a=acc_2, v=vel_2)
  movel(waypoint_3, a=acc_2, v=vel_2)
  movel(waypoint_4, a=acc_3, v=vel_3)
  movel(waypoint_5, a=acc_4, v=vel_4)
end
```
~~~~

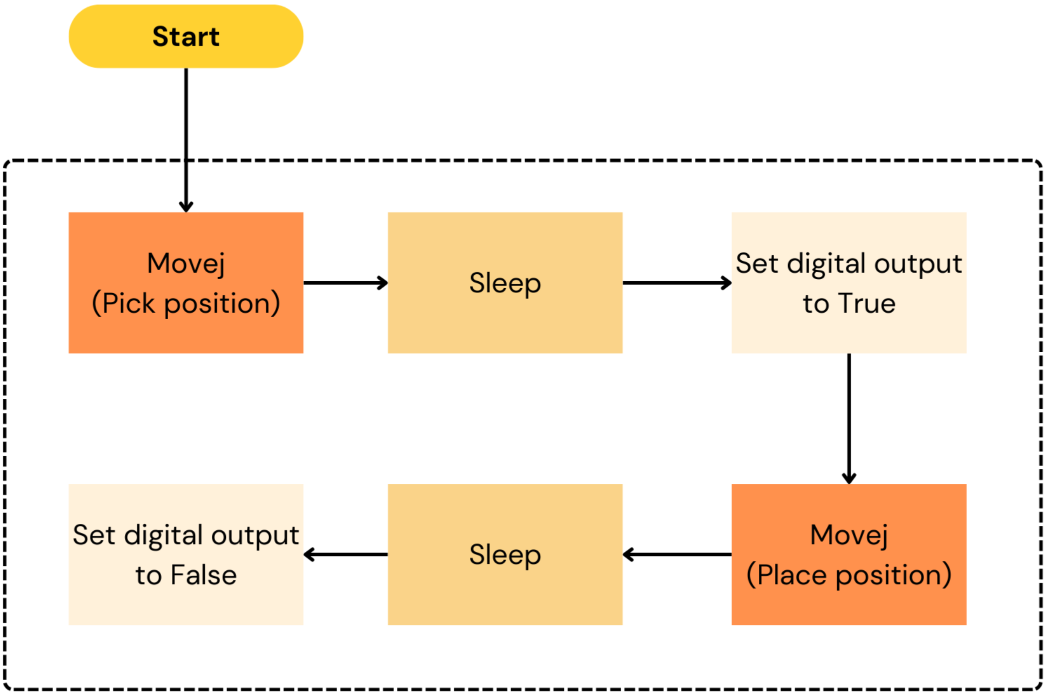


Figure G.3. Pick and place operations

Test case 3: Pick and place operations

Human prompt:

```
Write a URScript function to move a UR robot linearly in tool-space between
two waypoints.

Waypoint 1:
    Tool position: [-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]

Waypoint 2:
    Tool position: [-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]

Joint acceleration and speed: 1.39 rad/s², 1.04 rad/s (Waypoint 1)
and 0.78 rad/s², 0.87 rad/s (Waypoint 2).
Use active TCP for movement.
```

```
Implement a Pick and Place operation by controlling the digital output at
each waypoint.

Provide the complete URScript code.
```

Qwen2.5-Coder-7B without fine-tuning Response:

```
def move_linear_tool_space():
    # Set the active TCP
    set_tcp(active)

    # Waypoint 1
    waypoint1 = [-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
    joint_acc1 = 1.39
    joint_speed1 = 1.04

    # Move to Waypoint 1
    movej(waypoint1, a=joint_acc1, v=joint_speed1)
    # Perform Pick operation
    set_digital_out(0, True)

    # Waypoint 2
    waypoint2 = [-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    joint_acc2 = 0.78
    joint_speed2 = 0.87

    # Move to Waypoint 2
    movej(waypoint2, a=joint_acc2, v=joint_speed2)
    # Perform Place operation
    set_digital_out(0, False)
```

GPT-4 Turbo Response:

```
# URScript function to perform linear movement between two waypoints with a
# pick and place operation
def pick_and_place():

    # Set active TCP (tool center point) if required
    # Use pose_trans() if you need to offset TCP

    # Move to Waypoint 1
    waypoint1 = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
    speed1 = 1.04
    accel1 = 1.39

    movel(waypoint1, a=accel1, v=speed1)

    # Activate gripper (simulate pick operation)
    set_digital_out(0, True)  # Set digital output 0 to high (on)
    sleep(1.0)  # Wait for the pick action to complete

    # Move to Waypoint 2
    waypoint2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    speed2 = 0.87
```

```
    accel2 = 0.78

    movel(waypoint2, a=accel2, v=speed2)

    # Deactivate gripper (simulate place operation)
    set_digital_out(0, False)  # Set digital output 0 to low (off)
    sleep(1.0)  # Wait for the place action to complete

    # Return to Waypoint 1
    movel(waypoint1, a=accel1, v=speed1)

end

# Execute the function
pick_and_place()"
```

Qwen2.5-3B-Coder Response:

```
```URscript
def move_linear_pick_place():
  local waypoint1 = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
  local waypoint2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local a1 = 1.39
  local v1 = 1.04
  local a2 = 0.78
  local v2 = 0.87

  movej(waypoint1, a=a1, v=v1)
  movel(waypoint1, a=a1, v=v1)
  set_digital_out(0, True)
  sleep(0.5)
  movej(waypoint2, a=a2, v=v2)
  movel(waypoint2, a=a2, v=v2)
  set_digital_out(0, False)
  sleep(0.5)
end
```
```

**Qwen2.5-3B Response:**

```
```URscript
def move_between_waypoints():
  local waypoint_1_pose = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
  local waypoint_2_pose = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_1_speed = 1.04
  local waypoint_1_accel = 1.39
  local waypoint_2_speed = 0.87
  local waypoint_2_accel = 0.78
  local output_pin = 0

  movej(get_actual_joint_positions(), a=waypoint_1_accel, v=waypoint_1_speed)
  set_digital_out(output_pin, True)
  sleep(1)
  movej(get_actual_joint_positions(), a=waypoint_2_accel, v=waypoint_2_speed)
```
```

```
  set_digital_out(output_pin, False)
  sleep(1)
end
```

Qwen2.5-7B-Coder Response:

```URscript
def move_linear_toolspace():
  local waypoint1_p = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
  local waypoint2_p = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]

  movej(get_inverse_kin(waypoint1_p, qnear=get_joint_positions()), a=1.39, v=1.04)
  set_digital_out(0, True)
  sleep(1)
  movel(waypoint1_p, a=0.78, v=0.87)
  set_digital_out(0, False)
  sleep(1)
  movel(waypoint2_p, a=0.78, v=0.87)
  set_digital_out(0, True)
  sleep(1)
  movel(waypoint1_p, a=0.78, v=0.87)
  set_digital_out(0, False)
end
```

Qwen7B Response:

```URscript
def pick_and_place():
  local waypoint_1_pose = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
  local waypoint_2_pose = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_1_joints = get_inverse_kin(waypoint_1_pose)
  local waypoint_2_joints = get_inverse_kin(waypoint_2_pose)

  movej(waypoint_1_joints, a=1.39, v=1.04)
  set_digital_out(0, True) # Activate digital output
  sleep(1) # Wait for pick

  movel(waypoint_1_pose, a=0.78, v=0.87)
  set_digital_out(0, False) # Deactivate digital output
  sleep(1) # Wait for place

  movel(waypoint_2_pose, a=0.78, v=0.87)
  set_digital_out(0, True) # Activate digital output
  sleep(1) # Wait for pick

  movej(waypoint_2_joints, a=1.39, v=1.04)
  set_digital_out(0, False) # Deactivate digital output
end
```

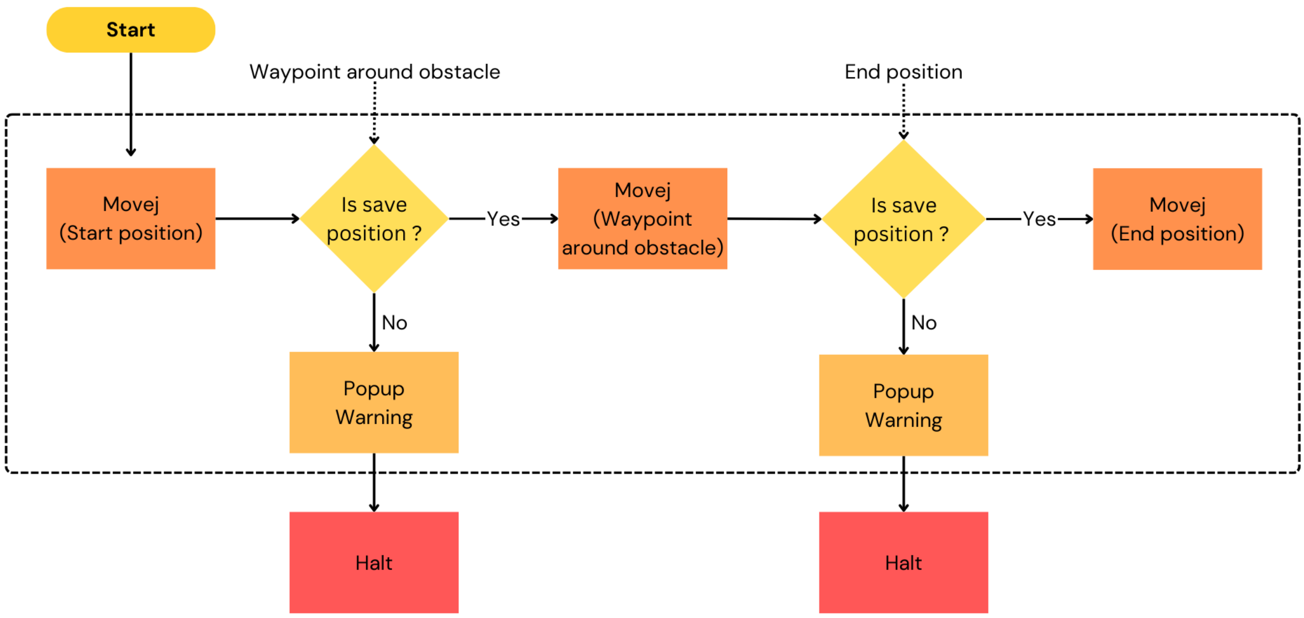


Figure G.4. Move with safety conditions

Test case 4: Move with safety conditions

Human prompt:

```
Write a URScript function to move a UR robot linearly in tool-space between
two waypoints.

Waypoint 1:
    Tool position: [-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]

Waypoint 2:
    Tool position: [-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]

Waypoint 3:
    Tool position: [0.12, -0.15, 0.35, 0.15, 2.9, -0.1]

Joint acceleration and speed: 1.39 rad/s², 1.04 rad/s (Waypoint 1) and
0.78 rad/s², 0.87 rad/s (Waypoint 2).
Use active TCP for movement.
Before executing the motion to each waypoints, ensure safe operation by verifying the tools
positions of next waypoints. If necessary, alert the user and halt execution.
Waypoint 1 not need to check safety
Provide the complete URScript code.
```

Qwen2.5-Coder-7B without fine-tuning Response:

```
def move_linear_tool_space():
    # Waypoint 1
```

```
    tool_position_1 = [-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
    joint_acceleration_1 = 1.39
    joint_speed_1 = 1.04

    # Waypoint 2
    tool_position_2 = [-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    joint_acceleration_2 = 0.78
    joint_speed_2 = 0.87

    # Waypoint 3
    tool_position_3 = [0.12, -0.15, 0.35, 0.15, 2.9, -0.1]

    # Check safety for Waypoint 2
    if not check_safety(tool_position_2):
        print(""Safety check failed for Waypoint 2. Halt execution."")
        return

    # Move to Waypoint 1
    move_l_tool_space(tool_position_1, joint_acceleration_1, joint_speed_1)

    # Check safety for Waypoint 3
    if not check_safety(tool_position_3):
        print(""Safety check failed for Waypoint 3. Halt execution."")
        return

    # Move to Waypoint 2
    move_l_tool_space(tool_position_2, joint_acceleration_2, joint_speed_2)

    # Move to Waypoint 3
    move_l_tool_space(tool_position_3, joint_acceleration_2, joint_speed_2)

def check_safety(tool_position):
    # Implement your safety check logic here
    # Return True if safe, False otherwise
    pass

def move_l_tool_space(tool_position, joint_acceleration, joint_speed):
    # Implement your linear motion in tool-space logic here
    pass
```

GPT-4 Turbo Response:

```
def move_linear():
    # Define waypoints
    wp1 = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
    wp2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
    wp3 = p[0.12, -0.15, 0.35, 0.15, 2.9, -0.1]

    # Define joint acceleration and velocity for waypoints
    a1 = 1.39
    v1 = 1.04
    a2 = 0.78
    v2 = 0.87
```

```
    # Function to check safety (e.g., limit positions, collisions)
    def check_safety(target_pose):
        # Example safety check: Ensure Z position is within a safe range
        if target_pose[2] < 0.1 or target_pose[2] > 0.5:
            textmsg(""Unsafe waypoint detected, halting execution."")
            halt()

    # Move to Waypoint 1 directly
    movel(wp1, a=a1, v=v1)

    # Check safety before moving to Waypoint 2
    check_safety(wp2)
    movel(wp2, a=a2, v=v2)

    # Check safety before moving to Waypoint 3
    check_safety(wp3)
    movel(wp3, a=a2, v=v2)

end
```

Qwen2.5-3B-Coder Response:

```
```URscript
def move_linear_waypoints():
  local waypoint_1 = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
  local waypoint_2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_3 = p[0.12, -0.15, 0.35, 0.15, 2.9, -0.1]

  # Move to waypoint 1
  movel(waypoint_1, a=1.39, v=1.04)

  # Move to waypoint 2
  if (norm(pose_sub(get_actual_tcp_pose(), waypoint_2)) > 0.01):
    popup(""Safety check failed: Waypoint 2 is not reachable"",
          ""Safety Error"", False, True, True)
    halt
  end
  movel(waypoint_2, a=0.78, v=0.87)

  # Move to waypoint 3
  if (norm(pose_sub(get_actual_tcp_pose(), waypoint_3)) > 0.01):
    popup(""Safety check failed: Waypoint 3 is not reachable"",
          ""Safety Error"", False, True, True)
    halt
  end
  movel(waypoint_3, a=0.78, v=0.87)
end
```
```

**Qwen2.5-3B Response:**

```
```URscript
def move_linear_waypoints():
  local waypoint_1 = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
```
```

```
  local waypoint_2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_3 = p[0.12, -0.15, 0.35, 0.15, 2.9, -0.1]

  # Check safety for waypoint 1
  if (norm(get_actual_tcp_pose() - waypoint_1) > 0.01):
    popup(""Safety check failed for waypoint 1"", ""Error"", False, True, False)
    halt
  end

  # Check safety for waypoint 2
  if (norm(get_actual_tcp_pose() - waypoint_2) > 0.01):
    popup(""Safety check failed for waypoint 2"", ""Error"", False, True, False)
    halt
  end

  # Move to waypoint 1
  movel(waypoint_1, a=1.39, v=1.04)

  # Move to waypoint 2
  movel(waypoint_2, a=0.78, v=0.87)

  # Move to waypoint 3
  movel(waypoint_3, a=1.39, v=1.04)
end
```

Qwen2.5-7B-Coder Response:

```URscript
def move_linear_toolspace():
  local waypoint_1_p = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
  local waypoint_2_p = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
  local waypoint_3_p = p[0.12, -0.15, 0.35, 0.15, 2.9, -0.1]

  movej(get_inverse_kin(waypoint_1_p, qnear=get_joint_positions()), a=1.39, v=1.04)
  movel(waypoint_2_p, a=0.78, v=0.87)
  if (not is_safe()):
    popup(""Safety check failed at waypoint 2"", ""Safety Error"", False, True, True)
    halt
  end
  movel(waypoint_3_p, a=0.78, v=0.87)
  if (not is_safe()):
    popup(""Safety check failed at waypoint 3"", ""Safety Error"", False, True, True)
    halt
  end
end
```

Qwen2.5-7B Response:

```URscript
def move_linear_tool_space():
  local waypoint_1 = p[-0.13, -0.26, 0.14, -0.001, 3.11, 0.03]
  local waypoint_2 = p[-0.008, -0.26, 0.28, -0.27, -2.71, 0.217]
```

````
    local waypoint_3 = p[0.12, -0.15, 0.35, 0.15, 2.9, -0.1]

    # Move to Waypoint 1
    movel(waypoint_1, a=1.39, v=1.04)
    # Check safety for Waypoint 2
    if (norm(waypoint_2 - get_actual_tcp_pose()) > 0.01):
      popup(""Safety violation at Waypoint 2"", ""Safety Alert"", False, True, False)
      halt
    end
    # Move to Waypoint 2
    movel(waypoint_2, a=0.78, v=0.87)
    # Check safety for Waypoint 3
    if (norm(waypoint_3 - get_actual_tcp_pose()) > 0.01):
      popup(""Safety violation at Waypoint 3"", ""Safety Alert"", False, True, False)
      halt
    end
    # Move to Waypoint 3
    movel(waypoint_3, a=0.78, v=0.87)
  end
  ```
````

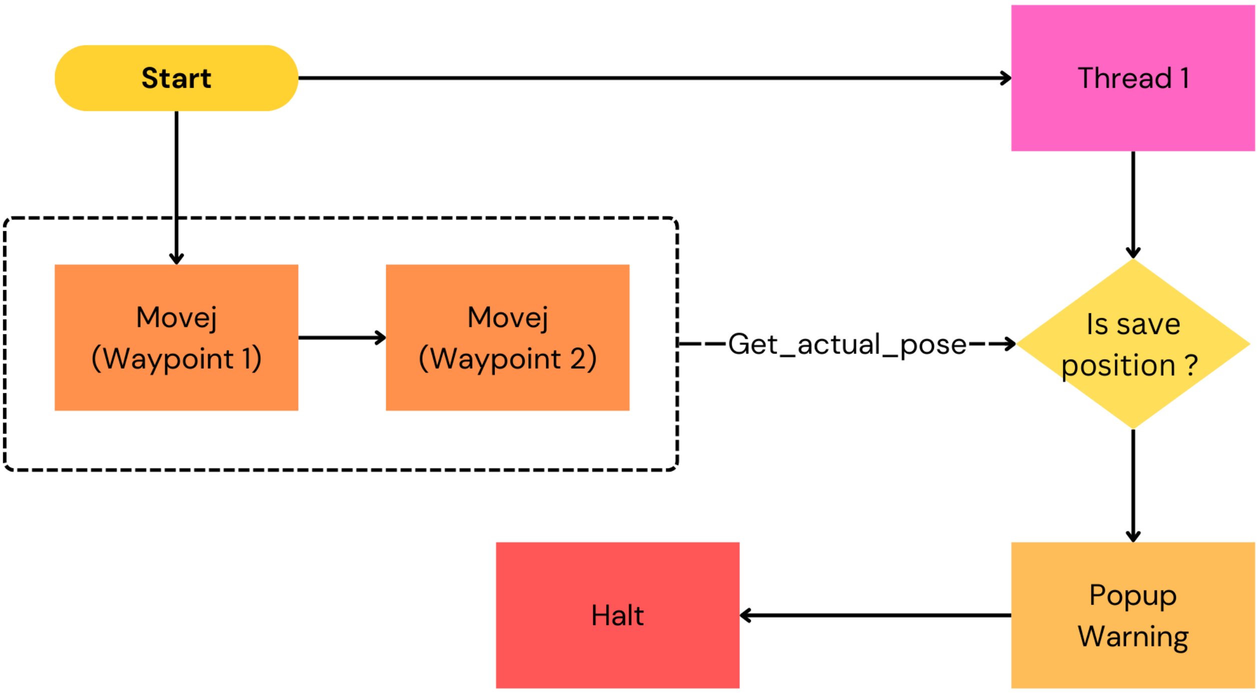


Figure G.5. Safety thread checking

Test case 5: Safety thread checking

Human prompt:

```
Write a URScript function to move a UR robot linearly in tool-space to 2 waypoints.
```
```

```
    Waypoint 1:
        Tool position: [1, 1, 1, 1, 1, 1]
        Joint acceleration: 1.39 rad/s²
        Joint speed: 1.04 rad/s

    Waypoint 2:
        Tool position: [0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
        Joint acceleration: 0.9 rad/s²
        Joint speed: 0.8 rad/s

Use active TCP for movement.
Implement a thread to monitor the first element of the joint position at waypoint.
If it exceeds 1.0, the program should halt.
Provide the complete URScript code.
```

Qwen2.5-Coder-7B without fine-tuning Response:

```
def move_to_waypoints():
    # Set the active TCP
    set_tcp([0, 0, 0, 0, 0, 0])

    # Define the waypoints
    waypoint1 = [1, 1, 1, 1, 1, 1]
    waypoint2 = [0.3, -0.4, 0.2, 0.1, -1.5, 0.5]

    # Define the joint acceleration and speed for each waypoint
    joint_acc1 = 1.39
    joint_speed1 = 1.04
    joint_acc2 = 0.9
    joint_speed2 = 0.8

    # Move to the first waypoint
    movej(waypoint1, a=joint_acc1, v=joint_speed1)

    # Monitor the first element of the joint position at the first waypoint
    while True:
        joint_pos = get_actual_q()
        if joint_pos[0] > 1.0:
            print(""Joint position exceeded 1.0, halting program."")
            break
        else:
            print(""Joint position is within limits."")

    # Move to the second waypoint
    movej(waypoint2, a=joint_acc2, v=joint_speed2)

    # Monitor the first element of the joint position at the second waypoint
    while True:
        joint_pos = get_actual_q()
        if joint_pos[0] > 1.0:
            print(""Joint position exceeded 1.0, halting program."")
            break
        else:
            print(""Joint position is within limits."")
```

```
# Call the function to move to the waypoints
move_to_waypoints()
```

ChatGPT-4 Turbo Response:

```
# URScript to move UR robot linearly to 2 waypoints with monitoring
def linear_move():
    # Start monitoring joint position
    thread monitor_joint():
        while True:
            current_joint_position = get_actual_joint_positions()
            if current_joint_position[0] > 1.0:
                popup(""Joint 1 exceeds limit. Halting!"", error=True)
                halt()
            sync()
        end
    end

    monitor_thread = run monitor_joint()

    # Move to Waypoint 1
    pose1 = p[1, 1, 1, 1, 1, 1]
    movel(pose1, a=1.39, v=1.04)

    # Move to Waypoint 2
    pose2 = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
    movel(pose2, a=0.9, v=0.8)

    # Stop monitoring
    kill monitor_thread
end

# Execute the function
linear_move()
```

Qwen2.5-3B-Coder Response:

```
```URscript
def move_between_waypoints():
  local waypoint1 = p[1, 1, 1, 1, 1, 1]
  local waypoint2 = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
  local joint_acceleration = 1.39
  local joint_speed = 1.04
  local tcp_acceleration = 0.9
  local tcp_speed = 0.8

  thread monitor_joint():
    while True:
      local current_joint_positions = get_actual_joint_positions()
      if current_joint_positions[0] > 1.0:
        halt
      end
      sync()
    end
```
```

```
  end

  thread monitor_joint_thread = thread monitor_joint()

  movel(waypoint1, a=joint_acceleration, v=joint_speed, t=0, r=waypoint1)
  movel(waypoint2, a=tcp_acceleration, v=tcp_speed, t=0, r=waypoint2)
  kill monitor_joint_thread
end
```
```

Qwen2.5-3B Response:

```URscript
def move_to_waypoints():
  local waypoint_1 = p[1, 1, 1, 1, 1, 1]
  local waypoint_2 = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
  local joint_acceleration_1 = 1.39
  local joint_speed_1 = 1.04
  local joint_acceleration_2 = 0.9
  local joint_speed_2 = 0.8

  thread joint_monitor():
    while True:
      local joint_positions = get_actual_joint_positions()
      if joint_positions[0] > 1.0:
        halt
      end
      sleep(0.1)
    end
  end

  thread move_thread():
    movel(waypoint_1, a=joint_acceleration_1, v=joint_speed_1)
    movel(waypoint_2, a=joint_acceleration_2, v=joint_speed_2)
  end

  thread start joint_monitor()
  thread start move_thread()
end
```

Qwen2.5-7B-Coder Response:

```URscript
def move_linear_with_monitor():
    local waypoint1 = p[1, 1, 1, 1, 1, 1]
    local waypoint2 = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]

    movel(waypoint1, a=1.39, v=1.04)
    movel(waypoint2, a=0.9, v=0.8)

    thread monitor_thread():
        while True:
            local current_joint_pos = get_actual_joint_positions()
```
```

```
            if current_joint_pos[0] > 1.0:
                halt
            end
            sync()
        end
    end
    monitor_thread()
end
```

Qwen2.5-7B Response:

```URscript
def move_with_monitor():
  # Waypoint 1
  local waypoint_1_pose = p[1, 1, 1, 1, 1, 1]
  local waypoint_1_joint_acc = 1.39
  local waypoint_1_joint_speed = 1.04

  # Waypoint 2
  local waypoint_2_pose = p[0.3, -0.4, 0.2, 0.1, -1.5, 0.5]
  local waypoint_2_joint_acc = 0.9
  local waypoint_2_joint_speed = 0.8

  # Move to waypoint 1
  movej(get_inverse_kin(waypoint_1_pose, qnear=get_joint_positions()),
        a=waypoint_1_joint_acc, v=waypoint_1_joint_speed)

  # Monitor joint 1
  thread joint_monitor():
    while (True):
      local joint_positions = get_joint_positions()
      if (joint_positions[0] > 1.0):
        halt()
      end
      sync()
    end
  end

  # Move to waypoint 2
  movej(get_inverse_kin(waypoint_2_pose, qnear=get_joint_positions()),
        a=waypoint_2_joint_acc, v=waypoint_2_joint_speed)
end
```