Automating Flexible Manufacturing: A Generative AI Approach for Code Generation and Validation for Robotics

Master's Thesis in Data Science and AI

ZHICHAO ZHOU

DEPARTMENT OF MECHANICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

Automating Flexible Manufacturing: A Generative AI Approach for Code Generation and Validation for Robotics

ZHICHAO ZHOU

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mechanical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

Automating Flexible Manufacturing: A Generative AI Approach for Code Generation and Validation for Robotics
ZHICHAO ZHOU

Supervisor: Siyuan Chen, Department of Mechanical Engineering
Supervisor: Omkar Salunkhe, Department of Mechanical Engineering
Examiner: Johan Stahre, Department of Mechanical Engineering

Master's Thesis 2026
Department of Mechanical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: An industrial robot performing automated manufacturing tasks in a simulation environment.

Typeset in LaTeX
Gothenburg, Sweden 2026

Automating Flexible Manufacturing: A Generative AI Approach for Code Generation and Validation for Robotics
Master's thesis in Data Science and AI, MSc Programme
ZHICHAO ZHOU
Department of Mechanical Engineering
Chalmers University of Technology

Abstract

Flexible manufacturing demands that industrial robots be reprogrammed whenever product variants change. Writing executable code in specific robot languages is unreliable, and verifying such code against physical constraints still depends on slow manual simulation by domain experts. Automating both steps is therefore a prerequisite for wider robotic deployment.

This thesis presents an agentic system that generates, validates, and iteratively corrects ABB RAPID code from natural language task descriptions. A dual stream Retrieval Augmented Generation (RAG) pipeline, combined with Hypothetical Document Embeddings (HyDE), grounds generation in verified technical documentation and production code templates. This grounding removes most of the domain specific hallucinations observed in ungrounded large language models. Four cooperating roles run the workflow and catch semantic defects that rule based static analysis often misses.

At the heart of the system sits a custom Model Context Protocol (MCP) server that plugs the AI agent directly into ABB RobotStudio. Code uploads, simulation runs, and diagnostic feedback all flow through this channel, with no manual step in between. Inside the loop the agent catches physical constraint violations—wrist singularities, joint limit exceedances, arc geometry errors, suction release failures—that would pass through both static checks and semantic review. Over seven experiments of rising complexity, the simulation feedback loop cut correction rounds from six on the first task to first-attempt success by the fifth, since fixes learned earlier carried over to new tasks.

Across three compared configurations, retrieval grounding alone lifts the content pass rate from 20% to near-complete; adding the MCP simulation loop on top catches a further family of physically grounded errors that would otherwise go unnoticed before deployment. Such a pipeline suits industrial settings that call for quick reconfiguration and little expert time in the loop.

Keywords: Retrieval Augmented Generation, Agentic AI, RAPID, Robotics, Model Context Protocol, Code Generation, Flexible Manufacturing.

Acknowledgements

This master's thesis was carried out at the Department of Mechanical Engineering, Chalmers University of Technology.
My biggest thanks go to Siyuan Chen, my supervisor. He spent plenty of stuck moments with me. Held a lot of meetings between us, help and see what problem I encounter. Most of what I now know about the domain, I picked up across those conversations. The MCP RobotStudio integration as the core of this thesis was not originally my idea. It came from one of his suggestions. Thank you, Siyuan.
Omkar Salunkhe, also my supervisor, gives me great help around the specific robot problems. Omkar is an expert in this field, and he knows a lot.
Marcus Sörensson, on the industrial side, gave me some really good suggestions of how to improve my work. It really means a lot.
And to my family and friends: thanks for bearing with me through these Chalmers years.

Zhichao Zhou, Gothenburg, April 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

ANN	Approximate Nearest Neighbor
API	Application Programming Interface
BOM	Byte Order Mark
CoT	Chain of Thought
EIO	Electronic I/O (ABB signal configuration system)
HNSW	Hierarchical Navigable Small World
HyDE	Hypothetical Document Embeddings
I/O	Input/Output
LLM	Large Language Model
LSTM	Long Short-Term Memory
MCP	Model Context Protocol
MRR	Mean Reciprocal Rank
PLC	Programmable Logic Controller
RAG	Retrieval-Augmented Generation
RAPID	Robot Application Programming Interface for Development (ABB)
RNN	Recurrent Neural Network
RQ	Research Question
TCP	Tool Center Point

Nomenclature

Below is the nomenclature of key domain-specific terms and data types that have been used throughout this thesis.

RAPID Data Types

`robtarget`	Cartesian position and orientation of the robot tool (includes position, quaternion, robot configuration, and external axes)
`tooldata`	Tool definition including Tool Center Point (TCP) coordinates and load parameters
`wobjdata`	Work object definition specifying the coordinate system of the workpiece
`speeddata`	Motion speed specification (e.g., `v100`, `v500`)
`zonedata`	Path zone size for corner blending (e.g., `fine`, `z10`)
`signaldo`	Digital output signal variable
`signaldi`	Digital input signal variable

RAPID Instructions

`MoveL`	Linear motion to a Cartesian target
`MoveJ`	Joint motion to a target (non-linear path)
`MoveC`	Circular arc motion through a via-point to a target
`MoveAbsJ`	Absolute joint motion to specified joint angles
`Set`	Set a digital output signal to high
`Reset`	Set a digital output signal to low
`WaitDI`	Wait until a digital input signal reaches a specified value

Evaluation Metrics

$P@K$	Precision at K: fraction of top-K retrieved chunks that are relevant
$R@K$	Recall at K: fraction of all relevant chunks appearing in the top-K results
MRR	Mean Reciprocal Rank: average of 1/rank of the first relevant result

Contents

List of Figures

List of Tables

1
Introduction

1.1 Case background

This thesis is part of the "Code Agents: AI powered end to end solutions for flexible manufacturing" project[1], which aims to improve manufacturing systems by integrating Artificial Intelligence and automation, with a focus on robotics and Programmable Logic Controllers (PLCs). The project's goal is to help manufacturers reach higher productivity, lower costs, and better quality through AI technologies.
A central part of this effort is the collaboration with a leading Nordic vehicle manufacturer. This manufacturer faces considerable challenges with the flexibility of its production lines. Although the industrial robots are mechanically capable of complex tasks, programming them remains a bottleneck. The current workflow depends on static programs that need manual updates and adjustments, a process that is slow, labor intensive, and often unable to keep up with the demands of high mix and low volume production. The manufacturer wants to make this programming process faster and more adaptive.
To solve these issues, there is growing interest in deploying "Code Agents," semi autonomous software systems that can generate and validate code. Given the recent progress of Generative AI (GenAI), using GenAI to support these Code Agents is a promising direction. GenAI can potentially overcome current limitations by letting devices learn from programming data, adapt to changing environments, and support better collaboration between humans and automated systems.

1.2 Problem description

Although GenAI has shown strong capabilities in software engineering, a significant gap remains when it comes to industrial robotics. Large Language Models (LLMs) perform well at generating code for general purpose programming languages such as Python or Java [1, 2], and recent evaluations show that models like GPT-4 can pass university level programming assessments [3]. These languages are widely used, and the models have vast amounts of training data covering their syntax and logic. In these domains, AI assisted code generation is becoming routine.
A critical research gap exists, however, when these models are applied to domain specific industrial languages such as RAPID, used by ABB robots [4]. Prior work has explored grounding LLMs in robotic affordances [5] and translating natural

[1] `https://research.chalmers.se/en/project/11944`

language into executable robot policies [6], while other efforts have targeted behavior tree generation [7], collaborative robotic development workflows [8]. Despite these advances, industrial robot programming involves specialized syntax, strict safety constraints, and complex interaction with physical hardware, features that set it apart from general robotic control. Standard LLMs lack this specialized knowledge and cannot guarantee that the code they generate is executable or safe. They are prone to "hallucinations," producing code that may look syntactically correct but is functionally invalid or dangerous when run on real hardware.
The first challenge is how to combine the general capabilities of GenAI and the specialized industrial robot programming. A method is needed that grounds the generation process in verified domain knowledge so that the produced code is both syntactically and semantically correct.
But even a perfectly grounded code generation system does not solve the full problem. A second challenge is the validation gap. Industrial robot code interacts with physical hardware whose behavior depends on kinematic constraints, joint limits, singularity conditions, and collision geometry. Verifying correctness currently requires a human expert to manually load the code into a simulation environment, run it, and interpret the results. This manual verification loop is slow and puts a hard limit on the system's ability to self correct iteratively [9]. Closing this loop through automated, programmatic interaction between the AI agent and the simulation environment is not merely a desirable extension but a necessary requirement for a truly autonomous code generation pipeline.

1.3 Purpose and aim

The purpose of this thesis is to enable the automated generation and verification of valid, executable industrial robot code using GenAI, with the goal of making manufacturing processes more flexible and efficient for the partner manufacturer.
The aim is to design and evaluate a complete agentic pipeline that addresses both the knowledge gap and the validation gap identified above, drawing on recent multi agent approaches to industrial code generation [10, 11].

1.4 Research questions

To evaluate this pipeline, the following RQs have been formulated:

RQ1. How can a Retrieval Augmented Generation (RAG) architecture be designed to effectively support the generation of executable RAPID code for industrial robots?

RQ2. To what extent does agentic planning and validation improve code quality beyond a standard RAG baseline?

RQ3. How can automated simulation based verification close the validation loop for industrial robot code?

1.5 Delimitations

This study focuses on the generation of RAPID code for ABB industrial robots, since this is the language used by the case company. While the proposed framework may apply to other languages (e.g., KRL or PLC code), validation across different platforms is outside the scope of this thesis. Generated code is validated through both static syntax checking and automated simulation via a custom Model Context Protocol (MCP) server connected to the ABB RobotStudio virtual controller; physical deployment on real hardware was not performed due to safety constraints and limited access to production equipment. The MCP server targets the ABB RobotStudio environment specifically, and its generalization to other simulation platforms is not addressed here.

1.6 Thesis outline

This thesis is organized as follows. Chapter 2 covers the theoretical background, including industrial robotics programming, the evolution of LLMs, RAG frameworks, and agentic AI architectures. Chapter 3 describes the system architecture, knowledge base construction, agentic planning, structured code generation, and the verification and validation process via a custom MCP server. Chapter 4 reports the experimental results across retrieval evaluation, code generation quality analysis, simulation based verification experiments, and a detailed case study. Chapter 5 discusses the findings in relation to the research questions, identifies limitations, and outlines directions for future work. Chapter 6 concludes the thesis with a summary of contributions and key findings.

2
Theoretical background

This chapter covers the background the rest of the thesis relies on. The topics range from industrial robotics and the RAPID language to RAG and agentic AI.

2.1 Industrial robotics development

Industrial automation runs on control systems that must be precise [12]. Software here is not like everyday programming. Here, timing is critical. A signal late by a few milliseconds can damage a product or a machine. Robots have also come a long way. The old ones were isolated arms bolted to the floor. Today's robots are plugged into the plant's network, they read sensors and PLCs directly, and they show up well beyond the car industry; electronics lines and logistics warehouses run them too [13].

2.1.1 Major vendors and closed standards

The global industrial robotics is dominated by four major manufacturers—ABB, KUKA, FANUC, and Yaskawa [14]. Each has developed its own robotic programming languages, and they don't share the same programming languages and control standards. KUKA robots run on KUKA Robot Language (KRL), FANUC uses KAREL and TP, and ABB systems rely on the RAPID programming language [15]. This lack of standardization creates barriers to solve problems, since expertise in one system does not transfer easily to another.
Robotic systems are generally categorized by how they interact with human operators [16]. Traditional industrial robots are designed for high speed and heavy payloads, and they must be physically isolated from workers for safety. Collaborative robots, or cobots, represent a move toward shared workspaces where machines and humans work side by side [17]. This thesis focuses primarily on the programming requirements of industrial manipulators, where software errors can cause serious physical harm, though the principles discussed are increasingly relevant to the expanding cobot sector.

2.1.2 ABB platform and digital twins

Founded in 1988 through the merger of ASEA and BBC Brown Boveri [18], ABB Robotics has built a strong presence in industrial automation. The performance of these robotic systems rests on advanced controller platforms, which have evolved

from the legacy IRC5 architecture to the modern OmniCore platform [19]. The controller runs programs, processes sensor signals, and coordinates motion axes with path accuracies that can reach sub millimeter precision.
A special element in modern industrial development is the Digital Twin [20]. In the ABB system, this digital twin is RobotStudio, a simulation environment built on VirtualRobot Technology [21]. RobotStudio runs an exact emulation of the real robot controller software, so virtual simulations replicate the functional behavior, kinematics, and cycle times of the physical system. This offline programming capability lets engineers design, validate, and optimize robotic workcells virtually without interrupting production. The ability to verify logic and motion paths digitally is important for reducing risk before code is deployed to physical hardware.
Digital twins are also becoming part of broader AI supported maintenance workflows. Chen [22] frames digital twins and AI as enablers for moving from predictive maintenance toward prescriptive maintenance, where models not only detect or forecast problems but also support actionable maintenance decisions. Related work on Transformer based remaining useful life prediction and fault diagnosis shows how sensor data can be used to jointly estimate machine health and classify fault conditions [23]. Although this thesis focuses on robot code generation rather than maintenance decision making, these studies show the same industrial trend: AI systems need reliable digital representations of equipment, operational data, and verification mechanisms before they can be useful in production environments.

2.1.3 Industrial robot programming languages

Industrial robots are usually programmed with their manufacturer's specific languages [15]. In practice, this means that a program written for one brand of robot normally cannot be moved directly to another brand without rewriting or adapting it. The main languages used in industry also differ quite a lot in style and in the tools used to write them.
KUKA robots use KUKA Robot Language, or KRL. A typical KUKA program is split into a `.src` file for the actual executable code and a `.dat` file for stored data such as positions. KRL is useful for common robot motions, including point to point and linear movements, and KUKA also provides inline forms to make these motions easier to program.
FANUC is slightly different. Simple operations are often written as Teach Pendant programs, which can be created directly on the robot controller.
Universal Robots uses URScript, which is closer to Python in its appearance. It is designed to be relatively easy to read and edit, while still giving access to instant robot control. For example, programmers can use both high level motion commands and lower level robot commands when you need more precise control.
Although these languages look different, they share several underlying principles. The robot code must be validated in simulator before it can run on real hardware. Table 2.1 summarises the main vendor languages discussed above.

Table 2.1: Comparison of major industrial robot programming languages.

Vendor	Language	Syntax style	Program unit	Typical use
ABB	RAPID	Pascal-like	`.mod` / `.sys` modules	Industrial manipulators
KUKA	KRL	Pascal-like	`.src` + `.dat` pair	Standard motions, inline forms
FANUC	TP	Teach pendant	`.tp` (on-controller)	Pick-and-place, basic ops
FANUC	KAREL	Pascal-like	`.kl` compiled module	Complex logic and I/O
Universal Robots	URScript	Python-like	Script file	Cobots, real-time control

2.1.4 The RAPID language

RAPID is the high level programming language developed by ABB for controlling their robotic systems [4]. It shares some syntax similarities with structured languages like Pascal, but is strictly domain specific and enforces formal semantics to ensure operational safety.
The language has a modular architecture. A application usually consists of one or more modules, which contain data declarations and executable routines. Within a module, logic is organized into three routine types: procedures, which define sequences of actions; functions, which return values; and trap routines, which handle interrupts for reactive behaviors.
Safety is the overriding constraint. RAPID manages operational risks through deterministic error handling mechanisms [24]. Unlike general software, where an unhandled exception might crash an application, a fault in a robot program can cause physical collisions and may not be detected in the IDE before the process is simulated. This rigidity presents a particular challenge for LLMs, which must produce code that is not only syntactically perfect but also semantically compliant with the physical constraints of the robot and its environment.

2.2 LLMs in engineering

Generative AI has made automated code generation much more practical [25]. But the idea did not appear suddenly. Today's LLMs are built on many years of work in natural language processing and sequence modeling.

2.2.1 Evolution of generative AI

Before Transformers, many NLP systems were based on Recurrent Neural Networks (RNNs), including Long Short-Term Memory (LSTM) networks [26]. These models read text step by step and use a hidden state to carry information from earlier parts of the sequence. For short inputs this work reasonably well, but for longer context the model has trouble remembering earlier information. Training was also slow because each step depended on the previous one, which made parallel processing difficult.

The Transformer architecture [27] changed this approach. Instead of using recurrence, it uses self-attention, which allows the model to compare different tokens in a sequence directly. The model can pay attention to relevant words even when they are far apart. Transformer is also much easier to train in parallel, which made possible to build much larger models on much larger datasets. Later, models were trained first on large amounts of unlabeled text and then adapted to specific tasks, including code generation [28].

2.2.2 In context learning

One useful ability of modern Transformer models is in context learning [29]. Instead of retraining the model for every new task, the user can give examples directly in the prompt, and the model uses those examples as a guide for its answer.
This is useful for robot programming. If verified RAPID code snippets are included in the prompt, the model can follow their structure, naming style, and error handling patterns. The prompt acts as a short reference manual. The model is not learning new parameters, but it can still adjust its output based on the examples in the context window.

2.2.3 Hallucination problem

LLMs can still produce hallucinations [30], which means they may generate information that sounds correct but is actually wrong. In normal software development this might appear as a fake library function, an incorrect parameter, or an API call that does not exist. These errors are already a problem in regular programming, but they become more serious in industrial robotics.
A robot program is connected to physical hardware. If the model invents a motion command or uses the wrong structure, the result may not just be a compiler error. It could also cause the robot to move in an unsafe or unexpected way. So LLM generated robot code should not be trusted by itself. It needs to be checked against reliable documentation, verified examples, and safety rules before being used on a real controller.

2.3 Retrieval augmented generation

RAG [31] improves LLMs by retrieving relevant documents from an external knowledge base and conditioning generation on the retrieved context. This bridges the gap between the frozen parametric memory of the model and the specific technical requirements of the task. RAG has been applied to various specialized domains, including field robotic operations where retrieval based language models ground task execution in operational compliance requirements [32]. Figure 2.1 summarises the standard RAG pipeline: user queries are matched against a knowledge base built from pre-chunked, embedded, and indexed source documents, and the retrieved passages are concatenated with the query before being passed to the generator.

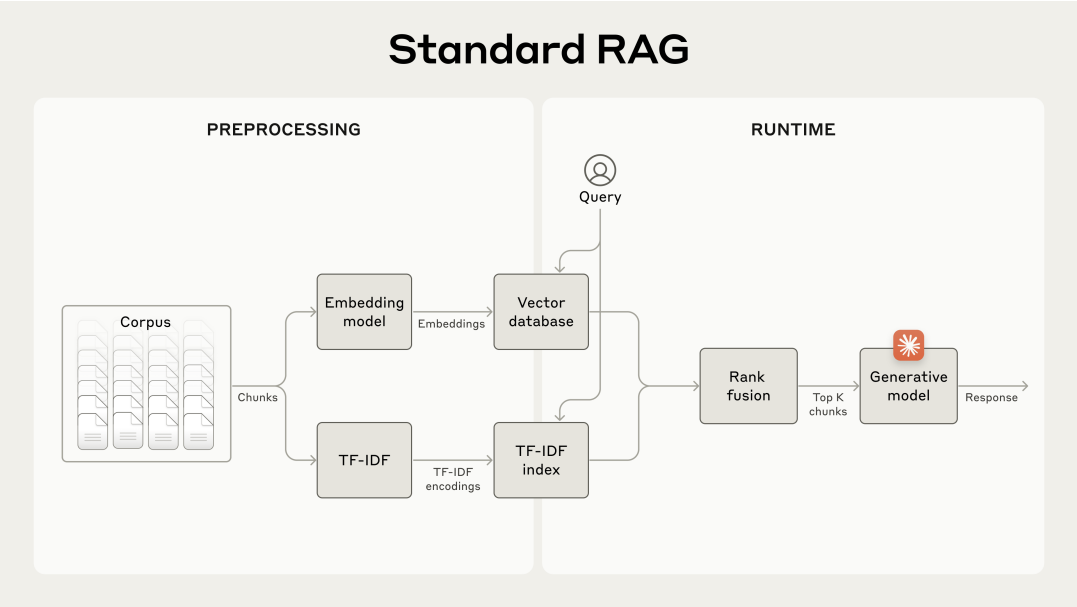


Figure 2.1: Standard RAG pipeline. Source: Anthropic [33].

2.3.1 Layout aware knowledge ingestion

Good retrieval starts with accurate data ingestion. Technical documentation in robotics typically has complex layouts: multi column text, hierarchical headers, and nested parameter tables [34]. Traditional optical character recognition or simple text extraction often linearize this content, destroying the semantic relationships between elements.

Document Layout Analysis bridges raw pixel data and high level semantic understanding. It split the process into physical layout analysis for spatial partitioning and logical layout analysis for assigning semantic roles such as titles and paragraphs. Modern approaches use multimodal frameworks that jointly model text, layout, and visual features [35]. This supports layout aware chunking, where documents are partitioned based on logical boundaries rather than fixed character limits. When a segment of text is retrieved, it keeps its original structural context, so the model can interpret technical parameters accurately.

2.3.2 Comparative analysis of RAG versus fine-tuning

When deploying generative AI for specialized domains, there is a choice between model fine tuning and RAG [36]. Fine-tuning updates the internal weights of a pre trained model on a domain specific dataset. It can internalize the syntax and style of a language like RAPID, but in an industrial setting it has several drawbacks. Facts learned during fine tuning are static; updating the model for new robot controller versions requires expensive retraining. Fine tuning is also susceptible to catastrophic forgetting [37], where the model lose its general reasoning abilities as it overfits to the new data. RAG keeps the model frozen and inject knowledge at inference time, which gives better traceability and easier maintenance.

2.3.3 Vector embedding in Semantic Space

The core of the retrieval system is vector embeddings [38]. Embedding models transform discrete text tokens into continuous vectors in a high dimensional space. The basic assumption is that the geometric distance between two vectors can reflect their semantic relationship.
The Similarity between a user query vector and a document chunk vector is often measured by Cosine Similarity, which is defined as the cosine of the Angle between the two vectors [39].
Unlike Euclidean distance, cosine similarity focuses more on direction rather than length. This is particularly useful in text retrieval because of its ability to normalize differences in document length. In this way, a short query can be successfully matched as long as it is semantically consistent with a longer paragraph.

2.3.4 Approximate nearest neighbor search

Computing the cosine similarity between the query vector and each stored vector would be prohibitively expensive in a real system with thousands of document chunks. To solve this problem, vector databases often use Approximate Nearest Neighbor (ANN) algorithms. Among them, Hierarchical Navigable Small World (HNSW) [40] is a standard method.
HNSW constructs a multi layer graph structure: the upper layer is used for long distance jumps in the whole data space, and the lower layer is used for accurate local search. With the help of this structure, the search algorithm is able to locate the relevant region in the vector space in logarithmic time.

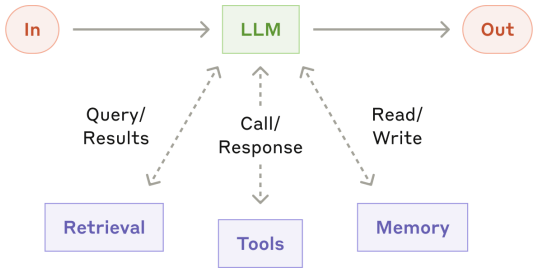
2.3.5 Hypothetical document embedding

A long standing problem in dense retrieval is the lexical mismatch between short, informal user queries and the more technical language in the indexed documents. Hypothetical Document Embeddings (HyDE) [41] attempts to solve this problem.
This approach first uses a language model to generate a hypothetical answer to the query, and then encodes this synthetic document as a retrieval key. Since this generated text is much closer to the target document in terms of vocabulary and structure than the original query, HyDE is able to improve the precision of zero shot retrieval without relevance labeling training data. This technique is particularly well suited to bridging the semantic gap between user requests, which are often formulated in natural language, and knowledge bases, which use formal instruction syntax, for industrial programming.

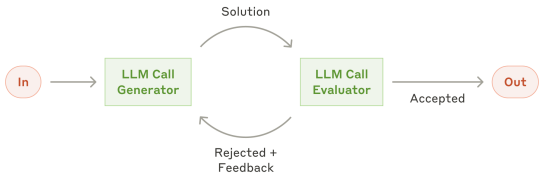
2.4 Agentic AI architecture

Standard RAG systems typically retrieve first and then generate. But complex engineering tasks often require iterative reasoning. Therefore, researchers have proposed agentic AI architectures, in which the model is no longer simply generating answers

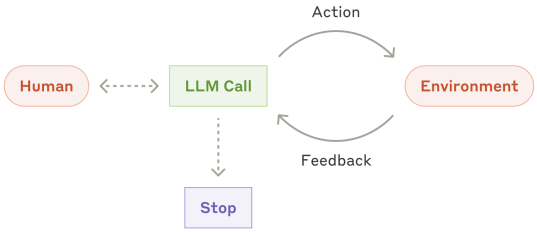
but acts as a reasoning engine with planning, tool use, and self correction capabilities [42]. Figure 2.2 illustrates three representative patterns in this evolution, from tool augmented LLMs all the way to fully autonomous agents [43].



(a) Augmented LLM: a single model enhanced with retrieval, tools, and memory.



(b) Evaluator-optimizer: one model generates output while a second provides iterative feedback.



(c) Autonomous agent: independent operation with tool use and environment feedback in a loop.

Figure 2.2: Three representative agentic architecture patterns, progressing from a tool augmented LLM to an autonomous agent with environment feedback [43].

2.4.1 From passive RAGs to active agents

Passive systems typically respond to queries with a single retrieval step. An active agent, on the other hand, views a query as a goal to be accomplished. It dynamically determines which tools should be used and whether to retrieve documentation, look for code templates, or both based on the complexity of the request [44]. It is the transition from static pipeline to dynamic workflow that makes the system better able to deal with ambiguities in user requirements.

2.4.2 Planning and decomposition

For complex requests, one retrieval is often not enough. Planning agents use Chain of Thought (CoT) [45] to break high level goals into executable subtasks. For example, a planner might first decompose the request into several logical steps: first retrieve the signal configuration and then search for the motion logic template. Such a decomposition enables the system to perform targeted retrieval of different aspects of the problem, thus providing a more complete context for the generation phase.

Figure 2.3 illustrates this pattern: the output of one LLM call becomes the input to the next, and optional check gating can be set between steps.

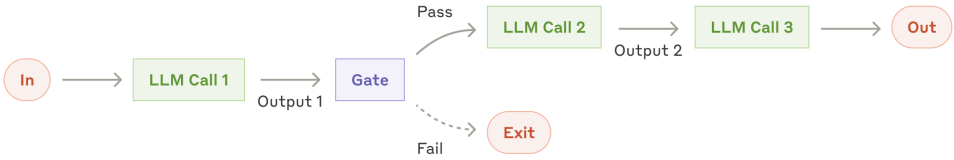


Figure 2.3: Prompt chaining decomposes a task into a fixed sequence of LLM calls, with optional programmatic gates between steps to verify that intermediate outputs meet expected criteria [43].

2.4.3 Self correcting with a validation loop

An important weakness of generative models lies in their inability to actively verify facts during generation. The Reflexion framework [46] proposes an approach to "linguistic reinforcement learning" that enables agents to critique and reflect on their own output. For robot programming, this translates to a Validation Loop: after code generation, a dedicated verification agent checks the output against predefined constraints. If the validation fails, the agent feeds this feedback into the generative model, prompting it to try again.
This iterative loop significantly reduces syntax errors and helps ensure that the generated results conform to project specific templates and specifications.

2.4.4 Tool Usage Paradigm for LLM Agents

LLM agents are not limited to generating text. They can also call external tools and thus interact with software systems, files, apis, and even physical processes. In industrial robotics scenarios, the most valuable external tool is usually the simulation environment. If an agent could upload robot code to a simulation controller, run the program, and read execution feedback, a lot of work in the testing pipeline could be automated.
However, there is not only one way that LLM agents can access tools. Different ways affect the maintainability of the system and whether the system can clearly "understand" the tool itself.
The simplest method is a command line call. In this setting, the agent generates shell commands and then reads the text output. This approach is easy to implement, but also very fragile. The inputs and outputs are just plain text, so the model doesn't really know what input each tool needs, and there's no built in mechanism to discover which tools are available. The error handling capability is also limited, as failures are usually just in the form of terminal output or standard error messages. As a result, each tool integration often requires custom development, and even small changes in tools may require modifications to the surrounding integration code.
A more structured approach is platform native function calls. In this approach, the LLM provider provides the model with a list of available functions, typically including the function name, parameters, and the expected output format. During

the generation process, the model can request to call a certain function according to this predefined schema. This approach is more reliable compared to the original command line approach because the model works with typed input rather than free text. However, this approach is usually tied to a specific vendor's API. Function definitions written for one platform may not be directly transferable to another. In addition, adding or changing tools often requires changes to the application code that manages the list of functions.
This thesis adopts the third approach, Model Context Protocol (MCP). MCP uses a client server structure to separate the LLM application from the actual tool implementation. In this design, the tools are exposed by the MCP server, while the agents are connected as clients. This makes the tool layer more modular: agents can dynamically discover available tools, and it is possible for the same tool server to be reused by different LLM clients. For this project, this separation is particularly useful because the robot simulation environment can be treated as an external service and does not have to be hard coded into the function call system of a particular model provider.
Figure 2.4 illustrates a common pattern of function calls in many LLM systems. The process starts by defining the available tools and then having the model select a structured tool call and execute the tool outside the model. The result is then returned to the model, which generates the final response based on the result.
Previous studies also support the view that tool usage is an important ability of language models. Schick et al. [48] showed that models can be trained to learn when and how to use external tools, rather than just relying on hand written prompts. Qin et al. [49] provide a more comprehensive review of tool learning for foundational models, including systems that use tools, learn around tools, and even create tools. This work mainly falls into the category of tool augmented, as the model itself is not retrained, but its capabilities are extended by accessing the robot simulation environment during the inference phase.

2.4.5 Model context protocol

MCP [50] is an open interface for connecting LLMs to external tools and data sources. It follows a client server architecture: the AI assistant acts as the host running one or more MCP clients, and each client maintains a connection to a dedicated MCP server that exposes a set of typed tools. The protocol defines a lifecycle with four phases: initialization, discovery, invocation, and shutdown. This means the agent does not need to embed knowledge of what tools exist; it discovers them at runtime and adjusts its behavior.
Another property of MCP is its vendor and model neutrality. The protocol only defines the communication contract. It does not specify what implementation language to use, which deployment model to pick, or which AI provider to connect. So the same MCP server can be used by different LLM clients without modification. This matters for industrial environments where the AI model may change over time but the integration with shop floor systems must stay stable.
Figure 2.5 shows the general MCP architecture: a single host connects to multiple servers, each exposing its own set of tools [51].

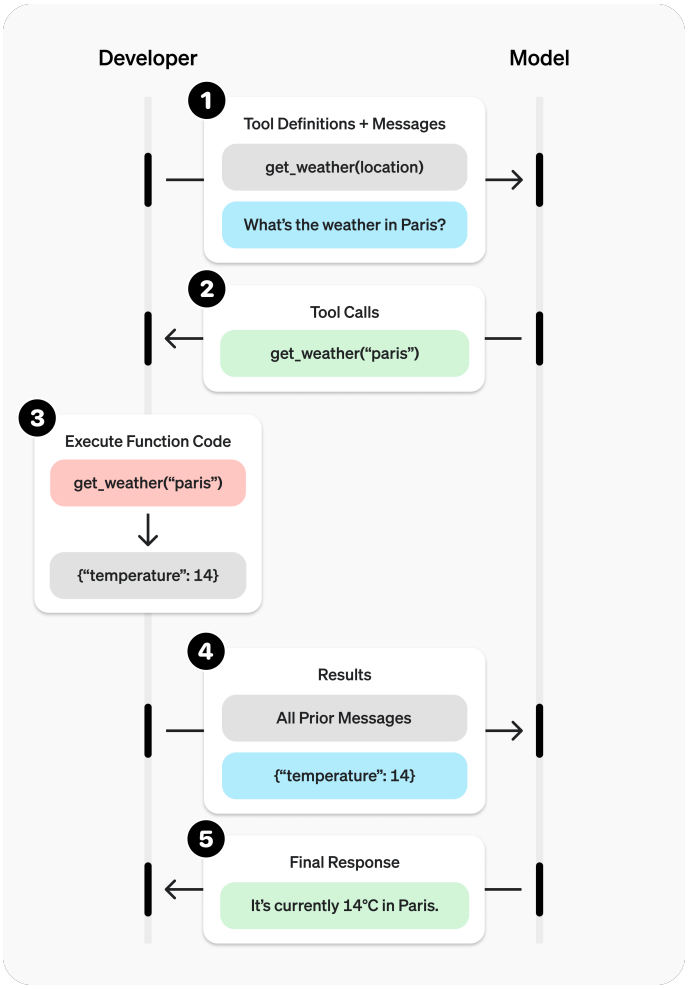


Figure 2.4: The function calling interaction cycle between a developer application and an LLM. The model receives tool definitions alongside the user query, emits a structured tool call, and incorporates the execution result into its final response. Source: OpenAI [47].

For this thesis, the choice of using MCP came from two directions. One was a GitHub repo where someone built a small MCP server for RobotStudio that only handled uploading a RAPID module. Because RobotStudio also has a very complete SDK toolkit, there appeared to be much more that could be done through AI — reading back controller error logs, monitoring execution status, and so on. Around the same time Claude released this MCP concept, which further increased the relevance of this approach.
But in practice the implementation was not smooth. At the beginning, the server could only upload a module and run it. Later, it became clear that the controller error logs could be retrieved too, and this made the feedback loop actually useful for iterative correction. The main blocking problem was that RobotStudio could not directly connect to the MCP server. Considerable debugging was required, and the solution turned out to be that a plugin had to be written inside RobotStudio itself, so the external MCP server could reach the controller at all. Once this plugin was in place, the server started working as expected.

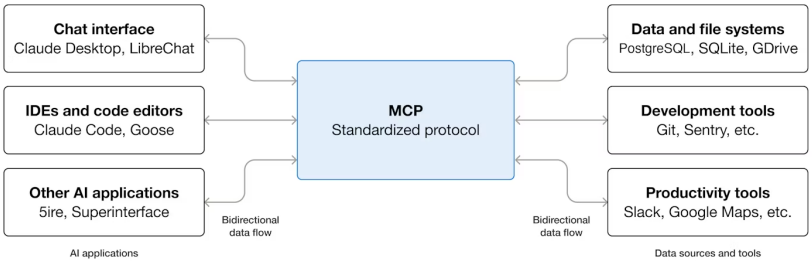


Figure 2.5: General architecture of MCP, connecting AI applications to data sources and tools through a standardized protocol [51].

2.4.6 Simulation in the loop verification

Using simulation as an automated verification layer for AI generated artifacts has precedent in several domains. In autonomous driving, the CARLA simulator [52] established the paradigm of testing AI generated driving policies against realistic traffic scenarios in a closed loop, where simulation outcomes feed back into policy refinement. Nouri et al. [53] applied this directly to LLM based code generation: their pipeline uses CARLA to automatically execute LLM generated vehicle control code, evaluates the result against safety scenarios, and feeds structured failure reports back to the model for iterative correction. This workflow is close to the one developed in this thesis for industrial robotics. In robotic manipulation, Huang et al. [54] showed that LLMs can compose 3D value maps that are executed and refined through interaction with a simulated environment.
These systems share a common pattern: generate an artifact (code, policy, or plan), run it in a simulation environment, extract structured feedback, and iterate. For this to work, the AI agent needs programmatic access to the simulator. It must be able to upload, execute, and query the simulation without manual intervention. For industrial robotics, the simulation environment (e.g., ABB RobotStudio) runs a certified copy of the real controller software. This checks generated code against the same kinematic model, joint limits, and collision geometry that govern the physical robot. MCP acts as the communication protocol in this thesis, and a high fidelity digital twin serves as the verification backend.

2.4.7 LLM applications in robot simulation

A growing body of research applies LLMs to robot programming and simulation. Table 2.2 summarizes the key approaches and their characteristics.
These studies span a spectrum from open loop code generation [6, 55, 57] to affordance grounded planning [5, 56] and simulation verified evolutionary optimization [60, 61]. In the industrial domain specifically, Li et al. [4] showed that general purpose LLMs frequently violate domain specific constraints when generating RAPID or KRL code, while Omkar et al. [59] demonstrated that RAG grounded approaches can improve the correctness of generated industrial robot programs.

Table 2.2: Summary of LLM based approaches to robot programming and simulation.

Work	Focus	Verification	Platform
Liang et al. [6]	Policy code from NL	None (open loop)	General
Vemprala et al. [55]	Control scripts via ChatGPT	Manual	General
Ahn et al. [5]	Affordance grounded plans	Affordance scoring	Mobile robot
Singh et al. [56]	Pythonic task planning	Simulated execution	VirtualHome
Wake et al. [57]	Manipulation sequences	Manual	General
Bennulf et al. [58]	LLM driven kitting	Physical execution	Cobot (UR)
Li et al. [4]	RAPID / KRL generation	Syntax analysis	ABB / KUKA
Omkar et al. [59]	RAG for robot code	Correctness evaluation	Industrial
Ma et al. [60]	Reward function evolution	IsaacGym simulation	RL tasks
Wang et al. [61]	Self guided skill learning	Sim. environments	RL tasks

Bennulf et al. [58] showed that natural language instructions can reconfigure a collaborative robot for kitting tasks without reprogramming.
Despite this progress, most systems remain open loop: they generate code but do not verify it against a simulated or physical environment [62]. Furthermore, the majority target general platforms (e.g., ROS based systems) rather than closed industrial environments [63], leaving the challenges of strict type systems, safety critical error handling, and vendor specific motion instructions largely unaddressed. This thesis addresses both gaps by combining RAG grounded code generation with an agentic architecture that interfaces directly with ABB RobotStudio through MCP. The simulation in the loop approach enables automated verification against the same controller software and kinematic model used in production [64], positioning the contribution at the intersection of LLM based code generation, industrial robot programming, and simulation in the loop verification.

3
Methodology

This chapter describes the technical implementation of the proposed RAG system. Building on Chapter 2, the work develop a domain-specific solution for automating industrial robot code generation. The sections that follow cover the system architecture, knowledge base construction, planning and retrieval, structured code generation, and the multi level verification pipeline. The main contribution of this thesis sits in the last of these: a custom MCP server that links the agent directly to ABB RobotStudio, so uploading code, running it, and reading back diagnostic feedback all happen automatically.

3.1 System overview and architecture

The methodological contribution of this thesis is a modular, multi agent orchestration framework. Instead of direct prompt to code generation, the system implement a structured pipeline that separates task planning, knowledge retrieval, code synthesis, and verification into distinct interacting components.

3.1.1 Architectural design

Because industrial robotics is specialized and safety critical, the system architecture is designed to minimize hallucination through strict grounding. The framework retrieves first and then generate, with an autonomous planning layer that decomposes complex user requests into executable retrieval sub-tasks.
Figure 3.1 shows the workflow. A user query first goes through a processing module, then a dual stream retrieval engine, and finally the generation and validation loops. The backend is a Python service, and it connect to the Azure OpenAI LLMs [65] and a local persistence layer that holds the vectorized knowledge base. Azure OpenAI is used here mostly because the larger research project to which this thesis belongs already uses Azure, so API access and credit were already set up.
RobotStudio [21] is not only a simulation tool in this system, it also act as part of the validation. The code generation and reasoning happen in the Python backend, but the functional verification actually run inside the RobotStudio virtual controller, which has the real kinematic constraints of the robot.

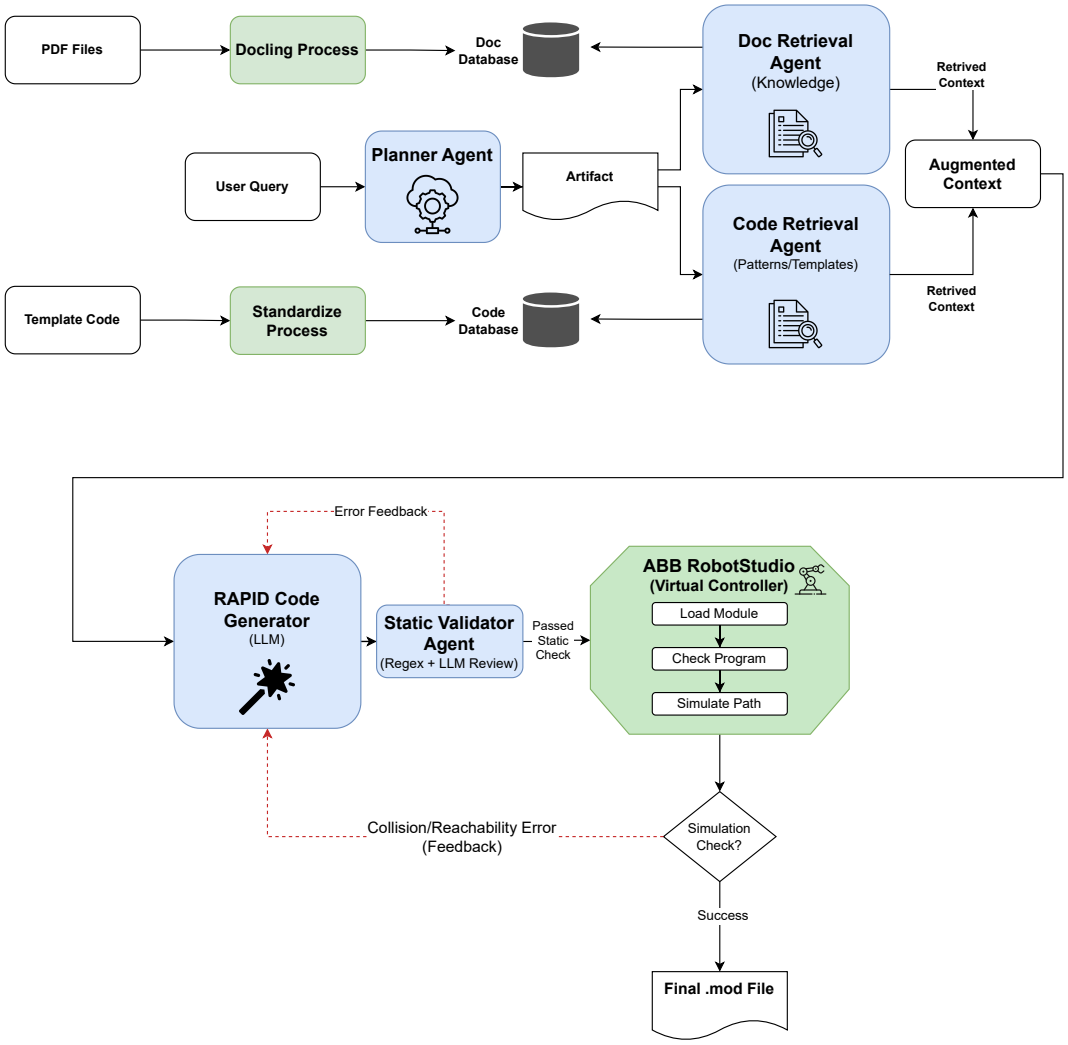


Figure 3.1: The proposed multi agent system architecture.

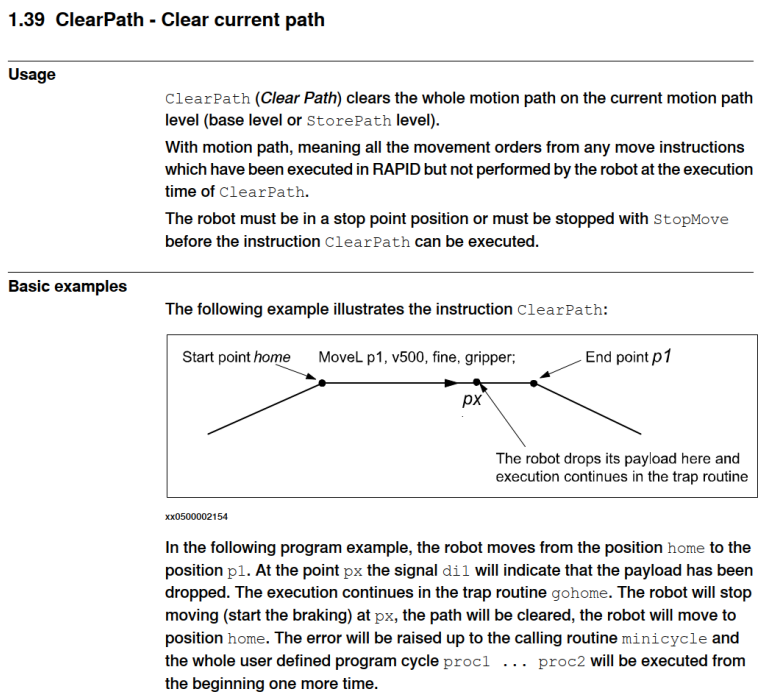
3.2 Knowledge base construction

The first operational phase establishes the system's long term memory. Industrial robot programming depend on vendor specific documentation for both the programming interface and its operational semantics. The primary source of unstructured knowledge in this thesis is the official technical reference manual for the ABB RAPID language [66], which serves as the ground truth for instructions, functions, and data types.

The manual is over 1,700 pages long, which poses significant challenges for automated processing. It is organized into four parts: Instructions, Functions, Data types, and Programming type examples. The content range from simple arithmetic operators like Add to motion control primitives like MoveL and sensor integration routines like EGMActJoint. Each entry has strict syntactic and semantic constraints, so the ingestion pipeline need to be high fidelity.

3.2.1 Page oriented document processing

ABB's source documents are difficult to process not only because they are long, but also because the information in them is very structural. A typical entry may contain a short description, a formal syntax definition, a parameter list, error handling instructions, illustrations, and code examples. These parts are related to each other. For example, a parameter table is only meaningful if it remains associated with the instruction or function it describes.
This poses a problem for simple text extraction. If the PDF is directly converted to plain text line by line, the reading order may be lost, the table may be disentangled, and the example may be separated from its associated instructions. In RAG systems, this leads to poorer retrieval, as the model receives only a portion of the information it needs.
To avoid this problem, this paper uses Docling as the document processing layer [67]. Docling analyzes the layout of each page and identifies elements such as headings, paragraphs, lists, and tables. It also restores a reliable reading order before the text is stored. This is particularly important for technical manuals, where the position of a table or block of code often affects its meaning.



1.39 ClearPath - Clear current path

Usage

`ClearPath` (*Clear Path*) clears the whole motion path on the current motion path level (base level or `StorePath` level).

With motion path, meaning all the movement orders from any move instructions which have been executed in RAPID but not performed by the robot at the execution time of `ClearPath`.

The robot must be in a stop point position or must be stopped with `StopMove` before the instruction `ClearPath` can be executed.

Basic examples

The following example illustrates the instruction `ClearPath`:

xx0500002154

In the following program example, the robot moves from the position `home` to the position `p1`. At the point `px` the signal `di1` will indicate that the payload has been dropped. The execution continues in the trap routine `gohome`. The robot will stop moving (start the braking) at `px`, the path will be cleared, the robot will move to position `home`. The error will be raised up to the calling routine `minicycle` and the whole user defined program cycle `proc1 ... proc2` will be executed from the beginning one more time.

Figure 3.2: Representative excerpt from the RAPID technical reference manual.

A practical feature in the process is table reconstruction. ABB manuals often contain parameter tables with merged cells, multi column structures, or sparse bounding boxes. If these tables are not handled properly, details such as parameter names, data types, and descriptions can get mixed up. By keeping the table in a structured format, the system is able to retain this information for subsequent retrieval and code generation.
PyMuPDF was initially used when the manual was first imported. It works well

on plain text pages, but it doesn't preserve the layout structure of the parameter tables and merged cells in the RAPID manual. After discussion with an industry expert, Docling was suggested instead. Docling turns out to be better suited for such documents, not least because it supports structure based chunking and respects tables and chapter boundaries rather than randomly slicing content into scattered fragments.

3.2.2 Semantic chunking and metadata retention

After the completion of the page-oriented processing, the document also needs to be divided into smaller units for retrieval. A simple split based on character count was not used because that would have cut the definition of an instruction in the middle or separated the syntax from its example. In an earlier version of the experiment with a fixed character splitter, the MoveL instruction was truncated in the middle, returning only the first half of the argument list. Therefore, the document is then chunked according to its natural structure [68].
In practice, this means that related information is kept together as much as possible. For example, a retrieval block might also contain the name, syntax, parameter description, usage description, and a short example of a certain RAPID instruction. In this way, the retrieved context is much more useful for the code generation stage, because the model receives the complete specification instead of the separated pieces. The system also retains metadata for each chunk. This metadata can include information such as the source document, section titles, page number locations, and document element types. Metadata makes the retrieval process easier to inspect and also helps to trace the generated answers back to the original document. This traceability is important because the generated code should be based on a reliable technical source, rather than relying solely on existing knowledge within the model.

3.2.3 Structured data processing: RAPID template library

To complement the unstructured documentation, the system also ingests the RAPID template library. Two sources were used for this. The first was a set of production templates from the industrial partner involved in the project. These have strict format and naming conventions. The second was student practice RAPID files provided by supervisor Omkar from his teaching. When the generation was first tested with the industrial templates, the output code had the right shape on the surface but the logic inside was often wrong. Omkar's simpler student files were then added to the index as well. The final retrieval is weighted to prefer these simpler files for generation. Each file still gets a label from its parent directory name, for example welding or gripping, so a welding query land on welding templates.

3.3 Agentic planning and dual retrieval strategy

3.3.1 Intention recognition and task decomposition

In order to connect user requests with the retrieval system, this paper uses an LLM based planning module called PlannerAgent. Instead of sending the user's query directly to the vector database, it analyzes the request to determine what type of information is needed and then converts it into one or more `RetrievalTask` objects that can be executed.
Each `RetrievalTask` represents a smaller search problem. For example, one task might be used to find the RAPID motion grammar, another to retrieve examples of `tooldata` or `wobjdata`, and yet another to search for error handling patterns. This makes the search more focused when the user's request contains more than one request.
This design follows the general idea of layered retrieval, which dynamically adjusts the retrieval process according to the query complexity [69]. Simple requests may only require a single search, while more complex programming tasks are split into multiple targeted searches. PlannerAgent also fits into multi agent code generation methods that are more widely used in engineering, such as geospatial analysis [70] and manufacturing [11]. In the present paper, its main role is to act as a bridge between natural language task planning and the underlying embedding based retrieval.

3.3.2 Dual retrieval mechanism

Industrial robot programming requires both conceptual level understanding (from technical manuals) and syntactical level reference (from code templates). To meet these two different requirements, a dual search strategy is adopted to decouple the search space into two parallel pipelines: one is the document stream for semantic knowledge, and the other is the code sample stream for syntactic patterns.
After the initial vector retrieval, the system uses a domain adaptive reranking algorithm to further optimize the results [71]. This post-retrieval phase employs a heuristic weighting mechanism to prioritize different content types according to user intent. If the query contains Rapid specific keywords (e.g., "MODULE", "PROC", "MoveL"), the system increases the similarity score of the code block, so that near production ready templates are preferred over theoretical descriptions in the context of the final prompt.

3.4 Structured code generation

3.4.1 Prompt engineering and context assembly

Once the search phase produces candidate blocks, the PromptTemplateManager takes over and assembles the actual input for the code generation model. It combines retrieved manual chapters, RAPID syntax instructions, verified code examples, and the user's original task into a structured prompt. The information must be organized

so that the model clearly understands the task requirements, the relevant ABB rules, and the examples it should follow.
Retrieved text snippets and code examples are processed in two separate streams, each truncated to a predefined token budget and formatted as a numbered list so that the model can refer to specific sources. These chunks are then assembled into a single template that clearly distinguishes between the document context, the code sample context, and the user query, with mandatory generation directives attached at the end [72]. Figure 3.3 shows this assembly process.

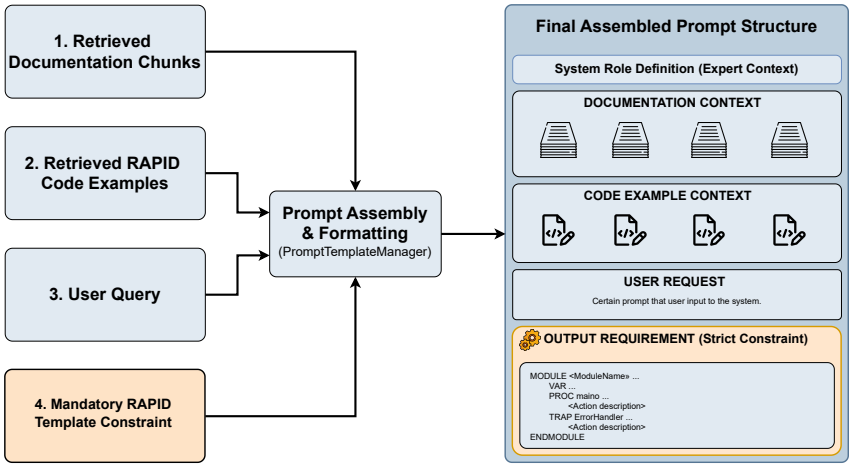


Figure 3.3: Schematic representation of the prompt assembly pipeline.

3.4.2 Standardize the output template

To ensure that the generated artifacts can be directly executed on the ABB IRC5 / OmniCore controller [73], the system enforces a "standardized RAPID module template". Unlike general purpose code generation, which often may output scattered code fragments or conversational markdown, this architecture imposes a strict syntactic skeleton.
The prompt manager injects a mandatory scaffolding that contains the key constructs needed for industrial automation: `MODULE` headers, persistent variable declarations, deterministic error handling traps, and a well defined entry procedure. The model is required to leave this overall layout intact, replacing only specific placeholders (e.g., `<Action description>`). This changes the code generation task from open ended text completion to a more precise template filling process.

3.5 Verification and validation process

3.5.1 Rule based static analysis

The ValidationAgent performs a rule based static scan immediately after generation: it verifies `MODULE`/`ENDMODULE` framing, checks for required `TRAP` handlers, and rejects any module still containing placeholder tags such as `<Action description>`. Physical feasibility, which this scan cannot cover, is then handed off to the simulation based verification described in Section 3.5.2, whose two component architecture is shown in Figure 3.4.

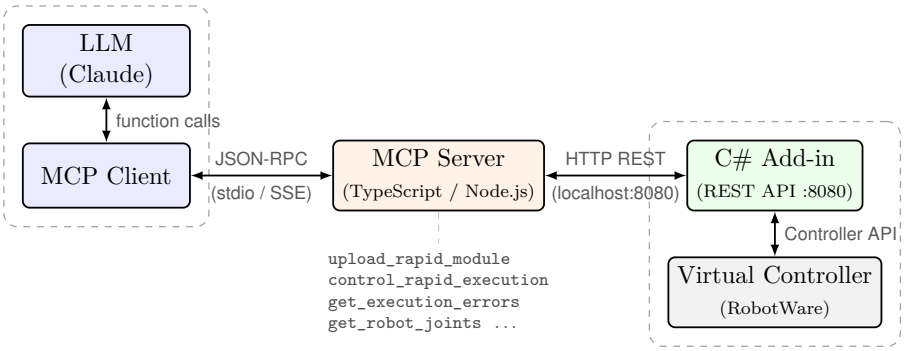


Figure 3.4: MCP based integration architecture.

3.5.2 Automated simulation based functional verification via MCP

The second level of validation implements an automated simulation loop that connects the AI agent directly to ABB RobotStudio through a custom MCP server [50]. The MCP server has a two component architecture. A C# add in, built on .NET Framework 4.8, runs inside the RobotStudio process and exposes a local HTTP REST API on port 8080 via a raw `TcpListener`. A TypeScript MCP server (Node.js) translates the standardized MCP tool invocations from the AI client into HTTP requests to this local API. This design avoids the need for Windows URL ACL permissions (which `HttpListener` would require) and lets the add in run without elevated privileges.

One subtlety surfaced early in the implementation: the RobotStudio SDK does not fully mirror the user interface. Pressing Stop in the UI halts the robot arm and also resets the scene back to its initial state, whereas the equivalent SDK call only halts the arm and leaves every object in the scene wherever the last run left it. If the server does not reset the scene explicitly between iterations, later runs start from drifted positions and fail for reasons that have nothing to do with the generated code. The open source RobotStudio MCP project by Seyser[1], which served as the starting point here, only supported code upload and program execution and did not address this at all. The version developed in this thesis adds an explicit scene

[1] `https://github.com/DavidSeyserGit`

reset step along with execution control, status polling, controller event log retrieval, RAPID declaration introspection, scene inspection with object bounding boxes, and I/O signal control. The resulting ten tools are summarized in Table 3.1.

Table 3.1: MCP server tools for automated simulation verification.

Tool	Description
`upload_rapid_module`	Writes a RAPID module to the virtual controller and loads it. Enforces BOM-free UTF-8 with CRLF. Deletes existing program modules (except `BASE` and `user`) before loading.
`control_rapid_execution`	Starts, stops, or resets the program pointer on the virtual controller.
`get_rapid_execution_status`	Returns the execution state (running, stopped, or error) and program pointer position.
`get_execution_errors`	Reads the controller event log and returns recent errors and warnings for diagnostic feedback.
`get_robot_joints`	Reads real-time joint positions (J1–J6) in degrees for target verification.
`control_simulation`	Starts or stops the RobotStudio simulation.
`get_station_status`	Returns station, simulation, and controller state.
`list_rapid_variables`	Enumerates the declared items (`VAR`, `PERS`, `CONST`) in RAPID, and supports optional type filtering to enable the discovery of objects such as work objects, tools, and targets at runtime.
`get_scene_objects`	Returns the scene graph, which includes the positions, orientations, visibility, and bounding box dimensions of the objects, for use in spatial reasoning.
`set_io_signal`	Writes digital or analog I/O signals to trigger actions of the Smart Component, such as generating the enclosure or controlling the conveyor belt.

Figure 3.5 illustrates the verification loop. For each generated RAPID module, the agent uploads the code, starts its execution on the virtual controller, polls the running status, and obtains runtime errors and the final joint configuration through the corresponding MCP tool. All the returned diagnostic information is appended to the generated context, so that the module can be corrected in the next iteration. One of the tools, `get_scene_objects`, was not included in the initial design. It was added later during the testing process after a recurring pattern was identified: a significant portion of the verification failures were not actually due to code errors, but rather scene drift. That is to say, the positions of the target points, fixtures, or workpieces were not at the positions assumed by the generated code. Once the agent could query the actual scene graph before generating motion code, this type of "false negative" problem basically disappeared.

3.5.3 Deployment Configuration of MCP Server

The MCP server is deployed on an IRB120 3/0.58 virtual controller within RobotStudio 2024 (School Edition) [21]. During the actual deployment process, some integration constraints beyond the architecture design were exposed. The RAPID parser identifies the byte order mark (BOM) as an illegal character, so all uploaded files must use UTF-8 encoding without BOM and CRLF line break format.

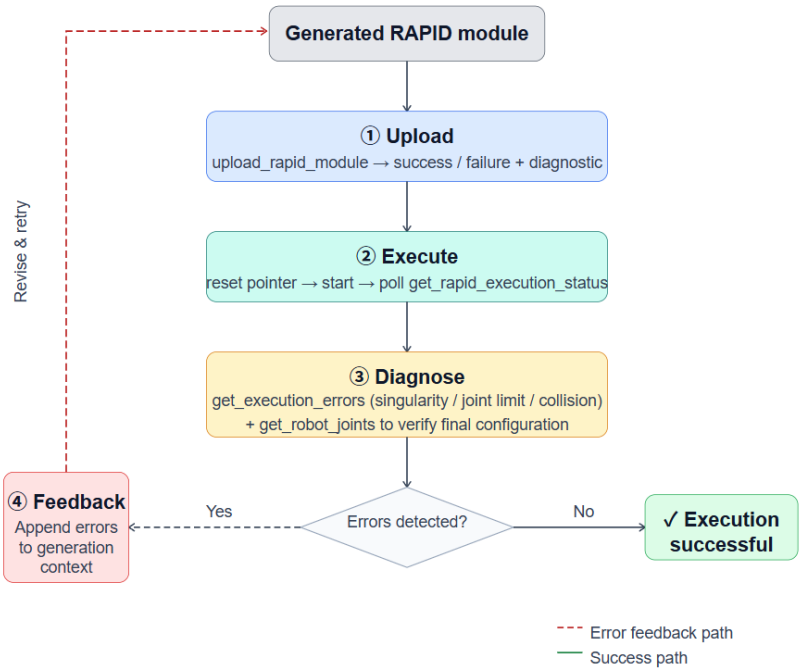


Figure 3.5: Automated verification cycle for generated RAPID modules.

The module upload also employs a destructive replacement strategy: Before loading the new module, all existing program modules (except the system module `BASE` and the user module) will be deleted to prevent the occurrence of the error "Global routine name main ambiguous". This error typically occurs when multiple modules define the `main()` procedure. The impact of this strategy on multi module programs will be discussed in Section 4.3.4.

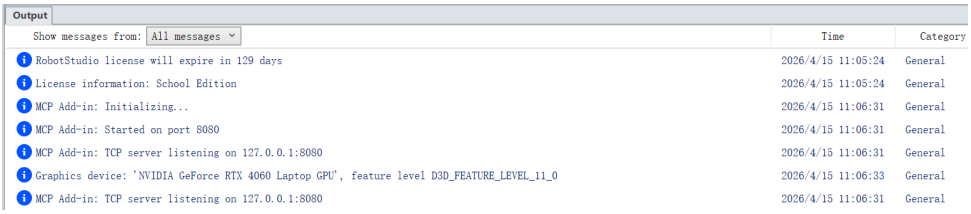


Figure 3.6: RobotStudio controller event log during an automated MCP verification session.

3.5.4 Expert Review

As the final supplementary step, domain experts will review the generated code to check for some implicit industrial best practices that automated simulations may not be able to detect, such as optimal path sorting, or whether they comply with specific on-site safety regulations.
Their feedback mainly focused on the following aspects: whether the structure was reasonable, whether it followed the specific naming conventions at the site, and

whether these generated modules were feasible for actual deployment under real operational constraints. Such expert evaluations bridge the gap between the content that can be verified through simulation and the requirements that must be met for actual deployment.

4
Results

This chapter reports the experimental results of the proposed RAG system. After describing the experimental setup, the evaluation is organized into two main parts: RAG based results covering retrieval performance and code generation quality, followed by a case study of the end to end workflow; and MCP based results covering simulation based validation through the custom MCP server integration with ABB RobotStudio. A summary of findings concludes the chapter.

4.1 Experimental setup

4.1.1 Dataset description

To evaluate the system, a test set of 21 code generation sessions was constructed. Each session consists of a natural language user query and the corresponding generated RAPID module. The task categories covered by these sessions represent common application scenarios encountered in industrial robotic arm deployments, including material handling, path following, tool management, and welding [12]. This coverage makes the test set well suited for validating the system under conditions that reflect real world industrial use. The queries were designed to span a range of complexity levels, from single procedure motion programs to multi procedure industrial library modules, and cover seven representative task categories common in industrial robot programming. Table 4.1 summarizes the distribution across these categories.
Several queries include domain specific constraints (such as specifying particular tool names (`PENNTCP`) or work objects (`REFRAM_RUTA1`)) to test the system's ability to incorporate user specified parameters into the generated code.

4.1.2 System configurations

To isolate the contribution of each system component, three configurations were evaluated.
The first, Pure LLM Baseline ($N = 5$), generates RAPID code by prompting the LLM (GPT-5.2 [74]) directly with only the user query and a minimal instruction to produce a valid RAPID module. No retrieval context, domain documentation, or code templates are provided. This configuration measures what a state of the art general purpose LLM can produce for industrial robot code without domain grounding. Five sessions were selected to cover distinct task categories: basic ma-

Table 4.1: Distribution of test sessions across task categories.

Task Category	Sessions	Description
Drawing & path following	8	Pattern tracing with `MoveL`/`MoveC`, custom work objects and tool definitions
Pick and place	5	Material handling with gripper actuation, approach/retract motions, I/O handshakes
Gripper tool libraries	3	Module level library generation for pneumatic gripper control, open/close sequences, wear tracking
Simple motion	2	Fundamental robot motion to target positions
Arc welding	1	Safety critical welding sequences with pre/post weld checks
Tool definition	1	Custom tool data with TCP coordinates and load parameters
Target based programming	1	Motion sequences from user provided `robtarget` coordinate data

nipulation, domain specific welding, error handling, conveyor tracking, and multi station assembly.
The second, Naive RAG ($N = 11$), performs dual stream retrieval (documentation and code examples) and generates RAPID code from the retrieved context, but without the Planner Agent's task decomposition or the post generation validation loop; this is the standard RAG baseline. The third, Full Pipeline ($N = 10$), implements the complete agentic architecture: the Planner Agent for query decomposition, dual retrieval with targeted sub-queries, structured prompt assembly, code generation, and the multi level validation loop combining static analysis with expert level review. Four test queries were run under both RAG configurations to allow direct, paired comparisons on identical inputs.

4.1.3 Evaluation metrics

System performance is assessed along four dimensions. The retrieval metrics used in this work are defined as follows. Precision@K is the fraction of the top-K retrieved chunks that are relevant to the query, computed as the number of relevant chunks in the top-K results divided by K; a high value indicates that the retriever returns few irrelevant results [39]. Recall@K is the fraction of all relevant chunks in the corpus that appear in the top-K results; a high value indicates that the retriever successfully surfaces most of the relevant material [39]. Mean Reciprocal Rank (MRR) is the average of the reciprocal of the rank at which the first relevant chunk appears:

$$\mathrm{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\mathrm{rank}_i}$$

where rank_i is the position of the first relevant result for the i-th query; an MRR of 1.0 means the first retrieved chunk is always relevant, while lower values indicate that relevant results are buried deeper in the ranked list [39]. In the experiments reported below, $K = 5$ is used for both Precision and Recall. Structural Pass Rate reports the percentage of generated modules that satisfy all basic structural requirements: correct `MODULE`/`ENDMODULE` encapsulation, balanced `PROC`/`ENDPROC` and `TRAP`/`ENDTRAP` delimiters, and absence of residual template placeholders. Structural Completeness Score is a checklist-based score (out of 6) evaluating the presence of key elements: module encapsulation, entry procedure, error handling (`TRAP`), placeholder-free output, and balanced delimiters. Functional Compliance is a qualitative assessment of whether the generated code satisfies the specific requirements of the user query, including correct use of specified tools, work objects, and operational sequences.

4.2 RAG based results

This section presents experimental results of the RAG pipeline, including retrieval performance under three system configurations as well as code generation quality.

4.2.1 Retrieval performance

4.2.1.1 Quantitative retrieval metrics

To evaluate the performance of retrieval components separately from end to end code generation, a benchmark set of 30 test queries was constructed, covering five categories of tasks: pick and place operations, pallet flow, solder sequence, signal processing, and error recovery logic. This test set is larger and more extensive than the 21-session test set used for code generation evaluation (see Section 4.1) because the retrieval evaluation needs to cover query types that do not necessarily all translate into a full code generation task. For each query, the authors manually annotated relevant ground truth chunks from both the document database and the code database [75]. Retrieval performance is measured by Precision@5, Recall@5, and MRR.

Table 4.2: Retrieval performance comparison across configurations. Results are averaged over 30 test queries.

Configuration	Precision@5	Recall@5	MRR
Naive RAG (Doc only)	0.52	0.41	0.58
Dual Retrieval (no HyDE)	0.68	0.57	0.72
Dual Retrieval + HyDE	0.78	0.65	0.83

The experimental results in Table 4.2 show a clear and gradual improvement in retrieval quality with the addition of system components. The Naive RAG baseline, which searches only with raw user queries in the document index, achieves 0.52 at Precision@5, 0.41 at Recall@5, and 0.58 at MRR. This relatively limited performance is mainly due to a semantic mismatch between the natural language task descriptions and the technical terms in the RAPID reference manual [76]. For example, a query

like “pick up the part from conveyor” does not directly match to a document entry indexed by a directive name like `MoveL` or `SearchL`.
After adding the dual retrieval strategy, the performance has been significantly improved: Precision@5 increased to 0.68, Recall@5 increased to 0.57, and MRR increased to 0.72. Much of this gain comes from the stream of code examples, as it provides more syntactically specific references than the stream of documentation alone can provide. The two streams perform different roles: the document retriever fetches conceptual and parameter knowledge, while the code retriever provides production ready templates showing how the instructions are combined in practice.
After adding HyDE [41] to the complete pipeline, the indicators are further improved: Precision@5 reaches 0.78, Recall@5 reaches 0.65, and MRR reaches 0.83. HyDE addresses the vocabulary gap problem by first generating a hypothetical intermediate answer based on the query and then using this answer as the retrieval key. This synthetic document is semantically closer to the indexed text block than the original natural language query, thus enabling a more targeted retrieval. This improvement is particularly noticeable in abstract or high level queries, since hypothetical documents bridge the gap between user intent and technical terms in the knowledge base.
Still, some failures still exist. For queries that involve multiple steps and span several RAPID instructions that are not directly related to each other, the retrieval results sometimes cover only part of the required knowledge. The task decomposition of Planner Agent alleviates this problem to some extent, but the retrieval effect is still limited by the chunking granularity strategy.

4.2.1.2 Qualitative retrieval analysis

To illustrate the practical impact of the Planner Agent’s task decomposition on retrieval quality, consider a query like this: *“Create procedures for opening and closing gripper tools.”*
In the Naive RAG configuration, the original query is used directly as the search key for both indexes. The document stream returns a general entry on signal processing, while the code stream retrieves a broader library template. These results are topically relevant, but lack the concreteness needed to build a complete gripper module, such as I/O confirmation and error recovery.
In a full pipeline, the Planner Agent breaks down the same query into multiple targeted subtasks: one document query retrieves `Set/Reset` signal semantics, another retrieves `TRAP` error handling patterns, one code query retrieves gripper opening and closing sequences, and another retrieves tool library module organization patterns. The text blocks obtained in this way have high pertinence, such as the industrial gripper template `LGripp1_H_7V03.tmod`, which can directly provide the basis for the generation of code structure and safety mode. The end result is that the full pipeline generates a module with 261 lines for the same query, while Naive RAG generates only 147 lines, reflecting the richer retrieval context of the former.

4.2.2 Code generation quality

This subsection evaluates the syntactic and structural quality of the generated RAPID code under three system configurations.

4.2.2.1 Pure LLM baseline results

To establish a lower bound, the study first generated five RAPID modules using Pure LLM Baseline, that is, GPT-5.2 without retrieval context. Each module is evaluated in two dimensions: structural correctness and content correctness.
All five baseline modules pass structural validation. The LLM consistently produces well formed `MODULE`/`ENDMODULE` structures, pairwise balanced separators, and entry procedures. However, in terms of content validation, only 1 module passed, which is a pass rate of 20%. Table 4.3 summarizes these semantic errors.

Table 4.3: Content issues in Pure LLM Baseline modules ($N = 5$). Each row reports a category of RAPID specific semantic error and the number of sessions in which it occurred.

Issue Category	Sessions	Total Occurrences
Hallucinated instructions (`DIRead`, `DInput`, `DOutput`, `TriggL\On`)	2	11
`CTime()` used as numeric (returns `string` in RAPID)	2	8
`SetDO`/`ResetDO` with string constant instead of signal variable	1	20
Signal variable assigned string literal (signals are EIO configured)	1	4
Signal declared as `CONST` (not valid in RAPID)	1	4
Redeclares built in constant (`fine`, `z10`)	3	3
`Offs()` applied to `pos` instead of `robtarget`	1	2
Non standard error handling syntax (`ENDERROR`)	1	1
`PulseDO` with positional duration (should use `\PLength`)	1	1

The most common type of error is fictitious instructions. The LLM makes up function names like `DIRead()` and `DInput()` that do not actually exist in the RAPID specification. They sound a lot like functions in other automation languages, and therefore have a certain amount of "rationality", but will directly cause compilation failures on ABB controllers.
The second type of systematic error is the misuse of `CTime()`. RAPID's `CTime()` returns a string, but the model uses it for arithmetic comparisons, for example, `CTime() - tStart > timeout`. This might work in Python or C, but it does not work in RAPID.
The third category of common problems is related to I/O systems. The baseline model would try to assign string literals to signal variables or declare signals as `CONST`, which is a misinterpretation of RAPID's approach to managing hardware

I/O through EIO configuration systems, rather than through direct assignment in the program.
These results show that even advanced LLMs, without the support of retrieval context, can generate structurally correct RAPID modules, but still produce a large number of domain specific semantic errors [77]. The 20% content pass rate constitutes a baseline reference point for subsequent evaluation of various RAG configurations.

4.2.2.2 Structural validation results

In both configurations supported by RAG, the 21 generated modules were subjected to the same rule based static analysis process, checking six structural criteria: correct `MODULE`/`ENDMODULE` encapsulation, balanced `PROC`/`ENDPROC` with `TRAP`/`ENDTRAP` separators, inclusion of entry procedures and `TRAP` error handlers, and absence of residual template placeholders.
The results show that all 21 modules under these two configurations pass the structural verification, and the pass rate is 100%. Each module has the correct separator, `main` entry procedure, and no residual placeholders. This stable high performance comes from the common ground of both configurations: the dual knowledge base provides a syntactically concrete reference, while the structured prompt template enforces the RAPID module skeleton, whether the Planner Agent is enabled or not.
The difference between the two configurations is mainly in the secondary integrity metrics. The Full Pipeline achieves 100% in terms of motion instruction coverage, while Naive RAG achieves 91%. In terms of I/O instruction coverage, the Full Pipeline is 50% while the Naive RAG is 36%. This reflects that the Planner Agent is able to form more targeted retrieval tasks through task decomposition, which exposes more relevant context.
However, there are also two modules generated by the Full Pipeline that lack the `TRAP` error handler, giving them a ratio of 80% for this item and 100% for Naive RAG. Closer inspection reveals that in these sessions, the Planner Agent generates highly targeted subqueries that place more emphasis on operational logic than safety scaffolding, so the generator tends to output more detailed functional code and omit the error handler.
Overall, once code generation is built on a validated knowledge base and a rigorous prompt template, structural correctness—the dimension that usually gets the most attention in code generation benchmarks [1]—is no longer an issue for either configuration. The really meaningful difference is at the semantic level, which is discussed next.

4.2.2.3 Semantic validation and expert review

Structural verification confirms syntactic correctness, but does not determine whether the generated code actually meets the user's intent or conforms to industrial best practices. The Full Pipeline solves this problem by integrating a verification agent. This agent performs semantic review after code generation, evaluating functional conformance, safety mechanisms, and domain specific correctness that are invisible to rule based static analysis.

Out of 10 Full Pipeline sessions, the verification agent flagged 4 modules (40%) to be fixed. Table 4.4 categorizes the issues found.

Table 4.4: Semantic issues identified by the Full Pipeline validation agent across 10 generated modules.

Issue Category	Occurrences
Missing user specified tool/work object	2
Incorrect RAPID data type structure	1
Non compiling function misuse	1
Missing required section markers	4
Unsafe motion patterns (no approach/retract)	3
Excessive speed or blending for task type	1
Total issues across 4 failed modules	**12**

The most common problem is not including parameters explicitly specified by the user. Two of these modules were originally generated for the "plot with `PENNTCP` and `REFRAM_RUTA1`" request, but the default `tool0` and `wobj0` are used in the code, without incorporating the tool and work object coordinates specified in the user query. This is a requirements compliance error, not a syntax error. The code is structurally correct, but functionally incorrect.
The validation agent also found an error in the declaration of the `zonedata` data structure, which would pass the structural check but cause a compilation failure on the real controller. Another module mistakenly uses the `Present()` function for I/O signal checks, which RAPID does not compile for signal types. This illustrates that many language level semantic errors are not identified by basic syntax checks.
The real value of the Full Pipeline's verification layer is not in catching structural flaws—both configurations do a very good job of that—but in intercepting functional and semantic errors that would otherwise flow to a human operator or, worse, a robot controller.

4.2.3 Case study: generation of gripper tool library

This section presents a detailed overview of the entire end to end RAG workflow through a representative industrial task: generating a RAPID pneumatic gripper tool library module that can be used in the production environment.

4.2.3.1 User query

The natural language instructions received by the system are as follows:

> *"Generate a RAPID module for the pneumatic gripper tool library. This module should support the opening and closing process with digital I/O confirmation, include error handling based on timeout, and support an optional servo mechanical unit. Additionally, incorporate the gripper tip wear tracking function and perform a safe retreat movement after the opening operation."*

4.2.3.2 Task decomposition of the Planner Agent

The Planner Agent breaks down this request into four retrieval sub tasks, shown in Table 4.5. Each sub query focuses on a different aspect of the task, such as I/O control logic, security mechanisms, and the overall module architecture. As a result, the retrieved results are more focused than those from a single broad query.

Table 4.5: Planner Agent decomposition for the gripper tool library query.

Stream	Sub query	Target Knowledge
Documentation	"RAPID digital I/O signal handling Set Reset WaitDI"	Instruction semantics for signal control
Documentation	"RAPID error handling TRAP RAISE timeout"	Error recovery patterns and `TRAP` mechanism
Code Example	"gripper open close sequence with I/O confirmation"	Structural references for gripper operations
Code Example	"tool library module with service schedule"	Module level organization patterns

4.2.3.3 Retrieved context

The dual search mechanism returned relevant context from both search streams. Table 4.6 summarizes the key items retrieved.

Table 4.6: Key retrieved chunks for the gripper tool library query.

Stream	Content Summary	Score	Source
Documentation	PLC signal handling patterns: `WaitSignal`, `MachineZone`, `Release` sequences with zone based coordination	0.6255	RAPID Technical Ref.
Documentation	`TRAP` routine syntax and `RAISE` error propagation semantics	0.5890	RAPID Technical Ref.
Code Example	`LGripp1_H_7V03.tmod`: Industrial gripper library with `InitTool`, `ServSch`, `CheckHome` procedures	0.5505	Template Library
Code Example	Gripper open/close sequence with `CheckPart` and `AllSequences` calls	0.5120	Template Library

Among all the search results, the code template `LGripp1_H_7V03.tmod` has the greatest impact on the final generated result. It provides a reference close to the production environment for the module level organization method. This template follows ABB's industrial conventions, including header blocks with version information, signal declarations, initialization routines, and maintenance planning processes.

Although this template uses parameterized placeholders (such as `<ToolLibTag>`) and vendor specific library calls (such as `GripperPilotAirConnect`), its structural pattern—such as separating initialization, operation processes, and maintenance routines—provides an architectural blueprint for generating the result.
The document retrieval flow provides supplementary support at the semantic level, offering the correct basis for I/O signal operations, including the correct usage of `Set`/`Reset` for digital outputs, as well as timeout based waiting logic.

4.2.3.4 Generation of code analysis

The system produced a complete RAPID module called `GripperToolLib`, which is approximately 250 lines long. Representative sections (including the header specification, I/O declarations, error handling traps, and the opening process with retreat actions) are included in Appendix A.1.
The quality of the generated code can be evaluated from four perspectives. First, in terms of structural conformity, this module adheres to the organizational conventions in the retrieved template. The `MODULE`/`ENDMODULE` delimiters are placed correctly, and all processes are properly ended with `ENDPROC`.
Second, regarding safety mechanisms, the generated code includes a dedicated `TRAP` routine named `Trap_GripperToolLib`. This routine captures the error number through `ERRNO`, stops motion and clears the path using `StopMove` and `ClearPath`, and then continues to propagate the error through `RAISE`. This is consistent with the safety requirements in the RAPID Technical Reference Manual and also reflects the advantages of RAG based generation, as the system replicates patterns that have already been verified in the retrieved documents rather than inventing its own error handling structure.
In addition, this module also covers all the key requirements in the user's request, including: the opening and closing processes with I/O confirmation, optional servo unit support, advanced wear tracking implemented through incremental and reset logic, and the safe retreat action (`MoveToPreClose`) that is executed after the opening sequence.
Finally, the influence of the retrieval context on the generated result is also reflected in the organizational structure of the module, especially in the hierarchical division between the initialization process (`GripperToolLib_Init`), auxiliary processes (`EnsureInit`, `WaitDIHighWithTimeout`), and operation sequences. This is consistent with the structure of the retrieved `LGripp1_H_7V03.tmod` template. However, the generated code is not a word for word copy of the template; rather, the structure has been adapted to the specific requirements of the query. For example, the `<ToolLibTag>ServSch` maintenance planning pattern in the template was not copied directly. Instead, the system generated processes like `Gripper_CloseHigh-SpeedToContact_OpenCloseLowForce_UpdateWear` that are more consistent with the current domain requirements for wear tracking.

4.2.3.5 Verification results

The generated module has undergone both structural verification and semantic verification processes. Table 4.7 summarizes the structural validation outcome.

Table 4.7: Structural validation results for the generated `GripperToolLib` module.

Validation Check	Result	Notes
`MODULE`/`ENDMODULE` encapsulation	Pass	Correctly delimited
Entry procedure (`PROC Main`) present	Pass	Main entry point
`TRAP` error handler present	Pass	`Trap_GripperToolLib` defined
No residual placeholders	Pass	No `<...>` tokens
All `PROC` properly terminated	Pass	Matched with `ENDPROC`
Motion instruction compliance	Pass	`MoveJ` syntax correct
I/O instruction correctness	Pass	`Set`/`Reset` correct
Hallucinated function detection	Pass	No invalid RAPID functions
Overall	**Pass**	8/8 checks passed

This module passed all 8 verification checks in the first generation attempt without the need for iterative corrections. This demonstrates the effectiveness of the dual retrieval strategy combined with template constraint prompts: by relying on both verified document knowledge and production level code patterns, the system can generate well structured results that meet static analysis standards, without the need for multiple rounds of revision.
However, although static analysis and semantic analysis have confirmed the grammatical and structural correctness of this module, to complete the full functional verification, simulation in ABB RobotStudio is still necessary. This will be further demonstrated in Section 4.3.

4.3 Simulation results based on MCP

This section shows the experimental results of the automated simulation verification system. The system connects AI agents to ABB RobotStudio through a custom MCP server as described in Section 3.5.2.

4.3.1 Overview of experimental design

Unlike static and semantic verification in the previous section, simulation based verification is performed by running the generated RAPID code on a virtual controller to test whether it satisfies the physical constraints of the robot. Each experiment follows an iterative "generation–test" cycle described in Section 3.5.2: a RAPID module is first generated and then uploaded to the virtual controller via the MCP server, executed in simulation and diagnosed. If an error is found, diagnostic feedback is appended to the generation context, and the cycle is repeated. The individual experiments are arranged in order of increasing complexity to assess whether the experience gained in earlier experiments can be transferred to subsequent experiments, thereby reducing the number of iterations required.
There are seven groups of experiments in this study: (1) Cartesian path drawing with a two workpiece coordinate system; (2) digital rendering with arc constraints and cross robot migration; (3) pick and place based on a conveyor belt with conditional placement logic; (4) multi position palletizing with spatial reasoning; (5) a small

pyramid stack as a generalization test; and (6–7) two groups of large scale 14-piece pyramid stacking experiments, used to test scene introspection and joint limit recovery on different pallets.

4.3.2 Experiment 1: Drawing patterns using a two workpiece coordinate system

The first experiment required the generation of a RAPID program that draws two identical "star plus circle" patterns at two different locations in the virtual plane. This task examines multiple aspects of RAPID programming, including Cartesian path control (`MoveL`, `MoveC`), parameterization of the workpiece coordinate system, and joint space recovery. The star pattern consists of five `MoveL` line segments that connect the five vertices of a pentagram (35 mm radius); the outer circle consists of four `MoveC` quarter arcs (40 mm radius).
The experiment took six iterations to generate a successful program. Table 4.8 summarizes each iteration, the error encountered, and the corrective action taken.

Table 4.8: Iterative correction process for the dual pattern drawing task. Each iteration was executed automatically through the MCP server.

Iter.	Error from Simulation	Corrective Action
1	*Close to singularity* (wrist singularity at $z = 0$ mm, J5 $\approx 0°$)	Raised work objects from $z = 0$ to $z = 200$ mm
2	*Joint out of range: rob1_4* (J4 exceeded $\pm 160°$ limit after adding `SingArea \Wrist`)	Removed `SingArea \Wrist` instruction
3	*Joint out of range: rob1_4* (tool orientation tilt of 15° insufficient)	Reverted to vertical orientation; kept $z = 200$ mm
4	*Joint out of range: rob1_4* (180° work object rotation caused wrist configuration change)	Removed work object rotation; used position offset only
5	*Joint out of range: rob1_4* (robot stuck at J4=159.4° from previous failed run)	Added `MoveAbsJ` to home position as first instruction
6	*Success*, both patterns executed, robot returned to home	–

This experiment demonstrates the value of simulated feedback loops in diagnosis. Each error message returned by the virtual controller points directly to the failed constraint and leads to the next correction. Among them, a key realization brought by Iteration 5 is that a single previously failed run can leave the robot stuck in a joint configuration from which it cannot recover. This problem is not found by static analysis and shows why simulation based verification is necessary. The successful program used two workpiece coordinate systems that were offset in the Y-direction (`wobj1` at $[380, 80, 200]$ mm and `wobj2` at $[380, -80, 200]$ mm) and defined a procedure with `PERS wobjdata`, which allows the same drawing logic to be reused in

both locations. Figure 4.1 shows the simulation results with TCP traces enabled in RobotStudio, and Figure 4.2 shows the corresponding trajectory visualization.

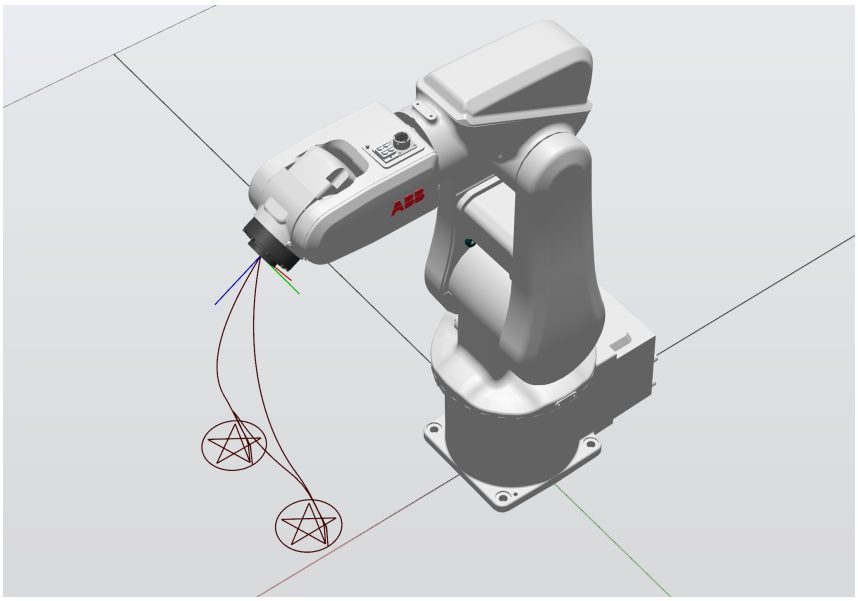


Figure 4.1: RobotStudio simulation result for the dual star-and-circle drawing task.

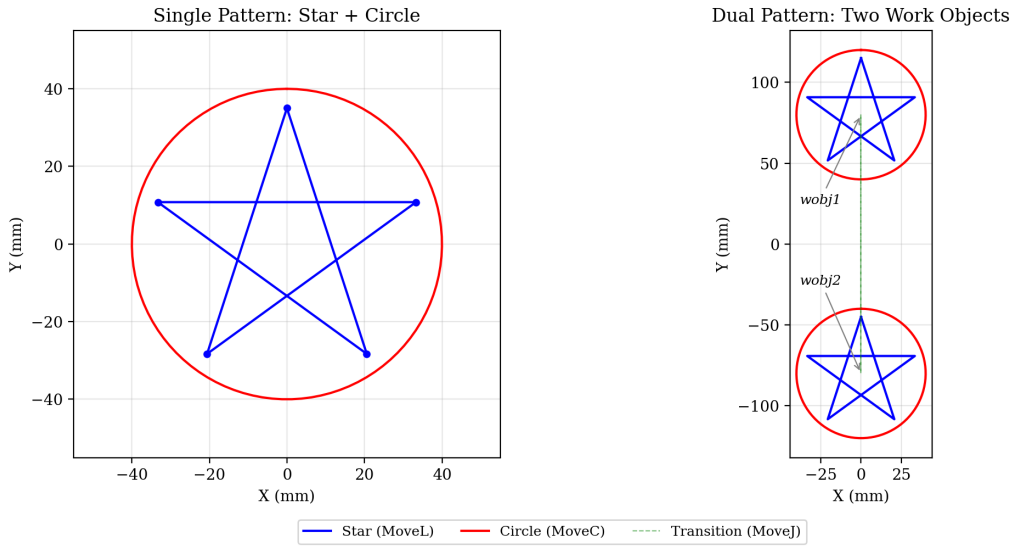


Figure 4.2: Trajectory visualization of the drawing task. Left: single pattern showing the five pointed star (`MoveL`, blue) inscribed in a circle (`MoveC`, red). Right: dual pattern with two work objects (`wobj1` and `wobj2`) offset in Y, connected by a `MoveJ` transition (green dashed line).

4.3.3 Experiment 2: Drawing numbers with MoveC constraints

The second experiment required the generation of RAPID programs to draw individual numbers (“2”, “3”, “4”, “5”) as well as combined numbers (“34”) on a horizontal plane. The task was adapted from a university laboratory course (PPU055 “Robotised Engraving” for the IRB1400) to test the system’s ability to handle arc constraints in RAPID.

Table 4.9: Simulation results for number drawing tasks. “Iter.” indicates the number of generate test cycles required to produce a successful program.

Task	Iter.	Result	Key Error / Note
Draw “2”	2	Pass	Iter. 1 produced closed loop path (heart shape); redesigned as 3 open strokes
Draw “3”	2	Pass	Iter. 1 failed: *MoveC arc >240°*; split each C-arc into two quarter arcs
Draw “4”	1	Pass	`MoveL`-only; first attempt succeeded
Draw “34”	1	Pass	Combined program with dual work objects; first attempt succeeded
Draw “5”	1	Pass	Reused quarter-arc pattern from “3”; first attempt succeeded

As shown in Table 4.9, the early programs (“2” and “3”) each required two iterations because they both had errors that would only be exposed at runtime: one when the closed path caused the drawn graph to be incorrect, and the other when the `MoveC` arc exceeded RAPID’s 240° limit [66]. The “3” was designed as two upper and lower C-shaped arcs, each half formed by a single `MoveC`. The bottom arc failed because a C-arc spans more than 240° when the arc is wide. The solution was to split each arc into two quarter arcs (∼90° each). After these fixes were added to the generative context, subsequent programs (“4”, “34”, “5”) succeeded on the first try, demonstrating that the feedback loop is able to transfer lessons learned from earlier failures to subsequent tasks. Figure 4.3 illustrates the simulation result of the combined “34” program.

A significant technical challenge in this experiment was RAPID’s use of backslash syntax for optional parameters (e.g., `\Off`, `\WObj`), which conflicts with JSON’s escape rules. Uploading directly via `curl` failed because the JSON parser on the C# side would treat sequences such as `\O` as invalid JSON escapes. The solution was to write the RAPID code to a `.mod` file and then properly encode it using Node.js `JSON.stringify()` before sending the HTTP request.

To test portability across robots, the same “34” program was migrated from the IRB120 (working radius ∼0.58 m) to the IRB2400_10_150 (working radius ∼1.55 m)

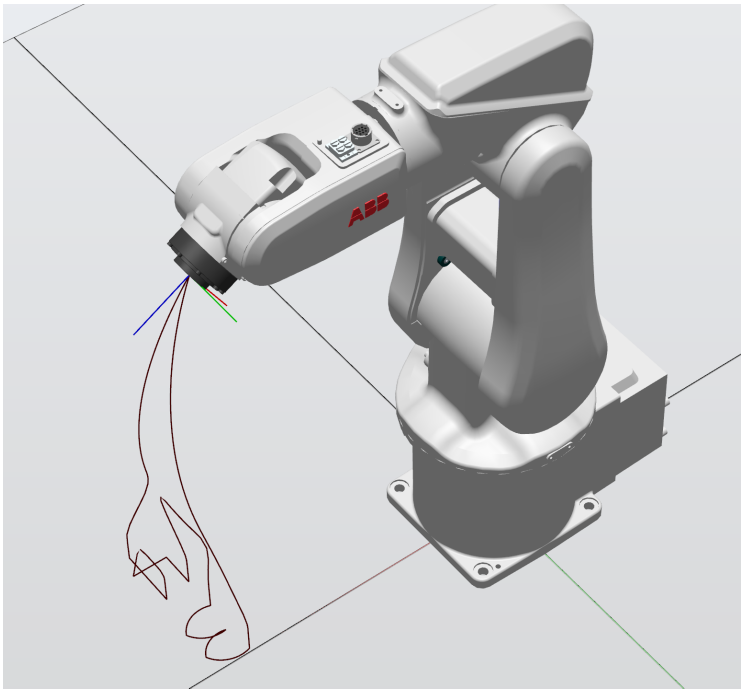


Figure 4.3: RobotStudio simulation result for the combined "34" drawing task.

without human intervention. The agent discovered the existing workpiece coordinate system through `list_rapid_variables` and queried the working range of the new robot through `get_station_status`. It identified that the workpiece coordinate system originally located at $X = 400\,\text{mm}$ fell within the inner reachable range limit of the IRB2400 ($\sim$850 mm) and was therefore relocated to $X = 1200\,\text{mm}$. At the same time, it scaled all drawing coordinates by a factor of three so that the numbers still had a proper visible size in the larger workspace, and adjusted the approach height and pen-lift height accordingly. The migrated program succeeded in a single run within about three minutes of agent processing time, without going through any debugging iterations. This result suggests that as long as kinematic parameters and existing declarations are exposed through MCP introspection, the generation pipeline is able to complete the transfer between different robot models without specifically redesigning the prompts. Figure 4.4 illustrates the final state of the IRB2400 after it has finished drawing the numbers.

4.3.4 Experiment 3: Conveyor-based pick and place with conditional placement

The third experiment tested the MCP system's ability to modify and validate an existing industrial pick and place program. The basic program implements a conveyor belt sorting system: a Smart Component alternately generates small boxes (orange, 100 mm high) and large boxes (green, 200 mm high) on the conveyor belt; sensors detect the box type, and the robot sorts them into two pallets (orange boxes are placed in `WO_Place_pq`, green boxes go to `WO_Place_gr`). The modification requirement in this experiment is to redirect the first green box to the orange pallet

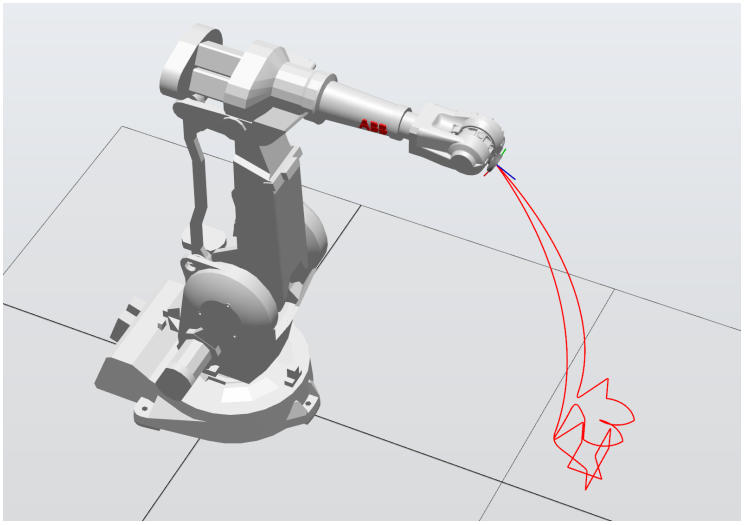


Figure 4.4: Cross robot migration of the "34" drawing task to the IRB2400_10_150.

while still maintaining the correct stacking of all subsequent boxes.
This experiment exposed a range of failure modes related to physical simulation constraints that cannot be detected by static analysis. A total of 11 iterations were performed throughout the experiment, and the failures fell mainly into two categories: module management problems and release-height calibration problems. Table 4.10 summarizes the iterative correction process:
The root cause of the release failures (Iterations 4–10) was geometric. The suction cup TCP must be located at or above the top surface of the box when released. The original program used a release offset of +40 mm, a value that was calibrated for the orange box, which is 100 mm high, so that the TCP sits 40 mm above the top of the orange box. However, if the same +40 mm offset is applied directly to the green box (200 mm high), then the TCP ends up 60 mm below the top of the green box, embedded inside the box, and the Smart Component is unable to release the payload. For taller boxes, the correct offset should be $+40+(200-100) = +140$ mm, such that the TCP is again located 40 mm above the top of the box. At intermediate offset values (e.g., +60, +90 mm), the TCP is still inside the box, so the release still fails. This failure mode is completely undetectable by code analysis: the RAPID syntax is correct and the motion targets are reachable, but the physical interaction between the suction cup and the box geometry prevents the desired behavior from occurring.
Figure 4.5 illustrates these three cases. The same z-axis scale (pallet top at $z = 0$) is used in all three panels, and the dashed horizontal line marks the release plane commanded by the program. Panels (a) and (b), marked by the red brace, use the same commanded offset of +40 mm (release plane at $z = 140$ mm), but the green box is 100 mm higher than the orange box, so while the orange box maintains a 40 mm clearance above its top, for the green box the plane falls 60 mm inside the box, and the release is blocked. Panel (c), marked by the blue brace, uses the corrected offset $+140 = +40 + (200 - 100)$ mm, derived at runtime from the block heights returned by `get_scene_objects`, thus restoring the 40 mm safety gap above the top of the

Table 4.10: Iterative correction process for the conditional pick and place task.

Iter.	Error / Observation	Corrective Action
1	Upload deleted `CalibData` module (tool/wobj declarations undefined)	Merged all declarations into single `Module1`
2	Green box knocked orange box off pallet (release clearance +40 mm too deep for 200 mm box)	Experimented with increased release offset
3	J3 out of range at +100 mm extra offset (placement near workspace boundary)	Reduced offset increment
4–8	Suction cup failed to release at +60, +90 mm (TCP embedded inside box mesh)	Systematic offset testing
9	Success at +40 mm (original offset), but stacking height incorrect for next box	Added height correction after placement
10	Suction release non deterministic at +40 mm boundary	Investigated simulation state consistency
11	*Success* at +140 mm (= +40 + (200 − 100) mm), TCP at box top surface	–

green box.
This experiment also shows that the MCP server's module upload strategy, which deletes all existing modules before loading, is destructive when the program depends on multiple modules. The original program used a separate `CalibData` module to hold tool and workpiece coordinate system declarations. When the modified `Module1` was uploaded, `CalibData` was deleted, resulting in undefined reference errors. The solution was to merge all declarations into a single module.

4.3.5 Experiment 4: Double stack stacking layer by layer

The fourth experiment tested the performance of the system in a palletizing task: six green boxes were placed on the left pallet to form a two-column, three-tier stack structure, and a layer by layer placement strategy was adopted (i.e., stack 1, layer 1 → stack 2, layer 1 → stack 1, layer 2 → . . .). This task adds complexity by using Linear congruence generators (LCG) [78] for random box generation, stack height tracking and overflow detection, and handling multi column space layouts.
The original implementation was to generate all six bins at once on the conveyor before starting the grasp. This scheme failed because the conveyor belt could only hold about four green bins at most, after which the bins would pile up, fall, or overlap. The AI agent initially misdiagnosed the problem as a placement problem and tried to adjust the offset, `confdata`, and timing controls without realizing that the underlying problem was insufficient conveyor capacity. The reason for this debug-

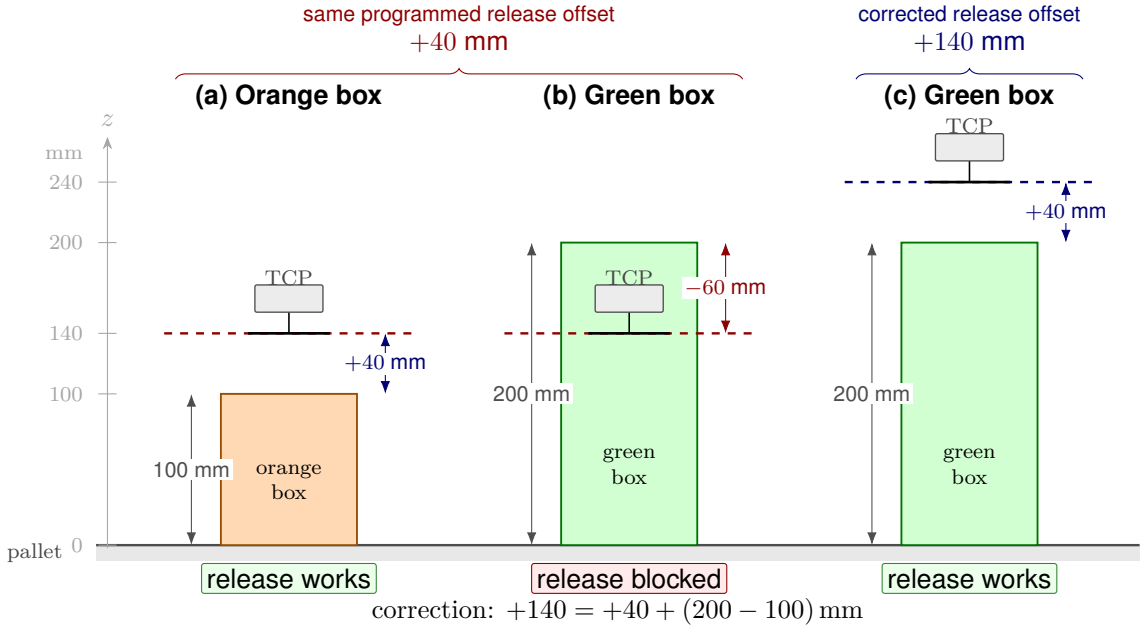
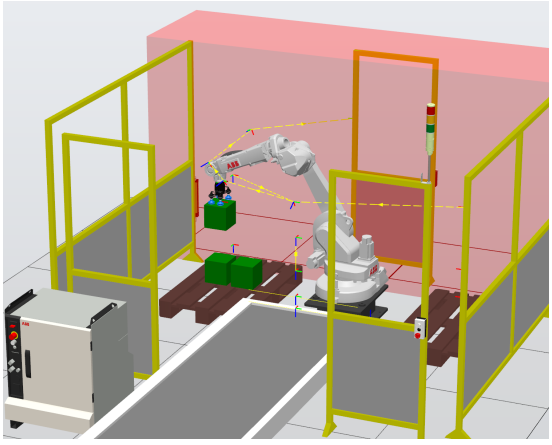
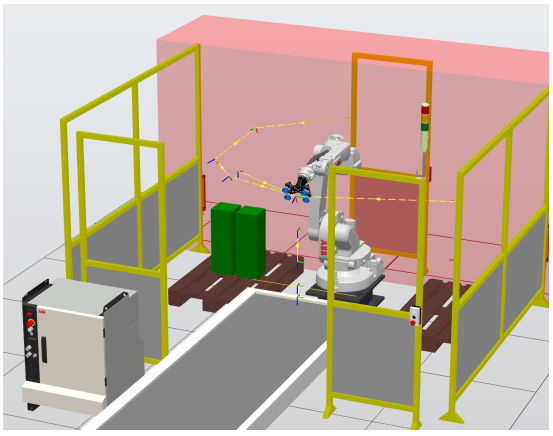


Figure 4.5: Release-height geometry for the conditional pick and place task.

ging failure lies in the fact that it only checks the final state of the simulation and does not observe intermediate states.
The final fix is to replace batch generation with an alternating generation-grab-place pattern: you make a bin, wait for it to reach the sensor location, grab and place, and regenerate the next one. That is standard industry practice; real production lines do not pre-pile all the workpieces on the conveyor belt at once. After this modification, the double stack palletizing task was successfully completed. Figure 4.6 shows the process and final result.



(a) Mid-process: robot placing the third box while two boxes are already stacked on the pallet.



(b) Final result: six green boxes placed in two stacks of three using a layer by layer strategy.

Figure 4.6: Dual-stack layer by layer palletizing in RobotStudio.

Another debugging episode is related to LCG-based random box type selection. The original LCG uses a multiplier of 1103 (odd) and an increment of 12345 (odd) and makes decisions based on `seed MOD 2`. Since "even × odd + odd = odd" and "odd × odd + odd = even", no matter what the initial seed is, its parity will be fixed alternating, and the result is a deterministic alternating sequence rather than a truly random sequence. The correction is to switch to a different criterion (`seed > 5000`), which makes use of better statistical properties in the high bits to make decisions.
Empirical tests show that the maximum safe stacking height is 600 mm for the IRB120 robot located at `WO_Place_pq` position. When `off_pq` = 400 mm, which is the third green bin in a single column, the approach point will push the TCP to $z = 886$ mm in the workpiece coordinate system, at which point the manipulator will collide with the existing stack during the approach and evacuation. Table 4.11 summarizes the height limit testing results.

Table 4.11: Stack height limit testing with worst-case scenario (all green boxes, 200 mm each). "Corner path failure" indicates the motion planner is near workspace limits.

MAX_STACK	Box #3 (off=400)	Box #4 (off=600)	Result
600 mm	Blocked by limit	–	3 boxes, safe
700 mm	Pass (corner path warning)	Blocked by limit	3 boxes, marginal
800 mm	Collision	Collision	Boxes scattered

4.3.6 Experiment 5: Stacking Pyramids (generalization test)

The fifth experiment was designed as a generalization test: can the MCP system handle a previously unseen novel spatial arrangement task? The task is given entirely in natural language:

> *"Place five green boxes on the left tray: the bottom is two boxes placed side by side in the Y direction, the top is one box centered directly above it (forming a pyramid), and next to this pyramid is a two-story vertical tower."*

The task required spatial reasoning that had not been attempted in previous experiments: the side-by-side spacing is set (two boxes with a width of 200 mm are placed at Y = ±100 mm to meet the edges), the top is centered (Y = 0 mm, at the midpoint of the bottom two boxes), and the tower structure is offset (Y = +300 mm, one box width from the nearest pyramid box).
Without explicit prompting, the AI agent correctly applied the four lessons learned from the previous experiments: the "alternate generation-grab-place pattern" in Experiment 4, the required +140 mm release gap for the green bin on the pq side in Experiment 3, the `MoveAbsJ` return home as the first command in Experiment 1, and the merging of declarations into a single module in Experiment 3. Instead of writing a generic loop, the agent explicitly writes sequential steps for the five placement locations, reusing the same parameterized procedure `Place_green(y_off, z_off)` from Experiment 4.

The program completed successfully on the first attempt without any errors. Figure 4.7 shows the final result. The MCP tool `get_scene_objects` is used to read the scene object positions, and it is also confirmed that the placement results are completely correct.

Table 4.12: Pyramid stacking verification via scene object positions (global coordinates). All boxes have consistent orientation ($r_z = -90°$), confirming stable placement.

Box	Role	Y (m)	Z (m)	Verification
Caja_gr_75	Bottom-left	1.24	0.348	–
Caja_gr_76	Bottom-right	1.04	0.348	$\Delta Y = 200\,\text{mm}$ (edge-to-edge)
Caja_gr_77	Top-center	1.14	0.548	$Y = (1.24 + 1.04)/2$ (centered)
Caja_gr_78	Tower base	0.84	0.348	$\Delta Y = 200\,\text{mm}$ from pyramid
Caja_gr_79	Tower top	0.84	0.548	$\Delta Z = 200\,\text{mm}$ (one box height)

Table 4.13 summarizes the key metrics across all seven MCP experiments. These small-scale tasks (Experiments 1–5) show a trend of decreasing the number of iterations required from an initial 6 to success on the first attempt as experience is gained into the generative context. Experiments 6 and 7 further expanded the scale of the pyramid task, and tested the scene introspection and joint limit recovery ability on different trays. Among them, Experiment 6 succeeded the first time when the learned pattern could be successfully applied, while Experiment 7 required 8 iterations to complete because of a previously unencountered asymmetry in the robot joint configuration space.

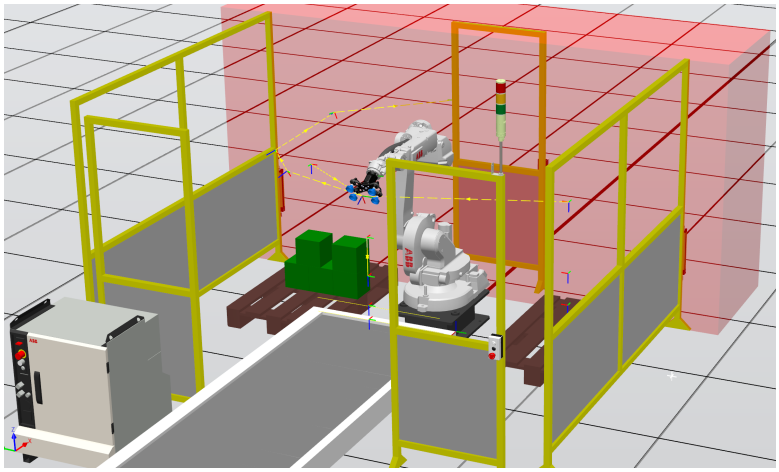


Figure 4.7: Pyramid stacking final result. Left: a three-box pyramid (two boxes side by side with one centered on top). Right: a two-layer vertical tower.

4.3.7 Experiment 6: Large scale pyramids leveraging scene introspection

The sixth experiment expanded the pyramid stacking task from 5 bins to 14 bins: on the right tray `WO_Place_pq`, a three-level pyramid ($3\times3+2\times2+1$) was constructed

Table 4.13: Summary of MCP simulation experiments.

#	Experiment	Iter.	Lessons	Key challenge
1	Dual star-circle drawing	6	0	Singularity, joint limits, state recovery
2	Number drawing (2, 3, 4, 5, 34) + IRB2400 port	8	2	`MoveC` arc limit, JSON escaping, cross robot scaling
3	Conditional pick and place	11	1	Release height, module management
4	Dual-stack palletizing	4	2	Conveyor capacity, LCG parity bug
5	Small-scale pyramid stacking	1	4	Spatial reasoning (first-attempt success)
6	14-block pyramid (right pallet)	1	5	Scene introspection at scale
7	14-block pyramid (left pallet)	8	3	J5 limit recovery, `MoveAbsJ` vs `MoveJ`

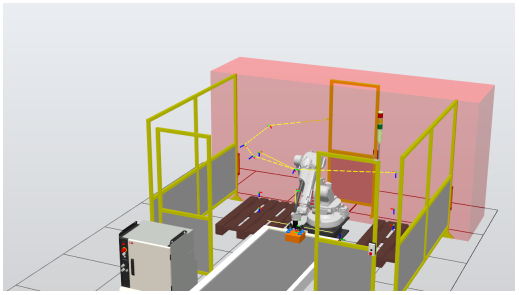
with orange squares (`Caja_pq`, size 200×200×100 mm) with a grid center distance of 210 mm. Compared to the five-box pyramid in Experiment 5, this experiment examines whether the agent can handle substantially increased placement plan sizes while reusing the alternate generation, grab, and place pattern in Experiment 4 and avoid the release height errors seen in Experiment 3.

Two MCP introspection tools played a central role in this experiment. First, the agent calls `list_rapid_variables` and sets `typeFilter="wobjdata"` to discover `WO_Place_pq` directly from the controller, thus avoiding the expensive coordinate transformation from workstation to workpiece coordinate system. It then calls `get_scene_objects`, which reads the size of `Caja_pq` as $(0.2, 0.2, 0.1)$ m through the bounding box metadata. These values were used to calculate the release height for each level of the three-level pyramid without the need to hard-code any physical dimensions in the prompts. Subsequently, the agent generates a unified `GenPickPlace` process that: (i) triggers box generation via `Set DO_Caja_pq`; (ii) waits for the signal from the sensor of the conveyor belt; (iii) grabs and places the box at the (x, y, z) offset position of the specific layer. This procedure is called 14 times using the precomputed offset.

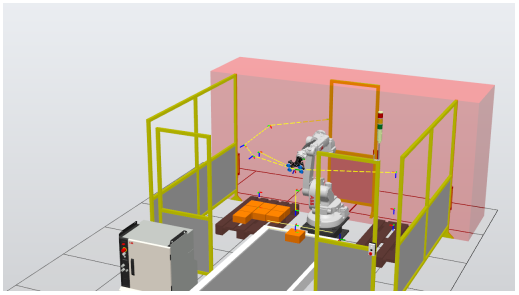
The program completed successfully on the first attempt. The total end to end time from natural language instruction input to pyramid stack completion is about 8 minutes, which includes scenario query, code generation, upload, and simulation execution. Figure 4.8 shows the mid-process and final states.

4.3.8 Experiment 7: Extended pyramid with joint limit constraints

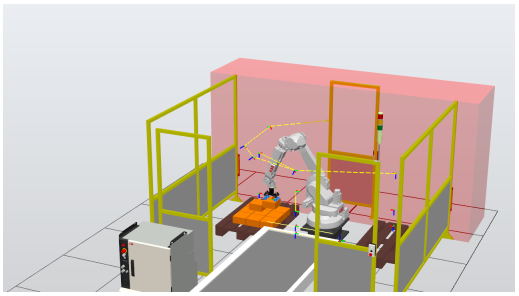
The seventh experiment repeated the 14-block pyramid task on the left tray `WO_Place_gr`, this time with a taller green square (`Caja_gr`, 200×200×200 mm in size, twice the height of the orange square). Both pallets are approximately the same distance from



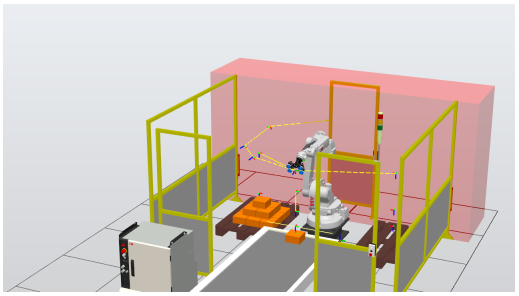
(a) $T = 0\,\text{s}$: simulation start, empty pallet.



(b) $T = 15\,\text{s}$: bottom 3×3 layer partially complete.



(c) $T = 25\,\text{s}$: middle 2×2 layer in progress.



(d) Final state: all 14 orange blocks placed in a three-layer pyramid.

Figure 4.8: Large scale $(9 + 4 + 1)$ pyramid stacking on the right pallet. Block dimensions were read at runtime through `get_scene_objects` bounding box metadata; the work object was discovered through `list_rapid_variables`.

the base of the IRB 1660ID robot, but they do not behave symmetrically in the joint configuration space. The placement action on the left tray would continuously push the fifth joint towards its ±120° limit, while the placement on the right tray in Experiment 6 would always remain within the nominal range. A higher box also places the proximity points higher in the upper level of the pyramid, thus exposing a gap in the recovery flow.

The role of scene introspection. Due to the different size of the box used this time, the release offset in Experiment 6 cannot be directly copied. The agent queries `get_scene_objects` again, reads that `Caja_gr`'s bounding box size is $(0.2, 0.2, 0.2)$ m, and recomputes the placement height accordingly. Without this query, the only other options are to hardcode the previous value of 100 mm or try to infer the height of the box from the grasp point coordinates. But both methods are unreliable once the scene changes.

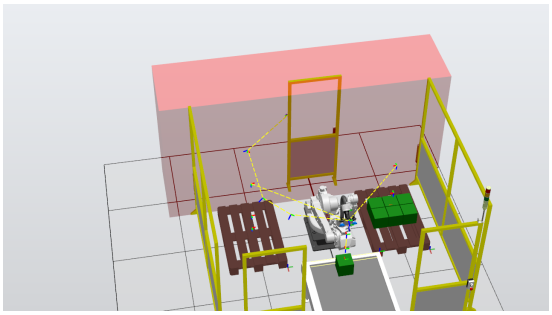
Joint 5 limit failures (attempts 1–7). The full run took around 51 minutes of agent time across about eight attempts. Attempts v1–v2 placed the pyramid base outside the reachable region, triggering the *Position outside reach* controller error. After the base was shifted inward, attempt v3 placed six blocks successfully, then Joint 5 exceeded its limit when placing block seven. With `ConfJ\Off` enabled, the motion planner could choose any joint solution freely, and J5 position was lifted higher each cycle until block seven, where the chosen solution passed over ±120°. Attempts

v4–v7 each tried a different recovery strategy: altered tool orientation, `MoveAbsJ` back to `jHome` between cycles, `jCalib` recovery at $J5 = 0°$, and combine `jCalib` and `jHome` together. None of them solved the problem. Table 4.14 lists the change and outcome of each version.

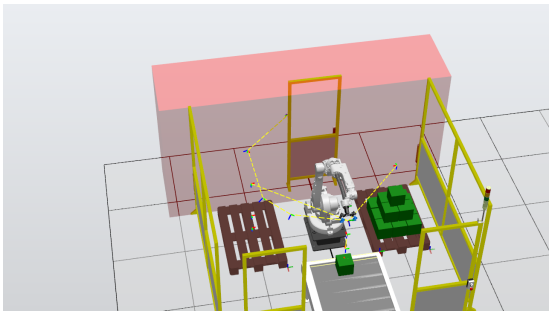
Table 4.14: Debugging attempts for the large scale green pyramid.

Attempt	Change applied	Outcome
v1–v2	Placement base shifted inward	*Position outside reach, J3*; still unreachable
v3	Dynamic approach height	6 blocks placed; *J5 out of range* at block 7
v4	Altered tool orientation	Broke the pick targets entirely
v5	`MoveAbsJ` to `jHome` between cycles	`jHome` itself has $J5 = 111.7°$, near the limit
v6	Added `jCalib` recovery (set $J5$ to $0°$)	Worked for one cycle, then J5 drifted up again
v7	Combined `jCalib` and `jHome`	`MoveAbsJ jHome` still pushed J5 high
v8	`MoveAbsJ jCalib` then `MoveJ HOME`	*Success*; all 14 blocks placed

`MoveAbsJ` vs `MoveJ`. `MoveAbsJ jHome` force every joint to its stored `jHome` value. `jHome` has J5 at 111.7°, near the limit. So `MoveAbsJ jHome` always put J5 back close to the limit. `MoveJ HOME` under `ConfJ\Off` is different. It only matches the Cartesian pose, and the planner picks any reachable joint configuration. In v8 the end-of-cycle became `MoveAbsJ jCalib` (force J5 to 0°) then `MoveJ HOME`. The 14-block run finished in about six minutes. Figure 4.9 compare v3 and v8.



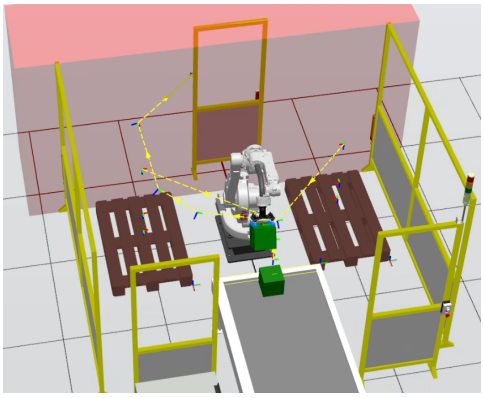
(a) Attempt v3: six blocks placed before J5 exceeded its ±120° limit at block 7.



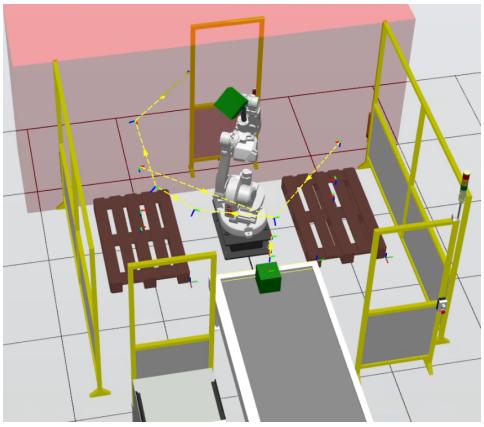
(b) Attempt v8: all 14 green blocks placed after introducing the `jCalib` → `MoveJ HOME` recovery pattern.

Figure 4.9: Large scale green pyramid on the left pallet. The failure at v3 and the fix in v8 illustrate the importance of distinguishing joint space (`MoveAbsJ`) from Cartesian space (`MoveJ`) recovery motions under `ConfJ\Off`.

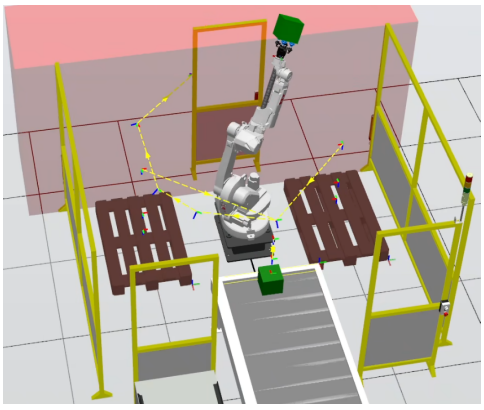
Simulation-state persistence. Restarting the RobotStudio simulation does not reset the robot's joint positions. Once J5 drifted past its limit, the controller stayed in the error state even when the simulation restarted. The agent remained unable to recover until it was informed that the robot arm was not moving, even though the code had been changed several times. It then realised the issue and uploaded a small RAPID calibration module to drive joints back to zero. Figure 4.10 shows some of the unusual joint configurations the planner explored during debugging.



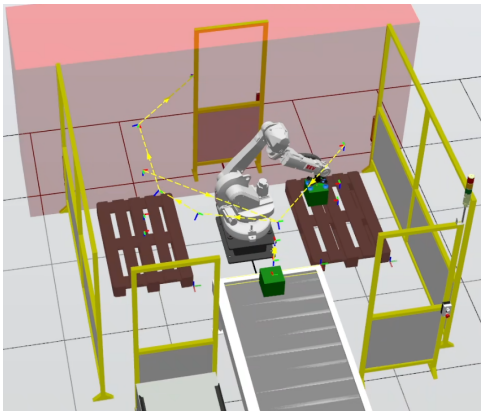
(a) Pose 1: arm stretched wide, block held low.



(b) Pose 2: arm raised with block overhead.



(c) Pose 3: extended reach, arm fully stretched upward.



(d) Pose 4: twisted approach from above.

Figure 4.10: Unusual joint configurations explored by the motion planner while debugging the green pyramid.

4.4 Summary of findings

This section summarizes the main findings from all experiments.
At the retrieval level, the dual stream retrieval strategy combined with HyDE improves Precision from 0.52 to 0.78.

At the code generation level, all three configurations reach a 100% structural pass rate, which shows that modern LLMs can produce well formed RAPID module skeletons even without domain grounding. However, the configurations differ clearly at the semantic level. The Pure LLM Baseline only reaches a 20% content pass rate, and most of its outputs contain RAPID specific hallucinations. The RAG grounded configurations remove these hallucinations, and the Full Pipeline can further detect functional defects that static analysis would miss.
At the simulation level, the MCP server based verification system was tested on seven experiments with increasing difficulty, ranging from simple path drawing to large scale pyramid stacking. The system caught physical constraint violations that static analysis cannot detect. More importantly, the agent gradually needed fewer iterations across experiments, showing that it could reuse corrections learned from earlier tasks and eventually solve new tasks on the first attempt.

5
Discussion

This chapter presents the answers to the three RQs, identifies the limitations of the study, and outlines directions for future work.

5.1 Answers to research questions

This thesis set out to investigate how Generative AI can be used to automate the generation and validation of executable industrial robot code, focusing on ABB's RAPID language. A RAG framework was designed, implemented, and evaluated by a embedded iterate process in simulator. The following subsections synthesize the answers to each RQ.

5.1.1 Answer to RQ1: RAG architecture for RAPID code generation

RQ1: *How can a RAG architecture be designed to effectively support the generation of executable RAPID code for industrial robots?*
The HyDE architecture proved effective for RAPID code generation. With this architecture, the system can maintain separate indices for technical documentation and production code templates. This gives the system two complementary views of the RAPID domain: one capturing conceptual knowledge, and the other offering syntactic patterns. In this way, Precision@5 increased from 0.52 to 0.68. Adding HyDE further improved Precision@5 to 0.78 and MRR to 0.83, mainly by closing the vocabulary gap between natural language user queries and the technical register of the RAPID reference documentation.
These results suggest that a RAG architecture for specialized industrial languages should maintain distinct retrieval streams for different content types, use a vocabulary translation mechanism such as HyDE to address domain specific terminology mismatch, and use structured prompt templates that enforce the target language's module conventions. The RAG configurations eliminated all categories of domain specific hallucinations seen in the ungrounded baseline, confirming that inference time knowledge injection is a viable alternative to fine-tuning when training data is scarce.

5.1.2 Answer to RQ2: Agentic planning and validation

RQ2: *To what extent does agentic planning and validation improve code quality beyond a standard RAG baseline?*
The agentic layer improves code quality in two distinct ways. First, the Planner Agent's query decomposition produces more targeted retrieval sub queries, which increases the functional richness of the generated code: the Full Pipeline achieves 100% inclusion of motion instructions and 50% I/O instruction coverage (versus 36%). The gripper tool library case study illustrates that the Planner Agent's decomposition yielded a 261 line module with complete I/O handling, compared to a 147 line module from the same query under Naive RAG.
Second, the integrated validation agent catches functional and semantic defects that static analysis based on rules cannot detect. It flagged 40% of Full Pipeline modules for issues such as missing tools that only user can use and incorrect data type structures. These are exactly the errors that survive structural validation but cause failures in deployment.
The agentic layer does create an unexpected trade off. Under agentic mode, sub agents tend to focus on operational details rather than safety instructions. Two modules omitted `TRAP` error handlers as a result. This finding suggests that safety critical code elements should be enforced as hard template constraints rather than left to retrieval. Overall, the agentic architecture provides a meaningful quality improvement over standard RAG, but its benefits are concentrated at the semantic rather than the structural level.

5.1.3 Answer to RQ3: Simulation based verification

RQ3: *How can automated simulation based verification close the validation loop for industrial robot code?*
The integration between the MCP server and ABB RobotStudio provides an automated way to test RAPID code generated by AI. Instead of checking the code only through static rules, the AI agent can upload the program, execute it in simulation, and read error feedbacks from the controller. This creates a closed loop of generation, execution, and correction.
This simulation layer was able to detect several types of problems that would be difficult to identify through text based analysis alone. Examples include wrist singularity conditions, joint limit violations, and MoveC arc geometry errors. These issues depend on the robot model, target positions, so they usually cannot be found by syntax checks alone. Running the program in RobotStudio therefore adds an important physical verification step.
The test campaigns also showed the value of iterative feedback. Some early programs required several correction cycles before they could run successfully. Each cycle dealing with a specific issue reported by the simulation. In later tasks within the same session, the system sometimes produced valid programs with fewer corrections, sometimes even on the first attempt. This means that AI can learn from it's previous errors and try to avoid them in the future.
The MCP based simulation bridge is therefore a useful contribution of this thesis. Based on the reviewed literature, this specific combination of MCP based tool use

and ABB RobotStudio verification has not been widely reported. More importantly, the results indicate that simulation based verification is technically feasible and practically useful for improving LLM generated industrial robot code.

5.2 Limitations

Several limitations bound the scope and generalizability of these findings.
First, the evaluation scale is limited. The code generation test set includes 21 sessions across seven task categories, with a stronger focus on drawing and path following tasks. The Pure LLM baseline was evaluated on only five sessions, and the paired comparisons between RAG configurations were also based on a small sample. Because of this, no formal statistical significance testing was conducted. The reported results should therefore be interpreted as preliminary evidence rather than as a definitive performance benchmark.
Second, the simulation based verification covered drawing, pick and place, palletizing, and pyramid stacking tasks. However, it was limited to a few robot models in a single environment, RobotStudio 2024 School Edition. Industrial applications such as welding, conveyor synchronization, and multi robot coordination were not tested and may reveal different failure modes. The MCP server is also limited to the ABB RobotStudio API, may need more work if migrate to other manufators is needed.
Third, the session knowledge transfer observed during simulation campaigns is ephemeral: correction heuristics do not persist across sessions. The system currently has no mechanism for saving and reusing validated constraints.

5.3 Future work

The findings and limitations of this study point to several directions for future research.
A natural first extension is to build a persistent knowledge base that stores both correction rules learned from simulation runs and required safety elements. For example, validated constraints such as `MoveC` arcs must subtend less than 240° can be added to the retrieval index. Safety elements such as `TRAP` handlers and speed limits can be enforced as hard template constraints rather than relying on retrieval. This design could reduce the number of correction cycles and also address the agent's tendency to ignore error handling during planning.
Second, the MCP based simulation approach should be extended to other robot platforms and programming languages. While the MCP protocol is platform specific, tool implementations would need to be developed for KUKA (KRL), FANUC (KAREL/TP), and PLC environments. Demonstrating cross platform applicability would establish the generality of the simulation approach.
Third, the generation strategy should be extended along two axes: supporting a library of scaffolding templates selected by inferred task type, and integrating physical deployment verification on real hardware. The former would address the rigidity of the current single template approach for non standard module structures, while

the latter would provide the strongest validation of code correctness and build the trust required for industrial adoption.
These directions connect with broader work on AI enhanced industrial automation. Recent studies on AI enhanced digital twins [79], unsupervised anomaly detection in manufacturing [80], human centered AI in Industry 5.0 [81], and code agents for robot programming [82] all point toward growing industrial interest in the type of AI driven robot code generation explored in this thesis.

6
Conclusion

This thesis investigated how Generative AI can automate the generation and validation of executable industrial robot code, focusing on ABB's RAPID language. The inspiration for this work comes not only from the fact that LLMs can quickly generate usable programs, but also from the ability of the industrial simulation software to determine whether these programs are executable and physically feasible. By integrating these functions, it reduces the need for manual trial-and-error operations, and provides a practical and feasible approach to improving the efficiency and reliability of the robot programming process.

A Retrieval Augmented Generation framework was designed, implemented, and evaluated for this purpose. The system combined a dual stream retrieval architecture with HyDE, an agentic planning and validation layer, and an MCP server connecting the language model agent to ABB RobotStudio. This design enables the generation of code based on domain knowledge, improves the accuracy of code syntax through iteration, and forms a closed loop by incorporating simulation feedback from the target environment.

A series of experiments have demonstrated the quality of the retrieval and the quality of the agent code. The results show that the retrieval architecture enhances the relevance and reliability of the generated RAPID code and reduces the illusion phenomenon in specific domains. The agentic layer helps identify functional defects that cannot be reliably captured through single-time generation or static checks alone. Based on simulation verification, it further reveals violations of physical constraints, including singularities, joint limit problems, and geometry-related execution errors. In all experiments, this agent also generates more stable solutions in subsequent tasks by leveraging previous correction experiences, thereby achieving session adaptation.

These findings indicate that the integration of RAG, MCP, and industrial simulation provides a practical foundation for AI assisted robot programming. Beyond demonstrating technical feasibility, the study shows that generative AI platforms can be combined with industrial verification tools to shorten debugging cycles, expose execution risks earlier, and improve development efficiency in industrial settings. This makes the approach meaningful not only as a research prototype, but also as a promising direction for faster and more dependable robot program development.

Bibliography

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

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

[3] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. Thrilled by your progress! large language models (GPT-4) no longer struggle to pass assessments in higher education programming courses. In *Proceedings of the 2023 ACM Conference on International Computing Education Research (ICER)*, volume 1, pages 588–594. ACM, 2023.

[4] Zichen Wang, Jingyi Wang, Fu Song, Kun Wang, Hongyi Pu, and Peng Cheng. K-RAPID: A formal executable semantics of the RAPID robot programming language. In *Proceedings of the 10th ACM Cyber-Physical System Security Workshop (CPSS '24)*, pages 64–76. ACM, 2024.

[5] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as I can, not as I say: Grounding language in robotic affordances. In *Conference on Robot Learning (CoRL)*, 2022.

[6] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500, 2023.

[7] Artem Lykov and Dzmitry Tsetserukou. LLM-BRAIn: AI-driven fast generation of robot behaviour tree based on large language model. *arXiv preprint arXiv:2305.19352*, 2023.

[8] Zhirong Luan, Yujun Lai, Rundong Huang, Xiaruiqi Lan, Liangjun Chen, and Badong Chen. Automatic robotic development through collaborative framework by large language models. *arXiv preprint arXiv:2402.03699*, 2024.

[9] Marcela G. dos Santos, Sylvain Hallé, Fabio Petrillo, and Yann-Gaël Guéhéneuc. AAT4IRS: Automated acceptance testing for industrial robotic systems. *Frontiers in Robotics and AI*, 11, 2024.

[10] Pengcheng Pei. Multi-agent system for cross-platform PLC code generation with domain adaptation. *International Journal of Emerging Technologies and Advanced Applications (IJETAA)*, 2(10):1–8, 2025.

[11] Teng Mao, Shuangtao Yang, and Bo Fu. A multi-agent framework for multi-source manufacturing knowledge integration and question answering. In *Companion Proceedings of the ACM on Web Conference 2025*, pages 1–4. ACM, 2025.

[12] Mikell P. Groover. *Automation, Production Systems, and Computer-Integrated Manufacturing*. Pearson, 5th edition, 2019.

[13] Martin Hägele, Klas Nilsson, J. Norberto Pires, and Rainer Bischoff. Industrial robotics. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 1385–1422. Springer, Cham, 2016.

[14] International Federation of Robotics. World robotics 2024: Industrial robots. Technical report, International Federation of Robotics (IFR), 2024.

[15] Geoffrey Biggs and Bruce MacDonald. A survey of robot programming systems. In *Proceedings of the Australasian Conference on Robotics and Automation*, Brisbane, Australia, 2003.

[16] International Organization for Standardization. Robotics — vocabulary. ISO 8373:2021, 2021.

[17] Valeria Villani, Fabio Pini, Francesco Leali, and Cristian Secchi. Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications. *Mechatronics*, 55:248–266, 2018.

[18] ABB Ltd. Our history. `https://global.abb/group/en/about/history`. Accessed March 2026.

[19] ABB Robotics. OmniCore robot controller. `https://new.abb.com/products/robotics/controllers/omnicore`, 2024. Accessed March 2026.

[20] Fei Tao, He Zhang, Ang Liu, and Andrew Y. C. Nee. Digital twin in industry: State-of-the-art. *IEEE Transactions on Industrial Informatics*, 15(4):2405–2415, 2019.

[21] ABB Robotics. RobotStudio suite — offline programming and simulation software. `https://new.abb.com/products/robotics/software-and-digital/robotstudio`, 2024. Accessed March 2026.

[22] Siyuan Chen. *Towards Prescriptive Maintenance Using Digital Twins and Artificial Intelligence*. Licentiate thesis, Chalmers University of Technology, Gothenburg, Sweden, 2025.

[23] Yuchen Liu, Siyuan Chen, and Ebru Turanoglu Bekar. A transformer approach for remaining useful life prediction and fault diagnosis of mechanical equipment. *IOP Conference Series: Materials Science and Engineering*, 1342(1):012059, 2026.

[24] ABB Robotics. Technical reference manual — RAPID instructions, functions and data types, RobotWare 7. Document ID: 3HAC065038-001, 2024. Accessed March 2026.

[25] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464. Association for Computational Linguistics, 2023.

[26] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008, 2017.

[28] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. Technical report, OpenAI, 2018.

[29] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[30] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):248, 2023.

[31] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.

[32] Muhammad Fadhil Ginting, Dong-Ki Kim, Sung-Kyun Kim, Bandi Jai Krishna, Mykel J. Kochenderfer, Shayegan Omidshafiei, and Ali-akbar Agha-mohammadi. SayComply: Grounding field robotic tasks in operational compliance through retrieval-based language models. *arXiv preprint arXiv:2411.11323*, 2024.

[33] Anthropic. Introducing contextual retrieval. Blog post, 2024. Available at `https://www.anthropic.com/news/contextual-retrieval`.

[34] Qintong Zhang, Bin Wang, Victor Shea-Jay Huang, Junyuan Zhang, Zhengren Wang, Hao Liang, Conghui He, and Wentao Zhang. Document parsing unveiled: Techniques, challenges, and prospects for structured information extraction. *arXiv preprint arXiv:2410.21169*, 2024.

[35] Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. LayoutLM: Pre-training of text and layout for document image understanding. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1192–1200. ACM, 2020.

[36] Oded Ovadia, Menachem Brief, Moshik Mishaeli, and Oren Sigal. Fine-tuning or retrieval? comparing knowledge injection in LLMs. *arXiv preprint arXiv:2312.05934*, 2023.

[37] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017.

[38] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of the 1st International*

Conference on Learning Representations (ICLR 2013), Workshop Track, 2013. arXiv:1301.3781.

[39] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[40] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[41] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. Precise zero-shot dense retrieval without relevance labels. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1762–1777. ACL, 2023.

[42] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[43] Anthropic. Building effective agents. Blog post, 2024. Available at `https://www.anthropic.com/research/building-effective-agents`.

[44] Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. Agentic retrieval-augmented generation: A survey on agentic RAG. *arXiv preprint arXiv:2501.09136*, 2025.

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

[46] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[47] OpenAI. Function calling — OpenAI API documentation. `https://platform.openai.com/docs/guides/function-calling`, 2025. Accessed April 2026.

[48] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[49] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–64, 2024.

[50] Anthropic. Introducing the model context protocol. Technical announcement, 2024. Available at `https://www.anthropic.com/news/model-context-protocol`.

[51] Anthropic. Model context protocol: Architecture overview. Official documentation, 2025. Available at `https://modelcontextprotocol.io/docs/learn/architecture`.

[52] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning (CoRL)*, PMLR 78, pages 1–16, 2017.

[53] Ali Nouri, Johan Andersson, Kailash De Jesus Hornig, Zhennan Fei, Emil Knabe, Hakan Sivencrona, Beatriz Cabrero-Daniel, and Christian Berger. On simulation-guided LLM-based code generation for safe autonomous driving software. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025)*, 2025. arXiv:2504.02141.

[54] Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. VoxPoser: Composable 3D value maps for robotic manipulation with language models. In *7th Conference on Robot Learning (CoRL)*, PMLR 229, 2023.

[55] Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. ChatGPT for Robotics: Design principles and model abilities. *IEEE Access*, 12:55682–55696, 2024.

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

[57] Naoki Wake, Atsushi Kanehira, Kazuhiro Sasabuchi, Jun Takamatsu, and Katsushi Ikeuchi. ChatGPT empowered long-step robot control in various environments: A case application. *IEEE Access*, 11:95060–95078, 2023.

[58] Mattias Bennulf, Fredrik Danielsson, and Xiaoxiao Zhang. Human-robot communication using large language models for automated kitting. *IOP Conference Series: Materials Science and Engineering*, 1342(1):012049, mar 2026.

[59] Omkar Salunkhe. Developing RAGs for robot code generation. In *Proceedings of the Swedish Production Symposium (SPS 2026)*, 2026. To appear.

[60] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.12931.

[61] Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory Erickson, David Held, and Chuang Gan. RoboGen: Towards unleashing infinite data for automated robot learning via generative simulation. In *International Conference on Machine Learning (ICML)*, pages 51936–51983. PMLR, 2024. arXiv:2311.01455.

[62] Fanlong Zeng, Wensheng Gan, Zezheng Huai, Lichao Sun, Hechang Chen, Yongheng Wang, Ning Liu, and Philip S. Yu. Large language models for robotics: A survey. *arXiv preprint arXiv:2311.07226*, 2023.

[63] Christopher E. Mower, Yuhui Wan, Hongzhan Yu, Antoine Grosnit, Jonas Gonzalez-Billandon, Matthieu Zimmer, Jinlong Wang, Xinyu Zhang, Yao Zhao, Anbang Zhai, Puze Liu, Daniel Palenicek, Davide Tateo, Cesar Cadena, Marco Hutter, Jan Peters, Guangjian Tian, Yuzheng Zhuang, Kun Shao, Xingyue Quan, Jianye Hao, Jun Wang, and Haitham Bou-Ammar. ROS-LLM: A ROS framework for embodied AI with task feedback and structured reasoning. *arXiv preprint arXiv:2406.19741*, 2024.

[64] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024. arXiv:2304.05128.

[65] Microsoft. Azure OpenAI Service documentation. `https://learn.microsoft.com/en-us/azure/ai-services/openai/`, 2024. Accessed March 2026.

[66] ABB Robotics. Technical reference manual — RAPID overview, RobotWare 7. Document ID: 3HAC065040-001, 2024. Accessed March 2026.

[67] Nikolaos Livathinos, Christoph Auer, Maksym Lysak, Ahmed Nassar, Michele Dolfi, Panagiotis Vagenas, Cesar Berrospi Ramis, Matteo Omenetti, Fabian Lindlbauer, Kasper Dinkla, Lokesh Mishra, Yusik Kim, Shubham Gupta, Rafael Teixeira de Lima, Valery Weber, Lucas Morin, Ingmar Meijer, Viktor Kuropiatnyk, and Peter W. J. Staar. Docling: An efficient open-source toolkit for AI-driven document conversion. *arXiv preprint arXiv:2501.17887*, 2025.

[68] Antonio Jimeno Yepes, Yao You, Jan Milczek, Sebastian Laverde, and Renyu Li. Financial report chunking for effective retrieval augmented generation. *arXiv preprint arXiv:2402.05131*, 2024.

[69] Jie Huang, Mo Wang, Yunpeng Cui, Juan Liu, Li Chen, Ting Wang, Huan Li, and Jinming Wu. Layered query retrieval: An adaptive framework for retrieval-augmented generation in complex question answering for large language models. *Applied Sciences*, 14(23):11014, 2024.

[70] Huayi Wu, Haoyue Jiao, Shuyang Hou, Jianyuan Liang, Zhangxiao Shen, Anqi Zhao, Yaxian Qing, Fengying Jin, Xuefeng Guan, and Zhipeng Gui. GeoCo-lab: An LLM-based multi-agent collaborative framework for geospatial code generation. *International Journal of Digital Earth*, 18(1), 2025.

[71] Rodrigo Nogueira and Kyunghyun Cho. Passage re-ranking with BERT. *arXiv preprint arXiv:1901.04085*, 2019.

[72] Manish Acharya, Yifan Zhang, Kevin Leach, and Yu Huang. Optimizing code runtime performance through context-aware retrieval-augmented generation. *arXiv preprint arXiv:2501.16692*, 2025.

[73] ABB Robotics. Product specification — controller IRC5. `https://library.e.abb.com/public/da1fb6e18f914cb1a4d4e6cfec1d49fd/IRC%205_product%20specification_2019.pdf`, 2019. Accessed March 2026.

[74] OpenAI. Introducing GPT-5.2. Technical announcement, 2025. Available at `https://openai.com/index/introducing-gpt-5-2/`.

[75] Ellen M. Voorhees. The TREC-8 question answering track report. *NIST Special Publication*, 500(246):77–82, 1999.

[76] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human–system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[77] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. LLM hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA), 2025.

[78] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997.

[79] Siyuan Chen, Ebru Turanoglu Bekar, Jon Bokrantz, and Anders Skoogh. AI-enhanced digital twins in maintenance: Systematic review, industrial challenges, and bridging research–practice gaps. *Journal of Manufacturing Systems*, 82:678–699, 2025.

[80] Siyuan Chen, Sunith Bandaru, Silvan Marti, Ebru Turanoglu Bekar, and Anders Skoogh. Comparison of unsupervised image anomaly detection models for sheet metal glue lines. *Engineering Applications of Artificial Intelligence*, 153:110740, 2025.

[81] Gregoris Mentzas, Karl Hribernik, Johan Stahre, David Romero, and John Soldatos. Editorial: Human-centered artificial intelligence in Industry 5.0. *Frontiers in Artificial Intelligence*, 7:1429186, 2024.

[82] Omkar Salunkhe, Clarissa A. González Chávez, Hao Wang, Anna Syberfeldt, David Romero, and Johan Stahre. Developing code agents for robot programming: Technical and managerial perspectives. In *Advances in Production Management Systems. Cyber-Physical-Human Production Systems: Human-AI Collaboration and Beyond (APMS 2025)*, volume 764 of *IFIP Advances in Information and Communication Technology*. Springer, 2026.

A

Generated code examples

This appendix presents representative code examples from the generated RAPID modules discussed in the main text. These listings are provided here to preserve the readability of the Results chapter while making the full code available for inspection.

A.1 Gripper tool library module example

Figure A.1 shows a representative example from the generated `GripperToolLib` module (Section 4.2.3), illustrating the header conventions, I/O declarations, error handling trap, and the open-with-retreat procedure.

```
MODULE GripperToolLib
  CONST string Version_GripperToolLib := "ABB 1V00 - 2026-01-20";

  CONST signaldo DO_GripperOpen  := "doGripperOpen";
  CONST signaldo DO_GripperClose := "doGripperClose";
  CONST signaldi DI_GripperOpened := "diGripperOpened";
  CONST signaldi DI_GripperClosed := "diGripperClosed";

  PERS num pTipWearCount := 0;
  CONST num cTipWearMax  := 999999;

  TRAP Trap_GripperToolLib
    VAR errnum e;
    e := ERRNO;
    TPWrite "Error in GripperToolLib. ERRNO=" \Num:=e;
    StopMove;
    ClearPath;
    RAISE;
  ENDTRAP

  PROC Gripper_Open_AndRetreat()
    EnsureInit;
    ServoOpenIfExists;
    TipForceToZero;
    Reset DO_GripperClose;
    Set DO_GripperOpen;
    WaitDIHighWithTimeout DI_GripperOpened,
        tOpenTimeout_s, "Gripper open timeout";
    MoveToPreClose;
  ENDPROC
  ...
ENDMODULE
```

Figure A.1: Example from the generated `GripperToolLib` module, showing the header conventions, I/O declarations, error handling trap, and the open-with-retreat procedure.

DEPARTMENT OF MECHANICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY