University of Southern Denmark
Department of Mechatronics (CIM)
Study Program MSc. in Mechatronics

Exploring Gen-AI in the context of the embedded software-hardware interface

Thesis submitted in partial fulfilment of the requirements for the degree of
Master of Science in Engineering (Mechatronics)
Cyber Physical Systems

Student: Levi Fechete
Date of birth: 19 11 1996

Supervisor: Benaoumeur Senouci
Department: SDU Mechatronics (CIM)

Keystrokes: [Generative (Gen-AI), Large Language Models (LLMs), Code Generation Agent (CGA), Robotics, Digital-to-physical]

2025

Contents

Abstract

The rapid advancement of generative (Gen-AI) has had a significant impact on the software development industry. Within this domain, the fields of embedded systems and robotics are particularly noteworthy. Although Gen-AI holds substantial potential for transformation in these areas, its adoption has not yet reached the same level of integration as seen in other sectors of the coding industry. A large portion of this potential has not yet been released. This thesis seeks to address that gap. The proposed solution involves the use of Gen-AI tools, specifically CGAs, to help bridge the divide between the digital and physical worlds. The core idea is that robots should be capable of dynamically interacting with their physical environment based on AI-driven code generation. In addition, this work explores howLLMs can serve as bridges between the user and the CGA. The system aims to receive user instructions, interpret them via an LLM, and then communicate the appropriate commands to the CGA, which in turn generates the necessary code to control a robotic system. By presenting this framework and its implementation, the thesis aspires to provide valuable insights into the application of Gen-AI in embedded systems, ultimately contributing to the advancement of intelligent, adaptive robotics.

1 Introduction

As we know, there is not yet a clear definition of (AGI). One can say that AGI is a hypothetical stage in the development of ML in which an (AI) system can match or exceed the cognitive abilities of human beings across any task ([1]). Others can say that AGI is a hypothetical form of (AI) where a machine learns and thinks like a human does ([2]). But no one can tell for sure what AGI is defined as. This is one of the questions that still have to be answered, but there are other questions as well. One is, how capable is the current generative (Gen-AI)? Can it reach outside its virtual environment? How will this change the surroundings? Yes, the current versions of Gen-AI are very capable when it comes to virtual environments, but when it comes to breaking out of this environment, it seems to reach its limitations. Notice the formulation "it seems to", but is this the case? This project will try to answer this exact question.

1.1 Motivation

The previous section outlined several key questions concerning AGI and Gen-AI. These questions form the backbone of this thesis's problem description, with particular emphasis on the question of whether Gen-AI can extend beyond virtual environments, potentially progressing toward AGI. One motivation for exploring this issue is to evaluate the capabilities of current AI models and to determine how rapidly a functional product can be developed using these tools. In addition, there are several practical applications worth considering, such as enhancing robotics through OS improvements, increasing the efficiency of system setup, reducing robotics maintenance time, and resolving system compatibility issues (see Fig. 1). It is, therefore, of significant interest to delve

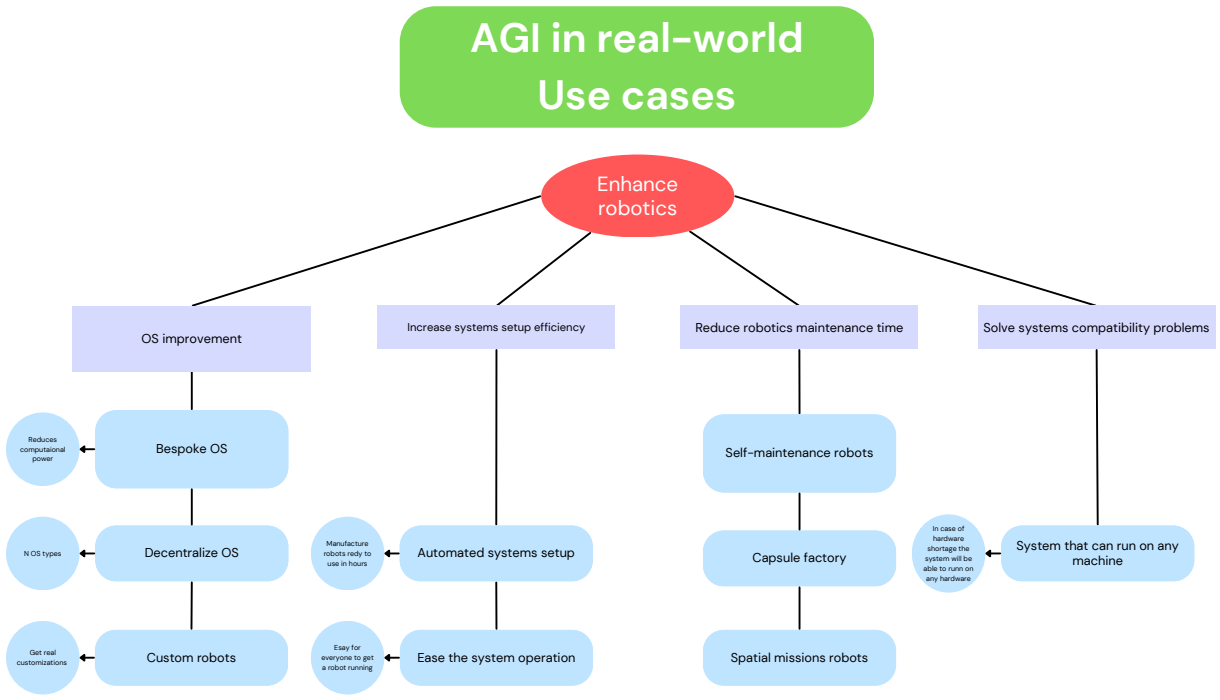

Figure 1. Potential application of AI in robotics

deeper into the field of Gen-AI and leverage the resulting insights to advance robotics. Accordingly, this thesis will address the following matters:

- Prove the concept of AI being able to reach out from the digital environment into the physical environment
- Explore the field of Gen-AI in embedded systems
- Develop a showcase robot

Problem statement

One may ask: why pose this question? Can Gen-AI extend beyond the virtual environment into the physical world? The reason lies in the clear fact that AI systems excel in their native environment, the digital domain, where they were fundamentally designed to operate. However, whenever the integration of AI into robotics is discussed, a degree of scepticism arises regarding its feasibility. This scepticism is justified, as the physical world presents a highly dynamic and complex environment, markedly different from the controlled, structured nature of digital systems.

Currently, the power of AI in robotics is primarily leveraged for visual recognition and data processing, which represents a valuable starting point. However, this approach comes with certain

limitations. For instance, a single codebase is often expected to handle all tasks while dynamically adapting to various scenarios, an objective that presents significant challenges.

Hence, this thesis adopts an approach that combinesLLMs, Python code, and CGAs to achieve the goal of dynamically controlling robot behaviour through real-time code generation. This approach implies that continuous communication between the environment, the LLM, the Python code, and the CGA enables the on-the-fly generation of code, allowing the robot to modify its behaviour easily and adapt in a significantly broader range of scenarios.

Feasibility study

While the proposed ideas appear promising, a critical question arises: How feasible is it to implement all of these approaches, particularly within the scope and constraints of this thesis? To address this, a feasibility study was conducted. This study included a concise review of the current state of the art in the relevant domains, namely AI in robotics, component identification, and AI-based OS. Additionally, a set of preliminary practical implementations of the proposed methods was carried out. These early trials served as a proof of concept, aimed at assessing the viability of each approach before proceeding with full-scale development. Based on the results of this study, the methods with the highest success rates were selected for further implementation in the thesis. Meanwhile, those approaches that showed potential but lacked immediate feasibility were paused, with the prospect of being further explored in future research projects.

1.2 Objectives

Given that the project timeline spans only four months, with the first month primarily dedicated to project planning, certain limitations were imposed on both the scope of work and the outcomes that could realistically be achieved. As a result, careful prioritisation was necessary to focus on the most impactful and feasible components of the project. The overall roadmap of the thesis is illustrated in Fig. 2 and is described in greater detail in the following section.

With all this in mind, it is important to outline some of the key delimitations of the project. One significant limitation arises from the area of component identification. As will be discussed later, this aspect presents the greatest complexity, requiring substantial time investment and potentially novel methodological approaches. Indeed, it could constitute an independent project in its own right. Consequently, it was decided to set aside this area for the current thesis and instead focus on system identification and control through AI. Even within this more focused domain, certain delimitations and goals were established at the beginning. These goals were as follows:

- Given a set of components(assuming these components are the ones forming the robot), the system should identify the robot type and its task execution capabilities

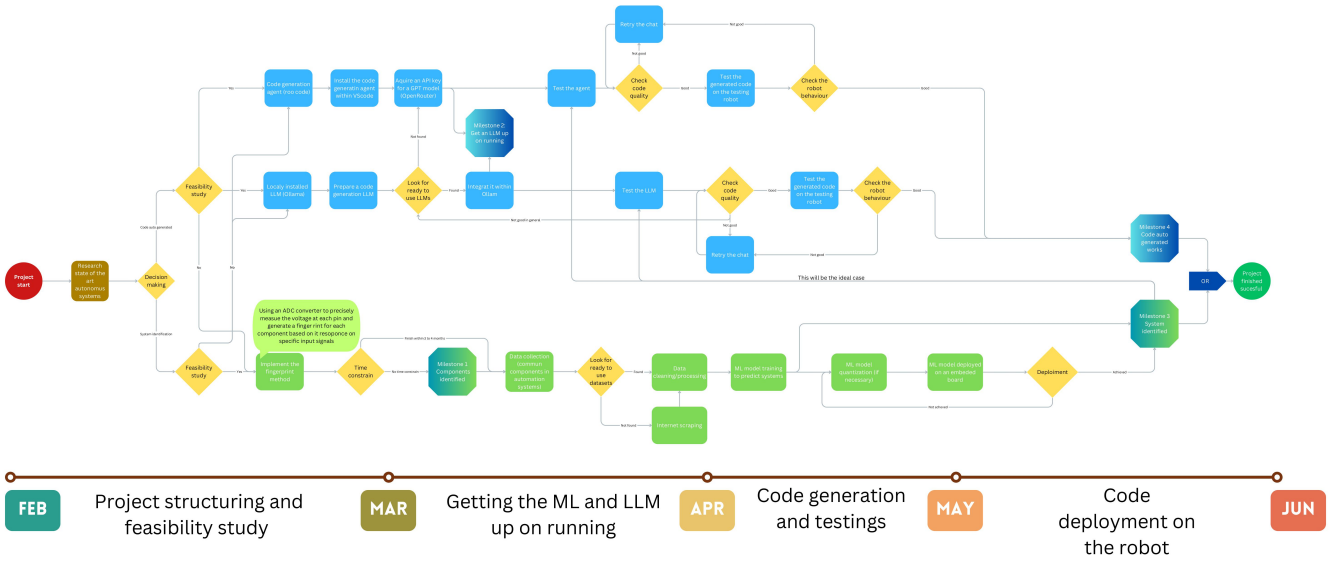

Figure 2. Thesis road map

- Given the system description, the system should be able to autonomously write and run the necessary code to control the robot.
- The system should be able to take user commands and act on these commands, or refuse if the command is outside the robot's capabilities.

1.3 Literature review

This section builds upon the foundations laid in the feasibility study, maintaining a strong research-oriented approach. However, the emphasis here shifts toward the acquisition and synthesis of knowledge across key technological domains, specifically Generative AI (Gen-AI), robotics, and AI-driven industries. The primary aim is to delve deeply into the current landscape of these rapidly evolving fields and extract insights that are directly applicable to the integration ofLLMs within robotic systems.

Unlike the feasibility study, which focused more on assessing project viability, this literature review is designed to enrich the theoretical and technical understanding necessary for advanced implementation. It involves identifying and critically analysing the state-of-the-art applications of LLMs in robotics, exploring their current capabilities, examining existing limitations, and uncovering areas

where future innovation is possible. The review also aims to reveal emerging trends that may shape the next generation of AI-augmented robotic systems.

Through this process, the review serves as both a knowledge-gathering exercise and a conceptual bridge between foundational research and practical application. The knowledge acquired will inform future development choices and help ensure that the final system leverages cutting-edge methods in perception, planning, interaction, and control. Ultimately, this section lays the groundwork for integrating intelligent, adaptive, and context-aware behaviour into robotic platforms, guided by the latest advancements in multi-modal AI technologies.

Review objectives

Investigate the current state-of-the-art applications of Large Language Models (LLM) in robotics. This research investigates the current state-of-the-art applications ofLLMs in robotics, with particular emphasis on advancements in perception, interaction, planning and control. MLMs play a central role in enabling robots to process and integrate visual, textual and embodied information.

Research Questions

- What are the latest advancements in LLM applications in robotics?
- How are LLMs being integrated into robotic systems?
- What are the challenges and limitations faced in this integration?

Advancements in LLM Applications

Perception and Interaction MLMs have significantly enhanced robots' ability to perceive their environment and to interact within both virtual and physical domains [3]. LLMs now enable robots to interpret and execute natural-language instructions with greater accuracy [3]. EmbodiedGPT(Generative Pre-training Transformer) aligns embodied, visual and textual data streams, thereby facilitating task planning that accounts for the robot's internal state and surroundings [3]. LEO employs visual tokens extracted from two-dimensional egocentric images and three-dimensional scene reconstructions, granting robots a more comprehensive understanding of spatial environments [3]. Integration of LLMs with vision-language models has further refined the joint processing of visual and linguistic inputs [3].

Planning At the high-level, LLMs decompose abstract and complex tasks into specific subtasks, leveraging their internal world knowledge and CoT reasoning [3], similar to how humans reason through steps [3]. This CoT reasoning [3], which involves rationales demonstrating step-by-step reasoning [4], helps break down complex tasks into manageable steps [3], enabling embodied agents to understand human intention and execute appropriate actions [3]. RT-2, for example, features CoT

reasoning abilities for multi-step semantic reasoning [3]. ReAct also incorporates CoT reasoning with plan generation [3]. Some approaches consider LLMs as world models for plan sequence search [3], leveraging methods like Monte Carlo Tree Search (MCTS) [3]. Research in this area includes studies on reasoning with language models as planning with world models [3] and language-guided MCTS [3]. Another line of work considers LLMs as instruction translators, utilizing their powerful semantic understanding to convert natural-language plans into planning-specific languages such as planning domain definition language (PDDL) [3], which are then used by traditional AI planners [3] This approach has been explored for translating natural language to planning goals [3] and empowering LLMs with planning proficiency [3] Meanwhile, low-level Embodied Action Planning [3] addresses real-world uncertainties not covered by high-level task planning [3]by utilizing pre-trained embodied perception models as tools [3]or leveraging the capabilities of foundation models [3], [5]like Vision-Language-Action (VLA) models [3]. These models, such as the RT series [3], [5], integrate visual [5], language [5], and action observations and can provide policy functions for action execution [3], enabling tasks under multi-sensory perceptions [3].

Grasping and Control LLMs now enable robots to interpret grasping directives and execute manipulations based on human instructions [3]. MLM-driven semantic scene reasoning allows robots to select appropriate grasping strategies in context [3]. Robotics Transformer architectures (for example, RT-1 and RT-2) map combined language and vision inputs directly to robot control commands [3], [5]. Trained on extensive datasets of robotic demonstrations, RT-1 executes over seven hundred distinct instructions with high success and generalises to novel tasks, objects and environments [3], [5]. RT-2 further extends this capability by transferring knowledge learned from web-scale data to on-robot control policies [3].

Embodied Agents The development of embodied agents that is, robots capable of perceiving their surroundings [3], understanding instructions [3]and self-monitoring[3] relies heavily on LLMs [3]. EmbodiedGPT aligns multimodal inputs in service of sophisticated task planning [3], while LEO's encoding of 2D egocentric and 3D scene data supports three-dimensional environmental comprehension and task execution [3].

Integration of LLMs in Robotic Systems

Robotic architectures typically adopt a modular design comprising an embodied multimodal foundation model, a visual perception module, a high-level task planner and a low-level action executor [3]. LLMs interface with well-trained policy models via APIs, generating code libraries of task-specific functions [6], [7]. Transformer networks are employed to map multimodal observations, visual, auditory and textual, into control signals [3], [5]. In certain frameworks, LLMs can resolve embodied question-answering tasks [3]and even generate complete control code ("code as policies") [3]without additional fine-tuning [3].

Emerging Trends and Future Directions

Development of natural-language–action world models that utilize video-based state representations is gaining traction [8]. Continual learning techniques are increasingly important for deploying policies across varied environments [3], [9]. Hybrid planning methods that combine LLMs with search-based approaches (for example, MCTS) represent an active area of research [3]. Aggregating datasets from multiple robotic platforms promises to enhance model robustness [3], [5]. Expansion of perception modalities to include tactile, auditory and force sensors will further refine planning capabilities [3]. Improved simulators are under development to generate higher-fidelity training dat [3], [8]. Mitigation of data bias [3]and exploration of hybrid generative architectures (combining autoregressive and diffusion models) are also key frontiers [8].

Challenges and Limitations

LLM-driven robotics faces several hurdles. Extensive, high-quality datasets are required, and real-world data collection is particularly challenging [3], [5]. Heterogeneous sensor data introduces inconsistencies that demand careful preprocessing [3]. The simulation-to-reality "gap" persists, with skills trained in virtual environments often failing to generalise fully [3], [10]. Models exhibit limited long-term memory [3]and may struggle with instructions outside their training distribution [3], [5]. Token-probability based planning can lack rigorous logical inference [3]. Vision-only inputs cannot capture all environmental complexities [3], and additional sensors incur cost and variability [3]. High computational demands of transformer-based LLMs impede real-time deployment [9], [11]. Extended context requirements further reduce efficiency [3], and the opacity of decision processes raises concerns over stability, fairness and accountability [9], [12]. Energy consumption and environmental sustainability also require attention [9](Here it comes the idea of using ML models for visual recognition [3]that will be used for specific tasks, hence reducing the computational demand [6], [11].).

Findings

LLM integration in robotics has driven significant progress across perception [3], planning and control domains [5], [8]. Nevertheless, overcoming challenges related to data acquisition, generalization [3], [5], interpretability and computational efficiency [5], [13]remains critical. Future research should focus on robust data methodologies [3], [14], enhanced memory and reasoning mechanisms, interpretable architectures [3], [12]and sustainable computation, [12], [13].

Solution/ Framework	Description	Key AI Technique/Model	Application in Robotics	Strengths/Features
Multi-modal Large Model (MLMs)	Models exhibiting remarkable perception, interaction, and reasoning capabilities by processing information from multiple modalities [3].	Large Models, Multi-modal Processing	Used as a promising architecture for the brain of embodied agents [3]. Significant in facilitating interactions in dynamic digital and physical environments [3]. Can combine multi-modal information like images, language, and audio [3].	High perception, interaction, and reasoning capabilities [3]. Integrate diverse information sources [3].
World Models (WMs) / Cosmos WFMs	Models designed to represent and predict the dynamics of the environment [3]. NVIDIA Cosmos World Foundation Models (WFMs) are presented as a specific implementation.	World Modelling, (potentially) Diffusion and Autoregressive models for Cosmos WFMs.	Promising architecture for the brain of embodied agents [3]. Essential for sim-to-real adaptation [3]. Cosmos WFMs can be fine-tuned for robotic manipulation [8] and autonomous vehicle systems [8]. Applicable to tasks demanding 3D consistency and action controllability [8].	Can serve as general-purpose simulators for the physical world [8]. Adaptable to various physical AI tasks [3], [8]
Robotics Transformer 1 (RT-1)	A specific Transformer-based robot learning method and policy [5]. It maps language and vision observations to robot actions by treating it as a sequence modelling problem.	Transformer Architecture [5]	Real-world robotic control using mobile manipulators, such as those from Everyday Robots [5]. Capable of performing diverse tasks and executing over 700 language instructions [5]. Can execute long-horizon tasks in frameworks like SayCan [5].	Effectively absorbs large amounts of data and scales with data quantity and diversity [5]. Demonstrates high performance and generalisation to new tasks, objects, and environments [5]. Shows robustness to distractors and backgrounds. It can absorb diverse data from simulations and various robot designs without sacrificing performance. [3], [5].
Gato	A Transformer-based multi-task robot policy [5]. It was used as a state-of-the-art baseline architecture for comparison with RT-1 [5].	Transformer Architecture	Used for multi-task robotic control [5]. Can serve as a manipulation policy in the SayCan framework (tested as a baseline) [5].	Based on the Transformer architecture, capable of multi-task learning [5].
BC-Z	A ResNet-based robot policy used as a baseline for comparison with RT-1 [5]. It is a feedforward model that uses continuous actions [5].	ResNet Architecture [5]	Used for multi-task robotic learning and control [3]. Functions as a manipulation policy within the SayCan framework [3].	A foundational model used for comparison and integration into frameworks like SayCan [5]. Can be scaled up to a larger version (BC-Z XL) [5].
SayCan Framework	A framework designed for grounding language in robotic affordances [5]. It enables the execution of long-horizon tasks by combining language grounding with a manipulation policy [5].	(Utilises underlying AI policies like BC-Z, Gato, RT-1)	Allows robots to accomplish complex tasks specified by natural language instructions that require chaining multiple manipulation and navigation skills [5].	Facilitates the breakdown and execution of complex, long-sequence tasks. Can be integrated with different underlying manipulation policies [5].

Table 1. Overview of AI Frameworks and Solutions in Robotics

1.4 Thesis road map

This section elaborates on Fig. 2 to clarify the starting points and Overall objectives of the thesis. As illustrated in the figure, the project is divided into two main branches.

The green branch focuses on system identification and description. The objective here is to develop a comprehensive framework that outlines all the components comprising the system, their interconnections, a concise description of the system these components form, and a corresponding list of tasks the system is capable of performing.

The blue branch focuses on code generation. Its primary goal is to utilise the system "recipe" produced by the green branch to generate the necessary code required to operate the system. Additionally, this branch is responsible for enabling communication with the user to receive subsequent tasks, thereby allowing for dynamic and continuous system control.

Potential applications

Figures 3 and 4 present flowcharts illustrating potential use cases for the proposed AI-based approach. Figure 3 demonstrates howLLMs can be integrated into the robot acquisition process within a factory setting. In this diagram, the right-hand column (in green) outlines the actions performed by the LLM, while the left-hand column represents the corresponding physical actions taking place in the real world. In this context, the LLM functions as a bridge between the environment and the robotic system, interpreting environmental input and, based on this interpretation, issuing commands to the CGA.

Furthermore, the factory depicted is highly interconnected, with all operations being logged into a centralised cloud network. This infrastructure facilitates robot integration by enabling newly deployed robots to access comprehensive information about the factory, allowing them to autonomously suggest tasks they can perform (assuming the robots are sufficiently versatile).

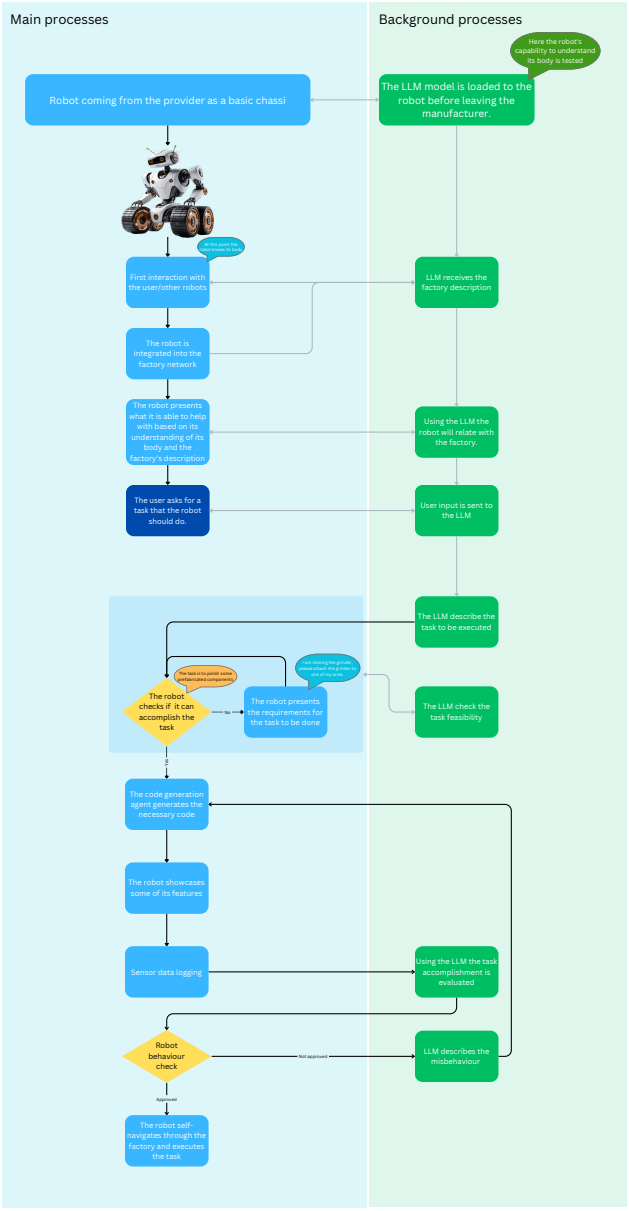

Figure 3. Gen-AI integration within the factory workflow

Additionally, because the CGA operates on the factory's server, the need for high-end computational components within the robot itself is reduced, thereby lowering hardware costs while simultaneously improving overall performance. Figure 4 expands on one of these scenarios in greater detail.

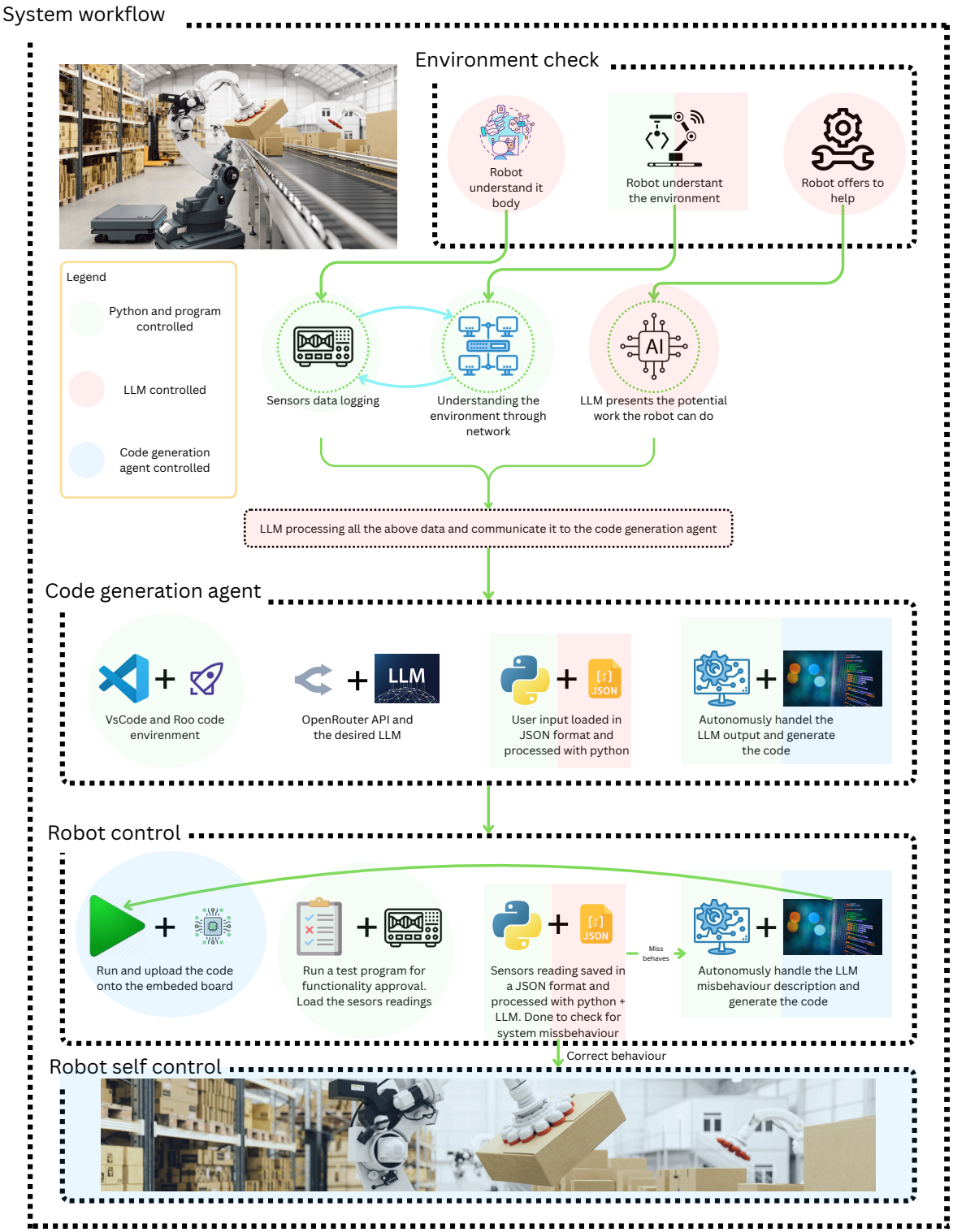

Figure 4. A detailed flowchart of Gen-AI integration within a factory workflow

2 Methodology

This section presents the data collection procedures, system development processes, and system testing. It begins with the environment setup, which involves preparing the necessary datasets, source code, diagrams, and other supporting materials required for the construction of the final showcase robot. This is followed by a description of the testing phase, which was carried out to evaluate the system's overall performance and functionality.

2.1 Fesability tests

This section primarily focuses on the initial phase of the project, during which the feasibility of the component identification process was assessed.

Proposal: The core idea of this proposal is that the system should be capable of automatically identifying the hardware components connected to it, similar to how a computer recognizes newly connected peripherals. One might reasonably ask: if computers can do this, why shouldn't robots be able to do the same? This is an excellent question. However, the reality is that such functionality is considerably more difficult to implement in non-industrial robotic systems. The primary reason lies in the limited communication protocols employed by many hardware components in this domain. Unlike USB or other standardised protocols used in computers, many robotic components communicate using basic on–off signals or analogue voltage levels, which provide minimal information and make component identification highly challenging.

A promising solution to this issue is proposed in [15]. Although the referenced work focuses on wireless components, the underlying method may be adaptable to other types of hardware as well. The suggested approach involves using an ADC to capture signal characteristics from each component and generate a unique electrical "fingerprint." Ideally, this fingerprinting process would be conducted at the manufacturing level, allowing each component to be labeled with a distinct signature that could later be recognized by any robotic system.

Due to the complexity and time requirements of this approach, this area of the project was ultimately bypassed. For the purposes of this thesis, it is assumed that the system is already aware of the connected components and their respective port assignments.

Implementation: In this phase, data obtained from the RPi's GPIO pin state checks were logged into a CSV file (see Fig. 7) for further analysis. The underlying idea was that by reading the voltage levels at each pin, it would be possible to determine the identity of the component connected to that specific pin. However, due to the absence of a built-in ADC on the RPi board, accurate voltage measurements could not be performed. As a result, this approach was postponed and ultimately abandoned.

Another method explored involved using a known sensor to assist in identifying other connected

peripherals. In this case, an accelerometer was employed to register the vibrations generated by DC gear motors (see Fig.5 and Fig.6). By applying a voltage to the pins connected to the DC gear motors, vibrations were induced, which were then detected by the accelerometer. This approach enabled the identification of the DC gear motors based on their vibration.

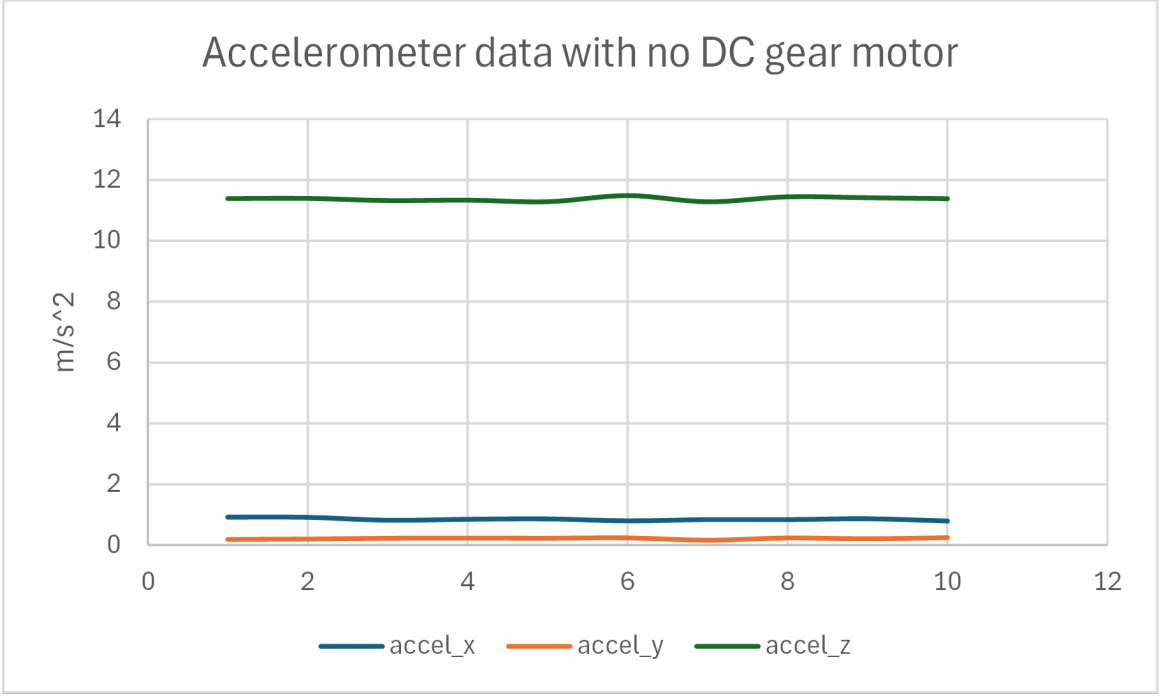

Figure 5. Accelerometer readings when the DC motors are disconnected

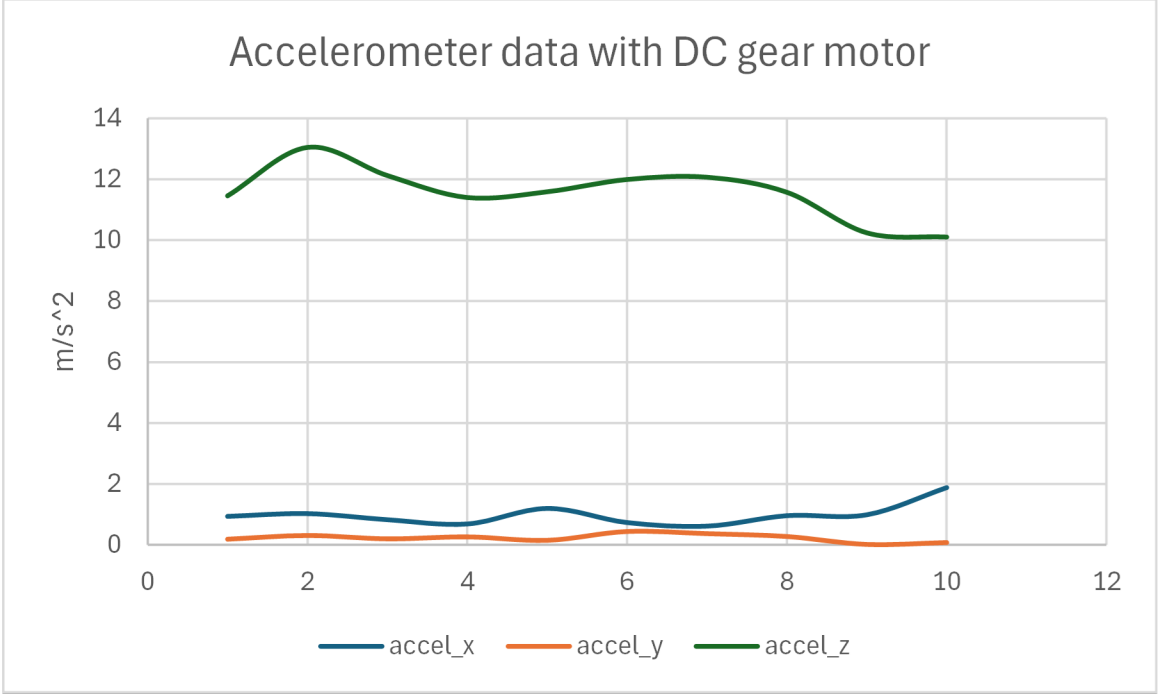

Figure 6. Accelerometer readings when the DC motors are connected

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	timestamp	fired_pin	pin_2_res	pin_4_res	pin_6_res	pin_12_re	pin_13_re	pin_16_re	pin_17_re	pin_18_re	pin_19_re	pin_20_re	pin_21_re	pin_22_re	pin_23_re	pin_24_re	pin_25_re	pin_26_re	pin_27_response	
2	2025-02-0	2		1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	
3	2025-02-0	4	1		1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	
4	2025-02-0	6	1	1		1	1	1	0	0	1	1	1	1	1	1	1	1	1	
5	2025-02-0	12	1	1	1		1	1	0	0	1	1	1	1	1	1	1	1	1	
6	2025-02-0	13	1	1	1	1		1	0	0	1	1	1	1	1	1	1	1	1	
7	2025-02-0	16	1	1	1	1	1		0	0	1	1	1	1	1	1	1	1	1	
8	2025-02-0	17	1	1	1	1	1	1		0	1	1	1	1	1	1	1	1	1	
9	2025-02-0	18	1	1	1	1	1	1	0		1	1	1	1	1	1	1	1	1	
10	2025-02-0	19	1	1	1	1	1	1	0	0		1	1	1	1	1	1	1	1	
11	2025-02-0	20	1	1	1	1	1	1	0	0	1		1	1	1	1	1	1	1	
12	2025-02-0	21	1	1	1	1	1	1	0	0	1	1		1	1	1	1	1	1	
13	2025-02-0	22	1	1	1	1	1	1	0	0	1	1	1		1	1	1	1	1	
14	2025-02-0	23	1	1	1	1	1	1	0	0	1	1	1	1		1	1	1	1	
15	2025-02-0	24	1	1	1	1	1	1	0	0	1	1	1	1	1		1	1	1	
16	2025-02-0	25	1	1	1	1	1	1	0	0	1	1	1	1	1	1		1	1	
17	2025-02-0	26	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1		1	
18	2025-02-0	27	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1		
19																				
20																				

Figure 7. RPi pins states

Yet another method considered for peripheral identification was the use of an object recognition ML model. While this approach appeared promising, it presented several challenges, most notably, its dependence on the quality of the camera module, camera positioning and mobility, and visibility of the internal components. Despite these limitations, the method was pursued to explore its viability. A pretrained model was acquired from [16], and the dataset was further refined using the Roboflow platform.

The model was successfully deployed; however, the results were suboptimal. As is often the case with ML applications, the primary challenge stemmed from the dataset, which lacked sufficient examples of some components. An augmentation process was applied to expand and enhance the dataset, but this did not lead to a significant improvement in model performance. Given the increasing time demands of this approach and the limited gains, it was ultimately decided to move on. For the remainder of the project, it was assumed that the peripherals were already identified, including both their component names and their respective connections to the main board.

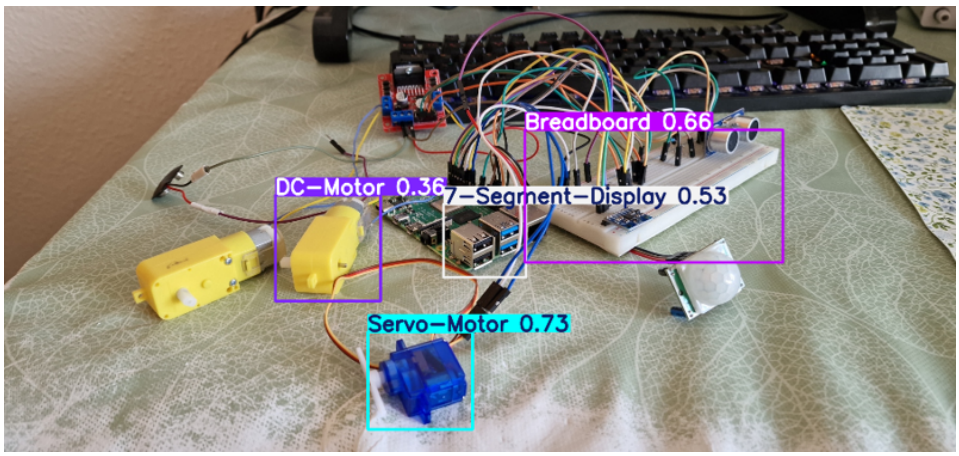

Figure 8. Object recognition model; result

2.2 Environment Preparation

As is common in robotics, this project involves two primary areas of exploration: software and hardware. The software component focuses on the application of various ML andLLMs. This includes the selection, fine-tuning, and comparative evaluation of multiple models to determine which is best suited for the specific requirements of this case. It also encompasses the integration of the selected model within the CGA environment. The hardware component then builds upon the software outcomes, applying the finalised solution to a physical system in order to validate and demonstrate its practical functionality.

2.2.1 ML and LLM models training

ML

To support component-based robotic system identification, a lightweight ML model was developed and trained. This model complements the use of large-scale language models by offering a computationally efficient alternative tailored specifically for classification tasks on resource-constrained devices. The following subsections outline the complete training pipeline, from data acquisition to model evaluation.

Data Collection and Preprocessing: The dataset used for training the model was constructed by scraping publicly available robotics-related resources. These sources provided component lists along with their associated robotic systems. The resulting dataset was stored in JSON format under the filename Data_generation_from_links.json. Each entry in the dataset includes the robot name, a list of components that constitute the robot, and a corresponding description of the robot's function and purpose, as illustrated below.

```
{
    "robot": "robot name"
    "components": "components that form the robot"
    "expected": "robot description"
},
```

Prior to training, the textual component lists were normalised (e.g., converted to lowercase) and vectorised using Term Frequency Inverse Document Frequency (TF-IDF). This method was chosen to represent the importance of each term (component) relative to the entire corpus, enabling effective text classification. Labels were encoded using scikit-learn's LabelEncoder to transform the categorical robot classes into numerical indices suitable for supervised learning.

Model Architecture and Training: The classification model was implemented using TensorFlow Keras. It consists of a multi-layer neural network trained to predict the most likely robot type from

a given list of components. The model was trained on the preprocessed data, with TF-IDF vectors as inputs and encoded robot types as targets.

Upon completion of training, the model was serialised and saved in HDF5 format (systemsfinder.h5). In parallel, the TF-IDF vectorizer and label encoder were also saved (vectorizer.pkl and label_encoder.pkl, respectively) to allow seamless reuse during inference without reprocessing or retraining.

Model deployment process For deployment, a modular function (run_ml_model) was developed to load the pre-trained model and its associated preprocessing objects. The function accepts a new component list, transforms it using the vectorizer, and generates a prediction. The model outputs a class probability distribution, from which the most probable class is selected and decoded into the corresponding robot type. To enhance usability, the model returns a structured response comprising:

- The original input components,
- The predicted robot type, and
- A predefined description of capabilities (e.g., "basic movement", "obstacle avoidance", "line following").

LLM

Fine-tuning the 7-billion-parameter Mistral language model followed a reproducible pipeline(Fig.9) that begins with curating a domain-specific dialogue corpus in JSON-Lines format, each entry containing paired user and assistant messages that encode the desired response structure. The data is processed with the Hugging Face datasets library and formatted by a custom formatting_func, which linearises each conversation into an instruction-response sequence and tokenises it with the Mistral SentencePiece tokeniser. Next, the base model (mistralai/Mistral-7B-Instruct-v0.1) is loaded in 4-bit weights via bitsandbytes to minimise GPU memory, and a Low-Rank Adaptation (LoRA) configuration (rank = 8, α = 16, target modules = q_proj, v_proj) is applied through the peft API, restricting training to <1% of the parameters. Supervised fine-tuning is carried out with the TRL SFTTrainer, using a small learning rate (2×10^{-4})), gradient accumulation, and three epochs, which produces a LoRA adapter (adapter_model.safetensors) whose loss converges toward 0.27, indicating successful task adaptation. The adapter is then merged into the full-precision base weights (merge_and_unload) to yield a single Hugging Face checkpoint, subsequently exported to the language-model-agnostic GGUF format with convert_hf_to_gguf.py. Finally, the float-16 GGUF model is quantised to Q4_K_M using llama-quantize.exe, reducing the footprint to 4GB while preserving accuracy, and the resulting file is packaged in an Ollama Modelfile for turnkey, offline inference.

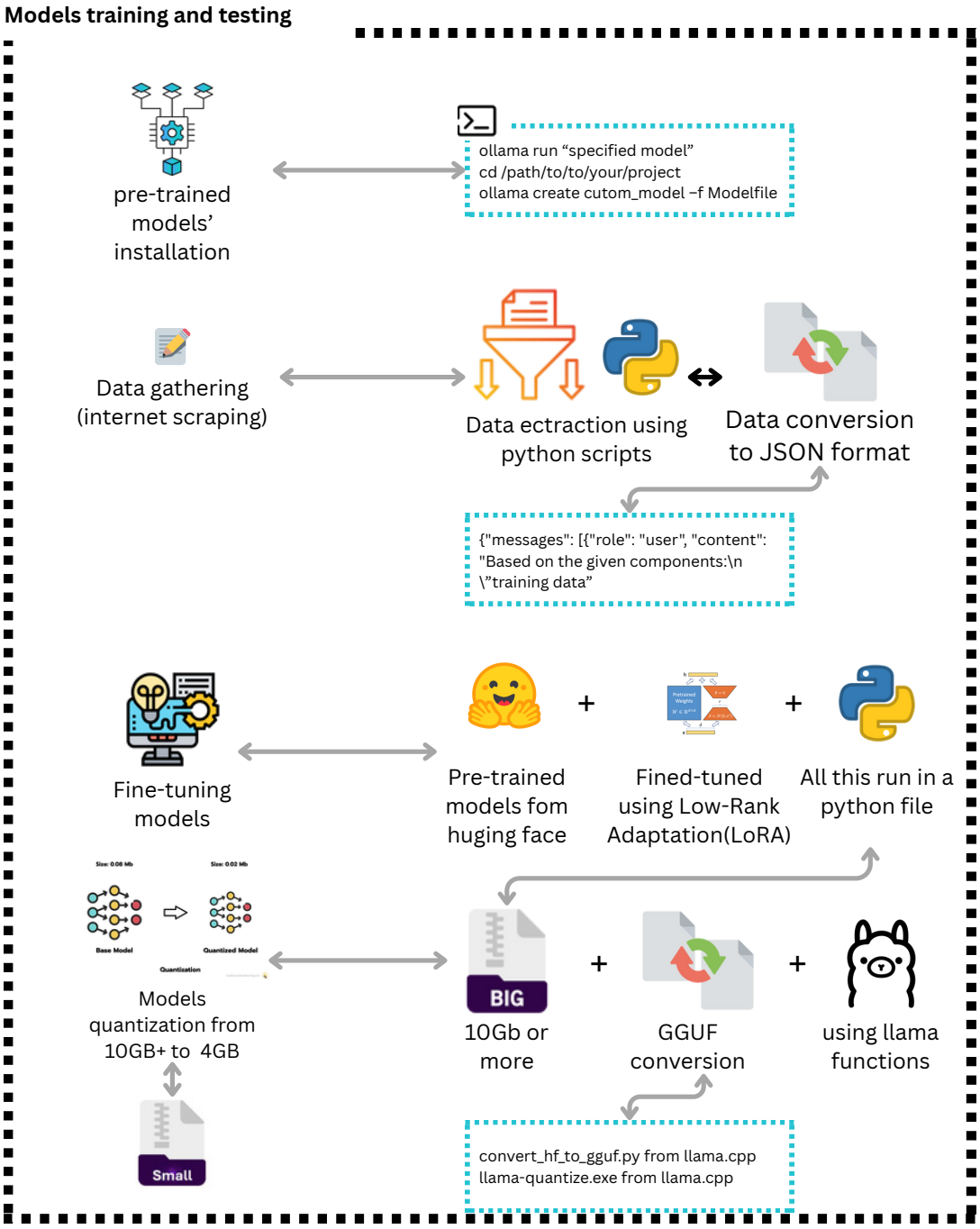

Figure 9. Flowchart of LLM training

Function	Purpose	Key Steps (abridged)
connect_db()	Connects to an SQLite database (benchmark_results.db) and ensures the table model_performance exists.	• Creates a table model_performance with columns for storing model name, accuracy, focus factor, inference time, setup complexity, and a timestamp. • Returns the database connection and cursor for further operations.
calculate _partial _accuracy (prediction, expected)	Calculates the accuracy of a model's prediction based on the number of keywords in the expected response that are found in the predicted response.	• Normalises the prediction and expected strings to lowercase. • Excludes predefined structural sections from the expected response. • Extracts keywords from the expected response and counts how many are found in the prediction. • Defines a dynamic threshold for 100% accuracy based on the prediction's word count. • Returns a partial accuracy score between 0.0 and 1.0.
get_setup _complexity (model_name)	Prompts the user to input a setup complexity score (1-10) for a given model.	• Continuously prompts the user until a valid integer between 1 and 10 is entered. • Returns the setup complexity score.
calculate_focus _factor (prediction, expected _structure)	Calculates a focus factor based on how closely the prediction matches the expected structure, ignoring content.	• Defines the expected structure format with specific sections. • Checks if each part of the expected structure is present in the prediction. • If all parts are present, verify the correct structure order. • Returns a focus factor score between 0.0 and 1.0.
evaluate_model (model_name)	Evaluates a single model by running it on the test dataset and calculating performance metrics.	• Iterates through the test dataset (TEST_DATA). • Depending on the model type: – For Python-based ML models (.py), uses the run_ml_model function to get predictions. – For specific moditem els like deepseek-r1:1.5b, uses the ollama.Client to generate predictions with a predefined system message. – For other models, generates predictions in JSON format and parses the output. • Calculates accuracy, focus factor, and inference time for each test case. • Returns a list of results for the model.
store_results (model_name, accuracy, focus, time, complexity)	Stores the performance metrics of a model in the SQLite database.	• Connects to the database using connect_db(). • Inserts the model's performance metrics into the model_performance table. • Commits the transaction and closes the connection.
Main Benchmarking Loop	Runs the benchmarking process for all models in the MODELS list.	• Iterates through each model in MODELS. • Calls evaluate_model() to get results for the model • Prompts the user for the setup complexity using get_setup_complexity(). • Aggregates accuracy, focus factor, and inference time across all test cases for the model. • Stores the results in the database using store_results(). • Prints the average performance metrics for the model.

Table 2. Core functions used in the benchmarking script.

2.2.2 Models Benchmark

This section involves testing and comparing the models. Figure 10 outlines the benchmarking process, which includes environment setup, function definitions, and data analysis. It will then explain the code structure in more detail.

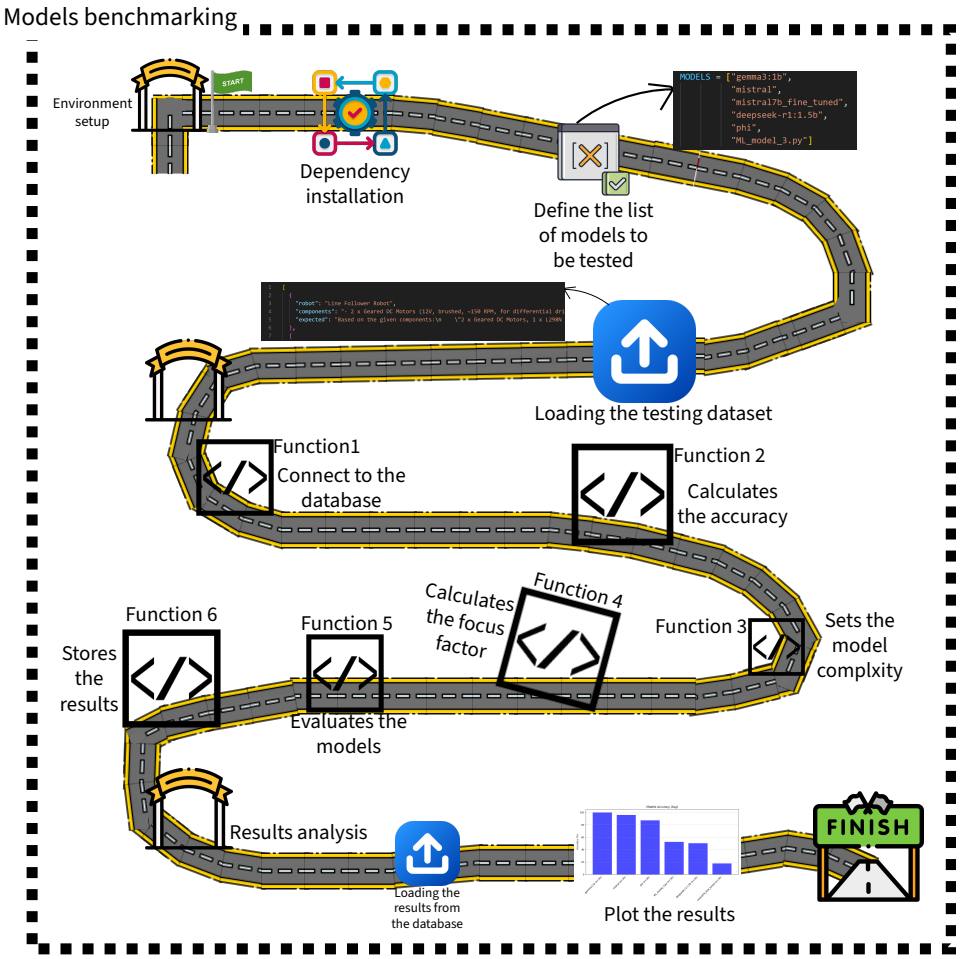

Figure 10. Models benchmark roadmap

Dependencies

1. `time`: Used to measure the inference time for each model by capturing start and end times.
2. `sqlite3`: Provides an interface to interact with an SQLite database, used for storing benchmarking results.
3. `ollama`: A library for interacting with the Ollama API for language model predictions.
4. `json`: Loads test data from a JSON file and parses JSON responses from models.
5. `warnings`: Suppresses specific warnings to avoid cluttering output.

6. `benchmark_ml`: A custom module containing the `run_ml_model` function for executing Python-based ML models.

With all these dependencies, the necessary functions were built for specific tasks. Table:2 presents a brief explanation of these functions:

Before detailing the process of customising the LLM model, it is important to explain how the decision to use one specific model was reached. This decision was based on a benchmarking process in which multipleLLMs and one ML model were evaluated. The models tested included: gemma3:1b, mistral, mistral7b_fine_tuned, deepseek-r1:1.5b, phi, and ML_model_3.py.

The performance metrics used in this benchmark were: accuracy, computational time, and the focus factor. In the following section, each of these parameters will be defined and their relevance to the model selection process explained.

Accuracy This was calculated using the following formula:

$$
\begin{aligned}
P &\coloneqq \text{predicted answer (lower-cased)},\\
E &\coloneqq \text{expected answer (lower-cased)},\\
S &\coloneqq \{\text{“Based on the given components:”...}\},\\
K &\coloneqq \{\, w \in \text{words}(E) \mid w \notin S \,\},\\
m &\coloneqq \big|\{\, k \in K \mid k \in P \,\}\big|,\\
n &\coloneqq \big|\text{words}(P)\big|,\\
T &\coloneqq \max\big(1,\ \lfloor 0.3\, n \rfloor\big).
\end{aligned}
$$

$$
\boxed{\text{Acc}(P,E) = \begin{cases} 1, & m \geq T, \\ \dfrac{m}{|K|}, & m < T. \end{cases}}
$$

Given that the model's output varies slightly with each execution, the likelihood of achieving a perfect word-for-word match is relatively low. To address this, a threshold-based accuracy metric was introduced. Specifically, if at least 30% of the words in the model's output match those in the expected output, the prediction is considered fully correct and awarded an accuracy of 100%. The threshold `T` thus grants full credit when 30% or more of the predicted tokens overlap with the informative keywords of the expected response. If this condition is not met, the accuracy is

calculated as a linear proportion of the matched keywords relative to the total number of keywords in the expected output.

Computational time This is simply the time it takes the model to generate the output.

Focus factor The formula used in the calculate_focus_factor function can be broken down into two stages, which determine the focus factor score based on how well the prediction string matches the expected structure, regardless of content.

Stage 1: Format Part Match The function checks for the presence of three expected format parts in the prediction:

- "Based on the given components:"
- "The most likely robot these components form is:"
- "I can do the following tasks:"

It counts how many of these parts are found in the prediction, then calculates:

$$\text{focus factor} = \frac{\text{format match count}}{3} \tag{1}$$

Stage 2: Structural Integrity Check Even if the three key phrases are present in the prediction, the order and nesting of those phrases matter. It's not enough to just include the right parts; they must also be arranged correctly.

If all three parts are found (i.e., focus_factor == 1.0), the function performs a deeper structure check by splitting the string in sequence:

```
split("Based on the given components:")
 then split by "The most likely robot these
 components form is:"
 then split by "I can do the following tasks:"
```

Only if each split yields exactly two parts does the focus factor remain at 1.0; otherwise, it is downgraded to 0.3.

So it can be said that if:

$$\text{focus factor} = \begin{cases} 1.0, & M = 3 \text{ and structure correct,} \\ 0.3, & M = 3 \text{ and structure incorrect,} \\ M/3, & M < 3, \\ 0.5, & \text{expected structure is empty.} \end{cases}$$

Where:

- M = number of format parts presented in prediction
- T = Total number of expected parts

Now that the performance parameters have been defined, it is time for a discussion of the types of models evaluated in this benchmark. The models can be broadly categorised into three main groups:

- The LLMs pulled from the Ollama website (`gemma3:1b`, `mistral`, `deepseek-r1:1.5b`, and `phi`). These are simply the models as they come, and are customised through a text definition before running the model, explaining how it should behave. Below is an example of this text-based description:

  > You are an automation system prediction assistant. Your task is to analyze the provided components and determine the most likely robot (or system) they form.
  >
  > Your output must be a JSON object with the following keys, and ONLY these keys:
  >
  > – `"Based on the given components"`: A list of components exactly as entered by the user.
  > – `"The most likely robot these components form is"`: A detailed 50 to 100-word description of the predicted robot, explaining how the components work together.
  > – `"I can do the following tasks"`: A list of specific tasks that the described robot is capable of performing.
  >
  > Ensure your response is a valid JSON object, without any additional explanations, reasoning, or preamble beyond the required format.

- One of the models evaluated was pulled from Hugging Face `mistral7b`, which was subsequently fine-tuned into `mistral7b_fine_tuned`. Unlike the other models that rely solely on pre-trained internet-scale datasets, this version was trained on a custom dataset curated through targeted web scraping. The intention was to feed the model only domain-relevant data, thereby improving task-specific performance. While this approach appeared promising in theory, it became

evident that a significantly large volume of high-quality data would be required to meaningfully influence or override the model's pre-trained knowledge. As a result, the performance of the fine-tuned model remained relatively mediocre in comparison to expectations.

- The ML model that was locally trained based on a given dataset. This dataset comprises 34 distinct robot types, each associated with a list of components and a corresponding description. Notably, the same dataset was also employed in benchmarking the performance of the LLMs to ensure consistency in evaluation. The dataset was loaded in JSON format, and its structure is outlined below:

```
{
  "robot": "Arduino Robotics Kit (Smart Car)",
  "components": {
    "DC Motors": [
      { "name": "Drive DC Motor", "count": 4 },
      { "name": "Geared DC Motor", "count": 4 },
      { "name": "12V DC Motor", "count": 4 }
    ],
    "Microcontroller": [
      { "name": "Arduino Uno", "count": 1 },
      { "name": "ATmega328", "count": 1 },
      { "name": "Arduino Nano", "count": 1 }
    ],
    "Sensors": [
      { "name": "Ultrasonic Sensor", "count": 2 },
      { "name": "IR Sensor", "count": 2 },
      { "name": "Line Sensor", "count": 2 }
    ],
    "Motor Drivers": [
      { "name": "L298N", "count": 2 },
      { "name": "L293D", "count": 2 },
      { "name": "TB6612FNG", "count": 2 }
    ]
  },
  "description": "A beginner-friendly smart
  car kit that employs four DC motors for
  movement, an Arduino Uno for control, and
  multiple sensors including ultrasonic and
  IR for obstacle avoidance and line tracking."
}
```

2.2.3 Software

Among the primary models, namely ML and LLM, certain models will be utilised in their original form, while others will undergo fine-tuning to potentially enhance their performance.

Now that the chosen model has been identified (refer to the Results and Discussion sections), let us delve into the methodology used to custom-train an LLM for this specific application. As illustrated in Figure 9, the process begins with the installation of a pre-trained Mistral model, which serves

as the foundational architecture for further customisation. Using a pretrained model significantly reduces the computational burden and training time, as the model already possesses a rich set of general linguistic and reasoning capabilities.

To adapt the model for the specific task of robotic component prediction, a domain-specific dataset was created by scraping and compiling relevant information from publicly available sources on the internet. This dataset forms the core of the training process, as it encapsulates the specialised knowledge required by the model. The quality, structure, and relevance of this dataset are critical; ultimately, the dataset's richness will determine the fine-tuned model's accuracy and reliability.

The fine-tuning itself was conducted using the LoRA technique, a parameter-efficient method that allows for modifying only a subset of the model's internal weights. This approach is particularly effective for resource-constrained environments, where full model training would be computationally prohibitive. After fine-tuning, the model undergoes quantisation to further reduce its size and make it more suitable for deployment on edge devices such as RPi.

Despite these optimisation strategies, the computational requirements for training a high-performing custom model remain substantial. Given the limited resources available(i.e. the dataset used for training), the custom-trained model demonstrated relatively modest performance. As a result, a more capable alternative, gemma3:1b, was adopted for final deployment, offering improved accuracy and generalisation thanks to its larger training corpus and architectural enhancements.

Once the components are identified, the robot is described, and the next step will be to use this description to generate the necessary code that will run the system. How is this achieved? Well, this is where the CGA comes in to use. For this project, the Roo Code extension was used inside VS Code. Roo Code is an AI-powered coding assistant designed to integrate seamlessly into VS Code, providing developers with a suite of intelligent tools to enhance their workflow. It functions as an autonomous coding agent, capable of reading and writing files, executing terminal commands, automating browser actions, and integrating with various AI models. Unlike traditional code completion tools, Roo Code offers multiple specialised modes, including Code Mode for general coding tasks, Architect Mode for planning and technical leadership, and Debug Mode for systematic problem diagnosis. One of its standout features is its ability to adapt to different development needs through customizable modes, allowing users to tailor its behaviour for specific tasks such as security auditing, performance optimisation, and documentation. Additionally, Roo Code supports multi-file edits, ensuring that changes are applied cohesively across an entire codebase. Developers can also integrate their own AI models or use local AI, avoiding vendor lock-in. The installation of this extension is quite straightforward. You need to first install the extension, then in the extension settings, you will find the areas where the registration for an API is made. After the registration(in this case using OpenRouter API) the desired AI model is chosen (in this case, google/gemma-3-27b-it:free).

Before proceeding further, it is important to note that a comparison was conducted between Roo Code and other competing tools. The evaluation focused on several key criteria:

- VS Code extension availability
- Device/platform compatibility (specifically, the ability to run on a RPi 4 Model B)
- Automation capabilities
- Customizability and availability of specialised modes
- AI model integration

These criteria were selected to ensure relevance to the project's requirements. The comparison results are summarised in Table 3, which serves to highlight the relative strengths and limitations of each tool under consideration.

The LLM outputs from the previous part are fed into the CGA chat box, from where the CGA will take the robot description and code-writing commands, then write the necessary code to run the robot. Figure 11 explains how this process will take place. It has to be mentioned that throughout the developing period there were tested different ways of establishing the link between the user and the CGA. Initially, the input was done directly through text, where the user would type in the text for the desired task.

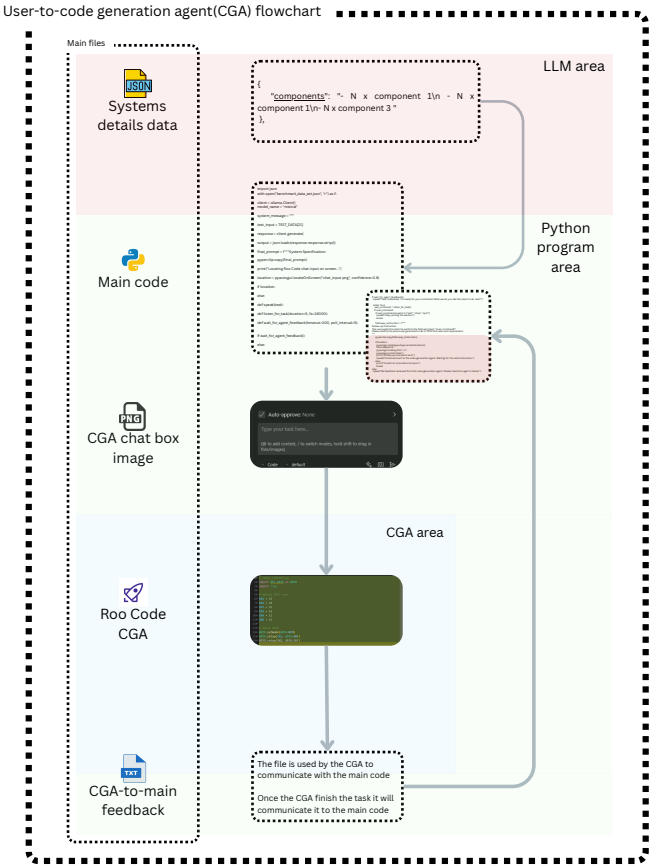

Figure 11. User-to-robot communication

Once the initial text-based interaction was successfully implemented, the next step was to enhance the communication process by introducing vocal input and output. To enable this feature, the main program was modified to incorporate voice-related libraries tailored to each operating environment.

In the Windows environment, the program utilized the pyttsx3 library for speech synthesis and the speech_recognition library for voice input. This combination allowed users to communicate with the CGA using spoken commands and receive verbal responses, creating a smooth and interactive experience.

For the RPi environment, alternative tools were required due to compatibility issues with pyttsx3. Initially, the program employed festival, an offline TTS system accessed via Python's subprocess module. However, to achieve significantly more natural and realistic speech output, the system was later upgraded to use Piper, a lightweight, offline TTS engine. Piper was integrated through its command-line interface, with text passed via standard input and audio played back using system utilities. Additionally, the sounddevice library was used to record microphone input, offering better integration with RPi hardware. The speech_recognition library remained in use for interpreting vocal commands across both platforms.

During the early development stages, the code was executed on a standard computer. However, for deployment, the complete program had to be transferred to a RPi, raising concerns about the device's limited computational resources. While the CGA, relying primarily on a stable internet connection, was expected to function reliably, the main concern centred around the ability of the LLM to effectively interpret the robot's structure and behaviour within the constraints of the RPi.

To address this challenge, the results from the benchmarking phase were used to identify a model that balanced performance with efficiency. Based on this evaluation, the gemma3:1b model was selected, as further detailed in the benchmark section.

The results obtained from this phase were notably positive, with communication between the user and the CGA functioning smoothly and without significant errors. While some trial and error was necessary during the initial stages, the system ultimately achieved stable performance. Once operational, the software appeared well-prepared for integration with the hardware component of the project.

Agent/Tool	VS Code Extension	Compatibility (incl. RPi4)	Automation Capabilities	Custom Modes / Personas	AI Model Integration
Roo Code	Yes (Marketplace extension). Open-source.	**High**Cross-platform (works on RPi4 with VS Code).	**HighHigh** – Reads/writes files, creates files, runs terminal commands, controls browser. Can automate multi-step tasks (code-gen, test, debug loops).	**HighYes** – Multiple built-in modes (Code, Debug, Architect, etc.) and unlimited user-defined custom modes. Personalities can be tailored.	**HighFlexible** – Supports OpenAI API and custom endpoints. Can use **local models** for offline use (if provided via API or MCP).
GitHub Copilot (Agent Mode)	**High**Yes (Copilot Chat/Agent extension). Proprietary (subscription based).	**High**Cross-platform (runs on RPi4 under VS Code, needs internet).	**High** – With *Agent Mode*, can edit multiple files, run terminal commands, respond to compiler errors in a loop. No web browsing, but integrates with build/test feedback.	**No** – No user-defined modes; single agent behaviour (context-aware but not user-customizable).	**Cloud-only** – Uses GitHub's cloud (OpenAI/Anthropic models). **No local model** support. Requires online access.
Continue.dev	Yes (VS Code extension). Open-source.	Cross-platform (works on RPi4; light extension, model can be remote).	**High** – Autonomous agent with tools: file read/write, create files, project search, run terminal, web search. Can perform multi-step coding tasks with minimal input.	**Partial** – No named "modes," but highly customizable via rules and prompts. Users can craft custom instructions and tool-use policies.	**Flexible** – Works with OpenAI, Anthropic, etc., or **local models** (via API e.g. Ollama). Ideal for offline or self-hosted setups.
Sourcegraph Cody	Yes (VS Code extension). Free & enterprise tiers.	Cross-platform (runs on RPi4 with VS Code). May use Sourcegraph Cloud or self-hosted (x86) for indexing.	**Moderate** – Excellent code insight and generation (uses entire repo context). Can modify code via chat commands (with user approval). **Not fully autonomous** – no direct terminal control or multi-file edits without user prompt.	**Limited** – No multi-persona modes. Supports saved prompts and intent detection, but not user-defined agent personas.	**Semi-flexible** – Defaults to cloud models (OpenAI/Claude). Allows **model selection** (Claude, GPT-4, etc.) and even **local model integration** via experimental config.
Codeium	Yes (VS Code extension). Free for individuals.	Cross-platform (works on RPi4; uses cloud service by default).	**Low** – Focus on code completions and one-file suggestions. Has chat and "AI commands" for refactoring/explanation, but **no autonomous multi-step execution**. No direct file system or shell access in agent.	**No** – No concept of modes/personas. Some configurability in suggestion style, but not role-based.	**Cloud or Self-Hosted** – Uses Codeium's cloud models (based on **Meta's Llama 3.1** fine-tunes) or can be self-hosted on x86 servers. **No local on-device model** support for ARM yet.
AWS CodeWhisperer	Yes (via AWS Toolkit extension).	Cross-platform (VS Code on RPi4 supported, internet required).	**Low** – Real-time code completion similar to Copilot. Now includes an inline chat for guidance. **No autonomous file ops or command execution**.	**No** – Operates as a simple assistant, no custom modes.	**Cloud-only** – Utilizes AWS-managed models. No local deployment.

Table 3. Feature comparison of AI-powered coding agents for VS Code.

An example of a prompt sent from the LLM to the CGA is provided below. The structure of this prompt was defined during the LLM training process (see Section 2.2.2), and it clearly demonstrates the model's ability to follow instructions accurately. Notably, the list of components and their connections is repeated twice within the prompt to ensure that all necessary information is effectively communicated to the CGA, as LLMs may occasionally omit specific details. The complete version of this prompt is available in Appendix A.

```
<task>
System Specification:
{
  "Based on the given components": [
    "Raspberry Pi 4 Model B",
    "L298N Motor Controller 1",
...
  "The most likely robot these components form is":
...
  "I can do the following tasks": [
    "Navigate an environment",
...
Instruction to Agent:
Write a complete control code to operate the system
...
```

2.2.4 Communication establishment with the CGA

One thing that has to be mentioned is that the main code (LLM_text_extract.py, see Ammendix D) was used on both operating systems (Windows and Linux). However, the program execution was different, while on Windows the program was manually executed since this was part of the developåment, on Linux the program had to be self-executed by the OS. To achieve this, several additions were implemented in the main code, but also in the RPi boot-up settings.

The main code modification consists of one extra use of the "pyautogui" function. This was used to access the Roo Code extension in VSCode upon system bootup.

The RPi boot-up configuration was updated to ensure the autonomous execution of the robot control system using VS Code's native task automation. A dedicated task, defined in the workspace-level `tasks.json` file located in `/home/GenAI/.vscode`, is configured with the `"auto": true` attribute to trigger automatically upon opening the folder. The task launches a shell that activates the Python virtual environment and executes the script `LLM_text_extract.py`. A short startup delay (via `sleep`) is added before execution to ensure all extensions and dependencies are fully loaded.

```
cd /home/GenAI
sleep 5
(.venv)
python3 LLM_text_extract.py
```

To initiate this process at boot, VS Code is launched automatically by adding the following line to the autostart configuration file:

```
~/.config/lxsession/LXDE-pi/autostart
@code --folder-uri file:///home/GenAI
```

This setup ensures that VS Code opens the correct project folder on startup, which in turn triggers the automated task. Previous automation mechanisms, including the shell script `start_robot_control.sh` and any associated `systemd` services, were removed to eliminate redundancy and potential race conditions.

During the initial launch, a keyring-related warning was encountered. Selecting the "Use weaker encryption" option allowed the Roo Code extension to persist its API token without relying on the OS keyring, thereby avoiding repeated disconnection errors. With this configuration, the virtual environment is automatically activated, the Roo Code extension is initialized, and the script executes on boot, achieving seamless and fully automated system startup. A detailed breakdown of this setup is provided in Appendix C.

2.2.5 Hardware

Now that the software component has been validated, attention turns to the hardware. Two robotic setups were developed to test and demonstrate the functionality of the software. The first setup consists of a simple circuit, which is presented in the following section. The second setup involves a small four-wheel robot designed to showcase the system's capabilities and illustrate potential use cases. While the robot itself is relatively simple, it effectively serves the purpose of highlighting the flexibility and applicability of the developed system. Detailed descriptions of both robotic setups are provided in the subsequent subsections.

Build the first robot testing setup

The first testing setup consisted of a simple circuit (see Fig. 12) without a physical robot body. It included only two basic DC gear motors mounted in an improvised fixture. The primary objective of this setup was to verify whether meaningful interaction between the digital and physical domains could be established, specifically, whether commands generated by the software could successfully control physical hardware components. Due to this focus, a minimal and straightforward setup was deemed sufficient. This configuration also formed part of the initial feasibility study conducted earlier in the project.

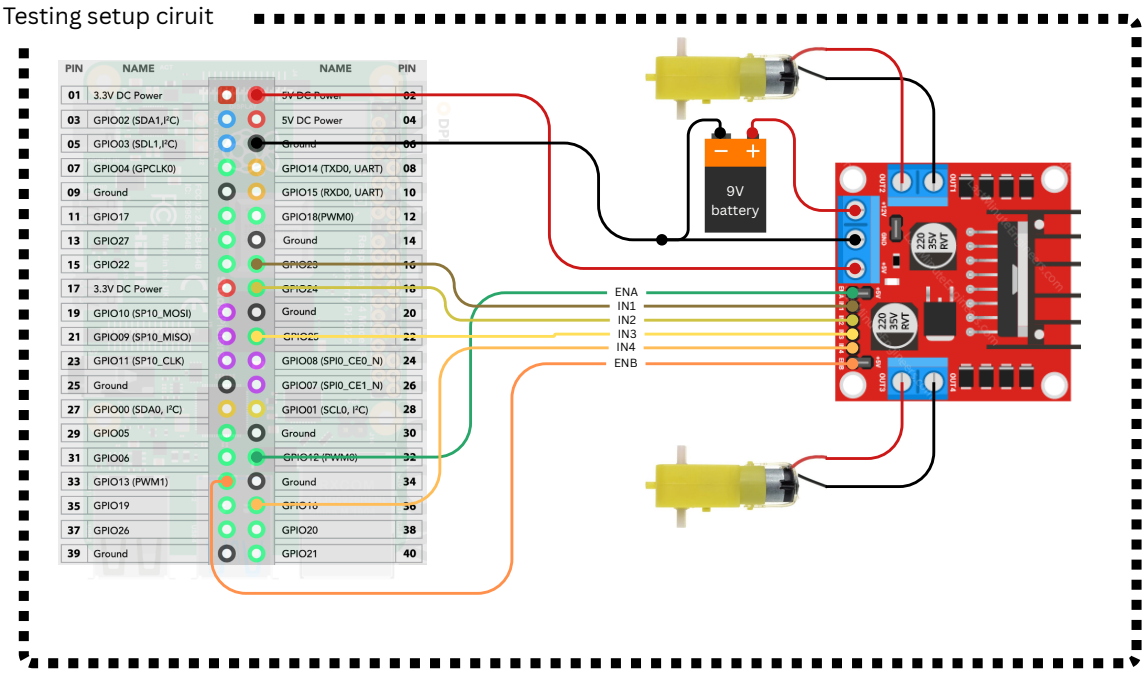

Figure 12. The circuit diagram of the first testing setup

Building and testing the final robot

As outlined above, the final testing platform (see Fig. 14, 16, 17) is a four-wheel robot. This configuration was chosen due to its simplicity and ease of control, aligning with the primary objective of the project, to demonstrate the core hypothesis that AI can extend its influence from the digital environment into the physical world. By selecting a straightforward and manageable robot, the focus remains on validating the software's capabilities and illustrating the practical feasibility of the proposed approach.

The robot is equipped with a USB camera, a USB microphone, and USB speakers. The use of USB peripherals was a deliberate design choice, primarily due to the fact that the RPi lacks a built-in ADC. Handling analog signals would have required the integration of an external microcontroller, such as an Arduino Nano, to process the analog inputs. To simplify the system architecture and reduce complexity, this requirement was bypassed in favour of USB-based peripherals, which operate entirely in the digital domain. The complete list of components used in the final robot build is provided in Table 4, and the corresponding circuit diagram is shown in Fig. 13.

Category	Components
Structural components	- The main chassis, made out of PLA (Polylactic Acid) - The cover with mounting slots for fan, microphone, and speakers, also PLA - The four wheels
Electrical components	- The Raspberry Pi 4 Model B - Two L298N motor drivers - One 14.8V LiPo battery - One voltage regulator - One USB microphone - Two USB speakers - One USB camera - One 9G SG90 servomotor - Three HC-SR04 ultrasonic sensors

Table 4. Complete list of the components that form the robot

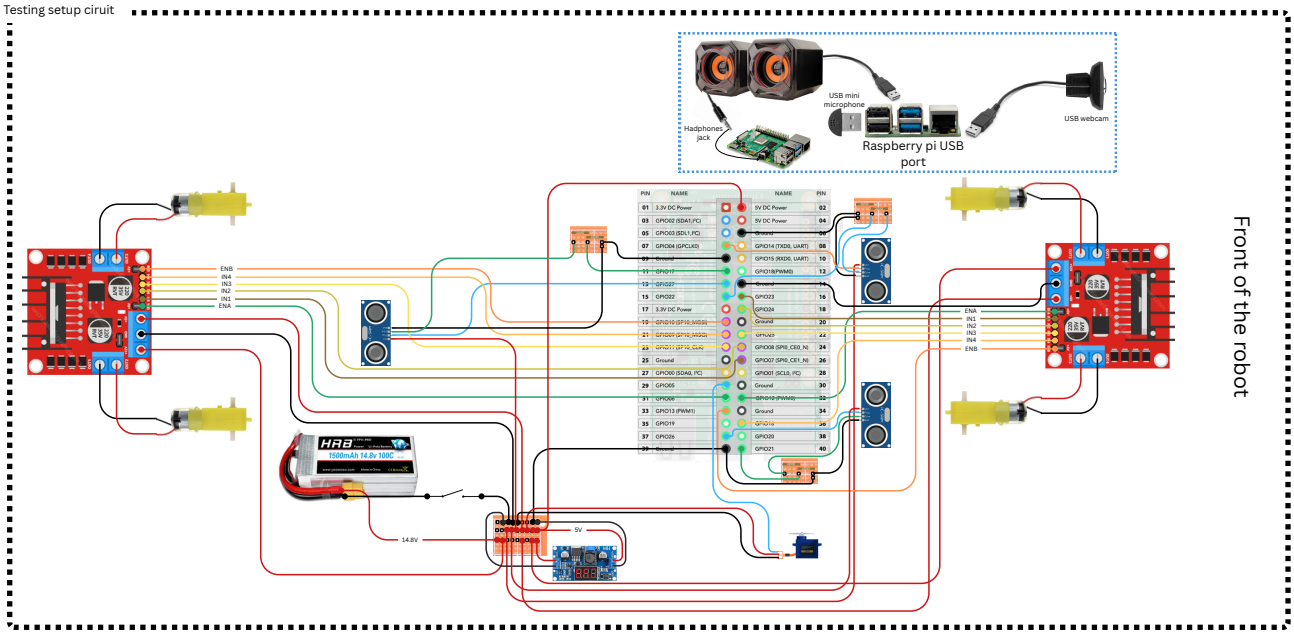

Figure 13. Testing robot circuit diagram

The Computer-Aided Design (CAD) of the robot was completed entirely using Fusion 360. The final models were exported in STL format to prepare them for the slicing process required for 3D printing. It is important to note that, due to the size of the robot's structure exceeding the available printing volume of the 3D printer, the design was divided and printed in two separate parts.

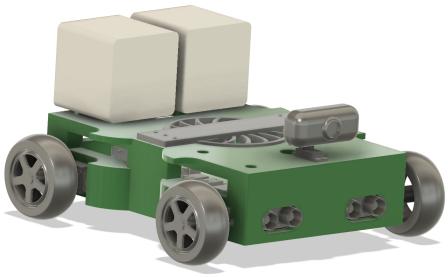

Figure 14. Testing robot assembly-in CAD

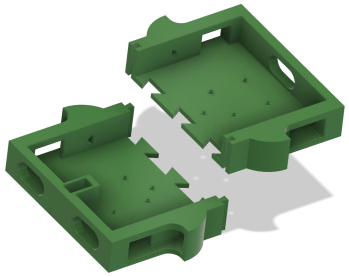

Figure 15. Perspective view of the use of bowties in structure assembly

To ensure a strong and stable connection between the two printed parts, five bowtie connectors were used, as illustrated in Fig.15. The STL files were then processed using Creality Print 5.0 for slicing. The slicing settings were largely left at their default values, with the primary modification being the enabling of support structures. The same printing and assembly method was also applied to the robot's lid. Once all parts were printed, they were press-fitted together, resulting in a robust and cohesive structure, as shown in Fig.18.

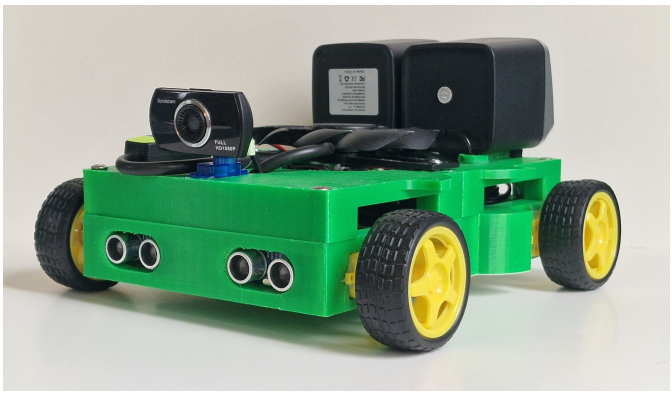

Figure 16. Final testing robot front view

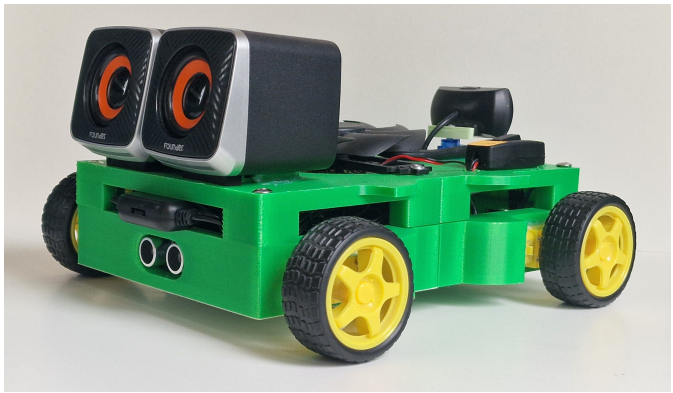

Figure 17. Final testing robot back view

2.2.6 Software and Hardware Integration

To make sure the testing would succeed, several "forced" interventions were made. The most significant being the use of a specific ML model for image recognition. This was done to avoid complications in the code generation section, where the CGA will have to pull one of such models from the internet, which could introduce errors in the execution. Therefore, a GitHub link was provided from which the model could be extracted. The model used was a TensorFlow Lite version of SSD MobileNet V1 trained on the COCO dataset, a ML model designed for object detection. SSD (Single Shot Multibox Detector) is a deep learning architecture that predicts object locations and classes in images in a single forward pass. MobileNet V1 is a lightweight convolutional neural network optimised

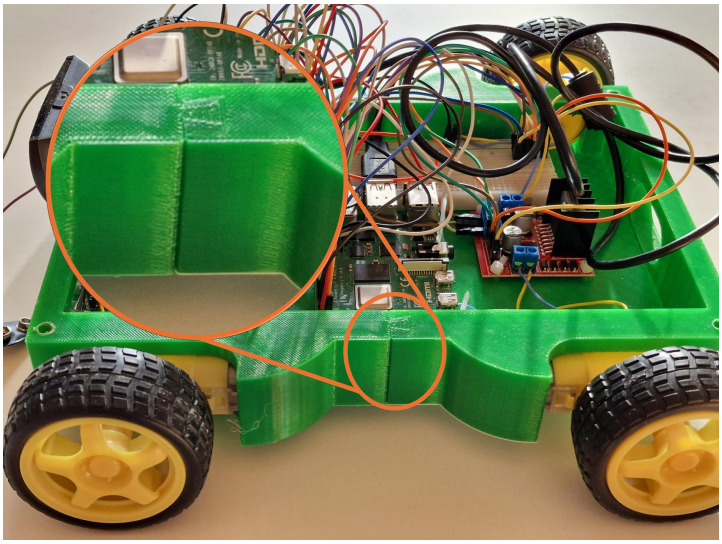

Figure 18. Robot printed parts assembly

for efficient performance on mobile and embedded devices. The COCO dataset provides annotated training data for 90 common object categories, including “apple”(Fig.19). This model was converted to TensorFlow Lite format for deployment on resource-constrained environments and manually uploaded to a GitHub repository as a release asset. A direct download link was then generated via the GitHub Releases system, ensuring that the model could be retrieved programmatically at runtime using a stable URL.

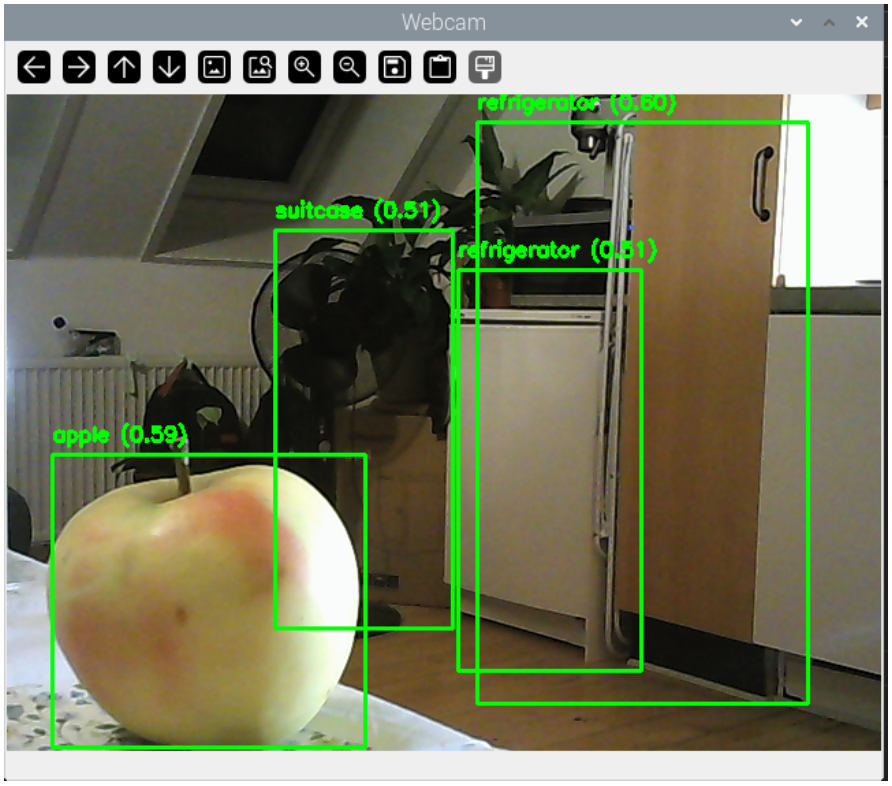

Figure 19. Object detection model functionality check

3 Results

This section presents the findings from each of the areas of study explored in this thesis. It encompasses both the components that were successfully implemented and those that, while not fully realised within the scope of this project, yielded valuable insights. These insights contribute to a broader understanding of the challenges and opportunities in the field and may serve as a foundation for future research and development efforts.

3.1 Components identification system

As the results for this section have already been presented in the feasibility tests, this part will provide only a brief summary, with further analysis included in the Discussion section. The feasibility study yielded several important observations. As shown in Fig.7, it was concluded that the state of the RPi's GPIO pins could not reliably be used for component identification. However, data presented in Fig.5 and Fig. 6 suggest a potential alternative approach, wherein an accelerometer could be used to identify components based on vibration patterns. Although these initial results are promising, it is important to note that the method was tested only on a single component type, the DC gear motors. As such, generalising the results across a broader range of components would be premature and potentially misleading.

3.2 Benchmarking of the models

As shown in Fig. 20, the models most suitable for use in the subsequent phases of the project were those available through the Ollama platform. Given that the primary focus of this thesis is to investigate the potential for AI to extend its capabilities from the digital environment into the physical world, it was essential to select a model that not only delivered strong results but was also quick and straightforward to implement. Figure 20 was instrumental in guiding the model selection process. As described in the Methodology section, accuracy was calculated using a specific threshold-based approach tailored to the nature of LLM outputs. While this may make the models appear to perform exceptionally well, it is important to emphasise that the same evaluation method was consistently applied across all models. Therefore, the relative ranking remains valid regardless of the specific metric used. Examining the results, it is evident that `gemma3:1b` achieved the highest accuracy score. However, in terms of focus factor, there was little variation among the models, with the exception of `deepseek-r1:1.5b`, which showed noticeably different behaviour. The reasons for this deviation will be further explained in the Discussion section. Additionally, setup complexity was generally consistent across all models, except for the ML model, which required a more time-intensive training process.

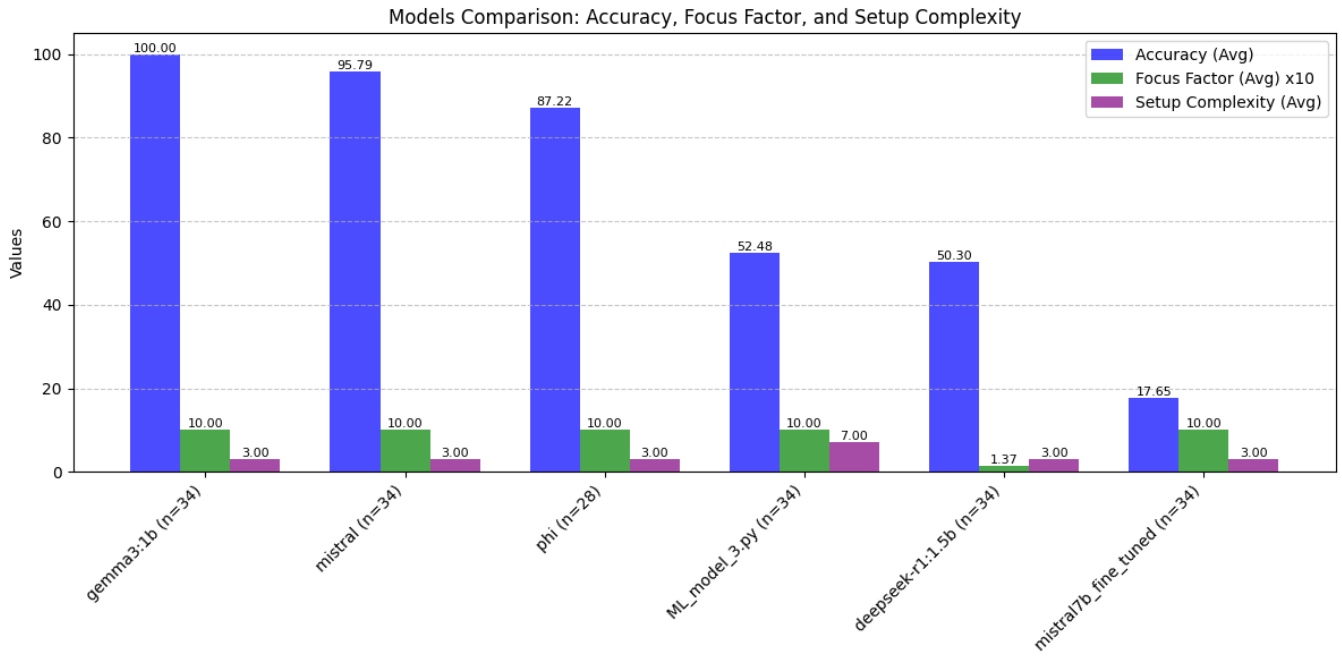

Figure 20. Ml and LLM models bencmark chart

3.3 Robots tests

First testing robot

The findings from this stage were highly promising, as the system operated reliably with no major errors or need for human intervention. After initiating the Python script, the process followed the steps outlined in the flowchart shown in Fig. 11. The LLM successfully predicted the most likely robot configuration, and the resulting description was transmitted to the CGA chat interface. From there, the CGA generated the corresponding code required to operate the system.

The generated code compiled without issues, and the two DC gear motors rotated the wheels in a controlled and expected manner. This outcome significantly increased confidence in the feasibility of the proposed approach, suggesting strong potential for its application in more complex systems. Based on these results, it was concluded that this test was a clear success.

The results obtained from implementing vocal communication were also highly encouraging, both on the PC and the RPi platforms. Communication between the user and the CGA functioned smoothly and without notable errors. While some trial and error was involved during the initial stages, once the system was fully operational, the software demonstrated strong reliability and appeared ready for integration with the hardware.

Figures 21, 22, and 23 illustrate one such interaction. In this example, the CGA has already generated the base code, and the user subsequently issues a vocal command to modify the robot's behavior.

This command is received and processed by the CGA, as shown in Fig.11, after which the CGA begins updating the base code in accordance with the user's request (Fig.23). The complete sequence of operations executed in this example is detailed in Fig. 21.

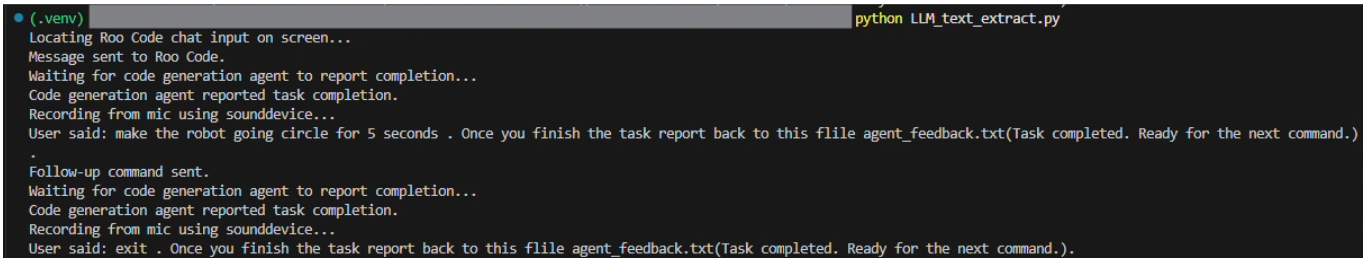

Figure 21. Prints of the main processes run during the program execution

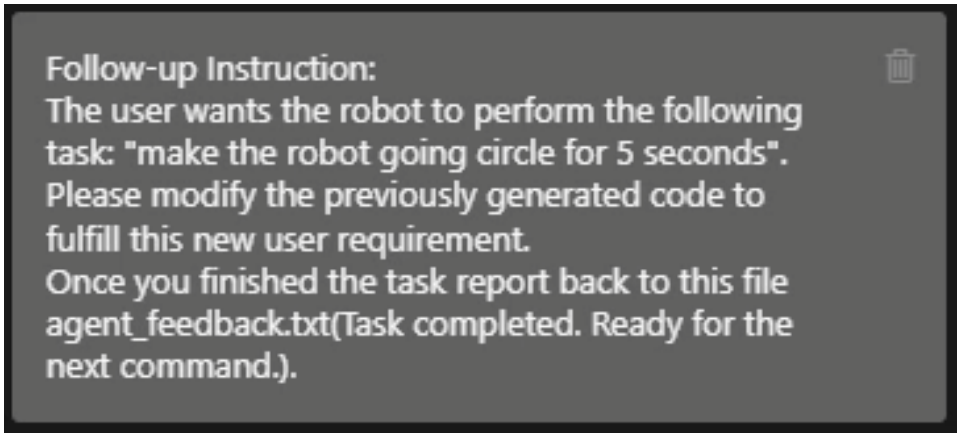

Figure 22. The message received by the CGA coming from the user

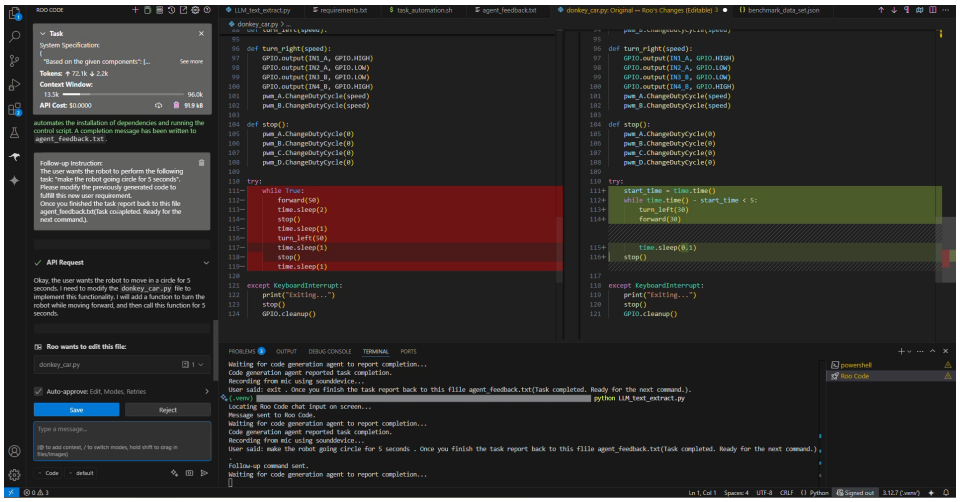

Figure 23. Print screen of the process where the CGA modifies the code based on the user input

Final testing robot

The primary objective of the final test was to evaluate whether the system could operate as a fully autonomous process. To achieve this, several critical aspects had to be addressed: the system should automatically boot into the VS Code virtual environment, it should self-navigate to the CGA, it should be capable of autonomously initiating communication with the user, it should be able to respond dynamically to user input, and it should execute the generated code autonomously to control the robot. In the following sections, the results corresponding to each of these key criteria will be presented. The results are presented through a series of figures, each corresponding to one of the defined criteria. Each figure visually demonstrates that the respective functionality operates as intended, accompanied by a brief description outlining the associated process.

Automatically boot into the VS Code environment

To demonstrate the results corresponding to this criterion, several screenshots were captured at different time intervals during the system boot-up process. It is important to note that the RPi's GUI was streamed wirelessly using the RealVNC Viewer application. This remote streaming introduced additional processing overhead, which led to slower performance on the RPi.

Figure 24.(a),(b), and (c) illustrate the sequence of events. In the initial stage following boot-up, the GUI appears as a black screen, indicating that the desktop environment has not yet fully loaded. As the initialisation process progresses, VS Code begins to launch and its GUI becomes visible. This is followed by the activation of the Python virtual environment and, finally, the execution of the main control script. These steps collectively confirm that the system initialises successfully and autonomously, despite the performance limitations introduced by remote desktop streaming.

Self-navigate to the CGA

Figure 24.(d) presents the prompt in which the main program attempts to locate the Roo Code extension within the VS Code interface. Once the extension is detected, the program proceeds to access it automatically as part of the initialisation sequence.

Autonomously interactive communication with the LLM

Figure 24.(e) and (f) illustrate a prompt from the main program that demonstrates LLM-to-CGA communication. The prompt clearly shows that the LLM's output was accurately captured and successfully sent to the CGA chat box. Additionally, it provides visual confirmation that the main program correctly processes the LLM's recipe (for the entire system's generated recipe, see Appendix B) and forwards it to the CGA for further handling.

Figure 24. System-bootup print screens at different time steps

Respond dynamically to user input

Figure24.(g) and (h) illustrate a prompt from the main program that demonstrates user-to-robot communication. The prompt clearly shows that the user's voice input was accurately captured and successfully converted into text. Additionally, it provides visual confirmation that the main program correctly processes the speech-to-text output and forwards it to the CGA for further handling. This interaction highlights the system's effective integration of voice-based commands within the control workflow. Following this, the CGA receives the user input (as shown in Fig.24.(g) and begins modifying the existing code to accommodate the user's request. Figure24.(h) clearly demonstrates that the user-to-robot communication loop has been successfully completed, with the only remaining step being the execution of the newly generated code. This illustrates the system's ability to adapt the robot's behaviour dynamically based on real-time user input.

Execute the generated code autonomously to control the robot

And lastly but not least, and most importantly, the CGA executes the generated code directly within the terminal, resulting in the control of the physical robot. Figure 25 illustrates how the generated code successfully interacts with the robot's hardware, including retrieving sensor data from an ultrasonic sensor, which is returned in centimetres. This final stage of execution clearly demonstrates the core premise of this thesis: that has the capability to extend beyond the digital environment and actively influence the physical world through robotic systems.

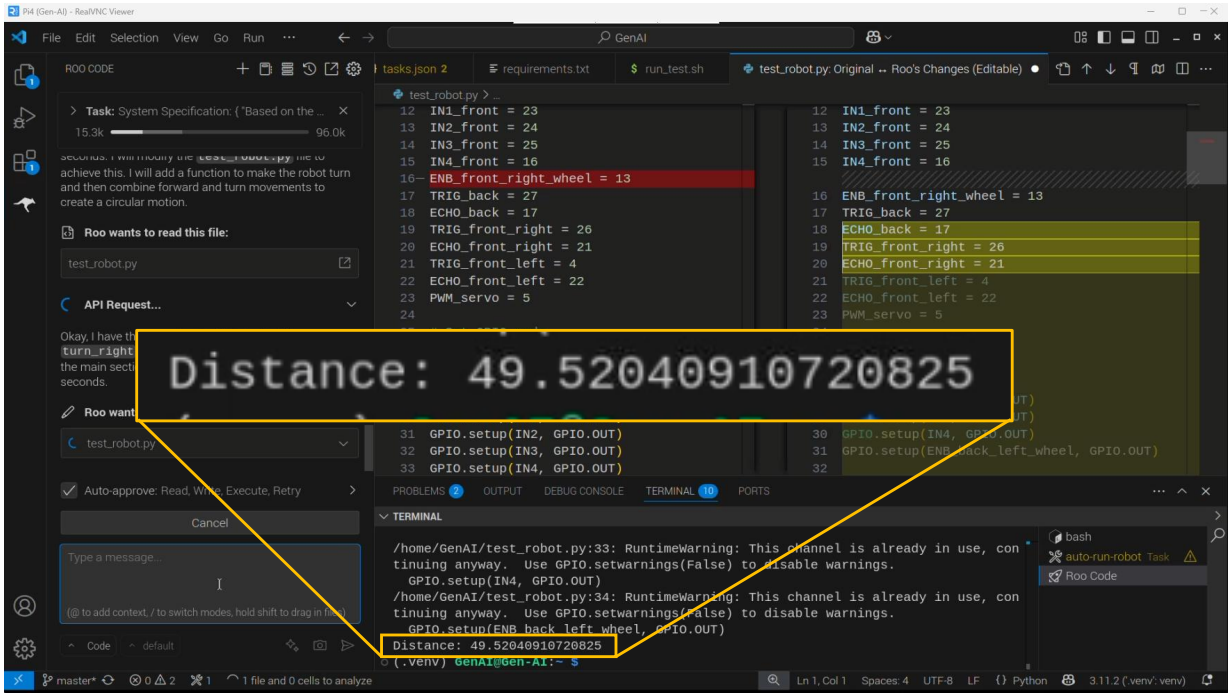

Figure 25. Prompt showing the readings of the sensors based on the generated code

4 Discussion

This section provides an interpretation of the results, discusses their broader implications, and compares the findings with existing literature. It addresses the observed outcomes, offers possible explanations, outlines limitations encountered during the project, and proposes directions for future research to further explore this topic. With that framework in mind, let us begin with a discussion of the results related to component identification.

4.1 Methodology

The Methodology section represents one of the most comprehensive parts of this thesis, and it is worthwhile to highlight several key points related to its structure and focus. As illustrated in Fig. 2, the project was initially divided into multiple subtasks, each of which is discussed in detail within the Methodology section.

One point of discussion involves the distribution of effort and focus across different areas of the project. Although no formal time-tracking was conducted, the level of detail presented in the methodology offers insight into how project priorities evolved. Specifically, the early-phase tasks, such as peripheral identification and system description, were eventually deprioritised following the decision to focus more heavily on code generation and AI integration. As a result, these earlier components are described with relatively less detail.

In contrast, the second half of the project, which explores the integration of AI models into embedded systems for real-time code generation, is covered more extensively. This reflects both the great complexity of the task and its central importance to the overall research question. The increased level of technical depth in this area also provides valuable insights into the practical challenges and opportunities of deploying AI within resource-constrained embedded environments.

4.2 Results

Components identification system

As presented in the results section, the outcomes of this work were mixed; some aspects demonstrated clear limitations, while others revealed promising potential. One of the less successful approaches involved using the pin status of an embedded board to identify connected components. This method proved unpromising, primarily due to its dependence on the capabilities of the embedded platform. For instance, the RPi lacks a built-in ADC [17], which significantly complicates the task. The test results confirmed that the RPi's GPIO pins report only binary values (0 or 1) [17]. Once all pins are in use, the outputs become uniformly high or low, making it virtually impossible to distinguish between different connected components based solely on pin state.

However, this limitation is not inherent to the approach itself, but rather to the hardware used [17], [13]. On embedded boards that do include ADCs, such as Arduino boards, this method could be viable [17]. In such cases, each component could be associated with a unique electrical "fingerprint" generated in response to a predefined input signal. This concept aligns with the approach proposed in [15], where component identification is facilitated through analogue signal profiling. Therefore, while not feasible with the RPi in its current configuration [17], this method remains a potential avenue for future exploration on more capable hardware platforms [13].

Benchmarking of the models

Regarding the benchmarking of models, several key points are worth further discussion, particularly with respect to the interpretation of the results [4], [11]. From the outcome, it is evident that the models sourced directly from the Ollama platform demonstrated the strongest performance across the evaluated criteria. However, as previously mentioned, one of the challenges encountered during fine-tuning was the need for a sufficiently large and high-quality dataset [18]. Constructing such a dataset requires a considerable time investment [18], which would likely have conflicted with the overall timeline of this project.

Given that the primary objective of the thesis was to demonstrate the ability of AI to interact with the physical world [3], the fine-tuning of models was considered a secondary priority [3]. The pre-trained Ollama models already delivered results that were consistent with the performance expectations necessary for conducting the system-level tests [3]. Nonetheless, it is important to acknowledge that with access to a more extensive and task-specific dataset [3], a fine-tuned model could potentially surpass the performance of the pre-trained Ollama models for this particular application [8]. This represents a valuable direction for future research [7].

Another discussion point is the reason why deepseek-r1:1.5b scores so low on the focus factor parameter. This is because the model is a reasoning one, which leads to a lot of extra text(the reasoning text) that was not supposed to be in the final output. Several prompts were tried out with the scope of providing the model to output the reasoning text, but the model kept doing that, resulting in a deviation from the set output structure. But if this reasoning were to be deleted, the model output would have a similar result to the other models.

Another important aspect to address in the model benchmarking process [3] is the method used to calculate accuracy [18], [14]. As explained in the Methodology section, the nature of LLMs is inherently non-deterministic [4], meaning that the same input may yield slightly different outputs on different occasions [7]. This variability, combined with the use of a fixed benchmarking dataset [6], upon which some models may have been trained or evaluated [4], renders the use of traditional exact-match accuracy metrics [18] potentially misleading [18].

Expecting a model to achieve 100% exact-match accuracy under such conditions is unrealistic, as even semantically correct answers may differ in phrasing or structure [4]. To address this limitation, a threshold-based keyword matching approach was adopted. In this method, a prediction is considered fully accurate if at least 30% of the keywords from the expected output are found in the model's response. If this threshold is not met, accuracy is calculated proportionally based on the number of matched keywords. This approach provides a more flexible and meaningful evaluation of model performance, while still maintaining consistency across all benchmarked models. Further details regarding the implementation of this method are provided in the Methodology section.

Following PANDA's analysis of open-domain question-answering (QA) metrics [19], which identified a 30% token-overlap threshold as yielding the highest correlation with human judgment across various answer styles, the same 30% threshold was adopted in this study for evaluating robotics component descriptions.

First testing robot

As noted in the Results section, the initial test yielded surprisingly positive results. This outcome strongly indicates the feasibility of achieving the core objective of the project, namely, demonstrating that AI can interface with and control elements of the physical world through robotic systems. The successful generation of functional code by the CGA, which enabled two DC gear motors to operate in a controlled manner, serves as clear proof of concept.

It is important to qualify this success by acknowledging that the system used in this test was relatively simple in terms of complexity. Nonetheless, the fact that the AI-driven pipeline from description to code execution functioned as intended provides a solid foundation for continued exploration. While the result represents only an early-stage implementation, it is sufficiently robust to justify further development and testing in more complex scenarios.

Final testing robot

The main points of discussion regarding the final testing robot pertain to the execution of the test procedures. As illustrated in Fig. 24(e), the slowest step in the process involves the loading and execution of the local LLM, which results in a noticeable time delay, denoted as ΔT. This performance bottleneck was anticipated, given the limited computational capabilities of the RPi 4 Model B.

While the system operated within acceptable bounds, there is clear potential for improvement. Performance could be significantly enhanced by either upgrading to more powerful hardware or by deploying a smaller, purpose-built LLM that is optimised for the specific tasks required by the robot. This trade-off between model complexity and system responsiveness should be considered in future

iterations of the system, particularly if real-time performance becomes a critical requirement.

One other thing to point out is that for the final test, the LLM model gemma3:1b was replaced by the phi model. This was because, despite the gemma3:1b results from the benchmark, when put into a real test, the model did not perform as good as in the benchmark. This shows that there could be some more improvements in the benchmarking to better assess the model's performance.

Finally, let us discuss the final test results. Here, man can say that the results are a successful proof of the concept that AI is able to reach out from the virtual environment into the physical environment. However, although the system did perform well, it is important to acknowledge that several aspects still require further development before man can confidently say that AI has fully achieved this transition. As explained in [4], one crucial criterion of such an autonomous system is the system's ability to specify and understand its configuration, in other words, the list of components that form the system and the system's description. As already mentioned, in this thesis, the list of components was assumed to be given; therefore, this could be one area of further work that can be done so that the entire system will be running autonomously. Addressing these limitations will be critical to moving closer to a fully autonomous AI-driven system capable of seamlessly integrating with physical hardware in dynamic, real-world environments.

4.3 Ethics and safety considerations

Given the significant potential for social disruption in the fields of Gen-AI and robotics, it is a matter of common sense and academic responsibility to include a dedicated section on ethical and safety considerations. This section is divided into two primary subsections. The first addresses the technical risks and safety concerns associated with the proposed system, while the second discusses the broader societal implications that this technology may introduce. Together, these subsections aim to promote responsible development and deployment by evaluating both the immediate and long-term consequences of AI-driven robotic systems.

Technical concerns

As outlined throughout this paper, the proposed system relies on cloud-based operations, which inherently introduces concerns regarding cybersecurity. The dependence on external servers may expose the system to potential cyberattacks, data breaches, or service disruptions. To mitigate these risks, it is recommended that a local server infrastructure be deployed. Hosting the core components, such as the CGA and the associated AI models, on a local network would reduce the system's exposure to external threats and improve overall security.

Another technical consideration involves the integration of human-robot interaction. In industrial settings, robots are typically equipped with extensive safety mechanisms designed to prevent harm to

humans. In contrast, the scenario presented in this thesis does not yet address such safeguards, as the focus is primarily on demonstrating a proof of concept. Nonetheless, it is important to acknowledge potential safety measures that could be incorporated in future implementations.

For instance, designated operational zones, similar to those used in factories for fixed-arm robots, could be established for mobile robots. Such defined areas would restrict human access during operation and help minimise the risk of accidents. Another practical measure could involve equipping robots with auditory signalling systems, such as beeps or verbal alerts, to notify nearby individuals of their presence.

Although these safety features were not implemented in the current system, the concept envisions a fully autonomous, self-contained environment (as illustrated in Fig. 1). In such a setup, the risk of human harm would be greatly reduced. The system would operate in a closed-loop environment, and during major maintenance or upgrades, full power shutdown procedures would be employed to ensure safety. These considerations highlight the importance of incorporating ethical and technical safeguards as the system matures from a prototype to a deployable solution.

Social impact

When discussing the social implications of AI and robotics, a major concern is the potential increase in unemployment. If this issue is not addressed on time, it could lead to serious social problems, such as unemployed individuals resorting to violent movements or widespread unrest. To mitigate these risks, various solutions could be implemented. One widely discussed approach is the upskilling of production workers, which may provide some immediate relief. However, if technological advancements continue at their current pace, it may be difficult for humans to keep up with robots. Therefore, another way to maintain social stability could be to promote lifelong learning in areas such as art, philosophy, and pursuing individual hobbies as long as they do not harm others.

5 Conclusion

To conclude, this thesis has explored several key aspects at the intersection of generative AI and embedded robotic systems. The primary objective was to demonstrate that can extend beyond the virtual environment and apply meaningful control over physical systems. The second objective was to evaluate the current capabilities of AI in achieving this goal and identify the limitations that still exist.

Regarding the first objective, the thesis successfully demonstrated that AI can interact with and control physical systems. The proposed framework enabled a four-wheel robot to operate autonomously in response to user commands, with the CGA dynamically generating and executing the necessary

control code. This achievement confirms the feasibility of using Gen-AI-based tools to bridge the digital and physical environments. Another important conclusion drawn during the project was that leveraging a fine-tuned, task-specific LLM requires access to a large, high-quality dataset, something not easily achievable within the constraints of this project. As such, for smaller or time-sensitive tasks, it may be more practical to utilise publicly available models. Furthermore, assuming the rapid advancement of AI capabilities continues, it is plausible that CGAs may eventually replace the role currently held by local LLMs.

In relation to the second objective, the findings revealed that while significant progress has been made in AI-driven code generation and robotic control, the field is still lacking a robust solution for system self-identification, particularly the automatic detection and classification of peripheral hardware. This limitation highlights the point that AI, in its current state, has not yet fully achieved the capability to autonomously bridge the gap between digital and physical environments. Although the robot control part is ready for deployment, the absence of a reliable method for autonomous hardware recognition remains a critical barrier.

Future work should focus on addressing this challenge, particularly by developing a comprehensive solution for system self-identification. Achieving this would allow AI systems to operate with autonomy, fully closing the gap between perception, reasoning, and physical action.

To close it out, this thesis has successfully demonstrated the potential of Gen-AI, particularly through the integration of CGAs and LLMs, in the context of embedded systems and robotics. The results contribute valuable insights and provide a foundation for continued research and development in this emerging and impactful domain.

Acknowledgements

Gratitude is extended to the thesis supervisor, Benaoumeur Senouci, for his continuous support, guidance, and valuable feedback throughout the course of this project. Thanks should also be extended to the University of Southern Denmark for making this thesis possible.

References

[1] C. S. Dave Bergmann, "What is artificial general intelligence (agi)?" *IBM*, 2024. [Online]. Available: https://www.ibm.com/think/topics/artificial-general-intelligence?form=MG0AV3

[2] C. staff, "What is artificial general intelligence? definition and examples," *Coursera*, 2024. [Online]. Available: https://www.coursera.org/articles/what-is-artificial-general-intelligence

[3] "Aligning Cyber Space with Physical World: A Comprehensive Survey on Embodied AI," Sep. 2024. [Online]. Available: https://www.researchgate.net/publication/382111447_Aligning_Cyber_Space_with_Physical_World_A_Comprehensive_Survey_on_Embodied_AI

[4] M. S. Patil, G. Ung, and M. Nyberg, "Towards Specification-Driven LLM-Based Generation of Embedded Automotive Software," Nov. 2024, arXiv:2411.13269 [cs]. [Online]. Available: http://arxiv.org/abs/2411.13269

[5] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, J. Dabis, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, J. Hsu, J. Ibarz, B. Ichter, A. Irpan, T. Jackson, S. Jesmonth, N. J. Joshi, R. Julian, D. Kalashnikov, Y. Kuang, I. Leal, K.-H. Lee, S. Levine, Y. Lu, U. Malla, D. Manjunath, I. Mordatch, O. Nachum, C. Parada, J. Peralta, E. Perez, K. Pertsch, J. Quiambao, K. Rao, M. Ryoo, G. Salazar, P. Sanketi, K. Sayed, J. Singh, S. Sontakke, A. Stone, C. Tan, H. Tran, V. Vanhoucke, S. Vega, Q. Vuong, F. Xia, T. Xiao, P. Xu, S. Xu, T. Yu, and B. Zitkovich, "RT-1: Robotics Transformer for Real-World Control at Scale," Aug. 2023, arXiv:2212.06817 [cs]. [Online]. Available: http://arxiv.org/abs/2212.06817

[6] W. Chen, Z. Li, and M. Ma, "Octopus: On-device language model for function calling of software APIs," Apr. 2024, arXiv:2404.01549 [cs]. [Online]. Available: http://arxiv.org/abs/2404.01549

[7] Z. Englhardt, R. Li, D. Nissanka, Z. Zhang, G. Narayanswamy, J. Breda, X. Liu, S. Patel, and V. Iyer, "Exploring and Characterizing Large Language Models for Embedded System Development and Debugging," in *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '24. New York, NY, USA: Association for Computing Machinery, May 2024, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3613905.3650764

[8] NVIDIA, N. Agarwal, A. Ali, M. Bala, Y. Balaji, E. Barker, T. Cai, P. Chattopadhyay, Y. Chen, Y. Cui, Y. Ding, D. Dworakowski, J. Fan, M. Fenzi, F. Ferroni, S. Fidler, D. Fox, S. Ge, Y. Ge, J. Gu, S. Gururani, E. He, J. Huang, J. Huffman, P. Jannaty, J. Jin, S. W. Kim, G. Klár, G. Lam, S. Lan, L. Leal-Taixe, A. Li, Z. Li, C.-H. Lin, T.-Y. Lin, H. Ling, M.-Y. Liu, X. Liu, A. Luo, Q. Ma, H. Mao, K. Mo, A. Mousavian, S. Nah, S. Niverty, D. Page, D. Paschalidou, Z. Patel, L. Pavao, M. Ramezanali, F. Reda, X. Ren, V. R. N. Sabavat, E. Schmerling, S. Shi, B. Stefaniak, S. Tang, L. Tchapmi, P. Tredak, W.-C. Tseng, J. Varghese, H. Wang, H. Wang, H. Wang, T.-C. Wang, F. Wei, X. Wei, J. Z. Wu, J. Xu, W. Yang,

L. Yen-Chen, X. Zeng, Y. Zeng, J. Zhang, Q. Zhang, Y. Zhang, Q. Zhao, and A. Zolkowski, "Cosmos World Foundation Model Platform for Physical AI," Jan. 2025, arXiv:2501.03575 [cs]. [Online]. Available: http://arxiv.org/abs/2501.03575

[9] M. A. Haque, "LLMs: A Game-Changer for Software Engineers?" Nov. 2024, arXiv:2411.00932 [cs]. [Online]. Available: http://arxiv.org/abs/2411.00932

[10] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 3803–3810, arXiv:1710.06537 [cs]. [Online]. Available: http://arxiv.org/abs/1710.06537

[11] H. Zhang, A. Ning, R. Prabhakar, and D. Wentzlaff, "A Hardware Evaluation Framework for Large Language Model Inference," Dec. 2023, arXiv:2312.03134 [cs]. [Online]. Available: http://arxiv.org/abs/2312.03134

[12] L. Jiao, X. Song, C. You, X. Liu, L. Li, P. Chen, X. Tang, Z. Feng, F. Liu, Y. Guo, S. Yang, Y. Li, X. Zhang, W. Ma, S. Wang, J. Bai, and B. Hou, "AI meets physics: a comprehensive survey," *Artificial Intelligence Review*, vol. 57, no. 9, p. 256, Aug. 2024. [Online]. Available: https://doi.org/10.1007/s10462-024-10874-4

[13] Z. Zhang and J. Li, "A Review of Artificial Intelligence in Embedded Systems," *Micromachines*, vol. 14, no. 5, p. 897, May 2023, number: 5 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2072-666X/14/5/897

[14] S. Joel, J. J. Wu, and F. H. Fard, "A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages," Nov. 2024, arXiv:2410.03981 [cs]. [Online]. Available: http://arxiv.org/abs/2410.03981

[15] L. Xie, L. Peng, and J. Zhang, "Towards Robust RF Fingerprint Identification Using Spectral Regrowth and Carrier Frequency Offset," Dec. 2024, arXiv:2412.07269 [eess]. [Online]. Available: http://arxiv.org/abs/2412.07269

[16] M. F. A. Sayeedi, A. M. I. M. Osmani, T. Rahman, J. F. Deepti, R. Rahman, and S. Islam, "ElectroCom61: A multiclass dataset for detection of electronic components," *Data in Brief*, vol. 59, p. 111331, Apr. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352340925000630

[17] A. Joice, T. Tufaique, H. Tazeen, C. Igathinathane, Z. Zhang, C. Whippo, J. Hendrickson, and D. Archer, "Applications of Raspberry Pi for Precision Agriculture—A Systematic Review," *Agriculture*, vol. 15, no. 3, p. 227, Jan. 2025, number: 3 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2077-0472/15/3/227

[18] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," Nov. 2024, arXiv:2406.00515 [cs] version: 2. [Online]. Available: http://arxiv.org/abs/2406.00515

[19] Z. Li, I. Mondal, Y. Liang, H. Nghiem, and J. L. Boyd-Graber, "PANDA (Pedantic ANswer-correctness Determination and Adjudication):Improving Automatic Evaluation for Question Answering and Text Generation," Feb. 2024, arXiv:2402.11161 [cs] version: 1. [Online]. Available: http://arxiv.org/abs/2402.11161v1

Abbreviations

ADC	Analog-to-Digital Converter
AGI	Artificial General Intelligence
AI	Artificial Intelligence
API	Application Programming Interface
CAD	Computer Aided Design
CGA	Code Generation Agent
CoT	Chain of Thought
GUI	Graphical User Interface
JSON	JavaScript Object Notation
LLM	Large Language Model
LoRA	Low-Rank Adaptation
ML	Machine Learning
MLM	Multi-modal Large Model
MCTS	Monte Carlo Tree Search
OS	Operating System
PWM	Pulse-Width Modulation
RPi	Raspberry Pi
TTS	Text-to-Speech
TF-IDF	Term Frequency–Inverse Document Frequency
VLA	Vision-Language-Action
VS Code	Visual Studio Code
WFM	World Foundation Model

A Benchmark Test Dataset Structure

he dataset used for benchmarking consisted of robot definitions stored in JSON format. Below is an example entry:


```
{
  "robot": "Arduino Robotics Kit (Smart Car)",
  "components": {
    "DC Motors": [
      { "name": "Drive DC Motor", "count": 4 }
    ],
    "Microcontroller": [
      { "name": "Arduino Uno", "count": 1 }
    ]
  },
  "description": "A beginner-friendly smart car kit..."
}


  {
    "robot": "Arduino Robotics Kit (Smart Car)",
    "components": "- 4 x DC Drive Motors (e.g.,
    12V
    geared motors)\n  - Mounted with wheels for
    four-wheel drive or differential control\n\n-
    2 x L298N or TB6612FNG Dual H-Bridge Motor
    Drivers\n  - Allow bidirectional control of 4
    motors\n  - Connected to Arduino PWM and
    digital pins\n\n- 1 x Arduino Uno or Nano
    Microcontroller(ATmega328P)\n  - Programmed
    via USB with Arduino IDE\n\n- 2 x Ultrasonic
    Sensors (e.g., HC-SR04)\n - For obstacle
    detection (front-mounted)\n\n- 2 x IR Sensors
    (Line Tracking or Obstacle Avoidance)\n
    - Detect surface contrast or nearby objects\n\n-
    2 x Line Tracking Sensors\n  - Mounted under the
    chassis for line-following tasks\n\n- 1 x 7.4V or
    9V Battery Pack (Li-ion or AA holder)\n  -
    Powers motors and Arduino (with voltage regulation)
    \n\n- 1 x Robot Chassis Frame (plastic or metal
    base)\n - Includes wheels, caster, and motor
    brackets\n\n- Optional: Bluetooth Module
    (e.g., HC-05), Buzzer, LED\n - For wireless control
    or feedback features", "expected": "Based on the
    given components:\n \"4 DC motors, dual motor
    drivers, Arduino Uno, ultrasonic and IR sensors,
    and a mobile chassis\"\n\nThe most likely robot
    these components form is:\n    \"An Arduino smart
```

```
    car - a beginner-level programmable robot that can
    perform line following, obstacle avoidance, and
    basic navigation. It serves as an introduction to
    microcontrollers, motor control, and sensor
    integration.\"\n\nI can do the following tasks:\n
    \"- Follow a line using bottom-mounted IR sensors\n
    - Avoid obstacles using ultrasonic distance
    measurement\n    - Be remotely controlled via
    Bluetooth or IR remote\n    - Emit sounds or
    lights for interactive feedback\n
    - Serve as an educational platform for
    learning Arduino coding and robotics principles\""
  },
```

Model List

- gemma3:1b
- mistral
- mistral7b_fine_tuned
- deepseek-r1:1.5b
- phi
- ML_model_3.py

B LLM system recipe generation example

```
System Specification:
{
  "Based on the given components": [
    "Raspberry Pi",
    "four ruber wheels",
    "L298N Motor Controller back",
    "12V to 9V battery",
    "GND",
    "5V DC power",
    "ENA_back_right_wheel: GPIO 6",
    "IN1: GPIO 7",
    "IN2: GPIO 8",
    "IN3: GPIO 11",
    "IN4: GPIO 9",
    "ENB_back_left_wheel: GPIO 10",
    "L298N Motor Controller_front",
    "12V to 9V battery",
    "GND",
    "5V DC power",
```

```
    "ENA_front_left_wheel: GPIO 12",
    "IN1: GPIO 23",
    "IN2: GPIO 24",
    "IN3: GPIO 25",
    "IN4: GPIO 16",
    "ENB_front_right_wheel: GPIO 13",
    "HC SR04 Ultrasonic Sensor 3 (back)",
    "VCC: 5V DC power",
    "Trig: GPIO 27",
    "Echo: GPIO 17",
    "GND",
    "HC SR04 Ultrasonic Sensor 2 (front right)",
    "VCC: 5V DC power",
    "Trig: GPIO 26",
    "Echo: GPIO 21",
    "GND",
    "HC SR04 Ultrasonic Sensor 1 (front left)",
    "VCC: 5V DC power",
    "Trig: GPIO 4",
    "Echo: GPIO 22",
    "GND",
    "Micro Servo Motor 9G SG90",
    "GND",
    "5V",
    "PWM Control: GPIO 5",
    "USB microphone",
    "USB port",
    "USB Web Cam",
    "USB speaker",
    "Headphones jack",



    "2x 9V batteries",
    "4x DC Gear Motors",
    "Front Right: L298N 1 ENB",
    "Front Left: L298N 1 ENA",
    "Back Right: L298N 2 ENA",
    "Back Left: L298N 2 ENB"
  ]
}
more details about components connections:
- Raspberry Pi 4 Model B
 -four ruber wheels  - L298N Motor Controller back
    - 12V to 9V battery
    - GND
    - 5V DC power
```

```
    - ENA_back_right_wheel: GPIO 6
    - IN1: GPIO 7
    - IN2: GPIO 8
    - IN3: GPIO 11
    - IN4: GPIO 9
    - ENB_back_left_wheel: GPIO 10
  - L298N Motor Controller_front
    - 12V to 9V battery
    - GND
    - 5V DC power
    - ENA_front_left_wheel: GPIO 12
    - IN1: GPIO 23
    - IN2: GPIO 24
    - IN3: GPIO 25
    - IN4: GPIO 16
    - ENB_front_right_wheel: GPIO 13
  - HC SR04 Ultrasonic Sensor 3 (back)
    - VCC: 5V DC power
    - Trig: GPIO 27
    - Echo: GPIO 17
    - GND
  - HC SR04 Ultrasonic Sensor 2 (front right)
    - VCC: 5V DC power
    - Trig: GPIO 26
    - Echo: GPIO 21
    - GND
  - HC SR04 Ultrasonic Sensor 1 (front left)
    - VCC: 5V DC power
    - Trig: GPIO 4
    - Echo: GPIO 22
    - GND
  - Micro Servo Motor 9G SG90
    - GND
    - 5V
    - PWM Control: GPIO 5
  - USB microphone
- USB port  - USB Web Cam
    - USB port
  - USB speaker
    - Headphones jack
  - 2x 9V batteries
  - 4x DC Gear Motors
    - Front Right: L298N 1 ENB
    - Front Left: L298N 1 ENA
    - Back Right: L298N 2 ENA
    - Back Left: L298N 2 ENB
```

```
  - 4x Wheels


Instruction to Agent:
Write a smal testing code to operate the system described
above using appropriate hardware logic, sensor
integration,dependency installation file, task automation
and run the code in the terminal using python3. Once you
finish the task report back to this file
agent_feedback.txt(Task completed.Ready for the next
command.).

# Model used: phi
```

C Automating LLM_text_extract.py at Boot

C.1 Prerequisites

1) Project directory structure

   ```
   /home/GenAI/
    LLM_text_extract.py
    (.venv)/
   ```

2) Python ≥3.11 installed inside `myenv`.

3) Auto launch extension installed

4) `sudo` privileges on the Raspberry Pi.

C.2 Create the task Script

```
nano /home/GenAI/.vscode/tasks.json
```

File: `/home/GenAI/.vscode/tasks.json`

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "auto-run-robot",
      "type": "shell",
      "command": "bash",
      "args": [
        "-c",
        "sleep 50 && python3 LLM_text_extract.py"
      ],
```

```
      "options": {
        "cwd": "/home/GenAI"
      },
      "problemMatcher": [],
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "runInTerminal": true,
      "auto": true
    }
  ]
}
```

C.3 Set autostart:

```
nano ~/.config/lxsession/LXDE-pi/autostart
```

Add

```
@code --folder-uri file:///home/GenAI
```

D Main code (LLM_text_extract.py)

```
import json
import time
import ollama
import pyautogui
import pyperclip
import pyttsx3
import sounddevice as sd
import numpy as np
import speech_recognition as sr
import os
import subprocess
import tempfile
from piper import PiperVoice



# Wait to give VSCode time to open
print("Waiting for VSCode to load...")
time.sleep(1)

voice_path = os.path.expanduser("~/.local/share/piper/voices/en_US/en_US-amy-medium.onnx")
config_path = voice_path + ".json"
voice = PiperVoice.load(voice_path, config_path)

# === Step 1a: Speak function ===

def speak(text):
    try:
        with tempfile.NamedTemporaryFile(delete=False, suffix=".wav") as tmp_wav:
            result = subprocess.run(
                [
                    "piper",
                    "--model", voice_path,
                    "--output_file", tmp_wav.name
                ],
                input=text.encode(),
                stdout=subprocess.PIPE,
                stderr=subprocess.PIPE
            )
            if result.returncode != 0:
                print("Piper CLI error:", result.stderr.decode())
                return
            subprocess.run(["aplay", tmp_wav.name])
            os.remove(tmp_wav.name)
    except Exception as e:
        print("Piper system error:", e)
```

```
# === Step 1b Locate roo extension on screen ===

print("Locating roo extension on screen...")
location_roo = pyautogui.locateOnScreen("roo_extension.png", confidence=0.8)

if location_roo:
    pyautogui.click(pyautogui.center(location_roo))
    time.sleep(5)
    speak("Roo Code has been accessed")

else:
    print("Could not find roo extension. Make sure 'roo_extension.png' matches what's on screen.")

    exit()

# === Step 1c: Run the model ===

with open("benchmark_data_set.json", "r") as f:
    TEST_DATA = json.load(f)

client = ollama.Client()
model_name = "phi"

system_message = """
You are an automation system prediction assistant. Your task is to analyze the provided components
and determine the most likely robot (or system) they form.

Your output must be a JSON object with the following keys:

- "Based on the given components": A list of components exactly as entered by the user.
- "The most likely robot these components form is": A detailed 50 to 100-word description of the
predicted robot,explaining how the components work together.
- "I can do the following tasks": A list of specific tasks that the described robot is capable of
performing.
Ensure your response is a valid JSON object, without any additional explanations or reasoning beyond
the required format.
"""

test_input = TEST_DATA[22]

feedback_file = "agent_feedback.txt"

response = client.generate(
    model=model_name,
    prompt=test_input["components"],
    system=system_message,
    options={"temperature": 0.1},
    format="json"
)
```

```
output = json.loads(response.response.strip())

# === Step 2: Format final message ===
final_prompt = f"""System Specification:
{json.dumps(output, indent=2)}
more details about components connections:
{test_input["components"]}

Instruction to Agent:
Write a small testing code to operate the system described above using appropriate hardware logic,
sensor integration, dependency installation file, task automation and run the code in the terminal
using python3.
Once you finish the task report back to this file agent_feedback.txt(Task completed. Ready for the
next command.).

# Model used: {model_name}

"""
pyperclip.copy(final_prompt)

# === Step 3: Locate chat input on screen ===

print("Locating Roo Code chat input on screen...")
location = pyautogui.locateOnScreen("chat_input_2.png", confidence=0.8)

if location:
    pyautogui.click(pyautogui.center(location))
    time.sleep(1)
    pyautogui.hotkey('ctrl', 'v')
    pyautogui.press('enter')
    print("Message sent to Roo Code.")
else:
    print("Could not find chat input. Make sure 'chat_input.png' matches what's on screen.")
    exit()

# === Step 5.a: Listen for user input ===

def listen_for_task(duration=10, fs=16000):
    recognizer = sr.Recognizer()
    print("Recording from mic using sounddevice...")
    audio_data = sd.rec(int(duration * fs), samplerate=fs, channels=1, dtype='int16')
    sd.wait()
    audio = sr.AudioData(audio_data.tobytes(), fs, 2)
```

```
    try:
        command = recognizer.recognize_google(audio)
        print("User said:", command)
        return command
    except sr.UnknownValueError:
        print("Could not understand audio.")
        return None
    except sr.RequestError:
        print("Speech recognition service error.")
        return None

# === Step 5.b: Wait for agent feedback ===
def wait_for_agent_feedback(timeout=900, poll_interval=5):
    print("Waiting for code generation agent to report completion...")
    start_time = time.time()

    while time.time() - start_time < timeout:
        if os.path.exists(feedback_file):
            with open(feedback_file, "r") as f:
                content = f.read().strip().lower()
                if "task completed" in content:
                    print("Code generation agent reported task completion.")
                    return True
        time.sleep(poll_interval)



    print("Timeout reached. No feedback received from code generation agent.")
    return False

def clear_agent_feedback():
    with open(feedback_file, "w") as f:
        f.write("")

# === MAIN LOOP ===
while True:

    # Step 1a: Wait for recepy
    if final_prompt != "":
        print("Message is beein sent to Roo Code.")
        speak("Message is being sent to Roo Code")
        time.sleep(5)
```

```
    # Step 1b: Wait for agent's feedback
    if wait_for_agent_feedback():
        # Step 2: Prompt user for vocal input
        speak("Task completed. I'm ready for your command. What would you like the robot to do
        next?")
        user_command = listen_for_task()

        if user_command:
            if user_command.lower() in ["exit", "stop", "quit"]:
                speak("Okay, exiting the session.")
                break

            # Step 3: Send task to code generation agent
            followup_instruction = f"""

Follow-up Instruction:
The user wants the robot to perform the following task: "{user_command}".
Please modify the previously generated code to fulfill this new user requirement.
Once you finished the task report back to this file agent_feedback.txt(Task completed.
Ready for the next command.).
"""
            pyperclip.copy(followup_instruction)
            if location:
                pyautogui.click(pyautogui.center(location))
                time.sleep(0.5)
                pyautogui.hotkey('ctrl', 'v')
                pyautogui.press('enter')
                print("Follow-up command sent.")
                speak("Command sent to the code generation agent. I'll notify you when the
                task is complete.")
            else:
                print("Could not re-locate chat input.")
                break

            # Step 4: Reset agent_feedback.txt
            clear_agent_feedback()
        else:
            speak("I didn't catch that. Please try again next time.")
            clear_agent_feedback()
    else:
        speak("Timeout reached. No response from code generation agent.")
        break
```