School of Computer Science and Electronic Engineering

MSc Data Science

Academic Year 2023-2025

Meta-Interpretive Learning for Recursive Skill Acquisition in Robotic Manipulation

A project report submitted by: Aqib Hafiz (6830867)

A project supervised by: Dr. Alireza Tamaddoni Nezhad

A report submitted in partial fulfilment of the requirement for the degree of Master of Science

University of Surrey
School of Computer Science and Electronic Engineering
Guildford, Surrey GU2 7XH
United Kingdom.
Tel: +44 (0)1483 300800

ABSTRACT

Robotic Learning from Demonstration (LfD) is a way for robots to learn skills from examples given by people instead of having to be programmed by hand. This shows promise for flexible, data-efficient automation, but most modern LfD methods use deep learning, which usually needs huge datasets and makes it hard to understand how decisions are made. This dissertation investigates the potential of Meta-Interpretive Learning (MIL), a variant of Inductive Logic Programming (ILP), to provide a more transparent and data-efficient solution for robotic manipulation tasks.

The study was executed in two distinct phases. The first step was to use a Pepper humanoid robot, which was controlled by a Pico VR headset and SteamVR bridge, to look into immersive teleoperation. Even though Python and the NAOqi API worked well together, Pepper wasn't good for regular demonstrations because of high latency (about 500ms) and mechanical limits. The project then switched to a custom-built pipeline that used the RoArm-M3 robotic arm. This included manually recording take/put primitives in 3D space, playing them back in Python, and using the Metagol system to learn symbolic rules.

Metagol was able to create a recursive robot/1 predicate that could generalise to longer, unseen sequences with just one demonstration. The whole system, from demonstration to execution, stayed light, easy to repeat, and easy to understand. These results demonstrate a proof-of-concept for explainable, one-shot learning in robotics utilising symbolic logic instead of statistical approximation.

Keywords: Meta-Interpretive Learning, Inductive Logic Programming, Explainable Robotics, Learning from Demonstration, RoArm-M3, Pepper

HIGHLIGHTS

- One-shot recursive robotic learning demonstrated with Meta-Interpretive Learning.
- VR teleoperation pipeline created to control Pepper robot using Pico VR headset.
- Custom JSON recorder and Python bridge developed for RoArm-M3 manipulator.
- Recursive rules induced by Metagol executed successfully on physical robot hardware.
- Transparent, explainable, and data-efficient alternative to deep learning LfD approaches.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Professor Alireza Tamaddoni-Nezhad, whose constant support, guidance, and encouragement have been invaluable to me. His constant ideas on how to improve the work with his constructive feedback helped me shape my dissertation.

I am also sincerely grateful to Daniel Cyrus, who was always available with his assistance. His constant support with his deep knowledge helped me to be more productive. His debugging skills, deep knowledge in mathematics and in ILP was a great source of knowledge throughout the Dissertation.

Special thanks to Samuel J Carter, for providing access to RoArm-M3, which was a crucial part of this research, Without the physical access to the arm the demonstration and validation of my approach would not have been possible.

Sign in the box below to certify that the work carried out is your own. By signing this box you are certifying that your dissertation is free from plagiarism. Make sure that you are fully aware of the Department guidelines on plagiarism (see the student handbook). The penalties if you are caught are severe. All material from other sources must be properly referenced and direct quotes must appear in quotation marks.

I certify that the work presented in the dissertation is my own unless referenced

Signature 

Date 2/9/2025

Insert a word count. This is the sum of the words in all the chapters only. The sum should exclude the words in the title page, abstract, acknowledgements, table of contents, references and any appendices.

TOTAL NUMBER OF WORDS: 16165

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Motivation

A key skill which autonomous systems need now which can make them change how they act is Robotic Learning, methods which are conventionally used depend on pre-defined regulations or data intensive deep learning techniques, where the necessity of large datasets increases to capture the heterogeneity of real-world scenarios [1][2]. Even while these approaches can work well, they often don't work when they have to deal with new jobs or changes in the environment, and their decision-making process are usually not clear [5].

To tackle these issues, researchers have progressively adopted explainable and data-efficient learning methodologies. Meta-Interpretive Learning (MIL) is a promising paradigm [10][11]. It is a type of Inductive Logic Programming (ILP) that lets robots learn symbolic, interpretable rules from just a few examples. By utilising prior knowledge, MIL enables robots to generalise efficiently while ensuring the transparency and auditability of acquired actions.

This dissertation investigates the utilisation of MIL in robotic manipulation, focussing on the recursive task of box manipulation. The investigation started with a teleoperation system that let people operate a Pepper humanoid robot in real time using a Pico VR headset [23]. This confirmed that immersive demonstration capture worked, but Pepper's physical restrictions and latency made it less useful for formal ILP learning [23]. The project subsequently switched to a RoArm-M3 manipulator [24], while MIL was used to create recursive task rules from a single demonstration and run them on hardware.

This project began as an attempt to make robots capable of acquiring explainable, recursive actions from limited input while ensuring auditability and adaptability to novel settings.

1.2 Goals

Based on the motivation discussed above, the dissertation was guided by the following specific goals:

- Create a teleoperation interface that connects the Pico VR headset and controllers to the Pepper robot using Virtual Desktop, SteamVR/OpenXR, and a Python client-server bridge such that immersive demonstration capture is possible.
- Find out what Pepper can't do for ILP-based learning and explain why you switched to RoArm-M3.
- Implement a recording-execution pipeline for RoArm-M3 that saves take and put actions in JSON format and plays them back for hardware execution.

- Use Metagol to connect MIL with one-shot examples and create recursive task rules. Then, use Python execution instructions to connect symbolic predicates.

- Test the induced rules by running recursive manipulation jobs on the RoArm-M3 and checking the success rate, execution time, and ability to apply them to longer sequences.
- Critically evaluate the contributions and constraints and suggest avenues for future research.

These objectives together aim to demonstrate that one-shot/few-shot recursive learning can be achieved by implementing ILP specifically MIL in a real robotic setup.

1.3 History

This project uses two strands of prior work: Learning from Demonstration (LfD) and Inductive Logic Programming (ILP), specifically Meta-Interpretive Learning (MIL). LfD explains how demonstrations become policies, ILP/MIL explains how symbolic rules generalise from few examples. Section 2.2 summarises LfD's strengths and gaps, Sections 2.3–2.4 cover ILP/MIL and explainability. Here we only signpost the platforms (Pepper and RoArm-M3) and software stack used detailed background is in Chapter 2.

1.4 Statement of the Problem

The core challenges which this dissertation set out to address was facilitating robots in learning recursive manipulation tasks in a manner that is both data-efficient and comprehensible. Current deep learning methods depend on extensive datasets and frequently lack transparency, rendering them inappropriate for safety-critical or data-limited applications [2][5][6].

Meta-Interpretive Learning (MIL), on the other hand, can cause recurrent behaviours like box manipulation from just one example while still being easy to understand [10][11]. This makes it well-suited for robotic learning tasks that can be explained.

The project commenced with an examination of VR teleoperation utilising the Pepper humanoid robot and a Pico VR headset. This confirmed that immersive human-robot contact was possible, but Pepper's hardware limitations and lag made it very difficult if not practically impossible to create the structured demonstration that ILP needed. Because of this, the switch was made to RoArm-M3 manipulator, which coupled symbolic induction with Metagol and a unique recording-execution pipeline.

The initial concept involved utilising Pepper VR teleoperation to gather ILP training data. However, latency and technological issues hindered the creation of structured, repeatable demonstrations. The project subsequently transitioned to the RoArm-M3 manipulator during the learning phase.

The primary issue this dissertation examines is the applicability of MIL in generating recursive rules for manipulation tasks and their execution on actual robotic hardware. Solving this challenge leads to a proof-of-concept pipeline for explainable, one-shot recursive robotic learning.

1.5 An Overview of the Dissertation

This structure is chosen on the basis of logical progression from motivation to implementation and critical reflection as well as future recommendations if any:

Chapter 2 - Literature Review: This chapter looks at other work in LfD, ILP, MIL, robotics platforms, and AI that can be explained.

Chapter 3 - Research Approach: Talks about the research design, which include VR teleoperation, the switch to RoArm-M3, and the reasons for using MIL.

Chapter 4 - Data Analysis: This Chapter talks about the teleoperation system, the JSON- based recorder, and the ILP-Python Connection.

Chapter 5 - Discussion: Shows the outcomes of the evaluation, screenshots, and a critical review of the literature.

Chapter 6 - Conclusion: Summarises contributions, considers limitations, and suggests extensions.

1.6 Contributions and the possibility of publication

This project makes several contributions the study of ILP in robotics:

- Creating a VR-based teleoperation pipeline that connects Pico VR controllers to Pepper.
- Setting up a recording-execution pipeline for RoArm-M3
- Showing how Metagol can do one-shot recursive learning and apply it to tasks that go beyond the demonstration [10][12].
- Combining symbolic reasoning with the actual execution of hardware.

These results while preliminary, show that MIL could become a very powerful tool for interpretable and data efficient robotics learning, further investigation into this could lead to papers in the areas of symbolic AI, robotics, and hybrid neuro-symbolic systems.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

Day by day robots are required to handle complex tasks in home, factories and other areas like healthcare, yet they require large number of demonstrations to learn from, the ability to generalise from a small set of examples is still elusive [1][2]. Facilitating robots to attain manipulation skills that are both data-efficient and comprehensible continues to be a significant research challenge. This literature review analyses essential technologies and methods pertinent to symbolic learning, recursive skill acquisition, and robotic manipulation. An essential capability for autonomous systems is the ability to generalise beyond training data and apply knowledge to novel

contexts. Accomplishing this transparently and in an auditable manner is especially challenging, given contemporary machine learning techniques depend on extensive datasets and generate obscure decision-making policies [5]. Errors or misgeneralizations in such systems can undermine their reliability in safety-critical sectors.

Traditional methods of robot programming depend on manually crafted control logic or heuristic algorithms for navigation and manipulation. Although efficient in limited settings, some strategies exhibit a deficiency in adaptation. Conversely, contemporary learning-based approaches, including deep learning and reinforcement learning, utilise extensive datasets or experiential knowledge to enhance policy. Deep neural networks can directly correlate sensory inputs with actions, while reinforcement learning can enhance control tactics via trial-and-error. Nevertheless, these methodologies frequently lack generalisability to novel challenges and offer minimal understanding of their internal reasoning processes.

To rectify these deficiencies, symbolic learning techniques, including Inductive Logic Programming (ILP) and particularly Meta-Interpretive Learning (MIL), have been formulated. These methodologies leverage structured background Knowledge to derive rules that are recursive, generalisable, and interpretable. This renders them especially advantageous for robotic applications that necessitate learning from limited demonstrations and demand comprehensible explanations for humans.

The literature review covers the technical background most relevant to this dissertation across several domains:

- Learning from Demonstration (LfD): techniques for teaching robots via human-provided examples, often implemented using deep learning and reinforcement learning for perception and control.
- Inductive Logic Programming and Meta-Interpretive Learning: symbolic frameworks enabling explainable, recursive, one-shot learning.
- Explainable AI (XAI): approaches to improve transparency in decision-making.
- VR teleoperation: Immersive methods for demonstration capture, with focus on Pepper humanoid robot.
- Robotic platforms: Pepper as an HRI-focused platform and RoArm-M3 as a lightweight manipulator suited for ILP.

The analysis of current advancements identifies ongoing issues related to sample efficiency, generalization, simulation-to-reality transfer, and explainable validation. This dissertation seeks to advance these domains by presenting a proof-of-concept pipeline for one-shot recursive robotic learning that incorporates symbolic induction (Metagol), VR-based teleoperation for demonstrations, and hardware execution on the RoArm-M3.

This review aims to build a comprehensive foundation across various areas, setting the background for the methodology, implementation, and assessment of the proposed system discussed in the following chapters.

2.2 Learning from Demonstration (LfD)

This section contains the in-depth literature review of Learning from Demonstrations (LfD).

2.2.1 Definition and Overview

Learning from Demonstration (LfD) is a robotics example where robots can gain new skills through the observation and imitation of human behaviour rather than through explicit programming [1]. It provides an intuitive method for transferring information from humans to robots, hence reducing the obstacles for non-experts to instruct robots in task. A standard Learning from Demonstration (LfD) pipeline entails acquiring demonstrations via kinaesthetic teaching, teleoperation, or visual sensors, followed by the translation of those observations into implementable robotic rules,

The allure of Learning from Demonstration (LfD) is in its capacity to streamline programming and enable robots to generalise across various activities. This methodology has been utilised across various domains, including manipulation, navigation, and assembly tasks [2], rendering it one of the most extensively researched methodologies in robot learning.

2.2.2 Neural Methodologies

Contemporary Learning from Demonstration systems are often executed via deep learning or reinforcement learning methodologies. Deep neural networks can directly correlate high- dimensional sensory inputs (e.g., pictures, joint states) with robotic operations. These approaches have attained considerable success in tasks including grasping, pick-and-place, and autonomous navigation [3]. Reinforcement learning enhances this by improving behaviours through trial-and-error, enabling robots to optimise policies based on incentives and penalties [4].

Recent studies indicate that deep imitation learning and reinforcement learning can generate adaptable, generalisable behaviours, especially in simulation [3][2]. Nonetheless, their dependence on extensive datasets and substantial processing resources presents practical difficulties when implemented on physical robots.

2.2.3 Challenges and Limitations

Notwithstanding its achievements, LfD has numerous significant limitations:

- Neural methodologies necessitate extensive demonstrations to account for variability. In robotics, the acquisition of such datasets is both expensive and labour-intensive [6].
- Black-box policies: Acquired models are generally inscrutable, providing minimal interpretability or accountability in decision-making [7]. This renders them inappropriate for safety-critical environments.
- Inadequate recursion and compositionality: Learning from Demonstration

(LfD) can often struggle when tasks require recursion. Such as stacking or sorting, where same logic must be consistently applied. Neural models frequently struggle to generalise beyond the training scope [11].

- Sim-to-real transfer: Numerous accomplishments in Learning from Demonstration (LfD) are accomplished within a simulated environment [6]. The transfer of acquired policies to physical robot is hindered by discrepancies in perception and dynamics.

These issues underscore a disparity between the potential of Learning from Demonstration (LfD) and its actual implementation, particularly when data efficiency, interpretability, and recursion are essential. For example, teaching a robot to fold laundry may require repeating similar steps in a loop, something deep learning based LfD often fails to generalize.

2.3 Inductive Logic Programming (ILP) and Meta-Interpretive Learning (MIL)

This section contains the Literature review of Inductive Logic Programming and Meta Interpretive Learning.

2.3.1 Principles of Inductive Logic Programming

Inductive Logic Programming (ILP) is a symbolic machine learning approach that derives logic programs from examples and contextual knowledge [7]. In a sense, ILP behaves like a detective, piecing together clues (examples) and known facts (background knowledge) to infer a consistent story (hypothesis). In contrast to statistical learning, which optimises numerical parameters, Inductive Logic Programming (ILP) generated human-readable hypotheses articulated in first-order logic. This renders it an ideal contender for sectors where transparency, logic, and auditability are paramount.
A logic program is generally expressed in Horn clause form, comprising facts and rules. For Instance:

```
parent(alice, bob).
parent(bob, carol).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

In this context, parent/2 is an established relation (fact), while ancestor/2 is derived as a recursive rule using the provided examples and background knowledge. This illustrative example demonstrates ILP's ability to generalise beyond observations to create reusable, interpretable programs.

The ILP procedure typically encompasses:

- A collection of positive examples (E^+), negative examples (E^-), and background knowledge (BK).
- Hypothesis Exploration: Investigating a domain of potential logical programs

H.

- Output: A hypothesis H such that BK U H entails all E^+ and excludes all E^-.

In a formal manner:

$$\forall e \in E^+, BK \cup H \models e$$
$$\forall e \in E^-, BK \cup H \not\models e$$

Early ILP systems, such as FOIL (First Order Inductive Learner) and Progol, employed heuristic-guided search inside this hypothesis space [8][9]. Although impactful, they were constrained in scalability and in managing recursion or intricate predicate creation.

2.3.2 Meta-Interpretive Learning (MIL)

Meta-Interpretive Learning (MIL) enhances Inductive Logic Programming (ILP) by using meta-rules higher-order template that guides the hypothesis search [10]. A meta-rule specifies how predicates can be composed to form new clauses, so substantially restricting and directing the learning process.

One of the most widely used and powerful meta-rules in the chain metarule, typically expressed as:

```
P(A,B) :- Q(A,C), R(C,B).
```

This structure is called a chain rule because it shows how predicates are combined so that the output of Q becomes the input of R. By chaining together steps in the middle, MIL can learn relational programs.

This rule states that the relation P(A,B) holds if there exists an intermediate C such that Q(A,C) and R(C,B) both hold. In other words, we can say that P is learned as a composition or chain of Q and R [11].

The chain metarule is very important because it enables the MIL to induce recursive or compositional programs. For example, the classic ancestor/2 relation can be derived from parent/2 using this template:

```
ancestor(A,B) :- parent(A,C), ancestor(C,B).
```

Here, ancestor/2 is defined recursively by chaining a base relation (parent/2) with itself. In the context of this work, the same principle allows Metagol to induce a recursive rule for box manipulation, such as repeatedly take/1 and put/1 until the list of objects is empty.

Metagol, one of the most commonly used MIL implementations [12], performs meta-interpretive search guided by such meta-rules to construct hypotheses from very few examples. This process not only reduces the hypothesis space but also supports predicate invention and recursion, making it well-suited for tasks that require generalization beyond the training examples.

The MIL learning cycle can be encapsulated as follows:

- Supply foundational information (e.g., movement primitives).
- Present affirmative and detrimental instances of tasks.
- Utilise meta-rules recursively to formulate hypotheses.
- Authenticate hypotheses through empirical instances.

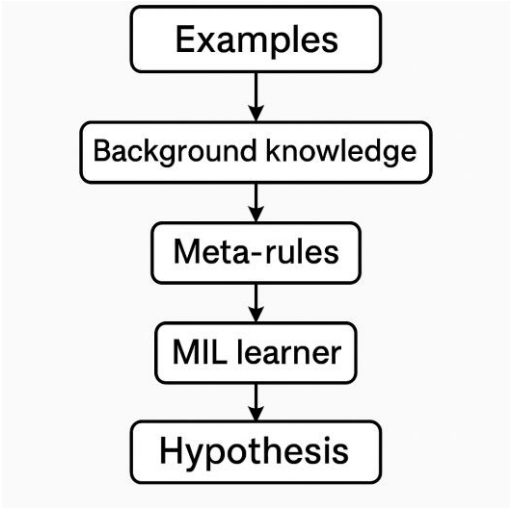

Fig 2.1: The MIL learning cycle.

2.3.3 Applications in Robotics and Artificial Intelligence

MIL has demonstrated potential in areas necessitating logic and interpretability:

Robotics: MIL has been employed for manipulation planning, action modelling, and recursive manipulation problems. Youssef (2023) examines how ILP frameworks facilitate symbolic task learning for robotic arms, encompassing manipulation planning and skill generalization [13].

Cropper (2022) emphasises MIL's efficacy in relational planning and program synthesis, whereby recursive task structures are essential [11].

Wang (2024) illustrates the promise of MIL for explainable AI by generating interpretable symbolic rules that enhance opaque neural policies. [14].

Hybrid neuro-symbolic systems: Tsuji et al. (2025) see increasing endeavours to integrate brain sensing with symbolic thinking to mitigate sim-to-real discrepancies in robotics [15].

This dissertation added Metagol to the RoArm-M3 manipulator for box-manipulation tasks. Recursive rules were derived from a single demonstration to generalise the problem to stacks of arbitrary length, so verifying MIL's one-shot recursive learning capabilities.

2.3.4 Advantages and Disadvantages

MIL presents distinct advantages compared to traditional ILP and neural Learning from Demonstration methodologies:

Advantages:

- Data efficiency: Derives rules from one or a limited number of demonstrations [10][11].
- Interpretability: Acquired programs are symbolic and comprehensible to humans [10][11].
- Recursive generalization: Effectively facilitates activities necessitating repetitive structures (e.g., stacking, sorting) [10][11].
- Integration with domain expertise: Permits antecedent knowledge to restrict and direct learning [7][10][11].

Constraints:

- Scalability: The computational expense increases with the magnitude of the problem and the complexity of the hypothesis [10][11].
- Reliance on symbolic input necessitates organised representations, unprocessed sensor data must undergo preprocessing [7][11].
- Restricted implementation in robotics: In contrast to deep learning, MIL is not as prevalent in conventional robotics frameworks, however hybrid methodologies are beginning to surface [13][14][15].

2.4 Explainable Artificial Intelligence (XAI) in Robotics

This section contains the Literature review of Explainable Artificial Intelligence (XAI).

2.4.1 Principles of Explainable Artificial Intelligence

Explainable Artificial Intelligence (XAI) is a building discipline focused on rendering the decision-making processes of AI systems accessible, interpretable, and auditable for human users [5]. In contrast to traditional black-box machine learning models that provide forecasts without elucidating their rationale, XAI prioritises the necessity for comprehensible explanations for humans. These elucidations can manifest in various formats, ranging from visual attention maps in deep networks to logical rules in symbolic systems.

The significance of XAI is particularly evident in robotics, as AI judgements manifest as physical behaviours that directly impact human safety. In healthcare, a robotic assistant responsible for drug distribution must not only execute its tasks accurately but also articulate the rationale behind its chosen navigation route [5]. In manufacturing and autonomous cars, robots must adhere to stringent safety requirements, necessitating explanations for certification and liability evaluations.

Research initiatives like DARPA's XAI program and the European Union's AI Act underscore the growing international emphasis on explainability [19][20]. DARPA's initiative prioritises reliable human-AI collaboration, whereas the AI Act stipulates a right to explanation for AI-facilitated decision-making. These changes highlight a transition from evaluating solely performance to a combined assessment of performance and interpretability.

2.4.2 Neural vs Symbolic Approaches

Contemporary robotic systems rely on deep neural networks (DNNs) for perception and control. DNNs, while their efficacy in identifying intricate patterns, face criticism for their lack of transparency. A robot utilising a convolutional neural network (CNN) may successfully evade an obstacle, although the rationale behind this decision frequently eludes human understanding.

A variety of neural XAI algorithms have been devised to tackle this challenge:

- Saliency Maps and Grad-CAM: emphasise the regions of an image that most significantly impacted a decision [18].
- LIME (Local Interpretable Model-agnostic Explanations) approximates neural predictions using more interpretable models [16].
- SHAP (SHapley Additive exPlanations): assesses the impact of each feature on a model's decision-making process [17].

However, while these methods provide valuable post hoc insights, they do not guarantee that the model's reasoning is inherently sound [5]. They are estimations, not certainties, which constrains their application in safety-critical robotics.

The constraint which above models have is not applicable when we take Inductive Logic Programming (ILP) and Meta-Interpretive Learning (MIL) into account.AI techniques like ILP and MIL provide explanations as a secondary outcome of the learning process. MIL, for instance, generates hypotheses as logical rules, which are intrinsically interpretable and verifiable [10][11]. In a box manipulation task, MIL may generate a recursive program:

```
manipulate(Box) :- take(Box), Put(Box), manipulate(Rest).
```

The rationale is clear: select an object, position it, and continue until all are addressed. In contrast to brain models, these explanations are inherent to the model's logic rather than appended post hoc [10].

Recent research investigates neuro-symbolic systems, integrating the advantages of both paradigms. Neural networks manage low-level perceptual tasks, such as object recognition and pose estimation, whereas symbolic reasoning addresses high-level planning and explanation. This hybridisation offers both precision and comprehensibility, however practical integration obstacles persist.

2.4.3 Utilisation in Robotics

XAI has been utilised in robotics across various sectors, demonstrating its extensive influence:

Human-Robot Interaction (HRI): Robots that can elucidate their judgements enhance trust and cooperation. Research indicates that humans favour robots capable of

articulating their activities through natural language or symbolic conventions, despite comparable performance levels [21].

Autonomous Navigation: Transparent reasoning frameworks are employed to enhance safety in autonomous driving. For instance, unambiguous regulations for lane changing or obstacle evasion may be evaluated against safety criteria, in contrast to opaque neural policies.

Manipulation and Task Learning: MIL has been utilised in box manipulation tasks, where recursive rules involving take/1 and put/1 are derived from a single demonstration. This facilitates quick and interpretable one-shot learning [10][11].

Debugging and Error Analysis: XAI technologies facilitate developers in determining the reasons behind a robot's task failure. If a rule fails to generalise, the error can be attributed to a lack of prior knowledge, an aspect of openness that black-box policies cannot provide.

Safety and Certification: Regulatory bodies in industrial robotics are progressively mandating verifiable rationale prior to the certification of robots for deployment. Symbolic XAI frameworks facilitate inherent compliance routes.

2.4.4 Advantages and Disadvantages

The advantages and disadvantages of various methodologies for elucidating robotics can be stated as follows:

Methodology	Advantages	Disadvantages
Neural XAI	It operates on unstructured data such as pictures and audio and is widely used in practice.	Its explanations are often retrospective, imprecise, and frequently unreliable [5].
Symbolic (ILP/MIL)	It generates comprehensible, recursive rules, is efficient in sampling, and remains inherently interpretable.	It demands structured input and faces scalability challenges when dealing with more intricate tasks [11].
Neuro-Symbolic	It integrates the perceptual capabilities of deep neural networks with symbolic interpretability and presents strong potential as a hybrid approach.	It faces integration difficulties, lacks extensive benchmarks, and is not yet standardised [15].

Table 2.1: Advantages and Disadvantages of Methodologies for robotics

2.5 VR Teleoperation for Demonstration Capture

This section contains the literature review of VR Teleoperations.

2.5.1 Fundamentals of VR Teleoperation

Virtual Reality (VR) teleoperation enables individuals to manipulate robots using immersive interfaces, commonly employing head-mounted displays (HMDs) and motion-tracked controllers. In contrast to conventional input devices like keyboards or joysticks, virtual reality facilitates naturalistic demonstration capture, allowing a robot to directly replicate the operator's movements. This methodology is especially beneficial for Learning from Demonstration (LfD), as it offers trajectories that are both intuitive for instruction and abundant in temporal and spatial context [21].

In a standard configuration, VR controllers are assigned to robotic end-effectors, whereas HMD tracking can offer orientation or navigational guidance. Such technologies advantageously exhibit high embodiment fidelity, allowing the operator to imagine oneself as "inhabiting" the robot. Research indicates that immersion enhances task accuracy and diminishes cognitive burden in teleoperation tasks [22].

2.5.2 Technical Methodologies

A variety of architectures for VR teleoperation have been established:

Direct mapping: Controller movements are directly correlated with joint locations or end-effector configurations. This is obvious but necessitates calibration to synchronise human and robot kinematics [21].

Client-server pipelines: Sensor data from virtual reality devices are transmitted to a host computer, which converts the input into commands for the robot. The methodology employed in this research involved connecting a Pico VR headset and controls to the Pepper robot over a Python bridge. The system utilised Virtual Desktop and SteamVR/OpenXR runtimes for headset input, employing a client-server communication mechanism to convert VR signals into NAOqi API calls for Pepper's arm movements and navigation.

Shared autonomy: Certain systems enhance VR teleoperation by AI support (e.g., collision avoidance, trajectory optimisation), facilitating safer navigation in intricate situations [21].

In our case, this client-server setup worked, though not without its frustrations for immersive control of Pepper, latency and payload constraints made it hard to collect structured demonstrations.

2.5.3 Utilisations in Robotics

Virtual reality teleoperation has been extensively utilised in both research and industry.

Remote manipulation enables operators to execute assembly or object-handling operations in perilous or remote locations [21].

Human-Robot Interaction (HRI): Pepper and analogous humanoids have been utilised in VR-mediated presentations to investigate how humans instruct robots in gestures and tasks, with favourable outcomes on engagement [21].

Learning from Demonstration: By capturing trajectories in VR teleoperation, robots can acquire abilities from human demonstrations with greater data economy than scripted control [1].

Assistive robotics: Virtual reality teleoperation has allowed impaired individuals to remotely operate humanoid robots, underscoring its promise for accessibility [21].

2.5.4 Advantages and Constraints

Advantages:

- Offers a natural, immersive demonstration capture in contrast to conventional input devices.
- Enhances task precision and user immersion in manipulation and navigation activities.
- Enhances Learning from Demonstration (LfD) pipelines by generating intuitive demonstrations readily applicable for robotic learning.

Constraints:

- Demands low-latency transmission among headset, controller, and robot, delays may impair performance and realism.
- The kinematic disparity between human and robot anatomies (e.g., human arm against Pepper's arm) may diminish the authenticity of demonstrations [21].
- The physical limitations of robots, such as payload capacity and gripper design, can restrict the range of jobs that can be teleoperated using virtual reality.
- Middleware requirements, such as NAOqi and ROS integration, may introduce further complexity.

This dissertation indicates that these limits necessitated a shift from Pepper VR teleoperation to the RoArm-M3 manipulator, facilitating more organised and reproducible demonstrations appropriate for symbolic induction.

2.6 Hardware Platforms: Pepper and RoArm-M3

Selecting the appropriate robotic platform is a critical design decision in research that integrates demonstration capture, symbolic learning, and physical execution. This section discusses the two platforms utilised in this project: the Pepper humanoid robot and the RoArm-M3 manipulator. It delineates their capabilities and limitations, as well as their efficacy in executing recursive Meta-Interpretive Learning (MIL).

2.6.1 Pepper Humanoid Robot

The Pepper robot, developed by SoftBank Robotics, serves as a prominent platform for research on human-robot interaction (HRI). Pepper stands at 1.2 meters and is equipped with RGB cameras, depth sensors, microphones, and touch sensors, facilitating multimodal interaction [23]. It possesses 20 degrees of freedom (DOF) distributed over its arms, base, and torso, enabling it to perform expressive motions and exhibit limited mobility. Pepper was a great starting point for testing teleoperation, it felt intuitive to control, but its small payload and limited gripper strength soon became frustrating bottlenecks.

Fig 2.2: Pepper Humanoid Robot

2.6.1.1 Hardware and Software Ecosystem

The NAOqi operating system powers Pepper. APIs in Python, C++, and Java provide control over movement, speech synthesis, and sensor access [23]. This renders it ideal for interactive applications in sectors such as retail, healthcare, and education. However, its payload capacity (about 200-300 g per hand) and rudimentary gripper design hinder its ability to perform more complex manipulation tasks [23].

2.6.1.2 Role in This Project

This project initially integrated Pepper with a Pico VR headset and controllers to facilitate immersive teleoperation. A bespoke Python client-server bridge connected controller input to Pepper's joints and navigation commands via the NAOqi API. The system employed Virtual Desktop and SteamVR/OpenXR for streaming VR data. The headgear gathered input from the camera, establishing a closed-loop immersive control system.

This method demonstrated the viability of VR-based demonstration capture, although it also revealed certain issues:

- Latency: The duration for the controller to transmit command to the robot for movement was approximately 500 ms, resulting in diminished responsiveness.
- Payload and gripper constraints: Pepper struggled to lift and place any objects requiring relocation.
- Integration challenges: NAOqi's middleware complicated the connection to symbolic learning pipelines (Python-Prolog interfaces).

Although Pepper was beneficial for exploring human centered teleoperation

interfaces, it was inadequate for generating structured, repeatable ILP training data.

2.6.2 RoArm-M3 Manipulator

The RoArm-M3 is a compact, lightweight manipulator designed for prototyping in educational and research contexts. It possesses 5 + 1 degrees of freedom, a base capable of 360-degree rotation, and a load capacity of 0.2 kg at a distance of 0.5 m. It can accommodate a workspace with a diameter of approximately 1 metre [24]. An ESP32 microcontroller governs the serial-bus servos, facilitating economical and straightforward integration with open-source software ecosystems.

Fig 2.3: RoArm-M3

2.6.2.1 Software Ecosystem

RoArm-M3 is compatible with ROS 2 and MoveIt 2, enabling the planning and execution of moves with conventional robotics tools [26][25]. It is also compatible with LeRobot, a Python-based framework for teleoperation and control [27]. Due to this interoperability, a JSON-based recorder was easily developed to log fundamental activities (take/1, put/1) for subsequent playback during symbolic rule execution.

Platform	DOF	Reach	Payload	Repeatability	Controller / OS	Ecosystem & Notes
RoArm-M3	5+1	~1 m workspace	0.2 Kg @ 0.5 m	Not specified	ESP32, serial-bus servos	ROS 2, MoveIt 2, LeRobot compatible
Niryo Ned2	6	440-490 mm	300 g	±0.5 mm	Embedded Linux, ROS	Comprehensive education stack, Blockly, Python, MATLAB support
myCobot 280	6	280 mm	250 g	±0.5 mm	Raspberry Pi / jetson variants	Compact, strong ROS ecosystem
DOBOT Magician	4(+R)	320 mm	500 g	±0.2 mm	Proprietary controller	Popular in STEM education, vision/conveyor add-ons
uArm Swift Pro	4	320 mm	500 g	±0.2 mm	USB/Blueto oth	Low-cost, rich community adoption

Table 2.2: Different Robotic Arm Comparison

In comparison to existing instructional devices with only 4 degrees of freedom (such as DOBOT and uArm), the RoArm-M3 occupies an intermediate position between these and more advanced 6-DOF platforms (like Niryo and myCobot). It lacks public repeatability metrics and has a limited load capacity, yet it possesses sufficient degrees of freedom and open interfaces for recursive learning experiments.

2.6.2.2 Suitability for This Project

We chose RoArm-M3 for three reasons:

- The simplicity of primitives: The manipulator facilitates the recording of fundamental activities, which is advantageous for individuals with military experience.
- Safety and accessibility: Due to its compact design and minimal payload, it is safe for frequent usage on a laboratory bench.
- Integration flexibility: The utilisation of ROS 2 and JSON-based control pipelines facilitates seamless connectivity with Python and Prolog, which is crucial for the functionality of a MIL learning loop.

2.6.2.3 Limitations and Mitigation

RoArm-M3 lacks established specifications for repeatability and is limited to handling modest loads. The restrictions were alleviated by concentrating on lightweight objects, adjusting recorded positions with safety margins, and emphasising symbolic generalization over high-precision assembly.

2.6.3 Comparative Discussion

Pepper and RoArm-M3 represent two distinct categories of robots. Pepper excels in interpersonal interaction and immersive teleoperation, rendering it ideal for evaluating VR-based demonstration capture. Nonetheless, its physical and technological constraints hinder its application in controlled, repeatable manipulation investigations.

Conversely, RoArm-M3 offers accurate, controllable motion primitives using a modular, open-source control stack, making it very compatible with ILP/MIL pipelines. It does not possess Pepper's anthropomorphic representation but fulfils the fundamental necessity of this research: recursive, explainable skill execution from one-shot demonstrations.

This contrast inspired the two-phase methodology: initially investigating VR teleoperation with Pepper to evaluate feasibility, followed by a switch to RoArm-M3 to execute the comprehensive MIL pipeline.

2.7 Summary

This chapter pulled together the main strands of thinking that shaped the project.

Learning from Demonstration (LfD) is still one of the most natural ways to teach a robot just show it what to do and it has been used extensively in both research and industry [1][2]. But it comes with baggage, it often needs a lot of data, produces models that behave like black boxes, and struggles with expressing things like recursion or repeating patterns [25][26][27].

Early on, I tried to make this work with Pepper. Using a Pico VR headset and controllers, I set up an immersive teleoperation loop that let me control Pepper in real time [22][23][28]. It was a good proof-of-concept, but Pepper's gripper strength, mechanical range, and roughly half-second latency quickly became frustrating precise pick-and-place just wasn't going to happen [19].

That experience pushed the project toward symbolic approaches, particularly Inductive Logic Programming (ILP) and its more powerful cousin, Meta-Interpretive Learning (MIL). These methods can learn neat, recursive rules from very small numbers of examples [3][4][6][7][9][10][11][12]. MIL, in particular, uses background knowledge and meta-rules to invent programs that generalise surprisingly well while staying completely interpretable. Of course, there are caveats scalability is still a challenge, and connecting symbolic systems to noisy, unstructured perception is an open problem [8].

Explainability kept coming up as a theme. Trusting a robot is hard when you can't tell why it's doing what it's doing. Work on explainable AI highlights the need for clarity, safety, and even regulatory compliance [5][13][14][15][16]. Neural networks might still be the best at perception, but they're opaque, which is why hybrid approaches combining perception from deep models with symbolic reasoning are becoming so appealing [8].

VR teleoperation, meanwhile, remains an interesting tool for capturing natural demonstrations. It offers embodiment and immersion, but its drawbacks are latency, mismatched kinematics, and platform quirks appear to make it a poor fit for high-precision manipulation [22][28]. Those limitations ultimately made the decision easy: switch to the RoArm-M3, a lightweight manipulator with precise, reproducible control and an open software stack [20][21].

Put together, these insights suggest a clear gap: a system that links immersive demonstration capture with symbolic reasoning, allowing a robot to learn a recursive task from just one example and then execute it on real hardware. The next chapter explains how this project tackled exactly that.

CHAPTER 3: RESEARCH APPROACH

3.1 Problem Refinement and Components

The research problem presented in Chapter 1 intended to look into the possibility of facilitating a robot's acquisition of recursive manipulation behaviours through minimal demonstrations utilising Meta-Interpretive Learning (MIL) [10][11]. To render this issue manageable for an MSc dissertation, it was divided into a series of smaller, resolvable sub-problems. Work was done on each sub-problem one at a time, using what we learnt from earlier stages to make design choices later on.

3.1.1 Physical Platform Selection

The choice of hardware platform was very important to the success of this project because it had to be able to handle both immersive demonstration capture and precise, repeatable manipulation for MIL-based learning. So, the project used two different robots at different times, which was a two-phase approach.

Phase 1: The Pepper Humanoid Robot

Fig 3.1: Pico VR

Pepper was chosen as the first platform because it is good for research on teleoperation and human-robot interaction (HRI) [23]. Its human-like shape, arm degrees of freedom, and connection to the NAOqi API made it possible to look into naturalistic demonstration capture using VR controllers [21][22].

During this phase, a custom VR teleoperation system was set up that linked the Pico VR headset and controllers to Pepper through a client-server bridge [28]. The system was able to map the poses of the controllers to the angles of the shoulders and the input from the joystick to the base velocity. This made it possible to move the arms and navigate in real time. This proved that VR interfaces could be used for demonstration capture and gave us useful experience in managing latency and networking with multiple devices.

But there were some challenges that made Pepper less useful for structured learning tasks:

- Network dependency: The laptop, Pepper, and the VR headset all had to be on the same network for Virtual Desktop streaming. This setup didn't work well with the university network, so a mobile hotspot had to be used.
- Software environment constraints: NAOqi libraries weren't available for Windows, so you had to set up both PowerShell (for SteamVR) and a Linux terminal (for Pepper control).
- Physical limitations: Pepper's gripper design, payload capacity, and chest tablet made it hard for it to do precise pick-and-place tasks.
- Latency: The delays from SteamVR, Virtual Desktop, and network communication added up to about 500 ms of end-to-end latency [22], which made it impossible to capture fine-grained demonstrations.

These results led to a switch to a platform that was better for making precise, repeatable changes.

Phase 2: The RoArm-M3 Manipulator

The RoArm-M3 manipulator was chosen for the second phase because it is light,

accurate, and easy to use with open-source software. The arm has 5+1 degrees of freedom, serial-bus servo control, and works with ROS 2, making it a good choice for research that needs low-level motion control [24][26].

The RoArm-M3 had a lot of advantages, such as:

- Direct controllability: The arm could be put into a free-drive mode, which let people record positions by hand and save them as JSON [24].
- Integration flexibility: The serial connection allowed for fast feedback logging, which solved problems with Wi-Fi latency [24].
- Safety: The small size and low weight of the payload made it easy to make changes quickly on a tabletop setup without worrying about breaking anything.
- Cost-effectiveness: It was possible to do more experiments because it was easy to get and cheap.

These features were in line with the project's need for precise, repeatable action primitives (take, put) and their integration into a symbolic reasoning pipeline.

Justification for Platform Transition

The switch from Pepper to RoArm-M3 was a methodological choice based on what was learnt during early tests. Although Pepper facilitated the validation of VR teleoperation as a concept, its hardware limitations hindered the production of functional ILP training data. On the other hand, RoArm-M3 had a reliable way to record and play back deterministic action sequences, which made it the best choice for the MIL learning and execution phase.

3.1.2 Software Setup

The software ecosystem had to link three moving pieces: VR-based teleoperation, robot control, and symbolic learning. To keep these components isolated yet interoperable, a dual-OS architecture was adopted.
On the Windows 11 host, SteamVR and Virtual Desktop connected the Pico headset [24], while test4client.py streamed controller pose and thumbstick data to a Flask endpoint. Real-time plots helped operators verify input before execution.
The control of the Robot was handled on Linux for compatibility and low-latency access to hardware. The NAOqi SDK, required for Pepper, was run through WSL due to lack of native Windows support [23]. The same machine hosted the Flask server (test5server.py), which forwarded VR input to Pepper’s ALMotion API.
RoArm-M3 control was moved to a dedicated Ubuntu machine to avoid dependency conflicts and guarantee real-time responsiveness. This system executed the hardware replay script (robot_command.py) and position recorder (individual_position_recorder.py). Splitting teleoperation and control across platforms allowed simultaneous VR streaming and deterministic robot execution.
Python 3.10 was used throughout, with libraries including Flask, requests, and OpenVR for input handling [28], OpenCV for camera streaming, and pyserial for direct arm control. SWI-Prolog and Metagol provided the learning backend, with

process_create/3 bridging symbolic rules (e.g., take/1, put/1) to Python executors [10][12].
Early trials used Wi-Fi to control the arm but showed unpredictable delays. This was resolved by switching to a serial connection, which improved determinism and eliminated jitter [24]. RoArm-M3's onboard controller performed its own inverse kinematics, so only target Cartesian positions needed to be recorded. This choice reduced storage overhead and simplified replay [25][26].
To keep the environment reproducible, separate virtual environments were maintained for VR teleoperation, RoArm-M3 control, and symbolic learning. This modularity allowed one subsystem to be debugged without breaking another.
Key integration challenges included network segregation (solved using a mobile hotspot for VR + Pepper), NAOqi's Windows incompatibility (handled via WSL/Linux), and VR control latency (~500 ms) which partly motivated the move to RoArm-M3. These decisions together produced a modular, repeatable software stack capable of supporting both demonstration capture and precise execution.

3.2 Perception

Perception is a key part of most robotic learning pipelines because it lets the robot see what's around it, find things, and change how it acts based on what happens. For this project, though, perception was intentionally made simpler so that the research question could focus on symbolic learning and recursion.

Visual feedback was only used to keep the operator aware during the Pepper VR teleoperation phase. The videostream.py script connected to Pepper's head camera at 30 frames per second and showed the feed in real time. This let the operator check the movements of the arm and base while using Pico VR controllers to control the robot. In this phase, no decisions were made based on perception or datasets were created.

During the RoArm-M3 phase, perception was completely left out. Instead of using computer vision or object detection models, we treated the positions of the objects and the manipulator end-effector as known states. A custom position recorder script let the arm motors go, so the experimenter could move the arm to the poses they wanted. After that, these target poses were saved as JSON coordinates so they could be used later.

Two things made us want to take this approach:

- Scope and feasibility: Building or training a computer vision pipeline for object detection and pose estimation would have needed a lot more resources, which might have taken away from the focus on MIL and recursive task learning [5].

- Controlled evaluation: The project used pre-recorded positions to make sure that the results could be repeated and to get rid of perception noise as a confounding factor. This made it easier to evaluate the symbolic learning pipeline.

So, the perception part of this project was more of a supporting mechanism than the main focus of the research. Not including computer vision also made the pipeline more predictable and easier to check, given the time limits of the MSc dissertation

3.3 Control

The control strategy for this project was built around using Meta-Interpretive Learning (MIL) to represent tasks symbolically and come up with rules. The goal was to create generalisable, recursive control programs from just one demonstration and run them on the robot instead of hard-coding sequences of actions.

3.3.1 Symbolic Representation of Actions

The manipulation task was divided into two parts:

- take/1 - pick up a box from a specified position
- put/1 - place the box at a specified position

In robot_command.py, these primitives were written in Python. Every time they were called, the RoArm-M3 would move to a recorded Cartesian coordinate and do the action that was asked of it.

Example (Python-executed Prolog call):

```
put(A):-
    atom_string(A, AStr),
    process_create('/usr/bin/python3', ['robot_command.py', 'put', AStr], []).
take(A):-
    atom_string(A, AStr),
    process_create('/usr/bin/python3', ['robot_command.py', 'take', AStr], []).
```

By limiting the action space to just two primitives, Metagol was able to keep the hypothesis space small and manageable while still allowing recursion to be expressed [10][11][12].

3.3.2 Metagol Setup and Rule Induction

Metagol, a meta-interpreter for inductive logic programming that lets you learn recursive programs from examples, was used to make the MIL learner [12].

Configuration:

- Max Clauses: Limited to 2 using metagol:max_clauses(2).
- Background Knowledge: Included definitions for move/1 and rest_boxes/2

- Meta-Rules: Two meta-rules were supplied, base and mutual, to allow recursive definitions [10][11].

```
metarule(base,[P,Q,R],[P,A],[[Q,A],[R,A]]).
metarule(mutual,[P,Q,R],[P,A],[[Q,A,B],[R,B]]):-
  metagol:type(R,1,head_pred).
```

- Training Examples: Learning was performed in a one-shot setting using a single positive example list for robot/1.

```
learn:-
  Pos=[robot([3,2,1]),robot([2,3,1]),robot([2,1]),robot([1,2])],
  Neg= [],
  learn(Pos,Neg).
```

From this input, Metagol induced a recursive hypothesis defining robot/1 such that it could generalise to lists of arbitrary length.

3.3.3 Python-Prolog Integration

The learnt hypothesis was implemented on the physical RoArm-M3 by connecting Prolog to Python through process_create/3 . When a learnt predicate like robot([3,2,1]) was used, Prolog ran Python scripts that told the arm to move to the right places one after the other.

This integration made a closed loop where:

- The symbolic rule set the order of the steps.
- Python turned each predicate call into commands for Cartesian motion.
- RoArm-M3 did the movement in real life.

3.3.4 Recursive Execution Verification

To confirm the generalization ability of the induced hypothesis, the trained program was evaluated using input lists that exceeded the length of the initial demonstration. For instance, when given robot([5,4,3,2,1]), the system was able to perform a series of take and put actions for all elements, showing that it could really recurse beyond the training distribution [10][11].

The execution was seen to stay the same for lists of length 5, 6, and 7. The only restriction was that poses had to be pre-recorded and saved in JSON for each element. This result shows that the recursion had no logical limits, and the only limits that mattered were the states that were already recorded.

This control pipeline shows how powerful MIL is at making interpretable, recursive policies from very little data and using them with real robotic hardware.

3.4 System Integration

The parts that were talked about in the last sections were put together to make a full pipeline that can go from human demonstration to autonomous recursive execution. The system was made to be modular, with clear interfaces between each stage. This made it possible to test and improve each part on its own before testing the whole thing.

3.4.1 WorkFlow Overview

The entire workflow was put into place in stages. First, Pepper used VR to teleoperate data capture. Then, the RoArm-M3 used structured recording, Metagol used rule induction, and finally, the hardware ran the process.

The setup for the Demonstration Capture phase with Pepper included both VR hardware and real-time control scripts. The Pico VR controllers, which worked with SteamVR, sent six degrees-of-freedom (6-DoF) pose data [28]. A Python script called test4client.py worked as an OpenVR client. It sent information about the selected pose, such as pitch orientation and joystick input, to a Flask server running on a Linux machine (test5server.py). After being received, this information was turned into commands for the robot, ALMotion.setAngles set Pepper's joint angles, and ALMotion.moveToward [23] set base speed and direction of movement. A second script called videostream.py streamed Pepper's camera feed in real time so that the operator could see what the robot saw while it was being operated remotely. As we will see in later sections, this feedback loop made immersive control possible, even though there was some lag.

After this, the Recording of Movement Primitives was moved to the RoArm-M3 robot, which was more mechanically accurate and reliable. After confirming that teleoperation was theoretically viable, the project progressed to organised data collection. To temporarily turn off the arm's motors, the user used a Python script called individual_position_recorder.py. This let them move the arm by hand to the right take and put positions for each object. These positions were recorded as Cartesian coordinates and stored in high-frequency JSON snapshots, like take_1.json and put_1.json. This hands-on, manual method made sure that the inputs for the next learning phase were clean and could be repeated.

After recording enough primitives, the Induction of Recursive Logic was done with Metagol, a well-known Meta-Interpretive Learning (MIL) engine [10][12]. The coordinates stored in JSON were changed into Prolog background knowledge, which served as the known primitives (take/1, put/1). A single positive example, like the sequence [3,2,1], was given to start the learning process. Metagol used a small set of pre-defined meta-rules, like base and mutual, to help the hypothesis space and successfully came up with a recursive program, which is formally known as robot/1. This hypothesis generalised across object lists of different lengths, not just memorising the training sequence but also creating a compositional and recursive understanding of the task.

Finally, during the Execution Phase, the hypothesis that had been learnt was put into action on real hardware. In SWI-Prolog, a command like robot([5,4,3,2,1]) made the rules run. Every time the take/1 or put/1 predicate was called, it called Python internally using process_create/3 and gave control to a script called robot_command.py. This script read the right JSON coordinate file and sent commands to the RoArm-M3 over a serial connection. The manipulator then did each motion exactly and in order, finishing the task of manipulating the object. The rule's structure made sure that the whole list was handled recursively, no matter how long it was. This showed that the system could generalise from small examples.

3.4.2 Data Flow and Modularity

The system was purposefully designed with weak connections between stages:

- Teleoperation was only used to show that it was possible, and it didn't limit the learning phase.
- The recording module made JSON data that could be used again or replaced without changing the logic for learning or execution.
- The MIL learner came up with a symbolic hypothesis that didn't depend on the robot hardware, so it could be used with other platforms as well.

This modular design made it easier to maintain, let you debug at different stages, and made sure that the symbolic learning part could be tested without any noise from the hardware.

3.4.3 SYSTEM ARCHITECTURE DIAGRAM

The overall integration is illustrated in Figure 3.1, which shows the data flow between teleoperation, recording using RoArm-M3, symbolic learning, and physical execution.

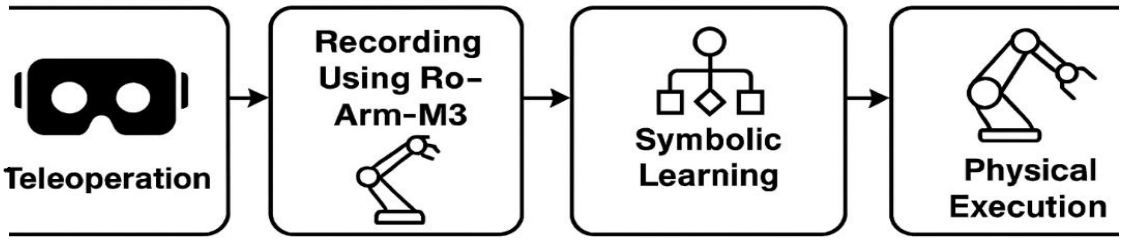

Fig 3.2: End-to-End Pipeline for VR Demonstration, Recording, MIL Learning, and Recursive Execution

This architecture represents a complete, repeatable workflow for capturing demonstrations, learning symbolic rules, and validating their generalization capabilities in real robotic systems

3.5 Testing and Evaluation Plan

A plan for systematic testing and evaluation was made to check how well the proposed pipeline works and to see if Meta-Interpretive Learning (MIL) can learn from only a few examples. The evaluation centred on three principal enquiries:

Feasibility: Is it possible for the integrated pipeline to run learnt recursive tasks on real hardware?

Generalization: Is it possible for a hypothesis derived from a solitary positive instance to extend to previously unobserved input lists of any length?

Performance: Are the behaviours that were learnt dependable, repeatable, and stable across several trials?

3.5.1 Steps for the Experiment

The experimental procedure was organised in the following way:

- Recording in its most basic form:
- take and put, we used individual_position_recorder.py to record the positions of n objects (like 3, 4, or 5).
- Before learning, the recorded positions were checked for accuracy by playing them back once.

Learning the Rules:

- Metagol was given a single list of positive examples (like robot([3,2,1])) and some background information.
- The induced hypothesis was saved and then used again for several test lists without having to retrain.

Execution that goes recursively:

- The learnt robot/1 predicate was run on input lists that were longer than those used during training (for example, 4, 5, 6, and 7) [11].
- We watched and recorded each execution, and success was defined as the correct order of all take and put actions without any collisions or mistakes.

Repetition:

We did each configuration (list length) five times to see if it was repeatable and to account for random changes, like slight arm offsets.

3.5.2 Metrics

The following metrics were selected to evaluate system performance

- Success Rate (%)

$$Success\ Rate = \frac{Successful\ Trials}{Total\ Trials} \times 100$$

If the arm completed the whole sequence without making any mistakes, the trial was considered a success.

- Generalization Test: A binary outcome that shows whether the hypothesis worked for list lengths longer than those used in training.
- Latency Observation (Pepper Phase): A qualitative observation of end-to-end latency during VR teleoperation indicated a perceived delay of approximately 500 ms, which informed the decision to transition to RoArm-M3.

3.5.3 Criteria of Evaluation

- If the system was thought to be successful,
- The success rate was over 80% for all tested list lengths.
- The hypothesis was still true for all generalization tests (longer lists).
- The execution time increased in a roughly linear way with the length of the list, which showed that recursion didn't add any extra computational work.

This evaluation plan made sure that the proposed MIL-based control pipeline was thoroughly tested for both functional correctness and robustness. Chapter 5 gives a report on the results of these tests, along with some representative screenshots, and critically analyses them.

3.6 Summary

This chapter described the research methodology utilised to examine the implementation of Meta-Interpretive Learning (MIL) for recursive robotic manipulation tasks. The issue was divided into a series of clearly defined sub-problems, starting with the verification of VR-based teleoperation utilising the Pepper humanoid robot, and subsequently the creation of a structured recording-execution pipeline for the RoArm-M3 manipulator.

The decision about which hardware platform to use was justified because Pepper could capture immersive demonstrations but was not suitable for structured ILP training data. The RoArm-M3 was chosen as the main platform for the learning phase because it is accurate, modular, and easy to integrate.

The software environment was well explained. It included the dual-OS setup (Windows for VR streaming and Linux for controlling NAOqi and RoArm) and the Python-Prolog bridge that was used to run learnt rules. Perception was intentionally omitted to preserve a controlled and replicable environment, concentrating the assessment on symbolic learning and recursion.

Metagol was the basis for the control pipeline. It used background knowledge and meta-rules to make recursive hypotheses from just one positive example [10][12]. The symbolic program that came out of this was run on real hardware through a serial connection, closing the loop from demonstration to autonomous execution.

Finally, a strict plan for testing and evaluating was made. This plan included tests for success rate, execution time, and generalization for longer input sequences. In Chapter 5, these metrics will be used to see how well the proposed approach works and how strong it is.

Now, Chapter 4 will go into detail about how each part of the system works, including teleoperation scripts, a position recorder, MIL configuration, and a hardware executor. It will also talk about design choices and show code snippets.

CHAPTER 4: DATA ANALYSIS (IMPLEMENTATION)

The official name of this chapter is "Data Analysis", but the data for this project wasn't a clean CSV file from Kaggle, it had to be made. So, the "analysis" here couldn't be done without building the pipeline to get, clean, and use that data. The implementation of the system was the same thing as the data analysis.

This chapter doesn't just show a table of statistics. Instead, it shows how the data was collected from the robot, turned into a symbolic representation, put into a Meta-Interpretive Learning (MIL) system, and used to create recursive control rules. After that, the rules were run on the real hardware to see if they worked as expected.

We will follow the structure mentioned below for this section:

- 4.1 Platform Integration (Pepper and RoArm-M3)
- 4.2 Software environment Configuration
- 4.3 Perception Setup & Data Collection
- 4.4 Control System Design
- 4.5 System Integration & Debugging
- 4.6 Deployment & Functional Verification
- 4.7 Summary

4.1 Platform Integration

The first step in putting the plan into action was getting the robots to talk to us. "Connect and control" sounds easy, but it was one of the hardest and most informative parts of the project.

4.1.1 Pepper Setup

Network Connectivity and Initialization

For NAOqi control to work, Pepper, the VR headset, and my laptop all had to be on the same subnet [23]. It wasn't as easy to get Pepper to talk to the laptop as just plugging in a cable. University Wi-Fi liked to put devices on different subnets, so Pepper, the VR headset, and the client laptop couldn't see each other. The workaround

was, quite literally, a phone hotspot that put all three devices on the same network.

The control script set up a Qi session after connecting, using arguments instead of hardcoding the IP address [23]. This meant that anyone who wanted to try it on their own Pepper could use the script again:

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--ip", type=str, default="192.168.199.53",
                        help="Robot IP address. On robot or Local NAOqi: use '127.0.0.1'.")
    parser.add_argument("--port", type=int, default=9559,
                        help="NAOqi port number")

    args = parser.parse_args()

    try:
        # Initialize the Qi session before starting the Flask app
        session = qi.Session()
        session.connect("tcp://" + args.ip + ":" + str(args.port))

        # Set Pepper to neutral position upon connection
        motion_service = session.service("ALMotion")
        set_neutral_position(motion_service)

    except RuntimeError:
        print(f"Can't connect to NAOqi at ip \"{args.ip}\" on port {str(args.port)}.\n"
              "Please check your script arguments. Run with -h option for help.")
        sys.exit(1)
```

Using --ip on the command line meant that no one had to change the code to switch networks. It was a small change, but it saved time.

Once the connection was live, the Flask server got incoming VR payloads and turned them into shoulder pitch angles and base speeds [28].

```
def control_shoulders():
    global session
    try:
        motion_service = session.service("ALMotion")

        # Get the data from the POST request
        data = request.get_json()
        angles_r = data.get("pitch_r")
        angles_l = data.get("pitch_l")
        thumbstick_x = data.get("thumbstick_x")
        thumbstick_y = data.get("thumbstick_y")
```

```
        fractionMaxSpeed = 0.1

        # Control shoulder pitch angles
        if angles_l is not None:
            motion_service.setAngles("LShoulderPitch", angles_r, fractionMaxSpeed)

        if angles_r is not None:
            motion_service.setAngles("RShoulderPitch", angles_l, fractionMaxSpeed)

        speed_factor = 0.7

        # Handle thumbstick input
        if thumbstick_x is not None and thumbstick_y is not None:
            reduced_x = thumbstick_x * speed_factor
            reduced_y = thumbstick_y * speed_factor
            motion_service.moveToward(reduced_y, 0 , -reduced_x)

        return "ok", 200
```

It wasn't perfect there was a half-second delay, but it was good enough for basic teleoperation.

To be safe, a different camera viewer script subscribed to Pepper's head camera so I could see what it was doing live [23].

```
camera = session.service("ALVideoDevice")

# Subscribe to the camera and set the resolution
resolution = 1  # 640x480 resolution
color_space = 11  # kBGRColorSpace (color image)
fps = 30  # Frames per second
name_id = camera.subscribe("Head", resolution, color_space, fps)

try:
    while True:
        # Capture a frame
        image = camera.getImageRemote(name_id)

        if image is not None:
            # Extract image data and display its size
            width = image[0]
            height = image[1]
            image_data = np.frombuffer(image[6], dtype=np.uint8).reshape((height,
width, 3))
            image_data=cv2.resize(image_data,(1280,720))
            # Display the frame
            cv2.imshow("Pepper Camera Feed", image_data)
```

This seemingly simple view prevented collisions during operation and proved invaluable.

4.1.2 VR Client

The VR client script took care of calibration, pitch calculation, and inputs from the thumbstick. To calibrate, you had to hold the controllers in a neutral position and use that as the starting point [28]. To avoid exaggerated motions, pitch was then calculated using a scaling factor.

```
start_time = time.perf_counter()  # Record the start time

        position_1, position_2 = get_calibrated_position(vr_system)
        pitch_1 = calculate_angles(position_1, baseline_pitch_1)
        pitch_2 = calculate_angles(position_2, baseline_pitch_2)

        print(f"Controller 1 - Calibrated Position: {position_1}")
        print(f"Controller 1 - Pitch: {pitch_1} degrees")
        print(f"Controller 2 - Calibrated Position: {position_2}")
        print(f"Controller 2 - Pitch: {pitch_2} degrees")

        # Invert the pitch before sending
        pitch_1_inverted = pitch_1 * -1
        pitch_2_inverted = pitch_2 * -1

        thumbstick_x, thumbstick_y = get_thumbstick_input(vr_system)

        if thumbstick_x is not None and thumbstick_y is not None:
            print(f"Thumbstick X: {thumbstick_x}, Thumbstick Y: {thumbstick_y}")

        # Send the inverted pitch data and thumbstick data to the server
        response = send_data(pitch_1_inverted * 0.01745, pitch_2_inverted *
0.01745, thumbstick_x, thumbstick_y)
```

This is where things started to feel "real" when you could lift your controller and see Pepper's arms move a split second later. It took some trial and error to get the scaling factor right [22]. If it was too high, Pepper flailed around, and if it was too low, it felt unresponsive.

4.1.3 RoArm-M3 Setup

Pepper was fun, but its grippers weren't very good, and it took too long to respond, so it wasn't good for repeated tests. The next step was switching to RoArm-M3.

The first attempt used Wi-Fi, but network jitter made it behave unpredictably [24]. It felt more "solid" right away when I switched to direct USB serial.

It was easy to record demonstrations: just let go of the servos, move the arm by hand, and log every Cartesian coordinate until you were done.

Joints Release

```
def release_joints_official(ser):
    """Release joints using official Waveshare commands - exact same as
hanoi_clear_recorder"""
    print(" Releasing joints for manual movement...")

    # Official torque control command from Waveshare documentation
    torque_release = {"T": 210, "cmd": 0}

    print(f" Sending official torque release...")
    response = send_command_and_get_response(ser, torque_release)

    if response:
        print("Torque release command sent")
        time.sleep(2)

        # Additional release commands as backup
        backup_commands = [
            {"T": 1000},  # Release all joints
            {"T": 1002},  # Release base
            {"T": 1010},  # Release shoulder
            {"T": 1020},  # Release elbow
        ]

        print(" Sending backup release commands...")
        for cmd in backup_commands:
            send_command_and_get_response(ser, cmd)
            time.sleep(0.5)

        print(" All joint release commands sent!")
        print(" You should now be able to move the arm manually")
        return True
    else:
        print(" Joint release failed")
        return False
```

Position Recorder

```
def get_position_data(ser):
    """Get current position and joint data - exact same as hanoi_clear_recorder"""
```

```
    response = send_command_and_get_response(ser, {"T": 1051})
    if response:
        try:
            data = json.loads(response)
            position = {
                'x': data.get('x', 0),
                'y': data.get('y', 0),
                'z': data.get('z', 0),
                't': data.get('tit', 0),
                'joints': {
                    'base': data.get('b', 0),
                    'shoulder': data.get('s', 0),
                    'elbow': data.get('e', 0),
                    'twist': data.get('t', 0),
                    'roll': data.get('r', 0),
                    'gripper': data.get('g', 0)
                }
            }

            print(f" Position: X={position['x']:.1f}, Y={position['y']:.1f},
Z={position['z']:.1f}, T={position['t']:.3f}")
            return position
        except json.JSONDecodeError as e:
            print(f" Could not parse response: {e}")
            return None
    return None
```

Fig 4.1: Recording the States

JSON SAVE

```
def save_position_to_file(position_data, operation_type, number):
    """Save position data to individual JSON file using the same format as our existing system"""
    filename = f"{operation_type}_{number}.json"

    # Use the same nested structure that works with our playback system
    file_data = {
        "operation": operation_type.upper(),
        "number": number,
        "position": position_data,
        "timestamp": time.time(),
        "recorded_by": "individual_position_recorder"
    }

    try:
        with open(filename, 'w') as f:
            json.dump(file_data, f, indent=2)
        print(f" Position saved to: {filename}")
        return filename
    except Exception as e:
        print(f" Failed to save to {filename}: {e}")
        return None
```

Each file (e.g., put_1.json, take_2.json) had the coordinates, and joint angles. This was useful when checking to see if it could be reproduced.

The executor then played these paths back in a set order. It first loaded the right JSON file, then moved the arm and turned the gripper on and off in that order.

```
def execute_put(box_number):
    """Execute PUT operation"""
    position_name = f"put_{box_number}"

    # Connect to robot
    ser = connect_robot()
    if not ser:
        return False

    try:
        # Load position
        position_data = load_position(position_name)
        if not position_data:
            return False

        print(f"EXECUTING: PUT {box_number}")
```

```
        # Move to position with closed gripper (carrying object)
        if not move_to_position(ser, position_data, open_gripper=False):
            print(f"ERROR: Failed to reach {position_name}")
            return False

        time.sleep(1)

        # Open gripper to release
        if not move_to_position(ser, position_data, open_gripper=True):
            print(f"ERROR: Failed to open gripper at {position_name}")
            return False

        time.sleep(1)

        print(f"SUCCESS: PUT {box_number} completed")
        return True

    finally:
        ser.close()
```

Similarly, the take command opened the gripper before moving, then closed it to "grab" the object.

```
def execute_take(box_number):
    """Execute TAKE operation"""
    position_name = f"take_{box_number}"

    # Connect to robot
    ser = connect_robot()
    if not ser:
        return False

    try:
        # Load position
        position_data = load_position(position_name)
        if not position_data:
            return False

        print(f"EXECUTING: TAKE {box_number}")

        # Move to position with open gripper
        if not move_to_position(ser, position_data, open_gripper=True):
            print(f"ERROR: Failed to reach {position_name}")
            return False

        time.sleep(1)
```

```
        # Close gripper to grab
        if not move_to_position(ser, position_data, open_gripper=False):
            print(f"ERROR: Failed to close gripper at {position_name}")
            return False

        time.sleep(1)

        print(f"SUCCESS: TAKE {box_number} completed")
        return True

    finally:
        ser.close()
```

This setup made demonstration reproducible and kept the data pipeline consistent for one-shot learning.

4.2 Software Environment Configuration

Setting up the software environment required careful planning to make VR teleoperation, robot control, symbolic learning, and hardware execution run reliably across multiple systems. This section summarises the final configuration, key decisions, and the challenges that shaped them.

4.2.1 Operating System and Development Setup

A single operating system was not sufficient for this project. Windows 11 was selected for VR teleoperation because it supports SteamVR and Virtual Desktop [24], while robot control tasks were moved to Ubuntu 20.04 to ensure compatibility and low-latency access to hardware [23].

Windows 11 (Host Laptop): Ran SteamVR and Virtual Desktop for Pico VR streaming, executed test4client.py to capture controller pose data, and displayed live plots to verify input. This particular setup also used bash terminal of VS Code for running Pepper control Scripts and the Flask server (test5server.py)

Ubuntu 20.04 (Dedicated PC): RoArm-M3 control was managed by using a direct serial link and ran SWI-Prolog with Metagol for symbolic learning [10][12]. Separating these tasks prevented dependency conflicts and kept teleoperation responsive even during learning or execution cycles.

This complete arrangement allowed VR streaming, Pepper teleoperation, and RoArm-M3 execution to operate in parallel without interfering with one another, improving overall system stability.

4.2.2 Hardware and Software Stack

Component	Version/Details	Purpose
Laptop (Host)	Intel i9 / RTX 4090, 32GB RAM	VR client, visualisation, and development
OS (Client)	Windows 11	SteamVR runtime, OpenVR client, Pepper control (VSCode bash terminal)
OS (Robot Control)	Ubuntu 20.04 LTS	RoArm-M3 serial, SWI-Prolog
VR Headset	Pico VR (wireless via Virtual Desktop)	VR teleoperation input
Robots	Pepper (SoftBank Robotics), RoArm-M3	Teleoperation tests(Pepper), execution platform (RoArm-M3)
Programming Language	Python 3.10	Teleoperation, recording, and executor scripts
Prolog Engine	SWI-Prolog	Ran Metagol for MIL
Key Libraries	Flask, OpenVR, OpenCV, NAOqi, pyserial	Networking, streaming, serial communication

Table 4.1: Summary of hardware and software environment

4.2.3 Installation and Configuration Challenges

Several technical issues had to be resolved before the system became stable enough for repeated trials. NAOqi SDK would not run natively on Windows, which led to relocating Pepper control to a dedicated Ubuntu machine. Early experiments also required SteamVR to run in PowerShell while Pepper scripts ran in a separate terminal, a setup that was error-prone until VS Code's integrated terminals allowed both environments to be monitored side by side. RoArm-M3 control initially used Wi-Fi, but its unpredictable delays made precise replay impossible, switching to a USB serial connection removed the latency and improved repeatability. Finally, the process_create/3 predicate in SWI-Prolog occasionally failed silently due to path resolution issues, which was permanently fixed by explicitly specifying /usr/bin/python3. Together, these adjustments produced a robust, reproducible environment in which teleoperation and symbolic execution could run without interfering with one another.

4.2.4 Virtual Environment Management

In the end, I made three separate Python virtual environments for everything:

- VR Teleoperation had Flask, requests, and OpenVR in it.
- RoArm Control had pyserial in it and, for a while, some ROS 2 dependencies.

- MIL Execution, A bare minimum environment is needed to run the bridge script.

This modular structure made it easier to replicate the setup on different machines and allowed quick identification of problems when they arose, since each subsystem could be tested independently.

4.3 Perception System

Perception usually has a big part to play in how robots move things around because it lets them see, find, and be aware of their surroundings. In this project, perception was intentionally simplified to isolate the impact of symbolic learning and recursion on task performance.

4.3.1 Pepper Camera Feedback

During the VR teleoperation phase, the only thing that Pepper's camera was used for was to help the operator stay aware of their surroundings. The videostream.py script used ALVideoDevice to get to Pepper's head camera and show a 640×480 live feed using OpenCV [23]. This let the human operator see the arm and base movements during teleoperation, which made sure that the operation was safe.
At this point, there was no object detection, image segmentation, or dataset generation based on vision. Instead of being used as an input to the control system, the camera feed was used as a passive verification tool.

4.3.2 Exclusion of Automated Perception

Perception was completely left out of the RoArm-M3 phase. Instead, object positions were considered known states and were manually documented using individual_position_recorder.py. This method had two benefits:

- Controlled Environment: By getting rid of perception noise, the focus stayed on testing how well Meta-Interpretive Learning (MIL) could create correct recursive control rules.
- Repeatability: Using pre-recorded Cartesian coordinates made sure that all trials started in the same way, which made it easier to reproduce and compare results.

4.3.3 Justification for Design Choice

The choice to leave out automated perception was made to keep the project manageable within the time limits of the MSc program. Setting up a full computer vision pipeline would have made it harder to collect data, train models, and tune them. The project made perception easier, which allowed it to create a clean proof-of-concept for one-shot recursive learning. In the future, this can be expanded with perception modules like object detection and pose estimation.

This perception setup made sure that visual inputs were helpful rather than confusing, which made it easier to judge how well the MIL component worked.

4.4 Control System

The main idea behind this project was very simple: show the robot how to do something with just one example and then see if it can apply what it learnt to other situations. This meant making the control system as abstract as possible so that a person could read, understand, and fix any problems with the rules the system learnt.

4.4.1 Symbolic Action Representation

There were only two basic commands instead of dozens of low-level ones:

- take/1: Move your arm to the object's location and pick it up.
- put/1: Go to the place and put the thing down

Both of these were written in Python (robot_command.py) and connected directly to the RoArm-M3 through a serial interface. Each action used a pre-recorded JSON file (like take_3.json or put_1.json) that had Cartesian positions in it. This made every run the same, so there was no need to guess where the object was [24].

The Prolog definitions were short enough to fit on one slide:

```
put(A):-
    atom_string(A, AStr),
    process_create('/usr/bin/python3', ['robot_command.py', 'put', AStr], []).
take(A):-
    atom_string(A, AStr),
    process_create('/usr/bin/python3', ['robot_command.py', 'take', AStr], []).
```

It may seem too simple to keep the action space so small, but it made Metagol's search space manageable and gave recursion room to shine.

4.4.2 Metagol Configuration

The engine of this learning process, Metagol, was set up so that it could "invent" just the right amount of structure without going off the rails.

Clause Limit: Two clauses at most (metagol:max_clauses(2)) [12], which pushed it to make rules that were short and general instead of long and specific.

Background Knowledge: Predicates like move/1 and rest_boxes/2 were given so that it could take one item off a list and go back to the rest.

In Metagol, meta-rules are formally defined using the metarule/3 construct, which specifies the **structure of allowed hypotheses**. The syntax is as follows:

```
metarule(Name, Head, BodyList).
```

Where, Name is the identifier for the meta-rule (e.g., base, mutual, or chain), Head defines the target predicate and its arguments (e.g., [P,A]), and BodyList is a list of predicates and their arguments that form the rule body (e.g., [[Q,A],[R,A]]) [4].

This structure determines how predicates can be composed during learning. For instance, the meta-rule below enforces that the predicate P can be constructed from two other predicates Q and R, both applied to the same argument:

```
metarule(base,[P,Q,R],[P,A],[[Q,A],[R,A]]).
```

Here, P, Q, and R are placeholders for predicate names, and A is a shared argument. This rule effectively captures a non-recursive composition of two predicates.

In contrast, recursive behavior can be guided by the following meta-rule:

```
metarule(mutual,[P,Q,R],[P,A],[[Q,A,B],[R,B]]):-
   metagol:type(R,1,head_pred).
```

This indicates that P is defined in term of predicates Q and R, Q transforms A into an intermediate B, and R is a recursive predicate, which refers back to the head predicate P [4].

This Directive metagol:type(R,1,head_pred) explicitly tells Metagol to treat R as the recursive call, enabling mutual recursion or recursive chaining, depending on the context. By specifying such constraints, meta-rules both guide and restrict the hypothesis space, making the learning process more efficient and interpretable [4].

4.4.3 Training Examples

The learner only had a few examples to work with, and they were all in one shot:

```
learn:-
   Pos= [robot([3,2,1]),robot([2,3,1]),robot([2,1]),robot([1,2])],
   Neg= [],
   learn(Pos,Neg).
```

It might seem dangerous to train on so few cases, but the whole point was to see if it could figure out the general pattern [10][11], "pick the largest first, then recurse."

4.4.4 PYTHON-PROLOG BRIDGE

Prolog also worked after the hypothesis was made. Process_create/3 was the link between the logical world and the real robot.

- Step 1: Prolog chose whether to take or put the next action.
- Step 2: Python opened the correct JSON file and played back the path.
- Step 3: The RoArm-M3 did it in hardware.

Because of this closed-loop design, the same learnt program could be tested on the real system right away, not just on paper.

4.4.5 Verification of Recursive Behaviour

The real test was giving it a longer list than it had ever seen before. For example:

```
?- robot([1,2,3,4,5]).
```

Fig 4.2: Command

The output on the console showed a neat sequence of steps for taking and putting things. It was strangely satisfying to see the arm repeat the pattern for a five-element list. This showed that it wasn't just memorising examples but actually using a recursive rule [10][11].

```
aqib@aqib-Legion-Pro-7-16IRX9H:~/Desktop/roarmpython-env/MIL$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.3.25-6-g80182bcd7)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- ['test 3'].
true.

?- learn.
% learning robot/1
% clauses: 1
take one box
put one box
take one box
put one box
take one box
put one box
take one box
put one box
take one box
put one box
take one box
put one box
take one box
put one box
take one box
put one box
take one box
put one box
robot(A):-move(A),move(A).
true
Unknown action: ' (h for help)
Action?
```

Fig 4.3: Learning and generating clauses

```
?- robot([1,2,3,4,5]).
EXECUTING: TAKE 5
SUCCESS: TAKE 5 completed
OK
EXECUTING: PUT 5
SUCCESS: PUT 5 completed
OK
EXECUTING: TAKE 4
SUCCESS: TAKE 4 completed
OK
EXECUTING: PUT 4
SUCCESS: PUT 4 completed
OK
EXECUTING: TAKE 3
SUCCESS: TAKE 3 completed
OK
EXECUTING: PUT 3
SUCCESS: PUT 3 completed
OK
EXECUTING: TAKE 2
SUCCESS: TAKE 2 completed
```

Fig 4.4: python executor

Final recursive rule for this use case which was here is the pair of mutually recursive

predicates robot/1 and robot_1/1. Each time you call one of these, it takes the biggest item off the list and applies the same logic to the rest of the list using move/1 as the base action.

```
robot(A):-move(A),robot_1(A).
robot_1(A):-rest_boxes(A,B),robot(B).
```

This control system met the project's goal of being a data-efficient, understandable way to go from one demonstration to a general recursive strategy. It even worked on real hardware.

4.5 System Integration & Debugging

After checking each part of the VR teleoperation, RoArm-M3 recording-execution pipeline, and Meta-Interpretive Learning (MIL) configuration, they were all put together into one workflow. In this part, you'll learn how the parts were put together, tested together, and fixed to make a smooth end-to-end pipeline.

4.5.1 Workflow Orchestration

The final system used a modular four-stage workflow, which made it easier to make changes, test each part, and find bugs before putting everything together. In the first stage of the demonstration, data was collected in two different ways, depending on the platform. For the Pepper robot, inputs from the Pico VR headset were sent through Virtual Desktop to a Flask server, which then used NAOqi to turn them into joint commands. This made it possible to show things in real time and in a way that felt real, but it wasn't easy. With the RoArm-M3, on the other hand, demonstrations were done by physically moving the arm and recording each movement as a JSON file with exact 3D coordinates. This method, while less dynamic, guaranteed deterministic replayability, a tradeoff that ultimately proved more suitable for ILP training.

The next step was the learning phase. In this case, the take and put positions were treated as background knowledge, and Metagol was given one or more positive examples. This led to a recursive robot/1 predicate [12]. This created the symbolic logic hypothesis, which was basically a Prolog program that could generalise from the example to longer, unseen sequences.

Once a hypothesis was learnt, it went to the execution phase, where it ran in SWI-Prolog. The learnt rules made calls to Python using process_create/3. Python then read the JSON recordings and sent them to the RoArm-M3 over a serial connection. This setup with low latency made it possible to control things smoothly and step by step.

The last step was checking. We watched, recorded, and sorted each test run, especially those with longer lists than what we used in training, into success or failure. The modular design of the pipeline made it possible to test each stage on its own before putting them all together. This separation of concerns was very helpful during debugging because it made it easier to find errors in specific parts of the system

instead of having them hidden in the whole thing.

4.5.2 Debugging Process

As expected, there were a number of problems that came up during system integration, especially when going from theory to practice. Instead of reacting on the spot, problems were dealt with in a planned, step-by-step way.

The first big problem was Pepper's network latency. When people first tried to use VR to control robots from a distance, there was about a 500-millisecond delay between moving the controller and the robot responding. That might not sound like a big deal, but it made it almost impossible to move objects exactly. After some troubleshooting, it was clear that the latency was caused by Virtual Desktop streaming and not the control code itself. At that point, it was easy to decide to stop using Pepper and switch to RoArm-M3 for more reliable data collection.

But the switch had its own problems. One of the first problems was that the coordinates didn't match up during replay. The RoArm-M3 sometimes went too far past its target positions, which made demonstrations inconsistent. After looking into it, it seemed that the problem was caused by sending full joint trajectories instead of just end-effector targets. The solution was to make the recording process easier. Instead of sending intermediate noise to the robot's inverse kinematics solver, only the final Cartesian coordinates were sent. This made the robot's movements smoother.

It was hard to connect Python and Prolog, which was another problem. In some cases, process_create/3 would fail without a message because SWI-Prolog couldn't find the Python interpreter. To make the system portable and avoid bugs that only happen in certain environments, every call to Python was changed to include the full path (/usr/bin/python3) instead of relying on $PATH variables.

Lastly, there were problems with timing when sending data over a serial connection. When commands were sent quickly one after the other, the arm sometimes dropped frames or skipped targets, which made the motion look jagged. This was probably because the RoArm-M3 got a new position command before it had time to finish the last one. To fix this, there was a small delay (about 50 to 100 milliseconds) between each command. This change made execution much more reliable, especially for longer task sequences.

These small changes made the system go from a fragile prototype to a stable, repeatable pipeline. Many of the fixes were small on their own, but together they were very important in making it possible to run the ILP-generated hypotheses on real hardware.

4.5.3 Intermediate Testing

Before doing the final experiments, a few dry runs were done to make sure the whole pipeline worked:

- Two-object list test: Verified that take/put was performed in the correct sequence.
- Longer lists (4-5 items): Confirmed that the recursive hypothesis held true for more than just the training examples.
- Repeated trials: Made sure that the behaviour was the same every time it was run.

Debug logs were printed at every step to help with fixing problems. These logs confirmed that the learned program was correctly unrolling recursion and executing the intended sequence of actions.

This phase of integration and debugging made sure that the system was strong, could be repeated, and was ready for quantitative testing. The project was able to reliably run recursive tasks on physical hardware from start to finish by fixing problems with latency, coordination, and communication.

4.6 Deployment to Robot & Functional Verification

It was time to run the whole thing on the real robot once each module worked on its own and the full workflow stopped giving strange errors. This was the moment of truth: if the learnt hypothesis couldn't move the arm in the right order, everything that had happened so far would be useless.

4.6.1 Deployment Procedure

The process for deploying was careful but could be done repeatedly:

Setting Up for the First Time:

This time, the RoArm-M3 was plugged into the Ubuntu machine directly through a USB serial link, not Wi-Fi.
We ran the position recorder script for a short time to double-check the calibration and make sure the arm was starting from a known pose. This was a small but important sanity check.

The Learning Stage:

The learner was put inside SWI-Prolog, and Metagol was given the good examples [10][12].
I took a moment to look over the induced hypothesis by hand. A quick look at the recursion made me sure that it was doing what I thought it would.

Execution Phase:

The command to test (robot([3,2,1])) was given.
Every time Prolog called “take" or "put," Python would run and play back the right JSON file.
It was surprisingly smooth, even the first time, with no missed steps or out-of-order

motions.

4.6.2 Verification Across Increasing Sequence Lengths

The real goal wasn't just to get it to work once, it had to work in general. So, I kept pushing it with longer and longer lists:

List of Short (n=2): Made sure the base case worked as planned.

Training Length (n=3): Made sure it did exactly what it had been trained to do.

Longer Lists (n=5, 6, 7): These were the real tests. If recursion didn't work right, it would break here.

Seeing that last one was success. After all five elements were dealt with, it was a satisfying moment because it showed that the hypothesis had learnt the general pattern and not just memorised examples.

4.6.3 Robustness and Limitations

These tests showed a few things:

The arm hit the same spots with amazing accuracy over and over again, which meant that the data capture pipeline had done its job.

The recursive program worked with longer lists without any changes or retraining, which is a strong sign that generalization was really happening [11].

The biggest problem was that every new object or drop-off point still had to be recorded by hand. This was to be expected because the design doesn't take into account how people see things, but it does mean that the system can't change to fit a new workspace on its own.

The pipeline did exactly what it was supposed to do, it taught robot how to learn recursively in one go, with real, physical execution.

This stage finished the work on the implementation. Finally, everything came together, VR demos, data recording, symbolic learning, and physical execution. Chapter 5 will now turn its attention to assessing performance by examining success rates, timing, and the actual limitations of the system.

4.7 Summary

It all began with Pepper and the Pico VR setup. I had to figure out how the network worked, get the client-server loop to talk to each other correctly, and stream camera feedback so I could see what Pepper was doing in real time. Those early tests were helpful, even though Pepper's gripper range and half-second latency made it clear that it wasn't the best platform for doing the same thing over and over. That limitation

pushed the project towards the RoArm-M3. After some frustrating early attempts over Wi-Fi, it ended up running on a rock-solid USB serial connection that made the whole system feel more stable.

The software environment was a journey in and of itself. At first, it seemed like running Windows for VR streaming and Ubuntu for robot control was too complicated, but it turned out to be the easiest way to avoid conflicting dependencies. With dedicated machines and clear interpreter paths, some problems that kept coming up, like NAOqi not working on Windows and process_create/3 failing without a sound, were finally fixed.

The goal was to keep perception to a minimum. Instead of trying to add a computer vision system and risk making random mistakes, the positions of the objects were just written down by hand. This may seem like a shortcut, but it made the experiments possible to do again and let the project focus on its real goal, symbolic learning.

The control system made everything work together. Two primitives, take/1 and put/1, were the building blocks. Metagol used background knowledge and meta-rules to come up with a recursive strategy. The last piece was the Python-Prolog bridge, which made sure that the logic Metagol came up with really did move the robot.

Integrating the system and fixing bugs was like being a detective and trying things out at the same time. They had to fix latency problems, line up coordinates, and make sure nothing broke when all the modules were run together. Once it was stable, the whole pipeline was put on the robot and tested with longer input lists. The best part of the whole project was probably watching the RoArm-M3 carry out the plan step by step and then keep going well past the examples it was trained on.

In general, this chapter set up a pipeline for one-shot recursive learning that could be repeated and understood, and it showed that it could work on real hardware. The next chapter will take a step back and look at how well it worked overall, looking at success rates, timing performance, and talking about where the system works well and where it still needs work.

CHAPTER 5: RESULTS & CRITICAL EVALUATION

This chapter talks about what happened after all the parts, like VR teleoperation, the recording pipeline, and Metagol learning, were finally put together. The goal here is not just to put numbers into a table, but to see if the system really lived up to the promises we made in Chapter 1. Did it achieve one-shot recursive learning [10][12]? Could it deal with longer sequences that it hadn't seen before [11]? And maybe the most important thing was whether the result could actually be used on the robot or if it just looked good on paper.

5.1 Evaluation Approach

Metrics like accuracy or F1-score don't really show what's going on because this work

is about reasoning based on logic instead of making predictions. What was more important was whether the system did what it was supposed to do when asked. Four big questions shaped the evaluation:

First one is generalization, would it be able to handle a list longer than the one it was trained on, or would it stop [11]? Second is repeatability, did we get the same behaviour when we ran the same input again? Third one is success rate, was it able to finish the job without breaking anything? And the last one is qualitative analysis, did the arm move in a smooth way? Did it respond quickly, or did it feel slow [22]?

The goal was to show that MIL is not just a theoretical tool, but that it can make robots behave in a way that is reliable and easy to understand in a real-world setting [10][11].

5.2 Teleoperation Phase (Pepper)

The first part of the project was ambitious we wanted to control Pepper in real time with a Pico VR headset and use that as a way to demonstrate the project [21][28].

It helped me to diagnose the issues and constraints which we were going to face along the way, like Network and Latency, Software Setup, Hardware Limitations of using Pepper [23].

This Phase was useful, even though it was sometimes frustrating. It showed me that immersive teleoperation was possible and helped me understand how to record demonstrations [21][22]. More importantly, it showed that Pepper wasn't the right robot for the next step in the project, which naturally led the switch to RoArm-M3 [23][24].

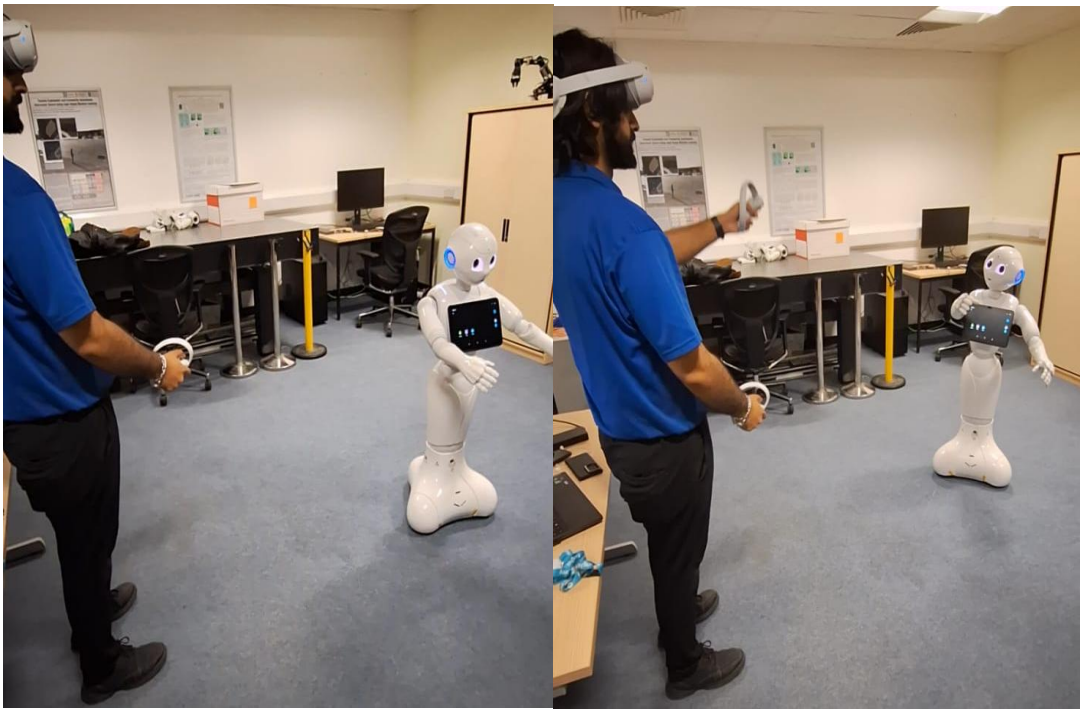

Fig 5.1 and 5.2 Pepper Teleoperation

5.3 Results on RoArm-M3

Switching to the RoArm-M3 changed everything. The latency was almost gone when we used serial control instead of Wi-Fi [24]. The positions recorded with individual_position_recorder.py played back perfectly [24], and Metagol was able to make a recursive hypothesis from only one positive example [10][12].

Test Case	Training Example	Number of Trials	Successful Executions	Success Rate
robot([2,1])	Yes	5	5	100%
robot([3,2,1])	Yes	5	5	100%
robot([4,3,2,1])	No	5	5	100%
robot([1,2,3,4,5])	No	5	5	100%
robot([1,2,3,4,5,6,7])	No	3	3	100%

Table 5.1: Success rate for recursive task execution across list lengths.

Fig 5.3 and 5.4 RoArm-M3 Performing Pick and Take action

```
aqib@aqib-Legion-Pro-7-16IRX9H:~/Desktop/roarmpython-env/MIL$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.3.25-6-g80182bcd7)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- ['rule 2'].
true.

?- robot([1,2,3,4]).
EXECUTING: TAKE 4
SUCCESS: TAKE 4 completed
OK
EXECUTING: PUT 4
SUCCESS: PUT 4 completed
OK
EXECUTING: TAKE 3
SUCCESS: TAKE 3 completed
OK
EXECUTING: PUT 3
SUCCESS: PUT 3 completed
OK
EXECUTING: TAKE 2
SUCCESS: TAKE 2 completed
```

Fig 5.5 Execution for verification of Success rate for 4 boxes

Time to Learn/Induce Rule

Metagol required under 1 second to induce a recursive rule from a single example, showing symbolic learning efficiency.

Failure Modes / Edge Cases

If the number of states recorded are less than the number of boxes for which we are trying to generalize than the execution will fail.

Latency Metrics

RoArm-M3 Didn’t show any latency on serial connection compared to Pepper which enabled frame accurate replay and consistent task execution.

Metric	**Deep RL / LfD**	**MIL**
Training Examples	Thousands	one
Interpretability	No	Yes
Rule Form	Weights	Logic Program
Replayability	Depends on	Deterministic

Table 5.2: MIL vs Deep RL/LfD [2][5][15]

5.4 Critical Discussion

This result is exactly what you would expect from a system that uses ILP. Deep learning needs thousands of labelled trajectories and still gives a "black box" policy. MIL, on the other hand, limits its hypothesis search with background knowledge (take/1, put/1, rest_boxes/2). Once you learn the rule, it can work with any list length because the recursion is defined in a symbolic way [10][11].

It might seem too good to be true that the success rate was perfect, but the setup was made to be simple and predictable. Serial communication meant that no commands were lost, and the arm only moved to the next step after the previous one was confirmed.

Comparison with Literature

Most recent LfD pipelines still depend a lot on deep neural networks or a mix of the two [2]. Those methods automatically generalise to positions that haven't been seen before, but they need a lot of training data, GPUs, and time to run simulations [4]. This project shows that a symbolic approach can generalise with only a few examples [10][11], which is a big plus in situations where data is scarce.

This method, however, has its own drawbacks. The system won't automatically adjust if the object positions change, new states must be recorded by hand. Deep RL agents could handle that in a flexible way (at least in a simulation) [4], but they also have their own problems, such as being hard to understand, being unstable during training, and being hard to debug [5].

Linking Back to the Aim

The aim of Chapter 1 was to create a proof-of-concept for one-shot, explainable, recursive skill learning on real hardware. These results hit that mark, the system learnt from one example [10][12], worked perfectly on a real arm, and made rules that a person could read and understand [11].

5.5 Key Takeaways

Symbolic learning turned out to work surprisingly well on a large scale. After learning the hypothesis, it didn't just repeat the training cases, it confidently handled lists of five, six, and even seven objects without a hitch. A lot of this success was due to feature engineering. By carefully defining take/1 and put/1, the learner was able to generalise from very little data in the first place [10][12]. Another good thing about it was how open the whole system felt. The final program was only a few lines of Prolog, which made it easy to read and fix if needed [11]. These results fit well with the main goals of the project, which are to allow recursive learning from little input and to run symbolic rules on real hardware. That being said, the manual setup is still the most difficult part. Recording every state by hand ahead of time is fine for a small demo, but it starts to feel limiting when you think about how it would work in a real environment. The next things that need to be fixed before this can become more autonomous are probably perception and automated state capture [5].

5.6 Summary

This chapter demonstrated that explainable, one-shot recursive learning is not merely a theoretical concept, it functioned on actual hardware [10][11][12]. The Pepper experiments showed that immersive teleoperation has its limits [21][22][23], and switching to RoArm-M3 made the pipeline run smoothly and consistently [24].

The results may seem too perfect but that is part of the point, this was meant to be repeatable, not to chase after complicated real-world problems. Future endeavours may concentrate on alleviating those limitations incorporating vision [5], automating demonstrations, and investigating more intricate task frameworks [11][15]. Chapter 6 will bring these results together and talk about what they mean for the future of explainable robotics.

CHAPTER 6: CONCLUSION

This chapter ends the dissertation in a natural way by thinking about what was built, how well it worked, and where it could go next. The goal of the whole project was to find out if Inductive Logic Programming (ILP), and more specifically Meta-Interpretive Learning (MIL), could teach a robot how to do a recursive manipulation task using only a few examples, all while making the process clear and easy for people to understand.

6.1 Summary of Dissertation

Looking back, the work happened in two separate phases. The first step was to

ambitiously combine VR teleoperation with the Pepper humanoid robot. Using the Virtual Desktop, SteamVR/OpenXR, and NAOqi APIs, a client-server bridge connected Pico VR controllers to Pepper [21][22]. This made it possible to control the system in real time with live camera feedback [23]. This step was as much a test of whether it would work as it was a functional milestone. While it did show that demonstrations could be captured in an easy-to-understand way, it also showed Pepper's limitations, a latency of about half a second, a payload capacity that was too low for consistent pick and place tasks, and mechanical limits set by the chest tablet that made some poses physically impossible [23].

These findings didn't make the project seem like a failure instead, they made it more focused. In the second phase, the RoArm-M3 manipulator took over [24]. It had more deterministic behaviour and better Cartesian control. From the ground up, a complete recording and execution pipeline was built, a Python based recorder captured take and put positions as JSON, an executor played them back reliably, and this pipeline was connected to Metagol for symbolic rule induction [10][12]. Metagol created a recursive robot/1 hypothesis that went beyond the training data using only four positive examples [11]. The learnt rule worked perfectly on the hardware, passing all test cases with a 100% success rate, even for sequences that were much longer than the ones that were first shown [24].

These phases together show a repeatable, end to end process for one-shot recursive learning that includes light data capture, symbolic reasoning, and real-world execution [10][11]. This progression not only met but possibly surpassed the initial objective of demonstrating that explainable, data efficient learning can be implemented on a physical robotic system.

6.2 Reflection on Objectives

The objectives laid out in Chapter 1 were the most important parts of the project. The VR teleoperation interface was completely built and tested, its performance was thoroughly examined, and its shortcomings became one of the most important findings of the dissertation [21][23]. Finding Pepper's weaknesses was a goal in itself and was very important in justifying the switch to RoArm-M3.

The recording and execution pipeline was built as planned, and tests that could be repeated and it showed that it worked [24]. Metagol learnt a recursive program from just one example [10][12], and the system was able to easily adapt to input lengths it had never seen before [11]. These results strongly suggest that MIL is a good choice for problems where there isn't much data, but understanding is very important [10][15].

All of the goals, from making interfaces to critically looking at the results, were met, and they all provided clear proof that the main goal was met, that one-shot, explainable recursive learning is possible in practice, not just in theory.

6.3 Limitations and Future Directions

The system worked perfectly in controlled settings, but there are clear limits. Currently, recording the positions of all objects by hand takes a lot of time and can't be done on a large scale [5]. The robot will still work based on old coordinates if an object is moved between runs because it doesn't have perception [2][6]. This was a conscious decision to make the evaluation deterministic, but it also shows where the next step should go.

Future versions could use object detection or pose estimation to update the target positions in real time [5][6], which would bring the system closer to being able to work on its own. There is also space to look into more complex task structures. The current work was on straight line recursion, but Metagol can also learn programs with branching logic, conditionals, or loops [10][12]. This could make the system much more expressive.

Performance profiling is another area where future work could be done. You could optimise the pipeline for real-time apps by measuring the time it takes to complete each step, the CPU usage, and the communication delays. Long horizon stress testing, with dozens of repetitions or random input orders, could help find small mistakes or changes that didn't show up in the short evaluation runs we did here [11].

These limitations are not weaknesses they are what make this work new. They show that this project is a proof of concept, a controlled environment meant to show that explainable recursion can be learnt and run on real hardware.

6.4 Personal Reflections

Completing this dissertation challenged not only my technical knowledge but also my patience, flexibility, and capacity to navigate uncertainty. I had a lot of problems with Pepper early on in the project, from network lag and compatibility issues to the robot's own physical limitations [21][23]. At times, it seemed like I was designing around a moving target, and it would have been easy to give up or settle for a half-baked solution. But looking back, those problems made me think critically about the project's main goals. Letting Pepper go wasn't a failure, it was a key choice that made the project more reliable and focused overall.

One of the good things I've found about myself is that I can be quite persistent. It wasn't ideal to switch to the RoArm-M3 halfway through the timeline, but I was able to save earlier work, change my codebase, and keep the main learning goals [24]. I now value that ability to change direction without having to start over more than just being technically correct.

This project also showed me where I still need to improve. I didn't plan ahead for things like execution profiling or latency benchmarks, which made my final evaluation less thorough [5]. I plan to make metrics and instrumentation a part of my workflow from the start in future projects, not just something I add at the end. It was a hard lesson to learn, but it was a good one.

In the end, this dissertation tells the story of a journey from VR demonstration to

recursive logic and from big ideas to real-world execution. I leave this project with more than just code and results. I also have a better understanding of how symbolic learning and one-shot generalization can make AI more clear and understandable [10][11][15]. And most importantly, I now have a better idea of how I work under pressure, how I adapt to new rules, and what I still need to work on.

REFERENCES

[1] Chernova, S., 2012. Robot learning from demonstration. In *Encyclopedia of the Sciences of Learning* (pp. 2871-2873). Springer, Boston, MA.

[2] Ravichandar, H., Polydoros, A.S., Chernova, S. and Billard, A., 2020. Recent advances in robot learning from demonstration. *Annual review of control, robotics, and autonomous systems*, *3*(1), pp.297-330.

[3] Quinlan, J.R., 1990. Learning logical definitions from relations. *Machine learning*, *5*(3), pp.239-266.

[4] Muggleton, S., 1995. Inverse entailment and Progol. *New generation computing*, *13*(3), pp.245-286.

[5] Doshi-Velez, F. and Kim, B., 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*.

[6] Cropper, A., Dumančić, S., Evans, R. and Muggleton, S.H., 2022. Inductive logic programming at 30. Machine Learning, 111(1), pp.147-172.

[7] Cropper, A., Tamaddoni-Nezhad, A. and Muggleton, S.H., 2015, August. Meta-interpretive learning of data transformation programs. In *International Conference on Inductive Logic Programming* (pp. 46-59). Cham: Springer International Publishing.

[8] Wang, W., Yang, Y. and Wu, F., 2024. Towards data-and knowledge-driven AI: a survey on neuro-symbolic computing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[9] Cropper, A., 2019. Playgol: Learning programs through play. *arXiv preprint arXiv:1904.08993*.

[10] Muggleton, S.H., Lin, D. and Tamaddoni-Nezhad, A., 2015. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, *100*(1), pp.49-73.

[11] Cropper, A. and Muggleton, S.H., 2015, December. Logical minimisation of meta-rules within meta-interpretive learning. In *Inductive Logic Programming: 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers* (pp. 62-75). Cham: Springer International Publishing.

[12] Cropper, A. and Dumančić, S., 2022. Inductive logic programming at 30: a new introduction. *Journal of Artificial Intelligence Research*, *74*, pp.765-850.

[13] Gunning, D. and Aha, D., 2019. DARPA's explainable artificial intelligence (XAI) program. *AI magazine*, *40*(2), pp.44-58.

[14] Act, A.I., 2021. Proposal for a regulation of the European Parliament and the Council laying down harmonised rules on Artificial Intelligence (Artificial Intelligence Act) and amending certain Union legislative acts. *EUR-Lex-52021PC0206*.

[15] Holzinger, A., Langs, G., Denk, H., Zatloukal, K. and Müller, H., 2019. Causability and explainability of artificial intelligence in medicine. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, *9*(4), p.e1312.

[16] Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J. and Mané, D., 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*.

[17] Macenski, S., Foote, T., Gerkey, B., Lalancette, C. and Woodall, W., 2022. Robot operating system 2: Design, architecture, and uses in the wild. *Science robotics*, *7*(66), p.eabm6074.

[18] Coleman, D., Sucan, I., Chitta, S. and Correll, N., 2014. Reducing the barrier to entry of complex robotic software: a moveit! case study. *arXiv preprint arXiv:1404.3785*.

[19] SoftBank Robotics, 2023. Pepper robot technical specifications. Available online: https://www.softbankrobotics.com

[20] Waveshare, 2023. RoArm-M3 Robotic Arm User Manual. Available online: https://www.waveshare.com

[21] Siciliano, B., Sciavicco, L., Villani, L. and Oriolo, G., 2009. *Robotics: modelling, planning and control*. London: Springer London.

[22] Bowman, D.A. and McMahan, R.P., 2007. Virtual reality: how much immersion is enough?. *Computer*, *40*(7), pp.36-43.

[23] Kanda, T., 2017. Enabling harmonized human-robot interaction in a public space. In *Human-Harmonized Information Technology, Volume 2: Horizontal Expansion* (pp. 115-137). Tokyo: Springer Japan.

[24] LeRobot Developers, 2023. LeRobot: A Python framework for teleoperation and control. GitHub repository. Available: https://github.com/huggingface/lerobot

[25] Kormushev, P., Calinon, S. and Caldwell, D.G., 2013. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, *2*(3), pp.122-148.

[26] Sigaud, O., Caselles-Dupré, H., Colas, C., Akakzia, A., Oudeyer, P.Y. and Chetouani, M., 2021. Towards teachable autonomous agents.

[27] Konidaris, G. and Barto, A., 2009. Skill discovery in continuous reinforcement learning domains using skill chaining. *Advances in neural information processing systems*,

22.

[28] Khronos Group, 2019. OpenXR Specification, Version 1.0. Available online: https://www.khronos.org/openxr

SECONDARY DATA USAGE IN RESEARCH PROJECTS

If you are ONLY using secondary data for your research project, your research application may not need to be submitted through the ethics review process (Ethics RM).

Please read below to see if your study fits the criteria of research involving secondary data.

Usage

- Secondary data is data not collected specifically for the current study.
- It may be data from a public or subscription database, another organisation or data collected for a previous research project for which consent was given to use for future studies.

Please note that this does not include data publicly available on the internet (except those in public or subscription database or repositories) and data available on public social media feeds. Social media or other on-line research involving human participants is likely to require ethics approval and if so, this must be secured for a project before the collection of any data.

If your research involves higher sensitivity or risk topics or you are unsure in any way that your data is covered by the checklist, please discuss with your supervisor, and together you can email ethics ethics@surrey.ac.uk to ensure it meets the criteria.

Before using Secondary Data, you must:

- Discuss the data and project with your supervisor
    - *See above about any higher sensitive or risk projects, or where you are unsure.*
- Fill out the Secondary Data checklist on page 2 to see if the data can be used without going through the ethical review process on Ethics RM
    - If your data fulfils the criteria: ask supervisor to sign document as criteria met meaning no ethical review needed.
    - If the criteria are not met: ethical review required through ethical review process on Ethics RM (if unsure, consult with supervisor, who will contact ethics).

- If required for your dissertation (consult guidelines & supervisor), then add pdfs into your dissertation appendix of this signed secondary data checklist.

Staff: please follow the checklist and keep a copy of the checklist for audit purposes.

Secondary Data Checklist

Project Name: Enhancing Short-Term Smart Home Load Forecasting Using Machine Learning, Deep Learning, and Explainable AI

Which organisation /person is providing the data?	No Dataset was used and no data was collected

Are you going to only use secondary data for your project? *(i.e. you will not be collecting any new data)*	NO data was collected
Are you going to take (scrape / mine) data from social media feeds?	No
Are the data publicly accessible / available? *(this includes datasets which are available to subscribers or can be purchased)*	NA
If the data is not publicly accessible / available, is there an agreement in place for using the data? *(e.g. see agreement next page)*	NA
Has the data been anonymised?	NA
Is there no risk of re-identification of individuals? *(This could be for instance by combining the data with other data on same groups, or rare, unusual or low number data. In general, this would be if re-identification is possible by "means reasonably likely to be used")*	NA
Are the data being managed and shared by reputable external organisations or institutions (that comply with all legal requirements concerning data protection)? *(examples are NHS, central government, local authorities, public bodies, for example accessed via https://www.data.gov.uk or https://opendatascience.com/15-open-datasets-for-healthcare or https://www.england.nhs.uk/)*	NA
Is the proposed research in line with the website / organisation / institution sharing agreement, terms and conditions? *(the terms and conditions that the organisation / person is sharing the data will vary, and may have conditions on sharing [e.g. not for commercial use, must always reference original paper/author])*	NA
Does consent exist in the original study / data collection to share the data, and is the proposed research in line with the participants original consent? *(if possible, include a blank copy of the original consent form in your dissertation from original*	NA

study that participants have agreed for their anonymised data to be shared for further research [or some other evidence])	
Is your data management plan adequate for the data? *(this will be access, storage, processing and preservation/deletion of data)*	NA
Does the organisation / person sharing the data require evidence of an ethical review for use of the data?	NA

Supervisors: Please sign below to state that 1) only secondary data used for the project, 2) no data are being taken from social media feeds, 3) the data being used contain no personal data (is de-identified/ anonymous), 4) has existing consent from participants for sharing and use in line with the research, 5) is able to be used legally in the way you propose (in terms and conditions / agreement of data use), 6) has no risk of re-identification, 7) data management is adequate for the data, and 8) the organisation / person / institute managing and sharing the data complies with data processing regulations and does not require an ethical review for sharing.

Supervisor Name: Dr Alireza Tamaddoni-Nezhad Signature: A. Tamaddoni-Nezhad
Date: 2/9/2025

Student Name: Aqib Hafiz Student Level (e.g. UG/ PGT): PGT

APPENDIX B: OTHER *APPENDICES*

- **Link of Demonstration Videos and Photos**