UNIVERSITY OF TARTU

Faculty of Science and Technology
Institute of Technology
Robotics and Computer Engineering
curriculum

Natural Language Human-Robot Interaction: A Modular Framework for Conversational Robot Control using Large Language Models

Master's Thesis (30 ECTS)

Julian R. Leclerc

Supervised by:

Robert Valner, PhD and Karl Kruusamäe, PhD

2025

Natural Language Human-Robot Interaction: A Modular Framework for Conversational Robot Control using Large Language Models

Abstract:

As robots are increasingly used in everyday situations beyond industrial environments, traditional controller-based interfaces that require specialized training pose significant barriers to widespread adoption. This thesis introduces a modular natural language human-robot interaction system designed for conversational robot control. It integrates large language models with robust robotic frameworks to enhance usability. The system tackles key challenges, such as the reliability of large language models, contextual awareness, and bridging the semantic gap. It includes specialized components for error handling, memory management, and cross-platform deployment through a web interface. Validated on various platforms, including TIAGo and Boston Dynamics Spot, the system demonstrates platform-agnostic functionality and accessibility for non-technical users. This advancement helps democratize robot control through natural language while ensuring operational reliability.

Keywords:

Human-Robot Interaction (HRI), Large Language Models (LLMs), Natural Language Processing, Robot Operating System (ROS2), TeMoto Framework, Conversational Interfaces, Unified Meaning Representation Format (UMRF), Error Handling, Modular Architecture, Platform-Agnostic Design

CERCS:

P170 - Computer science, numerical analysis, systems, control; T125 - Automation, robotics, control engineering

Loomuliku keele inimese-roboti suhtlus: Modulaarne raamistik vestluspõhiseks roboti juhtimiseks kasutades suuri keelemudeleid

Resümee:

Kuna roboteid kasutatakse üha enam igapäevastes olukordades väljaspool tööstuskeskkondi, kujutavad traditsioonilised kontrolleripõhised liidesed, mis nõuavad spetsiaalset väljaõpet, märkimisväärseid takistusi laialdasele kasutuselevõtule. Käesolev väitekiri tutvustab modulaarset loomuliku keele inimese-roboti suhtlussüsteemi, mis on loodud vestluspõhiseks roboti juhtimiseks. See integreerib suured keelemudelid töökindlate robotiraamistike abil kasutusmugavuse suurendamiseks. Süsteem tegeleb oluliste väljakutsetega, nagu suurte keelemudelite usaldusväärsus, kontekstiteadlikkus ja semantilise lõhe ületamine. See sisaldab spetsiaalseid komponente vigade käsitlemiseks, mälu haldamiseks ja platvormideüleseks juurutamiseks veebiliidese kaudu. Süsteemi on valideeritud erinevatel platvormidel, sealhulgas TIAGo ja Boston Dynamics Spot, demonstreerides platvormist sõltumatut funktsionaalsust ja ligipääsetavust mittetehnilistele kasutajatele. See edasiminek aitab demokratiseerida roboti juhtimist loomuliku keele kaudu, tagades samal ajal töökindluse.

Võtmesõnad:

Inimese-roboti suhtlus (HRI), suured keelemudelid (LLM), loomuliku keele töötlemine, robotite operatsioonisüsteem (ROS2), TeMoto raamistik, vestlusliidrsed, ühtne tähenduse esitusformaat (UMRF), vigade käsitlemine, modulaarne arhitektuur, platvormist sõltumatu disain

CERCS:

P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine; T125 - Automatiseerimine, robootika, juhtimistehnika

Contents

List of Figures

Nomenclature

API Application Programming Interface

BT Behavior Tree

GUI Graphical User Interface

HRI Human-Robot Interaction

JSON JavaScript Object Notation

LAN Local Area Network

LLM Large Language Model

Nav2 Navigation 2

NL Natural Language

QR Quick Response

ROS Robot Operating System

ROS2 Robot Operating System 2

SLAM Simultaneous Localization and Mapping

TeMoto Task Execution and MOnitoring TOolkit

UMRF Unified Meaning Representation Format

XML Extensible Markup Language

YOLO You Only Look Once

Part I

Introduction

1.1 Motivation

1.1.1 Context

In robotics, despite significant advances in autonomous capabilities, current technologies still face fundamental challenges in environmental perception, contextual reasoning, and adaptive decision-making that prevent the development of fully autonomous robots. To address these shortcomings, human involvement is often necessary, which means that the effectiveness of a system largely depends on the quality of human-robot interaction (HRI) [1]. Historically, these HRI systems have primarily been designed for expert operators with extensive training, relying on manual control interfaces such as joysticks and button-based systems, each mapped to specific robot functions [2]. Because these HRI systems were primarily designed for expert use, their development has traditionally focused on practical usability rather than accessibility [3].

The integration of robots has expanded beyond rigid industrial settings into various sectors, including healthcare, manufacturing, disaster response, and customer service. In these fields, robots help mitigate risks for humans, enhance efficiency, and provide assistance in understaffed areas. For instance, in healthcare, elderly individuals rely on intuitive human-robot interaction systems to communicate with robot assistants [4]. In manufacturing, factory workers increasingly collaborate with collaborative robots [5]. During disaster response efforts, emergency personnel deploy mobile robots to navigate hazardous environments [6], while in customer service, robotic assistants offer relevant information to customers [7]. This rapid expansion into diverse sectors has created a need for intuitive and universally operable HRI systems that can be utilized by individuals with varying levels of technical expertise [8].

To enhance the accessibility of Human-Robot Interaction systems across various sectors, researchers have implemented several modifications to traditional controllers. These improvements encompass adopting ergonomic button designs, clearer labeling, and optimized layout organization [3]. Nevertheless, it is important to recognize that even with these improvements, for complex robotic systems, using controller-based interaction still requires a significant amount of time for users to operate effectively [5].The emergence of Large Language Models (LLMs) has created new possibilities for interacting with robotic systems through natural language [8, 7], potentially eliminating the need for specialized training and enabling intuitive communication between humans and robots.

1.1.2 Problem Statement

While natural language offers a promising approach to human-robot interaction, developing an effective HRI system based on large language models presents several critical challenges that cannot be solved through simple LLM-to-motor command connections. Current LLM technology, while impressive in its language understanding abilities, does not possess the robustness, reliability, or physical grounding needed for direct control of robotic systems in real-world environments. Although these models excel at understanding semantics, they are not equipped to manage the precise requirements, physical constraints, and error recovery mechanisms that are essential for ensuring safe robot operation. This gap can be addressed by developing hybrid systems that utilize large language models for understanding natural language, while also

integrating them with established robotic frameworks for management and control of execution.

A closer examination of these hybrid HRI systems reveals fundamental challenges throughout the robot control pipeline. While controller-based interactions require operators to actively manage the perception-decision-action cycle [6], LLM-based approaches shift these cognitive responsibilities to the robot itself. This transfer introduces significant technical hurdles: the system must interpret natural language commands that may contain ambiguities or contextual references, convert these abstract instructions into sequences of executable robot actions, and handle execution uncertainties without requiring technical expertise from users [8, 6, 5]. To achieve real-world applicability, these systems must balance intuitive accessibility for novice operators with the operational robustness traditionally reserved for expert-driven interfaces [3]. Successfully addressing these interconnected challenges necessitates a modular architecture that effectively bridges the semantic gap between human communication and robotic execution while maintaining consistent reliability across diverse tasks and environmental conditions.

1.2 Approach

1.2.1 Objectives

This thesis aims to develop and validate a natural language-based HRI system that addresses the identified problems through the following specific objectives:

1. Design a modular architecture that effectively bridges the semantic gap between conversational human commands and precise robot actions across multiple platforms.
2. Establish context-aware interactive capabilities that facilitate intuitive task planning and modification through natural dialogue, ensuring operational continuity across multiple commands.
3. Create an adaptable error-handling framework that improves operational reliability by identifying ambiguities and resolving execution failures with minimal technical expertise required from operators.
4. Develop an intuitive interface system that enables users with diverse technical backgrounds to easily monitor and direct robot operations through natural language.
5. Assess the system's effectiveness through demonstrations on various robotic platforms and application scenarios, confirming its viability as a platform-agnostic solution.

1.2.2 Structure

This work presents a comprehensive Natural Language Human-Robot Interaction system that enables complete control of robots through natural language interaction. The solution integrates a web-based interface for accessibility with a human-robot interaction ROS2 package embedded within the TeMoto framework — a robust architecture designed to address fundamental challenges in system reliability, human-robot collaboration, and multi-robot coordination.

The study begins by critically examining current human-robot interaction technologies and analyzing existing LLM-based robot control systems to establish a solid foundation for the proposed approach. It then outlines the specific technical and functional requirements that guided the development of the system and explores the architecture in detail, focusing on three essential components:

- A **Web Application** that facilitates real-time natural language communication, visual feedback, and operation monitoring through an intuitive interface accessible across multiple devices.
- A **Natural Language Interface Package** that processes conversational commands, maintains contextual awareness, generates structured action sequences, and autonomously handles execution errors through specialized ROS2 nodes.
- A **TeMoto Action Engine Package** that executes the planned operations through a modular framework that abstracts hardware-specific implementations, enabling platform-agnostic operation across various robotic systems

Experimental validations were conducted, demonstrating the system's effectiveness across simulated environments (Turtlebot3) and physical platforms (Pal Robotics' TIAGo and Boston Dynamics' Spot). The concluding chapters discuss the system capabilities, technical limitations, research contributions, and future development directions.

Throughout this thesis, the focus is on the system's innovative method for bridging the semantic gap between unclear natural language instructions and exact robot actions, while also ensuring an accessible interface for users lacking specialized training in robotics or programming.

All software components developed for this project are publicly available through GitHub repositories in the Appendix. These repositories include the complete NL HRI system, individual packages for the TeMoto Action Engine, Web Application, NL Interface Package, and demonstration resources. Furthermore, a dedicated repository features clearer images and demonstration videos of the system in operation, offering more precise visual documentation of the experimental results presented in this thesis.

Part II

Background

2.1 Preliminaries

This section outlines the foundational technologies underpinning our natural language-based robot control system. We examine Robot Operating System 2 (ROS2), which provides the middleware infrastructure for robotic communications, and the TeMoto framework, which offers a robust and modular environment for executing action sequences using the Unified Meaning Representation Format (UMRF). These essential components provide the technical context for the system architecture, which will be elaborated on in subsequent chapters.

2.1.1 ROS2 Framework

Robot Operating System 2 (ROS2) provides the middleware infrastructure that underpins the communication architecture of this work. As the evolution of ROS1, it offers enhanced capabilities in reliability, security, and real-time operations critical for modern robotics applications.

ROS2's key features relevant to our work include:

- **Decentralized Communication**: A peer-to-peer architecture that eliminates single points of failure, facilitating robust multi-robot operations.
- **Flexible Communication Patterns**: Support for publish-subscribe topics, request-response services, long-running actions with feedback, and runtime-configurable parameters.
- **Modular Structure**: Package-based organization that enables code reuse and clear separation of concerns, essential for our integration of natural language processing with robotic control systems.
- **Concurrent Processing Architecture**: Support for parallel execution through MultiThreadedExecutor and ReentrantCallbackGroup mechanisms, enabling non-blocking communication patterns for multi-robot operations.

These capabilities provide the foundation upon which our natural language command processing and robotic action execution are built, enabling seamless communication between the linguistic and physical domains of our system.

2.1.2 TeMoto and UMRF: A Modular Framework for HRI

2.1.2.1 TeMoto's framework for modular and scalable HRI Systems

While LLM-driven systems enhance accessibility, they must be structured within modular, scalable frameworks to facilitate integration into diverse robotic applications. **TeMoto** is a ROS-based software framework that addresses key HRI challenges, including system reliability, multi-robot collaboration, and task execution flexibility [9]. Through the dynamic resource management capabilities, TeMoto enables robots to allocate computational and physical resources based on task requirements, significantly improving fault tolerance and system adaptability.

Figure 1 [9] illustrates TeMoto's layered architecture, which separates system functionality pipeline into several key components:

1. **Command Sources**: The entry point of the framework, providing the necessary task descriptions used by TeMoto. These include task planners where operators can select pre-configured task sequences, and voice interfaces that leverage systems like Alexa to generate one-shot graphs or task generation through non-verbal gesture interpretations [10, 11].
2. **Task Descriptions**: Establishes the action sequences to be followed during robotic operations. Within TeMoto, these task descriptions are defined via UMRF (Unified Meaning Representation Format).
3. **Task Management**: Executes action sequences provided by the Task Descriptions step by step, simultaneously allowing for multi-robot operation through the TeMoto Action Engine.
4. **Resource Management**: Oversees the allocation, monitoring, and failure handling of hardware and software components, ensuring optimized utilization of sensors, actuators, and computation modules.
5. **Components**: Consists of the fundamental software and hardware elements that interact with the physical world, including perception, control, and communication subsystems.

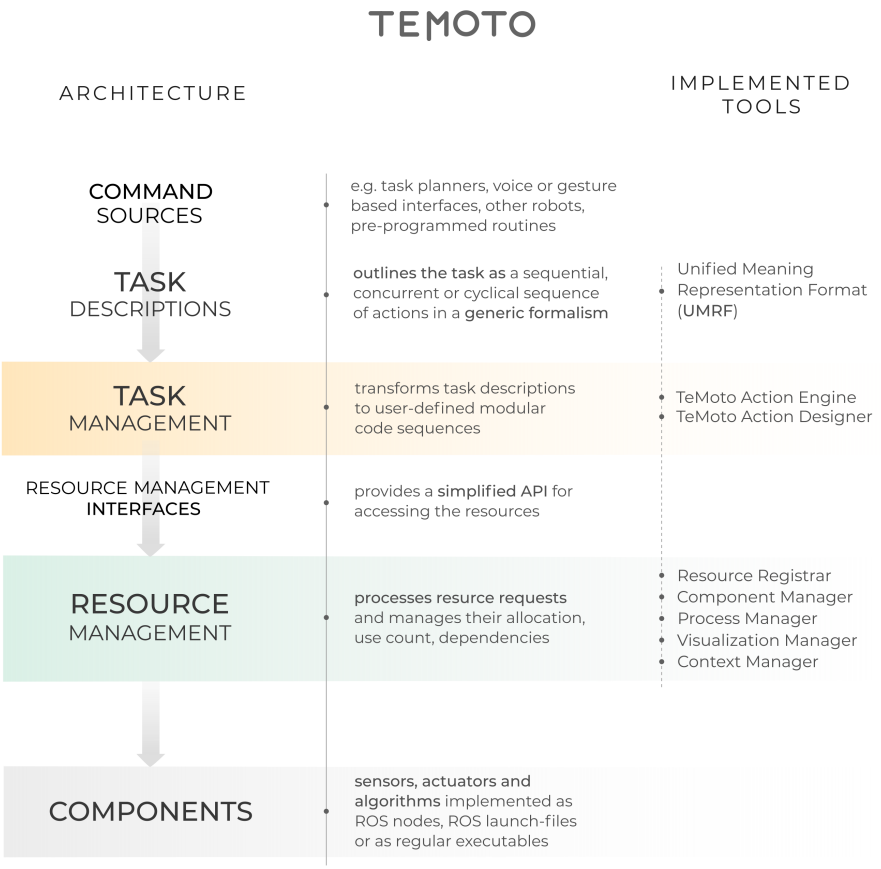


Figure 1: Temoto Framework Architecture

2.1.2.2 UMRF Graphs

For the Task Descriptions generated by the Command Sources, TeMoto employs the Unified Meaning Representation Format (UMRF)[1], a formalism crafted for task execution that systematically organizes and represents robot actions in a structured sequence. This structure is founded on principles that emphasize:

- **Ease of adoption**: UMRFs syntax is described in JavaScript Object Notation (JSON), a widely recognized data format supported by all major programming languages.
- **Structural expressiveness**: UMRFs enable the expression of tasks as actions that can be executed sequentially, concurrently, or cyclically.
- **Reusability**: UMRFs support parameterization, meaning actions can accept and output parameter data. For example, a navigation action can accept a navigation goal and output the final location.
- **Enhanced human-robot interaction**: UMRFs are grounded in linguistic theories of meaning representation capturing the semantic meaning of commands in a format that computers can understand.

In the context of the natural language HRI system, the UMRF structure serves as an essential link between natural language commands and structured action execution sequences that robots can perform. It provides a systematic format for representation that captures the semantic depth of natural language while ensuring the precision needed for robotic execution.

2.1.2.3 TeMoto's Action Engine

For Natural Language HRI applications, the Action Engine delivers the execution capability that standalone language models cannot provide. Although large language models excel at interpreting intent and generating overarching plans, they often fall short regarding the robustness required for direct robot control. The Action Engine effectively addresses this gap by facilitating a structured and controlled execution process, enhancing the reliability of natural language-driven robotics. As the Task Management layer of the TeMoto Framework, it manages robot control by interpreting the UMRF graphs and sequentially executing the defined actions, as illustrated in Figure 2 [1]. The actions declared within these graphs are implemented as reusable C++ code modules maintained as shared libraries, each encapsulating developer-defined task logic with specific input and output parameters.

Furthermore, this Task Management system allows for multi-robot collaboration where each robot within the network is associated with a distinct Action Engine responsible for executing the processed UMRF sequences. Each Action Engine is designated with a unique "actor" name to ensure clear differentiation, and they actively monitor UMRF events directed at their specific actor.

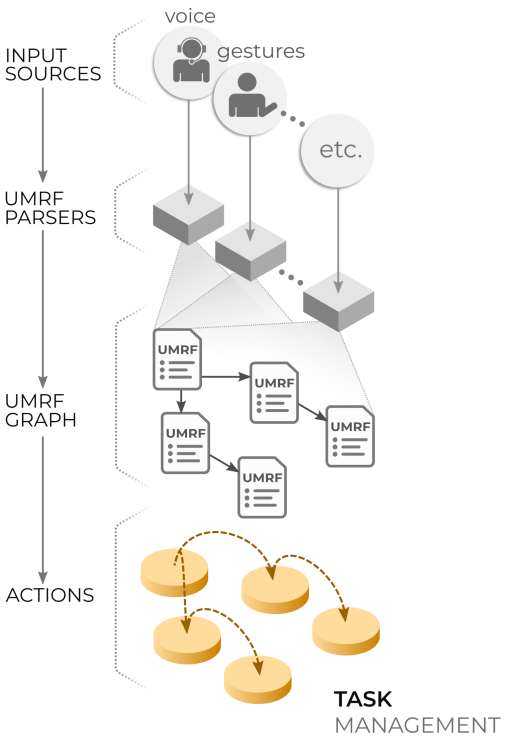


Figure 2: Task Management pipeline in TeMoto

2.2 State of the Art

As robotic systems increasingly proliferate beyond controlled industrial environments into domains such as healthcare assistance for elderly patients[4], collaborative manufacturing workspaces[5], high-stakes disaster response scenarios[6], and customer service applications[7], the integration of LLMs into the HRI frameworks has proven to be a key resource to enable effective human-robot collaboration across diverse application domains.

This section provides a thorough review of current human-robot interaction technologies and examines solutions leveraging LLM integration in HRI. This foundational analysis serves to contextualize the proposed solution within the wider academic context and highlights its potential contributions to advancing the field.

2.2.1 Human-Robot Interaction Solutions

HRI systems have employed explicit manual control interfaces, such as joysticks, keyboards, buttons, and graphical user interfaces, that impose considerable cognitive demands onto the operator [6]. These interfaces utilize direct mapping mechanisms where the specific controller inputs correlate with individual robot functions. Industrial applications exemplify this approach, as evidenced by Fanuc's Teach Pendant illustrated in Figure 3a [12], where operators must manipulate the robot joint movements through a series of reconfigured controls thus necessitating not only knowledge on how the pendant work but process advanced spatial reasoning abilities [13]. Similarly, in disaster response scenarios, this cognitive burden intensifies further, as demonstrated by iRobot's Packbot (Figure 3b [14]), where in addition to having to control the mobility of the robot, the robot arm attached and the end effector, the operator must also rely on mounted on cameras for visual feedback, thereby generating substantial mental workloads during time-sensitive operations [6].

(a) Teach Pendant used to control Fanuc's industrial robots

(b) iRobot's Packbot alongside its controller

Figure 3: Examples of traditional HRI manual control interfaces.

A fundamental limitation of traditional HRI interfaces is their reliance on low-level command structures that require operators to translate their intentions into discrete control inputs manually and must simultaneously manage four critical cognitive tasks: environmental perception, situation interpretation, action planning, and execution monitoring. This approach results in steep learning curves, with courses requiring novice users between 25 to 40 hours of training to develop basic proficiency with industrial robot controllers (ABB, FANUC). While experts eventually internalize control mappings and develop automated cognitive schemas, this extensive training period creates a significant barrier for widespread adoption by professionals across industries such as healthcare, manufacturing, disaster response, and customer service. [8]

To reduce the users' cognitive demands, research has investigated multimodal interaction methods, such as speech, gesture, gaze, and haptic feedback [3]. However, without an adaptive framework, these systems often require users to conform to predefined commands and structured inputs, going against their built intent and limiting their intuitiveness [15]. For instance, early gesture-based systems such as those integrated with industrial robots like Baxter by Rethink Robotics, relied on fixed motion patterns, making it difficult for users to interact naturally[16].

2.2.2 LLM-Based Human-Robot Interaction Solutions

Recent advancements in Large Language Models have opened new possibilities for controlling robots through natural language commands. Several research teams have developed frameworks that translate human instructions into executable robot actions, addressing various aspects of this challenge.

2.2.2.1 LLM-Based Task Planning and Execution

Wang et al. proposed a constrained LLM prompt scheme with an exceptional handling module to generate executable action sequences from natural language commands. Their approach defines eight primitive actions (move_to, look_for_obj, look_for_person, follow, grasp, pass_to, speak, and answer) and uses few-shot learning examples to guide the LLM in generating task plans in a specific format. The system achieved an 83% success rate in correctly decomposing tasks into primitive functions, though only 69% could be completely executed with their

implementation. A key contribution was their exception handling module that addresses hallucinations by detecting and correcting problematic outputs, particularly when responses deviate from the specified format or include actions beyond the provided primitive set [17].

Similarly, Lykov and Tsetserukou introduced LLM-BRAIn, a system that generates robot behavior trees (BTs) from natural language commands. Their approach uses a fine-tuned LLM (Stanford Alpaca 7B) to convert text descriptions into executable robot behavior trees in XML format. The system consists of three main components: a nodes library containing available actions and conditions, the LLM-BRAIn processor that generates BTs, and a ROS2 BT interpreter that executes the generated plans. User studies showed that human experts could not reliably distinguish between LLM-generated and human-written behavior trees, suggesting the high quality of the generated behaviors [18].

2.2.2.2 Multi-Agent Robot Task Planning

Kannan et al. developed SMART-LLM, a framework for multi-robot task planning that addresses the challenge of coordinating heterogeneous robots with different capabilities. Their system uses a structured four-stage process: task decomposition, coalition formation, task allocation, and task execution. The framework leverages LLMs to analyze both the environment (objects and their properties) and available robot skills to form appropriate teams for complex tasks. When tested with multiple LLM backends (GPT-4, GPT-3.5, Llama2-70B, and Claude-3-Opus), SMART-LLM achieved 70-100% success rates across different task categories, with GPT-4 showing the best overall performance. The system demonstrated the ability to handle previously unseen tasks in real-world experiments, though performance varied based on task complexity [19].

2.2.2.3 Humanoid Control and Imitation Learning

Sun et al. presented a framework combining adversarial imitation learning with LLMs for humanoid robot control. Their system enables robots to learn reusable skills through a single policy network and solve zero-shot tasks under LLM guidance. The approach uses codebook-based vector quantization to handle unseen textual commands and incorporates specialized reward functions for humanoid robots. The system consists of three components: an adaptive language-based skill motion policy using Generative Adversarial Imitation Learning, a CLIP-based adaptive language discrete encoder, and an LLM-based motion planner. This integration allows for complex task planning, such as navigating around obstacles, with high robustness to unseen rephrased commands (up to 91% accuracy for certain motion types) [20].

2.2.2.4 End-User Development Systems

Karli et al. developed Alchemist, an integrated development system leveraging LLMs to enable end-users with minimal programming knowledge to create robot applications. The system integrates visualization, programming, and execution in a single interface, allowing users to program robots through conversation with an LLM. It includes a 3D visualization panel, chat interface, terminal, and optional code editing capabilities. Alchemist incorporates a function library with two levels of abstraction (high-level task-oriented functions and more flexible low-level functions) to accommodate different user expertise levels. An exploratory study with 10 participants found that both novices and experts completed tasks in similar time frames but with different approaches—novices preferred debugging by prompting the LLM rather than editing code directly, while experts were more likely to use general functions and direct code editing [21].

2.2.2.5 Limitations

LLM Hallucinations and Reliability: LLMs frequently generate semantically plausible but physically unexecutable actions. Wang et al. identified two types of hallucinations: faithfulness hallucinations (divergence from user instructions) and factuality hallucinations (using actions that don't exist in the primitive set)[17]. These hallucinations can lead to failed task execution even when the high-level planning seems correct. Current mitigation approaches like Wang et al.'s exceptional handling module and Karli et al.'s code verification are reactive rather than preventive, often requiring multiple correction cycles [17][21]. The inherent non-determinism of LLMs also introduces variability in performance, particularly for complex tasks, as noted in Kannan et al.'s evaluation of SMART-LLM [19].

Complex Task Decomposition and Error Propagation: Long-horizon tasks consistently show lower success rates across multiple studies. Liu et al. reported significant performance degradation for complex tasks due to error accumulation across sub-tasks [22]. When a single component in a sequence fails, it often leads to cascading failures in subsequent actions. Kannan et al.'s evaluation of SMART-LLM revealed that even GPT-4, the best-performing model in their study, struggled with scenarios requiring strategic team formation due to skill limitations of individual robots [19]. The challenge grows exponentially with task complexity, as each additional step introduces new opportunities for error.

Environmental Perception and Grounding: Many systems struggle with the "symbol grounding problem" – connecting abstract language commands to physical objects and actions in the real world. Liu et al. found that lower success rates for complex tasks were primarily due to perception limitations, specifically YOLO bounding box inaccuracies [22]. Without accurate environmental perception, even perfectly planned action sequences fail during execution. The gap between the LLM's semantic understanding and the robot's physical capabilities remains a fundamental challenge, with most current approaches relying on pre-defined object coordinates or simplistic perception systems.

High-Level to Low-Level Execution Gaps: Standard LLM-based approaches lack the ability to generate appropriate motion trajectories for complex physical tasks. Lykov and Tsetserukou noted that their LLM-BRAIn system couldn't support recursive generation of subtrees, limiting the size and complexity of generated behavior trees [18]. Sun et al. acknowledged that their robot models and reference motion data were relatively idealized compared to real-world conditions [20]. The disconnect between high-level planning and low-level motion control represents a significant barrier to deployment in unstructured environments.

Scalability and Adaptability Challenges: Current approaches often rely on carefully curated prompts with extensive few-shot examples, limiting their adaptability to novel scenarios. Wang et al. used 23 parsing examples to demonstrate correct task planning[17], while Lykov and Tsetserukou's dataset required 8,500 instruction-following demonstrations[18]. This dependency on extensive pre-defined examples conflicts with the goal of creating truly adaptable systems that can generalize to new tasks and environments. Furthermore, many systems require robot-specific function libraries, making cross-platform deployment challenging.

Part III

Requirements

To address the critical challenges identified in current HRI systems, this thesis must deliver:

- **Natural Language Fluency:** The system must enable complete robot control through natural language conversation, maintaining contextual awareness and supporting fluid dialogue between operators and robots. This includes handling ambiguous commands, maintaining conversation history, and allowing for dynamic task refinement through follow-up interactions.
- **Robust Error Handling:** The system must autonomously detect, communicate, and resolve execution failures and ambiguities without requiring technical expertise from operators.
- **Accessibility:** The system emphasizes user-friendliness, allowing users to complete tasks efficiently, regardless of their familiarity with the system. This approach reduces the learning curve and technical barriers, often limiting robot operation to trained professionals.
- **Operational Transparency:** The system must allow operators to monitor robot operations to understand the intentions behind the robot's behavior. By creating this level of transparency, the operator can quickly identify instances where the system may misinterpret commands or encounter execution failures, reducing the risk of error propagation in multi-step tasks.
- **Robot Coordination:** The system must effectively coordinate activities for single or multiple robots.
- **Modularity:** The system should be modular to ensure seamless integration within existing applications and robotic systems.
- **Cross-Platform Compatibility:** The system must demonstrate high adaptability across numerous devices and platforms, promoting extensive accessibility and enhanced usability. This flexibility facilitates robotic systems' seamless initiation and deployment, irrespective of the underlying hardware configurations.

Part IV

Natural Language HRI System

4.1 Overview

The Natural Language Human-Robot Interaction (NL HRI) system enables intuitive communication between operators and robots through three integrated components:

1. **Web Application**: A user-friendly interface enabling operators of all technical backgrounds to issue commands, monitor task progress, and receive visual feedback.
2. **Natural Language Interface Package**: A language processing system that interprets and coordinates human instructions through specialized nodes for handling conversation flow, task planning, context awareness, and error management.
3. **TeMoto Action Engine Package**: A modular execution framework that converts action plans into robot-specific commands, providing hardware independence and cross-platform compatibility.

This architecture creates a robust bridge between natural human communication and precise robotic execution while maintaining adaptability across different platforms and operational contexts. By integrating advanced language understanding capabilities within the established TeMoto framework, the system delivers a modular, accessible solution for intuitive robot control, as illustrated in Figure 4.

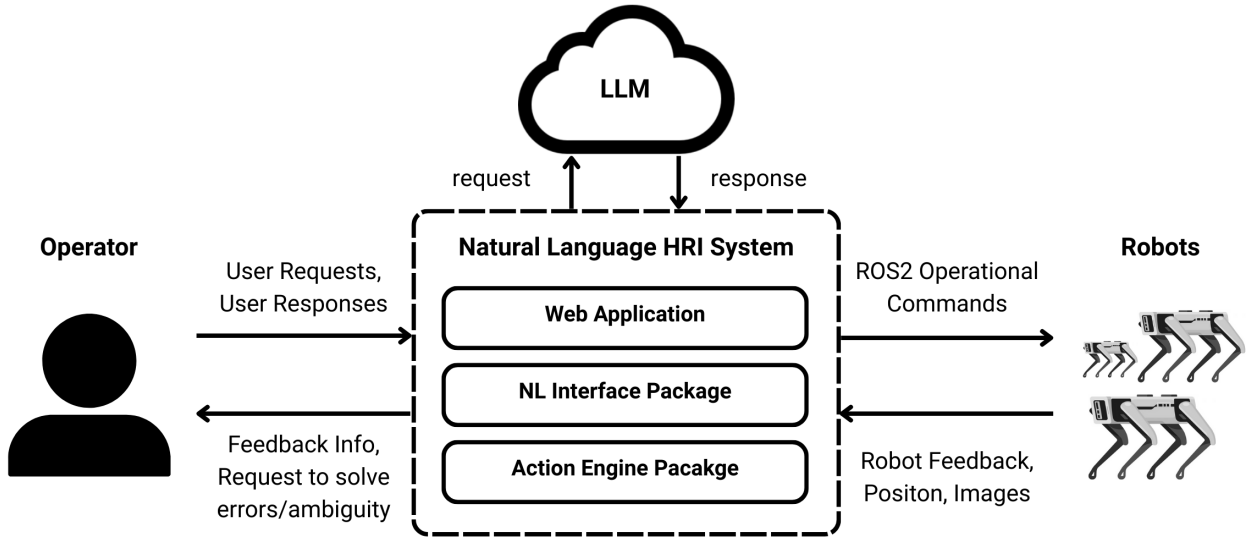


Figure 4: High-level NL HRI System functionality

The operational workflow begins when an operator issues natural language commands through the Web Application. These commands are processed by the NL Interface Package, which generates structured execution sequences for the Action Engine. The Action Engine executes each action from the sequence, controlling the robot's execution through ROS2 protocols. Throughout this process, a continuous feedback loop keeps operators informed with execution updates, generated information, and visual feedback.

The following sections examine each component of this integrated architecture in detail, beginning with the foundational TeMoto Action Engine that provides the execution framework for robotic tasks.

4.2 Temoto Action Engine Package

Building upon the TeMoto framework introduced in Section 2.1.2, the Natural Language HRI system leverages the Action Engine component to execute robot commands effectively. It accomplishes this by transforming natural language instructions into properly formatted UMRF graphs that the Action Engine can process.

The upcoming sections explore the graph execution process, UMRF structure requirements, and specialized actions developed to demonstrate the system's capabilities.

4.2.1 Graph Execution

The integration approach maintains TeMoto's actor-based architecture, where each robot in the network is associated with a distinct Action Engine instance identifiable by a unique "actor" name (Section 2.1.2.3). These Action Engines instances continuously monitor for UMRF events directed at their specific actor, processing them through dedicated ROS2 topics that control graph execution:

- **umrf_graph_start**: Starts the execution of a provided UMRF sequence in the Action Engine when no graph is currently running.
- **umrf_graph_modify**: Updates the graph being processed by the Action Engine. In addition to the updated UMRF graph, the name and ID of the action at which execution should resume are included.
- **umrf_graph_pause**: Pauses the execution of the graph in the Action Engine.
- **umrf_graph_resume**: Resumes the graph from the point where the Action Engine was paused.
- **umrf_graph_stop**: Stops the Action Engine execution and clears any graph that is currently being executed.

Throughout the execution of the graph, the Action Engines initiated publish feedback to the **umrf_graph_feedback** topic. This feedback provides essential information about the state of the graph and actions, including statuses such as "RUNNING," "UNINITIALISED," "PAUSED," "STOPPED," and "ERROR." Upon any "ERRORS," the UMRF feedback includes a list of messages generated during execution to help resolve any errors that may have occurred.

4.2.2 UMRF Structure

UMRF graphs provide the structured format for defining robot execution sequences within the Temoto Action Engine. Each graph contains several key elements that establish its structure and operational parameters:

- **Name**: Defines the name / ID of the graph, must be unique during the Action Engine execution to prevent conflicts with other graphs.
- **Description**: Provides context but does not affect behavior.

- **Graph Entry and Exit**: Defines the initial and concluding actions within the graph execution flow.
- **Actions List**: Defines all actions to be executed during the operation.

Within the Action List, each action requires specific attributes to be properly identified and executed within the system. These attributes ensure that the action can be properly located, parameterized, and connected to other actions:

- **Action name**: References to the name of the action executed by the Action Engine (such as GetCoordinates, NavigateTo, Inspect, see Section 5.1)
- **Input parameters**: Specifies the required data (e.g., coordinates, thresholds) needed for action execution.
- **Output parameters**: Defines the data produced that can be passed to subsequent actions.
- **Instance ID**: Unique identifier for referencing this specific action instance.
- **Parent-child relations**: Establishes the execution sequence where completed parent actions execute child actions.

Additionally, conditional parameters can be added to the parent and graph exit parameters, enabling dynamic flow control based on action execution outcomes, such as pausing execution upon errors or continuing to subsequent actions upon action completion.

The accurate generation of UMRF Graphs is essential for the Natural Language Interface Package. Even minor errors in parameters or flow definitions can lead to the Action Engine rejecting commands from the Natural Language Interface, which could disrupt the interaction between humans and robots.

4.3 Natural Language Interface Package

Having established the foundational TeMoto Action Engine, which manages robot control through UMRF graphs and action execution, we now focus on the core contribution of this thesis: the Natural Language Interface Package. This package enhances TeMoto's capabilities without changing its core architecture. Instead, it serves as an intelligent "Command Source" layer (see Figure 1) that connects human communication with the task execution capabilities of the Action Engine.

This NL HRI package addresses the key requirements outlined in Section III by developing specialized components that convert natural language intent into properly formatted UMRF graphs. Operators can issue conversational commands through this architecture, while the underlying framework guarantees robust and reliable control of the robots. The following sections explain how these components transform conversational commands into precise, actionable robot instructions.

4.3.1 Architecture

To effectively process natural language commands within the NL Interface Package, the system distributes functionality across multiple specialized ROS2 nodes. This package addresses the requirements for natural language fluency through conversational processing, robust error handling via autonomous and active resolution mechanisms, and operational transparency through

real-time feedback. The modular design enables dynamic task updates while maintaining contextual awareness throughout robot operations.

Based on these design principles, the following nodes were developed to process natural language commands into executable UMRF graphs for the Action Engine:

1. **Chat Interface**: Processes natural language inputs through LLM integration and manages the conversational flow with the operator.
2. **UMRF Planner**: Converts processed language commands into formalized action graphs using TeMoto's Unified Meaning Representation Format.
3. **Error Handler**: Multi-stage resolution strategy for handling ambiguities and failures.
4. **Memory**: Maintains contextual awareness by storing environmental information, object locations.
5. **Display Interface**: Standardizes visual feedback channels between robots and the web application.

Through this modular architecture, each node addresses a specific aspect of the human-robot interaction, creating dynamic communication flows as it interacts with the operator through the Web Application and the Action Engine.

4.3.2 Communication Flow

Building on this modular approach, Figure 5 illustrates the communication flows within the Natural Language HRI system, showing how the five NL Interface Package nodes interact with each other and external components during operation. The diagram demonstrates the distinct workflows that ensure reliable robot control: action sequence generation, graph execution, error handling, memory management, and visual feedback.

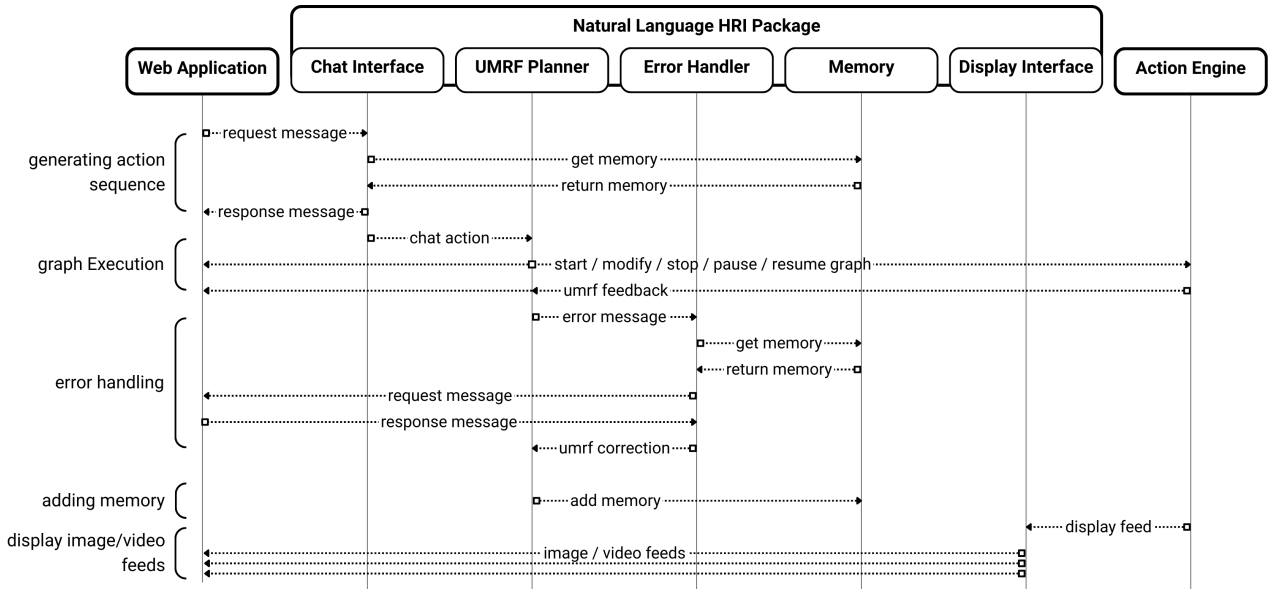


Figure 5: Natural Language HRI package communication flow

The communication flow between these components follows distinct patterns for different operational scenarios:

1. **Action Sequence Generation**: When the operator issues a natural language command through the Web Application, it passes through the Chat Interface node, which retrieves relevant context from the Memory node before generating appropriate NL responses transmitted back to the operator, system commands, and action sequences delivered to the UMRF Planner.
2. **Graph Execution**: From the generated system commands and action sequences, the UMRF Planner manages execution control and constructs UMRF graphs, which it publishes to both the Web Application (for operator monitoring) and the Action Engine (for robot execution). As execution progresses, the Action Engine continuously sends feedback about the execution state back to the UMRF Planner to maintain synchronization.
3. **Error Handling**: When execution failures occur, the UMRF Planner node triggers the Error Handler node to attempt resolution autonomously by consulting Memory or requesting clarification from the operator.
4. **Memory Management**: Throughout operations, the UMRF Planner contributes to the Memory database through commands generated within the Action Sequence Generation process.
5. **Display Management**: The Display Interface maintains visual communication channels for monitoring robot activities.

Through the presented architecture and communication flow, the NL Interface package successfully bridges the gap between conversational language and action sequences (UMRF graph) required by the TeMoto Action engine. The internal architecture, functionality, and role of each node will be explored further in the subsequent sections.

4.3.3 LLM Configuration

Throughout the operations, the system utilizes large language models (LLMs) to address various problems. This includes generating action sequences through the Chat Interface, resolving execution failures with the Error Handler, and accessing relevant information from memory. To maintain modularity, all interactions within the NL Interface package are funneled through the **llm_core** function. This function is conveniently located in a designated folder within the NL Interface package, along with the instruction prompts used to generate responses for each process.

Before sending requests to the LLM API, the **llm_core** function configures the LLM model and the key parameters that guide response generation. The LLM model is defined within this function and set to OpenAI's **GPT-4**, chosen for its performance in other solutions presented in the Related Works, Section 2.2.2. Then, while the parameters are set within individual nodes, they maintain consistent values across the system. The temperature is set to 0.1 to promote predictable output patterns, while the top-p value is configured at 0.6 to allow for controlled variability within reliability constraints. Both the frequency penalty and presence penalty are set to zero to allow necessary repetition of actions and JSON structure elements in graph generation. The maximum tokens parameter varies depending on the specific task; for action sequence generation, it is set to a higher value (2048 tokens) to ensure complete responses without truncation, whereas memory retrieval requires fewer tokens (1024) for efficiency.

4.3.4 Multi Robot Considerations

Building on TeMoto's framework, the Natural Language Interface package must handle simultaneous conversations and action planning for multiple Actors. This multi-robot capability

introduces complexity in maintaining separate conversation contexts, preventing command interference, and ensuring efficient resource utilization. The system addresses these challenges through a specialized architecture that isolates robot-specific data while enabling coordinated operations.

4.3.4.1 Target and TargetManager Classes

The system employs a two-tier management structure to support multiple robots operating simultaneously. The Target class functions as an individual container for each robot's operational state, preserving elements such as conversation contexts, action queues, memories, and execution status. This isolation prevents commands intended for one robot from affecting others.

These individual Target containers are coordinated by a centralized TargetManager that maintains the system-wide robot registry. When the operator sends a command to multiple robots, the TargetManager distributes these instructions to the appropriate Target instances while ensuring data consistency through protective "lock" mechanisms.

4.3.4.2 Concurrency Management

The nodes leverage ROS2's ReentrantCallbackGroup and MultiThreadedExecutor capabilities to achieve concurrency while managing multiple targets. Reentrant Callback Groups are established for each topic subscription and service request interaction, creating clear boundaries between different types of operations.

To implement parallel processing within a single callback group, tasks are distributed across multiple threads, avoiding bottlenecks typical of sequential processing. This structure ensures that if processing for a particular target encounters delays, the others continue processing without interruption.

With these measures, the system can effectively manage multiple robots while ensuring consistent performance and coherent communication throughout the execution. This multi-target architecture forms the foundation for each specialized node in the NL Interface package, beginning with the Chat Interface node, which serves as the entry point for processing natural language commands provided by the Operator.

4.3.5 Chat Interface Node

The NL Interface pipeline begins with the Chat Interface node, providing a conversational gateway between human operators and the robot control system. This component transforms unstructured natural language input from the Operator into structured action plans while maintaining contextual awareness across interactions.

Figure 6 illustrates the robust processing mechanism of the Chat Interface node as it handles diverse message types. These messages range from simple questions like "What functionalities can you perform?" to direct instructions like "Go inspect the Storage Area." Additionally, they can involve more abstract scenarios, such as "The package is missing from the loading area." Through this architecture, the system interprets natural language inputs and generates appropriate action sequences that fulfill operator requirements.

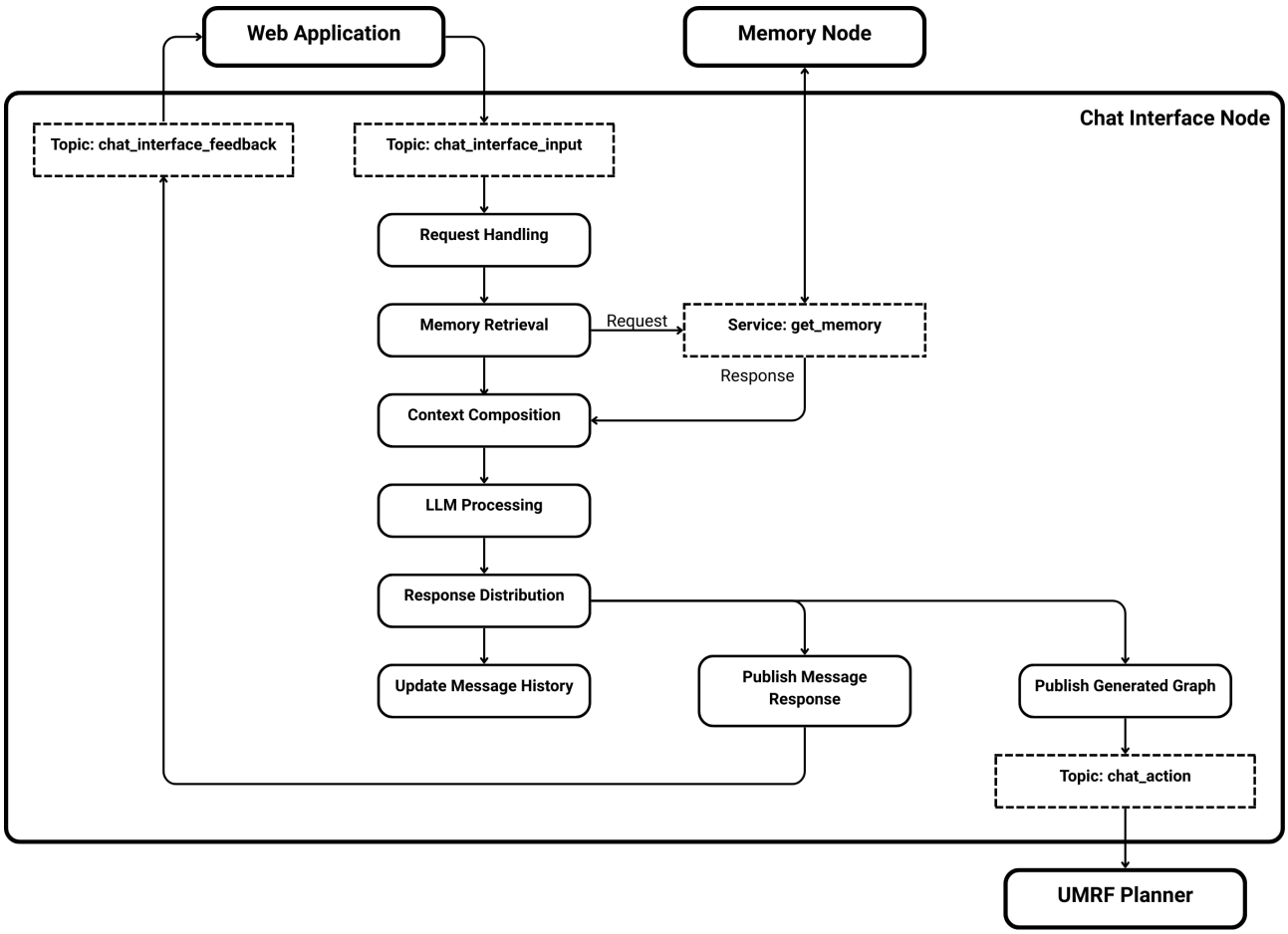


Figure 6: Chat Interface node flowchart

The data flow begins when the Web Interface publishes a request message on the **chat_interface_input** topic. The Chat Interface then processes the message through the following steps:

1. **Request Handling**: Processing the incoming JSON message to extract target, type, and content.
2. **Memory Retrieval**: Gathering relevant contextual information from the Memory node.
3. **Context Composition**: Constructing a comprehensive message for LLM with system instructions, conversation history, user request, and memory.
4. **LLM Processing**: Transforming the user message into structured action sequences.
5. **Response Distribution**: Validating and sending the natural language response to the web application and the action queue to the UMRF Planner.

These steps lay the foundation of the Chat Interface node and the first stage of the NL Interface package.

4.3.5.1 Request Handling

The operation begins by processing incoming message requests from the **chat_interface_input** topic, which provides JSON-formatted data containing three essential elements: the message content, target identifiers, and message type.

When extracting the message content from the input JSON, the node first identifies the message type, distinguishing between "request" messages (commands from operators) and "response" messages (answers to system queries, which the Chat Interface node ignores, see Section 4.4.4). With the request type validated, the node processes the target identifiers for request messages, which are crucial for executing operations for specific Actors running within the Action Engine and separating conversations between different targets.

This initial processing stage ensures that each message is correctly routed and contextually positioned within the ongoing interactions between operators and specific robot targets, establishing the foundation for all subsequent processing stages.

4.3.5.2 Memory Retrieval Operation

After extracting the request message, the Chat Interface node seeks relevant information from the Memory node. This critical step provides pertinent contextual data (such as descriptions of objects in the environment) to the LLM before generating action sequences, thereby grounding the system in relevant information.

During the memory retrieval process, the chat interface will submit the user message to the **get_memory** service and, in response, receive a JSON list of memories that includes:

- Memory type indicators (text or image format)
- Detailed content associated with each memory entry
- Brief description of the memory to summarize the more detailed content.

The integration of memory into the system ensures that action sequences are generated with awareness of previous operations, environmental conditions, and system states, enhancing the contextual intelligence of the overall human-robot interaction system.

The process underlying the system's capability to retrieve relevant memory will be elaborated in Section 4.3.8.

4.3.5.3 Context Composition

The quality of outputs generated by large language models greatly depends on the information provided. Therefore, compiling a well-structured and coherent context is essential to ensure the LLM generates precise outputs for the diverse range of operator message requests.

This context composition is organized into four key components: System Instructions, Target Conversational History, Relevant Memory and User Message:

- **System Instructions**: Provides a comprehensive list of directives for the LLM, establishing the context, outlining goals, specifying available actions (GetCoordinates, NavigateTo, Inspection, developed for Demonstration, Section 5.1), and detailing system commands (stop, pause, clear, skip, add_memory, expanded in UMRF Planner Section 4.3.6.2). These instructions conclude with examples that illustrate expected behavior patterns.
- **Conversational History**: Maintains a capped record of previous interactions for each target, providing essential short-term context without overwhelming the language model. This history ensures continuity across interactions while maintaining performance quality over time.

- **Relevant Memory**: Supplies contextual information that has been stored for long-term reference, grounding current operations in relevant historical data about the environment and previous activities.
- **User message**: Central operator query that directs the LLM response generation.

The carefully structured context ensures that the LLM receives all necessary information to generate coherent and contextually appropriate action sequences and responses. These messages are then sent to the LLM API through the configuration established in Section 4.3.3.

4.3.5.4 Response Distribution

Upon receiving the response from the LLM API, a validation and distribution process is established, ensuring system stability while maintaining effective communication with both operators and robot control systems.

The generated LLM response typically provides accurate results but may still contain errors and "hallucinations." The initial step is validating the JSON structure by parsing the LLM output and removing non-conforming content.

After validating the output formatting, the node extracts two key components from the processed JSON:

- **Natural Language Response**: An explanation that provides insight into how the LLM interpreted the request and what actions will be taken. This response is sent to the web application via the **chat_interface_feedback** topic for operator display. The system also logs this response in the target's conversational history for future context.
- **Action Directives**: The structured action and system command queues that define the target's operation. This component is forwarded to the UMRF Planner through the **chat_action** topic, which uses it to update the target's graph and control their execution dynamically.

4.3.5.5 Chat Action Message Structure

The Chat Interface node generates a custom "Chat Action" JSON to define action sequences and system commands. This approach deliberately avoids the complexity of complete UMRF sequences by omitting graph names, descriptions, parent-child relationships, and graph entry/exit parameters. This simplification offers several advantages: it enables less complex and reliable LLM generation by reducing the total parameters generated, allowing for more concise examples in the context, eliminating potential error sources from the generation process, and optimizing token usage in both input and output. Furthermore, it facilitates dynamic updates to the sequence execution throughout the robot's operation by defining actions as an organized queue.

This "Chat Action" output consists of two primary components:

- **Action Queue**: A streamlined list of actions organized by execution order, containing the action name alongside its input and output parameters.
- **System Commands**: A list of commands that facilitate dynamic modifications of graphs and their execution, enabling real-time adjustments to the robot's behavior.

This approach facilitates dynamic graph updates by allowing incremental changes to existing execution sequences through the UMRF Planner node instead of requiring a complete regeneration for new actions.

4.3.6 UMRF Planner Node

The UMRF Planner node acts as the central component within the NL Interface Package. It organizes and constructs UMRF Graphs for the Action Engine based on the list of actions generated by the Chat Interface and Error Handler nodes.

The UMRF Planner flow is distributed through several layers, shown in Figure 7, that provide the functionalities to dynamically update the graph structures and their execution depending on Operator-based input and the Action Engine execution.

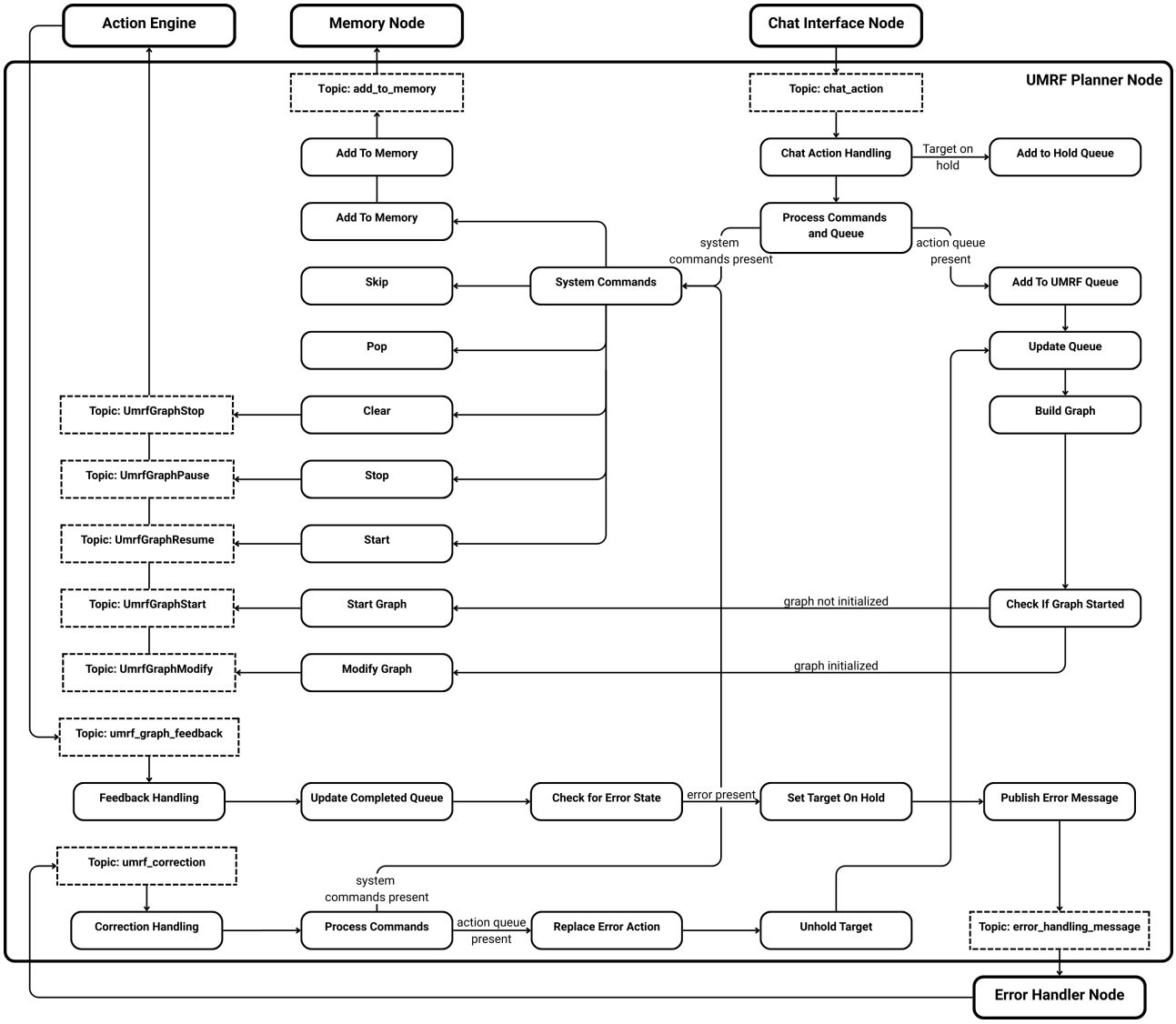


Figure 7: UMRF Planner node flowchart

The UMRF Planner node flow begins with the Chat Interface package publishing the generated "Chat Action" JSON onto the **chat_action** topic and follows the following steps:

1. **Chat Action Handling**: Processing the incoming Chat Action JSON message to extract targets, action queues, and system commands.
2. **Hold Status Verification**: Checking if the target is on hold due to error handling. If a target is on hold, the system temporarily stores the new actions in a dedicated hold queue until the Error Handler node resolves the current issue.

3. **Command Processing**: Executing system commands that control flow and graph execution within the Action Engine (stop, start, clear, pop, skip, add_memory).
4. **Queue Management**: Adding new actions to the target's pending queue while maintaining separation from completed actions.
5. **Graph Generation and Execution**: Building the action sequences into UMRF graphs and publishing them to the Action Engine.
6. **Execution Monitoring**: Processing feedback from the **umrf_graph_feedback** topic to update target's pending queues and detect errors.
7. **Error Handling**: When errors are detected, the target is placed on hold and error details are sent to the Error Handler.
8. **Correction Implementation**: Receiving error correction through **umrf_correction** and replacing failed actions.

These steps enable the UMRF Planner to dynamically structure the graph and its execution, adding to the robustness of the NL Interface package solution.

4.3.6.1 Chat Action Handling

The UMRF Planner begins by processing the "Chat Action" JSONs transmitted via the **chat_action** topic. Each message comprises two main components: the action **queue** to be executed and the set of **system_cmd** control directives.

After parsing the message, the planner evaluates whether each target is on hold due to error resolution. If a target is on hold, the list of actions is added to a hold queue for later processing once the error handling is completed. For targets not on hold, the planner proceeds by processing the command information and then updates the queue of actions with the provided sequence.

This initial handling ensures that each message is contextualized adequately within the current execution state of the robotic system before proceeding with execution planning.

4.3.6.2 Command Processing

The **system_cmd** control directives extracted from the incoming "Chat Action" provide directives on the execution and flow of actions within the Action Engine and UMRF Planner node. The UMRF Planner accommodates a plurality of command types, enabling effective control over the robot deployment:

- **stop**: Temporarily pauses the execution of the graph within the Action Engine while maintaining the current context. A message is published to the **umrf_graph_pause** topic, directing the Action Engine to suspend its operations at this point.
- **start**: Process is restarted from when it was halted. This command sends **umrf_graph_resume** instruction to the Action Engine, directing it to continue processing from the last active action.
- **clear**: Resets the execution context and halts all operations on the specified Action Engine. It resets all internal state variables, clears completed and pending action queues, and publishes the **umrf_graph_stop** topic to instruct the Action Engine to terminate any ongoing processes.

- **pop**: Removes an action from the queue at the specified index. After the removal, the system updates the graph structure to reflect the new execution sequence, thus enabling additional control for the operator to manage the graph dynamically.
- **skip**: Combination of commands designed as a convenience function to bypass the execution of the current node. This sequence briefly pauses execution, removes the first action from the queue, and then immediately resumes with the following action. As a result, it effectively skips one step in the sequence.
- **add_memory**: Creates persistent memory entries through the Memory node. This command prepares a memory entry that includes a timestamp, data content, and contextual information. It then publishes this entry to the **add_to_memory** topic. The Memory node receives this data and integrates it into its persistent storage for future reference.

4.3.6.3 Queue Management

After processing the **system_cmd** control directives, UMRF Planner extracts the **queue** of actions, integrating them into the target's action execution sequence. This process requires maintaining two separate queues for each target: a **completed** and a **pending** queue. This dual-queue system improves execution effectiveness by clearly organizing the action flow to the Action Engine and facilitating dynamic updates of its components.

This queue management approach provides a foundation for graph generation and enables flexible graph modifications in response to varying operator requests and environmental conditions.

4.3.6.4 UMRF Graph Generation and Execution

Following appending the generated action sequences to the pending queues, the system compiles the action sequences into executable UMRF graphs (see Section 4.2.2) through the **_build_umrf_graph** method for execution within the Action Engine.

The process starts by merging the completed and queued action queues into a single comprehensive list of actions, arranging them in chronological order. Next, it establishes a parent-child relationship among these sequential actions and identifies the entry and exit points of the graph. Conditional parent logic is then implemented to ensure a continuous execution flow until the Action Engine encounters an error within an action process, at which point it pauses. During initialization in the UMRF Planner, each graph receives a unique name and description based on the current timestamp, ensuring distinct identification for its respective target throughout the execution cycle.

The resulting graph structure is serialized as JSON and published to the appropriate Action Engine topic based on whether it represents a new execution (**umrf_graph_start**) or a modification to an existing sequence (**umrf_graph_modify**).

4.3.6.5 Execution Monitoring

UMRF Planner node continuously monitors execution feedback from the Action Engine via the **umrf_graph_feedback** topic. This feedback provides detailed information about action states, execution progress, and potential errors.

When the UMRF Planner receives feedback, it identifies the target actor and retrieves the internal pending and completed queues. The message from **umrf_graph_feedback** outlines the same UMRF graph generated by the UMRF Planner to the Action Engine with additional information conveying the execution states for each action.

Actions that have been successfully launched and fully executed are marked as FINISHED. Actions currently executed are marked as RUNNING, while those yet to be executed are labeled as UNINITIALIZED. Under certain conditions, actions may be instructed to halt the graph execution process; at this point, they are marked as STOPPED or ERROR. From this information, the UMRF Planner can track the completed actions and transfer those marked as FINISHED from the pending action queue to the completed queue, ensuring the system is up to date with the overall robot progress.

After updates to the target's queues, the UMRF Planner checks the first action in the pending actions for an "ERROR" state, which indicates that the current action failed during execution. If no errors are found, the feedback processing is considered complete. However, if an error is detected, the UMRF Planner initiates the error-handling process by compiling a message to send to the Error Handler node. Additionally, the target will be placed on hold within the UMRF Planner node until the issue is resolved.

To compile the message for the error handling process, the system retrieves the error message from the log associated with the failed action within **umrf_graph_feedback**. This error message, with the details of the Actor name, the failed action, and the pending queue, is used to formulate the message published to the **error_handling_message** topic.

This feedback ensures that the planner maintains accurate knowledge of execution progress, enabling effective error recovery and sequence modification.

4.3.6.6 Correction Implementation

Once the Error Handler node generates a solution to the execution failure, UMRF Planner node receives the correction through the **umrf_correction** topic. This correction message allows the UMRF planner to dynamically update the UMRF Sequence in an attempt to resolve the error during execution.

The **umrf_correction** topic sends messages in the same format as the "Chat Action" published by the Chat Interface node (see Section 4.3.5.5). The **system_cmd** command directives are processed similarly to the handling of Chat Action threads. However, instead of appending the list of actions to the pending queue, the list is inserted at the first index. This corrective-generated action sequence replaces the failed action while maintaining the overall execution sequence. Once this replacement is complete, the planner automatically unholds the affected target, releasing the execution hold placed during error detection.

When un-holding the target's queue, the process retrieves and adds any temporarily suspended actions to the pending queue, ensuring no generated actions are lost during error handling. The UMRF Planner node then rebuilds and publishes the corrected UMRF graph to the action engine, following the same procedure used for the Chat Action thread.

4.3.7 Error Handler Node

The Error Handler node is the recovery component in the NL Interface system, designed to resolve failures that arise during robot operations. When robots face ambiguities or errors while executing tasks within the Action Engine, this specialized node employs a multi-step resolution strategy combining autonomous problem-solving with guided user interaction.

Errors thrown within the Action Engine graph execution can vary significantly depending on the nature of the operation. System-level errors within these actions may include navigation failures when paths are obstructed or communication breakdowns that prevent commands from reaching the robot. Decision-based interruptions, on the other hand, occur when actions determine that

additional information or operator intervention is required to proceed. For example, during action processes that involve selecting a specific target object, errors can occur if multiple objects match the given description or if no objects meet the requested criteria. Similarly, errors may arise in inspection tasks when the system assesses that human judgment or intervention is needed.

To manage these scenarios, the Error Handler employs a multi-step approach, as illustrated in Figure 8.

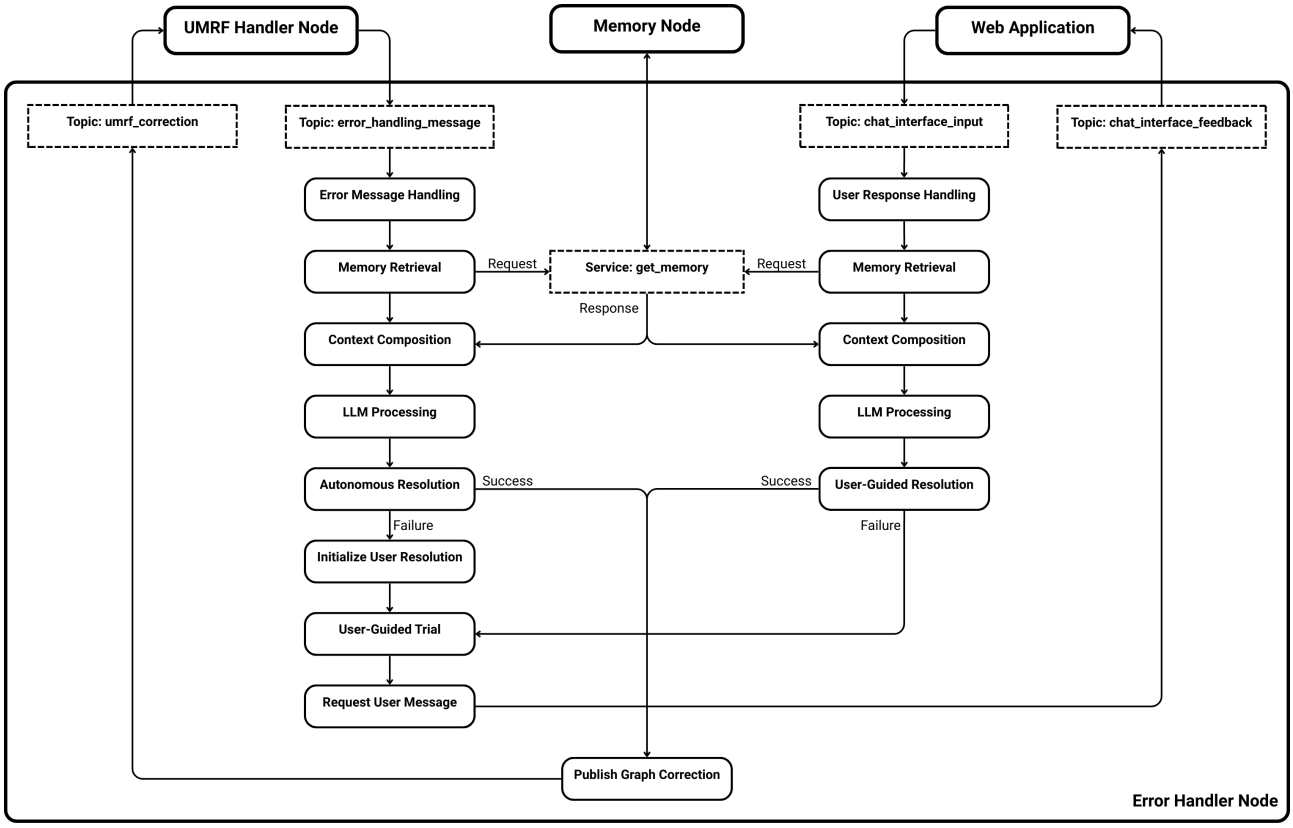


Figure 8: Error Handler node flowchat

The error handling process begins when an error entry is received from the UMRF Planner node via the **error_handling_message** topic, which includes the error message, the specific action where the issue occurred, the pending queue, and the targeted actor. The subsequent data flow from error reception to correction publication follows these steps:

1. **Error Message Handling**: Processing incoming error messages that contain error details, failed action information, and pending queue data.
2. **Memory Retrieval**: Requesting relevant contextual information from the Memory node to inform resolution strategies.
3. **Context Composition**: Combining error information with memory and system context to create comprehensive prompts for resolution.
4. **Autonomous Resolution**: Attempting to resolve the error without human intervention through Large Language Model processing.
5. **User-Guided Resolution**: When autonomous resolution fails, engaging the operator through the web application to provide guidance.

6. **Correction Publishing**: Generating UMRF correction messages that replace failed actions with viable alternatives.

These steps enable the Error Handler to systematically progress from error detection to successful resolution, ensuring robust operation across various failure scenarios.

4.3.7.1 Memory Retrieval

The Error Handler node retrieves contextual information from the Memory node following the same process as the Chat Interface node (see Section 4.3.5.2). This ensures that both Autonomous and User-Guided error resolution attempts are grounded in relevant environmental data and system context. Furthermore, as the following Autonomous Resolution section details, this memory retrieval enables the system to resolve errors or ambiguities automatically, minimizing the systematic need for operator intervention.

4.3.7.2 Context Composition

The Error Handler leverages LLMs to generate replacement actions for failed operations. To achieve this, the Error Handler composes a comprehensive message containing all necessary elements to guide effective error resolution. This message consists of five key components:

- **System Instructions**: Detailed directives for the LLM outlining the error resolution task, available actions, acceptable outputs, and recovery strategies. These instructions emphasize the primary goal of effectively addressing errors in the current graph execution.
- **Error Information**: The complete error message and details about the failed action, providing specific context about what went wrong during execution.
- **Relevant Memory**: Contextual information retrieved from the Memory node that might help in resolving the error, such as environmental descriptions or instructions on how to act in a precise error scenario.
- **Pending Queue Preview**: The first 1000 characters from the pending action queue, providing the LLM with insight into upcoming actions without overwhelming it with excessive information. This preview helps prevent the regeneration of actions already in the queue and ensures continuity in the execution plan.
- **Message History**: For user-guided resolution cases, a record of previous interactions related to this specific error, enabling conversational continuity during the resolution process.

This structured composition ensures that the LLM receives all relevant information to generate appropriate corrective actions while maintaining contextual awareness of the error situation and the planned execution sequence.

4.3.7.3 Autonomous Resolution

Having received an error message, retrieved relevant memory, and composed a comprehensive message, the Error Handler attempts to resolve execution failures autonomously to minimize operator intervention. This autonomous resolution approach leverages LLMs to address several categories of errors:

- **Improper Parameters:** Situations where the action's parameters are not correctly defined, occurring because of errors or hallucinations during the action sequence generation.

- **Memory Updates:** Errors caused by updated states of objects changing in the relevant memory. For example, if a package originally in area A has moved to area B, the system generates corrected navigation actions based on updated memory information.
- **Broad Action:** Situations where actions are defined too broadly, which may result in the execution missing the necessary steps to complete those actions. For instance, if a box inspection fails due to the robot not being located near the box, the system may generate additional navigation steps to position the robot near the box before retrying the inspection.
- **Protocol Implementation:** Memory may contain specific error-handling protocols that were defined by the operator, such as instructions to halt operations when safety concerns are detected during inspection.

The LLM processing returns a "Chat Action" type JSON response in the same format as the Chat Interface, containing the error resolution success status, natural language message response, corrected action sequences, and system commands. If the autonomous resolution succeeds, the Error Handler node publishes the correction to the UMRF Planner via the **umrf_correction** topic. However, if the LLM cannot confidently resolve the error autonomously, the system initiates user-guided resolution.

4.3.7.4 User Guided Resolution

The user-guided resolution process resolves the error by engaging in a dialogue with the Operator, helping the LLM generate appropriate corrective actions. The system utilizes the natural language message generated by the LLM from the previous resolution attempt to describe the error to the Operator in clear, non-technical terms. The LLM translates technical error messages into user-friendly explanations with resolution guidance. For instance, an "Action server '/navigate_to_pose' is not available after waiting" error would be presented as "Unable to start the navigation process. Please verify that the navigation stack is correctly configured and running on the robot." These error descriptions are sent to the Operator through the web application's chat panel via the **chat_interface_feedback** topic as request-type messages (unlike the Chat Interface node, which sends response-type messages).

The Error Handler continuously monitors operator responses through the **chat_interface_input** topic. Upon receiving a response from the Operator, the node retrieves additional relevant memories specific to the Operator's input to provide better context for the next LLM processing attempt. The LLM-generated messages and operator responses are maintained in a temporary conversation history, ensuring natural dialogue flow throughout the error resolution process.

With input from the Operator and new information from memory, the Error Handler node attempts to generate the appropriate corrective measures again. After processing, the LLM returns a response in the same "Chat Action" JSON format as the Autonomous Resolution, indicating whether the error resolution was successful.

If the LLM still cannot provide solutions confidently, the Error Handler initiates another user-guided resolution cycle, presenting a newly formulated question to the Operator. This iterative process continues until the LLM successfully generates corrective actions or the system reaches a maximum number of attempts to prevent infinite loops.

Upon successful resolution, the Error Handler publishes the corrective actions to the **umrf_correction** topic for processing by the UMRF Planner. Simultaneously, it notifies the Operator of the successful resolution by sending a response-type message to the **chat_interface_input** topic.

4.3.8 Memory Node

The Memory Node acts as a knowledge repository within the Natural Language Interface package, ensuring contextual awareness throughout interactions by storing and retrieving relevant information shared by the operator or generated throughout operations. This persistent memory system empowers robots to anchor natural language commands within the environment, reference previous interactions, and uphold a coherent understanding of their operational context during extended operations.

The Memory node aims to create a structured database of information that assists in generating action sequences within the Chat Interface and Error Handler nodes. The Memory ensures that the LLM's responses are grounded throughout robotic operations. To effectively manage this information, the node utilizes two callbacks: one for storing information and another for retrieving relevant data, as illustrated in Figure 9.

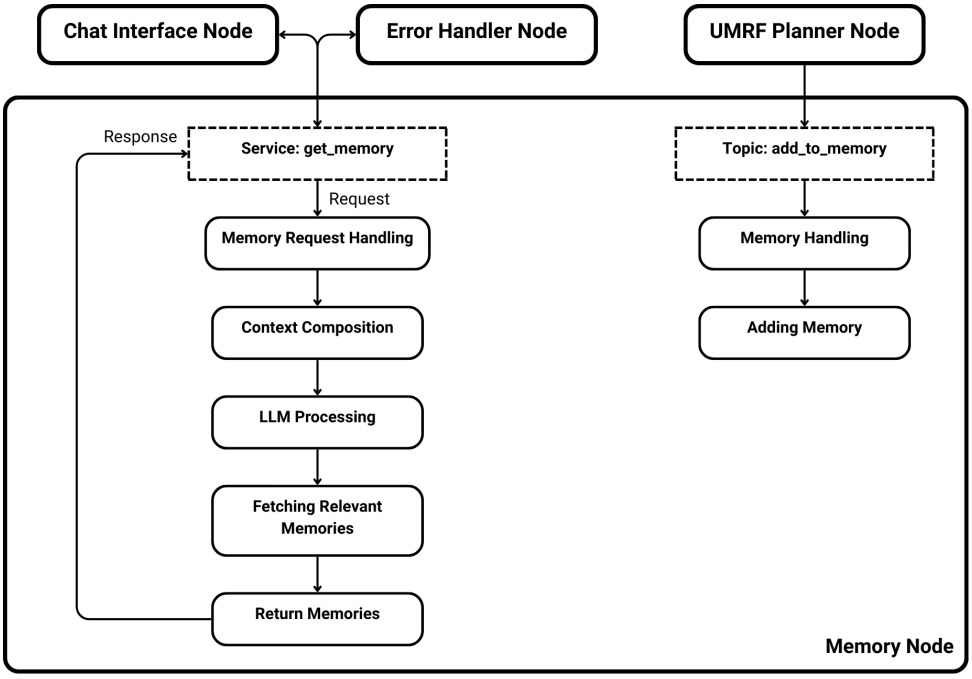


Figure 9: Memory node flowchart

The UMRF Planner and the defined actions within the Action Engine are capable of adding memories via the **add_to_memory** topic; this process follows:

1. **Memory Handling**: Receiving memory data in JSON format to extract name, type, data content, and descriptive information.
2. **Adding Memory**: Adding the memory either by updating existing entries or creating new Memory objects.

When the Chat Interface or Error Handler nodes need contextual information, they are capable of accessing the Memory through the **get_memory** service, obtaining relevant memories through the following process:

1. **Memory Request Handling**: Processing requests from the client node by extracting the query message that needs contextual information.
2. **Context Composition**: Combining system instructions, memory descriptions, and the original query to create a comprehensive prompt to fetch relevant memories.

3. **LLM Processing**: Leveraging LLMs to retrieve a list of relevant memories stored within the system.
4. **Fetching Relevant Memories**: Formatting all relevant memories into a JSON for the service response message.
5. **Return Memories**: Returning the list of relevant memories to the client node.

This dual-flow architecture ensures efficient contextual awareness throughout the system's operation.

4.3.8.1 Adding Memories

Memories are added to the system through the **add_to_memory** topic, where the UMRF Planner node publishes the memories that the Chat Interface or the Error Handler nodes might have generated through information provided by the operator.

These memories encompass various purposes, including providing information about the environment to guide navigation or offering instructions for troubleshooting errors as detailed in the Error Handler node. These memories serve as valuable pieces of information that, when integrated with the LLM inside the Chat Interface and the Error Handler nodes, empower it to assemble action sequences with more complexity and efficiency.

Furthermore, entry points can be established through actions within the TeMoto Action Engine, allowing individual actors to collect environmental data and update the global memory with new elements periodically. This ongoing process enhances the system's overall adaptability by incorporating real-time insights from the environment into one, ensuring that decisions are grounded in the most current and relevant data available.

4.3.8.2 Memory Structure

The individual Memory format is structured as a JSON object that includes four key components:

- **Name**: A unique identifier for each memory entry. Using the same name for new memories overwrites previous entries, which could be useful for temporary information like positions. Alternatively, attaching a timestamp to the name when adding to memory ensures a unique entry.
- **Type**: Specifies the format of the memory content, indicating whether it contains text data or a base64-encoded image. This helps the system process the content appropriately when retrieved.
- **Data**: Contains the complete memory content that the system uses to generate action sequences and operator responses. This field stores the primary information being preserved.
- **Information**: Provides a concise summary of the memory's purpose and content. This summary allows for quick assessment without processing the full content, particularly valuable for dense text or image memories.

Each memory is stored within the Memory node's structured database (see Section 4.3.4.1), enabling efficient addition of new entries and seamless retrieval of information when necessary.

4.3.8.3 Memory Retrieval Operation

When receiving a request from the **get_memory** service, the Memory node aims to retrieve relevant contextual information stored within the system and return it to the requesting client. These queries vary, from natural language interactions with operators to error messages during autonomous error handling operations.

To assess the relevance between the input and stored memories, the Memory node leverages the semantic understanding capabilities of a LLM by examining conceptual relationships between the user's request and memory descriptions.

Consider a typical scenario where an operator commands, "Go to the package." Utilizing LLM capabilities, the Memory node identifies relevant memories by processing the names and concise information associated with each memory stored. In this scenario, three memories are provided to the LLM: "package_location: Packages stored in area A," "maintenance_warning: Area A closed for maintenance, using area B temporarily," and "temperature: Weather outside is -4°C." After successfully processing the provided context, the LLM returns the list of relevant memory names, such as "package_location" and "maintenance_warning."

Upon receiving the list of relevant memories, the Memory node retrieves the complete content for each identified memory (including name, type, data, and information), compiles them into a structured JSON response, and returns this contextual package to the client node. This selective retrieval mechanism prevents overwhelming the action generation process with irrelevant information while ensuring all critical context remains available.

4.3.8.4 Context Composition

In order to retrieve the focused list of memory identifiers developed in the previous section, the Memory node constructs a message for the LLM with the following parameters:

1. **Instructions**: System prompt outlining the memory selection task, retrieval criteria, and expected output format. These instructions guide the LLM to identify precisely which memories are relevant to the current context without introducing extraneous information.
2. **Memory Information**: A condensed representation of available memories that includes only the descriptive information rather than complete data content. This approach prevents overwhelming the LLM with unnecessary details while providing sufficient context for relevance assessment.
3. **Request Message**: The original query from the system component, providing the specific context for which memories are needed.

This structured composition enables focused memory retrieval that aligns precisely with the current operational context.

4.3.9 Display Interface

The Display Interface node serves as the visual communication bridge between the robotic system and the web application, uniquely connecting the Action Engine directly to the Web Application within the NL Interface package network, as shown in Figure 10. This specialized component processes varied image and video feeds generated during UMRF graph execution, converting them from diverse robot sensor formats to a uniform web-compatible standard regardless of their original size, quality, format, or frame rate. The standardized visual data is then published to dedicated channels for seamless consumption and presentation in the web application's user interface.

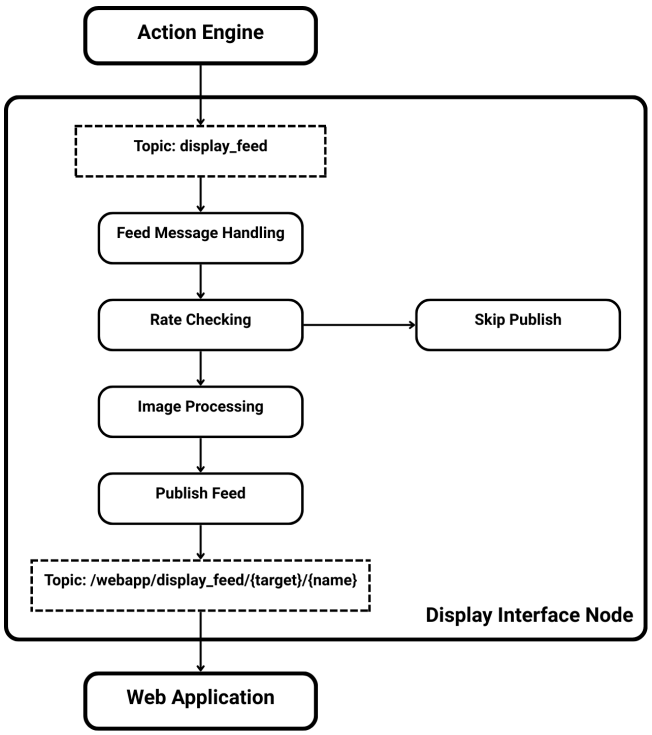


Figure 10: Display Interface node flow chart

The data flow initiates when an action within the Action Engine publishes visual information to the **/display_feed** topic. The Display Interface node processes this information through several crucial steps:

1. **Feed Message Handling**: The node first parses the incoming JSON message to extract the target, name, and image data fields.
2. **Rate Checking**: A rate-limiting mechanism prevents overwhelming the network and client application by ensuring each feed publishes at an appropriate frequency.
3. **Image Processing**: The node converts various image formats (RGB, RGBA, BGR, monochrome, Bayer pattern) into standardized and compressed JPEG images with configurable quality levels.
4. **Publish Feed**: Standardized images are published to web-compatible topics following the pattern **/webapp/display_feed/target/name**.

This standardized method ensures operators receive uniform visual feedback, enhancing the integration of actions within the NL HRI and the overall user experience, regardless of the camera or sensing technology.

4.3.9.1 Image Processing

The Display Interface node processes image data received in JSON format through the **/display_feed** topic. Within the Json, the actual image content within these messages can arrive in two distinct formats:

1. Base64-encoded images: Complete pre-compressed images (JPEG/PNG) converted to base64 strings
2. Raw image dictionaries: Structured data containing uncompressed pixel arrays with metadata (dimensions and encoding type)

The node's processing pipeline automatically detects the input format and converts all color encodings (RGB, BGR, RGBA, monochrome, and Bayer) to standardized RGB, compressing them to JPEG before publishing to specific feed topics. This standardization ensures consistent delivery while optimizing bandwidth usage and preventing backend overload.

4.3.9.2 Integration with Web Application

The Display Interface establishes a direct communication channel with the web application through dedicated ROS topics. For each image feed, the node creates a unique topic following the naming convention /webapp/display_feed/target/name, where:

- **target**: represents the robot or actor name
- **name**: identifies the specific feed (e.g., "camera," "inspection," "thermal")

This structured topic naming scheme enables the web application to efficiently categorize and present visual feeds by source and type, enhancing operator situational awareness during remote operations.

4.4 Web Application

The "Chat HRI" page (refer to Figure 11) serves as the visual interface for communication between the operator and the natural language system. It provides an integrated workspace for conversations, monitoring image and video feedback, and tracking action execution. By organizing and designing the web application to create a smooth, responsive, and engaging experience, we minimize barriers to entry, allowing for broad deployment across various sectors.

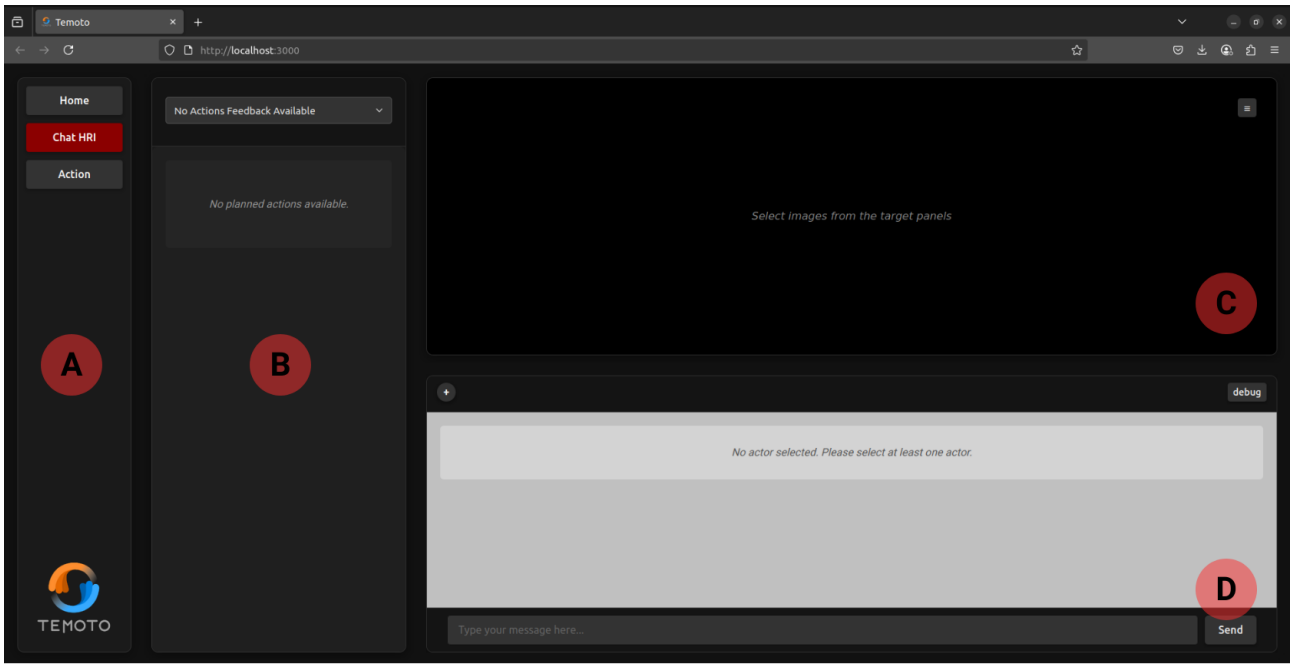


Figure 11: Chat HRI page developed for the Natural language HRI system with components: (A) Menu Panel, (B) Planned Action Panel, (C) Display Panel, and (D) Chat Panel

4.4.1 Component Overview

The web application consists of several key components that work together to provide a cohesive user experience:

- **Chat Panel:** Enables natural language communication with robots.
- **Display Panel:** Provides visual feedback for image and video feeds from robot sensors.
- **Planned Action Panel:** Visualizes the action sequence execution.
- **Menu Panel:** Facilitates navigation between different pages

Each component is designed to be responsive, adapting to different screen sizes from desktop monitors to mobile devices, enabling flexible deployment in various operational contexts.

4.4.2 Integration Approach

The web application leverages React.js for the frontend development, a component-based JavaScript library that facilitates the development of interactive user interfaces. It connects to the NL Interface and Action Engine ROS2 packages, enabling seamless communication with robotic systems through a modular backend architecture. This design effectively separates the presentation layer from the processing layer, allowing for the independent development of each component while ensuring interoperability as seen in Figure 12.

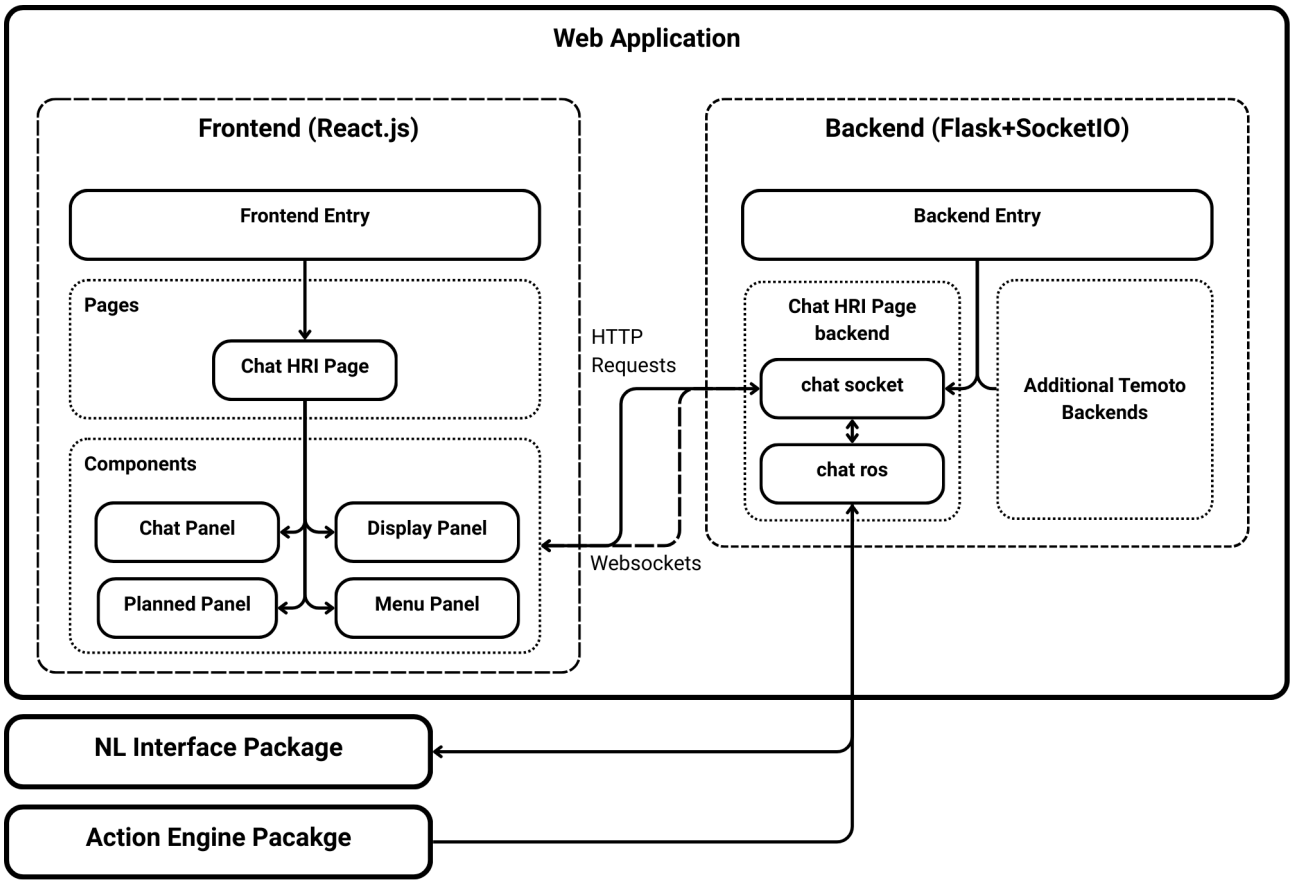


Figure 12: Web Application Architecture

The system architecture is organized into two distinct layers that work together to enable seamless communication between operators and robotic systems:

- **Frontend Layer**: A React.js-based interface organized into modular components. The frontend entry point routes to the Chat HRI page, which consists of four interconnected panels (Chat Panel, Display Panel, Planned Panel, and Menu Panel). Each panel is an independent component that can be developed and maintained separately while working together to provide a comprehensive user interface.

- **Backend Layer**: A Flask and SocketIO-based server with a modular backend entry point that initializes specialized modules. The Chat HRI page backend contains dedicated socket and ROS communication modules, while the architecture supports additional TeMoto integration and features.

The modular design allows for the independent development of individual components while ensuring their seamless integration within the overall system. This means new panels can be added to the frontend, or new backend modules can be integrated without disrupting the existing functionality. This approach supports both system extensibility and maintainability.

4.4.3 Technical Foundation

4.4.3.1 Framework Selection

React.js was chosen for its component-based architecture, which aligns with our modular panel design and efficient rendering capabilities for handling frequent UI updates. The framework's virtual DOM optimization ensures responsive interface performance when displaying rapid state changes in action sequences and live camera feeds from multiple robots, updating only modified components rather than re-rendering entire interface sections.

Flask was selected for its HTTP server foundation and Python ecosystem integration, enabling ROS2 system compatibility, while SocketIO fosters WebSocket communication for real-time bidirectional data exchange between the web interface and robotic systems. This combination supports Cross-Origin Resource Sharing, allowing operators to control robots from various devices across the network, including tablets and smartphones.

4.4.3.2 Backend Design

The backend architecture employs a modular design organized into three specialized components: the application launcher, socket interface module, and ROS bridge implementation.

- **Application Launcher:** The main module serves as the system entry point, managing startup configuration and component initialization. It allows for selective activation of backends through configuration flags (–chat, –action), enabling flexible deployment based on operational requirements. Upon initialization, it connects the requested socket interface modules to the overall runtime environment.
- **Socket Interface Module:** This component manages all client-server communication through HTTP routes for synchronous operations and WebSocket handlers for real-time interactions. For the Chat HRI page, this module maintains the application state, including conversation history, action sequences, and display configurations to ensure consistency throughout the operation. It also provides specialized handlers for chat processing, action visualization, and image feed management, ensuring seamless data flow between the web interface and backend services while initiating the ROS Bridge to enable web application connectivity.
- **ROS Bridge Implementation:** A dedicated ROS2 node facilitates communication between web protocols and the robotics ecosystem. The custom bridge setup subscribes to the NL Interface and Action Engine package topics (umrf_graph_feedback, display_feed, etc.) and publishes user commands (chat_interface_input), translating between ROS2 message formats and web-compatible JSON.

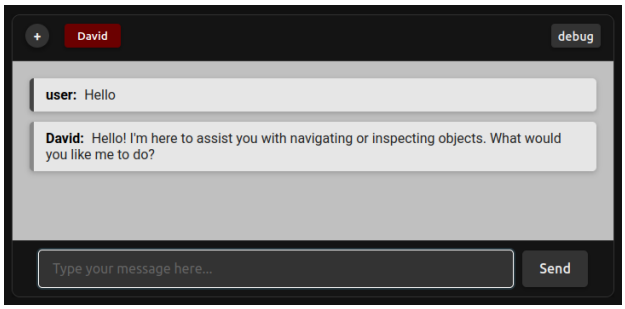
This modular architecture ensures maintainable code, enables independent component development, and supports flexible deployment configurations across different operational scenarios.

4.4.4 Chat Panel

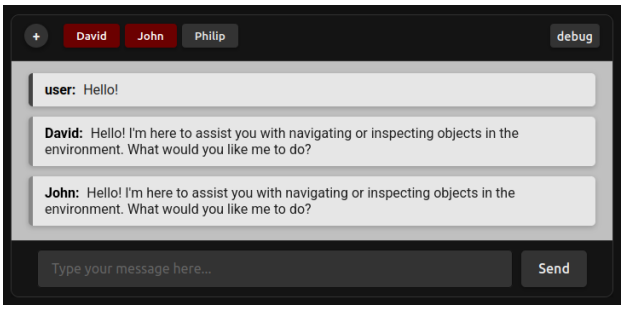
The Chat Panel provides the core conversational interface where operators interact with robots using natural language. Figure 13a shows the panel's layout: conversation history fills the central area with clearly labeled messages from operators and robots, while a text input field and "Send" button at the bottom enable command submission.

Beyond basic messaging, the system supports multi-robot operation via selectable actor tabs displayed across the top of the chat panel. Figure 13b shows this layout with actors "David," "John," and "Philip" established. These tabs represent the robots present in the TeMoto network that the operator wishes to manage and coordinate operations across. New actors can be added through the "+" button at the top left of the Chat Panel. When toggled, it displays a box where the operator can easily input the new actor name, as shown in Figure 13c.

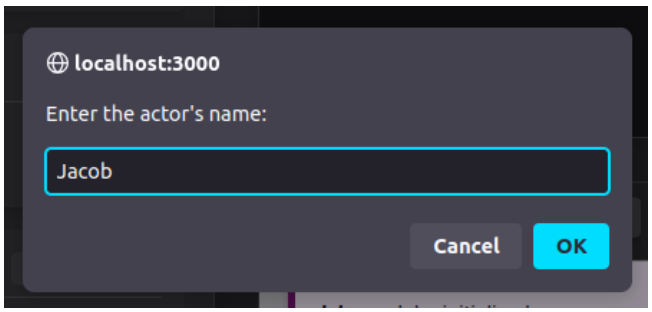
When multiple actors are selected, the system simultaneously sends messages to all chosen targets, facilitating effective broadcast commands. The panel consolidates communications from these selected actors while preserving a chronological conversation history, reducing redundancy, and maintaining context. This design approach alleviates the cognitive load for operators overseeing numerous robots. The actor selection process is intuitive: a simple click on a tab selects that specific actor, while shift-clicking allows for the easy addition or removal of actors from the current selection.



(a) Sending simple message



(b) Operating multiple actors



(c) Adding a new actor to the chat panel

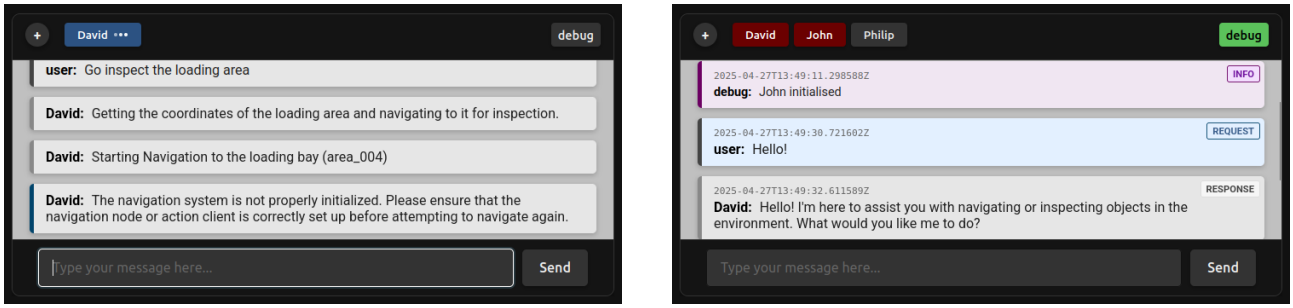
Figure 13: Chat panel single and multi-robot messaging functionality.

The chat panel operates on a request-response pattern, distinguishing between regular interactions and error handling. In typical situations, when an operator sends a message, it is considered a "request" directed at the appropriate actor(s) and processed by the Chat Interface node. However, when the system encounters errors during task execution, the Error Handler node generates "request" messages to prompt the operator for clarification. The operator's subsequent input is then classified as a "response" and used to formulate corrections.

When the system expects a "response", shown in Figure 14a, the Error Handler node's request

messages are prominently displayed in the chat panel with a distinct blue highlight on both the left border of the request message and the actor's tab awaiting a response, ensuring they are noticeable compared to standard interactions. This design choice aids in quickly identifying issues that need attention while preserving the overall coherence of the interface experience.

The Chat Panel includes a debug feature for development and troubleshooting, accessible via a button in the top-right corner, displayed in Figure 14b. Activating this mode enhances the interface by displaying hidden message metadata, including timestamps and type labels (INFO, REQUEST, RESPONSE, ERROR) as colored tags.



(a) Receiving request from actor *(b) Enabling debug on chat panel*

Figure 14: Chat panel error handling mechanism and debug preview.

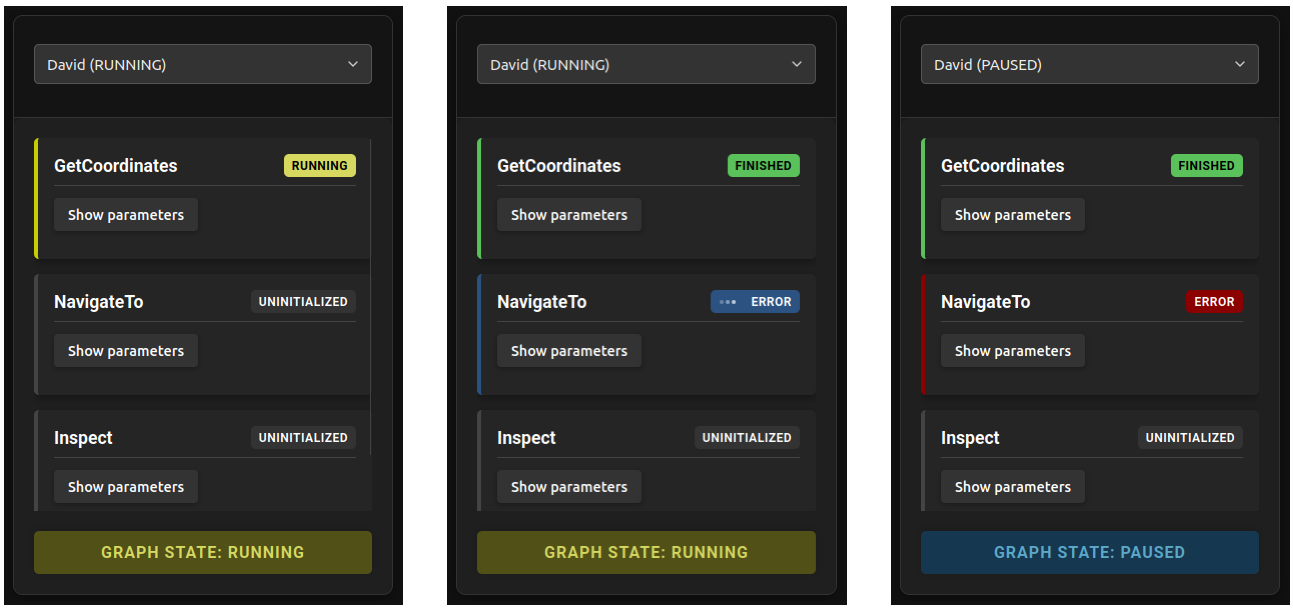
4.4.5 Planned Action Panel

The Planned Action Panel visualizes the real-time execution of robotic tasks, transforming UMRF graphs generated from natural language commands into visual action sequences. This component enhances operational transparency by allowing operators to monitor task progress, inspect execution parameters, and receive immediate feedback on system states without requiring technical expertise.

The panel displays actions as a sequence of cards organized chronologically, with each action's status indicated through color-coded markers and clear text labels. The system uses distinct visual cues for different execution states:

- UNINITIALIZED actions appear in gray, indicating tasks queued for execution
- RUNNING actions display in yellow with dynamic indicators showing active processing
- FINISHED actions are marked in green, confirming successful completion
- STOPPED and ERROR states use red and blue respectively, with error states featuring pulsing animations to draw attention

Figures 15a through 15c illustrate a typical execution sequence. Initially, Figure 15a shows operational flow with a "GetCoordinates" action running while subsequent actions remain uninitialized. After successful completion, the system encounters an error during navigation (Figure 15b), triggering visual alerts with pulsing animations to indicate required intervention. Following operator instructions to stop the process, the graph execution is paused with a "NavigateTo" action labeled as STOPPED (Figure 15c).

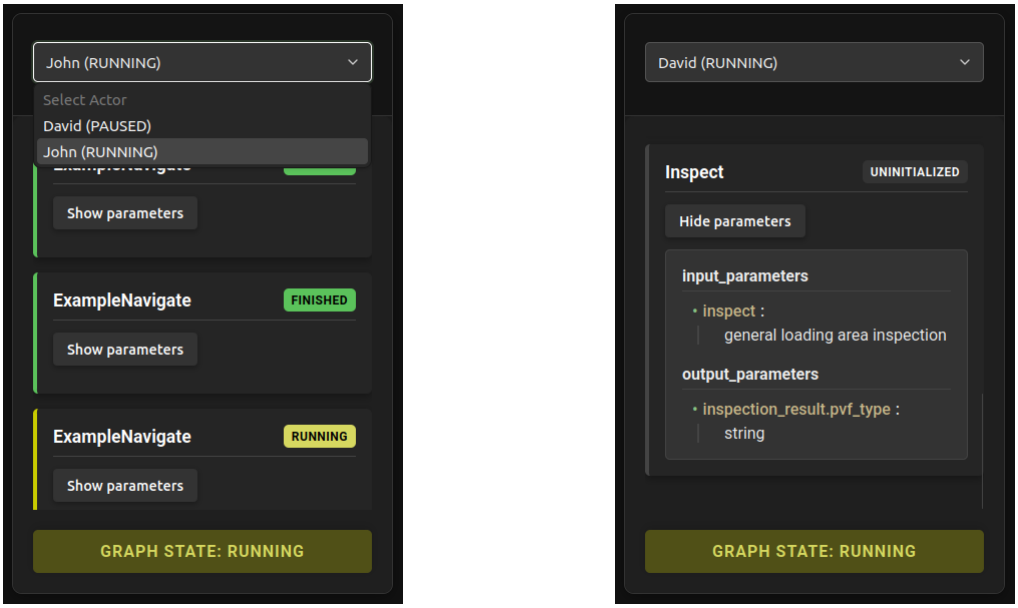


(a) Graph running *(b) Error handling* *(c) Stopped graph*

Figure 15: Planned action panel displaying different execution states.

For multi-robot operations, the panel includes an actor selection dropdown that allows operators to switch between different robots and their respective task sequences. As shown in Figure 16a, this feature automatically displays all active graphs launched through the TeMoto Action Engine, enabling efficient management of distributed operations.

In addition to passive monitoring, the panel offers interactive capabilities through the "Show Parameters" feature, allowing operators to examine detailed action specifications. Figure 16b demonstrates this functionality in the context of an inspection task, where operators can view both input and output parameters, as well as execution context. This organized presentation enables experienced users to access detailed information for troubleshooting while ensuring that novice users are not overwhelmed by excessive technical details.



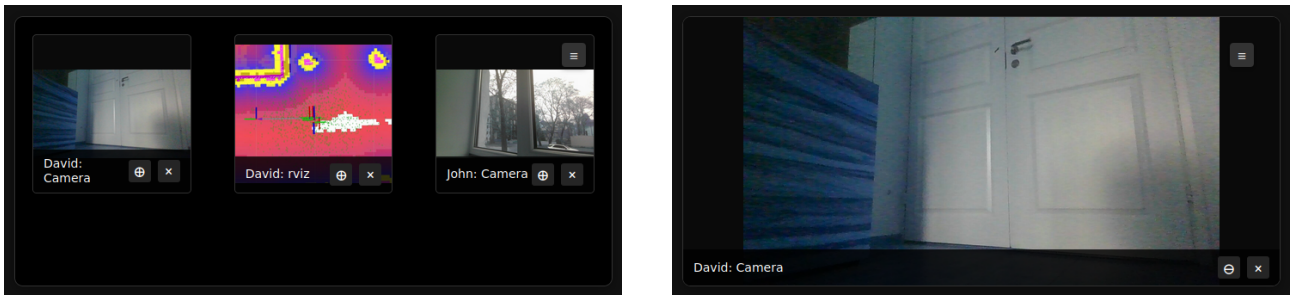
(a) Selecting actor *(b) Inspection task parameters*

Figure 16: Planned action panel interaction features.

4.4.6 Display Panel

The Display Panel provides visual monitoring capabilities, allowing operators to receive feedback from robot cameras and sensors during operations. While not essential for robot control within the NL HRI system, it enhances operational awareness by displaying images captured during task execution, such as inspection results or navigation context.

The panel organizes feeds in a responsive grid layout (Figure 17a), with each feed clearly labeled by actor name and type. For closer examination, operators can expand images to full-screen mode (Figure 17b). The demonstrated setup shows example feeds from two actors: "David" streaming both camera and RViz visualization feeds, and "John" providing a single camera feed.



(a) Multiple subscribed feeds *(b) Full screen view*

Figure 17: Display panel visualization capabilities.

Feed management is handled automatically through a dropdown menu located in the top-right corner (see Figure 18). The system detects available feeds as they become active and organizes them by source actor for easy selection. Once the operator selects desired feeds, the system establishes subscriptions to the Display Interface topics. Selected feeds are then displayed in the panel, automatically showing published images and video streams as robots execute tasks. This visual feedback maintains operational transparency during remote robot control, allowing operators to monitor actual robot states and task progress beyond the planned action sequences.

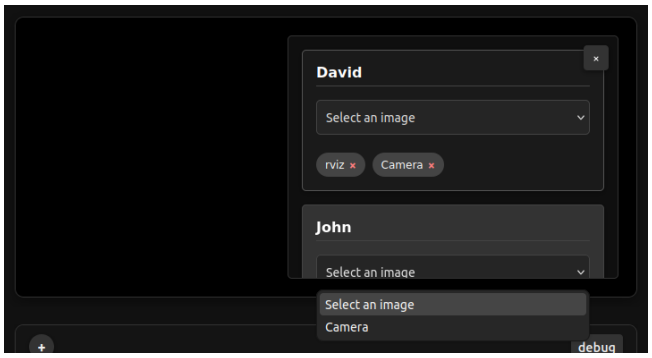


Figure 18: Selecting image/video feed from actors

4.4.7 Menu Panel

The Menu Panel provides navigation between different components of the Web Application. While currently centered on the Chat HRI page for the NL HRI System, the panel's modular design enables future expansion to include additional TeMoto framework features, such as action editors or system configuration tools (Figure 19).

The panel adapts its layout based on screen size - appearing as a sidebar on desktop views and collapsing to a dropdown menu on mobile devices, ensuring consistent navigation across different platforms.

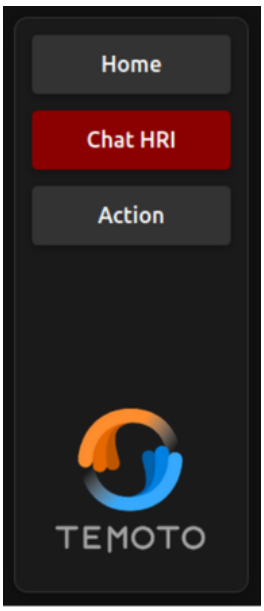


Figure 19: Menu panel page selection

4.4.8 Responsive Design and Cross-Platform Compatibility

The web application adapts to various screen sizes and devices, ensuring operators can control robots effectively whether working from a desktop workstation, tablet, or mobile phone. This responsive design maintains full functionality while optimizing the interface layout for each platform.

The standard desktop layout (Figure 11) positions the menu and planned action panels on the left, with display and chat panels arranged vertically on the right. On tablet-sized screens (Figure 20), the menu transforms into a top banner with dropdown navigation to maximize the available workspace for robot monitoring and communication.

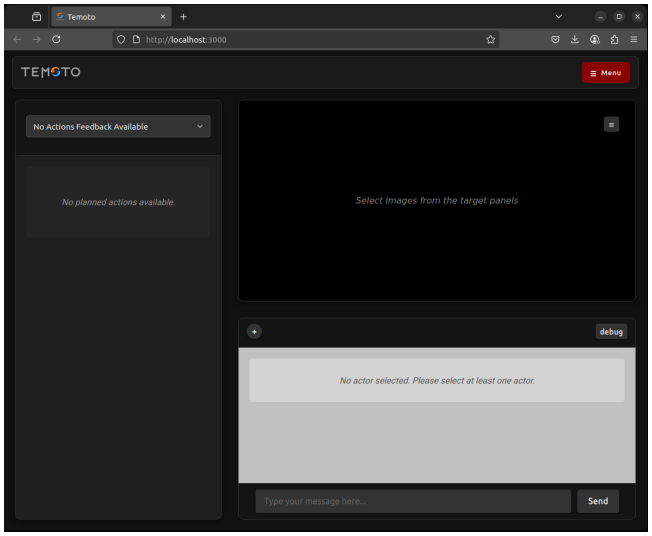
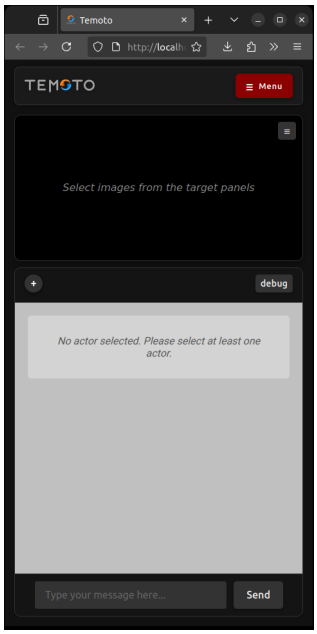
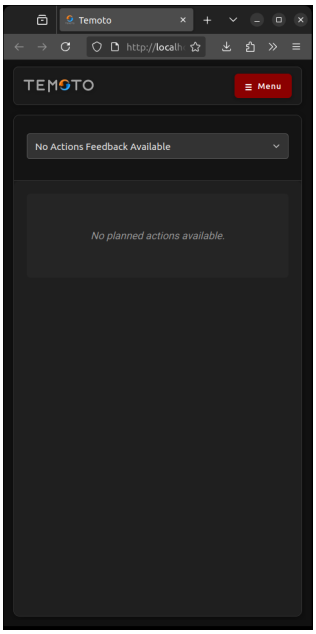


Figure 20: Tablet responsive interface

Mobile devices receive a streamlined interface (Figure 21a) that prioritizes the chat and display panels by moving the planned action panel to a separate page accessible through the menu (Figure 21b). This approach ensures essential robot control and monitoring features remain easily accessible on smaller screens.



(a) Chat and display view



(b) Planned actions view

Figure 21: Mobile responsive interfaces.

This multi-platform approach enables flexible deployment scenarios: operators can monitor robots from a control room using desktop computers, coordinate mobile robot operations through tablets, or quickly connect to robots using smartphones for on-site troubleshooting. These configurations are explored within the demonstrations that follow. The responsive design ensures consistent functionality across all scenarios without requiring separate applications for different devices.

Part V

Demonstrations

To prove the system's real-world viability, a series of increasingly complex demonstrations were constructed, placing robots in scenarios representative of domains with significant potential for robot integration. Beginning with a controlled simulation, we progressed to physical deployments that tested the system's robustness, cross-platform compatibility, and accessibility for non-expert users, ensuring all requirements for the NL HRI system outlined in Part III are met:

1. **Turtlebot3**: A simulated deployment in a healthcare scenario using robots to take care of a patient in a nursing home.
2. **TIAGo**: A physical robot deployment within an underground parking facility to determine potential threats in a security monitoring scenario.
3. **Spot**: A quadrupedal robot deployment in a warehouse management scenario to prepare upcoming shipments.

The following sections describe the specialized actions developed for the demonstrations, followed by the experimental setup, execution, and findings from each deployment scenario.

5.1 Demonstration Actions

Building upon the TeMoto Action Engine framework described in Section 4.2, the GetCoordinates, NavigateTo, and Inspect actions were developed to highlight different aspects of the NL HRI system, from natural language interpretation and error handling to multi-step task execution and human-robot communication.

5.1.1 GetCoordinates Action

The GetCoordinates action serves as the location identification component, taking natural language target location descriptions and returning specific coordinates within the robot's environment for navigation.

When executed, this action compares the natural language location description (such as "in front of the workstation" or "next to the green package") against a database of objects in the environment. Using LLM capabilities, it identifies the specific object that best matches the target description and retrieves its ID. Once the correct object is identified, the action employs an algorithm to calculate an unobstructed path to the target and determines the optimal coordinates and orientation for the robot to face the object properly. If no objects meet the specified criteria or if multiple objects could satisfy the description, the action triggers an error handling mechanism to resolve ambiguities with the operator (see section 4.3.7).

The output of this action includes both the position (x, y coordinates) and orientation (angle) necessary for the robot to approach the target appropriately. This information forms the input for subsequent navigation actions.

5.1.2 NavigateTo Action

The NavigateTo action serves as the mobility component, translating the coordinate information from GetCoordinates into actual robot movement. This action interfaces with the ROS2

Navigation stack (Nav2), which handles path planning and obstacle avoidance.

When executed, NavigateTo creates a navigation goal with the target pose (position and orientation) and sends it to the robot's navigation system. The action continuously monitors the execution status, providing feedback about the remaining distance and handling various completion states. Upon successful completion, the action returns the final robot position. If navigation fails due to unreachable targets, path obstruction, or navigation timeouts, the action throws an error that triggers the error handling system, allowing for operator intervention.

5.1.3 Inspect Action

The Inspect action serves as a perception component that enables visual assessment capabilities. It takes natural language inspection requests as input and returns visual analysis results to the operator through the Web Application.

Upon execution, the action captures and image via the robot's camera and transmits this visual data to a multimodal LLM with domain-specific inspection instructions. The LLM conducts computer vision analysis based on inspection requests, offering either a general assessment of the area or detecting specific objects.

The action produces two primary outputs: textual descriptions of the observations and the corresponding captured images. These are published to separate channels within the web application for operator review. When the analysis identifies objects or conditions that require attention, the action sends alerts through the error handling system to notify the operator of potential issues.

Together with GetCoordinates and NavigateTo actions, the Inspect action completes a functional sequence, enabling object localization, navigation, and visual analysis capabilities within the Natural Language HRI system, bridging natural language interpretation with physical robot operations throughout the demonstrations.

5.2 Turtlebot3 Simulated Demonstration

The Turtlebot3 simulation served as the initial validation platform for the NL HRI system, enabling controlled testing of core functionalities before deploying them on physical robots. This demonstration showcases a use case in which a robot is deployed in a nursing home, assisting a patient with hydration. The assessment focused on evaluating natural language comprehension, spatial reasoning skills, autonomous navigation, visual inspection, and the resolution of ambiguities in a simulated setting.

5.2.1 Setup

The simulation environment was created using the Webots robotic simulator, which features a virtual Turtlebot3 situated in a room resembling a typical domestic setting. Figure 22 illustrates the system deployment configuration used in the Turtlebot3 demonstration.

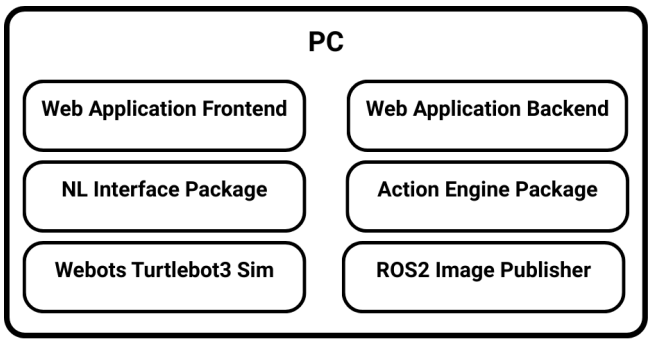


Figure 22: System deployment configuration for the Turtlebot3 simulated demonstration

5.2.1.1 Hardware Configuration

The entire configuration is performed locally on a PC. The PC is connected to a Wi-Fi network, providing access to the LLM API. The simulated Turtlebot3 utilizes a differential drive and a lidar sensor for navigation.

5.2.1.2 Software Configuration

The software deployment is centralized on a single PC, hosting the complete Natural Language Human-Robot Interaction stack. This includes the web application (frontend and backend), the Natural Language Interface Package, and the Action Engine running under the actor name "David." The simulation environment utilizes Webots to implement a virtual Turtlebot3 with integrated Nav2 navigation capabilities. Additionally, a custom ROS2 node publishes simulated camera feeds to enable visual inspection functionality. This centralized architecture mirrors scenarios where autonomous robots operate with onboard computing systems, representing standalone robotic deployments.

This integrated configuration allowed for rapid iteration and testing of the NL HRI system without hardware constraints, while still providing realistic robot behavior and sensor feedback.

5.2.1.3 Environment Configuration

A controlled domestic environment that mimics a nursing home setting was implemented within the Webots simulator. During the initial exploration phase, the Turtlebot3 generated a 2D map of the environment using its SLAM capabilities. The object coordinates, bounding boxes, and semantic descriptions were organized into a JSON file to construct the object map for the GetCoordinates action. The environment included a refrigerator, two sofas of varying sizes, five decorative plants, a dining table, and additional furniture distributed throughout the living area. Additionally, a picture of a bottle inside the refrigerator served as input for the simulated inspection.

5.2.2 Demonstration

The scenario involves using robots in a nursing home, where the role is to attend to the needs of the residents. In this instance, the patient, John, is thirsty, and the system must create a plan to address his need, guided by an external nurse operator. This demonstration is illustrated in Figure 23.

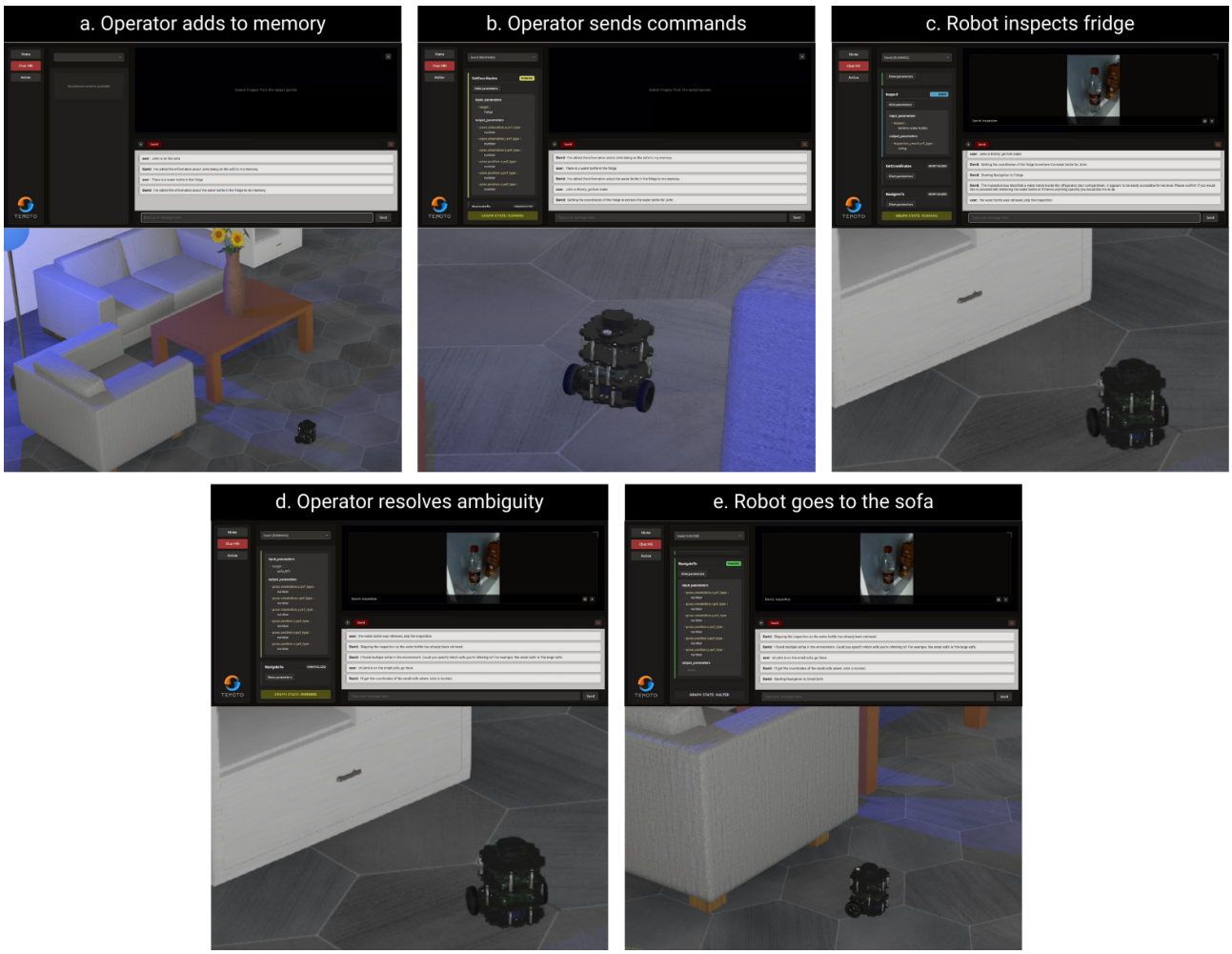


Figure 23: Turtlebot3 simulated demonstration

The demonstration proceeded through the following stages:

a. Operator Adds to Memory

The demonstration begins with the operator providing environmental context to the robot across two prompts: "John is on the sofa" and "There is a water bottle in the Fridge." The system processed this information and acknowledged receipt, confirming the elements were added to memory for future reference.

b. Operator Sends Commands

Next, the operator issues a general command: "John is thirsty, get him water." The natural language interface processes this request by retrieving the relevant information from memory. It recognizes that the water bottle is in the fridge and notes that John is sitting on the sofa. Based on this information, the system generates a sequence of actions to bring the water bottle to John by getting the coordinates of the fridge, navigating there, checking for the water bottle, then finding the coordinates of the sofa and navigating to John.

c. Robot Inspects Fridge

The Turtlebot3 autonomously navigates across the living room to the fridge location. Upon its arrival, it performs an inspection and successfully visualizes a water bottle inside the fridge. In this demonstration, there were no actions developed for the robot to pick up the water bottle, resulting in an error: while the robot correctly identifies the water bottle during the inspection,

it is unable to retrieve it. This error is circumvented by the operator stating that the robot has completed the retrieval.

d. Operator Resolves Ambiguity

After retrieving the water bottle, the system encounters an error while trying to obtain the coordinates for navigating to the sofa. In the living room, there are two sofas, which creates ambiguity regarding John's location. The system requests information from the operator: "I found multiple sofas in the environment. Could you specify which one? For example, small sofa or large sofa." The operator then clarifies the situation by indicating that it is the small sofa.

e. Robot Goes to the Sofa

With the ambiguity resolved, the robot moves to the small sofa where John is sitting. This completes the operation, with each action marked as FINISHED in the planned action panel and the graph execution paused, awaiting new instructions.

5.2.2.1 Discussion

The simulated Turtlebot3 demonstration successfully showcased the core functionality of the NL HRI system in a controlled environment. The system effectively interpreted conversational commands, such as "John is thirsty, get him water," utilizing the large language model's capability to connect words and context.

Additionally, the error-handling mechanism operated as intended during ambiguity resolution, prompting the operator to clarify when multiple sofas were detected and notifying the operator that the system could not retrieve the water bottle with the current actions. The system seamlessly continued execution once this ambiguity was resolved, while correctly noting the inability to retrieve the water bottle due to a lack of manipulation actions. This demonstration confirmed the system's ability to convert natural language into structured action sequences, effectively bridging the semantic gap highlighted in the Problem Statement.

While the simulation confirms the system's basic capabilities, it serves only as an initial evaluation step. Physical demonstrations of the robot in real-world environments are crucial to thoroughly assess the system's robustness and performance across different platforms. Simulations cannot replicate the environmental variability, sensor noise, and locomotion challenges that arise during practical deployments. These limitations emphasize the importance of the upcoming demonstrations with TIAGo and Spot, which will help validate the system's platform-agnostic design and adaptability to various operational contexts.

5.3 TIAGo Demonstration

Transitioning from the simulated healthcare environment, the NL HRI system was deployed on a physical TIAGo robot within a security monitoring scenario. This demonstration adds further depth to the system and explores alternative real-world application possibilities.

5.3.1 Setup

The TIAGo demonstration highlighted the NL HRI system's cross-platform capabilities and mobile accessibility, allowing the operator to control the robot remotely using a smartphone. This deployment required additional configuration to bridge the ROS version incompatibility: while the NL HRI system was developed using ROS2 Humble, TIAGo operates on the ROS1

Melodic distribution. The resulting configuration, shown in Figure 24, demonstrates the system's adaptability across different robotic platforms and its ability to overcome integration challenges.

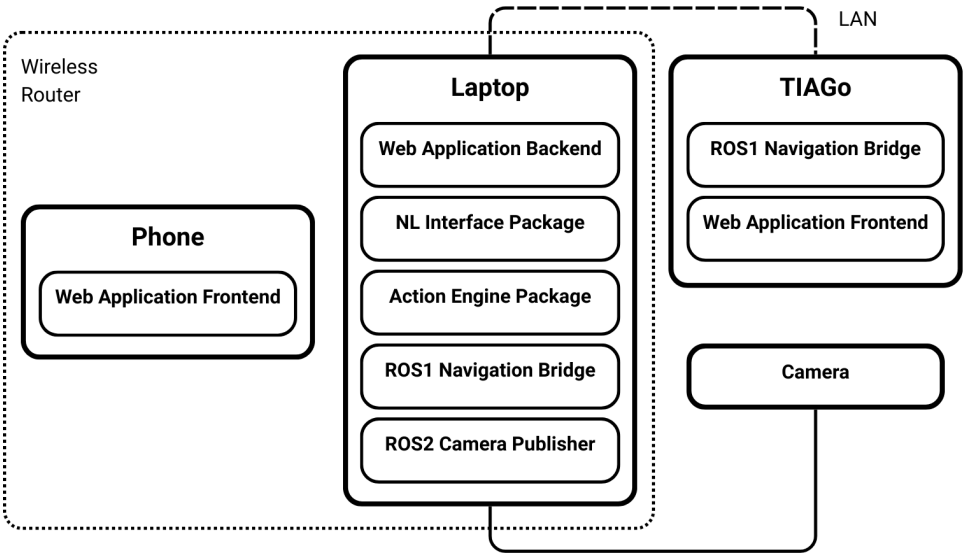


Figure 24: System deployment configuration for the TIAGo demonstration

5.3.1.1 Hardware Configuration

The demonstration features a hardware setup centered around Pal Robotics' TIAGo robot with a mobile base that utilizes a differential drive, a lidar sensor for SLAM and navigation, and a mounted touchscreen. A wired LAN connection between the TIAGo robot and the mounted laptop ensures low-latency transmission of critical movement commands. A wireless router established a connection between the operator's mobile phone and the laptop, hosting the NL HRI and allowing for remote control and providing access to the LLM API.

5.3.1.2 Software Configuration

The software system is distributed across two main computing platforms. The laptop hosts several essential components, including the Web Application Backend, which serves as the foundation for the user interface; the Natural Language Interface Package, responsible for natural language processing; the Action Engine Package for command execution planning, launched under "David"; the ROS2 Navigation Bridge, which facilitates communication with the robot's movement systems; and the ROS2 Camera Publisher, which publishes visual data from the camera to the system.

On the TIAGo platform, the ROS Navigation stack enables autonomous movement, along with a ROS1 Navigation Bridge to ensure compatibility between the robot's native ROS1 system and our ROS2-based system. The navigation instructions generated by the Action Engine in ROS2 are converted to ROS1 format and transmitted via a TCP connection over the LAN to the robot. The web application frontend is accessible from both the operator's mobile phone and the TIAGo's onboard touchscreen, providing users with flexibility in interacting with the system.

This integrated architecture demonstrates the advantages of combining ROS and Temoto in the NL HRI, enhancing both deployment modularity and system compatibility across various robots and devices.

5.3.1.3 Environment Configuration

The physical environment featured an underground parking lot that offered ample space for robot operation. The TIAGo robot created a map of the area using its SLAM capabilities. Additionally, coordinates for three distinct areas of interest (Area A, B, and C) were manually incorporated into the object map. The location of a suspicious bag was also included in the object map database. These predefined locations provided specific coordinates for the GetCoordinates action to target during the security inspection sequence. Prior to the demonstration, the system sent an empty frame to the display feed, allowing the operator to subscribe to the inspection feed and prevent loss of the initial inspection image.

5.3.2 Demonstration

The demonstration focuses on the deployment of a TIAGo robot for security monitoring where an operator dynamically accesses the NL HRI and is instructed to perform an inspection routine throughout the areas for any suspicious bags, these steps being shown in Figure 25.

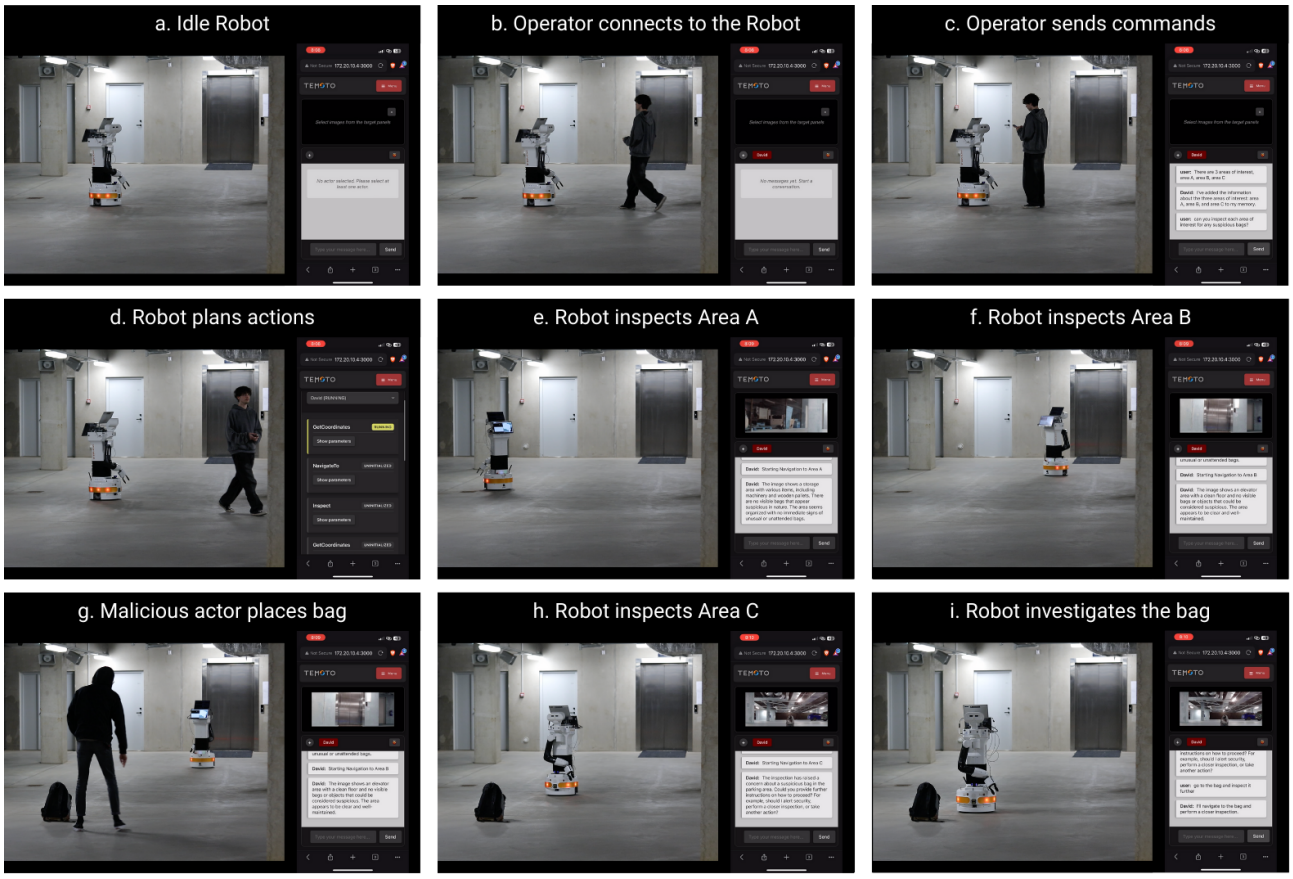


Figure 25: Tiago Demonstration

The demonstration proceeded through the following stages:

a. Idle Robot Initially, the TIAGo robot is in standby mode within the test environment. The robot's onboard systems are fully operational, with all sensors active and the natural language HRI interface prepared to receive commands.

b. Operator Connection The operator uses a mobile device to connect to the web application and registers the Actor's name to initiate a conversation with the NL HRI system.

c. Command Issuance The operator uses the natural language interface to inform the robot about three areas of interest: "There are three areas of interest: Area A, Area B, and Area C." The robot processes this input, adds these locations to its memory, and responds to the operator, "I've added the information about the three areas of interest: area A, area B, and area C to my memory." The operator then gives the robot investigation commands: "Can you inspect each area of interest for any suspicious bags?"

d. Action Planning Upon receiving the commands, the system generates an execution plan, successfully outlining nine actions to gather coordinates, navigate to, and inspect areas A, B, and C for any suspicious bags. Effectively demonstrating the system's ability to connect previously provided information to user requests and break down the sequence of actions into individual tasks. After generating the graph, the HRI confirmed the sequence generation to the operator and presented the planned sequence steps for their approval.

e. Area A Inspection The robot autonomously retrieves the coordinates for Area A and navigates there. Once it arrives, it inspects the area for any suspicious bags and thoroughly analyzes the environment, communicating to the operator that no suspicious bags are present in Area A and publishing the inspected image to the display panel.

f. Area B Inspection After completing the inspection of Area A, the robot moved on to Area B without needing any further commands, showcasing its ability to carry out a multi-step plan autonomously. Similarly, the inspection showed no bags in Area B, and the robot relayed these findings to the operator.

g. Malicious Actor Intervention While the robot was moving between inspection points, an unauthorized individual placed a suspicious bag in Area C.

h. Area C Inspection When the robot arrived at Area C and inspected the surroundings, it detected a suspicious bag, which triggered an automatic alert to the operator interface and halted all further actions until someone investigated the issue.

i. Suspicious Object Investigation The operator reviews the issue raised by the HRI and examines the displayed image. Following this, they issue a command in natural language: "Go to the bag and inspect it further." In response, the system generates additional actions within the current sequence, which include obtaining the coordinates, navigating to the bag, and performing a thorough inspection.

5.3.3 Discussion

The TIAGo demonstration successfully validated the Natural Language Human-Robot Interaction system in a physical security monitoring scenario. The system effectively interpreted area-based commands with contextual awareness, translating the inspection request into a coherent sequence of actions that covered all identified locations. The autonomous execution of this multi-step plan showcased the system's ability to maintain operational context throughout the workflow. Furthermore, the error-handling mechanism correctly detected the suspicious bag in Area C. The demonstration also confirmed the system's cross-platform compatibility by successfully implementing the ROS1-ROS2 bridge.

While this demonstration established the system's technical capabilities, it did not fully address the accessibility requirement for non-expert users. Testing with users unfamiliar with the system

in real operational contexts is necessary to evaluate this aspect comprehensively. This limitation is addressed in the subsequent Spot demonstration, where an external user with minimal prior exposure to the system could deploy and operate it independently, providing valuable insights into the accessibility of the Natural Language HRI interface.

5.4 Spot Demonstration

The Spot demonstration not only showcases the cross-platform capabilities of the NL HRI system but also offers valuable insights into the system's deployment capabilities by external users.

5.4.1 Setup

The NL HRI system was thoroughly documented and structured within a Git repository, enabling successful deployment by an external user with minimal guidance for the Spot Demonstration. Figure 26 illustrates the operator's setup configuration.

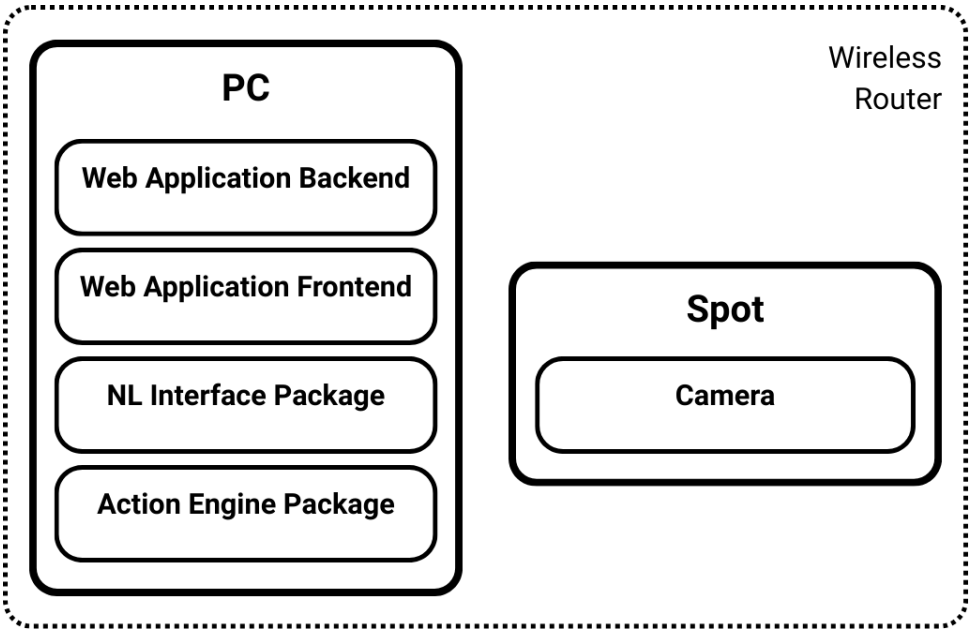


Figure 26: System deployment configuration for the Spot demonstration

5.4.1.1 Hardware Configuration

The hardware configuration focuses on deploying Boston Dynamics' Spot robot, which features quadrupedal locomotion and is equipped with 360° stereo cameras and a depth camera for environmental perception and Navigation. The system uses a wireless network architecture, with a laptop connecting to Spot via a wireless router. This laptop hosts the NL HRI system and maintains external connectivity to the LLM API provider, allowing for complete wireless operation throughout the warehouse environment without any cabling constraints.

5.4.1.2 Software Architecture

For the Spot demonstration, the software architecture uses a PC hosting the complete Web Application, Natural Language Interface Package, and Action Engine under "David." Motor commands were directly conveyed to the Robot through the ROS2 configuration using Nav2 for motor control.

5.4.1.3 Environment Configuration

An external operator performed the Spot demonstration in a complex workshop setting with multiple obstacles. Before the demonstration, the robot generated the environment map using Spot's SLAM capabilities, and coordinates for the loading area, storage areas, and charging dock were manually added to the object map JSON. The external user installed the entire NL HRI system with minimal assistance to ensure all packages were up-to-date and files were configured correctly. The actual demonstration proceeded without any oversight or assistance. Finally, similar to the TIAGo demonstration, the system sent an empty frame to the display feed to initialize the subscription before the demonstration.

5.4.2 Demonstration

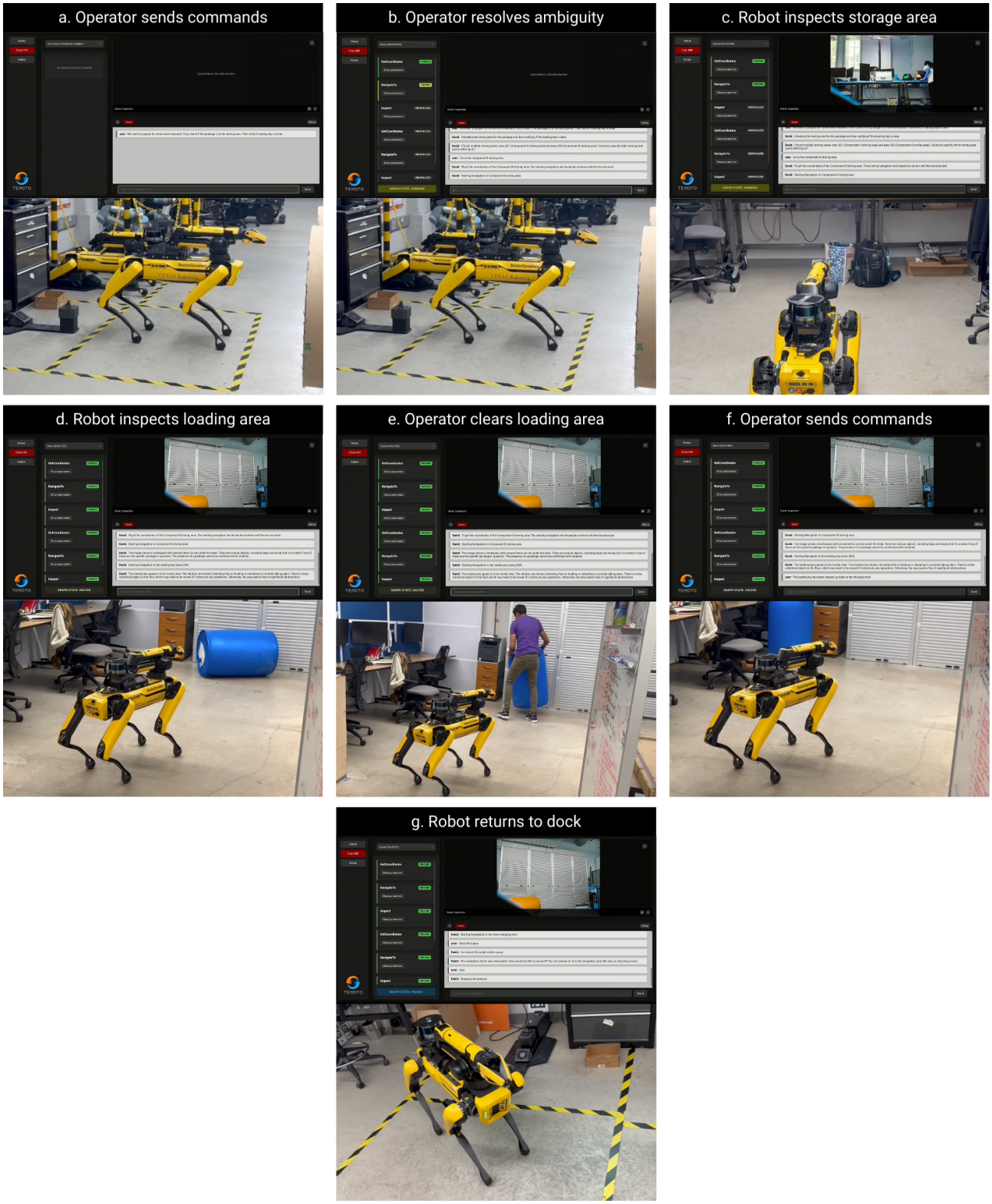


Figure 27: Spot demonstration

The demonstration proceeded through the following stages:

a. Operator Sends Commands Initially, the operator uses the natural language interface to instruct Spot: "We need to prepare for tomorrow's shipment. First, check if the package

is in the storage area. Then verify if the loading area is clear." The system processes this command and generates a comprehensive action plan involving multiple coordinate, navigation, and inspection tasks.

b. Operator Resolves Ambiguity During the GetCoordinate action, the system throws an error due to the package's location ambiguity. The system informs: "I found multiple storing areas: area_001 (Component A storing area) and area_002 (component B storing area). Could you specify which storing area you are referring to?" The operator resolves this ambiguity by commanding the robot to the storing area of component B.

c. Robot Inspects Storage Area Spot navigates autonomously to the storage area and performs a visual inspection using its onboard cameras. The system processes the camera feed, identifies the packages in this area, and proceeds to the loading area.

d. Robot Inspects Loading Area After inspecting the storage area, Spot moves to the loading area. During this inspection, the system notifies the operator about a "blue cylinder object on the floor, which may need to be moved if it obstructs any operation."

e. Operator Clears Loading Area After being notified about the blue cylinder obstructing the loading area, the operator goes to the area and removes the cylinder that had fallen over.

f. Operator Sends Additional Commands The operator issues a follow-up command: "The loading area has been cleared. Please return to the charging dock."

g. Robot Returns to Dock Spot recognizes the new information about the environment, integrates it into memory, and navigates back to its charging station, successfully completing the Spot demonstration workflow.

5.4.3 Discussion

The Spot demonstration successfully validated the system's cross-platform capabilities and ease of deployment, with an external user independently implementing the NL HRI system following minimal guidance. The NL interface effectively processed the Operator's commands and resolved ambiguities about storage locations, while the robot successfully navigated through a complex environment and performed inspection tasks. These achievements confirm the system meets its core requirements for accessibility, platform-agnostic operation, and intuitive natural language control in real-world applications.

Despite the overall success, several operational issues emerged during the demonstration. One significant problem was that the inspection action failed to correctly identify the blue cylinder as an obstacle that required immediate attention. Although the response mentioned the cylinder's presence, its position at the periphery of the loading area led the system not to classify it as an obstruction. This misclassification stemmed primarily from the operator's environmental configuration rather than algorithmic limitations.

Additionally, a synchronization issue occurred during the "return to dock" operation. There was a discrepancy between the Action Engine state and the NL HRI system's graph state. This issue stemmed from the NavigateTo unsuccessful action termination, which led to a mismatch between the graph state in the NL HRI and the Action Engine. Once the graph was cleared, the navigation action was terminated with an error, where the Action Engine feedback triggered error handling on a non-existent graph to address the issue.

Furthermore, the user experienced a slight confusion when manually adding actors to the system with no assistance, underlining the need for a more streamlined actor discovery process in future iterations.

Nonetheless, the spot demonstration effectively showcased the system's ability to provide platform-agnostic operation and intuitive control for non-expert users in practical warehouse applications.

Part VI

Discussion

6.1 Contributions

The Natural Language Human-Robot Interaction system successfully met all seven requirements outlined in Part III, demonstrating its effectiveness across various operational contexts. The system achieved natural language fluency by allowing complete robot control through conversational interaction while maintaining contextual awareness during multi-step tasks. Robust error handling was a cornerstone of the system's reliability, with a multi-stage resolution strategy effectively detecting and resolving ambiguities using natural language descriptions rather than technical jargon. Throughout all demonstrations, from the TIAGo inspection sequence to the Spot shipment preparation, the system maintained operational continuity by preserving conversation history and appropriately applying the previous context.

The accessibility requirement was validated through external user deployment on Spot, where a user with minimal prior experience successfully implemented the system using standard documentation. The web-based interface provided intuitive operation across devices, eliminating the learning curve associated with traditional robotic interfaces. Operational transparency was maintained through real-time visualization in the Planned Action Panel and visual feeds through the Display Interface, allowing operators to understand and monitor robot behaviors effectively without requiring technical expertise.

The system's modular ROS2 architecture satisfied both modularity and cross-platform compatibility requirements through its component-based design. Successful deployments across three diverse platforms—Turtlebot3, TIAGo, and Spot—demonstrated the framework's hardware adaptability without requiring platform-specific modifications. By integrating with TeMoto's Action Engine, the system leveraged established control frameworks while extending them with natural language capabilities, creating a robust execution pipeline that translated conversational inputs into precise robotic actions across all tested platforms.

These accomplishments clearly show that the research objectives were met successfully. The system effectively connects natural language commands with robotic actions while ensuring reliability, modularity, and accessibility, essential for practical implementation across various applications.

6.2 Limitations and Future Work

6.2.1 Temoto Integration

The current HRI provides an additional command source level to the Temoto framework for dynamically generating and planning UMRF graphs. However, there are still limitations to the NL Interface-Action Engine package comparability that must be completed to ensure seamless integration.

The NL Interface package can monitor the UMRF graph feedback to track the progress of graph execution within the Action Engine. However, the Action Engine cannot currently provide information regarding its state or any errors that may arise beyond the individual actions execution. When structural issues occur, such as missing input or output parameters resulting from LLM generation errors, the Action Engine silently rejects the graph without providing

diagnostic information. This results in an information gap between the Action Engine's internal execution state and the NL Interface system. Operators must manually check the Action Engine terminal and clear the execution queue to restart the entire process if graph rejection occurs.

Furthermore, the external operator experienced difficulties adding actors to the Chat Panel, highlighting a usability gap in the interface design. Although basic guidance resolved the immediate issue, an optimal HRI system should eliminate all operational confusion regardless of user expertise level. To enhance interaction fluidity, future implementations should automatically detect and register actors launched through the Action Engine on the network, dynamically populating the actor selection tabs without requiring manual configuration steps.

6.2.2 UMRF vs Behavior Trees

The UMRF graph serves as the task description format for describing the execution sequences for the Temoto system. Its JSON-based structure facilitates easier debugging through human-readable syntax and streamlined programmatic parsing. However, Behavior Trees present a compelling alternative with widespread industry adoption and abundant online resources, potentially enhancing LLM-based generation capabilities. While UMRF graphs contribute to a more cohesive Temoto framework ecosystem, this specialization may limit compatibility opportunities with external systems and tools that have standardized around Behavior Trees.

6.2.3 Robot Fleet Capabilities

Regarding the capabilities of the robot fleet, the NL HRI system allows operators to manage multiple robots. However, the various graphs generated are planned and executed individually for each robot. The NL Interface package does not share information between the different robots, which can lead to issues when two robots need to coordinate to complete a task. If an error occurs on one robot due to the failure of another, the Error Handler may not be able to resolve the tasks properly. In addition to the individual UMRF graphs for each robot, a higher-level task manager should be developed in UMRF Planner to coordinate the execution of these graphs throughout the robot fleet.

6.2.4 Speech Interface

Building on the successful implementation of natural language processing, the NL HRI system can be enhanced with a speech interface, allowing hands-free operation using verbal commands. This enhancement would involve integrating speech recognition capabilities into the web application and converting spoken instructions into text that the NL Interface package can process. Additionally, a text-to-speech synthesis system would convert the NL HRI's text responses into spoken feedback. This would create a complete voice interaction interface that closely mimics natural human conversation patterns.

6.2.5 Context and LLM Processing

The current implementation hardcodes action definitions and usage examples within static instruction messages are used by both the Chat Interface and Error Handler nodes for graph generation. To enhance flexibility and maintainability, an abstraction layer should be implemented that dynamically compiles instruction context from action definitions at runtime. This proposed system component would query the TeMoto Action Engine network to automatically discover and retrieve all registered actions with their corresponding usage patterns and parameters. Consequently, the operator would no longer need to update the instructions within llm_core repository to define the system's action sequence generation capabilities.

To extend the NL HRI capabilities in TeMoto, the system should implement a persistent storage mechanism for successfully executed UMRF graphs. This functionality would allow the NL HRI package to act as a tool for generating action sequences, which operators could further refine using the TeMoto Action Editor. These stored graphs would also create a dynamic repository of examples, improving the system's LLM instructions and leading to more cohesive and accurate action sequence generation.

Lastly, the system should be evaluated across various LLMs to measure performance differences. As language model technology advances, the NL HRI system will gain reliability and responsiveness, while smaller models may enable contained execution without external API dependencies.

Part VII

Conclusion

This thesis successfully demonstrates the development and validation of the Natural Language Human-Robot Interaction system that bridges the semantic gap between human communication and robotic execution. By leveraging Large Language Models with the established TeMoto framework, the system addresses critical limitations in current robotic control interfaces while maintaining the reliability and modularity essential for practical deployment.

The modular architecture, comprising specialized components for chat management, action planning, error handling, memory management, and visual feedback, creates a robust pipeline that transforms conversational commands into precise robotic actions. The system's effectiveness was validated through demonstrations across three distinct platforms—Turtlebot3, TIAGo, and Spot, confirming its adherence to the established requirements and cross-platform adaptability.

While certain limitations remain, such as the need for better integration with TeMoto's resource management and enhanced multi-robot coordination capabilities, the system represents a significant step toward democratizing robot control through natural language interaction. The modular design ensures that these limitations can be addressed through targeted improvements without requiring fundamental architectural changes.

This work contributes to the field of Human-Robot Interaction by providing a practical solution that combines the accessibility of natural language interfaces with the reliability of established robotic frameworks. As Large Language Models continue to advance, this architecture provides a stable foundation for even more sophisticated natural language robot control systems, potentially transforming how humans interact with robots across various domains.

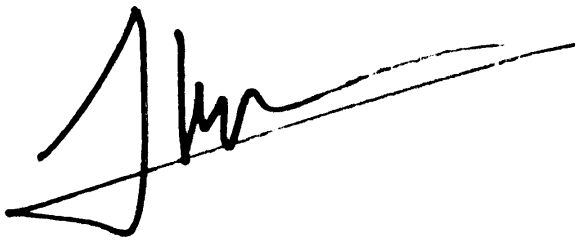
Acknowledgments

I want to express my sincere gratitude to my supervisors, Robert Valner and Karl Kruusamäe, for their invaluable guidance throughout the development of my thesis. Their regular feedback and suggestions were instrumental in organizing and finalizing the robust robotics system. Karl Kruusamäe's contributions were vital in project initiation and organizing my thesis for maximum clarity. Robert Valner's expertise in robotic capabilities and ongoing Action Engine development enabled seamless integration of the NL Interface package into the TeMoto System, culminating in a fully realized final system. This comprehensive support enabled me to achieve all my research objectives.

I extend special thanks to the team members who made critical technical contributions possible. Fabian Parra was instrumental in deploying the Boston Dynamics Spot robot, which proved vital for platform testing while assisting with TeMoto actions and robot deployment. Kaarel Kaarelson contributed to the Web Application, ensuring the TeMoto system's robustness and project continuity. I'm also grateful to the broader TeMoto team for providing the solid foundation of the Action Engine and UMRF Graphs framework.

The institutional support I received was equally crucial to this work's success. The University of Tartu provided access to the TIAGo robot, enabling physical validation beyond theoretical proof, and graciously provided a laptop to support my development work. I'm also grateful to ECAM LaSalle for enabling my participation in this double diploma program with the University of Tartu and to Guillaume Gibert for equipping me with the essential robotics foundation that made this thesis possible.

Finally, I wish to acknowledge the technological tools that greatly enhanced the quality of this work. Grammarly played a crucial role in refining the linguistic structure of this thesis, helping me create coherent and precise sentences throughout the document. I benefited from Claude AI's feedback in organizing ideas throughout my thesis document and refining my code architecture. Additionally, OpenAI's development environment and comprehensive API documentation provided a solid foundation for the technical implementation, ensuring a reliable process for converting natural language commands into structured sequences.



References

[1] Robert Valner, Selma Wanna, Karl Kruusamäe, and Mitch Pryor. Unified meaning representation format (umrf) - a task description and execution formalism for hri. *J. Hum.-Robot Interact.*, 11(4), September 2022.

[2] Roohollah Jahanmahin, Sara Masoud, Jeremy Rickli, and Ana Djuric. Human-robot interactions in manufacturing: A survey of human behavior modeling. *Robotics and Computer-Integrated Manufacturing*, 78:102404, 2022.

[3] Malak Qbilat, Ana Iglesias, and Tony Belpaeme. A proposal of accessibility guidelines for human-robot interaction. *Electronics*, 10(5), 2021.

[4] Induni N Weerarathna, David Raymond, and Anurag Luharia. Human-robot collaboration for healthcare: A narrative review. *Cureus*, 15(11):e49210, Nov 2023.

[5] Uqba Othman and Erfu Yang. Human–robot collaborations in smart manufacturing environments: Review and outlook. *Sensors*, 23(12), 2023.

[6] Jeffrey Delmerico, Stefano Mintchev, Alessandro Giusti, Boris Gromov, Kamilo Melo, Tomislav Horvat, Cesar Cadena, Marco Hutter, Auke Ijspeert, Dario Floreano, Luca M. Gambardella, Roland Siegwart, and Davide Scaramuzza. The current state and future outlook of rescue robotics. *Journal of Field Robotics*, 36(7):1171–1191, August 2019.

[7] Ida Skubis, Agata Mesjasz-Lech, and Joanna Nowakowska-Grunt. Humanoid robots in tourism and hospitality—exploring managerial, ethical, and societal challenges. *Applied Sciences*, 14(24), 2024.

[8] Josip Tomo Licardo, Mihael Domjan, and Tihomir Orehovački. Intelligent robotics—a systematic review of emerging technologies and trends. *Electronics*, 13(3), 2024.

[9] Robert Valner, Veiko Vunder, Alvo Aabloo, Mitch Pryor, and Karl Kruusamäe. Temoto: A software framework for adaptive and dependable robotic autonomy with dynamic resource management. *IEEE Access*, 10:51889–51907, 2022.

[10] Robert Valner, Selma Wanna, Karl Kruusamäe, and Mitch Pryor. TeMoto: A Software Framework Supporting Mixed-Modality Command Inputs Necessary for Integration of nonverbal-HRI. In *Workshop on Exploring Applications for Autonomous Non-Verbal Human-Robot Interactions, ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, Virtual Conference, March 2021. ACM/IEEE. March 8th, 2021.

[11] Robert Valner, Veiko Vunder, Andrew Zelenak, Mitch Pryor, Alvo Aabloo, and Karl Kruusamäe. Intuitive 'human-on-the-loop' interface for tele-operating remote mobile manipulator robots. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, Madrid, Spain, June 2018. ESA.

[12] Harun Ersöz. Fanuc robot eğitimi için celalettin Çakın. LinkedIn post, 2024. Accessed: May 20, 2025.

[13] Hidetoshi Fukui, Satoshi Yonejima, Masatake Yamano, Masao Dohi, Mariko Yamada, and Tomonori Nishiki. Development of teaching pendant optimized for robot application. In *2009 IEEE Workshop on Advanced Robotics and its Social Impacts*, pages 72–77, 2009.

[14] Buys Galore. irobot packbot mtrs 510 mk1 multi-mission robot (mtrs) - talon 4 alternative. Product listing. Accessed: May 20, 2025.

[15] Theresa Pekarek Rosin, Vanessa Hassouna, Xiaowen Sun, Luca Krohm, Henri-Leon Kordt, Michael Beetz, and Stefan Wermter. A framework for adapting human-robot interaction to diverse user groups, 2024.

[16] Wil Thomason and Ross Knepper. Recognizing unfamiliar gestures for human-robot interaction through zero-shot learning. pages 841–852, 03 2017.

[17] Ruoyu Wang, Zhipeng Yang, Zinan Zhao, Xinyan Tong, Zhi Hong, and Kun Qian. Llm-based robot task planning with exceptional handling for general purpose service robots, 2024.

[18] Artem Lykov and Dzmitry Tsetserukou. Llm-brain: Ai-driven fast generation of robot behaviour tree based on large language model, 2023.

[19] Shyam Sundar Kannan, Vishnunandan L. N. Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models, 2024.

[20] Jingkai Sun, Qiang Zhang, Yiqun Duan, Xiaoyang Jiang, Chong Cheng, and Renjing Xu. Prompt, plan, perform: Llm-based humanoid control via quantized imitation learning, 2024.

[21] Ulas Berk Karli, Juo-Tung Chen, Victor Nikhil Antony, and Chien-Ming Huang. Alchemist: Llm-aided end-user development of robot applications. In *Proceedings of the 2024 ACM/IEEE International Conference on Human-Robot Interaction*, HRI '24, page 361–370, New York, NY, USA, 2024. Association for Computing Machinery.

[22] Haokun Liu, Yaonan Zhu, Kenji Kato, Izumi Kondo, Tadayoshi Aoyama, and Yasuhisa Hasegawa. Llm-based human-robot collaboration framework for manipulation tasks, 2023.

Appendix

Project Repositories

The **NL HRI** can be found at: `https://github.com/julianleclerc/TeMoto-NLHRI`

The individual package repos are:

- **Temoto Action Engine**: `https://github.com/temoto-framework/temoto_action_engine_ros2`
- **Web Application**: `https://github.com/temoto-framework/temoto_action_assistant`
- **NL Interface Package**: `https://github.com/julianleclerc/temoto_nl_interface`
- **Demo Resources**: `https://github.com/julianleclerc/temoto_demo_resources`

Supplementary Materials

High-resolution images and demonstration videos for this thesis can be found at:

- **Thesis Archive**: `https://github.com/julianleclerc/thesis-nlhri-archive`

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Julian Rene Leclerc

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   "Natural Language Human-Robot Interaction: A Modular Framework for Conversational Robot Control using Large Language Models"

   supervised by Robert Valner and Karl Kruusamäe

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Julian R. Leclerc
20.05.2025