Universität Osnabrück
Institut für Informatik
AG Technische Informatik
Prof. Dr.-Ing. Mario Porrmann

Master's Thesis

Mixed Reality Environment for Robot-in-the-Loop-Testing using Digital Twins

Jonas Kittelmann

Betreuer: Prof. Dr.-Ing. Mario Porrmann
M.Sc. Philipp Gehricke

Osnabrück, 23 December 2025

M 00

Abstract

Hier sollte in einem Abstract kurz der Inhalt der Arbeit erläutert werden.
Zuerst auf deutsch.

Then an abstract of the thesis in english should follow.

Contents

1 Introduction

Robotic and Autonomous Systems (RAS) combine multiple disciplines, including control engineering and robotics, mechanical engineering, electronics, and software engineering. While traditional testing relies on domain-agnostic software metrics like code coverage and failure counts, the multi-multidisciplinarity of RAS requires a extension of these methods to verify the system's autonomous operation, continuous movement, and ability to handle uncertainty. For researchers and engineers specializing in software testing, adapting existing techniques for RAS presents a significant challenge. This is why there is extensive literature dedicated to proposing and evaluating different testing techniques and processes, showing how essential it is to establish structured methods for ensuring the reliability of these complex systems. [AMV23]

This validation challenge highlights a fundamental tension between simulation-only testing and full-scale physical experimentation, which are the two established methods in robotics evaluation. Simulation is central to the robotic development, offering a way to test control logic and experiment with system configurations before committing them to hardware [Hu05; Mic04]. Simulations provide a rapid prototyping environment that minimizes infrastructure costs and logistical difficulties [Mic04; DRC+17]. Furthermore, they utilize virtual time to execute computationally expensive algorithms significantly faster than real-time hardware, enabling efficient design exploration [Mic04]. A discontinuity often occurs during the transition to real hardware, where the models validated in simulation are discarded and the code is rewritten from scratch, introducing potential errors [Hu05]. Additionally, simulation environments rarely replicate the full complexity of the physical world, missing nuances such as sensor noise, surface friction, and the unique manufacturing variances found in individual motors [BM18]. Testing with real hardware allows for the highest fidelity and controls to be refined based on the presence of all the unpredictable physics and variability that simulations can't replicate [Hu05; CCC+21]. But this approach has its own drawbacks. Conducting physical experiments can be resource-intensive and time-consuming and require significant funds, manpower, and infrastructure [Hu05; DRC+17; Mic04]. Moreover, virtual environments are essential for validating scenarios that are hazardous or physically inaccessible in the real world, such as bomb disposal missions, space exploration, or pilot emergency training [BM18; Hu05]. Testing the entire system using only real components is also severely limited for complex, large cooperative systems because of issues relating to complexity and scale [Hu05]. As neither of these approaches seems satisfactory on its own, an intermediate solution is required that could connect simulation with real-world experimentation [Hu05].

To bridge this gap, hybrid methods like robot-in-the-loop (RitL) simulation have emerged, which allow physical robots to operate within virtual environments [Hu05]. This often centered on a Digital Twin, which is a real-time virtual replica of the physical robot [AA23] and employs Augmented Reality (AR) as a Mixed Reality medium to facilitate user interaction [MV20]. The combination of these technologies provides a robust paradigm for the comprehensive testing and validation of robotic systems.

The “Virtual Environment for mobile Robotic Applications” (VERA) framework integrates digital twins, AR, and vehicle-in-the-loop testing together into a single system [Geh24]. VERA provides a modular platform for creating, managing, and visualizing synchronized virtual environments, which are presented both in simulations and as real-world projections [Geh24]. This master’s thesis builds directly upon the foundation laid by this original framework.

While the VERA framework provides a strong conceptual foundation, its original technical implementation contained critical limitations regarding physical realism, scalability, and user interaction [Geh24]. Its custom environment manager lacked a realistic physics engine, while its 2D visualizer, built with Pygame, showed performance degradation when handling a large number of dynamic objects, leading to delayed and incomplete visualizations [Geh24]. Furthermore, while an AR interface was implemented, the original thesis identified immersive Virtual Reality (VR) interaction as a key area for future work [Geh24].

This thesis overcomes those limitations by offering a novel, hardware-accelerated architecture capable of real-time physics simulation replacing the custom manager and visualizer used within VERA. To address these needs, a game engine is selected to replace the custom solutions, specifically chosen to fulfill the requirements for high-fidelity graphics, integrated physics, and native support for VR. All communication and data synchronization between the physical robot and the simulation environment is carried out with the **Robot Operating System 2 (ROS 2)** [MFG+22], a middleware framework used within state-of-the-art robotics.

The core goal is to create a real-time digital twin [AA23] of the EMAROs robot [Geh24], integrating its complete model, live sensor data, and physical properties such as mass, inertia, and collision models. This digital twin serves as the foundation for robot-in-the-loop [Hu05] testing scenarios. Additionally, this project extends the original Framework [Geh24] to include Virtual Reality (VR) [EM21] and desktop interfaces for scenario modification, robot control, and data visualization. This also includes reimplementing and enhancing VERA's AR floor projection feature by using the selected engine's rendering tools to improve visual quality and system capabilities.

[**NOT FINAL**]The remainder of this thesis is structured as follows: after reviewing the basic concepts and technologies, such as RitL, Digital Twins, Mixed Reality, ROS 2, and several simulation engines in Chapter 2, an in-depth review of the original framework of VERA is presented, along with its limitations. In Chapter 3, the functional and non-functional requirements for the new system are identified. The architecture and implementation of the new framework are explained in detail in Chapter 4, with particular reference to the integration of **the game engine** with ROS 2, the development of the digital twin, and the mixed reality interaction system. Then, a quantitative evaluation regarding the performance of the system is presented along with a discussion of the results in Chapter 5. Finally, Chapter 6 summarizes the thesis contributions and gives an outlook on possible future research.[**NOT FINAL**]

2 Background and Related Work

Einführung in das kapitel [**NOT FINAL**]

2.1 Robot-in-the-Loop-Testing

The validation and verification of modern Robotic and Autonomous Systems (RAS) is a significant challenge due to their complexity [AMV23]. This is because they integrate software, mechanical, and electrical engineering all at once [AMV23]. A central problem is ensuring that the software and hardware work together seamlessly, especially when testing both simultaneously [AMV23]. The X-in-the-Loop (XitL) simulation paradigm addresses this challenge [AAR+19]. It provides a framework to integrate the flexibility of software simulation with the realism of physical experiments [AAR+19].

A foundational XitL method is Hardware-in-the-Loop HiL simulation, which is a technique for testing mechatronic systems [MTH22; AAR+19]. The main principle ofHiLis creating a closed loop between the real hardware that is being tested and a simulation that represents the rest of the system or its operational environment [MTH22]. This setup effectively simulates the hardware to respond as if it were operating in a real system, which allows for testing across a wide range of virtual scenarios [AAR+19]. The main reason for usingHiLis to shorten development cycles and prevent costly or dangerous failures by making exhaustive testing possible before the system is actually completely implemented [MTH22]. This is whyHiLis essential in industries like automotive, aerospace, and robotics, where real-world testing can be too expensive, dangerous, or even impossible [AAR+19].

Robot-in-the-Loop (RitL) [MTH22] simulation extends the Hardware-in-the-Loop HiL concept. Instead of just a component, the hardware under test is a complete robotic system, such as an uncrewed vehicle [MTH22]. RitL replaces components of an pure simulated setup with the actual robot, increasing the realism of testing [Hu05]. [**NOT FINAL**]As illustrated in Figure **??**[**NOT FINAL**], a typical RitL configuration has the robot's real actuators operating in the physical world, while while its sensors interact with a simulated environment instead [Hu05; MTH22]. This creates a hybrid setup where, for example, the robot might use virtual sensors to see objects in the simulation but use its real motors to move physically [Hu05]. To keep the physical and virtual worlds synchronized, the real robot often has a virtual counterpart in the simulation which state is updated as the physical robot acts and moves [Hu05].

Figure 2.1: [**NOT FINAL**]The real robot's actions affect its virtual counterpart, and the virtual environment provides sensor data back to the real robot.[**NOT FINAL**]

The main benefit of RitL is that it uses the dynamics of actual hardware for repeatable testing of high-level software, such as navigation algorithms [MTH22]. This method avoids the expense, complexity, and risk of full physical testbeds while being a secure and useful substitute [MTH22]. RitL becomes therefore an tool for safely evaluating system performance in the lab [Hu05; MTH22]. This is especially important when working on projects that are hard to replicate, such as large-scale robot swarms or Mars rover missions, or when safety is at risk [Hu05; MTH22].

The RitL paradigm has been widely applied to validate complex autonomous systems, particularly in the automotive field using Vehicle-in-the-Loop (VitL) testing. For example, the Dynamic Vehicle-in-the-Loop (DynViL) architecture integrates a real test vehicle with the high-fidelity CARLA simulator, which operates on the Unreal Engine [DSR+22]. As shown in Figure 2.2, this approach allows a vehicle on an empty track to be stimulated by sensor data from a virtual world [DSR+22]. This lets automated driving functions to be safely and reliably tested in situations that would be hazardous to physically replicate [DSR+22]. CARLA and the Unreal Engine are also used in a similar Vehicle-in-Virtual-Environment (VVE) method which establishes a closed loop in which the motion of the real vehicle is tracked and reflected in the virtual world, making it possible its control systems to respond to simulated events [CCG+23].

Figure 2.2: An example of a Vehicle-in-the-Loop (VitL) setup. A real car on a test track connected to a high-fidelity simulator like CARLA that generates virtual traffic and sensor data. [DSR+22]

These examples show a trend towards using game engine based simulators to evaluate autonomous vehicles. This method falls somewhere between physical testing, where safety and consistency can be problematic, and pure software simulation, which frequently lacks vehicle dynamics fidelity [CCG+23]. Some systems even stimulate the vehicle's actual sensors. For instance, the Radar Target Simulator (RTS) can feed artificial radar echoes from a virtual scene to the vehicle's actual radar sensor [DKK+21]. This allows for end-to-end validation of the entire perception and control pipeline [DKK+21].

The X-in-the-Loop approach is found in many different areas, not just a single field. In marine robotics, a VIL framework was developed to test the long term autonomy of a robot swarm. In this system, an Autonomous Surface Vehicle (ASV) interacts with multiple simulated underwater sensor nodes to test cooperative behaviors without the logistical cost of deploying a full swarm [BVM20]. In the aerospace domain, the

RFlySim platform uses an FPGA-basedHiLsystem to create a high fidelity simulation for testing UAV autopilot systems in a lab, which reduces the need for expensive and risky outdoor flights [DKQ+21].

These approaches continue to move toward deeper integration. The Scenario-in-the-Loop (SciL) paradigm creates a mixed reality that blurs the boundary between physical testing and virtual simulation [VTS21]. Unlike standard hardware-in-the-loop methods, SciL allows the simulation to actively drive the physical environment [HLT+19]. A central software system uses virtual triggers to physically actuate real-world infrastructure, such as moving robotic dummies or changing traffic signals [HLT+19]. Simultaneously, the system manipulates the vehicle's perception using augmented reality to inject synthetic data directly into the vehicle's communication network alongside physical sensor readings [VTS21]. As a result, the vehicle cannot distinguish between physical objects and virtual ones [VTS21]. This indistinguishable perception compels the vehicle to execute real control maneuvers, such as emergency braking, in response to purely digital stimuli [VTS21].

2.2 Digital Twins

The Digital Twin (DT) is an important concept in many current X-in-the-Loop frameworks. The DT serves as the virtual counterpart to a physical system, acting as a virtual copy that allows real-time monitoring and simulation [AA23]. The idea is to create a digital information model of a physical system that stays linked to it for the duration of its lifecycle [GV17].

This concept was initially known as the Information Mirroring Model and consists of three core elements: the physical product, its corresponding virtual model, and the data connection that connects them [GV17; AA23]. A diagram of this structure is shown in Figure 2.3. The virtual model extends beyond a representation of physical dimensions, it is often a detailed simulation that can model mechanical, electrical, and software properties of a system [LWS+21]. Information moves in both directions, so sensor data can flow from the real world to update the virtual model [GV17; LWS+21]. In turn, the virtual model can also send commands back to control or optimize the physical system [GV17; LWS+21]. This continuous, bidirectional data exchange characterizes a Digital Twin [AA23].

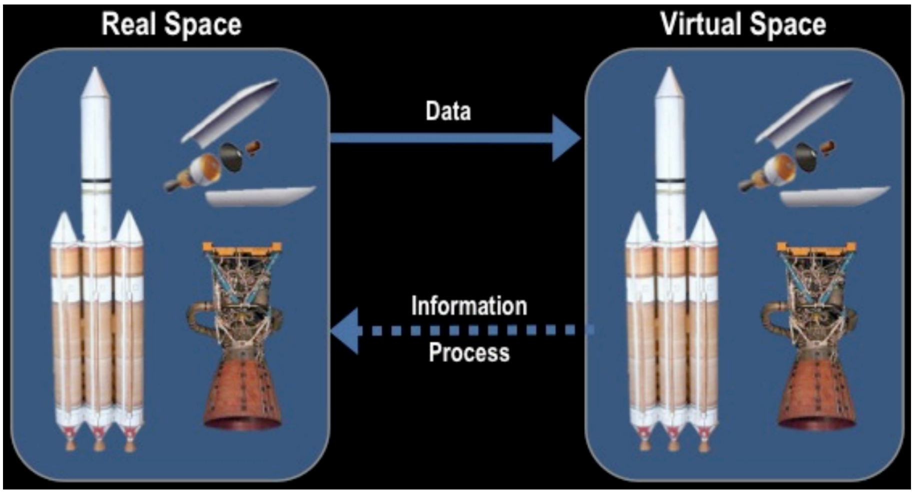

Figure 2.3: The foundational concept of a Digital Twin, illustrating the three core components: the physical entity, the virtual model, and the bi-directional data connection that links them [GV17; LWS+21]. [Gri15]

Digital Twins in robotics are frequently created using simulation software and the middleware that connects the virtual simulation to the actual hardware is often the Robot Operating System (ROS) [MFG+22]. Stączek et al. used the Gazebo simulator with ROS to create a DT of a factory floor, which they used to test and optimize the navigation algorithms of a mobile robot in narrow corridors [SPD+21]. To validate a deep learning-based harvesting robot, R. Singh et al. adopted a similar strategy and created a DT of a greenhouse in Gazebo [BSH24]. Other simulators like CoppeliaSim and Webots are also common. Magrin et al. used CoppeliaSim and ROS to create a DT as a learning tool for mobile robot design [MCT21], while Marques et al. used Webots for an Automated Guided Vehicle (AGV), synchronizing it via an OPC-UA server [MRS24]. These examples show a common approach of using simulators to simulate robot kinematics and sensor feedback for closed-loop testing.

More recently, game engines have become a popular choice for creating Digital Twins, as they offer more realistic graphics and physics. Unity [Uni23], for instance, is used to build highly realistic virtual environments, as seen in Figure **??**. Pérez et al. utilized Unity to develop a DT of a robotic cell, which is a manufacturing station where robots, conveyors, and safety systems work together to perform tasks [PRR+20]. In this framework, the Digital Twin acts as a live mirror of the physical cell [PRR+20]. The system uses the game engine to simulate physical laws and mechanical behaviors, ensuring that the virtual robots act exactly like the real ones [PRR+20]. To achieve this synchronization, the virtual model connects directly to the facility's control network to read real-time data from the sensors [PRR+20]. This connectivity allows engineers to virtually validate the cell's design and logic before any equipment is installed, and subsequently enables operators to train safely alongside the virtual robots [PRR+20].

Figure 2.4: [**NOT FINAL**]A comparison of robotic Digital Twin environments, showing a view from a traditional robotics simulator (left) versus a high-fidelity visualization from a modern game engine like Unity (right).[**NOT FINAL**]

Another important consideration are the technical capabilities of these engines. Yang et al. [YMZ20] simulated the physics of an UAV using Unity and its built-in NVIDIA PhysX engine. They also demonstrated the creation of virtual sensors, such as a LiDAR, directly within the game engine using its raycasting API [YMZ20]. Research has also been done on the performance and reliability of these game engine-based frameworks. Kwon et al. developed a safety-critical Digital Twin integrated with Unity and ROS 2, which achieves minimal data transmission latency to enable the prediction of collisions in real-time [KYS+25].

2.3 Mixed Reality

Beyond the testing paradigm and the digital twin, the way a user interacts with the system is another part of a robotics framework. Virtual, Augmented, and Mixed Reality (VAM) have become promising technologies for improving the information exchange between humans and robots [WPC+22]. In robotics, Augmented Reality (AR) is an especially useful tool for enhancing Human-Robot Interaction (HRI) by integrating virtual objects into a real-world environment in real-time [MV20].

The relationship between these technologies is formally described by the foundational Reality-Virtuality (RV) Continuum concept, first introduced by Milgram and Kishino [MK94]. As illustrated in Figure 2.5, this continuum is a scale that is anchored by a purely real environment at one end and a completely virtual one at the other [MK94; SSW21; MV20].

Figure 2.5: The Reality-Virtuality Continuum, illustrating the spectrum from a real environment to a completely virtual one. [WPC+22]

A Virtual Reality (VR) environment is an endpoint of the continuum, where the user is totally immersed in and can interact with a fully synthetic world [MK94]. This approach is useful for HRI research, as it allows for testing interactions with virtual robots in scenarios where it might be unsafe or too expensive for physical hardware [WPC+22]. The general term Mixed Reality (MR) describes the entire spectrum between the two extremes, where real and virtual worlds are combined into a single display [MK94]. MR is made up of two main categories: Augmented Reality (AR) and Augmented Virtuality (AV) [MK94; MV20]. AR is the process of adding virtual objects to a real environment [MK94]. This lets for example, HRI researchers to place 3D data and intentions of a robot directly into the physical space of an user [WPC+22]. The opposite is Augmented Virtuality (AV), where a primarily virtual world is enhanced with elements from the real world, like live video feeds [MK94].

2.4 Robot Operating System 2 (ROS 2)

ROS 2 is not a operating system but a middleware framework to simplify the creation of complex robotic systems [MFG+22]. It was redesigned from scratch to meet the challenges posed by modern robotics in domains ranging from logistics and agriculture to space missions and became the standard for both research and industry [MML+23; MFG+22]. At its core, ROS 2 is designed based on principles of distribution, abstraction, asynchrony, and modularity that together allow the development of scalable and robust applications [MFG+22]. The architecture of ROS 2 is based on a distributed network of independent programs called **Nodes** [MFG+22]. A node is a single, self-contained executable that performs a specific task, such as controlling a motor, processing sensor data, or, in the case of this thesis, bridging communication between robots and the simulation [MFG+22]. The nodes communicate through a set of defined patterns. The most common pattern is the publish-subscribe mechanism called **Topics**, illustrated in Figure 2.6 [MFG+22]. In this model, nodes can publish data as messages to a topic, while other nodes can subscribe to that topic to receive data asynchronously [MFG+22]. For tasks that require a direct request and a guaranteed response, ROS 2 offers **Services**, following a synchronous request-response pattern [MFG+22]. For long-running tasks that require continuous feedback and the ability to be preempted, ROS 2 provides a unique communication pattern called **Actions** [MFG+22]. An action consists of a goal, a feedback stream, and a final result, making it optimal for managing tasks like navigation where the progress of a robot toward a goal has to be monitored over time [MMW+20].

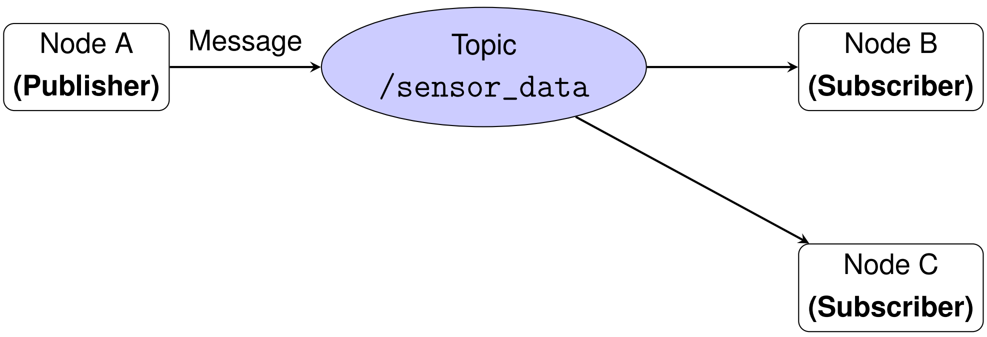

Figure 2.6: The ROS 2 publish-subscribe model. Node A publishes messages onto a central topic, and any number of subscriber nodes (B and C) may receive that data without direct knowledge of the publisher.

An essential part of any mobile robot is coordinate frame management, which in ROS 2 is accomplished through its transform library, **tf2** [Foo13]. The tf2 library standardizes the tracking of spatial relationships between the various parts of the robot and environment, placing them into a tree-like data structure, as illustrated in Figure 2.7 [Foo13]. This permits any node in the system to request at any time the position and orientation of any frame relative to another, a feature critical for transforming sensor data into a useful frame of reference [Foo13].

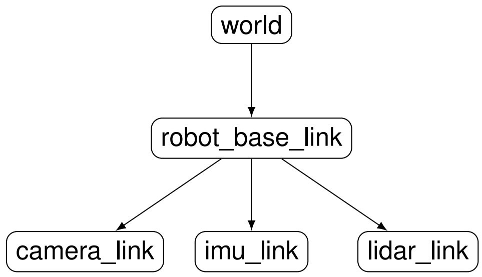

Figure 2.7: A simplified example of a ROS 2 transform tree (tf tree). The library maintains the hierarchical relationships so that a program can easily determine the transform from the `camera_link` to the `world` frame, for instance.

In the architecture of this thesis, ROS 2 provides the central **data backbone**, a term borrowed from digital engineering that defines an integrated communication layer for all relevant system knowledge [PKP+20]. This acts as the bridge that couples the physical EMAROs robot and its tracking system with the simulation environment. The digital twin of the EMAROs robot subscribes to ROS 2 topics to receive real-time data from the physical world and can synchronize its state through this communication layer [SKG+24]. For navigation for both the physical and virtual robots, this thesis utilizes **Navigation2 (Nav2)**, the official, next-generation autonomous navigation framework for ROS 2 [MMW+20]. Nav2 was designed from the ground up to orchestrate planning, control, and recovery tasks using configurable Behavior Trees that are highly modular and can be changed at runtime to create custom navigation behaviors [MMW+20]. Core to its operation, Nav2 separates the navigation task into two closely related, yet distinct components, namely a **global planner**, which seeks out an acceptable, long-range path through the environment, and a **local trajectory planner** or controller, which generates velocity commands in order to follow that route while reacting to immediate obstacles [MML+23]. Using the Nav2 stack, a high-level goal, e.g. a coordinate in the environment, can be sent via a ROS 2 action to a robot and the system will take care of all complex path planning and collision avoidance autonomously.

2.5 The VERA Framework

This thesis directly builds on the Virtual Environment for mobile Robotic Applications framework, which was developed earlier by Gehricke [Geh24]. VERA was designed initially as a modular and scalable platform to bridge the validation gap between pure software simulation and real-world testing. It combines the concepts Digital Twins, Augmented Reality, and Vehicle-in-the-Loop testing to enable robots to interact with a virtual environment projected into the real world [Geh24].

The core idea of VERA is projecting a dynamic, interactive virtual world onto the physical floor where a real robot operates. The system synchronizes the virtual state with the actions of the physical robot in such a way that scenarios are possible where the robot can detect and react to virtual obstacles as if they were real. This system provides a flexible testbed for the development and evaluation of robotic applications without having to physically build complex environments [Geh24].

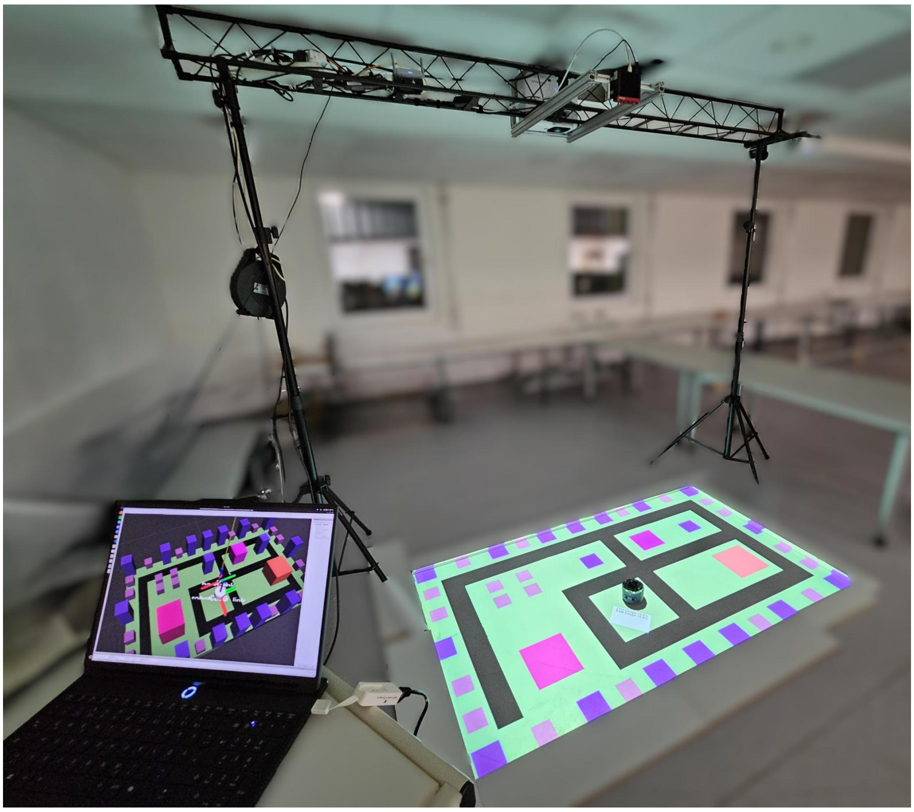

Figure 2.8: The physical setup of the VERA platform: A projector and tracking system are mounted on a frame above the test area. [Geh24]

2.5.1 EMAROs Test Platform

The capabilities of VERA are demonstrated by a mobile mini-robot called EMAROs (Educational Modular Autonomous Robot Osnabrück), shown in Figure 2.9, which was developed at Osnabrück University specifically for research and education in robotics and artificial intelligence [Geh24]. The modular hardware architecture of EMAROs is based on three main PCBs: a **Host-PCB** containing high-level computing, a **Power-PCB** managing energy supply and motor control and a **Base-PCB** integrating ground sensors [Geh24]. The specific variant used in this work is equipped with a Raspberry Pi Compute Module 4 (CM4) running Ubuntu 24.04 and the Robot Operating System 2 (ROS 2) [Geh24].

Figure 2.9: The EMAROs robot is the robotic platform used in the testbed, equipped with a modular sensor suite and running ROS 2. [Geh24]

For obstacle avoidance, the robot is equipped with Time-of-Flight (ToF) distance sensors and an Inertial Measurement Unit (IMU) for orientation tracking [Geh24]. Moreover, there are two wide-angle cameras that can be configured for stereo vision applications or line-following tasks [Geh24]. Within VERA, EMAROs takes on the role of the robot, streaming real-time telemetry including odometry, battery status, and sensor data to the simulation framework via ROS 2 to create a digital twin [Geh24].

2.5.2 Original VERA System Architecture

The software architecure of the original VERA framework was mainly a ROS 2 implementation, and its structure included only three main components: the **Virtual Environment Positioning System (VEPS)**, the **Environment Manager**, and the **Visualization System** [Geh24].

Accurate localization is important to synchronize the physical robot with the projected virtual world and was provided by the **VEPS** in the original VERA framework [Geh24]. The original VEPS implementation relied on a depth-based approach where a ceiling-mounted Allied Vision Ruby 3D stereo camera captured point clouds of the test area [Geh24]. The point clouds were filtered by a Median Absolute Deviation (MAD) algorithm to calculate the centroid of the robot [Geh24]. However, for maximum robustness and precision for this thesis, the positioning system was updated to use an ArUco marker-based tracking system developed in parallel research by [**JuliaBA**], as shown in Figure **??**. Although the point-cloud method provided a markerless solution, the ArUco-based approach has higher reliability and lower noise [**JuliaBA**; Geh24]. By attaching a fiducial marker to the top of the EMAROs robot, the system is able to calculate the 6D pose (position and orientation) of the robot [**JuliaBA**]. This pose is published into the ROS 2 network, where it can instantly update the position of the digital twin [**JuliaBA**].

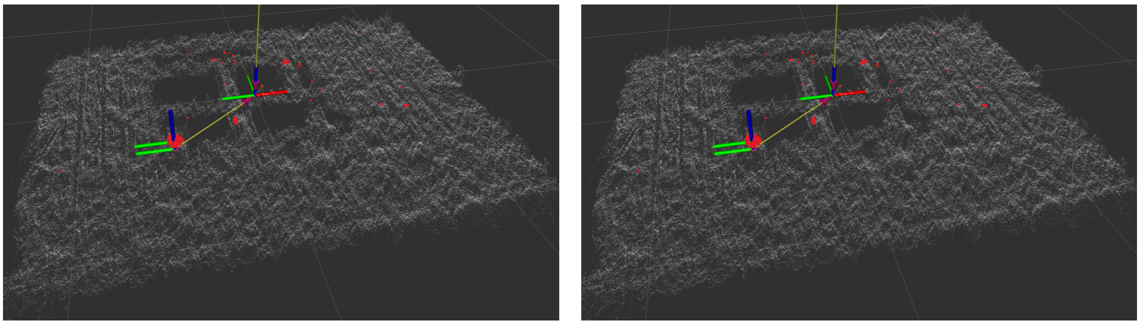

Figure 2.10: The original VERA used point clouds [Geh24], while the current iteration uses ArUco markers for pose estimation [**JuliaBA**].

The **Virtual Environment Manager** is the central brain of the simulation [Geh24]. Implemented as a C++ ROS 2 node, it parses environment definitions from standard Gazebo `world.sdf` files [Geh24]. It keeps track of the state of all virtual objects, including their positions and properties, such as whether an object is movable or static [Geh24]. One of the key roles of the manager is to manage the interactions of the robot with the virtual world [Geh24]. It operates a continuous loop, typically at 20 Hz, which checks the distance between the position of the robot given by the VEPS and the virtual objects [Geh24]. The manager triggers predefined actions if a collision is detected, such as removing a manipulatable object or stopping the robot [Geh24]. It relies on basic distance heuristics and not a physics engine, hence limiting interactions to simple ones [Geh24].

The **Visualization System** is responsible for rendering the virtual environment so it can be projected onto the physical floor [Geh24]. In the original VERA framework, this was achieved using a custom ROS 2 node and Pygame, a Python library designed for 2D game development [Geh24]. The visualizer subscribes to the object lists published by the Manager and the robot's path history [Geh24]. It provides a 2D, top-down, orthogonal view of the scene, drawing obstacles as simple geometric shapes [Geh24]. It gives Augmented Reality features, projecting dynamic information directly into the workspace [Geh24]. The driven path of the robot is displayed as a trail, while a system status panel with real-time CPU and battery usage is shown behind the robot, as illustrated in Figure 2.11 [Geh24].

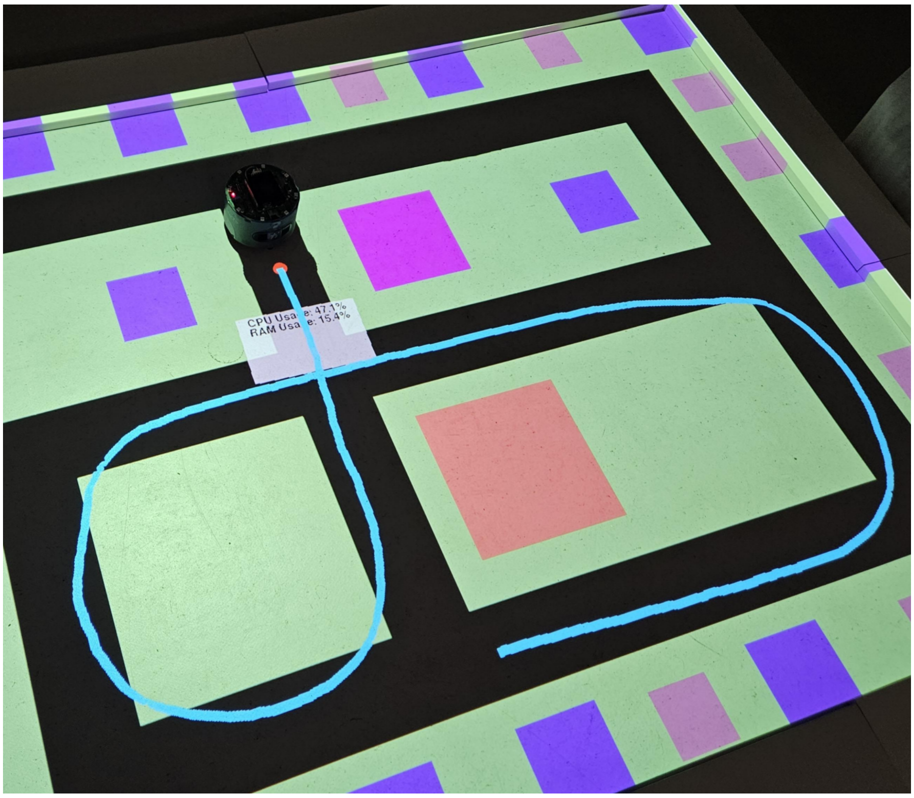

Figure 2.11: Original VERA visualization via Pygame, showing a projection of the robot's path and virtual obstacles onto the floor. [Geh24]

2.5.3 Identified Limitations

While VERA successfully demonstrated the concept of Robot-in-the-Loop testing with AR projections, its original software implementation contains several critical limitations, that this thesis sets out to overcome:

- **Lack of Realistic Physics:** The custom Environment Manager does not feature a physics engine [Geh24]. Collision detection is performed through simple radius checks, meaning the robot cannot push objects or experience friction nor can it interact with complex geometries [Geh24]. This reduces the realism of the Digital Twin [Geh24].
- **Performance Scalability:** Pygame-based visualizer utilizes CPU for rendering and has performance-related issues for complex scenarios [Geh24]. Gehricke [Geh24] noted that in an environments with a high number of dynamic objects (e.g., >800), the message queue became saturated, leading to considerable latency and incomplete visual updates.
- **Limited Interaction (No VR):** The system was designed strictly for 2D floor projections and Rviz [Geh24]. It does not support any kind of immersive Virtual Reality (VR) interfaces that would enable a human operator to view the scene in 3D or to interact with the robot remotely [Geh24]. Therefore, the integration of Virtual Reality was identified as a necessary future step for Human-Robot Interaction (HRI) [Geh24].

2.6 Simulation Engines for Robotics

The choice of a capable simulation engine is an important part in solving the limitations of the original VERA framework. The choice of platform affects the efficiency, accuracy, and applicability of the Digital Twin [SKG+25]. Based on this, the review of simulation engines in this thesis is guided by four criteria which are required for a functional Mixed Reality Digital Twin:

- **Visual-Fidelity:** Rather than focusing solely on graphical aesthetics, high visual fidelity is essential for generating accurate synthetic data for visual sensors (e.g., cameras) and ensuring user immersion in VR, addressing a significant limitation of robotics simulation platforms [DBF+25].
- **Physics Accuracy:** To accurately predict the kinematic and dynamic behaviour of the EMAROs robot, the engine needs to have a strong physics backend [SKG+25].

- **Native VR/AR Support:** In order to extend VERA with the proposed Virtual Reality interfaces, the engine must have a framework for VR devices and provide capabilities for creating interactive environments without extensive middleware [CIR23].
- **Robust ROS 2 Integration:** ROS 2 forms the data backbone [PKP+20] of the system and should enable low-latency, high-throughput communication to synchronize the digital twin with the physical robot in real-time [SKG+25].

2.6.1 Robotics Simulators

Gazebo is the most used simulator in Industry 4.0 applications because of its deep integration with ROS and it is open-source [SKG+25]. It delivers detailed physics simulations along with sensor data emulations, making it ideal for precise engineering applications [SKG+25]. However, while this simulator enjoys benefits from a large community and extensive documentation, there are noticeable drawbacks concerning visualization [DBF+25]. For example, the rendering quality of Gazebo is only moderate and not photorealistic to an extent that would enable immersive VR experiences [DBF+25]. Moreover, some works note that Gazebo has a steep learning curve for new users due to its complex configuration and less intuitive user interface compared to modern commercial tools [SKG+25; FCY25].

Webots well established opensource platform, appreciated for cross-platform support and a large library of robot models [KYH+24]. Although it is quite effective in educational applications, the rendering capabilities of the software are somewhat dated compared to the current game engines, making it not fitting for high-fidelity Mixed Reality graphics [KYH+24].

2.6.2 Physics-Based Simulators

A few other simulators were found in literature but considered less appropriate for the particular VR needs of this thesis. **CoppeliaSim (previously known as V-REP)** is widely applied in automation, yet it has less physics realism compared to the professional ones, and its scaling is limited [FCY25]. **MuJoCo** does well with contact-rich tasks and Reinforcement Learning but works almost exclusively on the stability of the physics and not the visualization, so it lacks the inbuilt VR frameworks necessary for Mixed Reality interaction [FCY25].

2.6.3 High-Visual-Fidelity and Game-Based Engines

Game engines have been widely adopted within the robotics community to overcome the visualization limitations of traditional robotics simulators.

NVIDIA Isaac Sim is the state-of-the-art in industrial simulation [DBF+25]. The platform is capable of photorealism through ray-tracing and has advanced GPU-accelerated physics through PhysX [DBF+25]. It excels in generating synthetic data for AI perception [KYH+24]. But Isaac Sim has a very high hardware barrier where high-end NVIDIA RTX GPUs are specifically mandatory [DBF+25]. Being proprietary, it more restricting for use in educational contexts compared to more open-source or widely accessible game engines [DBF+25].

Unreal Engine is renowned for photorealistic visuals is capable of handling big, open-world environments [FCY25]. But it has a steep learning curve because of its dependency on C++ and therefore is not very well-suited for rapid prototyping compared to other engines like Unity [CIR23].

Unity Engine is establishing itself as the primary platform for Industry 5.0 applications, in particular those focusing on HRI and MR [FCY25]. It boasts a balance of high-fidelity graphics with a robust physics engine, PhysX [SKG+25]. Unity is also known for having one of the most mature MR ecosystems, along with a asset store, which lowers the barrier to entry for developers [SKG+25; CIR23]. Literature points out that the user-friendly interface of Unity and supportive community help with a softer learning curve compared to Gazebo or Unreal, particularly for users without deep C++ expertise [SKG+25; CIR23]. ROS 2 [MFG+22] is not natively supported by Unity. Support must be provided through third-party plugins. For the work described in this thesis, the **ROS2 For Unity** plugin from Robotec.AI has been used [Rob21]. This asset offers high-performance communication by loading the ROS 2 middleware layer (RCL layer) directly inside the Unity engine [Rob21]. Unlike bridged solutions, simulation entities now can act as native ROS 2 nodes in the context of Unity, thus respecting QoS and reaching lower latencies [Rob21].

2.7 Synthesis? Discussion?

The review of related work confirms that RitL testing [Hu05] combined with Digital Twin technology [AA23] sets up a paradigm for validating robotic systems. Current research highlights the need to move from basic kinematic simulations to high-visual-fidelity, physics-based environments that can more realistically simulate complex interactions and sensor data [SKG+25; DSR+22].

However, due to its reliance on custom software components, the original VERA framework has significant technical limitations [Geh24]. As explained in Section 2.5.3, the Pygame-based visualiser experiences performance bottlenecks in complex scenes and realistic object manipulation is impossible due to the lack of a dedicated physics engine [Geh24]. Moreover, the potential for researching Human-Robot Collaboration is limited by the lack of Virtual Reality support [Geh24].

To overcome these limitations, this thesis proposes integrating a simulation engine into the architecture of VERA using the **Robot Operating System 2 (ROS 2)** [MFG+22]. The goal is to establish a robust, physics-enabled Mixed Reality environment that bridges the gap between simulation and reality. This new development will be verified by creating demonstration scenarios where the EMAROs robot actively interacts with virtual elements and will serve as a proof of concept for improved system physical realism scalability and immersive interaction.

3 Requirements

This chapter establishes the functional and non-functional requirements for the Mixed Reality Environment. Intended as a validation tool for Robot-in-the-Loop (RitL) testing, the system has to bridge the gap between virtual simulation and physical reality. The requirements follow a hierarchical structure, covering the simulation engine capabilities, system architecture, and digital twin interfaces. In addition, specific requirements regarding the autonomous agents acting within this environment to validate the system are designed.

3.1 Simulation Engine Capabilities

These requirements define the technical capabilities of the underlying simulation engine. The criteria provide a basis for the technology selection process in the implementation phase.

- **FR-01: Visual Fidelity:** The simulation engine shall be able to support rendering pipelines capable of producing image data representative of real-world lighting and material properties. This is required to provide realistic synthetic input for the computer vision algorithms of the robot.
- **FR-02: Physics Simulation:** Overcoming the limitations in the previous VERA framework, the engine shall provide a physics system that will be able to resolve mass, friction, collision, and drag to simulate physical interactions of such sort as a robot pushing obstacles.
- **FR-03: VR & AR Support:** The platform shall provide a mature basis for VR and AR. It shall enable seamless integration of output and input devices without needing to write custom drivers.

3.2 System Architecture & Middleware

The system architecture is defined by limitations of the hardware and the need for standardized robotic communication.

- **FR-04: ROS 2 Integration:** The system shall use ROS 2 as sole middleware for all communication. It shall act as first-party in the network to minimize latency and shall make use of standard message definitions for all sensor streams, telemetry, and control commands to ensure interoperability with the physical robot.
- **FR-05: Time Synchronization:** The system shall act as the simulation time source and publish the clock signal. To prevent data drift in control algorithms, all simulated sensors and publishers shall synchronize their timestamps to this source.
- **FR-06: Scenario Management:** The system shall provide a mechanism to dynamically load and unload simulation environments via a network command during runtime. This is to enable test runs across different environments without restarting the application.

3.3 Digital Twin Interfaces

The system shall provide a digital representation of the robotic platform that mirrors its physical counterpart. This involves synchronization with real-world tracking, kinematic simulation, and the publishing of coordinate transforms.

- **FR-07: Pose Synchronization:** The system shall take real-time input from the external tracking system to synchronize the position and orientation of the Digital Twin with the physical robot.
-
- **FR-08: Tracking Failsafe:** The system shall provide a failsafe that halts the movement of the robot in case there is an interruption of the tracking data stream beyond a threshold that is defined, so that unwanted movements are avoided due to signal loss.
- **FR-09: Standalone Mode:** The system shall be able to simulate the behavior of the robot in the absence of physical hardware. It shall accept velocity commands and publish resulting odometry to test algorithms in a completely virtual environment.

- **FR-10: Coordinate Transforms:** The system shall publish dynamic coordinate transforms representing the motion of the robot. It shall also support the configuration of static sensor offsets to match the physical mounting positions.

3.4 Environment Capabilities

These requirements detail capabilities related to the virtual environment itself, such as generation of sensor data and mechanisms for dynamic interaction and visualization.

- **FR-11: Sensor Simulation:** The system shall provide simulated sensors able to of generating synthetic data, including video feeds and ranging data. The sensors shall be capable of detecting both static environment structures and dynamic obstacles.
- **FR-12: Command Interface:** The system shall provide a generic network interface to receive commands and return status feedback so as to enable the robot to trigger simulation events.
- **FR-13: Dynamic Environment Interaction:** The system shall allow dynamic manipulation of the environment, including physical pushing and attaching of objects, modification of the state of objects, and changes in surface textures at runtime.
- **FR-14: Telemetry Display:** The system shall display real-time robot status data as floating UI elements attached to the Digital Twin within the 3D environment.

3.5 User Interface & Mixed Reality

These requirements outline how human operators visualize and interact with the system.

- **FR-15: AR Projection:** The system shall present a top-down, orthographic view of the scene. The projection parameters shall be configurable such that the virtual view can align 1:1 with the physical dimensions of the testbed floor.
- **FR-16: VR Interface:** The system shall provide a Virtual Reality mode in which the user can view the Digital Twin in 3D space. Information interfaces shall automatically orient themselves to stay readable from the user's perspective.

- **FR-17: Goal Setting:** The interface shall provide the user with the functionality to set navigation targets compatible with the Nav2 navigation stacks by pointing at the virtual ground using the mouse or the VR controller.
- **FR-18: Visualization Control:** The user interface shall allow runtime modification of the visual environment by toggling the visibility or active state of various system components, such as the robot virtual body, lighting, or telemetry displays.

3.6 Autonomous Agent Requirements

These requirements are for the autonomous agents developed to interact with the environment. These will help in demonstrating the capabilities of the system and testing the robot's response to dynamic scenarios.

- **FR-19: Sensor Dependence:** The autonomous agents shall make decisions based only on the environmental data available via the simulated sensors. They shall not directly use any knowledge from the internal state of the simulation engine.
- **FR-20: Standardized Communication:** The agents shall utilize standard robotic communication patterns to send action commands and monitor feedback topics to confirm task completion.
- **FR-21: Adversarial Behavior:** The system shall include an autonomous entity acting as an adversary, controlled by an algorithm that dynamically responds to the environment.

3.7 Non-Functional Requirements

The following constraints define the standards necessary to ensure reliability and extensibility.

- **NFR-01: Low Latency:** The system shall minimize the delay between the reception of a state update and the rendering of the corresponding visual frame. Latency shall remain lower than the simulation frame time to ensure immediate visual feedback.

- **NFR-02: Frame Rate:** The simulation shall maintain a constant target frame rate (at least 60 FPS) to avoid simulator sickness in VR [CKY20] and to ensure that AR projections remain visually stable and aligned with the physical environment.
- **NFR-03: Modularity:** The system architecture shall be modular, allowing individual components (e.g., sensor models, environment logic) to be added or modified without altering the core application.
- **NFR-04: Scalability:** The system shall support scenes with a high density of dynamic objects without violating frame rate stability requirements.
- **NFR-05: Configurability:** The system shall provide interfaces to configure robot parameters, sensor properties, and environment settings, enabling adaptation to different physical setups without recompilation of code.

4 Concept and Implementation

This chapter describes the development of the Mixed Reality Environment, a platform designed to support RitL testing by combining simulation with augmented reality projection. The resulting system bridges the gap between purely digital simulations and real-world experiments by creating a common workspace where physical robots and virtual objects coexist and interact.

The central concept of this environment is the projection of a simulated world directly onto the laboratory floor. Rather than testing robots in completely physical setups or purely within software simulations, this system projects dynamic, physics-based elements into the real world. As can be seen from Figure **??**, the physical robot moves across the real floor while sensing and acting upon virtual objects, such as a field, obstacles, or other entities. This projection has a dual purpose: it provides instant visual feedback to a human observer and enables the robot to receive synthetic sensor data that reflect the state of the virtual scene. This allows the robot to detect and respond to simulated entities as if they were physically present.

Figure 4.1: The Mixed Reality Environment in operation. The physical robot interacts with a projected Smart Farming scenario, where the robot and the digital twin are synchronized via ROS 2.

To enable these applications, the system integrates several key functions into an architecture. First, it maintains a dynamic twin of the robot. This digital twin is synchronized with the movement of the physical hardware but can interact with the simulation's physics engine. This allows for complex actions, such as pushing virtual boxes or colliding with moving objects. This ensures the simulation responds realistically to the robot's physical presence rather than serving as a static background.

In addition to maintaining a dynamic digital twin, the system also simulates sensor for the robot. It generates virtual LiDAR scans and camera images from the robot's perspective, allowing autonomous agents to receive virtual sensordata. This enables thorough testing of navigation and vision algorithms within the mixed reality environment.

The architecture uses ROS 2 [**MFG22a**] as the primary communication backbone. By integrating a ROS 2 nodes directly into the simulation engine, the virtual environment acts as a first-party participant in the robot's network. It manages time synchronization to prevent timing related errors, handles the loading of different scenarios and provides two-way exchange of status data and control commands.

Besides sensor simulation, the environment also supports real-time diagnostic feedback. The system functions as a monitoring tool by projecting the robot's internal operating data back onto the physical workspace and the VR interface. Performance metrics, such as temperature, RAM usage, and location coordinates, are shown on information panels attached to the digital twin in the 3D environment. Moreover, the system shows the robot's live camera feed within the virtual scene. This allows operators to observe the robot's physical behavior alongside its internal status and visual perception in real-time.

To demonstrate the flexibility and reliability of this architecture, three different Autonomous Agents were implemented:

- A Smart Farming Agent that autonomously explores a virtual field, identifies its boundaries, and operates simulated tools to perform fieldwork tasks.
- A Logistics Agent that autonomously searches for, identifies and sorts colored objects within the environment and brings them to designated zones.
- A Line Following Agent that uses visual tracking to follow paths drawn by a human user in real-time.

The platform provides an adaptable and scalable testing solution by integrating physics, sensor simulation, and autonomous control logic into a synchronised mixed reality environment. The only factor limiting the test environment's complexity is software, not its physical design. The technological decisions, system architecture, and particular implementation of these elements will be covered in detail in the sections that follow.

4.0.1 Comparative Analysis

The simulation engines that were considered as candidates to implement the Mixed Reality Environment are: Gazebo, Isaac Sim, Unreal Engine, and Unity. All were evaluated based on five critical criteria: visual fidelity, physics capabilities, learning curve, community support, and native integration with VR, AR and ROS 2. Table 4.1 summarizes the results of this evaluation.

Table 4.1: Comparison of Simulation engines for Mixed Reality Digital Twins [FCY25; KYH+24; SKG+25; CIR23].

Feature	**Gazebo**	**Isaac Sim**	**Unreal Engine**	**Unity**
Primary Use	Control & Navigation	AI & Photorealism	Photorealism	HRI & MR (VR/AR)
Visual Fidelity	Moderate	Very High	Very High	High
Physics Engine	ODE / Bullet	PhysX 5 (GPU)	Chaos / PhysX	PhysX
Learning Curve	Steep	Advanced	Steep (C++)	Moderate (C#)
Community	High (ROS)	Moderate	High (Gaming)	Very High (MR)
ROS 2 Integ.	Native	Bridge	Bridge	Plugin (Native)
Hardware	Low	Very High (RTX)	High	Moderate

4.0.2 Selection Rationale

Based on the comparative analysis, the **Unity Engine** was selected as the implementation platform for this thesis. This decision is driven by four key factors that align with the system requirements:

- **XR Framework (FR-09):** Unity has an integrated framework for VR applications [CIR23]. In contrast, simulators like Gazebo lack native VR support, which is critical for the proposed human-robot interaction interface.
- **Visual & Physics Balance (FR-04, FR-05):** Unity provides high-fidelity visualization alongside PhysX integration. This offers an optimal balance between performance and visual quality, avoiding the restrictive hardware requirements associated with NVIDIA Isaac Sim [FCY25].
- **Development Efficiency:** The use of C# scripting, combined with documentation and community support, makes Unity more accessible for rapid prototyping than the C++ environment of Unreal Engine [CIR23].
- **ROS 2 Integration (FR-01):** The availability of the `ros2-for-unity` asset enables the simulation to function as a first-party participant in the ROS 2 network. This means it supports the low-latency communication requirement by avoiding external bridge applications [Rob21].

4.1 System Architecture

The realization of the Mixed Reality Environment requires a distributed software architecture that connects the advanced rendering capabilities of the Unity engine with real-time robotic control. The system is designed to satisfy the requirement for **Middleware Integration (FR-04)** by establishing a one unified communication layer between the simulation host and the autonomous agents.

4.1.1 Hardware and Network Topology

To ensure high performance and support for specific hardware peripherals, the system topology is consolidated into a single high-performance workstation and the mobile robot. As shown in Figure **??**, the architecture is divided into three clear computational domains:

1. **Simulation Host (Windows 11):** The Unity engine runs natively on Windows. This operating system was chosen due to currently missing support of VR interfaces in the Unity Editor under Linux. This layer manages the physics engine (PhysX), renders the digital twin, and communicates with the VR headset. Additionally, a projector is connected to this host to visualize the environment on the physical floor [**PLACEHOLDER_HARDWARE_SETUP**].
2. **Control Environment (WSL/Ubuntu):** ROS 2 Jazzy runs within the Windows Subsystem for Linux within Ubuntu 24.04. This environment hosts the computationally intensive software, such as the autonomous navigation stack (*Nav2*) [**NAV2Ref**] and the mission logic agents. Running these components directly on the robot causes performance issues or system crashes due to limited onboard resources, so they are offloaded to the workstation to ensure stable and responsive operation.
3. **Physical Robot (Hardware Layer):** The EMAROs [**EMAROs_Ref**] robot operates as an independent node in the network. It uses Ubuntu 24.04 and runs ROS 2 Jazzy on an onboard Raspberry Pi to handle hardware drivers, such as motor controllers and IMU sensors. It connects to the workstation via Wi-Fi.

Figure 4.2: Hardware and Network Topology. The workstation handles both the Unity simulation (Windows) and the heavy control logic (Ubuntu/WSL) to preserve the robot's onboard resources. The robot, VR headset, and projector connect as peripherals to this central core.

For the Mixed Reality interface **(FR-16)**, the visual output from Unity is streamed via Wi-Fi to the Meta Quest Pro headset using *ALVR* [**ALVR_Github**], an open-source streaming solution. This setup allows the user to move freely without a cable tether.

4.1.2 Software Component Architecture

The software architecture follows a modular publisher-subscriber pattern. To integrate the Unity game engine with the robotic network, the system utilizes the *ROS2ForUnity* library [**ROS2ForUnityDocs**].

ROS 2 Integration via MonoBehavior

The core of this integration is the `ROS2UnityComponent` [**ROS2ForUnityDocs**] class. This component acts as a wrapper around the standard ROS 2 middleware. Inheriting from Unity's `MonoBehaviour` class, it must be attached to a `GameObject` in the scene to function. Upon startup, this component initializes the ROS 2 context, makes sure that a valid connection is established, and handles the lifecycle of the ROS 2 nodes running within the simulation. It allows other scripts to access the ROS 2 network by referencing this central component to create publishers, subscribers, and service clients.

Unity Scripting and Execution Loops

The functionality of the digital twin is driven by C# scripts that interact with the engine's internal state machine. To ensure that physical simulation remains deterministic while visualization remains smooth, the architecture uses specific phases of the Unity execution loop [**UnityDocumentation**]:

- **Initialization (`Start`):** This method is called once when the script is first enabled, before any frames are updated. In this architecture, it is used to retrieve the reference to the `ROS2UnityComponent` [**ROS2ForUnityDocs**], initialize the specific ROS nodes, and set up the necessary publishers and subscribers.
- **Physics Loop (`FixedUpdate`):** This method executes at a constant, user-defined time step, independent of the visual frame rate. All physics-based calculations, such as updating the robot's position based on received velocity commands or handling collision detection, occur in this loop. This ensures that the robot's physical behavior is consistent regardless of graphical performance.
- **Rendering Loop (`Update`):** This method runs once every frame. It is utilized for logic that is not physics-related, such as updating UI elements, rendering diagnostic lines, or capturing camera frames for visualization. This separation

ensures that high-frequency visual updates do not interfere with the stability of the physics engine.

Figure **??** provides an overview of how these Unity scripts interact with the external ROS 2 nodes via topics. Detailed descriptions of these components follow in Sections 4.2 and 4.3.

Figure 4.3: Software Component Architecture. Unity scripts act as ROS nodes, publishing sensor data and subscribing to control commands using ROS 2 topics. The external control stack, running in WSL, processes this data to make autonomous decisions.

4.1.3 Time Synchronization

A critical challenge in Robot-in-the-Loop testing is how to synchronize time between the simulation and the robot's software. If the simulation runs slower or faster than real-time, the navigation algorithms of the robot may calculate velocities that are incorrect. To fulfill the requirement for **Time Synchronization (FR-05)**, the system decouples the ROS network from the computer's system time.

The `ClockPublisher.cs` script in Unity acts as the master clock. It publishes the current simulation time to the `/clock` topic. The ROS 2 nodes running on the workstation such as the navigation stack are configured with `use_sim_time = true`, forcing them to synchronize their logic with the Unity engine.

To ensure consistency, a `RosTimeHelper.cs` utility is used by all simulated sensors. This script ensures that sensor data leaving Unity is stamped with the exact simulation time of the frame it was rendered in to prevent data drift.

4.1.4 Scenario Management

To support the requirement for **Scenario Management (FR-06)**, the architecture allows the system to switch between different test environments without restarting the entire application.

This is handled by the `SceneManager.cs` script. When it receives a command via the `/vera/load_scenario` topic, it performs a clean reset sequence:

1. **Shutdown:** The active ROS 2 connections are terminated. This ensures that all Unity nodes are properly removed from the network, preventing any lingering nodes from publishing outdated or invalid data.

2. **Load:** Unity loads the new environment scene asynchronously.
3. **Restart:** A fresh ROS2UnityComponent [**ROS2ForUnityDocs**] initializes, creating a new connection for the loaded scenario.

This fresh start approach ensures that Unity components specific to one scenario do not interfere with others.

4.2 Robot Representation and Control

The robot in the Mixed Reality Environment runs in one of two modes: either as a completely simulated Virtual Robot or as a synchronized Digital Twin of the physical EMAROs [**EMAROs_Ref**] platform. To ensure consistency, both of these modes use the same visual and geometric model. This model is a simple cylinder that approximates the physical size of the robot for collision detection while keeping computational costs low. The behavior of this cylinder depends on the active control script, allowing the system to switch between physics-based simulation and position-based synchronization based on the requirements of the test.

4.2.1 The Virtual Robot Implementation

In the standalone simulation mode, the robot is controlled by the `VirtualRobotController`. This component allows for the validation of high-level control logic in a safe environment before it is deployed on real hardware. Unlike simulations that teleport a robot to a target point, this implementation uses the Unity physics engine PhysX to simulate realistic movement (FR-09).

The controller acts as a ROS 2 node that accepts velocity commands (`cmd_vel`) and publishes odometry data. To handle data exchange between the asynchronous ROS 2 thread and the Unity main thread, incoming commands are buffered in a thread-safe `ConcurrentQueue`. During the physics step of Unity (`FixedUpdate`), the controller retrieves the latest command and applies it to the `Rigidbody` of the robot.

The script does not directly change the position of the robot. Instead, it modifies the velocity of the physics body. By applying limits to acceleration, the system simulates mass and inertia for the robot. The robot can push lightweight virtual objects, but heavy obstacles or walls will stop its movement. This provides realistic feedback to the navigation system.

Simultaneously, the controller provides ground truth data. It tracks the movement of the robot in Unity, converts the coordinates to the ROS standard, and publishes a timestamped odometry message. This closes the control loop for the external navigation logic [**MFG+22**].

4.2.2 The Digital Twin Implementation

When operating in RitL mode, the `PoseRobotController` takes control. In this configuration, the virtual robot stops acting as a dynamic entity and instead becomes a digital twin of the real EMAROss [**EMAROs_Ref**] robot (FR-07).

The controller listens to a pose topic given by the external ArUco tracking. On every new incoming pose, the script converts the coordinates into Unity world space. To ensure alignment, an offset parameter enables the operator to perform a calibration of the Digital Twin so that its center is aligned with the physical mounting position of the tracking marker.

Unlike the Virtual Robot, the Digital Twin overrides the physics engine. It is set to a kinematic state, meaning it is immune to virtual forces. If the physical robot moves, the Digital Twin moves with it, ignoring virtual barriers. This allows the physical robot to push virtual objects with absolute force. This way, the state of the virtual world always reflects physical reality.

Figure 4.4: Data flow comparison between the Virtual Robot and the Digital Twin. The Virtual Robot (left) is driven by internal physics and ROS commands, while the Digital Twin (right) is driven directly by external tracking data.

4.2.3 Safety and Failsafe Mechanisms

A major challenge in mixed reality testing is signal latency or loss. If the physical robot leaves the tracking area or if the ArUco marker is blocked, position updates will stop. Without a failsafe, the Digital Twin would freeze while the physical robot continues to move blindly, causing a mismatch between reality and simulation (FR-08).

To prevent this, the `PoseRobotController` implements a safety watchdog. The system continuously checks how long it has been since receiving the last pose update. If this delay exceeds a given safety limit, the system assumes tracking has been lost.

When this occurs, the controller marks the tracking state as invalid. Although the Digital Twin stops moving in the simulation, the system is designed to send a stop command to the motors of the physical robot. This ensures the robot does not perform actions without simulation guidance. Once the tracking system re-acquires the robot, the watchdog resets and the Digital Twin resumes synchronization immediately.

4.3 Environmental Simulation Capabilities

To serve as a valid testbed for autonomous agents, the virtual environment must provide more than static geometry. It must facilitate mechanical manipulation of objects, generate synthetic perception data that mirrors real hardware, and support the persistent modification of environmental states. This section details the technical implementation of these capabilities, addressing the requirements for Sensor Simulation (FR-11), Dynamic Environment Interaction (FR-13), and the Command Interface (FR-12).

4.3.1 Physics-Based Interaction

A critical feature of the system is the ability for the robot to mechanically couple with environmental objects, such as agricultural tools or logistics containers. While standard game engines often use physics joints to connect rigid bodies, these can introduce instability and jitter when subjected to the high-frequency velocity changes typical of robotic controllers. To address this, the architecture implements a robust kinematic attachment system managed by the `AttachmentCommandManager` and the `RobotAttachmentController`.

Command Interface and Logic

To enable interoperability with the ROS 2 ecosystem (FR-12), the system exposes a high-level command interface via the `/robot_command` topic. This interface utilizes standard string-based messages (e.g., "attach,plow" or "detach") rather than complex service calls. This design choice simplifies the debugging process, allowing developers to trigger complex physical sequences manually via the ROS 2 command line interface (CLI) without requiring custom service clients. The command manager operates on the Unity main thread, processing a thread-safe queue of incoming messages. When an attachment command is received, it performs a spatial query using `Physics.OverlapSphere` to identify valid target objects within a defined search radius.

Kinematic Coupling Strategy

Once a valid target is identified, the controller executes a kinematic handover. The target object is re-parented to the robot's transform hierarchy at a specific mount point defined by local offsets. Crucially, the system modifies the physics state of the attached object (FR-13). Upon attachment, the script sets the object's `Rigidbody.isKinematic` property to true. In the Nvidia PhysX engine, this removes the object from the dynamic simulation loop, effectively rendering it massless relative to the robot. This ensures that the robot's navigation controller does not need to compensate for the sudden addition of mass or the shifting center of gravity, resulting in stable and predictable movement during transport.

Simultaneously, the controller modifies the collision detection matrix using `Physics.IgnoreCollisi`
This explicitly suppresses collision checks between the robot's chassis colliders and the attached tool's colliders. Without this suppression, the instantaneous snapping of the tool to the robot's mount point would result in immediate interpenetration of geometry. The physics engine would resolve this by applying immense depenetration forces, causing the equipment to be violently ejected from the robot.

Figure 4.5: State diagram of the attachment logic. When the state transitions from Detached to Attached, the object is parented, its physics are disabled (Kinematic), and collisions with the robot are ignored.

4.3.2 Sensor Simulation

To validate the perception algorithms of the autonomous agents, the environment must generate synthetic data streams that are structurally identical to those produced by the physical EMAROs [**EMAROs_Ref**] robot (FR-11).

LiDAR Simulation

The `LidarSensor` component simulates a 2D planar laser scanner. Unlike GPU-based depth buffers often used in visual rendering, this implementation utilizes the Unity physics engine's raycasting API to query the scene geometry directly. In every simulation step, the component executes a loop corresponding to the configured number of beams (e.g., 360). For each beam, the system calculates a direction vector based on the scan angle. Since Unity utilizes a Left-Handed coordinate system (Y-up) and ROS uses a Right-Handed system (Z-up), the script employs a custom extension method to transform the desired scan angle into the correct Unity world-space vector.

A `Physics.Raycast` is projected along this vector. If the ray hits a collider on the collision layer, the distance is recorded; otherwise, the sensor returns infinity. The resulting range data is serialized into a standard `sensor_msgs/LaserScan` message. To ensure temporal consistency with the navigation stack, the message header is stamped using the `ClockPublisher` immediately after the raycast loop completes. This precise timestamping prevents the ROS 2 transform tree (TF) from rejecting the scan data due to extrapolation errors, which can occur if the timestamp lags behind the robot's rapid movement.

Figure 4.6: Visualization of the LiDAR simulation in the editor. Red lines indicate raycasts that did not hit an obstacle within range, while yellow lines indicate valid hits registered by the physics engine.

Camera Simulation

Visual perception is handled by the `CameraSensor` component. This script attaches a dedicated Unity Camera to the robot model, which is configured to render the scene into a `RenderTexture` rather than the user's screen. This allows the simulation to generate visual data at a resolution independent of the application window.

The data extraction pipeline transfers the raw pixel data from the Graphics Processing Unit (GPU) memory to the Central Processing Unit (CPU) memory. Once on the CPU, the data is encoded into the appropriate format—either raw RGB8 for local processing or JPEG for bandwidth-efficient network transmission. The resulting byte array is published as a `sensor_msgs/Image` or `CompressedImage`. To support various testing setups, the camera configuration is dynamic. The `RobotCameraManager` instantiates sensor prefabs at runtime based on the loaded scenario, attaching them to defined "bones" in the robot hierarchy. This allows the system to switch between a downward-facing camera for line following and a forward-facing camera for object detection without modifying the core robot definition.

4.3.3 Dynamic Visuals and Terrain Modification

A specific requirement for the system was the ability to support Reactive Behavior (FR-22), where the environment changes in response to the robot's actions. This is realized through a dynamic texture painting architecture implemented in the `TrackPainter` and `SharedPaintTextureRegistry` scripts. This feature allows specific interactions—such as a plow tool touching the ground—to persistently alter the visual appearance of the terrain. Unlike temporary decals or particle effects, this system modifies the underlying texture data of the ground material.

Shared Texture Registry

In complex scenarios, multiple agents (e.g., the robot, a VR controller, or a mouse interface) may attempt to modify the terrain simultaneously. To manage this, the `SharedPaintTextureRegistry` functions as a singleton manager. It maintains a dictionary of active textures and ensures that all painters operate on a single, shared instance of the `Texture2D`. This prevents race conditions where one agent overwrites the work of another and ensures data consistency across the simulation.

Persistent State Modification

The modification logic relies on physics raycasting to identify the precise UV coordinates of the surface at the point of contact. When a valid hit is detected on the target layer, the `TrackPainter` calculates the pixel coordinates on the texture map. It then modifies the pixel buffer directly using `SetPixels32`, changing the color of the "soil" to represent a new state (e.g., tilled earth or harvested crops).

Crucially, these changes are committed to the GPU using the `Apply` method. This ensures that the modification is not just visual but persistent. The changed pixels remain in the texture data, meaning that if the robot leaves the area and returns, the camera sensor will perceive the altered terrain. This closes the feedback loop between action and perception: the robot acts on the environment, the environment updates its state, and the robot's sensors perceive this new state to inform future decisions.

Figure 4.7: The dynamic terrain modification system. A raycast determines the UV coordinate of the tool's contact point (left), which is used to update the shared texture registry (right), permanently altering the ground appearance.

4.3.4 Scenario and Object Management

To support the requirement for comprehensive Scenario Management (FR-06), the system must be capable of reconfiguring the simulation context at runtime without requiring a restart of the entire application stack. This capability is essential for automated testing, allowing a test runner to cycle through various environments (e.g., a warehouse, a crop field, or a game arena) sequentially.

Dynamic Scenario Loading [**ACHTUNG DOPPENLT DRIN**] The transition logic is encapsulated in the `SceneManager` component. This script maintains a subscription to the `/vera/load_scenario` topic, listening for string-based triggers identifying the target environment. Upon receiving a valid command, the manager executes a rigid shutdown sequence to ensure system stability. First, it explicitly triggers a shutdown of the ROS 2 communication layer via `Ros2cs.Shutdown()`. This step is critical; it forces the destruction of all active nodes, publishers, and subscribers, preventing "zombie" processes from lingering and polluting the network with invalid data during the transition. Once the network is severed, the Unity Scene Management API loads the new environment asynchronously. Upon completion, the new scene initializes its own `ROS2UnityComponent` [**ROS2ForUnityDocs**], establishing a clean connection state for the next test iteration.

To ensure continuity of critical services—such as the simulation settings or the network configuration itself—specific GameObjects are governed by the `PersistAcrossScenes` utility. This implements a Singleton pattern using `DontDestroyOnLoad`, ensuring that essential infrastructure survives the scene transition while the environmental geometry is swapped out.

Dynamic Object Spawning

Beyond static geometry, the environment must support the lifecycle management of transient entities, such as the delivery boxes in the Logistics scenario. This is handled by the `DynamicObjectManager`, which acts as a bridge between the ROS 2 data stream and the Unity instantiation engine (FR-13).

The manager subscribes to the `/vera/virtual_objects` topic, processing a custom message structure that defines an object's ID, type, and pose. Internally, the system maintains a dictionary mapping string identifiers (e.g., "box_red", "obstacle_A") to Unity Prefabs. When an "ADD" or "MODIFY" command is received, the system checks if the object already exists. If it is new, the corresponding prefab is instantiated into the scene. If it exists, its transform is updated. Crucially, this component handles the coordinate conversion between the two systems, mapping the right-handed ROS position (x, y, z) and orientation (x, y, z, w) to the left-handed Unity world space. This allows external scripts or test runners to populate the scene with obstacles dynamically, validating the robot's ability to navigate changing environments.

4.4 Mixed Reality and User Interaction

A primary objective of the framework is to enhance the transparency of the robotic system by providing intuitive, real-time feedback to human operators. This is achieved through a Mixed Reality (MR) interface that visualizes internal robot states and allows for direct environmental manipulation. This section details the implementation of augmented telemetry, sensor projection, and the interactive control systems designed for both desktop and Virtual Reality (VR) contexts.

4.4.1 Augmented Telemetry and Diagnosis

To fulfill the requirement for Telemetry Display (FR-14), the system implements the `RobotInfoBillboard` component. This script aggregates critical health and status data—such as CPU usage, battery voltage, and the current mission task—and renders it as a floating interface anchored to the robot's coordinate frame.

Hybrid Visualization Strategy

For the AR floor projection and standard desktop views, the system utilizes Unity's immediate mode GUI (`OnGUI`) to render text overlays. This ensures that the text remains legible and strictly aligned with the screen plane, regardless of the projector's angle.

However, in a Virtual Reality environment, screen-space rendering breaks immersion and depth perception. To address this, the architecture supports a World Space Canvas solution for VR users. In this configuration, the textual information is rendered onto a 3D plane floating physically above the Digital Twin. To ensure readability from any perspective, the canvas implements a `LookAt` constraint in its update loop, continuously orienting the UI to face the VR headset's main camera. This allows an operator walking around the virtual field to inspect the robot's status by simply looking at it, mimicking a physical diagnostic display.

4.4.2 Sensor Data Projection

While the Digital Twin visualizes the robot's kinematics, it does not inherently convey what the robot "sees." To bridge this gap, the system utilizes the `RosImageToMaterial` component to project the robot's internal perception directly into the environment (FR-15).

This component subscribes to the camera topics published by the robot. Upon receiving a frame—either raw or compressed—it decodes the data into a Unity `Texture2D` and applies it to a material on a planar mesh attached to the robot's chassis.

In the AR setup, this plane is positioned horizontally above the robot, projecting the computer vision debug output (such as bounding boxes or detected lines) onto the floor moving along with the robot. This provides observers with immediate visual confirmation of the perception stack's performance without requiring a separate monitor. Similarly, in VR, this projection functions as a virtual dashboard, allowing the user to verify alignment between the physical world and the robot's internal model from a top-down perspective.

Figure 4.8: The Augmented Telemetry interface. A billboard displays system stats (top), while the `RosImageToMaterial` component projects the live OpenCV debug feed onto a plane attached to the robot (bottom), visualizing the internal state of the perception stack.

4.4.3 Interactive Control Interfaces

The system enables users to actively modify the simulation state and direct the robot's actions (FR-17, FR-18). The implementation follows an input-agnostic architecture, where the core logic relies on 3D raycasting rather than specific hardware events. This design allows the system to support both standard mouse inputs and 3D VR controllers interchangeably.

Goal Setting and Navigation

The `NavigationGoalController` facilitates high-level control of the robot. In the desktop configuration, it casts a ray from the camera through the mouse cursor coordinates. When the ray intersects with the ground layer, the script calculates the target point and orientation based on the user's drag gesture. This data is serialized into a `geometry_msgs/PoseStamped` message and published to the `/goal_pose` topic, triggering the ROS 2 navigation stack. For Virtual Reality, this logic utilizes the `XR Ray Interactor`, enabling the operator to point at the virtual floor and dispatch the robot using the controller's trigger.

Environmental Modification and Physics Interaction

To support dynamic scenarios like Line Following, the user must be able to alter the environment at runtime. The `MouseSurfacePainter` component enables users to draw directly onto the terrain texture using a continuous raycast. In VR, this allows the operator to naturally draw paths or obstacles in 3D space that the physical robot can immediately perceive and follow.

Beyond painting, the VR interface leverages the physics engine to allow direct object manipulation. Users can physically grab and move interactive objects, such as the logistics boxes, using the VR controller's grip function. This allows for the manual reset of test scenarios or the introduction of dynamic obstacles (e.g., dropping a box in the robot's path) to validate the system's reactive planning capabilities (FR-22).

Visualization Management

In complex mixed reality scenarios, the density of visual information can become overwhelming. To manage this, the `VisibilityToggleManager` allows users to selectively hide or reveal specific visualization aids attached to the robot. This component maps input actions to the rendering state of the `RobotInfoBillboard`, the projected camera plane, and even the robot's visual cylinder body. This feature allows operators to declutter the view when necessary—for example, hiding the large floating billboard to inspect the interaction between the robot's chassis and a tool—without disabling the underlying functionality.

4.5 Implementation of Autonomous Agents

To validate the capabilities of the Mixed Reality Environment, the system hosts three distinct autonomous agents. These agents are not simple hard-coded scripts but fully featured robotic applications that utilize the synthetic sensor data provided by the Digital Twin to make decisions (FR-19). They serve to demonstrate the system's ability to support various domains of robotics, from agricultural field coverage to logic-based logistics.

The high-level decision-making for all agents is orchestrated using *YASMIN* (Yet Another State MachINe), a ROS 2-native library that allows for the creation of hierarchical, interruptible mission behaviors. This architecture ensures that the robot can switch between reactive behaviors (e.g., obstacle avoidance) and deliberative tasks (e.g., path planning) seamlessly.

4.5.1 Navigation and Control Stack

The fundamental capability of movement is provided by the *Navigation2 (Nav2)* stack [**NAV2Ref**]. To address the varying physical characteristics of the simulated versus the physical robot, the architecture employs a dual-configuration strategy. Two distinct parameter sets—`nav2_params_virtual.yaml` and `nav2_params_robot.yaml`—are maintained. While they share the same behavioral tree structure, they utilize different tuning for inflation radii and cost scaling to account for the physical robot's safety margins and sensor noise.

A critical design choice in the controller configuration is the use of the **Rotation Shim Controller** wrapping the **DWB Local Planner**. In tight operating environments, such as the crop rows of the Smart Farming scenario or the dense storage areas of the Logistics scenario, standard differential drive controllers often fail to navigate sharp turns without collision. The Rotation Shim intercepts navigation commands and forces the robot to rotate in place to align with the path heading before attempting to move forward. This ensures that the robot can maneuver in confined spaces where the available turning radius is smaller than the robot's kinematic limits.

To ensure safety during autonomous operation, a **Collision Monitor** node runs in parallel with the navigation stack. Configured with a defined polygon representing the robot's footprint plus a safety margin, this node monitors the LiDAR stream directly. If an obstacle breaches the safety polygon, the monitor overrides the navigation stack and publishes a zero-velocity command to the motor controller, acting as a software-level emergency stop.

4.5.2 Modular Perception Stack

To interpret the visual data generated by the `CameraSensor`, the agents employ specialized computer vision pipelines tailored to their operational requirements. This satisfies the requirement for Perception Compatibility (FR-20) by ensuring algorithms function identically on virtual and real camera feeds.

Modular Perception Node (Farm & Logistics) For the Smart Farming and Logistics agents, a modular `perception_node.py` was developed. Rather than running all detection algorithms simultaneously, which would consume excessive computational resources, this node acts as a central dispatcher. It switches between specialized **Processors** based on the current mission state defined by the state machine (e.g., switching from *Field Detection* to *Edge Following*). All processors inherit from a base class (`base_processor.py`) that standardizes the ingestion of `sensor_msgs/CompressedImage` messages and their conversion to OpenCV format (BGR8).

Dedicated Line Perception Node The Line Following agent utilizes a standalone `line_perception_node.py`. This separation allows for a streamlined, high-performance pipeline optimized specifically for low-latency visual servoing, unburdened by the overhead of modular task switching.

4.5.3 Agent 1: Smart Farming

Scenario Context This agent operates in a simulated agricultural field consisting of crop rows and uncultivated soil. The goal is to identify the field boundaries, navigate along the crop edges without damaging them, and sequentially apply three tools (plow, seeder, harvester) to the field.

Perception Modes The agent utilizes four distinct perception processors depending on the active state:

1. **Field Detection (`FieldDetectionProcessor`):** Determines if the robot is currently on arable land by calculating the ratio of "field-colored" pixels in the image using HSV thresholding.
2. **Edge Detection (`EdgeDetectionProcessor`):** Used during the approach phase. It creates binary masks for both "field" and "outer" textures. By applying morphological dilation to the field mask and computing the bitwise AND with the outer mask, it identifies the boundary line where the two textures meet.
3. **Edge Adjustment & Following:** Implemented via two specialized processors (`EdgeAdjustmentProcessor` and `EdgeFollowingProcessor`), these modules are used for precise alignment and tracking. They utilize the Probabilistic Hough Transform (`cv2.HoughLinesP`) to find line segments along the detected boundary, calculating the robot's lateral error (distance from the line centroid to image center) and angular error (deviation from vertical).
4. **Equipment Detection (`EquipmentDetectionProcessor`):** Active during tool search. It isolates specific tool colors (e.g., yellow for harvester) using HSV masking and calculates the object's centroid to guide the docking approach.

The real-time debug stream (Figure **??**) visualizes these detections, overlaying the regression line (green) or the equipment bounding box (blue) onto the camera feed.

Figure 4.9: Debug view of the Smart Farming agent. Left: Edge detection visualizing the crop/soil boundary intersection. Right: Equipment detection highlighting the target tool for attachment.

State Machine Logic The mission is governed by a hierarchical YASMIN state machine (Figure **??**) which executes two distinct phases:

Phase 1: Field Mapping (Initialization)

1. **CheckOnField:** The robot verifies it is inside the field boundaries.

2. **DriveToEdge:** The robot drives forward until the `EdgeDetectionProcessor` identifies a boundary. It then executes an open-loop forward drive for a calibrated distance (e.g., 20cm) to account for the camera's blind spot, ensuring the robot's center of rotation is aligned with the edge.
3. **AdjustToEdge:** The robot rotates in place until the `EdgeAdjustmentProcessor` confirms that the edge line is vertical (angular error near zero) and centered.
4. **FollowEdge (Mapping):** The robot circumnavigates the field using a PID controller on the lateral error. Crucially, this action detects **Field Corners** by monitoring the robot's odometry. A sharp change in yaw (approx. 90 degrees) triggers the recording of the robot's current pose. Once four corners are recorded, the field boundary is defined.

Phase 2: Equipment Cycle

1. **FindEquipment:** The robot executes a **Spiral Search** pattern using the Nav2 stack [**NAV2Ref**] to locate a specific tool.
2. **EquipTool:** Upon detection, the robot approaches the tool. The action triggers the Unity `AttachmentCommandManager` to kinematically lock the tool to the robot.
3. **GenerateCoveragePath:** Using the four corners identified in Phase 1, this action calculates a boustrophedon (lawnmower) path that covers the entire field area.
4. **ExecuteCoverage:** The robot drives the generated path. During this phase, the Unity `TrackPainter` modifies the ground texture to visually represent tilled soil.
5. **Return & Unequip:** The robot returns the tool to its origin and detaches it. The state machine then loops back to search for the next tool in the sequence (Seeder, then Harvester).

Figure 4.10: The Smart Farming State Machine. The robot first maps the field boundaries (FollowEdge) to identify corners, then uses these corners to generate coverage paths for various tools.

4.5.4 Agent 2: Logistics

Scenario Context The Logistics agent operates in a dynamic warehouse environment populated with colored transport boxes (Red, Blue, Green, Yellow) and corresponding projected delivery zones. The robot's objective is to autonomously search for items, physically attach them, identify the correct sorting destination, and transport the payload. This scenario validates the system's capabilities regarding object manipulation (FR-13), spatial memory, and error recovery.

Perception Modes The agent employs the modular `perception_node.py`, switching between two specialized processors based on the mission phase:

1. **Box Detection (**`BoxDetectionProcessor`**):** This processor isolates manipulatable objects from the background. It applies an HSV color mask to filter specific target colors (or iterates through all known colors for an "Any" search). It utilizes `cv2.findContours` to identify object blobs, filtering them by a minimum area threshold to reject sensor noise. The centroid of the largest valid contour is calculated using image moments (`cv2.moments`) to provide a steering target.
2. **Zone Detection (**`ZoneDetectionProcessor`**):** This processor is tuned to detect the flat, projected delivery zones on the floor. While similar to box detection, it uses distinct HSV ranges to account for the additive lighting nature of the projector and calculates the zone's center to ensure the robot drops the box inside the boundary.

To assist in debugging, the perception node publishes an annotated video stream (Figure **??**). The system draws bounding contours around valid targets and overlays classification labels (e.g., "RED BOX", "Distance: 1.2m") and the active search status directly onto the feed.

Figure 4.11: Debug view of the Logistics agent. The processor identifies a red transport box, drawing a contour around it and calculating the centroid (red dot) for the approach vector.

State Machine Logic The mission logic is defined in `mission_state_machine.py` (Logistics variant) and implements a "Search-Retrieve-Deliver" loop. A key feature of this agent is the utilization of the YASMIN **Blackboard** to implement spatial memory, allowing the robot to learn the environment structure over time.

1. **FindBox (Spiral Search):** The robot executes the `find_box_action_server`. This generates a spiral pattern of waypoints using the Nav2 stack [**NAV2Ref**] to efficiently cover the floor plan. The robot navigates these waypoints until the `BoxDetectionProcessor` identifies a valid target in the camera's Region of Interest.
2. **AttachBox:** Upon reaching the target, the robot performs a precision approach. The `AttachBox` action sends a formatted string command (e.g., `"attach,box_red"`)

via the /robot_command topic to the Unity AttachmentCommandManager, which kinematically locks the virtual box to the robot's chassis for stable transport.

3. **Spatial Memory Query:** Before searching for the destination, the agent queries the Blackboard. If the location of the matching delivery zone (e.g., "Red Zone") was discovered and cached during a previous traversal, the search phase is skipped.
4. **FindDeliveryZone / NavigateToZone:** If the location is unknown, the robot triggers FindDeliveryZone to execute a new spiral search. If the location is known, it uses NavigateToZone to drive directly to the stored coordinates.
5. **VerifyZoneColor (Recovery):** Upon arriving at the target, the robot executes the VerifyZoneColor action. This serves as a critical reliability check. If the camera view is obstructed (e.g., by the carried box) or odometry drift has occurred, the robot performs a "Spin Recovery" maneuver. It rotates 90 degrees left and right to re-acquire the visual target before confirming the drop.
6. **DropBox:** Finally, the robot drives forward into the zone and triggers the detach command, releasing the object into the sorting area.

Figure 4.12: The Logistics Agent State Machine. The logic flows from searching for boxes to delivering them, utilizing the Blackboard to store and recall zone locations to optimize efficiency over time.

4.5.5 Agent 3: Line Follower

Scenario Context The Line Follower agent operates in an interactive playground where a human user can draw paths on the virtual ground in real-time. The robot must identify and track this line immediately. This scenario serves as a stress test for the system's end-to-end latency and validates the **Reactive Behavior (FR-22)** capability, as the path is not pre-calculated but defined dynamically by the user.

Perception Modes Unlike the other agents, this implementation utilizes a dedicated, high-performance node: `line_perception_node.py`. This node is optimized for low-latency visual servoing:

- **ROI Cropping:** The image is cropped to the **bottom 50% (configurable via parameters)**, focusing solely on the immediate path in front of the robot. This removes background noise and simplifies the processing required.

- **Line Fitting:** The processor applies a color threshold to isolate the high-contrast line drawn by the user. It then utilizes `cv2.fitLine` to fit a vector through the detected white pixels.
- **Error Calculation:** From this vector, the system calculates two control metrics: the *Lateral Error* (horizontal distance of the line from the image center) and the *Angular Error* (deviation of the line's heading from the robot's forward vector).

The debug view (Figure **??**) visualizes the binary mask and overlays the calculated heading vector, allowing the operator to see the error terms driving the control loop.

Figure 4.13: Debug view of the Line Follower. The system isolates the user-drawn path (binary mask) and fits a vector (blue line) to calculate lateral and angular errors for the PID controller.

State Machine Logic While the previous agents relied on the Nav2 stack [**NAV2Ref**] for motion planning, this agent implements a custom **Visual Servoing** control loop. The `mission_state_machine.py` orchestrates a three-stage sequence:

1. **FindLine:** The robot executes a spiral search pattern using the `find_line_action_server` to locate the start of the user-drawn path.
2. **AlignToLine:** Once the line is detected, the robot transitions to the `AlignToLine` action. The robot rotates in place (zero linear velocity) until the angular error returned by the perception node is minimized, ensuring the robot is parallel to the path before moving.
3. **FollowLine:** The robot engages a custom Proportional-Integral-Derivative (PID) controller implemented in `follow_line_action_server.py`. This controller calculates velocity commands (`cmd_vel`) directly from the perception error values. By bypassing the global path planning and costmap layers of [**NAV2Ref**], this direct control loop achieves the low latency required to react to sudden changes in the user-drawn line. If the line is erased or ends, the state machine transitions back to `FindLine` to search for a new path.

Figure 4.14: The Line Follower State Machine. The logic ensures the robot first locates and aligns with the path before entering the high-speed PID following loop.

5 Evaluation

This chapter presents the evaluation of the proposed framework. The core validation is performed through a set of complex application scenarios, which demonstrate the system's ability to provide a consistent and interactive Virtual Environment for Robot-in-the-Loop testing.

5.1 Methodology

5.1.1 Hardware Setup

5.1.2 Measurement Metrics

5.2 System Performance Analysis

5.2.1 Rendering Performance and Scalability

5.2.2 End-to-End Latency

5.3 Validation of the Virtual Environment

5.3.1 Smart Farming Scenario

5.3.2 Logistics Scenario

5.3.3 Adaptive Streets Scenario

5.3.4 Competitive Game Scenario (Pong)

5.4 Mixed Reality Validation

5.4.1 AR Projection Alignment Accuracy

5.4.2 VR Interface Usability

5.5 Discussion

6 Conclusion

summary of the work and discussion of future research directions.

List of Figures

Listings

Bibliography

[AA23] K. K. Alnowaiser and M. A. Ahmed. "Digital Twin: Current Research Trends and Future Directions". In: *Arabian Journal for Science and Engineering* 48.2 (2023), pp. 1075–1095.

[AAR+19] Bulgaria Aleksandrov, Chavdar Acad, Bulgaria Rumenin, et al. "Review of hardware-in-the-loop -a hundred years progress in the pseudo-real testing". In: 54 (Dec. 2019), pp. 70–84.

[AMV23] Hugo Araujo, Mohammad Reza Mousavi, and Mahsa Varshosaz. "Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review". In: *ACM Transactions on Software Engineering and Methodology* 32.2 (2023). ISSN: 1049-331X. DOI: 10.1145/3542945. URL: https://doi.org/10.1145/3542945.

[BM18] Mordechai Ben-Ari and Francesco Mondada. *Elements of Robotics*. Springer Open, 2018. ISBN: 978-3-319-62533-1. DOI: 10.1007/978-3-319-62533-1.

[BSH24] Rajmeet Bhourji, Lakmal Seneviratne, and Irfan Hussain. "A Deep Learning-Based Approach to Strawberry Grasping Using a Telescopic-Link Differential Drive Mobile Robot in ROS-Gazebo for Greenhouse Digital Twin Environments". In: *IEEE Access* PP (Jan. 2024), pp. 1–1. DOI: 10.1109/ACCESS.2024.3520233. URL: https://doi.org/10.1109/ACCESS.2024.3520233.

[BVM20] Anja Babić, Goran Vasiljevic, and Nikola Miskovic. "Vehicle-in-the-Loop Framework for Testing Long-Term Autonomy in a Heterogeneous Marine Robot Swarm". In: *IEEE Robotics and Automation Letters* PP (June 2020), pp. 1–1. DOI: 10.1109/LRA.2020.3000426.

[CCC+21] Giuseppina Lucia Casalaro, Giulio Cattivera, Federico Ciccozzi, et al. "Model-driven engineering for mobile robotic systems: a systematic mapping study". In: *Software and Systems Modeling* 21 (2021), pp. 19–49. DOI: 10.1007/s10270-021-00908-8.

[CCG+23] X. Cao, H. Chen, S. Y. Gelbal, et al. "Vehicle-in-Virtual-Environment (VVE) Method for Autonomous Driving System Development, Evaluation and Demonstration". In: *Sensors* 23.11 (2023), p. 5088. DOI: 10.3390/s23115088. URL: https://doi.org/10.3390/s23115088.

[CIR23] Enrique Coronado, Shunki Itadera, and Ixchel G. Ramirez-Alpizar. "Integrating Virtual, Mixed, and Augmented Reality to Human–Robot Interaction Applications Using Game Engines: A Brief Review of Accessible Software Tools and Frameworks". In: *Applied Sciences* 13.3 (2023). ISSN: 2076-3417. DOI: 10.3390/app13031292. URL: https://www.mdpi.com/2076-3417/13/3/1292.

[CKY20] Eunhee Chang, Hyun Taek Kim, and Byounghyun Yoo. "Virtual Reality Sickness: A Review of Causes and Measurements". In: *International Journal of Human–Computer Interaction* 36.17 (2020), pp. 1658–1682. DOI: 10.1080/10447318.2020.1778351.

[DBF+25] Juliana Dos Santos, Murilo Bicho, Tony Froes, et al. "Benchmarking Digital Twins for Tower Cranes: Isaac Sim vs. Gazebo". In: *IECON 2025 – 51st Annual Conference of the IEEE Industrial Electronics Society*. 2025, pp. 1–8. DOI: 10.1109/IECON58223.2025.11221823.

[DKK+21] Axel Diewald, Clemens Kurz, Prasanna Venkatesan Kannan, et al. "Radar Target Simulation for Vehicle-in-the-Loop Testing". In: *Vehicles* 3 (May 2021), pp. 257–271. DOI: 10.3390/vehicles3020016.

[DKQ+21] Xunhua Dai, Chenxu Ke, Quan Quan, and Kai-Yuan Cai. "RFlySim: Automatic test platform for UAV autopilot systems with FPGA-based hardware-in-the-loop simulations". In: *Aerospace Science and Technology* 114 (Apr. 2021), p. 106727. DOI: 10.1016/j.ast.2021.106727.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, et al. "CARLA: An Open Urban Driving Simulator". In: *1st Conference on Robot Learning (CoRL 2017)*. 2017. arXiv: 1711.03938.

[DSR+22] Maikol Drechsler, Varun Sharma, Fabio Reway, et al. "Dynamic Vehicle-in-the-Loop: A Novel Method for Testing Automated Driving Functions". In: *SAE International Journal of Connected and Automated Vehicles* 5 (June 2022), pp. 1–14. DOI: 10.4271/12-05-04-0029.

[EM21] Maria Engberg and Blair Macintyre. *Reality Media: Augmented and Virtual Reality*. Nov. 2021. ISBN: 9780262366250. DOI: 10.7551/mitpress/11708.001.0001.

[FCY25] Jose M. Flores Gonzalez, Enrique Coronado, and Natsuki Yamanobe. "ROS-Compatible Robotics Simulators for Industry 4.0 and Industry 5.0: A Systematic Review of Trends and Technologies". In: *Applied Sciences* 15.15 (2025). ISSN: 2076-3417. DOI: 10.3390/app15158637. URL: https://www.mdpi.com/2076-3417/15/15/8637.

[Foo13] Tully Foote. "tf: The transform library". In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. Apr. 2013, pp. 1–6. DOI: 10.1109/TePRA.2013.6556373.

[Geh24] P. Gehricke. "Virtual Environment for mobile Robotic Applications". MA thesis. Universität Osnabrück, 2024.

[Gri15] Michael Grieves. "Digital Twin: Manufacturing Excellence through Virtual Factory Replication". In: (Mar. 2015).

[GV17] Michael Grieves and John Vickers. "Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems". In: Aug. 2017, pp. 85–113. ISBN: 978-3-319-38754-3. DOI: 10.1007/978-3-319-38756-7_4.

[HLT+19] Márton Horváth, Qiong Lu, Tamas Tettamanti, et al. "Vehicle-In-The-Loop (VIL) and Scenario-In-The-Loop (SCIL) Automotive Simulation Concepts from the Perspectives of Traffic Simulation and Traffic Control". In: *Transport and Telecommunication Journal* 20 (Apr. 2019), pp. 153–161. DOI: 10.2478/ttj-2019-0014.

[Hu05] Xiaolin Hu. "Applying robot-in-the-loop-simulation to mobile robot systems". In: *2005 IEEE International Conference on Robotics and Automation*. 2005, pp. 506–513. ISBN: 0-7803-9178-0. DOI: https://doi.org/10.1109/ICAR.2005.1507456.

[KYH+24] Seyed Mohamad Kargar, Borislav Yordanov, Carlo Harvey, and Ali Asadipour. "Emerging Trends in Realistic Robotic Simulations: A Comprehensive Systematic Literature Review". In: *IEEE Access* 12 (2024), pp. 191264–191287. DOI: 10.1109/ACCESS.2024.3404881.

[KYS+25] Woojin Kwon, Jieun Yang, Seunghwa Song, et al. "Real-Time Digital-Twin-Based Cobot-Worker Collision Risk Prediction Using Unity, ROS, and UWB". In: *IEEE Access* 13 (2025), pp. 85967–85978. DOI: 10.1109/ACCESS.2025.3569332. URL: https://doi.org/10.1109/ACCESS.2025.3569332.

[LWS+21] Jiewu Leng, Dewen Wang, Weiming Shen, et al. “Digital twins-based smart manufacturing system design in Industry 4.0: A review”. In: *Journal of Manufacturing Systems* 60 (July 2021), pp. 119–137. DOI: 10.1016/j.jmsy.2021.05.011.

[MCT21] Carlos Magrin, Gustavo Conte, and Eduardo Todt. “Creating a Digital Twin as an Open Source Learning Tool for Mobile Robotics”. In: *2021 Latin American Robotics Symposium (LARS), Brazilian Symposium on Robotics (SBR) and Workshop on Robotics in Education (WRE)*. Oct. 2021, pp. 13–18. DOI: 10.1109/LARS/SBR/WRE54079.2021.9605457.

[MFG+22] Steven Macenski, Tully Foote, Brian Gerkey, et al. “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[Mic04] Olivier Michel. “Webots TM: Professional Mobile Robot Simulation”. In: *International Journal of Advanced Robotic Systems* 1.1 (2004), pp. 39–42. ISSN: 1729-8806.

[MK94] Paul Milgram and Fumio Kishino. “A Taxonomy of Mixed Reality Visual Displays”. In: *IEICE Transactions on Information Systems* E77-D.12 (Dec. 1994), pp. 1321–1329.

[MML+23] Steve Macenski, Tom Moore, David V. Lu, et al. “From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the Robot Operating System 2”. In: *Robotics and Autonomous Systems* 168 (2023), p. 104493. ISSN: 0921-8890. DOI: 10.1016/j.robot.2023.104493. URL: https://www.sciencedirect.com/science/article/pii/S092188902300132X.

[MMW+20] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. “The Marathon 2: A Navigation System”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 2718–2725. DOI: 10.1109/IROS45743.2020.9341207.

[MRS24] Nuno Marques, Marco Rodrigues, and Diogo Sousa. “Driving Forward Mobile Robotics: A Digital Twin Architecture Case Study for AGVs Data-Driven Autonomy”. In: (Jan. 2024). DOI: 10.21203/rs.3.rs-3837578/v1. URL: https://doi.org/10.21203/rs.3.rs-3837578/v1.

[MTH22] F. Mihalič, M. Truntič, and A. Hren. "Hardware-in-the-Loop Simulations: A Historical Overview of Engineering Challenges". In: *Electronics* 11.15 (2022), p. 2462. DOI: 10.3390/electronics11152462. URL: https://doi.org/10.3390/electronics11152462.

[MV20] Zhanat Makhataeva and Atakan Varol. "Augmented Reality for Robotics: A Review". In: *Robotics* 9.2 (Apr. 2020), p. 21. DOI: 10.3390/robotics9020021. URL: https://doi.org/10.3390/robotics9020021.

[PKP+20] Alexander Perzylo, Ingmar Kessler, Stefan Profanter, and Markus Rickert. "Toward a Knowledge-Based Data Backbone for Seamless Digital Engineering in Smart Factories". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2020, pp. 164–171. DOI: 10.1109/ETFA46521.2020.9211943.

[PRR+20] L. Pérez, S. Rodríguez-Jiménez, N. Rodríguez, et al. "Digital Twin and Virtual Reality Based Methodology for Multi-Robot Manufacturing Cell Commissioning". In: *Applied Sciences* 10.10 (May 2020), p. 3633. DOI: 10.3390/app10103633. URL: https://doi.org/10.3390/app10103633.

[Rob21] Robotec.AI. *ROS2 For Unity*. GitHub repository. 2021. URL: https://github.com/RobotecAI/ros2-for-unity.

[SKG+24] Maulshree Singh, Jayasekara Kapukotuwa, Eber Lawrence Souza Gouveia, et al. "Unity and ROS as a Digital and Communication Layer for Digital Twin Application: Case Study of Robotic Arm in a Smart Manufacturing Cell". In: *Sensors* 24.17 (2024). ISSN: 1424-8220. DOI: 10.3390/s24175680. URL: https://www.mdpi.com/1424-8220/24/17/5680.

[SKG+25] Maulshree Singh, Jayasekara Kapukotuwa, Eber Lawrence Souza Gouveia, et al. "Comparative Study of Digital Twin Developed in Unity and Gazebo". In: *Electronics* 14.2 (2025). ISSN: 2079-9292. DOI: 10.3390/electronics14020276. URL: https://www.mdpi.com/2079-9292/14/2/276.

[SPD+21] P. Stączek, J. Pizoń, W. Danilczuk, and A. Gola. "A Digital Twin Approach for the Improvement of an Autonomous Mobile Robots (AMR's) Operating Environment—A Case Study". In: *Sensors* 21.23 (2021), p. 7830. DOI: 10.3390/s21237830. URL: https://doi.org/10.3390/s21237830.

[SSW21] Richard Skarbez, Missie Smith, and Mary Whitton. “Revisiting Milgram and Kishino’s Reality-Virtuality Continuum”. In: *Frontiers in Virtual Reality* 2 (Mar. 2021). DOI: 10.3389/frvir.2021.647997. URL: https://doi.org/10.3389/frvir.2021.647997.

[Uni23] Unity Technologies. *Unity*. Version 2023.2.3. Game development platform. 2023. URL: https://unity.com/.

[VTS21] Balázs Varga, Tamas Tettamanti, and Zsolt Szalay. “System Architecture for Scenario-In-The-Loop Automotive Testing”. In: *Transport and Telecommunication Journal* 22 (Apr. 2021), pp. 141–151. DOI: 10.2478/ttj-2021-0011.

[WPC+22] Michael Walker, Thao Phung, Tathagata Chakraborti, et al. *Virtual, Augmented, and Mixed Reality for Human-Robot Interaction: A Survey and Virtual Design Element Taxonomy*. 2022. arXiv: 2202.11249 [cs.RO]. URL: https://arxiv.org/abs/2202.11249.

[YMZ20] Yuanlin Yang, Wei Meng, and Shiquan Zhu. “A Digital Twin Simulation Platform for Multi-rotor UAV”. In: *2020 7th International Conference on Information, Cybernetics, and Computational Social Systems (ICCSS)*. 2020, pp. 591–596. DOI: 10.1109/ICCSS52145.2020.9336872. URL: https://doi.org/10.1109/ICCSS52145.2020.9336872.

Declaration of Authorship

I hereby declare that I have written this thesis independently and without unauthorized assistance. I have not used any sources or aids other than those indicated. All passages taken verbatim or in spirit from the works of other authors have been properly acknowledged and cited.

Osnabrück, 23 December 2025

Jonas Kittelmann